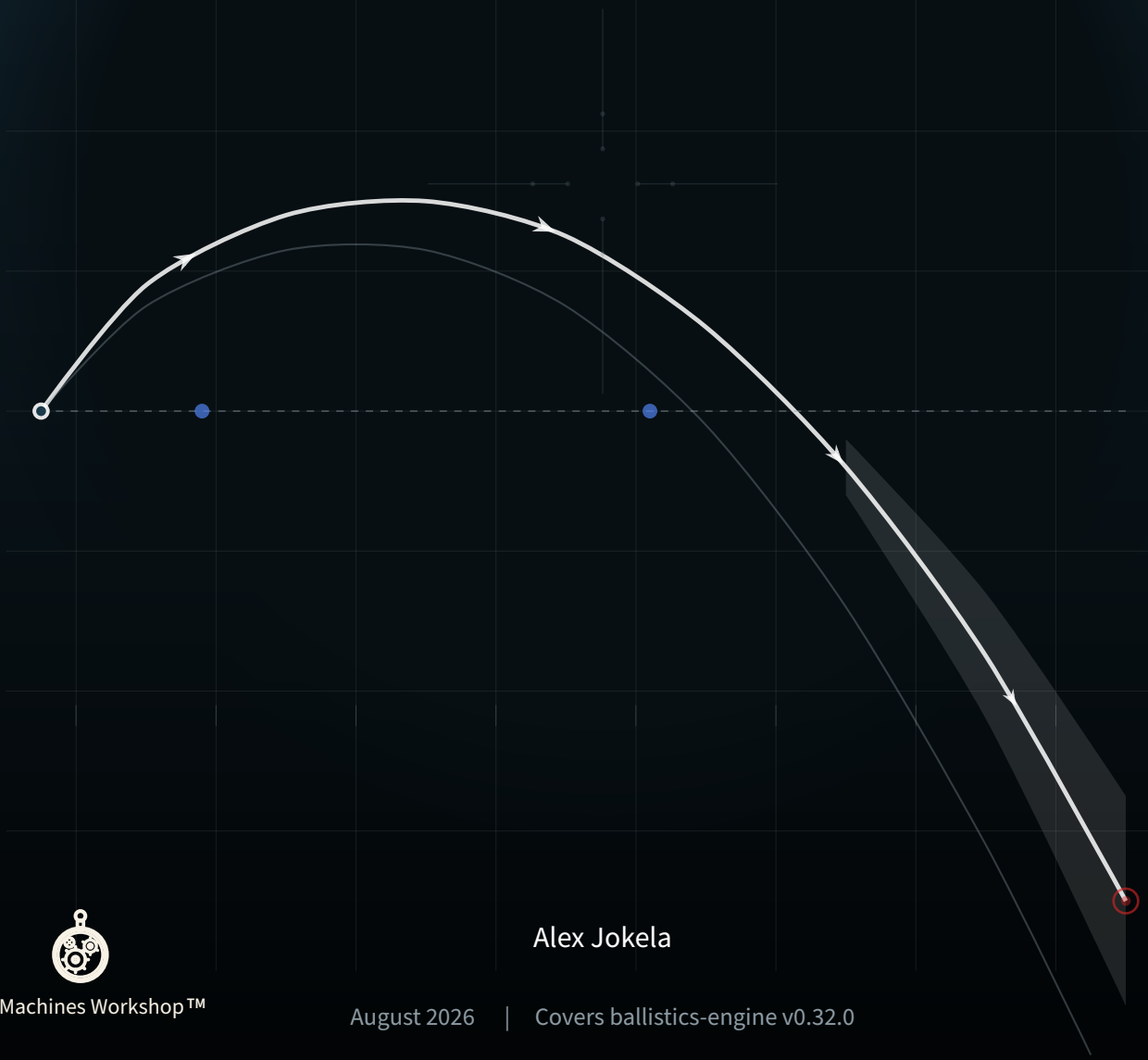
# The Ballistics Engine Handbook

A Comprehensive Guide to Computational Exterior Ballistics



Alex Jokela

Tiny Machines Workshop™

August 2026 | Covers ballistics-engine v0.32.0

**The Ballistics Engine Handbook**
A Comprehensive Guide to Computational Exterior Ballistics

Copyright © 2026 Alex Jokela.
All rights reserved.

Covers ballistics-engine v0.32.0.

*First edition, revised August 2026.*

ISBN 979-8-9951255-2-5

The ballistics-engine source code is available at:
https://github.com/ajokela/ballistics-engine

Published at https://ballistics.rs/

Published by Tiny Machines Workshop, LLC
https://tinymachines.io/

# Contents


**I Getting Started** 1

**1 Introduction** 3

1.1 What Is `ballistics-engine`? . . . 3

1.1.1 The CLI and the Library . . . 4

1.1.2 A Tour of the Subcommands . . . 5

1.2 Why Computational Ballistics Matters . . . 6

1.2.1 Bridging Theory and Practice . . . 6

1.2.2 Workflows That Depend on a Solver . . . 7

1.2.3 Why Another Ballistics Solver? . . . 8

1.2.4 When Not to Use a Solver . . . 8

1.3 Who This Book Is For . . . 9

1.3.1 How to Read This Book . . . 10

1.4 A Brief History of Exterior Ballistics Computation . . . 10

1.4.1 From Tartaglia to Newton . . . 10

1.4.2 The Age of Firing Tables . . . 10

1.4.3 Ballistic Computers and the BRL . . . 11

1.4.4 The Personal-Computer Revolution . . . 11

1.4.5 Where `ballistics-engine` Fits . . . 12

1.5 How This Book Is Organized . . . 12

Conventions Used in This Book . . . 13

Exercises . . . 14

What's Next . . . 15

**2 Installation & Your First Shot** 17

2.1 Installing Pre-Built Binaries . . . 17

2.1.1 macOS . . . 18

2.1.2 Linux . . . 18

2.1.3 Windows . . . 19
2.1.4 FreeBSD, OpenBSD, and NetBSD . . . 20
2.2 Building from Source with Cargo . . . 20
2.2.1 Cloning and Building . . . 20
2.2.2 Feature Flags . . . 21
2.2.3 Running the Test Suite . . . 22
2.2.4 Building for WebAssembly . . . 22
2.3 Verifying Your Installation . . . 23
2.4 Your First Trajectory: .308 Win at 1,000 Yards . . . 24
2.4.1 The Load . . . 25
2.4.2 Running the Command . . . 25
2.4.3 Reading the Results . . . 26
2.4.4 Adding the Full Trajectory Table . . . 27
2.5 Understanding the Output . . . 29
2.5.1 Drop and Drift . . . 29
2.5.2 Angular Corrections: MOA and MIL . . . 29
2.5.3 Adding Environmental Conditions . . . 30
2.5.4 Sight Height and Bore Height . . . 31
2.6 Output Formats: Table, JSON, CSV . . . 32
2.6.1 Table (Default) . . . 32
2.6.2 JSON . . . 32
2.6.3 CSV . . . 33
2.6.4 PDF Dope Cards . . . 34
2.7 Metric Units . . . 34
2.8 Working with the G7 Drag Model . . . 35
2.9 The Zeroing Workflow . . . 36
2.9.1 Come-Up and Range Tables . . . 37
Exercises . . . 37
What's Next . . . 39

**3 An Exterior Ballistics Primer** **41**
3.1 What Happens After the Muzzle . . . 41
3.2 Gravity: The Constant Pull . . . 42
3.2.1 Gravitational Drop . . . 43
3.2.2 The Zero Angle . . . 44
3.3 Drag: The Atmosphere Fights Back . . . 44
3.3.1 The Drag Equation . . . 44
3.3.2 Drag as a Function of Mach Number . . . 45
3.3.3 Standard Drag Functions . . . 46
3.4 Ballistic Coefficient: The Bullet's Aerodynamic Fingerprint . . . 48

3.4.1 Definition . . . 48
3.4.2 G1 BC vs. G7 BC . . . 49
3.4.3 How BC Affects the Trajectory . . . 50
3.4.4 Why BC Varies with Velocity . . . 51
3.5 The Standard Atmosphere (ICAO) . . . 52
3.5.1 Sea-Level Reference Conditions . . . 52
3.5.2 Atmospheric Layers . . . 52
3.5.3 Pressure and Density at Altitude . . . 53
3.5.4 Overriding Standard Conditions . . . 54
3.5.5 Humidity Effects . . . 54
3.6 Coordinate System Conventions . . . 55
3.6.1 Wind Direction Convention . . . 56
3.6.2 Relating Coordinates to Shooter Language . . . 57
3.7 Units: MOA, MIL, Inches, and Centimeters . . . 57
3.7.1 Minutes of Angle (MOA) . . . 57
3.7.2 Milliradians (MIL) . . . 57
3.7.3 Converting Between Linear and Angular Measures . . . 58
3.7.4 Linear Distance Units . . . 59
3.8 The Retardation Formula . . . 60
3.9 Putting It All Together: The Point-Mass Model . . . 60
Exercises . . . 63
What's Next . . . 66

**II The Commands** **67**

**4 Trajectory Computation** **69**
4.1 The `trajectory` Command . . . 69
4.2 Required Inputs . . . 70
4.2.1 Muzzle Velocity (`--velocity`) . . . 70
4.2.2 Ballistic Coefficient (`--bc`) . . . 71
4.2.3 Bullet Mass (`--mass`) . . . 72
4.2.4 Bullet Diameter (`--diameter`) . . . 72
4.3 Choosing a Drag Model: G1 vs. G7 vs. Custom . . . 73
4.3.1 G1: The Traditional Standard . . . 73
4.3.2 G7: The Modern Standard for Long-Range . . . 73
4.3.3 G6 and G8 . . . 74
4.3.4 Which Model Should You Use? . . . 74
4.4 Setting the Zero Range . . . 74
4.5 Output Columns Explained . . . 75

4.5.1 Range . . . 76
4.5.2 Velocity . . . 76
4.5.3 Energy . . . 76
4.5.4 Drop . . . 76
4.5.5 Windage . . . 77
4.5.6 Time of Flight . . . 77
4.5.7 Mach Number . . . 77
4.5.8 Output Formats . . . 78
4.5.9 Terminal Charts (`--plot`) . . . 80
4.5.10 Turret Adjustment Units . . . 80
4.5.11 The Mover Ring (`--target-speed`) . . . 82
4.6 Step Size and Resolution . . . 83
4.6.1 Integration Methods . . . 83
4.6.2 The `--time-step` Flag . . . 84
4.6.3 Trajectory Sampling . . . 84
4.7 Adding Wind: Speed and Angle . . . 85
4.7.1 Wind Speed . . . 85
4.7.2 Wind Direction . . . 85
4.7.3 Wind Shear . . . 86
4.8 Sight Height and Scope Offset . . . 86
4.8.1 Sight Height (`--sight-height`) . . . 87
4.8.2 Bore Height (`--bore-height`) . . . 87
4.9 Inclination: Uphill and Downhill Shooting . . . 88
4.9.1 Equivalent Horizontal Range . . . 88
4.9.2 Which Plane Is “Drop” Measured In? . . . 89
4.9.3 Sight Geometry: Offset Mounts and Deliberate Zeros . . . 90
4.9.4 The Rifleman’s Rule . . . 91
4.10 Atmospheric Conditions . . . 91
4.10.1 Temperature (`--temperature`) . . . 92
4.10.2 Pressure (`--pressure`) . . . 92
4.10.3 Humidity (`--humidity`) . . . 92
4.10.4 Altitude (`--altitude`) . . . 92
4.11 Practical Examples: Building a Dope Card . . . 93
4.11.1 Example 1: .308 Win Match Rifle at 5000 ft Elevation . . . 93
4.11.2 Example 2: 6.5 Creedmoor for PRS Competition . . . 93
4.11.3 Example 3: .338 Lapua ELR with PDF Output . . . 94
4.11.4 Example 4: Hunting Load with Uphill Angle . . . 94
4.11.5 Example 5: CSV Export for Spreadsheet Analysis . . . 95
4.12 Advanced Physics Flags . . . 95
4.12.1 Spin Drift (`--enable-spin-drift`) . . . 95

4.12.2 Coriolis Effect (`--enable-coriolis`) . . . 96
4.12.3 Magnus Effect (`--enable-magnus`) . . . 96
4.13 Exercises . . . 96

**5 Zeroing** **99**
5.1 What Zeroing Means and Why It Matters . . . 99
5.2 The `zero` Command . . . 100
5.2.1 Required Flags . . . 101
5.2.2 Optional Flags . . . 101
5.2.3 Working Backwards: `--from-angle` . . . 102
5.3 How the Zeroing Algorithm Works . . . 103
5.3.1 Brent's Method . . . 103
5.3.2 Under the Hood: The Height Error Function . . . 104
5.4 Near-Zero and Far-Zero Distances . . . 104
5.4.1 Why Near-Zero Matters . . . 105
5.5 Point-Blank Range and Maximum Ordinate . . . 105
5.5.1 Maximum Ordinate . . . 105
5.5.2 Maximum Point-Blank Range . . . 106
5.5.3 MPBR Output . . . 106
5.5.4 The MPBR Algorithm . . . 107
5.6 Zeroing at Altitude and in Non-Standard Conditions . . . 107
5.6.1 How Conditions Affect Zero . . . 108
5.6.2 Computing Zero Shift . . . 108
5.6.3 Using `auto-zero` with Field Conditions . . . 109
5.6.4 Zeroing on One Day, Shooting on Another . . . 110
5.6.5 Powder Temperature Sensitivity . . . 110
5.6.6 Measured Powder Curves . . . 111
5.7 Exercises . . . 112

**6 Monte Carlo Simulation** **115**
6.1 Why Monte Carlo Simulation? . . . 115
6.2 The `monte-carlo` Command . . . 116
6.2.1 Required Flags . . . 116
6.2.2 Monte Carlo–Specific Flags . . . 116
6.2.3 Output Modes . . . 117
6.3 Input Distributions: What Varies and by How Much . . . 118
6.4 Muzzle Velocity Variation (SD and ES) . . . 118
6.4.1 How Velocity Variation Affects Impact . . . 119
6.5 BC Uncertainty . . . 119
6.6 Wind Variability . . . 120

6.6.1 Practical Wind SD Values . . . 121
6.7 Convergence: How Many Iterations Are Enough? . . . 121
6.7.1 Practical Guidance . . . 121
6.8 Reading the Results: CEP, Confidence Ellipses, and Extreme Spread . . . 122
6.8.1 CEP (Circular Error Probable) . . . 122
6.8.2 Confidence Ellipses . . . 123
6.8.3 Extreme Spread . . . 124
6.8.4 Hit Probability . . . 124
6.9 WEZ: Hit Probability Across a Whole Range Band . . . 125
6.10 Practical Example: Predicting First-Round Hit Probability . . . 127
6.10.1 Setup . . . 127
6.10.2 Interpreting the Results . . . 127
6.10.3 Sensitivity Analysis . . . 128
6.10.4 What-If: Factory Ammo vs. Handload . . . 128
6.11 Advanced Considerations . . . 129
6.11.1 Correlated Inputs . . . 129
6.11.2 Non-Gaussian Distributions . . . 129
6.11.3 Computational Performance . . . 130
6.12 Exercises . . . 130

**7 BC Estimation & Truing 133**
7.1 Why True Your BC? . . . 133
7.2 The `estimate-bc` Command . . . 134
7.2.1 Fitting G1 and G7 Together . . . 135
7.2.2 More Than Two Points . . . 135
7.2.3 Fitting Velocity Instead of Drop . . . 136
7.2.4 Zeroed Cards and the Conditions They Were Shot In . . . 137
7.3 Estimating BC from Two-Velocity Measurements . . . 138
7.4 Truing BC to Observed Drop at Range . . . 138
7.4.1 How It Works . . . 139
7.4.2 Where the Chronograph Actually Reads . . . 139
7.4.3 Optional: Chronograph Comparison . . . 140
7.4.4 Confidence Levels . . . 140
7.4.5 Fitting Velocity and BC Together . . . 141
7.4.6 Stage Two: DSF Truing Below Mach 1.2 . . . 142
7.4.7 Designing the Experiment First . . . 144
7.4.8 Truing With Known Uncertainties . . . 145
7.5 BC Confidence Intervals . . . 146
7.5.1 Procedure . . . 146
7.5.2 BC Stability Across Velocity Ranges . . . 147

7.6 When to Use G1 vs. G7 for Truing . . . 148
7.6.1 The G1 Problem . . . 148
7.6.2 The G7 Advantage . . . 148
7.6.3 Generating BC Segments . . . 148
7.7 A Complete Truing Workflow . . . 149
7.8 Exercises . . . 150

**8 Profiles & Workflow 153**
8.1 Why Use Profiles? . . . 153
8.2 Creating and Saving a Profile . . . 154
8.2.1 Required Parameters . . . 154
8.2.2 Optional Parameters . . . 155
8.3 The Profile JSON Format . . . 156
8.3.1 Field Reference . . . 157
8.3.2 Profile Storage Location . . . 157
8.4 Loading and Using Profiles . . . 157
8.4.1 Profile Compatibility with Other Commands . . . 159
8.5 Managing Multiple Rifles and Loads . . . 159
8.5.1 Listing Profiles . . . 159
8.5.2 Showing Profile Details . . . 160
8.5.3 Deleting Profiles . . . 160
8.5.4 More Than One Zero . . . 160
8.5.5 Importing an ArcherBC2 Profile . . . 162
8.5.6 Naming Conventions . . . 163
8.6 Workflow: From Load Development to Range Day . . . 163
8.6.1 Step 1: Load Development . . . 163
8.6.2 Step 2: Generate an Initial Dope Card . . . 164
8.6.3 Step 3: Validate at Range . . . 164
8.6.4 Step 4: True the Solution . . . 164
8.6.5 Step 5: Update the Profile . . . 165
8.6.6 Step 6: Generate the Final Dope Card . . . 166
8.6.7 Step 7: Verify with Monte Carlo . . . 166
8.6.8 Step 8: Post-Match Review . . . 167
8.7 CSV-Based Profile Loading . . . 167
8.8 Exercises . . . 167

**III Atmosphere & Environment 171**

**9 The Atmosphere 173**

9.1 The ICAO Standard Atmosphere . . . 173
9.1.1 Atmospheric Layers . . . 175
9.1.2 The Barometric Formula . . . 176
9.2 Temperature, Pressure, and Humidity . . . 177
9.2.1 Temperature . . . 177
9.2.2 Barometric Pressure . . . 179
9.2.3 Humidity . . . 180
9.2.4 Saturation Vapor Pressure . . . 182
9.3 Air Density: The Variable That Matters Most . . . 182
9.3.1 The Density Ratio in Practice . . . 183
9.4 Density Altitude vs. True Altitude . . . 183
9.4.1 Entering Conditions As a Density Altitude . . . 184
9.5 How BALLISTICS-ENGINE Computes Atmospheric Density . . . 186
9.5.1 The Computation Pipeline . . . 186
9.5.2 The CIPM Density Formula . . . 186
9.5.3 Along-Track Density: The `get_local_atmosphere()` Function . . . 188
9.6 Non-Standard Conditions: Hot Days, Cold Days, Altitude . . . 189
9.6.1 Hot Days . . . 189
9.6.2 Cold Days . . . 190
9.6.3 Altitude . . . 190
9.7 Practical Impact: Sea Level vs. 5000 ft vs. 10 000 ft . . . 191
9.7.1 The .308 Win at Three Altitudes . . . 191
9.7.2 The 6.5 Creedmoor at Three Altitudes . . . 193
9.7.3 Visualizing the Difference . . . 193
9.7.4 How Much Does Each Variable Contribute? . . . 194
Exercises . . . 195
What's Next . . . 196

**10 Wind Modeling** **197**
10.1 How Wind Affects a Bullet . . . 197
10.2 The Wind Clock: 0 ° Headwind Through 360 ° . . . 198
10.3 Crosswind Deflection: The Lag Angle Explanation . . . 201
10.3.1 The Math Behind BALLISTICS-ENGINE's Wind Implementation . . . 202
10.4 Headwind and Tailwind Effects on Drop . . . 203
10.4.1 Quartering Winds . . . 204
10.5 Wind Shear: Vertical Wind Gradients . . . 205
10.5.1 Wind Shear Models . . . 205
10.5.2 Enabling Wind Shear . . . 206
10.5.3 How Wind Shear Enters the Trajectory Solver . . . 207
10.5.4 When Wind Shear Matters . . . 208

10.6 Modeling Variable Wind with BALLISTICS-ENGINE . . . 209
10.6.1 The WindSock Architecture . . . 209
10.6.2 Checked Construction for Library Callers . . . 210
10.6.3 Multi-Segment Wind via the CLI . . . 211
10.6.4 Multi-Segment Wind: A Practical Example . . . 211
10.7 Entering Wind the Way You Think About It . . . 213
10.7.1 Compass Bearings . . . 213
10.8 Truing the Wind Call . . . 214
10.9 Practical Wind-Reading Strategies . . . 216
10.9.1 Using a Wind Meter . . . 216
10.9.2 Reading Mirage . . . 216
10.9.3 Using Flags and Indicators . . . 217
10.9.4 The Bracket Technique . . . 217
10.9.5 Monte Carlo for Wind Uncertainty . . . 217
10.9.6 Developing Your Wind Estimation Workflow . . . 218
Exercises . . . 219
What's Next . . . 220

**IV Drag & BC Modeling** **221**

**11 Drag Models & Tables** **223**
11.1 What Is a Drag Model? . . . 223
11.1.1 Aerodynamic Drag in One Equation . . . 223
11.1.2 The Standard Projectile Idea . . . 224
11.1.3 A Family of Standards . . . 224
11.2 The G1 Standard Projectile . . . 225
11.2.1 History and Shape . . . 225
11.2.2 The G1 Drag Curve . . . 225
11.2.3 When G1 Works Well . . . 226
11.2.4 When G1 Fails . . . 226
11.3 The G7 Standard Projectile . . . 227
11.3.1 A Modern Shape for Modern Bullets . . . 227
11.3.2 The G7 Drag Curve . . . 227
11.3.3 Why G7 BCs Are Always Lower Than G1 BCs . . . 228
11.3.4 When to Use G7 . . . 228
11.4 Other Standard Models: G2, G5, G6, G8, GS, GI, RA4 . . . 229
11.4.1 G2: The Aberdeen J Projectile . . . 230
11.4.2 G5: Short Boat-Tail . . . 230
11.4.3 G6: Flat-Base, 6-Caliber Secant Ogive . . . 230

11.4.4 G8: Flat-Base, 10-Caliber Secant Ogive . . . 231
11.4.5 GI: Ingalls Tables . . . 231
11.4.6 GS: Spherical (Round Ball) . . . 231
11.4.7 Choosing the Right Model . . . 231
11.5 Custom Drag Tables . . . 232
11.5.1 Beyond the Standard Models . . . 232
11.5.2 Loading Custom Tables in ballistics-engine . . . 233
11.5.3 Vendor `.drg` Files . . . 234
11.5.4 Scaling a Whole Curve: `--cd-scale` . . . 234
11.5.5 The DragTable Data Structure . . . 235
11.6 Drag Table Interpolation . . . 236
11.6.1 Catmull–Rom Cubic Interpolation . . . 236
11.6.2 Why Catmull–Rom? . . . 237
11.6.3 Edge Handling and Extrapolation . . . 237
11.7 Form Factors: Connecting BC to Drag Coefficients . . . 238
11.7.1 The Form Factor Defined . . . 238
11.7.2 Form Factors in ballistics-engine . . . 238
11.7.3 Understanding the Values . . . 239
11.8 How ballistics-engine Evaluates Drag . . . 240
11.8.1 The Drag Evaluation Pipeline . . . 240
11.8.2 The Dispatch by Model . . . 241
11.8.3 Lazy Loading and Fallback . . . 241
11.8.4 Performance Considerations . . . 242
11.9 BC Interpolation by Mach Number . . . 242
11.10 Exercises . . . 243

**12 Advanced BC Modeling** **245**
12.1 Why BC Is Not Constant . . . 245
12.1.1 The Form Factor Problem Revisited . . . 245
12.1.2 Where the Error Shows Up . . . 246
12.2 Velocity-Dependent BC: The Cluster Degradation Model . . . 246
12.2.1 The Concept . . . 246
12.2.2 The Four Clusters . . . 247
12.2.3 Cluster Assignment . . . 247
12.2.4 Velocity-Dependent Multipliers . . . 248
12.2.5 Cluster Comparison . . . 249
12.3 BC Segments: Step-Function BC by Velocity Band . . . 250
12.3.1 A Simpler Approach . . . 250
12.3.2 Automatic Segment Estimation . . . 250
12.3.3 Writing the Ladder Yourself . . . 252

12.3.4 Sectional Density Adjustment . . . 253
12.4 Physics-Bounded BC: Preventing Impossible Values . . . 254
12.4.1 Why Bounds Matter . . . 254
12.4.2 The Fallback BC System . . . 254
12.4.3 Minimum Division Thresholds . . . 254
12.5 The Transonic Challenge: Mach 1.0–1.2 . . . 255
12.5.1 What Happens Near Mach 1 . . . 255
12.5.2 Why Standard Drag Tables Struggle Here . . . 255
12.6 Transonic Drag Corrections in ballistics-engine . . . 256
12.6.1 The Correction Pipeline . . . 256
12.6.2 Projectile Shape Classification . . . 256
12.6.3 The Transonic Drag Rise Model . . . 256
12.6.4 Post-Peak Drag Decay . . . 257
12.6.5 Wave Drag Contribution . . . 257
12.6.6 The Complete Transonic Correction . . . 258
12.7 Comparing Constant BC vs. Segmented BC vs. Cluster BC . . . 259
12.7.1 Setting Up the Comparison . . . 259
12.7.2 What to Expect . . . 260
12.7.3 Which Model to Use? . . . 260
12.8 Exercises . . . 260

**13 BC in Practice** **263**
13.1 Published BC Values: What They Mean and Where They Come From . . . 263
13.1.1 Manufacturer-Stated BC . . . 263
13.1.2 Independent Sources . . . 264
13.1.3 Which Atmosphere Was the BC Referenced To? . . . 264
13.2 Doppler-Derived vs. Two-Point Chronograph Measurements . . . 265
13.2.1 Doppler Radar: The Gold Standard . . . 265
13.2.2 Two-Point Chronograph Measurements . . . 266
13.2.3 How the Methods Compare . . . 267
13.3 Truing Your BC with Real-World Data . . . 267
13.3.1 What Is Truing? . . . 267
13.3.2 The Truing Workflow . . . 267
13.3.3 The BC Estimation Command . . . 269
13.4 The BC5D Offline Correction Tables . . . 269
13.4.1 What Are BC5D Tables? . . . 269
13.4.2 How BC5D Tables Are Generated . . . 270
13.4.3 The BC5D Binary Format . . . 270
13.4.4 4D Linear Interpolation . . . 271
13.4.5 Using BC5D Tables from the CLI . . . 271

13.4.6 BC5D vs. Other Correction Methods . . . 272
13.5 Common Mistakes and How to Avoid Them . . . 273
13.5.1 Mistake 1: Mixing G1 and G7 BCs . . . 273
13.5.2 Mistake 2: Trusting a Single Published BC . . . 273
13.5.3 Mistake 3: Truing BC Without Confirming Muzzle Velocity . . . 273
13.5.4 Mistake 4: Using Stale Atmospheric Data . . . 274
13.5.5 Mistake 5: Ignoring the Transonic Zone . . . 274
13.5.6 Mistake 6: Over-Trueing to One Distance . . . 274
13.6 Exercises . . . 275

**V Advanced Physics 279**

**14 Spin Effects 281**
14.1 Why Bullets Spin: Rifling and Gyroscopic Stability . . . 281
14.1.1 The Problem of Aerodynamic Instability . . . 281
14.1.2 Rifling: Imparting Spin . . . 282
14.1.3 Computing Spin Rate from Twist and Velocity . . . 282
14.2 Spin Drift: The Horizontal Deflection from Spin . . . 283
14.2.1 What Is Spin Drift? . . . 283
14.2.2 The Physics: Yaw of Repose . . . 283
14.2.3 The Litz Empirical Formula . . . 284
14.3 Computing Spin Drift in `ballistics-engine` . . . 284
14.3.1 Enabling Spin Drift . . . 284
14.3.2 The Standard vs. Advanced Model . . . 285
14.3.3 Bullet-Type Coefficients . . . 286
14.3.4 A Worked Example: .308 Win at 1 000 Yards . . . 286
14.4 Spin Decay over Distance . . . 287
14.4.1 Why Spin Decays . . . 287
14.4.2 The Exponential Decay Model . . . 287
14.4.3 Bullet Type and Surface Quality . . . 288
14.4.4 Moment of Inertia . . . 288
14.4.5 Observing Spin Decay in Practice . . . 289
14.5 The Magnus Effect and Its Contribution . . . 289
14.5.1 What Is the Magnus Effect? . . . 289
14.5.2 Magnus Coefficient vs. Mach Number . . . 290
14.5.3 Magnus Drift Magnitude . . . 290
14.5.4 The Enhanced Spin Drift: Combining Gyroscopic and Magnus Components 290
14.6 Twist Rate and Its Effect on Stability . . . 291
14.6.1 The Twist–Stability Relationship . . . 291

14.6.2 Using the `stability` Subcommand . . . 291
14.6.3 Over-Stabilisation and Its Effects . . . 292
14.7 Practical Implications: When Does Spin Drift Matter? . . . 292
14.7.1 The Range Threshold . . . 292
14.7.2 Spin Drift vs. Wind Drift . . . 293
14.7.3 Comparing With and Without Spin Drift . . . 293
14.7.4 High Altitude and Spin Drift . . . 294

**15 Coriolis & Eötvös Effects 297**
15.1 Why the Earth's Rotation Matters . . . 297
15.2 The Coriolis Effect: Horizontal Deflection . . . 298
15.2.1 The Basic Physics . . . 298
15.2.2 Projecting into the Shooter's Local Frame . . . 298
15.2.3 Direction of Deflection . . . 299
15.2.4 Magnitude Estimate . . . 299
15.3 The Eötvös Effect: Vertical Component . . . 300
15.3.1 What Is the Eötvös Effect? . . . 300
15.3.2 The Physics . . . 300
15.3.3 How Large Is It? . . . 301
15.3.4 How BALLISTICS-ENGINE Handles the Eötvös Effect . . . 301
15.4 Latitude and Heading Dependence . . . 302
15.4.1 Latitude Dependence . . . 302
15.4.2 Heading Dependence . . . 302
15.5 Computing Coriolis Corrections in `ballistics-engine` . . . 303
15.5.1 Enabling the Coriolis Effect . . . 303
15.5.2 How the Engine Computes It . . . 304
15.5.3 Combining Coriolis with Other Effects . . . 304
15.5.4 Southern Hemisphere and Negative Latitudes . . . 304
15.5.5 Viewing the Effect in Isolation . . . 305
15.6 Practical Significance: When to Include Coriolis . . . 305
15.6.1 The Range Threshold . . . 305
15.6.2 Coriolis vs. Wind Uncertainty . . . 306
15.6.3 When Heading Changes Matter . . . 306
15.6.4 Combining Coriolis and Spin Drift . . . 307
15.6.5 ELR and One-Mile Shooting . . . 308

**16 Precession & Nutation 311**
16.1 Gyroscopic Precession: The Slow Wobble . . . 311
16.1.1 Why Bullets Precess . . . 311
16.1.2 Precession Frequency . . . 312

16.2 Nutation: The Fast Wobble . . . 313
16.2.1 What Is Nutation? . . . 313
16.2.2 Nutation Frequency . . . 313
16.2.3 Nutation Damping . . . 314
16.3 Epicyclic Motion: The Combined Pattern . . . 315
16.3.1 Combining Precession and Nutation . . . 315
16.3.2 Visualising the Epicycle . . . 317
16.4 Aerodynamic Jump . . . 317
16.4.1 What Is Aerodynamic Jump? . . . 317
16.4.2 Causes . . . 317
16.4.3 Computing Aerodynamic Jump . . . 318
16.4.4 Jump Direction and Twist . . . 318
16.4.5 Crosswind Jump Sensitivity . . . 319
16.5 Pitch Damping: How Oscillations Decay . . . 319
16.5.1 The Role of Pitch Damping . . . 319
16.5.2 Pitch Damping Coefficient . . . 320
16.5.3 The Pitch Damping Moment . . . 320
16.5.4 How Pitch Damping Affects Spin Drift . . . 321
16.6 How `ballistics-engine` Models Precession and Nutation . . . 322
16.6.1 Enabling the Angular Motion Model . . . 322
16.6.2 The Angular State . . . 322
16.6.3 The Precession-Nutation Parameters . . . 323
16.7 When 4DOF Effects Change the Answer . . . 324
16.7.1 When They Matter . . . 324
16.7.2 Quantifying the Difference . . . 324
16.7.3 Recommendations . . . 325

**17 Stability & the Transonic Transition 329**
17.1 Gyroscopic Stability Factor ($S_g$) . . . 329
17.1.1 Definition . . . 329
17.1.2 How $S_g$ Changes During Flight . . . 330
17.2 Dynamic Stability Factor ($S_d$) . . . 331
17.2.1 Beyond Gyroscopic Stability . . . 331
17.2.2 The Dynamic Stability Criterion . . . 331
17.3 The Miller Stability Formula . . . 332
17.3.1 The Original Miller Formula . . . 332
17.3.2 Velocity and Atmospheric Corrections . . . 333
17.3.3 Implementation in `ballistics-engine` . . . 333
17.3.4 Advanced Stability: Bullet-Type Corrections . . . 334
17.3.5 Using the `stability` Subcommand . . . 335

17.3.6 Atmospheric Effects on Stability . . . 336
17.4 The Transonic Zone: What Happens at Mach 1.0–1.2 . . . 336
17.4.1 What Is the Transonic Zone? . . . 336
17.4.2 The Drag Rise . . . 337
17.4.3 Wave Drag . . . 338
17.4.4 Peak Drag at Mach 1.0–1.05 . . . 338
17.5 Why Bullets Destabilise in the Transonic . . . 338
17.5.1 The Pitch Damping Sign Reversal . . . 338
17.5.2 What the Shooter Sees . . . 339
17.5.3 The Reynolds Number Connection . . . 340
17.6 Modeling Transonic Behavior in `ballistics-engine` . . . 340
17.6.1 The Transonic Correction Pipeline . . . 340
17.6.2 Shape Estimation from Load Data . . . 341
17.6.3 Trajectory Stability Checking . . . 342
17.7 Practical Guidance: Staying Supersonic . . . 342
17.7.1 Choosing Bullets That Survive the Transonic . . . 342
17.7.2 Cartridge Selection for Beyond-Transonic Shooting . . . 343
17.7.3 Altitude and the Transonic Zone . . . 343
17.7.4 Full Advanced Physics: The Kitchen Sink . . . 344

**VI Numerical Methods** **347**

**18 The Trajectory Solver** **349**
18.1 The Equations of Motion . . . 349
18.1.1 Newton's Second Law in Vector Form . . . 349
18.1.2 The Drag Acceleration . . . 350
18.1.3 Gravity . . . 351
18.1.4 Coriolis Acceleration . . . 351
18.1.5 Magnus Acceleration . . . 351
18.2 State Vectors: Position, Velocity, and Spin . . . 352
18.2.1 Initial State Construction . . . 353
18.2.2 Time Span Estimation . . . 353
18.2.3 The `TrajectoryParams` Structure . . . 354
18.3 Computing Derivatives: Drag, Gravity, Coriolis, Spin . . . 354
18.3.1 Step 1: Wind-Adjusted Velocity . . . 355
18.3.2 Step 2: Atmospheric Conditions . . . 355
18.3.3 Step 3: Drag Coefficient Lookup . . . 355
18.3.4 Step 4: BC Interpolation . . . 356
18.3.5 Step 5: Assembling the Acceleration Vector . . . 356

18.4 The 4th-Order Runge–Kutta Method (RK4) . . . 357
18.4.1 The RK4 Algorithm . . . 357
18.4.2 Implementation in BALLISTICS-ENGINE . . . 357
18.4.3 Why RK4 for Ballistics? . . . 358
18.4.4 Target Detection and Interpolation . . . 358
18.5 The Adaptive RK45 (Dormand–Prince) Method . . . 359
18.5.1 The Dormand–Prince Algorithm . . . 359
18.5.2 Error Estimation and Step Control . . . 359
18.5.3 RK4 vs. RK45: When to Use Which . . . 360
18.6 Step Size Selection and Adaptive Stepping . . . 361
18.6.1 Fixed Step Sizes . . . 361
18.6.2 Adaptive Step Sizes in RK45 . . . 362
18.6.3 Wind Shear and Step Size . . . 362
18.6.4 Safety Limits . . . 362
18.7 Accuracy vs. Performance: Choosing the Right Step Size . . . 363
18.7.1 Convergence Test . . . 363
18.7.2 Practical Guidelines . . . 364
18.7.3 The Adaptive Advantage . . . 364
18.8 The Fast Trajectory Solver . . . 365
18.8.1 Design Philosophy . . . 365
18.8.2 The `FastSolution` Structure . . . 366
18.8.3 Event Detection . . . 366
18.8.4 Interpolation on the Fast Solution . . . 366
18.8.5 When to Use the Fast Solver . . . 367
Exercises . . . 368

**19 Trajectory Sampling** **369**
19.1 The Sampling Problem: Continuous Solver, Discrete Output . . . 369
19.2 Linear Interpolation Between Integration Steps . . . 370
19.2.1 The Interpolation Function . . . 370
19.3 The Trajectory Sampling Pipeline . . . 371
19.3.1 Input Data Structures . . . 371
19.3.2 Step 1: Extract Coordinate Arrays . . . 372
19.3.3 Step 2: Generate Sampling Distances . . . 372
19.3.4 Step 3: Interpolate All Quantities . . . 373
19.3.5 Step 4: Compute Drop Relative to Line of Sight . . . 374
19.3.6 Step 5: Assemble Sample Points . . . 374
19.4 Trajectory Flags: Zero Crossings, Apex, Mach Transitions . . . 375
19.4.1 Zero Crossing Detection . . . 375
19.4.2 Mach Transition Detection . . . 376

19.4.3 Apex Detection . . . 376
19.5 Output at Specific Ranges vs. Fixed Step Sizes . . . 377
19.5.1 Fixed-Step Output . . . 377
19.5.2 Full Trajectory Output . . . 378
19.5.3 Minimum Step Size Guard . . . 378
19.6 JSON, CSV, and Table Output Formatting . . . 378
19.6.1 Table Output . . . 378
19.6.2 JSON Output . . . 379
19.6.3 CSV Output . . . 380
19.6.4 PDF Output . . . 380
19.6.5 The `TrajectoryDict` Conversion . . . 381
Exercises . . . 381

**20 The Zeroing Algorithm** **383**
20.1 The Zeroing Problem: Finding the Launch Angle . . . 383
20.2 Bisection Search: Reliable but Slow . . . 384
20.2.1 Strengths and Weaknesses . . . 384
20.3 Newton's Method: Fast but Fragile . . . 385
20.4 The Hybrid Approach in BALLISTICS-ENGINE . . . 386
20.4.1 The Three Strategies . . . 386
20.4.2 Implementation Walkthrough . . . 386
20.4.3 Safe Division Guards . . . 388
20.4.4 The `AngleResult` Structure . . . 388
20.5 The Zeroing Pipeline . . . 388
20.5.1 The `zero_angle()` Function . . . 388
20.5.2 The `solve_muzzle_angle()` Function . . . 390
20.6 Convergence Criteria and Tolerances . . . 390
20.6.1 Primary Tolerance . . . 390
20.6.2 Scaled Tolerance . . . 391
20.6.3 Iteration Limits . . . 391
20.6.4 Relaxed Success Criteria . . . 391
20.7 Edge Cases: Steep Angles, Very Long Range, Subsonic Zeros . . . 392
20.7.1 Elevated and Depressed Targets . . . 392
20.7.2 Very Long Range . . . 393
20.7.3 Subsonic Zeros . . . 394
20.7.4 The Auto-Zero Workflow . . . 395
20.7.5 Powder Temperature Sensitivity . . . 395
20.7.6 Quick Drop Estimates . . . 396
Exercises . . . 397

**VII Online Mode & Weather** 399

**21 Online Mode** 401
21.1 What Online Mode Does . . . 401
21.2 The `--online` Flag . . . 402
21.2.1 Feature Gating in the Source . . . 403
21.2.2 Companion Flags . . . 403
21.3 Weather Data Sources and APIs . . . 404
21.3.1 The Flask API Architecture . . . 404
21.3.2 API Endpoints . . . 405
21.3.3 Error Handling . . . 406
21.4 Location Specification: `--lat` / `--lon` . . . 406
21.4.1 How Location Flows Through the Request . . . 407
21.4.2 Location from Profiles . . . 408
21.5 What Data Is Fetched . . . 408
21.5.1 The Outbound Request . . . 408
21.5.2 The Inbound Response . . . 409
21.5.3 Unit Conversion at the Boundary . . . 410
21.6 Comparing Offline vs. Online Predictions . . . 411
21.6.1 When to Use Compare Mode . . . 412
21.6.2 Interpreting Deltas . . . 413
21.7 Velocity Truing via the API . . . 413
21.8 BC Table Downloads . . . 414
21.9 Under the Hood: Request Lifecycle . . . 415
21.10 Practical Considerations . . . 415
21.10.1 When Online Mode Is Worth It . . . 415
21.10.2 Combining Online Mode with Profiles . . . 416
21.11 Exercises . . . 416

**22 Weather Integration** 419
22.1 Weather Station Data and METAR Decoding . . . 419
22.1.1 What Is a METAR? . . . 419
22.1.2 Converting METAR Data to CLI Flags . . . 420
22.1.3 Deriving Humidity from Dew Point . . . 421
22.2 Atmospheric Profiles Along the Trajectory . . . 421
22.2.1 The Homogeneous Atmosphere Assumption . . . 421
22.2.2 When Homogeneity Breaks Down . . . 422
22.3 Temperature Lapse Rates . . . 423
22.3.1 The ICAO Standard Lapse Rate . . . 423
22.3.2 Actual vs. Standard Lapse Rates . . . 424

22.3.3 Quantifying Lapse Rate Impact . . . 425
22.4 Humidity Effects on Air Density . . . 426
22.4.1 The Physics: Dalton's Law . . . 426
22.4.2 Implementation: The Arden Buck Equation . . . 426
22.4.3 The CIPM Formula for Precise Air Density . . . 427
22.4.4 Quantifying the Humidity Effect . . . 428
22.4.5 Humidity and the Speed of Sound . . . 429
22.5 Barometric Pressure Corrections . . . 429
22.5.1 Station Pressure vs. Sea-Level Pressure . . . 429
22.5.2 The Barometric Formula . . . 430
22.5.3 Practical Pressure Guidelines . . . 430
22.5.4 The `get_local_atmosphere` Function . . . 431
22.6 Wind Shear and Altitude-Dependent Wind . . . 432
22.6.1 Wind Shear Models . . . 432
22.6.2 Enabling Wind Shear in Practice . . . 435
22.7 Weather Zones and Interpolation . . . 436
22.7.1 What Are Weather Zones? . . . 436
22.7.2 Zone Interpolation Methods . . . 436
22.7.3 Weather Zones and the Wind Sock . . . 437
22.8 Practical Weather Workflow for Range Day . . . 437
22.8.1 Step 1: Before You Leave . . . 437
22.8.2 Step 2: At the Range . . . 438
22.8.3 Step 3: Validate and Adjust . . . 438
22.8.4 Step 4: Monitor Changing Conditions . . . 439
22.8.5 A Complete Range-Day Command . . . 439
22.9 Exercises . . . 440

**VIII Real-World Applications** **441**

**23 Hunting Applications** **443**
23.1 Point-Blank Range and Vital Zone Sizing . . . 443
23.1.1 What Is Point-Blank Range? . . . 443
23.1.2 Computing MPBR . . . 444
23.1.3 Vital Zone Sizing for Different Game . . . 445
23.1.4 MPBR and the Flat-Shooting Myth . . . 445
23.2 Maximum Effective Range: Energy and Accuracy Thresholds . . . 446
23.2.1 Energy Thresholds . . . 446
23.2.2 Finding the Energy-Limited Range . . . 447
23.2.3 Accuracy Threshold . . . 447

23.2.4 Combining Constraints . . . 448
23.2.5 Velocity and Bullet Expansion . . . 448
23.2.6 Wind Considerations for Hunters . . . 449
23.3 Uphill and Downhill Shooting: The Rifleman's Rule . . . 450
23.3.1 Why Angle Matters . . . 450
23.3.2 Computing Angled Shots . . . 451
23.3.3 How Much Does It Matter? . . . 451
23.3.4 When the Rifleman's Rule Breaks Down . . . 452
23.4 Altitude Corrections for Mountain Hunting . . . 452
23.4.1 The Magnitude of the Effect . . . 453
23.4.2 Re-Zeroing for Altitude . . . 453
23.4.3 Density Altitude: A Single Number . . . 454
23.5 Building a Field-Ready Dope Card . . . 454
23.5.1 What Goes on a Dope Card . . . 454
23.5.2 The Come-Ups Subcommand . . . 455
23.5.3 PDF Dope Cards . . . 455
23.5.4 Multiple Conditions, Multiple Cards . . . 456
23.6 Example Setups: Common Hunting Cartridges . . . 457
23.6.1 6.5 Creedmoor, 143 gr ELD-X . . . 457
23.6.2 .270 Winchester, 150 gr Partition . . . 458
23.6.3 .30-06 Springfield, 180 gr AccuBond . . . 458
23.6.4 .300 Winchester Magnum, 200 gr ELD-X . . . 459
23.6.5 .375 H&H Magnum, 300 gr Swift A-Frame . . . 460
23.6.6 Cartridge Comparison Summary . . . 461
Exercises . . . 461
What's Next . . . 463

**24 Precision Rifle Competition 465**
24.1 Competition Requirements: MOA and MIL Precision . . . 465
24.1.1 How Precise Is Precise Enough? . . . 465
24.1.2 The Precision Budget . . . 466
24.1.3 PRS/NRL vs. F-Class: Different Precision Demands . . . 466
24.1.4 Cartridge Selection for Competition . . . 467
24.2 Building Competition Dope Books . . . 468
24.2.1 Standard Dope Format . . . 468
24.2.2 Generating Fine-Grained Dope . . . 468
24.2.3 Truing Your Dope . . . 469
24.2.4 Exporting for External Tools . . . 470
24.3 Wind Calls: Using Monte Carlo for Probability of Hit . . . 470
24.3.1 Modeling Wind Uncertainty . . . 470

24.3.2 Understanding CEP and the Confidence Ellipse . . . 471
24.3.3 Practical Wind-Call Strategy . . . 471
24.4 Stage Planning with First-Round Hit Probability . . . 473
24.4.1 The Time-Accuracy Trade-Off . . . 473
24.4.2 Computing FRHP with Monte Carlo . . . 473
24.4.3 Stage Sequence Optimization . . . 473
24.4.4 Expected Value Decision-Making . . . 474
24.4.5 Transonic Awareness for Competition . . . 475
24.5 Spin Drift and Coriolis at Competition Ranges . . . 476
24.5.1 When Spin Drift Matters . . . 476
24.5.2 Coriolis Effect at Competition Ranges . . . 477
24.5.3 Combined Advanced Effects . . . 477
24.6 Does Your Scope Track? . . . 478
24.7 Making Power Factor . . . 480
24.8 Competition Workflow: Pre-Match Preparation . . . 481
24.8.1 Step 1: True Your Data . . . 481
24.8.2 Step 2: Save a Profile . . . 482
24.8.3 Step 3: Build Match-Specific Dope . . . 482
24.8.4 Step 4: Run Wind Scenarios . . . 483
24.8.5 Step 5: Monte Carlo Confidence Check . . . 483
24.8.6 Step 6: Print and Prepare . . . 484
Exercises . . . 484
What's Next . . . 486

**25 Load Development 487**
25.1 Predicting Muzzle Velocity from Barrel Length . . . 488
25.1.1 The Barrel Length–Velocity Relationship . . . 488
25.1.2 Modeling the Effect . . . 488
25.1.3 Short Barrels for Hunting . . . 489
25.2 BC Optimization: Selecting Bullets for Your Application . . . 489
25.2.1 The BC–Weight Trade-Off . . . 489
25.2.2 Comparing Bullet Choices . . . 490
25.2.3 Side by Side: the `compare` Command . . . 491
25.2.4 Sectional Density as a Selection Guide . . . 492
25.2.5 G1 vs. G7: Choosing the Right Drag Model for Your Bullet . . . 493
25.2.6 The Crossover Range . . . 494
25.3 Ladder Test Analysis with `ballistics-engine` . . . 494
25.3.1 What Is a Ladder Test? . . . 494
25.3.2 Using the Solver for Analysis . . . 495
25.3.3 Estimating BC from Ladder Data . . . 496

25.4 Free Recoil . . . 497
25.5 Velocity SD and Its Impact on Long-Range Accuracy . . . 498
25.5.1 Why Velocity Consistency Matters . . . 498
25.5.2 Quantifying the Effect . . . 498
25.5.3 The Drop Sensitivity Factor . . . 499
25.5.4 ES vs. SD: Which Metric to Trust . . . 500
25.5.5 How Many Rounds to Chronograph . . . 500
25.6 Working Up Loads: A Safe Workflow . . . 501
25.6.1 The Handloader's Workflow with a Solver . . . 501
25.6.2 Predicting the Effect of Charge Changes . . . 502
25.6.3 Powder Temperature Sensitivity . . . 503
Exercises . . . 506
What's Next . . . 508

**IX Under the Hood** **509**

**26 Architecture** **511**
26.1 Module Organization: The `src/` Directory . . . 511
26.1.1 Core Solver . . . 512
26.1.2 Drag and Ballistic Coefficient . . . 512
26.1.3 Atmosphere and Wind . . . 513
26.1.4 Advanced Physics . . . 513
26.1.5 Platform Bindings . . . 514
26.1.6 Monte Carlo and Statistical Analysis . . . 514
26.1.7 Online and Auxiliary . . . 515
26.1.8 Inspecting the Module Tree . . . 515
26.2 The Solver Pipeline . . . 516
26.2.1 Stage 1: Input Parsing . . . 517
26.2.2 Stage 2: Atmosphere Calculation . . . 518
26.2.3 Stage 3: Zeroing . . . 519
26.2.4 Stage 4: Numerical Integration . . . 519
26.2.5 Stage 5: Output Formatting . . . 521
26.3 Data Flow Through the Engine . . . 522
26.3.1 The State Vector . . . 522
26.3.2 Data Ownership and Lifetimes . . . 523
26.4 Key Data Structures . . . 524
26.4.1 `BallisticInputs` . . . 524
26.4.2 `InitialConditions` . . . 525
26.4.3 `TrajectoryParams` . . . 526

26.4.4 `TrajectoryPoint` and `TrajectoryResult` . . . 526
26.5 Design Decisions . . . 527
26.5.1 Why Rust? . . . 527
26.5.2 Why nalgebra? . . . 528
26.5.3 Why a Flat Module Layout? . . . 529
26.5.4 Three Crate Types . . . 529
26.5.5 Feature Gating . . . 530
26.5.6 Release Profile Optimization . . . 530
26.6 The Derivative Function: Where Physics Lives . . . 531
26.6.1 The Magnus Moment Coefficient . . . 533
26.7 The Constants Module . . . 533
26.7.1 Physical Constants . . . 533
26.7.2 Unit Conversion Factors . . . 534
26.7.3 Numerical Tolerances . . . 534
26.7.4 BC Fallback Tables . . . 534
26.8 The Dependency Graph . . . 535
26.9 Exercises . . . 536

**27 FFI, WASM & Python Bindings 539**
27.1 The C FFI . . . 539
27.1.1 Why a C ABI? . . . 539
27.1.2 FFI Data Structures . . . 540
27.1.3 Exported Functions . . . 542
27.1.4 Building the FFI Library . . . 543
27.1.5 Calling from C . . . 544
27.1.6 Calling from Swift . . . 546
27.2 WebAssembly: Compiling for the Browser . . . 547
27.2.1 The `WasmBallistics` API . . . 547
27.2.2 The `Calculator` API . . . 548
27.2.3 Building the WASM Module . . . 550
27.2.4 Integrating into a Web Page . . . 551
27.3 Python Bindings via PyO3 . . . 552
27.3.1 Architecture of the Python Bindings . . . 553
27.3.2 Installation . . . 553
27.3.3 Using from Python . . . 553
27.3.4 Monte Carlo in Python . . . 554
27.4 Protocol Surfaces: `solve-json` and MCP . . . 555
27.4.1 solve-json v1 . . . 555
27.4.2 Bounds, and One Place to Condition Inputs . . . 557
27.4.3 The MCP Server . . . 558

27.5 Embedding `ballistics-engine` in Your Application . . . 558
27.5.1 Choosing the Right Binding . . . 558
27.5.2 The `convert_inputs` Pattern . . . 559
27.6 Cross-Platform Considerations . . . 560
27.6.1 Surface Consistency: The 0.22.0 Sweep . . . 560
27.6.2 Browser Parity and a Hardened FFI Boundary . . . 561
27.6.3 Endianness and Alignment . . . 561
27.6.4 Thread Safety . . . 561
27.6.5 Error Handling Across Boundaries . . . 562
27.6.6 WASM Random Number Generation . . . 562
27.6.7 Supported Platforms . . . 563
27.7 Exercises . . . 563

**28 Performance** **567**
28.1 Benchmark Results . . . 567
28.1.1 The Criterion Benchmark Suite . . . 567
28.1.2 Single-Trajectory Performance . . . 568
28.1.3 Running Your Own Benchmarks . . . 569
28.1.4 Integration Method Comparison . . . 569
28.2 Monte Carlo Parallelism . . . 570
28.2.1 Current Implementation . . . 570
28.2.2 Parallelism with Rayon . . . 571
28.2.3 Scaling Behavior . . . 572
28.3 The Fast Trajectory Solver . . . 573
28.3.1 Design Differences . . . 573
28.3.2 The Simplified Derivative . . . 573
28.3.3 Raw Arrays vs. nalgebra . . . 575
28.3.4 When to Use the Fast Solver . . . 575
28.4 Memory Layout and Cache Efficiency . . . 576
28.4.1 Data Locality in the Hot Path . . . 576
28.4.2 Drag Table Lookup . . . 576
28.4.3 The `TrajectoryParams` Separation . . . 577
28.4.4 Allocation Patterns . . . 577
28.4.5 The Effect of LTO and codegen-units . . . 577
28.5 Alternative Allocators . . . 578
28.5.1 jemalloc . . . 578
28.5.2 mimalloc . . . 579
28.6 Profiling the Engine . . . 579
28.6.1 CPU Profiling with `perf` . . . 579
28.6.2 Flame Graphs . . . 580

28.6.3 Profiling on macOS with Instruments . . . 580
28.6.4 Cache Analysis . . . 581
28.7 WASM Performance Considerations . . . 581
28.8 Performance Optimization Checklist . . . 582
28.9 Exercises . . . 583

**X Reticles & Online Solvers 587**

**29 Reticles, Holds, and BDC Solvers 589**
29.1 Describing a Reticle . . . 589
29.2 Placing a Solution in the Glass . . . 591
29.2.1 First and Second Focal Plane . . . 592
29.2.2 Reticles on Profiles and in Solutions . . . 593
29.3 Working Backwards: The Inverse Solvers . . . 594
29.3.1 `mark-to-range`: What Is This Mark Good For? . . . 594
29.3.2 `bdc-match`: What Magnification Makes This Fit? . . . 595
29.3.3 `optimal-zero`: One Zero for a Whole Stage . . . 596
29.4 Hold Corridors Across Wind Scenarios . . . 597
29.4.1 Two Metrics, Because the Shapes Differ . . . 598
29.4.2 What the Command Refuses to Do . . . 599
29.5 Exercises . . . 599

**30 The Online Reverse Solvers 601**
30.1 Signing In . . . 601
30.2 `recommend-twist`: What Barrel Do I Need? . . . 603
30.3 `recommend-powder`: Getting to a Velocity . . . 603
30.4 `recommend-col`: Seating Depth . . . 604
30.5 `calibrate-bc`: A BC From Dimensions . . . 604
30.6 Importing a Third-Party Reticle . . . 605
30.7 Exercises . . . 605

**A CLI Reference 607**
A.1 Global Options . . . 607
A.2 trajectory — Single Trajectory Calculation . . . 608
A.2.1 Core Ballistic Parameters . . . 608
A.2.2 Profile and Location Loading . . . 608
A.2.3 Wind and Atmosphere . . . 609
A.2.4 Zeroing and Geometry . . . 610
A.2.5 Advanced Physics Flags . . . 610

A.2.6 Bullet and Barrel Parameters . . . 611
A.2.7 Zero-Day Conditions . . . 612
A.2.8 Powder Temperature Sensitivity . . . 612
A.2.9 BC Correction Tables . . . 613
A.2.10 Output and PDF Options . . . 613
A.2.11 Examples . . . 615
A.3 zero — Calculate Zero Angle . . . 615
A.3.1 Example . . . 616
A.4 monte-carlo — Monte Carlo Simulation . . . 617
A.4.1 Example . . . 618
A.5 BC Estimation and Modeling Commands . . . 618
A.5.1 estimate-bc — Estimate BC from Trajectory Data . . . 618
A.5.2 generate-bc-segments — Velocity-Dependent BC Segments . . . 620
A.5.3 true-velocity — Effective Muzzle Velocity from Observed Drop . . . 620
A.6 Utility Commands . . . 621
A.6.1 mpbr — Maximum Point-Blank Range . . . 621
A.6.2 come-ups — Elevation Adjustment Table . . . 622
A.6.3 wind-card — Wind Drift Card . . . 624
A.6.4 range-table — Comprehensive Range Table . . . 625
A.6.5 stability — Gyroscopic Stability Analysis . . . 626
A.7 Comparison and Calculator Commands . . . 626
A.7.1 compare — Multi-Load Comparison . . . 626
A.7.2 powder — Powder-Temperature Velocity . . . 627
A.7.3 tall-target — Scope Tracking Correction Factor . . . 628
A.7.4 true-wind — Wind-Call Truing . . . 628
A.7.5 reticle — Reticle Descriptions and Hold Points . . . 628
A.7.6 mark-to-range, bdc-match, optimal-zero . . . 629
A.7.7 hold-corridor — Robust Holds Across Wind Scenarios . . . 629
A.7.8 drag-curve — Print a Reference Drag Curve . . . 630
A.7.9 recoil — Free Recoil . . . 630
A.7.10 power-factor — Competition Power Factor . . . 631
A.7.11 plan-truing — Truing Experiment Design . . . 631
A.7.12 true-velocity: uncertainty-aware flags . . . 631
A.8 profile — Manage Saved Profiles . . . 632
A.8.1 profile save . . . 632
A.8.2 profile list . . . 633
A.8.3 profile show . . . 633
A.8.4 profile import . . . 633
A.8.5 profile delete . . . 634
A.9 Online Reverse Solvers . . . 634

A.9.1 login / logout . . . 634
A.9.2 recommend-powder . . . 635
A.9.3 recommend-twist . . . 635
A.9.4 recommend-col . . . 635
A.9.5 calibrate-bc . . . 635
A.10 Protocol Commands . . . 636
A.10.1 solve-json — One Solve, One JSON Document . . . 636
A.10.2 mcp — Model Context Protocol Server . . . 636
A.11 completions — Shell Completion Generation . . . 637
A.12 Output Formats . . . 637
A.12.1 Piping and Composition . . . 638
A.13 Online Mode Flags . . . 638
A.13.1 API Connection Flags . . . 638
A.13.2 Weather Zone Flags . . . 639
A.13.3 BC5D Auto-Download Flags . . . 639
A.13.4 Example: Online Trajectory with Weather Zones . . . 640

**B Physics Constants 641**
B.1 Gravitational and Atmospheric Constants . . . 641
B.2 Unit Conversion Factors . . . 642
B.3 Standard Projectile Reference Values . . . 643
B.3.1 Overall Fallback . . . 643
B.3.2 By Projectile Weight . . . 643
B.3.3 By Caliber . . . 643
B.4 ICAO Standard Atmosphere . . . 644
B.5 Air Density Calculation Constants . . . 645
B.5.1 Gas Constants and Molar Masses . . . 645
B.5.2 Saturation Vapor Pressure . . . 645
B.5.3 Enhancement Factor . . . 646
B.5.4 Compressibility Factor (Virial Coefficients) . . . 646
B.6 Numerical Stability Constants . . . 647

**C Drag Tables 649**
C.1 G1 Standard Drag Model . . . 650
C.2 G7 Standard Drag Model . . . 651
C.3 Other Standard Drag Models . . . 653
C.4 Interpolation Method . . . 655
C.5 Custom Drag Tables . . . 656

**D BC5D Table Format 659**

D.1 Overview . . . 659
D.2 BC5D Binary Header Format . . . 660
D.3 Five-Dimensional Bin Layout . . . 661
D.3.1 Bin Definitions . . . 661
D.3.2 Dimension Summary . . . 661
D.3.3 Data Section and Memory Layout . . . 662
D.3.4 Total File Size . . . 662
D.4 4D Linear Interpolation . . . 663
D.4.1 Finding Bracketing Indices . . . 663
D.4.2 Hypercube Interpolation . . . 663
D.5 Checksum Verification (CRC32) . . . 664
D.5.1 Algorithm . . . 664
D.5.2 Data Covered by the Checksum . . . 664
D.5.3 Verification Procedure . . . 664
D.6 File Naming and Discovery . . . 665
D.6.1 Naming Convention . . . 665
D.6.2 Caliber Key Calculation . . . 665
D.6.3 Discovery Algorithm . . . 666
D.7 Table Download and Caching . . . 666
D.7.1 Manifest Format . . . 666
D.7.2 Cache Directories . . . 667
D.7.3 Download Flow . . . 667
D.7.4 CLI Flags . . . 667
D.8 The Bc5dTableManager . . . 668
D.8.1 In-Memory Caching . . . 668
D.8.2 Key Methods . . . 668
D.9 Legacy BCCR Format . . . 669
D.10 Error Handling . . . 670

**E Glossary** **673**
E.1 Terms A–M . . . 673
E.2 Terms N–Z . . . 681

# Part I

# Getting Started

# Chapter 1

# Introduction

*"The bullet's path is the longest conversation between physics and the wind."*

Every time a bullet leaves a muzzle, it enters a world ruled by gravity, aerodynamic drag, and the capricious atmosphere. Whether you are a precision rifle competitor dialing a scope turret for a 1,000-yard shot, a hunter estimating hold-over at last light, or a ballistician validating a new projectile design, you need answers that you can trust—and you need them fast.

This book is about getting those answers with `ballistics-engine`, an open-source trajectory solver written in Rust.

## 1.1 What Is `ballistics-engine`?

`ballistics-engine` is a high-performance computational exterior ballistics tool that ships as both a command-line application and a reusable Rust library. At its core, it solves the differential equations of projectile motion—accounting for gravity, aerodynamic drag, wind, and the real atmosphere—and returns a full trajectory table from the muzzle to the target and beyond.

Listing 1.1: A first look: .308 Win trajectory to 1,000 yards.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000
```

That single command computes a zeroed trajectory for a .308 Winchester, 168-grain Sierra MatchKing (G1 BC `0.475`) at 2700 fps, printing drop, drift, velocity, energy, and time of flight at every range increment out to 1000 yards.

But `ballistics-engine` goes far beyond basic point-mass trajectories. Its feature set includes:

- **Full 3-D trajectory integration** with adaptive Dormand–Prince (RK45) and fixed-step Runge–Kutta 4 solvers.
- **Multiple drag models**: G1, G2, G5, G6, G7, G8, GI, and GS, plus custom drag curves with automatic transonic corrections.
- **ICAO Standard Atmosphere** modeling with temperature, pressure, humidity, and altitude effects across all seven atmospheric layers.
- **Automatic zeroing**: calculate the sight adjustment needed for any zero distance and apply it in a single command.
- **Monte Carlo simulation**: statistical analysis with parameter uncertainties for velocity, BC, wind, and pointing error.
- **BC estimation and truing**: estimate ballistic coefficients from observed trajectory data, or true your muzzle velocity from field measurements.
- **Advanced physics**: spin drift, Magnus effect, Coriolis and Eötvös corrections, gyroscopic precession and nutation, pitch damping, and transonic stability analysis.
- **Multiple output formats**: human-readable tables, JSON for automation, CSV for spreadsheets, and PDF dope cards for the range.
- **Cross-platform**: runs on macOS, Linux, Windows, FreeBSD, OpenBSD, and NetBSD, and compiles to WebAssembly for browser-based use.

**Note**

`ballistics-engine` is licensed under the MIT and Apache 2.0 licenses. The optional `--online` mode connects to a proprietary cloud service with separate terms; everything else runs entirely on your machine with no network access required.

The project is published on `crates.io` as the `ballistics-engine` crate (currently at version 0.14.1) and hosted on GitHub at `https://github.com/ajokela/ballistics-engine`. Language bindings for Python, Ruby, and WebAssembly are maintained in separate repositories, and a Foreign Function Interface (FFI) layer supports iOS and Android integration.

### 1.1.1 The CLI and the Library

It is important to understand that `ballistics-engine` is two things at once. The `ballistics` binary is a command-line interface (CLI) that parses flags, converts units, and formats output. Beneath it lies a pure Rust library—the `ballistics_engine` crate—that implements all the physics and numerics without any I/O or unit-system assumptions.

This separation matters because the library can be embedded in places the CLI cannot reach: a Swift application via the FFI layer, a Python script via PyO3 bindings, a browser tab via WebAssembly, or another Rust project via a standard `cargo` dependency. The CLI is the primary interface for this book, but whenever we discuss the underlying algorithms, we are really talking about the library.

The binary is defined in `src/main.rs`; the library is exported from `src/lib.rs`. `Cargo.toml` declares three crate types—`rlib` (the Rust library), `staticlib` (for C/Swift/FFI linking), and `cdylib` (the shared library for Python, Ruby, and WASM):

Listing 1.2: Crate type declaration in `Cargo.toml`.

```
[lib]
name = "ballistics_engine"
crate-type = ["rlib", "staticlib", "cdylib"]
```

The practical consequence is that every computation you run from the command line is also available programmatically. A JSON output from the CLI is structurally identical to what the library returns—just serialized for human or machine consumption.

### 1.1.2 A Tour of the Subcommands

The `ballistics` CLI organizes its functionality into subcommands, each targeting a specific workflow. Here is the complete list at the time of writing:

| Subcommand | Purpose |
|---|---|
| `trajectory` | Compute a single trajectory |
| `zero` | Calculate the zero angle for a target distance |
| `monte-carlo` | Run Monte Carlo statistical simulation |
| `estimate-bc` | Estimate BC from observed trajectory data |
| `true-velocity` | Calculate effective muzzle velocity from field data |
| `come-ups` | Generate an elevation adjustment table |
| `wind-card` | Generate a wind-drift card at multiple speeds |
| `range-table` | Generate a comprehensive range table |
| `mpbr` | Calculate Maximum Point-Blank Range |
| `stability` | Analyze gyroscopic stability (Miller stability factor) |
| `profile` | Manage saved ballistic profiles |
| `generate-completion` | Generate shell completions for Bash/Zsh/Fish/PowerShell |

Part II of this book (Chapter 4–Chapter 8) covers the primary subcommands in detail. We will use `trajectory`, `zero`, and `monte-carlo` most frequently.

## 1.2 Why Computational Ballistics Matters

A century ago, a rifleman's trajectory knowledge was limited to rules of thumb, come-up tables laboriously compiled at the range, and—if he was fortunate—a printed drop chart from the ammunition manufacturer. Each of those resources was tied to a specific set of conditions: a particular altitude, temperature, and wind speed. Change the conditions, and the data was stale.

Today, computational ballistics has transformed the way shooters interact with their rifles. A solver on a smartphone or a laptop can generate a custom trajectory table for the exact conditions at the firing point—right now, with the current temperature, barometric pressure, and wind call.

### 1.2.1 Bridging Theory and Practice

The physics of exterior ballistics are well understood. Newton's second law, the aerodynamic drag equation, and the standard atmosphere model have been established for over a century. What has changed is our ability to *solve* the resulting equations quickly and accurately on commodity hardware.

Consider a straightforward problem: a .308 Winchester, 175-grain bullet (G7 BC `0.253`) at 2650 fps. You want to know the drop at 600 yards under the following field conditions:

- Temperature: 75 °F (24 °C)
- Altitude: 5000 ft (1524 m)
- Barometric pressure: 24.92 inHg (844 hPa)
- 8 mph crosswind from 3 o'clock

Listing 1.3: A practical field problem.

```
ballistics trajectory \
  -v 2650 -b 0.253 -m 175 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 600 \
  --temperature 75 --pressure 24.92 --humidity 40 \
  --altitude 5000 \
  --wind-speed 8 --wind-direction 90
```

In well under a second, `ballistics-engine` returns the drop in MOA and milliradians, the wind deflection, the remaining velocity and energy, and the time of flight. Manually calculating this—with density-altitude corrections, velocity-dependent drag, and a proper crosswind component—would take an experienced ballistician many minutes with a scientific calculator.

### 1.2.2 Workflows That Depend on a Solver

The utility of a ballistics solver extends well beyond generating a single drop chart. Here are several workflows that are impractical without computational support:

**Velocity truing.** You fire a group at a known distance and measure the actual drop. The solver works backwards to find the effective muzzle velocity that produces that drop, correcting for chronograph error or lot-to-lot velocity variation:

```
ballistics true-velocity \
  --measured-drop 5.1 --range 600 \
  --bc 0.253 --drag-model g7 \
  --mass 175 --diameter 0.308 \
  --chrono-velocity 2650 --offline
```

**Monte Carlo hit probability.** You want to know not just where the bullet *should* go, but the spread of possible impacts given real-world variation in velocity, BC, wind, and pointing error:

```
ballistics monte-carlo \
  -v 2650 -b 0.253 -m 175 -d 0.308 \
  -n 1000 --velocity-std 10 \
  --bc-std 0.005 --wind-std 2 \
  --target-distance 600
```

**Maximum point-blank range.** For a hunting zero, you want the farthest distance at which the bullet stays within a defined kill zone (e.g., $\pm 3$ inches of line of sight):

```
ballistics mpbr \
  -v 2650 -b 0.460 -m 180 -d 0.308 \
  --vital-zone 6
```

**Stability analysis.** Before taking a .223 Remington to 600 yards, you want to know whether your 1:9 twist rate will keep a 77-grain bullet stable through the transonic regime:

```
ballistics stability \
  --mass 77 --diameter 0.224 \
  --length 1.0 --twist-rate 9
```

None of these workflows requires a degree in physics. They do require a solver that you can trust, and understanding enough of the underlying physics to recognize when inputs are wrong and outputs are suspect. That is the purpose of this book.

### 1.2.3 Why Another Ballistics Solver?

Several excellent ballistics applications already exist, from commercial products like Applied Ballistics to open-source efforts. So why build `ballistics-engine`? The project was motivated by several goals:

1. **Performance.** Written in Rust, `ballistics-engine` delivers native-code performance with memory safety guarantees. A single 1,000-meter trajectory completes in approximately 5 ms; a 1,000-iteration Monte Carlo simulation runs in about 500 ms. This speed enables real-time interactive use and large-scale batch processing.

2. **Transparency.** The source code is open, the physics models are documented, and the drag tables are embedded in the binary. You can inspect every constant, every integration step, and every atmospheric correction.

3. **Composability.** The CLI tool outputs structured JSON and CSV alongside human-readable tables, making it trivial to integrate into shell scripts, data pipelines, and web applications. The library crate can be embedded directly into other Rust projects, called from C/Swift/Java through the FFI layer, or compiled to WebAssembly for the browser.

4. **Extensibility.** The modular architecture separates drag models, atmospheric calculations, and advanced physics effects into individual modules with their own feature flags. Each advanced effect—spin drift, Coriolis, precession—can be enabled independently, so simple calculations remain fast and complex ones remain tractable.

5. **Cross-platform reach.** Pre-built binaries are available for macOS (x86_64 and ARM64), Linux (x86_64 and aarch64), Windows (x86_64), and the BSDs (FreeBSD, OpenBSD, NetBSD). The same engine runs in the browser via WebAssembly at `https://ballistics.sh`.

### 1.2.4 When Not to Use a Solver

**SAFETY: Safety Warning**

A ballistics solver is a *prediction tool*, not a substitute for live-fire verification. Always confirm your trajectory data at the range before relying on it in the field. Errors in input data—an incorrect muzzle velocity, an inaccurate BC, or a miscalibrated barometric sensor—propagate through the computation and can produce dangerous results. Never use computed trajectory data for load development without cross-referencing with published pressure data from a reputable source.

No solver, however sophisticated, can replace fundamentals. A clean trigger press, consistent shooting position, and accurate wind reading are worth more than another decimal place in your ballistic coefficient. The solver gives you the *mechanical* solution; you still supply the *human* judgment.

Furthermore, a solver's accuracy is bounded by its inputs. The most common sources of error in the field are:

1. **Incorrect muzzle velocity.** Chronograph errors of 30–50 fps are common, especially with optical chronographs in bright sunlight or at short muzzle distances. At 1000 yards, a 50 fps error can shift the predicted impact by several MOA.
2. **Inaccurate BC.** Manufacturer-published BCs often represent optimistic averages. Doppler-derived measurements are more reliable but not available for every bullet.
3. **Wind estimation.** The wind at the firing point is rarely the same as the wind at the target or along the bullet's flight path. No solver can compensate for a poor wind call.
4. **Atmospheric conditions.** Using "standard atmosphere" defaults instead of measured local conditions introduces systematic error that grows with range.

The `true-velocity` and `estimate-bc` subcommands exist precisely to address the first two problems—by working backward from observed field data to calibrate the solver's inputs. We will use these tools extensively in Part VIII.

## 1.3 Who This Book Is For

This book is written for three overlapping audiences:

**Practical shooters** who want to generate accurate trajectory tables, zero their rifles computationally, and understand the environmental factors that affect their shots. You do not need a physics degree—the early chapters introduce the relevant concepts from the ground up.

**Ballisticians and engineers** who want to understand the physics models, numerical methods, and drag data that power the engine. The later chapters walk through the solver's architecture, its Dormand–Prince integrator, the zeroing algorithm, and the Monte Carlo framework in detail, with references to the source code.

**Developers and integrators** who want to embed `ballistics-engine` into their own applications—whether through the Rust library API, the FFI layer, Python bindings, or WebAssembly. Dedicated chapters cover the architecture, the foreign-function interface, and performance tuning.

**Tip**

If you are a shooter who just wants to get up and running, start with Chapter 2 to install the tool and fire your first computed trajectory. You can always circle back to the physics chapters later.

We assume basic comfort with the command line (terminal on macOS or Linux, PowerShell or Command Prompt on Windows). No prior Rust experience is required to *use* the CLI tool, though the chapters on internals will be more rewarding if you can read Rust at a basic level.

### 1.3.1 How to Read This Book

You do not need to read this book front to back. The chapters are designed to be largely self-contained, with cross-references where dependencies exist. Here are three suggested reading paths:

**The Shooter's Path:** Chapters 1–5, then Chapter 23 (Hunting) or 24 (Competition) depending on your primary discipline. Revisit Chapter 9 (Atmosphere) and Chapter 12 (Advanced BC) when you want to push past 600 yards.

**The Engineer's Path:** Chapters 1–3, then Parts IV–VI for the full physics and numerics treatment. Chapter 26 (Architecture) ties it all together.

**The Developer's Path:** Chapters 1–2, then Chapter 26 (Architecture), Chapter 27 (FFI & WASM), and Chapter 28 (Performance). The Appendices provide the API reference.

## 1.4 A Brief History of Exterior Ballistics Computation

The science of predicting projectile flight is one of humanity's oldest applied-mathematics problems. Understanding its history helps us appreciate both the sophistication and the limitations of modern solvers like `ballistics-engine`.

### 1.4.1 From Tartaglia to Newton

In the sixteenth century, the Italian mathematician Niccolò Tartaglia published *Nova Scientia* (1537), one of the first attempts to apply mathematics to the trajectory of cannonballs. Tartaglia's model was crude—he assumed the trajectory consisted of straight-line segments connected by circular arcs—but it marked the beginning of a quantitative approach.

Galileo Galilei established the parabolic trajectory of projectiles in vacuum in his *Discorsi* (1638), and Isaac Newton's *Principia* (1687) introduced the concept of aerodynamic drag proportional to the square of velocity. Newton's drag law was the first physically motivated model, though it proved insufficiently detailed for accurate long-range prediction.

### 1.4.2 The Age of Firing Tables

The nineteenth century brought systematic experimental ballistics. Francis Bashforth, a British mathematician, developed chronograph techniques in the 1860s to measure projectile velocity at multiple points along the trajectory, enabling the first drag functions derived from real data. His work established the empirical foundation on which all subsequent drag models rest.

The Krupp works in Germany and the Gavre Commission in France conducted extensive firing trials, producing tables of drag coefficients as a function of velocity. These tables—predecessors of the G1 and G7 drag functions we use today—were interpolated by hand or with mechanical calculating machines.

By the early twentieth century, armies relied on printed "firing tables" that listed elevation and deflection for specific projectiles under standard conditions. Producing a single table required thousands of person-hours of manual computation.

### 1.4.3 Ballistic Computers and the BRL

The urgency of World War II led directly to the development of electronic computers. The U.S. Army's Ballistic Research Laboratory (BRL) at Aberdeen Proving Ground sponsored ENIAC (1945), the first general-purpose electronic computer, specifically to accelerate the production of artillery firing tables. A trajectory that took a human computer 20 hours could be completed in 30 seconds.

The BRL continued to advance the field through the Cold War era. Robert L. McCoy's *Modern Exterior Ballistics* (first published in 1999 by Schiffer Publishing) synthesized decades of BRL research into the definitive reference on the 6-degree-of-freedom (6-DOF) equations of motion for spinning projectiles. McCoy's work remains the theoretical foundation for `ballistics-engine` and virtually every other serious ballistics solver.

### 1.4.4 The Personal-Computer Revolution

The 1980s and 1990s saw ballistics software move from mainframes to personal computers. Sierra Bullets' *Infinity* software, Harold R. Vaughn's modeling work, and various shareware programs brought trajectory calculation to individual shooters for the first time.

The modern era was shaped by two developments:

1. **Bryan Litz and Applied Ballistics.** Beginning around 2009, Litz published doppler-radar–derived drag data for hundreds of commercial bullets, dramatically improving the accuracy of G7-referenced ballistic coefficients. His work demonstrated that measured drag data, properly referenced to an appropriate standard projectile, could bring predicted trajectories within fractions of an MOA of observed impacts.

2. **Smartphone solvers.** Applications like Applied Ballistics Mobile, Kestrel weather meters with built-in solvers, and various open-source projects put real-time trajectory computation in every shooter's pocket. The expectation shifted from "consult a printed table" to "compute a fresh solution at the firing point."

### 1.4.5 Where `ballistics-engine` Fits

`ballistics-engine` is a product of this lineage. It implements the point-mass (3-DOF) equations of motion with optional extensions toward 6-DOF effects (spin drift, precession, nutation), uses drag tables descended from the same military ballistics research that informed McCoy's work, and models the ICAO Standard Atmosphere for environmental corrections.

What distinguishes it is its implementation language (Rust), its open-source license, and its emphasis on composability: it is designed not merely to produce trajectory tables for human consumption, but to serve as a computational building block for other tools and workflows.

Listing 1.4: Piping JSON output to `jq` for post-processing.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  -o json | jq '.trajectory[] | select(.range >= 500)'
```

This heritage—from Tartaglia's geometry through McCoy's differential equations to a JSON-emitting CLI tool—is the thread that connects every chapter of this book.

## 1.5 How This Book Is Organized

*The Ballistics Engine Handbook* is divided into nine parts, progressing from basic use through advanced physics to the engine's internal architecture.

### Part I: Getting Started (Chapters 1–3)

You are here. This part introduces the project (Chapter 1), walks you through installation and your first trajectory computation (Chapter 2), and provides a primer on the essential physics of exterior ballistics (Chapter 3).

### Part II: The Commands (Chapters 4–8)

A comprehensive guide to every `ballistics` subcommand: `trajectory` computation, `zero` calculation, `monte-carlo` simulation, BC estimation and truing, and the profile system for storing and recalling load configurations.

### Part III: Atmosphere & Environment (Chapters 9–10)

How the atmosphere affects bullet flight and how `ballistics-engine` models temperature, pressure, humidity, altitude, and wind—including altitude-dependent wind shear.

### Part IV: Drag & BC Modeling (Chapters 11–13)

A deep dive into drag models (G1, G7, and others), the meaning and limitations of ballistic coefficients, velocity-dependent BC segmentation, and the BC5D correction-table system.

### Part V: Advanced Physics (Chapters 14–17)

Spin drift, Magnus effect, Coriolis and Eötvös corrections, gyroscopic precession and nutation, pitch damping, and the challenges of the transonic transition.

### Part VI: Numerical Methods (Chapters 18–20)

The trajectory solver's integrators (Euler, RK4, adaptive RK45 Dormand–Prince), trajectory sampling, and the binary-search zeroing algorithm—with references to the Rust source.

### Part VII: Online Mode & Weather (Chapters 21–22)

The optional online API, ML-augmented BC corrections, and weather-station integration.

### Part VIII: Real-World Applications (Chapters 23–25)

Practical workflows for hunting, precision rifle competition, and load development, demonstrating how to combine the engine's features into complete field solutions.

### Part IX: Under the Hood (Chapters 26–28)

The engine's module architecture, the FFI and WebAssembly layers, Python bindings, and performance profiling.

### Appendices

A complete CLI reference, physics constants, drag tables, the BC5D table format specification, and a glossary of terms.

**Tip**

Each chapter ends with a set of exercises—concrete, runnable `ballistics` commands that reinforce the chapter's concepts. Try them as you read; there is no better way to build intuition than to watch the numbers change as you vary the inputs.

## Conventions Used in This Book

Throughout this book, we use a consistent set of typographic and notational conventions.

`Monospaced text` indicates command names, flag names, file paths, and source-code identifiers.

`--auto-zero` indicates a CLI flag (the double-dash prefix is generated automatically).

`0.475` indicates a ballistic coefficient value.

**Imperial first, (metric)** Throughout this book, key quantities are given in imperial units first with metric equivalents in parentheses: "2700 fps (823 m/s)."

**Shell prompts** are omitted from CLI examples for clarity; every line beginning with `ballistics` is a command you can type into your terminal.

Colored boxes highlight special content:

**Notes**

Notes provide additional context or background information.

**Tips**

Tips offer practical advice for getting the most out of `ballistics-engine`.

**Warnings**

Warnings flag common mistakes or surprising behavior.

**SAFETY: Safety Warnings**

Safety warnings appear whenever the discussion involves pressure data, load development, or any situation where incorrect data could be dangerous.

**Example Definition**

**Definition boxes** introduce key terms and concepts with formal definitions for reference.

## Exercises

1. Verify that `ballistics-engine` is installed by running:

```
ballistics --version
```

   (If it is not yet installed, proceed to Chapter 2.)

2. Run the "first look" trajectory from Listing 1.1 and examine the output. How many yards does it take for the bullet to go subsonic?
3. Modify the command from Listing 1.1 to output JSON instead of a table:

   ```
   ballistics trajectory \
     -v 2700 -b 0.475 -m 168 -d 0.308 \
     --auto-zero 100 --max-range 1000 -o json
   ```

   Examine the JSON structure. What fields are present for each trajectory point?
4. Experiment with the `--drag-model` flag. Re-run the trajectory using G7 instead of G1 (note that you will need a G7-referenced BC—for the 168-grain SMK, use `0.224`):

   ```
   ballistics trajectory \
     -v 2700 -b 0.224 -m 168 -d 0.308 \
     --drag-model g7 \
     --auto-zero 100 --max-range 1000
   ```

   Compare the drop at 1000 yards between the G1 and G7 solutions. Which do you expect to be more accurate, and why? (We will explore this question in depth in Chapter 11.)
5. Generate a wind-drift card for the same .308 Win load at wind speeds of 5, 10, and 15 mph:

   ```
   ballistics wind-card \
     -v 2700 -b 0.475 -m 168 -d 0.308 \
     --zero-distance 100 --end 1000
   ```

   At 600 yards, how much additional drift does each 5 mph increment add?
6. Run the Monte Carlo simulation from the "Workflows" section and observe the hit probability and dispersion statistics. What happens when you double `--velocity-std` from 10 to 20?

## What's Next

Now that you know *what* `ballistics-engine` is and *why* it exists, it is time to get it running on your machine. Chapter 2 covers installation on every supported platform, building from source, and—most importantly—your first real trajectory calculation. Let's go.

# Chapter 2

# Installation & Your First Shot

Getting `ballistics-engine` running on your machine takes only a few minutes regardless of your operating system. This chapter walks you through every supported installation method, verifies that the binary works, and then—because there is no point in installing a ballistics solver without firing a shot—guides you through a complete 1000-yard .308 Winchester trajectory computation.

By the end of this chapter, you will have a working installation, an understanding of the CLI's flag structure, and the ability to produce trajectory data in three output formats.

## 2.1 Installing Pre-Built Binaries

Pre-built binaries are available for the following platforms:

| Platform | Architecture | Binary Name |
|---|---|---|
| macOS | x86_64, Apple Silicon (aarch64) | `ballistics` |
| Linux | x86_64, aarch64 | `ballistics` |
| Windows | x86_64 | `ballistics.exe` |
| FreeBSD | x86_64, aarch64 | `ballistics` |
| OpenBSD | x86_64 | `ballistics` |
| NetBSD | x86_64 | `ballistics` |

For all platforms, the simplest installation method is through Cargo, Rust's package manager. If you already have Rust installed, a single command installs the latest release from `crates.io`:

Listing 2.1: Universal installation via Cargo.

```
cargo install ballistics-engine
```

If you do not yet have Rust installed, the next few subsections show how to set it up on each platform.

### 2.1.1 macOS

The simplest path on macOS is to install the Rust toolchain via `rustup` and then use Cargo:

Listing 2.2: Installing the Rust toolchain and ballistics-engine on macOS.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source "$HOME/.cargo/env"
cargo install ballistics-engine
```

This downloads the source from `crates.io`, compiles an optimized release binary, and places it in `~/.cargo/bin/ballistics`. Ensure that `~/.cargo/bin` is on your `PATH`—the `rustup` installer typically adds it automatically.

**Note**

On Apple Silicon Macs (M1/M2/M3/M4), Cargo builds a native `aarch64` binary automatically. No cross-compilation is required—the resulting binary runs at full native speed.

If you prefer to download a pre-compiled binary from the GitHub releases page (`https://github.com/ajokela/ballistics-engine/releases`), download the appropriate archive, extract it, and copy the `ballistics` binary to a directory on your `PATH` (e.g., `/usr/local/bin`).

### 2.1.2 Linux

On Linux, the Cargo-based installation is identical:

Listing 2.3: Installing on Linux via Cargo.

```
# Install Rust if needed
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source "$HOME/.cargo/env"

# Install ballistics-engine
cargo install ballistics-engine
```

**Tip**

The Linux pre-built binaries are statically linked against `musl`, meaning they have no runtime dependencies on shared libraries. They will run on any Linux distribution regardless of glibc version—ideal for deployment on servers or embedded systems.

For Debian/Ubuntu-based distributions, ensure that the necessary build tools are present before building from source:

Listing 2.4: Build prerequisites on Debian/Ubuntu.

```
sudo apt-get update
sudo apt-get install build-essential pkg-config libssl-dev
```

On Fedora/RHEL-based distributions:

Listing 2.5: Build prerequisites on Fedora/RHEL.

```
sudo dnf groupinstall "Development Tools"
sudo dnf install openssl-devel
```

On Arch Linux:

Listing 2.6: Build prerequisites on Arch Linux.

```
sudo pacman -S base-devel openssl
```

### 2.1.3 Windows

On Windows, install the Rust toolchain via `rustup-init.exe`, available at https://rustup.rs. The installer will prompt you to install the Visual C++ Build Tools if they are not already present.

Once Rust is installed, open PowerShell or Command Prompt and run:

Listing 2.7: Installing on Windows.

```
cargo install ballistics-engine
```

The binary is placed in `%USERPROFILE%\.cargo\bin\ballistics.exe`. Windows adds this directory to the `PATH` during `rustup` installation.

**Warning**

On Windows, always use `ballistics.exe` (not just `ballistics`) when running from the command line. PowerShell and CMD may not find extensionless binaries automatically. Alternatively, if you use Windows Subsystem for Linux (WSL), you can follow the Linux installation instructions instead.

### 2.1.4 FreeBSD, OpenBSD, and NetBSD

The BSDs are fully supported through Cargo:

Listing 2.8: Installing on FreeBSD.

```
# FreeBSD: install Rust via pkg
pkg install rust

# Then install ballistics-engine
cargo install ballistics-engine
```

For OpenBSD:

Listing 2.9: Installing on OpenBSD.

```
# OpenBSD: install Rust via pkg_add
pkg_add rust

cargo install ballistics-engine
```

For NetBSD:

Listing 2.10: Installing on NetBSD.

```
# NetBSD: install Rust via pkgin
pkgin install rust

cargo install ballistics-engine
```

**Note**

Python wheels are available for FreeBSD but have limited support on OpenBSD and NetBSD. If you need Python bindings on the BSDs, building from source is recommended. See Chapter 27 for details.

## 2.2 Building from Source with Cargo

Building from source gives you access to the latest development code and allows you to enable or disable feature flags. It also lets you inspect and modify the engine—essential if you are reading Part IX on architecture and internals.

### 2.2.1 Cloning and Building

Listing 2.11: Building from source.

```
git clone https://github.com/ajokela/ballistics-engine.git
cd ballistics-engine
cargo build --release
```

The optimized binary is produced at `target/release/ballistics`. The release profile in `Cargo.toml` enables level-3 optimization, link-time optimization (LTO), and single-codegen-unit compilation for maximum performance:

Listing 2.12: Release profile from `Cargo.toml`.

```
[profile.release]
opt-level = 3
lto = true
codegen-units = 1
```

The combination of `opt-level = 3` and `lto = true` produces the fastest possible binary at the cost of slower compile times. LTO enables cross-module inlining, which is particularly beneficial for the tight inner loops of the trajectory integrator. Setting `codegen-units = 1` ensures that the entire crate is compiled as a single unit, maximizing optimization opportunities.

### 2.2.2 Feature Flags

`ballistics-engine` uses Cargo feature flags to control optional functionality. The following features are defined:

| Feature | Default | Description |
|---|---|---|
| `online` | Yes | HTTP client for the online API (`--online` flag). Uses the `ureq` crate for HTTP requests. |
| `pdf` | Yes | PDF dope card generation. Uses the `printpdf` crate for typesetting. |
| `jemalloc` | No | Use the `jemalloc` memory allocator (Linux/macOS only). |
| `mimalloc` | No | Use the `mimalloc` memory allocator. |

To build without network capabilities (no `--online` flag available):

Listing 2.13: Building without any default features.

```
cargo build --release --no-default-features
```

To build with only local capabilities (PDF support but no online):

Listing 2.14: Building with PDF but without online.

```
cargo build --release --no-default-features --features pdf
```

To build with the `jemalloc` allocator for potentially improved performance on Linux (see Chapter 28 for benchmarks):

Listing 2.15: Building with jemalloc.

```
cargo build --release --features jemalloc
```

**Tip**

For most users, the default features are correct. Disable `online` only if you want a binary with zero network capability—useful for air-gapped or security-sensitive environments. The `jemalloc` and `mimalloc` features are useful for benchmarking and may offer modest performance gains for Monte Carlo workloads; see Chapter 28 for details.

### 2.2.3 Running the Test Suite

After building, verify that everything is working by running the test suite:

Listing 2.16: Running the test suite.

```
cargo test
```

The test suite includes unit tests for drag table interpolation, atmospheric calculations, zeroing convergence, trajectory integration accuracy, and coordinate-system consistency. All tests should pass on any supported platform. If any tests fail, check that you are using Rust 1.70 or later (`rustc --version`).

### 2.2.4 Building for WebAssembly

The engine can also be compiled to WebAssembly for browser-based use (as demonstrated at https://ballistics.sh):

Listing 2.17: Building for WebAssembly.

```
# Install the wasm-pack tool
cargo install wasm-pack

# Build the WASM module
wasm-pack build --target web
```

This produces a `pkg/` directory containing the WASM binary, JavaScript bindings, and TypeScript type definitions. The WASM build automatically disables platform-specific features (jemalloc, mimalloc) and uses the `getrandom` crate's JavaScript backend for random number generation in Monte Carlo simulations. See Chapter 27 for complete WASM integration instructions.

## 2.3 Verifying Your Installation

Once installed, verify the binary by checking its version and running the built-in help:

Listing 2.18: Verifying the installation.

```
ballistics --version
```

You should see output similar to:

Listing 2.19: Expected version output.

```
ballistics 0.32.0
```

To see the full list of subcommands and global options:

Listing 2.20: Viewing available commands.

```
ballistics --help
```

The primary subcommands are:

| Subcommand | Purpose |
|---|---|
| `trajectory` | Compute a single trajectory |
| `zero` | Calculate the zero angle for a target distance |
| `monte-carlo` | Run Monte Carlo statistical simulation |
| `estimate-bc` | Estimate BC from observed trajectory data |
| `true-velocity` | Calculate effective muzzle velocity from field data |
| `come-ups` | Generate an elevation adjustment (come-up) table |
| `wind-card` | Generate a wind-drift card |
| `range-table` | Generate a comprehensive range table |
| `mpbr` | Calculate Maximum Point-Blank Range |
| `stability` | Analyze gyroscopic stability |
| `profile` | Manage saved ballistic profiles |

Each subcommand has its own help page. For example:

Listing 2.21: Getting help for the trajectory subcommand.

```
ballistics trajectory --help
```

**Tip**

If you installed via `cargo install` and the `ballistics` command is not found, ensure `~/.cargo/bin` is on your `PATH`. Add the following to your shell's configuration file (`~/.bashrc`, `~/.zshrc`, etc.):

```
export PATH="$HOME/.cargo/bin:$PATH"
```

You can also generate shell completions for tab-completion of subcommands and flags:

Listing 2.22: Generating shell completions for Zsh.

```
# For Bash
ballistics completions bash > ~/.local/share/bash-completion/completions/ballistics

# For Zsh
ballistics completions zsh > ~/.zfunc/_ballistics

# For Fish
ballistics completions fish > ~/.config/fish/completions/ballistics.fish

# For PowerShell
ballistics completions powershell > ballistics.ps1
```

## 2.4 Your First Trajectory: .308 Win at 1,000 Yards

Let's compute a real trajectory. We will model one of the most iconic cartridge-and-bullet combinations in precision shooting: the .308 Winchester with a 168-grain Sierra MatchKing bullet.

### 2.4.1 The Load

| Parameter | Value |
|---|---|
| Cartridge | .308 Winchester |
| Bullet | 168 gr Sierra MatchKing BTHP |
| Muzzle velocity | 2700 fps (823 m/s) |
| Ballistic coefficient (G1) | `0.475` |
| Bullet diameter | 0.308 in (7.82 mm) |
| Zero distance | 100 yd (91.4 m) |
| Drag model | G1 |

**SAFETY: Safety Warning**

The muzzle velocity of 2700 fps used in this example is typical for a 24-inch barrel with a mid-range powder charge. Your actual velocity will depend on barrel length, powder charge, primer, case, and ambient temperature. Always chronograph your loads and never exceed published maximum charges from a reputable reloading manual.

### 2.4.2 Running the Command

Open a terminal and type:

Listing 2.23: .308 Win, 168-gr SMK trajectory to 1,000 yards.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000
```

Let's break down each flag:

**`-v 2700`** Muzzle velocity in feet per second. The short flag `-v` is an alias for `--velocity`. (Imperial is the default unit system.)

**`-b 0.475`** Ballistic coefficient, referenced to the G1 drag model (the default). Short for `--bc`. Valid range: 0.001 to 2.0.

**`-m 168`** Bullet mass in grains. Short for `--mass`. Valid range: 0.1 to 2000.

**`-d 0.308`** Bullet diameter in inches. Short for `--diameter`.

**`--auto-zero 100`** Automatically calculate the launch angle that produces zero drop at 100 yards, accounting for a standard sight height above the bore.

**`--max-range 1000`** Compute the trajectory out to 1,000 yards.

**Note**

The `--auto-zero` flag replaces manual angle entry. Internally, `ballistics-engine` uses a binary search on the launch angle (see `calculate_zero_angle_with_conditions()` in `src/cli_api.rs`) to find the angle that places the bullet at sight height at the specified zero distance. Without `--auto-zero`, the default launch angle is $0°$ (horizontal), and the bullet begins falling immediately—useful for studying pure gravitational drop but not for practical shooting calculations.

### 2.4.3 Reading the Results

The default output format is a formatted ASCII table. You will see a summary box at the top:

Listing 2.24: Trajectory results summary (table output).

```
+========================================+
|         TRAJECTORY RESULTS             |
+========================================+
| Max Range:           514.95 yd         |
| Max Height:            1.72 yd         |
| Zero Angle:          0.0717°           |
| Time of Flight:       0.700 s          |
| Impact Velocity:    1805.40 fps        |
| Impact Energy:      1215.69 ft-lb      |
+========================================+
| Stability (SG):        2.00            |
| Status:              assumed           |
+========================================+

Bullet struck ground at 515 yd
```

**Trajectory Result Fields**

**Max Range** The maximum downrange distance computed. Here the run stopped short of the requested 1000 yards because the bullet reached the ground first—see the note below the box.

**Max Height** The highest point of the trajectory arc above the bore line, also called the *maximum ordinate*. This represents how far the bullet rises above the line of sight between the muzzle and the zero distance.

**Zero Angle** The launch angle `--auto-zero` solved for, in degrees. It appears whenever the engine zeroes the rifle for you.

**Time of Flight** Total flight time from muzzle to the last computed range point.

**Impact Velocity** The bullet's remaining speed at the last computed range point.

**Impact Energy** Kinetic energy at the last computed range point, calculated as $E_k = \frac{1}{2}mv^2$ and converted to foot-pounds.

**Stability (SG)** The Miller gyroscopic stability factor. We passed no `--twist-rate`, so the engine assumed one and labeled the **Status** `assumed`—the number is an estimate, not a stability verdict. Chapter 17 covers this in full.

Two things in that summary deserve attention.

The first is the range. We asked for 1000 yards and got 515. A rifle zeroed at only 100 yards is pointed very nearly flat, and from a muzzle five feet above the ground a .308 on that trajectory reaches the dirt a little past a quarter mile. The engine stops integrating at ground impact and says so. If you want the arc continued past the ground plane—for studying pure drop, or for a shot taken from elevation—pass `--ignore-ground-impact`.

The second is how much the bullet has slowed: from 2700 fps at the muzzle to 1805 fps at 515 yards, a loss of a third of its initial velocity in half a kilometer. The kinetic energy falls further still, from roughly 2718 ft-lb to 1216 ft-lb, because energy scales with the *square* of velocity.

### 2.4.4 Adding the Full Trajectory Table

To see the trajectory at regular range intervals, add `--sample-trajectory` and `--sample-interval`:

Listing 2.25: Full trajectory with sampling at every 100 yards.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 \
  --full
```

The `--sample-trajectory` flag enables trajectory data collection at regular distance intervals, and `--sample-interval` sets the spacing (100 yards in this example). The `--full` flag tells the engine to print the detailed point-by-point table alongside the summary.

Underneath the point-by-point dump, the sampled table appears:

Listing 2.26: The sampled trajectory table.

```
Sampled Trajectory (every 109 yd)
+----------+----------+----------+----------+----------+
| Dist(yd) | Drop(in) |Drift(in) | Vel(fps) |  Flags   |
+----------+----------+----------+----------+----------+
|      0.0 |     2.00 |     0.00 |   2700.0 | -        |
|    109.4 |     0.09 |     0.00 |   2492.9 | Apex     |
|    218.7 |     4.87 |     0.00 |   2295.1 | -        |
|    328.1 |    17.57 |     0.00 |   2106.3 | -        |
|    437.4 |    39.67 |     0.00 |   1926.7 | -        |
+----------+----------+----------+----------+----------+
```

Its columns are:

**Dist** Downrange distance in yards.

**Drop** Displacement of the bullet relative to the line of sight, in inches. Positive values here mean the bullet is *below* the aim point.

**Drift** Lateral displacement due to wind (zero when no wind is specified).

**Vel** Remaining velocity in fps.

**Flags** Events the sampler recognized at that station: `Zero` at a line-of-sight crossing, `Apex` at the maximum ordinate, and `MachTransition` where the bullet drops through Mach 1.2 and Mach 1.0. Chapter 19 covers the sampler in detail.

**The Interval Is Metric Internally**

We asked for `--sample-interval 100` and the header reads "every 109 yd". The sampler works in meters, so 100 became 100 m, which is 109.4 yards. Ask for the spacing you want in meters and read the header to confirm what you got.

**Tip**

The distinction between `--full` (show all computed points) and `--sample-trajectory` (show points at regular distance intervals) matters. The raw trajectory data from `--full` alone contains one point per integration step (with the default RK45 adaptive integrator, the step size varies, but there can be thousands of points). Adding `--sample-trajectory` with a `--sample-interval` collapses these into cleanly spaced range increments—much more useful for building a range card.

## 2.5 Understanding the Output

### 2.5.1 Drop and Drift

When auto-zero is enabled, drop is reported relative to the *line of sight* (LOS)—the straight line from the scope through the zero point to the target. At the zero distance, drop is zero by definition. The sign convention in the sampled table is that *positive drop means the bullet is below the line of sight*: the muzzle sits sight-height below the LOS and so reads positive, the value passes through zero at the zero range, and it grows steadily positive again as the bullet falls away downrange. Between a near zero and a far zero the bullet rides above the LOS and drop reads negative.

**Line of Sight (LOS)**

The line of sight is the straight line from the center of the optical sight to the point of aim. It does *not* follow the bullet's curved path. The angular offset between the bore axis and the LOS (the "zero angle") is what makes the bullet's arc cross the LOS at the zero distance.

Wind drift is the lateral displacement caused by wind. It is always zero when no wind flags are specified. Positive drift indicates deflection to the right; negative indicates deflection to the left (for a wind from the right/3 o'clock direction, drift is positive).

### 2.5.2 Angular Corrections: MOA and MIL

When `--auto-zero` is used, `ballistics-engine` reports the zero angle and the corresponding adjustments in both MOA and milliradians. These are the angular corrections a shooter would dial into a scope turret.

**MOA and MIL**

- **MOA (Minute of Angle)**: 1/60 of a degree. At 100 yards, 1 MOA subtends approximately 1.047 inches (often rounded to 1 inch).
- **MIL (milliradian)**: 1/1000 of a radian. At 100 yards, 1 MIL subtends approximately 3.6 inches.

For a detailed treatment of these angular measures, see Section 3.7 in Chapter 3.

### 2.5.3 Adding Environmental Conditions

The trajectory we just computed used default atmospheric conditions: 59 °F (15 °C), 29.92 inHg (1013.25 hPa), 50% humidity, sea level. These are the ICAO Standard Atmosphere conditions (see Chapter 3).

Real shooting rarely happens at sea level on a 59 °F day. Let's add realistic field conditions:

Listing 2.27: Adding environmental conditions.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --temperature 85 --pressure 29.12 --humidity 60 \
  --altitude 4500 \
  --wind-speed 10 --wind-direction 90
```

**`--temperature 85`** Temperature in Fahrenheit (default unit system is imperial).

**`--pressure 29.12`** Barometric pressure in inches of mercury (inHg).

**`--humidity 60`** Relative humidity as a percentage (0–100).

**`--altitude 4500`** Firing-point altitude above sea level in feet.

**`--wind-speed 10`** Wind speed in miles per hour.

**`--wind-direction 90`** Wind direction in degrees: 0° = headwind from downrange, 90° = full crosswind from the right (3 o'clock), 180° = tailwind, 270° = crosswind from the left (9 o'clock).

The trajectory results will now show wind drift in the output, and the drop values will differ from the standard-atmosphere case because thinner air at altitude produces less drag, which results in a flatter trajectory.

**Warning**

The `--pressure` flag expects barometric pressure (station pressure corrected to sea level) in imperial mode. If your weather station reports station pressure, you may need to apply a sea-level correction. Alternatively, use `--altitude` and let the engine compute the standard-atmosphere pressure correction—but this is less accurate than providing measured local pressure.

### 2.5.4 Sight Height and Bore Height

Two height parameters affect trajectory calculations:

**Sight height** The vertical distance between the center of the bore and the center of the optical sight. This is typically 1.5–2.0 inches (38–51 mm) for most rifle setups. Specified with `--sight-height` in inches (imperial) or millimeters (metric). The default is 1.8 inches (approximately 0.05 yards).

**Bore height** The height of the bore above the ground. This affects ground-impact detection: the engine tracks when the bullet descends below ground level and can stop the trajectory at that point. Default: 60 inches (1500 mm), the standing position. Specified with `--bore-height` in inches (imperial) or millimetres (metric)—the same units as `--sight-height` since v0.26.0, which changed them from feet and metres.

Listing 2.28: Custom sight height and bore height for prone shooting.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 --bore-height 24
```

If the trajectory reaches the ground before `--max-range`, computation stops. To override this behavior and force the trajectory to continue to the maximum range regardless of ground impact, use `--ignore-ground-impact`:

Listing 2.29: Ignoring ground impact.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 2000 \
  --ignore-ground-impact
```

## 2.6 Output Formats: Table, JSON, CSV

`ballistics-engine` supports three output formats, selected with the `--o` flag (or its long form, `--output`).

### 2.6.1 Table (Default)

The default format is a human-readable ASCII table suitable for terminal display. It uses box-drawing characters for the summary and aligned columns for the trajectory data.

Listing 2.30: Table output (default).

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 500 -o table
```

Table output is designed for quick visual inspection at the terminal. It is not easily machine-parseable—use JSON or CSV for automation.

### 2.6.2 JSON

JSON output produces a complete, structured representation of the trajectory results. It is the preferred format for scripting, data analysis, and integration with other tools.

Listing 2.31: JSON output.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 500 \
  --full -o json
```

The JSON structure includes a top-level object with summary fields (`max_range`, `max_height`, `time_of_flight`, `impact_velocity`, `impact_energy`) and, when `--full` is specified, a `trajectory` array containing one object per trajectory point with fields for `time`, `x`, `y`, `z`, `velocity`, and `energy`.

A practical use case: piping JSON to `jq` to extract the velocity at each 100-yard increment:

Listing 2.32: Extracting data from JSON output with `jq`.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 \
  --full -o json \
```

```
  | jq '.trajectory[] | {range: .z, velocity: .velocity}'
```

**Tip**

JSON output is particularly useful for comparing trajectories. Run two commands with different parameters, save each to a file, and use your favorite scripting language to diff the results:

```
ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 --full -o json > g1.json

ballistics trajectory -v 2700 -b 0.224 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full -o json > g7.json
```

### 2.6.3 CSV

CSV output is designed for import into spreadsheets and data analysis tools. It produces comma-separated values with a header row indicating the column names and units.

Listing 2.33: CSV output.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 \
  --full -o csv
```

With `--full` and `--sample-trajectory`, the CSV output contains trajectory data at each sampled range point. Without `--full`, only the summary statistics are output:

Listing 2.34: CSV summary example.

```
metric,value,unit
max_range,1000.00,yd
max_height,3.42,yd
time_of_flight,1.8720,s
impact_velocity,1198.35,fps
impact_energy,535.26,ft-lb
```

### 2.6.4 PDF Dope Cards

When built with the `pdf` feature (enabled by default), `ballistics-engine` can generate a printable PDF dope card:

Listing 2.35: Generating a PDF dope card.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 25 \
  --wind-speed 10 --wind-direction 90 \
  --bullet-name "168gr SMK" \
  --powder "Varget 44.0gr" \
  --location-name "Whittington Center" \
  --full -o pdf --output-file dope_card.pdf
```

The PDF includes the trajectory table, wind drift values, metadata (bullet, powder, location), and atmospheric conditions. The `--font-scale` and `--font-preset` flags control text sizing, and `--bold-data` makes data cells easier to read at the range.

## 2.7 Metric Units

All examples so far have used imperial units (the default). To switch to metric, add the `--units metric` flag. When metric is selected, *all* inputs and outputs use metric units:

| Quantity | Imperial (default) | Metric |
|---|---|---|
| Velocity | fps | m/s |
| Mass | grains | grams |
| Distance | yards | meters |
| Diameter | inches | millimeters |
| Temperature | °F | °C |
| Pressure | inHg | hPa (millibars) |
| Wind speed | mph | m/s |
| Altitude | feet | meters |
| Energy | ft-lb | Joules |

Listing 2.36: The same .308 trajectory in metric units.

```
ballistics --units metric trajectory \
  -v 823 -b 0.475 -m 10.9 -d 7.82 \
  --auto-zero 91 --max-range 914
```

**Warning**

When using `--units metric`, *all* numeric inputs must be in metric. A common mistake is to mix imperial velocity (e.g., 2700) with metric mode, which would be interpreted as 2700 m/s—roughly Mach 7.9—and produce nonsensical results.

Note that the `--units` flag is a *global* flag—it can appear before or after the subcommand name. The CLI parser accepts it in either position because it is defined with `global = true` in the argument declarations. Placing it before the subcommand makes the intent clearer:

```
ballistics --units metric trajectory -v 823 ...
```

## 2.8 Working with the G7 Drag Model

The default drag model is G1, which historically dominated the shooting industry. For modern boat-tail bullets, the G7 drag model typically provides a better match to the bullet's actual drag curve, especially at long range.

To use G7, you must provide a G7-referenced ballistic coefficient:

Listing 2.37: Using the G7 drag model.

```
ballistics trajectory \
  -v 2700 -b 0.224 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000
```

**Warning**

G1 and G7 ballistic coefficients for the same bullet are *different numbers*. A 168-grain Sierra MatchKing has a G1 BC of approximately `0.475` and a G7 BC of approximately `0.224`. Using a G1 BC with the G7 drag model (or vice versa) will produce wildly incorrect results. Always ensure the BC matches the drag model.

The available drag models are:

| Model | Best Match For |
|---|---|
| G1 | Flat-base bullets with short ogive noses; the traditional default |
| G7 | Boat-tail match bullets with long ogive noses; modern precision bullets |
| G6 | Flat-base military FMJ bullets |
| G8 | Very streamlined flat-base shapes |

We will explore drag models in depth in Chapter 11.

## 2.9 The Zeroing Workflow

So far we have used `--auto-zero` to handle zeroing automatically. You can also compute the zero angle explicitly using the `zero` subcommand:

Listing 2.38: Explicit zero calculation.

```
ballistics zero \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --target-distance 200
```

This returns the zero angle in degrees, the corresponding MOA and milliradian adjustments, and the maximum ordinate (the highest point of the trajectory arc between the muzzle and the zero distance).

You can then feed this angle back into the `trajectory` command using the `--angle` flag, though in practice `--auto-zero` is more convenient. The `zero` subcommand is most useful when you want to examine the zero solution in isolation or compare zero distances.

Listing 2.39: Comparing 100-yard and 200-yard zeros.

```
# 100-yard zero
ballistics zero -v 2700 -b 0.475 -m 168 -d 0.308 \
  --target-distance 100

# 200-yard zero
ballistics zero -v 2700 -b 0.475 -m 168 -d 0.308 \
  --target-distance 200
```

The 200-yard zero requires a steeper launch angle, which means the bullet rises higher above the line of sight between the muzzle and the zero point—a critical consideration for hunters, who must understand their maximum ordinate to avoid shooting over the backs of close targets. For a complete treatment of zeroing, see Chapter 5.

### 2.9.1 Come-Up and Range Tables

Two additional subcommands produce common range-card formats directly:

Listing 2.40: Generating a come-up (elevation) table.

```
ballistics come-ups \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --zero-distance 100 --end 1000 --step 100
```

The `come-ups` subcommand prints the elevation adjustment needed at each range increment. The `range-table` subcommand adds wind drift, velocity, energy, and time of flight:

Listing 2.41: Generating a comprehensive range table.

```
ballistics range-table \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --zero-distance 100 --end 1000 --step 50 \
  --wind-speed 10 --wind-direction 90
```

These subcommands are convenient shortcuts for common field-use outputs that would otherwise require combining `trajectory` flags.

## Exercises

1. Install `ballistics-engine` on your system using whichever method is appropriate. Run `ballistics --version` to confirm it works.
2. Compute the trajectory for a 6.5 Creedmoor, 140-grain match bullet (G7 BC `0.310`) at 2700 fps, zeroed at 100 yards, out to 1000 yards:

   ```
   ballistics trajectory \
     -v 2700 -b 0.310 -m 140 -d 0.264 \
     --drag-model g7 \
     --auto-zero 100 --max-range 1000
   ```

   Compare the 1,000-yard drop and remaining velocity with the .308 Win example from this chapter. Which cartridge retains more energy?
3. Add a 10 mph crosswind from the right (90°) to the 6.5 Creedmoor trajectory above. How much wind drift does the solver predict at 600 yards?
4. Output the .308 Win trajectory in CSV format and import it into a spreadsheet. Plot drop vs. range and velocity vs. range.

5. Use the `zero` subcommand to compute the zero angle for a 300-yard zero with the .308 Win load. What is the maximum ordinate, and at what range does it occur?

6. Try the metric unit system. Compute the .308 Win trajectory using metric inputs (823 m/s, 10.9 g, 7.82 mm) and verify that the results match the imperial calculation:

```
ballistics --units metric trajectory \
  -v 823 -b 0.475 -m 10.9 -d 7.82 \
  --auto-zero 91 --max-range 914
```

7. Build `ballistics-engine` from source with `-no-default-features`. Verify that `--online` is no longer accepted as a flag:

```
cargo build --release --no-default-features
./target/release/ballistics trajectory --help
```

8. Generate a come-up table for a .338 Lapua Magnum, 300-grain match bullet (G7 BC `0.383`) at 2750 fps, zeroed at 100 yards:

```
ballistics come-ups \
  -v 2750 -b 0.383 -m 300 -d 0.338 \
  --drag-model g7 \
  --zero-distance 100 --end 1500 --step 100
```

   How does the come-up at 1000 yards compare to the .308 Win?

9. **Challenge**: Compute the same .308 Win trajectory at sea level and at 8000 feet altitude (all other conditions equal). By how much does the higher altitude reduce the 1,000-yard drop? Why?

```
# Sea level
ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000

# 8000 feet
ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 --altitude 8000
```

## What's Next

You now have a working installation and have computed your first trajectory. But to *understand* what the solver is doing—and more importantly, to know when to trust its predictions and when to question them—you need a foundation in the physics of exterior ballistics. Chapter 3 provides exactly that: a concise primer on gravity, drag, ballistic coefficients, the standard atmosphere, and the coordinate systems that underpin every computation in this book.

# Chapter 3

# An Exterior Ballistics Primer

In Chapter 2 you ran your first trajectory and watched the numbers scroll past. This chapter explains what those numbers *mean*—and, more importantly, where they come from. We will walk through the essential physics of exterior ballistics: what forces act on a bullet after it exits the muzzle, how those forces are modeled mathematically, and what simplifications `ballistics-engine` makes (and when those simplifications break down).

This is not a physics textbook. We will cover just enough theory to make you a competent user of the solver and a critical consumer of its output. For the full mathematical treatment, see Parts IV–VI.

## 3.1 What Happens After the Muzzle

When the propellant gases push the bullet down the bore and out the muzzle, the bullet enters free flight. At this moment, we can characterize its state completely with two quantities:

1. **Position**: the bullet is at the muzzle, a known height above the ground.
2. **Velocity**: the bullet has a speed (the muzzle velocity) and a direction (determined by the bore axis and the launch angle computed by the zeroing algorithm).

From here, the trajectory is determined by Newton's second law:

$$\mathbf{F} = m\,\mathbf{a} \tag{3.1}$$

where $\mathbf{F}$ is the sum of all forces acting on the bullet, $m$ is the bullet's mass, and $\mathbf{a}$ is the resulting acceleration.

For a point-mass model—the model used by `ballistics-engine` in its default mode—three categories of force matter:

1. **Gravity**: a constant downward acceleration.
2. **Aerodynamic drag**: a velocity-dependent retarding force opposing the bullet's motion through the air.
3. **Wind**: the motion of the air itself, which modifies the aerodynamic drag by changing the bullet's velocity *relative to the air*.

Optional advanced effects—spin drift, Magnus force, Coriolis acceleration, gyroscopic precession—add additional forces that we will cover in Part V. For now, gravity and drag are the two dominant players.

Listing 3.1: A minimal trajectory: gravity and drag only, no wind, no zero.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --angle 0 --max-range 500
```

**Note**

The command above uses `--angle 0` (horizontal launch) and no `--auto-zero`, so the bullet immediately begins falling below the bore line. This is useful for visualizing pure gravitational drop before zeroing introduces the upward launch angle.

## 3.2 Gravity: The Constant Pull

Gravity is the simplest force in exterior ballistics and, for most practical distances, the largest contributor to bullet drop.

Near the Earth's surface, gravity produces a constant downward acceleration of approximately:

$$g = 9.806\,65 \text{ m/s}^2 \quad (32.174 \text{ ft/s}^2) \tag{3.2}$$

This is the *standard* value adopted by the International Committee of Weights and Measures (CGPM, 1901) and is the constant used in `ballistics-engine` (defined as `G_ACCEL_MPS2` in `src/constants.rs`):

Listing 3.2: Gravitational constant from `src/constants.rs`.

```
/// Gravitational acceleration in m/s^2
pub const G_ACCEL_MPS2: f64 = 9.80665;
```

Actual gravitational acceleration varies with latitude and altitude—from about $9.78\ \mathrm{m/s^2}$ at the equator to $9.83\ \mathrm{m/s^2}$ at the poles—but the variation is less than 0.3% and is negligible for trajectory computation at small-arms ranges.

### 3.2.1 Gravitational Drop

In the absence of drag, the vertical drop due to gravity alone is:

$$y_{\text{drop}} = \frac{1}{2}\, g\, t^2 \tag{3.3}$$

where $t$ is the time of flight. For a bullet traveling at 2700 fps (823 m/s) to 1000 yards (914 m):

$$t_{\text{vacuum}} = \frac{914\ \text{m}}{823\ \text{m/s}} \approx 1.11\ \text{s} \tag{3.4}$$

$$y_{\text{vacuum}} = \frac{1}{2}(9.81)(1.11)^2 \approx 6.04\ \text{m} \approx 238\ \text{inches} \tag{3.5}$$

But in reality, drag slows the bullet considerably, extending the time of flight. A .308 Win at 2700 fps takes approximately 1.87 seconds to reach 1000 yards through air (see Listing 2.23), and the actual drop is substantially greater:

$$y_{\text{actual}} = \frac{1}{2}(9.81)(1.87)^2 \approx 17.14\ \text{m} \approx 675\ \text{inches} \tag{3.6}$$

This is nearly three times the vacuum drop. The difference is entirely attributable to the extra time of flight caused by aerodynamic drag.

**Gravitational Drop vs. Trajectory Drop**

*Gravitational drop* is the total downward displacement caused by gravity, measured from the bore line. *Trajectory drop* (what the solver reports when zeroed) is the displacement below the *line of sight*—the straight line from the scope to the aim point. When a rifle is zeroed, the bullet's path crosses the LOS at the zero distance; beyond zero, trajectory drop increases rapidly.

The key insight is that gravity alone does not depend on the bullet's aerodynamic properties; it affects all projectiles equally. What changes between bullets is the *time of flight*, which is controlled by drag. A high-BC bullet retains velocity better, arrives at the target sooner, and therefore has less time for gravity to act.

Listing 3.3: Comparing time of flight: low BC vs. high BC at 1000 yards.

```
# Low BC: 150-gr FMJ, G1 BC 0.400
ballistics trajectory \
  -v 2800 -b 0.400 -m 150 -d 0.308 \
  --auto-zero 100 --max-range 1000

# High BC: 175-gr SMK, G1 BC 0.505
ballistics trajectory \
  -v 2600 -b 0.505 -m 175 -d 0.308 \
  --auto-zero 100 --max-range 1000
```

Even though the 175-grain bullet starts 200 fps slower, its higher BC means it arrives at 1000 yards with a shorter time of flight and therefore less gravitational drop.

### 3.2.2 The Zero Angle

Because gravity pulls the bullet downward from the instant it leaves the muzzle, the bore must be angled slightly *upward* relative to the line of sight for the bullet to arrive at the target at the same height as the aim point. This upward angle is the *zero angle* or *launch angle*.

For a 100-yard zero with a .308 Winchester at 2700 fps, the zero angle is approximately 0.06° (≈ 3.6 MOA). This tiny angle—invisible to the naked eye—is enough to loft the bullet slightly above the line of sight at mid-range, intersect the LOS at the zero distance, and then fall away below it at longer ranges. The entire arc above the LOS between muzzle and zero is called the *maximum ordinate*.

## 3.3 Drag: The Atmosphere Fights Back

Aerodynamic drag is the retarding force that the atmosphere exerts on the bullet as it plows through the air. It is the most complex force in exterior ballistics and the one that most strongly differentiates one bullet from another.

### 3.3.1 The Drag Equation

The aerodynamic drag force on a projectile is:

$$F_D = \frac{1}{2}\,\rho\, v_{\text{rel}}^2\, C_D\, A \tag{3.7}$$

where:

$\rho$ is the air density (kg/m³),

$v_{\mathbf{rel}}$ is the bullet's velocity relative to the air (accounting for wind),

$C_D$ is the drag coefficient (dimensionless), a function of Mach number and bullet shape,

$A$ is the bullet's reference area, typically the cross-sectional area $\frac{\pi d^2}{4}$ based on bullet diameter $d$.

Several important properties emerge from this equation:

1. Drag scales with the *square* of velocity. Doubling the speed quadruples the drag force. This is why high-velocity magnum cartridges lose velocity faster in proportional terms than moderate-velocity rounds.
2. Drag depends on *air density* ($\rho$). Shooting at high altitude, where the air is thinner, reduces drag and flattens the trajectory.
3. Drag depends on the *drag coefficient* $C_D$, which varies with velocity. At transonic speeds (roughly Mach 0.8 to Mach 1.2), $C_D$ can increase dramatically—the so-called *transonic drag rise*.

**Drag Coefficient ($C_D$)**

The drag coefficient is a dimensionless number that encapsulates the aerodynamic efficiency of the bullet's shape at a given Mach number. A lower $C_D$ means less drag and a flatter trajectory. Critically, $C_D$ is *not constant*—it varies with Mach number, meaning it changes as the bullet slows down.

### 3.3.2 Drag as a Function of Mach Number

The drag coefficient is not constant; it varies with the bullet's Mach number $M = v/c$, where $c$ is the local speed of sound:

$$M = \frac{v}{c}, \qquad c = \sqrt{\gamma \, R_{\text{air}} \, T} \tag{3.8}$$

where $\gamma = 1.4$ is the ratio of specific heats for air, $R_{\text{air}} = 287.05$ J/(kg·K) is the specific gas constant for dry air, and $T$ is the absolute temperature in Kelvin. At standard sea-level conditions (15 °C, 288.15 K):

$$c_0 = \sqrt{1.4 \times 287.05 \times 288.15} = 340.29 \text{ m/s} \quad (1{,}116.8 \text{ ft/s}) \tag{3.9}$$

This value is defined as `SPEED_OF_SOUND_MPS` in `src/constants.rs`:

Listing 3.4: Speed of sound constant from `src/constants.rs`.

```
/// Speed of sound at sea level, standard atmospheric conditions
/// Value: 340.29 m/s (1116.8 ft/s)
```

```
/// Conditions: 15 degC (59 degF), 1013.25 hPa, dry air
pub const SPEED_OF_SOUND_MPS: f64 = 340.29;
```

Our .308 Winchester load (2700 fps = 823 m/s) begins its flight at Mach 2.42. The behavior of $C_D$ as a function of $M$ divides into three distinct regimes:

**Subsonic ($M < 0.8$):** $C_D$ is relatively low and slowly varying. The air flows smoothly around the bullet, and drag is dominated by skin friction and base drag. For the G1 reference, $C_D \approx 0.27$ in this regime; for G7, $C_D \approx 0.12$.

**Transonic ($0.8 \leq M \leq 1.2$):** $C_D$ rises sharply as shock waves form on the bullet's surface. The G1 reference jumps from $C_D \approx 0.29$ at Mach 0.8 to $C_D \approx 0.48$ at Mach 1.0—a 65% increase. The G7 reference shows a similar but less extreme rise (from 0.12 to 0.38). The transonic regime is where most stability problems occur and where the drag model matters most.

**Supersonic ($M > 1.2$):** $C_D$ decreases gradually as Mach number increases. Wave drag from the bow shock dominates, but the bullet's aerodynamic behavior is relatively predictable.

Listing 3.5: Observing the transonic transition in a trajectory.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1200 \
  --sample-trajectory --sample-interval 50 \
  --full -o csv
```

Export this trajectory to CSV and plot velocity vs. range. You will see the velocity curve steepen as the bullet enters the transonic regime around 800–1000 yards. Enabling pitch damping analysis shows the Mach number where stability may be compromised:

Listing 3.6: Enabling pitch damping for transonic stability analysis.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1200 \
  --enable-pitch-damping
```

We will explore transonic modeling in detail in Chapter 17.

### 3.3.3 Standard Drag Functions

Rather than measuring the complete drag curve for every bullet ever manufactured, ballisticians define *standard drag functions*: reference $C_D$ vs. $M$ curves for idealized projectile shapes. Real

bullets are then characterized by how efficiently they "use" the atmosphere compared to the standard projectile—this ratio is the ballistic coefficient.

`ballistics-engine` implements the following standard drag functions, each defined as a table of (Mach, $C_D$) pairs in `src/drag.rs`:

| Model | Standard Projectile Shape |
|---|---|
| G1 | Flat-base with 2-caliber ogive nose—the traditional "standard" projectile. Highest $C_D$ of the common models. |
| G6 | Flat-base with 6-caliber secant ogive—typical of military FMJ bullets. |
| G7 | Long boat-tail with tangent ogive—best match for modern match-grade boat-tail bullets. |
| G8 | Flat-base with 10-caliber secant ogive—a very streamlined flat-base shape. |

The complete set of supported models is defined in `src/drag_model.rs`:

Listing 3.7: The `DragModel` enum from `src/drag_model.rs`.

```
pub enum DragModel {
    G1, G2, G5, G6, G7, G8, GI, GS,
}
```

The G2, G5, GI, and GS models are included for completeness: G2 is a historical Aberdeen boat-tail shape, G5 is a short-nose boat-tail, GI is the Ingalls drag function (similar to G1 with different coefficients), and GS is a sphere model for round projectiles like buckshot.

The drag tables are loaded at program startup using Rust's `LazyLock` mechanism. The engine first attempts to load high-resolution tables from NumPy binary files or CSV files in a `drag_tables/` directory; if those are not found, it falls back to hardcoded tables embedded in the source code.

Interpolation between table entries uses Catmull–Rom cubic spline interpolation when four surrounding points are available, with linear interpolation at the table edges. The `DragTable::interpolate()` method in `src/drag.rs` handles both cases:

Listing 3.8: Drag coefficient lookup in `src/drag.rs`.

```
/// Get drag coefficient for given Mach number and drag model
pub fn get_drag_coefficient(mach: f64, drag_model: &DragModel) -> f64 {
    match drag_model {
        DragModel::G1 => G1_DRAG_TABLE.interpolate(mach),
        DragModel::G6 => G6_DRAG_TABLE.interpolate(mach),
```

```
        DragModel::G7 => G7_DRAG_TABLE.interpolate(mach),
        DragModel::G8 => G8_DRAG_TABLE.interpolate(mach),
        _ => G1_DRAG_TABLE.interpolate(mach),
    }
}
```

For a visual comparison, export the same bullet's trajectory under both drag models to CSV:

Listing 3.9: Comparing G1 and G7 drag models for the same bullet.

```
# Using G1 BC
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --drag-model g1 \
  --auto-zero 100 --max-range 1000 -o csv --full \
  --sample-trajectory --sample-interval 100 > g1.csv

# Using G7 BC
ballistics trajectory \
  -v 2700 -b 0.224 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 -o csv --full \
  --sample-trajectory --sample-interval 100 > g7.csv
```

The G7 model will typically produce a slightly different trajectory at extreme range because its drag curve better matches the actual bullet behavior through the transonic transition. For a detailed treatment of drag models and when to prefer each one, see Chapter 11.

## 3.4 Ballistic Coefficient: The Bullet's Aerodynamic Fingerprint

The *ballistic coefficient* (BC) is the single most important number characterizing a bullet's aerodynamic performance. It appears in every trajectory command and fundamentally determines how quickly the bullet decelerates.

### 3.4.1 Definition

The ballistic coefficient is defined as the ratio of a bullet's ability to overcome air resistance compared to a standard reference projectile:

$$\text{BC} = \frac{m/d^2}{C_D/C_{D,\text{std}}} = \frac{m}{d^2 \cdot i} \tag{3.10}$$

where $m$ is the bullet mass, $d$ is the bullet diameter, $C_D$ is the bullet's actual drag coefficient, $C_{D,\text{std}}$ is the standard reference projectile's drag coefficient (at the same Mach number), and $i = C_D/C_{D,\text{std}}$ is the *form factor*.

In practical terms:

- A **higher BC** means the bullet is more aerodynamically efficient: it retains velocity better, drops less, and deflects less in the wind.
- A **lower BC** means the bullet decelerates faster and is more affected by environmental conditions.

**Form Factor ($i$)**

The form factor expresses how closely a real bullet's drag matches the standard drag function. A form factor of 1.0 means the bullet has exactly the same drag as the standard projectile. Most modern match bullets have G7 form factors between 0.9 and 1.1, meaning they closely match the G7 standard shape—which is why G7 BCs tend to be more consistent across velocity ranges than G1 BCs.

**Sectional Density (SD)**

The quantity $m/d^2$ in the BC formula is closely related to *sectional density* (SD), a measure of how heavy a bullet is for its caliber. Higher sectional density generally correlates with higher BC, all else being equal, because more mass per unit area means the bullet carries more momentum relative to the drag force acting on it.

### 3.4.2 G1 BC vs. G7 BC

The same physical bullet has different BC values depending on which standard drag function is used as the reference:

| Bullet | Weight | G1 BC | G7 BC |
|---|---|---|---|
| .223 FMJ | 55 gr | 0.250 | — |
| .223 Match | 77 gr | 0.362 | 0.182 |
| .308 Match (SMK) | 168 gr | 0.475 | 0.224 |
| .308 Match (SMK) | 175 gr | 0.505 | 0.253 |
| .308 Hunting | 180 gr | 0.480 | — |
| 6.5mm Match | 140 gr | 0.620 | 0.310 |
| .338 Match | 300 gr | 0.768 | 0.383 |
| .50 BMG Match | 750 gr | 1.050 | 0.520 |

**Warning**

G1 and G7 BCs are *not interchangeable*. A G1 BC is always numerically larger than the G7 BC for the same bullet. Using the wrong BC for the selected drag model is one of the most common sources of trajectory error. When in doubt, check the bullet manufacturer's data and note which drag model the BC references.

Listing 3.10: Comparing G1 and G7 for the same 168-gr SMK bullet.

```
# G1 BC for 168gr SMK
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000

# G7 BC for 168gr SMK
ballistics trajectory \
  -v 2700 -b 0.224 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000
```

These two commands model the same physical bullet but reference its drag to different standard projectiles. The results should be nearly identical at moderate ranges. At extreme range (beyond 800 yards), the G7 solution is generally more accurate for boat-tail bullets because the G7 form factor remains more constant as the bullet decelerates through the transonic regime.

### 3.4.3 How BC Affects the Trajectory

A higher BC means the bullet retains velocity better, which produces four cascading benefits:

1. **Less drop** at a given range (the bullet arrives sooner and gravity has less time to act).
2. **Less wind drift** (the bullet spends less time exposed to the crosswind).
3. **More retained energy** at the target (higher impact velocity means higher kinetic energy).
4. **Shorter time of flight** (less aerodynamic deceleration means the bullet covers the distance faster).

Let's demonstrate quantitatively by comparing two .30-caliber bullets:

Listing 3.11: Comparing a 150-gr flat-base (low BC) vs. 175-gr match (high BC).

```
# 150gr flat-base hunting bullet, G1 BC 0.370
ballistics trajectory \
  -v 2700 -b 0.370 -m 150 -d 0.308 \
```

```
  --auto-zero 100 --max-range 800 \
  --wind-speed 10 --wind-direction 90

# 175gr match boat-tail, G1 BC 0.505
ballistics trajectory \
  -v 2700 -b 0.505 -m 175 -d 0.308 \
  --auto-zero 100 --max-range 800 \
  --wind-speed 10 --wind-direction 90
```

The 175-grain bullet will show significantly less drop and dramatically less wind drift at 800 yards, despite being heavier. Its higher BC more than compensates for the higher mass.

### 3.4.4 Why BC Varies with Velocity

In theory, BC is defined at a single velocity (or Mach number). In practice, because no real bullet perfectly matches any standard drag function, the effective BC changes as the bullet decelerates. This effect is most pronounced for G1 BCs of boat-tail bullets, which are a poor shape match for the flat-base G1 standard.

Many manufacturers publish a single "average" BC, but this average is only accurate over a limited velocity range. Some publish multi-step BCs: one value for velocities above 2500 fps, another for 1800–2500 fps, and a third for below 1800 fps.

`ballistics-engine` addresses this with *BC segmentation*: velocity-dependent BC values that change as the bullet slows. Enable this with `--use-bc-segments`:

Listing 3.12: Enabling BC segmentation for velocity-dependent BC.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --use-bc-segments \
  --auto-zero 100 --max-range 1000
```

**Tip**

BC is the most common source of error in trajectory predictions. Manufacturer-published BCs are often measured under ideal conditions and may not reflect real-world performance. Doppler-derived drag data, available for many popular bullets, provides a more accurate characterization. The `estimate-bc` subcommand (see Chapter 7) lets you derive a field-calibrated BC from observed impacts.

For an in-depth treatment of velocity-dependent BC modeling, BC5D correction tables, and BC truing, see Chapter 12 and Chapter 13.

## 3.5 The Standard Atmosphere (ICAO)

Drag depends on air density, which depends on temperature, pressure, and humidity—all of which change with altitude and weather. To provide a common reference, `ballistics-engine` implements the *ICAO Standard Atmosphere* (International Civil Aviation Organization), defined in ISO 2533.

### 3.5.1 Sea-Level Reference Conditions

The standard atmosphere defines the following conditions at mean sea level:

| Quantity | Value | Unit |
|---|---|---|
| Temperature | 288.15 | K (15 °C / 59 °F) |
| Pressure | 101 325 | Pa (1013.25 hPa / 29.92 inHg) |
| Air density | 1.225 | kg/m³ |
| Speed of sound | 340.29 | m/s (1116.8 ft/s) |
| Lapse rate | −6.5 | K/km |

These values are the defaults used by `ballistics-engine` when no atmospheric flags are specified. They are defined in `src/atmosphere.rs` as part of the ICAO layer data:

Listing 3.13: ICAO constants from `src/atmosphere.rs`.

```
const G_ACCEL_MPS2: f64 = 9.80665;
const R_AIR: f64 = 287.0531;     // Specific gas constant for dry air
const GAMMA: f64 = 1.4;          // Heat capacity ratio for air
const R_DRY: f64 = 287.05;       // Gas constant for dry air
const R_VAPOR: f64 = 461.495;    // Gas constant for water vapor
```

### 3.5.2 Atmospheric Layers

The ICAO model divides the atmosphere into layers, each with a defined temperature lapse rate (the rate at which temperature changes with altitude). The implementation in `src/atmosphere.rs` models seven layers up to 84 km:

| Layer | Base Alt. | Base Temp. | Lapse Rate |
|---|---|---|---|
| Troposphere | 0 km | 288.15 K | −6.5 K/km |
| Tropopause | 11 km | 216.65 K | 0 (isothermal) |
| Stratosphere 1 | 20 km | 216.65 K | +1.0 K/km |
| Stratosphere 2 | 32 km | 228.65 K | +2.8 K/km |
| Stratopause | 47 km | 270.65 K | 0 (isothermal) |
| Mesosphere 1 | 51 km | 270.65 K | −2.8 K/km |
| Mesosphere 2 | 71 km | 214.65 K | −2.0 K/km |

For small-arms ballistics, only the troposphere matters: all practical shooting occurs below 11 km (about 36,000 ft). In the troposphere, temperature decreases at 6.5 °C per 1,000 m of altitude gain (approximately 3.6 °F per 1,000 ft).

### 3.5.3 Pressure and Density at Altitude

Within a layer that has a non-zero lapse rate, pressure decreases with altitude according to the *barometric formula*:

$$P = P_b \left( \frac{T}{T_b} \right)^{-g_0/(L \cdot R_{\text{air}})} \tag{3.11}$$

where $P_b$ and $T_b$ are the base pressure and temperature of the layer, $T = T_b + L \cdot (h - h_b)$ is the temperature at altitude $h$, $L$ is the lapse rate, $g_0 = 9.806\,65$ m/s$^2$ is standard gravitational acceleration, and $R_{\text{air}} = 287.0531$ J/(kg·K) is the specific gas constant for dry air.

For an isothermal layer (lapse rate = 0), the formula simplifies to exponential decay:

$$P = P_b \exp\left( \frac{-g_0 \, (h - h_b)}{R_{\text{air}} \, T_b} \right) \tag{3.12}$$

Air density is then calculated from the ideal gas law:

$$\rho = \frac{P}{R_{\text{air}} \, T} \tag{3.13}$$

These equations are implemented in the function `calculate_icao_standard_atmosphere()` in `src/atmosphere.rs`. As a rough guide, air density decreases by approximately 3% per 1,000 ft of altitude gain in the troposphere—so at 5000 ft, density is about 85% of the sea-level value.

### 3.5.4 Overriding Standard Conditions

The standard atmosphere is a *model*—it represents average conditions, not the conditions at your firing point. For accurate predictions, always override the defaults with measured local values when possible:

Listing 3.14: Measured atmospheric conditions at a mountain range.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --temperature 42 --pressure 25.06 --humidity 30 \
  --altitude 7500
```

When you provide `--temperature` and `--pressure`, the engine uses your values directly. The `--altitude` parameter determines local atmospheric properties only if temperature or pressure are *not* explicitly provided.

### 3.5.5 Humidity Effects

Counter-intuitively, humid air is *less dense* than dry air at the same temperature and pressure, because water vapor (molecular weight ∼18) is lighter than the nitrogen (∼28) and oxygen (∼32) it displaces. Less dense air means less drag and a slightly flatter trajectory.

`ballistics-engine` computes the saturation vapor pressure using the Arden Buck equation and combines dry-air and vapor partial pressures for the density calculation:

$$\rho = \frac{P_{\text{dry}}}{R_{\text{dry}}\, T} + \frac{P_{\text{vapor}}}{R_{\text{vapor}}\, T} \tag{3.14}$$

where $R_{\text{dry}} = 287.05$ J/(kg·K) and $R_{\text{vapor}} = 461.495$ J/(kg·K).

The speed of sound also increases slightly in humid air. The engine applies a humidity correction based on the mole fraction of water vapor in the `calculate_atmosphere()` function.

Listing 3.15: Comparing dry and humid conditions.

```
# Dry air (0% humidity)
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --humidity 0

# Very humid air (90% humidity)
```

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --humidity 90
```

The difference between 0% and 90% humidity at sea level is typically small—on the order of 1–2% in air density—but it is not zero, and at extreme range it can contribute a measurable difference in drop.

**Tip**

In practical shooting, altitude and temperature have a much larger effect on air density than humidity. If you can measure only one environmental parameter, measure temperature. If you can measure two, add altitude (or barometric pressure). Humidity is the least important of the three—but `ballistics-engine` includes it for completeness.

For the full treatment of atmospheric modeling, including the Arden Buck equation for saturation vapor pressure, the CIPM air-density formula, and the IAPWS-IF97 formulation used in the enhanced density calculation, see Chapter 9.

## 3.6 Coordinate System Conventions

`ballistics-engine` uses a right-handed Cartesian coordinate system with the following axis definitions:

| Axis | Direction |
|---|---|
| $X$ | Lateral (left/right, perpendicular to the line of fire) |
| $Y$ | Vertical (up/down) |
| $Z$ | Downrange (the primary direction of fire) |

**Coordinate System**

The origin is at the muzzle. The $Z$-axis points downrange—this is the axis along which "range" is measured. The $Y$-axis points upward; bullet drop is a decrease in $Y$. The $X$-axis points to the right of the shooter; positive $X$ values indicate rightward deflection (e.g., from a right-to-left crosswind or right-hand spin drift).

This convention is consistent throughout the codebase. In the `TrajectoryPoint` structure (`src/cli_api.rs`), each point stores a 3-D position vector with components `position.x` (lateral), `position.y` (vertical), and `position.z` (downrange).

### 3.6.1 Wind Direction Convention

Wind is expressed in this coordinate system as:

$$\mathbf{w} = \begin{pmatrix} w \sin\theta \\ 0 \\ w \cos\theta \end{pmatrix} \tag{3.15}$$

where $w$ is the wind speed and $\theta$ is the wind direction angle. The `--wind-direction` flag uses the following convention:

| Degrees | Meaning |
|---|---|
| 0° | Headwind (blowing from the target toward the shooter) |
| 90° | Crosswind from 3 o'clock (right to left in the bullet's frame) |
| 180° | Tailwind (blowing from the shooter toward the target) |
| 270° | Crosswind from 9 o'clock (left to right) |

A headwind (0°) increases the relative airspeed and therefore drag, slightly increasing drop. A tailwind (180°) decreases relative airspeed and slightly decreases drop. Only the crosswind component causes lateral drift.

Listing 3.16: Headwind vs. tailwind vs. crosswind.

```
# Headwind (0 degrees): slows the bullet more
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 15 --wind-direction 0

# Full crosswind from right (90 degrees): causes wind drift
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 15 --wind-direction 90

# Tailwind (180 degrees): slightly flattens trajectory
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 15 --wind-direction 180
```

### 3.6.2 Relating Coordinates to Shooter Language

| Physical Quantity | Axis | Shooter Term |
|---|---|---|
| Wind drift (crosswind deflection) | $X$ | "Windage" |
| Bullet drop / rise | $Y$ | "Elevation" / "Drop" |
| Downrange distance | $Z$ | "Range" |

When the engine reports a trajectory point with coordinates $(x, y, z)$, $z$ is the range in meters (or yards, in imperial), $y$ is the height above or below the bore line, and $x$ is the lateral displacement. After zeroing corrections are applied, $y$ at the zero distance is approximately equal to the sight height (because the bullet crosses the line of sight at that distance).

## 3.7 Units: MOA, MIL, Inches, and Centimeters

Shooters express angular corrections in two primary systems: *Minutes of Angle* (MOA) and *milliradians* (MIL, often written "mil" or "mrad"). Understanding the relationship between these angular measures and linear distances is fundamental to using trajectory data.

### 3.7.1 Minutes of Angle (MOA)

One Minute of Angle is $1/60$ of one degree:

$$1 \text{ MOA} = \frac{1}{60} \text{ degree} = \frac{\pi}{10800} \text{ radians} \approx 0.000290888 \text{ rad} \tag{3.16}$$

At a given distance $R$, 1 MOA subtends a linear distance of:

$$s = R \times \tan(1 \text{ MOA}) \approx R \times 0.000290888 \tag{3.17}$$

Practical approximations:

- At 100 yards: $1$ MOA $\approx 1.047$ inches (often rounded to 1 inch for field use).
- At 100 meters: $1$ MOA $\approx 2.908$ cm (approximately 29.1 mm).

Most American-made scope turrets click in $\frac{1}{4}$ MOA increments, so each click moves the impact approximately 0.262 inches at 100 yards (or about $\frac{1}{4}$ inch, hence the convenient approximation).

### 3.7.2 Milliradians (MIL)

One milliradian is $1/1000$ of a radian:

$$1 \text{ MIL} = 0.001 \text{ rad} \tag{3.18}$$

At distance $R$:

$$s = R \times \tan(1 \text{ MIL}) \approx R \times 0.001 \tag{3.19}$$

Practical approximations:

- At 100 yards: 1 MIL $\approx$ 3.6 inches.
- At 100 meters: 1 MIL $=$ 10.0 cm (exactly, by definition of the radian).

Most MIL-based scope turrets click in 0.1 MIL (one-tenth MIL) increments, so each click moves the impact approximately 0.36 inches at 100 yards, or 1 cm at 100 meters.

**Note**

The "MIL" used in precision shooting is the true mathematical milliradian (6,283.2 per full circle), *not* the NATO mil (6,400 per circle) sometimes used in artillery. `ballistics-engine` uses true milliradians throughout.

### 3.7.3 Converting Between Linear and Angular Measures

To convert a linear drop or drift (in inches or centimeters) to an angular correction:

$$\text{MOA} = \frac{\text{drop (inches)}}{\text{range (yards)} \times 1.047} \tag{3.20}$$

$$\text{MIL} = \frac{\text{drop (cm)}}{\text{range (meters)} \times 10} = \frac{\text{drop (meters)}}{\text{range (meters)}} \times 1000 \tag{3.21}$$

These conversions are what a shooter uses to translate the trajectory solver's output into turret clicks.

The conversion between MOA and MIL is:

$$1 \text{ MIL} = 3.438 \text{ MOA} \qquad \text{or equivalently} \qquad 1 \text{ MOA} = 0.291 \text{ MIL} \tag{3.22}$$

**Warning**

A common error is mixing MOA and MIL values. If your come-up table says "12.3 MOA at 600 yards" and your scope is marked in MILs, you must convert: $12.3 \text{ MOA} \div 3.438 \approx 3.58$ MIL. Always know which angular system your scope turrets use before dialing corrections.

### 3.7.4 Linear Distance Units

For drop and drift, `ballistics-engine` reports values in the user's selected unit system:

**Imperial:** Drop and drift in *inches* (small values) or *yards* (raw trajectory coordinates). Range in yards.

**Metric:** Drop and drift in *meters* (or millimeters for sampled trajectory output). Range in meters.

The `--units` flag controls both input and output units. Internally, all calculations are performed in SI units (meters, kilograms, seconds), and conversions are applied only at the input/output boundary.

Listing 3.17: Same trajectory, different unit systems.

```
# Imperial: drop in inches, range in yards
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --sample-trajectory --sample-interval 100 --full

# Metric: drop in meters, range in meters
ballistics --units metric trajectory \
  -v 823 -b 0.475 -m 10.9 -d 7.82 \
  --auto-zero 91 --max-range 549 \
  --sample-trajectory --sample-interval 91 --full
```

The numerical results should be equivalent (within rounding) between the two unit systems.

A comprehensive range table showing both angular and linear corrections can be generated with a single command:

Listing 3.18: Complete range table with MOA, MIL, and wind drift.

```
ballistics range-table \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --zero-distance 100 --end 1000 --step 100 \
  --wind-speed 10 --wind-direction 90
```

## 3.8 The Retardation Formula

`ballistics-engine` combines drag coefficient, BC, and air density into a single retardation (deceleration) calculation. The fundamental relationship is:

$$a_{\text{drag}} = \frac{v^2 \cdot C_D \cdot k \cdot \rho_{\text{ratio}}}{\text{BC}} \tag{3.23}$$

where:

$v$ is the velocity,

$C_D$ is the drag coefficient from the drag table at the current Mach number,

$k$ is a dimensional conversion constant defined as `CD_TO_RETARD` $= 2.08551 \times 10^{-4}$ in `src/constants.rs`—the imperial retardation coefficient for a density referenced to ICAO sea level. Through v0.21.x this was the Army Standard Metro value $0.000683 \times 0.30$, which made drag about 1.75% low for every ICAO-referenced BC,

$\rho_{\text{ratio}}$ is the ratio of local air density to standard sea-level density ($1.225$ kg/m$^3$),

**BC** is the ballistic coefficient.

The drag acceleration vector is then applied opposite to the bullet's velocity direction, and gravity is added as a constant downward acceleration:

$$\mathbf{a}_{\text{total}} = -a_{\text{drag}} \frac{\mathbf{v}_{\text{rel}}}{|\mathbf{v}_{\text{rel}}|} + \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} \tag{3.24}$$

This acceleration vector is what the numerical integrator (RK45, RK4, or Euler) uses to advance the trajectory at each time step. The integrators are covered in detail in Chapter 18.

## 3.9 Putting It All Together: The Point-Mass Model

The physics we have covered in this chapter—gravity, aerodynamic drag, the standard atmosphere, and the ballistic coefficient—are the ingredients of the *point-mass* or *3-degree-of-freedom (3-DOF)* trajectory model. This is the default model used by `ballistics-engine` and by most commercial ballistics solvers.

In the point-mass model, the bullet is treated as an infinitely small particle with mass but no shape—all of its aerodynamic behavior is captured by the BC and the drag-model table. The equation of motion is:

$$m\,\ddot{\mathbf{r}} = \underbrace{m\,\mathbf{g}}_{\text{gravity}} + \underbrace{F_D(\mathbf{v}_{\text{rel}})\,\hat{\mathbf{v}}_{\text{rel}}}_{\text{drag}} \tag{3.25}$$

where $\mathbf{r}$ is the position vector, $\mathbf{g}$ is the gravitational acceleration vector, $F_D$ is the drag force magnitude (from Equation (11.1)), and $\hat{\mathbf{v}}_{\text{rel}}$ is the unit vector opposing the bullet's motion relative to the air mass (incorporating wind).

This system of ordinary differential equations is solved numerically. `ballistics-engine` offers three integration methods:

1. **Adaptive RK45 (Dormand–Prince)**: The default. Automatically adjusts the time step to maintain accuracy, using a 5th-order solution with a 4th-order error estimate. Best for most applications.
2. **Fixed-step RK4**: Fourth-order Runge–Kutta with a fixed time step (default: 0.001 s). Slightly faster but less accurate than adaptive RK45. Selected with `--use-rk4-fixed`.
3. **Euler**: First-order integration. Fast but crude. Selected with `--use-euler`. Useful only for comparison and pedagogical purposes.

Listing 3.19: Comparing integration methods.

```
# Default: Adaptive RK45 (most accurate)
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000

# Fixed-step RK4 (faster, slightly less accurate)
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --use-rk4-fixed

# Euler (fast, crude)
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --use-euler
```

Compare the three outputs. At 1000 yards, the Euler and RK45 results may differ by several inches—a stark reminder that numerical method choice matters. The integration methods are covered exhaustively in Chapter 18.

Let us now trace through a complete trajectory to see how gravity, drag, and the atmosphere interact. Consider our familiar .308 Win load: 168-grain Sierra MatchKing, G1 BC 0.475, at 2700 fps, zeroed at 100 yards under standard atmosphere conditions.

At the muzzle ($t = 0$), the bullet is traveling at Mach 2.42 ($2700/1116.8$). At this speed, the G1 drag coefficient is approximately 0.56 (from the supersonic portion of the G1 table), and the drag force produces a deceleration of roughly $400$ m/s$^2$—about 40 $g$'s.

As the bullet decelerates, drag decreases (because drag scales with $v^2$). By 500 yards, the velocity has dropped to approximately 2000 fps (Mach 1.79) and drag deceleration is roughly half what it was at the muzzle.

Between 800 and 900 yards, the bullet enters the transonic regime ($M \approx 1.2$). Here, the drag coefficient rises sharply as shock waves reorganize on the bullet's surface. The bullet decelerates more rapidly, and any asymmetries in the bullet can cause unpredictable deflection—this is the notorious "transonic wall" (see Chapter 17).

By 1000 yards, the bullet is traveling at approximately 1200 fps (Mach 1.07), with a time of flight of about 1.87 seconds. Gravity has had 1.87 seconds to act, producing substantial drop below the line of sight.

Listing 3.20: Full 1,000-yard trajectory with 100-yard sampling.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 \
  --full
```

This command produces a table showing the progressive decay of velocity, the accumulating drop, and the time of flight at each 100-yard increment—a complete picture of the physical phenomena we have discussed in this chapter.

The point-mass model is remarkably accurate for most practical shooting scenarios. Its primary limitation is that it ignores the bullet's spin and shape—effects that become significant at extreme range (>1000 yards) or with unusual projectiles. When these effects matter, you can enable advanced physics flags as covered in Part V:

Listing 3.21: Enabling advanced physics for extreme long-range.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1500 \
  --enable-spin-drift --twist-rate 10 \
  --enable-coriolis --latitude 35.5 --shot-direction 180 \
```

```
  --enable-pitch-damping
```

## Exercises

1. **Vacuum vs. atmosphere.** Calculate the time of flight for a .308 Win (168 gr, 2700 fps) at 500 yards using the solver. Then compute the vacuum time of flight ($t = d/v$) and the vacuum drop ($y = \frac{1}{2}gt^2$). How much additional drop does atmospheric drag cause by increasing the time of flight?

2. **Time of flight and BC.** Compare the time of flight at 1000 yards for two bullets at the same muzzle velocity:

   ```
   # Low BC: 150-gr FMJ, G1 BC 0.400
   ballistics trajectory \
     -v 2800 -b 0.400 -m 150 -d 0.308 \
     --auto-zero 100 --max-range 1000

   # High BC: 175-gr SMK, G1 BC 0.505
   ballistics trajectory \
     -v 2600 -b 0.505 -m 175 -d 0.308 \
     --auto-zero 100 --max-range 1000
   ```

   Even though the 175-grain bullet starts 200 fps slower, which has less drop at 1000 yards? Calculate the gravitational component of drop using $\frac{1}{2}gt^2$ for each.

3. **Altitude effects.** Compute the .308 Win trajectory at sea level, 5000 ft, and 10 000 ft altitude (with temperature adjusted for the mountain):

   ```
   # Sea level, standard conditions
   ballistics trajectory \
     -v 2700 -b 0.475 -m 168 -d 0.308 \
     --auto-zero 100 --max-range 1000

   # 5,000 ft altitude
   ballistics trajectory \
     -v 2700 -b 0.475 -m 168 -d 0.308 \
     --auto-zero 100 --max-range 1000 --altitude 5000

   # 10,000 ft altitude, cold
   ballistics trajectory \
     -v 2700 -b 0.475 -m 168 -d 0.308 \
     --auto-zero 100 --max-range 1000 \
     --altitude 10000 --temperature 30
   ```

By how much does the 1000-yard drop decrease at each altitude? Can you explain why the effect is approximately proportional to the change in air density?

4. **Drag model comparison.** Run the following two commands and compare the remaining velocity at 800 yards:

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --drag-model g1 \
  --auto-zero 100 --max-range 800

ballistics trajectory \
  -v 2700 -b 0.224 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 800
```

   Which prediction do you expect to be more accurate for a boat-tail match bullet, and why?

5. **Wind direction exercise.** Run three trajectories with 15 mph wind at 0° (headwind), 90° (crosswind from right), and 45° (quartering):

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --wind-speed 15 --wind-direction 0

ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --wind-speed 15 --wind-direction 90

ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --wind-speed 15 --wind-direction 45
```

   Observe how the wind affects both drop (through velocity change) and drift (through lateral force). Does the quartering wind produce exactly half the drift of the full crosswind?

6. **BC sensitivity.** Run the .308 Win trajectory with BC values of `0.450`, `0.475`, and `0.500` and compare the 1000-yard drop:

```
ballistics trajectory \
  -v 2700 -b 0.450 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000

ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000

ballistics trajectory \
  -v 2700 -b 0.500 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000
```

How sensitive is the 1000-yard solution to a 5% change in BC? Is the sensitivity symmetric (is the effect of $+5\%$ the same magnitude as $-5\%$)?

7. **Integration method comparison.** Compare the three integration methods at 1500 yards using a .338 Lapua Magnum, 300-gr match bullet (G7 BC `0.383`) at 2750 fps:

```
ballistics trajectory -v 2750 -b 0.383 -m 300 -d 0.338 \
  --drag-model g7 --auto-zero 100 --max-range 1500

ballistics trajectory -v 2750 -b 0.383 -m 300 -d 0.338 \
  --drag-model g7 --auto-zero 100 --max-range 1500 \
  --use-rk4-fixed

ballistics trajectory -v 2750 -b 0.383 -m 300 -d 0.338 \
  --drag-model g7 --auto-zero 100 --max-range 1500 \
  --use-euler
```

How much does the Euler method diverge from RK45? Would this difference matter for a practical shot?

8. **Humidity investigation.** Compare trajectories at 0%, 50%, and 100% humidity at 86 °F. Is the effect larger or smaller than a 1000-foot altitude change?

```
ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 --temperature 86 --humidity 0

ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 --temperature 86 --humidity 100

ballistics trajectory -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 1000 --altitude 1000
```

## What's Next

With the physics foundation in place, Part I is complete. You now understand the physical forces that shape a bullet's trajectory, the atmospheric model that governs drag, the coordinate system in which the solver reports its results, and the angular units used for scope adjustments.

In Part II, we turn to the commands themselves. Chapter 4 takes you deep into the `trajectory` subcommand—its full flag set, its interaction with zeroing, and the advanced physics options that let you push trajectory prediction to its limits. From there, Chapter 5 covers the zeroing algorithm, and Chapter 6 introduces Monte Carlo simulation for answering the question every shooter eventually asks: "How confident am I in this firing solution?"

# Part II

# The Commands

# Chapter 4

# Trajectory Computation

You have a rifle, a load, and a target at 600 yards. The question every shooter asks—*where will this bullet be when it gets there?*—is the question that the `trajectory` command answers. It is the heart of BALLISTICS-ENGINE, and the command you will use more than any other.

In this chapter we will walk through the `trajectory` command from its most basic invocation to its most advanced. We will cover every required input, every optional flag, every column in the output table, and the physics that connects them. By the end, you will be able to build a complete dope card for any rifle and load in your safe.

## 4.1 The `trajectory` Command

Let's start with the minimal invocation. We need to tell the engine what bullet we are shooting and how fast it is going:

Listing 4.1: A first trajectory: .308 Win, 168gr SMK

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308
```

With just those four flags—muzzle velocity, ballistic coefficient, bullet mass, and bullet diameter—the engine computes a full trajectory from the muzzle out to its default maximum range of 1000 yards. It uses the default drag model (G1), standard atmospheric conditions (59°F, 29.92 inHg, sea level), and no wind.

The output is a table with range, velocity, energy, drop, time of flight, and Mach number at regular intervals. We will examine every one of those columns in Section 4.5.

**Imperial Units Are the Default**

When you pass `--velocity 2700`, BALLISTICS-ENGINE assumes feet per second because the default unit system is imperial. Velocity is fps, mass is grains, diameter is inches, distances are yards, temperature is Fahrenheit, and pressure is inches of mercury. Pass `--units metric` to switch everything to SI units (m/s, grams, mm, meters, Celsius, hPa).

The `trajectory` command also accepts a rich set of optional flags that control the atmospheric model, wind, sight geometry, inclination angle, output format, integration method, and advanced physics effects. We will build up to those one at a time.

**Numbers Moved in 0.22.0**

Version 0.22.0 was the product of an adversarial audit of the whole crate, and it corrected 44 confirmed defects across drag, atmosphere, ground and zero geometry, wind, spin physics, and Monte Carlo. **It changes numeric output for most trajectories.** Four of the fixes are large enough to see at any range:

- The ground plane no longer double-counts bore height, so ground-impact range and time of flight are no longer overstated (by roughly 37% and 45% in a typical flat-fire case).
- A transonic drag-rise multiplier that stacked a second rise on top of the one the G1/G7 tables already carry has been removed. Any shot touching Mach 0.85–1.2—and the entire flight of a subsonic load—was flying with too much drag.
- The drag retardation constant is now the ICAO-referenced $2.085\,51 \times 10^{-4}$; it had been the Army Standard Metro value, making drag uniformly about 1.75% low for any ICAO-referenced BC.
- `--shooting-angle` is now actually applied by the solver (see Section 4.9).

If you built reference tables, dope cards, or truing corrections against v0.21.x or earlier, re-derive them. Each of these fixes is described in the chapter that owns it.

## 4.2 Required Inputs

Four pieces of information are non-negotiable. You cannot compute a trajectory without them.

### 4.2.1 Muzzle Velocity (`--velocity`)

Muzzle velocity is the speed of the bullet as it exits the barrel, expressed in feet per second (imperial) or meters per second (metric). It is the single most influential variable in external ballistics—a 50 fps change in muzzle velocity can shift your impact point by several inches at long range.

Listing 4.2: Specifying muzzle velocity

```
ballistics trajectory --velocity 2700 \
```

```
  --bc 0.462 --mass 168 --diameter 0.308
```

Where does this number come from? Ideally, from a chronograph. Published load data from powder manufacturers provides starting points, but every rifle is different. A 24-inch barrel will produce different velocities than a 20-inch barrel with the same load.

**SAFETY: Always Verify Muzzle Velocity**

Never rely solely on published velocity data. Always chronograph your actual load in your actual rifle. Computational predictions are only as accurate as their inputs. A 100 fps error in muzzle velocity translates to roughly 0.5 MIL of error at 1000 yards with a typical .308 Win load.

The `--velocity` flag accepts values from 0 to 6000. If you have trued your velocity against observed drop data, you can apply a correction using `--velocity-adjustment`:

Listing 4.3: Applying a velocity adjustment

```
ballistics trajectory --velocity 2700 \
  --velocity-adjustment -25 \
  --bc 0.462 --mass 168 --diameter 0.308
```

This effectively computes the trajectory at 2675 fps—useful when you know your chronograph reads a bit high, or when you have trued your system using observed drop at range (see Chapter 7).

### 4.2.2 Ballistic Coefficient (`--bc`)

The ballistic coefficient (BC) quantifies how well a bullet overcomes air resistance. A higher BC means less drag and a flatter trajectory. The `--bc` flag accepts values from 0.001 to 2.0:

Listing 4.4: Specifying BC

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308
```

This value is always referenced to a specific drag model (G1 or G7—see Section 4.3). The same bullet will have different G1 and G7 BC values. For our .308 Win, 168gr Sierra MatchKing example:

- G1 BC: `0.462`
- G7 BC: `0.224`

Make sure you match the BC to the correct drag model flag. Mixing a G7 BC with the default G1 drag model will produce wildly incorrect results.

**BC and Drag Model Must Match**

A BC value is meaningless without its corresponding drag model. The G7 BC for a 168gr SMK (`0.224`) is much lower than the G1 BC (`0.462`), but both describe the same bullet. If you use the G7 BC with a G1 drag model, the engine will think your bullet has twice the drag it actually does.

You can also apply a BC adjustment multiplier with `--bc-adjustment`. This is a multiplicative factor, so `--bc-adjustment 0.95` reduces BC by 5%—useful for truing against observed data:

Listing 4.5: Truing BC with an adjustment factor

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --bc-adjustment 0.95 \
  --mass 168 --diameter 0.308
```

### 4.2.3 Bullet Mass (`--mass`)

Bullet mass is specified in grains (imperial) or grams (metric). It affects kinetic energy, time of flight, and how the bullet responds to wind:

Listing 4.6: Specifying bullet mass

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308
```

Internally, the engine converts grains to kilograms (1 grain $=$ 0.00006480 kg) for the physics calculations. The mass is used in the kinetic energy computation:

$$E_k = \frac{1}{2}mv^2 \tag{4.1}$$

where $m$ is bullet mass in kilograms and $v$ is velocity in $\mathrm{m/s}$. This energy is reported in foot-pounds (imperial) or joules (metric) in the output.

### 4.2.4 Bullet Diameter (`--diameter`)

Bullet diameter—commonly called *caliber*—is specified in inches (imperial) or millimeters (metric). It defines the reference area for drag calculations:

$$A = \frac{\pi d^2}{4} \tag{4.2}$$

Common values include 0.224 (.223 Rem), 0.264 (6.5 mm), 0.308 (.308 Win), and 0.338 (.338 Lapua).

Listing 4.7: Common cartridge configurations

```
# .223 Rem, 77gr SMK
ballistics trajectory --velocity 2750 \
  --bc 0.372 --mass 77 --diameter 0.224

# 6.5 Creedmoor, 140gr ELD-M
ballistics trajectory --velocity 2710 \
  --bc 0.610 --mass 140 --diameter 0.264

# .338 Lapua Mag, 300gr Berger Hybrid
ballistics trajectory --velocity 2725 \
  --bc 0.818 --mass 300 --diameter 0.338
```

## 4.3 Choosing a Drag Model: G1 vs. G7 vs. Custom

The drag model defines the *shape* of the drag curve—how the drag coefficient $C_D$ varies with Mach number. The BC then *scales* that curve to match a specific bullet. Think of the drag model as the template and the BC as the adjustment factor.

### 4.3.1 G1: The Traditional Standard

The G1 drag model is based on a flat-base, 2-caliber ogive projectile. It has been the industry standard since the early 20th century, and most published BC data uses G1:

Listing 4.8: Using the G1 drag model (default)

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g1
```

The G1 standard projectile does not closely resemble modern long-range bullets, which typically have boat-tail bases and long, secant-ogive noses. This means that the G1 BC for a modern boat-tail bullet varies with velocity—it is *not* constant across the flight envelope. At supersonic velocities the published G1 BC may be accurate, but as the bullet decelerates into the transonic regime, the G1 model diverges from reality.

### 4.3.2 G7: The Modern Standard for Long-Range

The G7 drag model is based on a boat-tail, long-ogive projectile that much more closely matches the shape of modern match and hunting bullets. Because the G7 reference projectile is a better geometric match, the G7 BC tends to remain more constant across a wider velocity range:

Listing 4.9: Using the G7 drag model

```
ballistics trajectory --velocity 2700 \
  --bc 0.224 --mass 168 --diameter 0.308 \
  --drag-model g7
```

**G1 vs. G7 Standard Projectiles**

**G1**: Flat-base, 2-caliber tangent ogive. $C_D \approx 0.48$ at Mach 1.0. Best for flat-base and round-nose bullets.
**G7**: Boat-tail, 10-caliber secant ogive. $C_D \approx 0.38$ at Mach 1.0. Best for modern boat-tail match bullets and VLD designs.

### 4.3.3 G6 and G8

BALLISTICS-ENGINE also supports the G6 and G8 drag models for specialized applications:

- **G6**: Flat-base with 6-caliber secant ogive. Common for military FMJ projectiles.
- **G8**: Flat-base with 10-caliber secant ogive. A compromise between G1 and G7.

The underlying drag tables for all models are stored as Mach-indexed arrays in `src/drag.rs`. The engine uses Catmull-Rom cubic spline interpolation between data points for smooth $C_D$ values at any Mach number (see Section 18.6).

### 4.3.4 Which Model Should You Use?

**Drag Model Selection Rule of Thumb**

Use G7 for any boat-tail bullet. Use G1 only when you have G1-referenced BC data and are shooting flat-base or round-nose bullets. When in doubt, ask your bullet manufacturer whether they publish G7 BCs. Many modern manufacturers now provide both.

For a deeper treatment of drag models, standard projectile shapes, and the underlying drag tables, see Chapter 11 in Part IV.

## 4.4 Setting the Zero Range

A raw trajectory shows bullet drop relative to the bore line. In practice, you want drop relative to your *line of sight*—the straight line from your scope to the target. This requires a zero: the range at which the bullet's path crosses the line of sight so that drop reads zero.

The `--auto-zero` flag tells the engine to compute the necessary launch angle so that the bullet impacts at zero drop at the specified distance:

Listing 4.10: Zeroing at 200 yards

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000
```

Internally, the engine uses binary search to find the muzzle angle that produces zero vertical deviation at the specified range. The `calculate_zero_angle()` function in `src/cli_api.rs` iterates with a height error tolerance of 0.001 m (approximately 0.04 inches)—more than sufficient for practical zeroing accuracy.

With `--auto-zero` set to 200, the output table will show:

- Positive drop values (bullet above line of sight) at ranges closer than 200 yards
- Zero drop at 200 yards
- Increasingly negative drop values beyond 200 yards

If you already know the launch angle from a previous zeroing calculation, you can specify it directly with `--angle` (in degrees). This overrides `--auto-zero`:

Listing 4.11: Specifying a known launch angle

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --angle 0.1186
```

We cover the zeroing algorithm in depth in Chapter 5. For now, know that `--auto-zero` is the flag you will reach for most often.

## 4.5 Output Columns Explained

The default output of the **`trajectory`** command is a table printed to standard output. Let's generate one and walk through each column:

Listing 4.12: Generating a trajectory table for .308 Win

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g1 --auto-zero 200 \
  --max-range 1000 --wind-speed 10 \
```

```
  --wind-direction 90
```

This produces a table with columns for range, velocity, energy, drop, windage, time of flight, and Mach number. Let's examine each.

### 4.5.1 Range

The downrange distance in yards (imperial) or meters (metric), sampled at regular intervals. By default, the table shows results every 100 yards out to `--max-range`. You can change the sampling interval with `--sample-interval` (see Section 18.6).

### 4.5.2 Velocity

The bullet's remaining velocity at each range, in fps (imperial) or m/s (metric). This is the magnitude of the velocity vector, accounting for drag deceleration. A .308 Win, 168gr SMK at 2700 fps muzzle velocity will typically show about 1950 fps at 500 yards and 1350 fps at 1000 yards.

Velocity matters because it determines both the energy available for terminal performance and the bullet's susceptibility to wind drift.

### 4.5.3 Energy

Kinetic energy at each range, in foot-pounds (imperial) or joules (metric), computed from Equation (4.1). Many hunters use minimum energy thresholds for ethical harvest: 1000 ft-lbs for deer-class game, 1500 ft-lbs for elk-class game. The energy column tells you at what range your load drops below these thresholds.

### 4.5.4 Drop

Vertical displacement relative to the line of sight, in inches (imperial) or centimeters (metric). When `--auto-zero` is set, this column shows how far above or below the point of aim the bullet strikes. Positive values mean the bullet is above the line of sight; negative values mean below.

This is the number you transcribe onto your dope card. A drop value of $-38.4$ inches at 600 yards means you need to dial up approximately $38.4/(3.6 \times 6) = 1.78$ MIL of elevation (or $38.4/(1.047 \times 6) = 6.11$ MOA).

**Drop Conversion: Inches to MIL and MOA**

At range $R$ (in hundreds of yards):

$$\text{MIL} = \frac{\text{drop (inches)}}{R \times 3.6} \tag{4.3}$$

$$\text{MOA} = \frac{\text{drop (inches)}}{R \times 1.047} \tag{4.4}$$

One MIL subtends 3.6 inches at 100 yards (equivalently, 1 m at 1000 m). One MOA subtends approximately 1.047 inches at 100 yards.

### 4.5.5 Windage

Lateral displacement due to wind, in inches (imperial) or centimeters (metric). This column appears when `--wind-speed` is specified. A positive windage value indicates deflection in the direction of the wind (e.g., to the left for a right-to-left crosswind).

We discuss wind modeling in detail in Section 4.7.

### 4.5.6 Time of Flight

The elapsed time from muzzle exit to each range, in seconds. Time of flight matters for several reasons:

- It determines how long wind acts on the bullet, which drives wind drift.
- It tells you how much a moving target will travel during bullet flight (lead calculation).
- It helps diagnose transonic issues—if time of flight to 1000 yards is 1.7 seconds, the bullet spent a long time decelerating.

### 4.5.7 Mach Number

The ratio of the bullet's velocity to the local speed of sound. At standard conditions, the speed of sound is approximately 1116 fps (340 m/s). Mach numbers help you understand where the bullet is in its flight regime:

- **Supersonic** (Mach $> 1.2$): Drag is predictable and well-modeled.
- **Transonic** ($0.8 <$ Mach $< 1.2$): Drag changes rapidly; accuracy can degrade.
- **Subsonic** (Mach $< 0.8$): Lower drag, but bullets may destabilize.

**Transonic Transition**

When the Mach number drops below approximately 1.2, the bullet enters the transonic regime. Shockwave patterns on the projectile change rapidly, and even small asymmetries in the bullet can cause significant dispersion. Keep your bullet supersonic at your maximum intended range for reliable accuracy.

### 4.5.8 Output Formats

The `--output` flag (short form: `-o`) controls how results are printed:

- `--output table` — Human-readable ASCII table (default)
- `--output csv` — Comma-separated values, suitable for import into spreadsheets
- `--output json` — Structured JSON, suitable for programmatic consumption
- `--output pdf` — PDF dope card (requires `--sample-trajectory` and `--output-file`)

Listing 4.13: Exporting a trajectory as CSV

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000 \
  --output csv > trajectory.csv
```

**JSON Output Honors `--units` (0.21.4)**

The table and CSV formatters have always respected `--units`. The JSON formatter did not—it emitted raw SI no matter what you asked for. As of v0.21.4 it converts to the requested unit system and adds an explicit `"units"` key so a consumer can tell. **Scripts that ran `--units imperial` and parsed JSON were receiving metric numbers; they now receive imperial.**

Listing 4.14: Generating a PDF dope card

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000 \
  --sample-trajectory \
  --output pdf --output-file dope_card.pdf \
  --bullet-name "168gr SMK" --wind-speed 10 \
  --wind-direction 90
```

The PDF output generates a formatted dope card with range, drop (in MIL), windage, velocity, and energy columns—ready to laminate and take to the range.

### PDF Cards Got Bigger, and Correct (0.30.0)

The PDF backend moved from printpdf 0.10 to 0.12.4 in v0.30.0. The old version emitted no font resources, no `/Encoding` and no `/ToUnicode`, which left the text to whatever the viewer happened to default to—so a card could render differently on your phone than on the machine that made it, and its text was neither searchable nor selectable. The new backend embeds subset fonts with a ToUnicode map, so a card renders identically everywhere and stays copyable. **The cost is file size**: a sample dope card goes from about 6 kB to about 836 kB, all of it embedded font data.

### The JSON Legend Block (0.25.0)

Trajectory JSON now carries a top-level `legend` object describing its own units and axis conventions, so a consumer does not have to guess which way $x$ points:

```
"legend": {
 "units": { "distance": "yd", "velocity": "fps", "energy": "ft-lb" },
 "axes": {
  "x": "lateral offset from the muzzle's initial aiming direction; positive
    means to the shooter's right ...",
  "y": "height above the ground in the world frame; positive means up ...",
  "z": "downrange distance from the muzzle; zero at the muzzle, always
    increasing."
 }
}
```

Every pre-existing key is unchanged—the block is purely additive.

### Exporting the Projectile's Own $C_D$ (0.30.0)

`--with-drag-coefficient` adds a `drag_coefficient` key to every point of `--full` JSON. This is the *bullet's* $C_D$—the reference-table value scaled by the form factor SD/BC—not the G-model curve, which would be identical for every bullet sharing a drag model and therefore useless to export. A segmented BC shows its band steps here, and a custom drag table passes through unscaled. Only the speeds actually flown are covered, and the $C_D$ is paired with the station speed of sound the document already reports Mach against. Off by default: JSON without the flag is byte-identical.

### 4.5.9 Terminal Charts (`--plot`)

Sometimes you want the shape of the thing, not the numbers. Since v0.25.0, `--plot` draws the trajectory inline in your terminal:

Listing 4.15: Inline terminal charts

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 600 --plot
```

Bare `--plot` uses Unicode braille dots, which give four times the vertical resolution of a character cell. If your terminal or font lacks braille coverage, `--plot ascii` falls back to one `*` per cell:

Listing 4.16: The ASCII renderer

```
+ energy (ft-lb) — y:[1141.89, 2718.96] ---------------------------------+
|*** *                                                                   |
|     ****                                                               |
|         ** **                                                          |
|              *****                                                     |
|                   *****                                                |
|                         *****                                          |
|                              ******                                    |
|                                    ******                              |
|                                          *******                       |
|                                                 ********               |
|                                                         *******        |
|                                                                ********|
+ x:[0.00, 536.62] ------------------------------------------------------+
```

Four panels are drawn: drop, lateral drift, velocity, and energy.

### 4.5.10 Turret Adjustment Units

Not every turret is graduated in mils or true MOA. Since v0.28.0 `--adjustment-unit` takes five values:

| Value | Meaning |
|---|---|
| `mil` | Milliradians. 1 MIL subtends 3.6 inches at 100 yards. |
| `moa` | True minutes of angle. 1 MOA subtends 1.047 inches at 100 yards. |

| Value | Meaning |
|---|---|
| `smoa` | *Shooter's* MOA—exactly 1 inch per 100 yards, the approximation most range talk actually uses. |
| `iphy` | Inches per hundred yards. Numerically identical to SMOA; two names for the same unit because two communities named it independently. |
| `clicks` | Whole turret clicks against a stated graduation. |

`clicks` needs to know how big a click is, so it requires `--elevation-click-value` (or a saved profile's `elevation_click`). The graduation carries its own unit suffix—`0.25moa`, `0.1mil`—and **a bare number is an error**, deliberately: "0.25" is not a click size until you say of what. Displayed clicks are rounded to whole numbers, ties away from zero, because a turret has no fractional detent.

Listing 4.17: A come-up table in whole clicks

```
ballistics come-ups --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 200 --sight-height 1.5 \
  --start 300 --end 700 --step 200 \
  --adjustment-unit clicks --elevation-click-value 0.1mil
```

Column headers carry the unit, so a table cannot be misread as mils when it is not:

Listing 4.18: Come-ups in Shooter's MOA

```
Come-Up Table (zero: 200 yd, SMOA)
+----------+----------+----------+----------+----------+----------+
|Range (yd)|Drop (SMOA)|Come-Up   | Vel (fps)|Energy    | Time (s) |
+----------+----------+----------+----------+----------+----------+
|      300 |     0.001 |    −     |     2318 |     2004 |    0.240 |
|      500 |     6.157 |    6.156 |     1969 |     1446 |    0.521 |
|      700 |    14.255 |    8.098 |     1654 |     1021 |    0.853 |
+----------+----------+----------+----------+----------+----------+
```

**On `trajectory`, the Flag Reaches Only Two Places**

`--adjustment-unit` on `trajectory` affects the PDF dope card (`-o pdf`) and the mover Ring column (`--target-speed`). It does *not* change the plain table, JSON or CSV. Supplying it with neither of those in play used to be silent; since v0.29.0 it warns. Commands whose whole job is dial output—`come-ups`, `wind-card`, `range-table`, `compare`, `lead`—honour it throughout, and `--windage-unit` lets you run mil elevation against MOA windage on those.

### 4.5.11 The Mover Ring (`--target-speed`)

Leading a moving target normally means computing a directional hold, which forces you to assume a crossing angle. The *mover ring* sidesteps that. Give the trajectory a target speed and every point gains a ring radius—target speed multiplied by the time of flight to that point—drawn around your hold:

Listing 4.19: A mover ring for an 8 mph target

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 500 --target-speed 8 --full
```

Listing 4.20: The Ring column

```
+----------+----------+----------+----------+----------+----------+
| Time (s) |  X (yd)  |  Y (yd)  | Vel(fps) |Energy(ft-lb)| Ring(mil)|
+----------+----------+----------+----------+----------+----------+
|    0.000 |     0.00 |     1.67 |  2700.00 |  2718.96 |        - |
|    0.006 |     5.43 |     1.68 |  2689.20 |  2697.25 |     4.35 |
|    0.032 |    28.38 |     1.71 |  2643.78 |  2606.91 |     4.39 |
|    0.062 |    54.56 |     1.74 |  2592.55 |  2506.87 |     4.43 |
|    0.092 |    80.24 |     1.76 |  2542.88 |  2411.72 |     4.48 |
```

Watch the target through the optic and fire the instant it crosses into the ring; no crossing angle is assumed because none is needed. The radius is pure post-processing on a trajectory the engine already solved, so it costs nothing. `--target-speed` takes mph in imperial and m/s in metric, is bounded to $[0, 300]$ (out-of-range values are rejected, not clamped), and defaults to 0, which leaves every output format byte-identical to a run without the flag. CSV carries `ring_mil`; JSON carries `mover_ring_m` and `mover_ring_mil`. The same flag drives the `Lead` column on the PDF dope card.

## 4.6 Step Size and Resolution

The trajectory is computed by numerically integrating the equations of motion—solving the differential equations that govern drag, gravity, and wind forces acting on the bullet at each instant in time. The *step size* controls the granularity of this integration.

### 4.6.1 Integration Methods

By default, BALLISTICS-ENGINE uses the **RK45 adaptive** (Dormand-Prince) method, which automatically adjusts the time step to maintain a specified error tolerance. This provides an excellent balance of accuracy and computational efficiency:

Listing 4.21: Default RK45 adaptive integration

```
# Default behavior --- RK45 adaptive is used automatically
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308
```

You can select alternative integration methods:

Listing 4.22: Selecting integration methods

```
# Fixed-step RK4 (faster, slightly less accurate)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --use-rk4-fixed

# Euler method (fastest, least accurate --- for debugging only)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --use-euler
```

The RK45 adaptive method starts with an initial step of 0.001 seconds and adjusts dynamically. It uses a tolerance of $10^{-6}$ and a safety factor of 0.9 for step-size adjustment, with a maximum step of 0.01 seconds and a minimum of $10^{-6}$ seconds. These parameters are defined in the `solve_rk45()` method of the `TrajectorySolver` in `src/cli_api.rs`.

**When to Use Fixed-Step RK4**

For most work, stick with the default RK45 adaptive integrator. Consider fixed-step RK4 (`--use-rk4-fixed`) only if you need perfectly uniform output spacing or are running thousands of Monte Carlo iterations where the small speed improvement matters.

### 4.6.2 The `--time-step` Flag

The `--time-step` flag sets the integration time step in seconds. For fixed-step methods, this directly controls resolution. For adaptive methods, it sets the initial step:

Listing 4.23: Controlling integration step size

```
# Finer step for maximum accuracy (0.0001 seconds)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --time-step 0.0001

# Coarser step for speed (0.005 seconds)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --time-step 0.005
```

The default of 0.001 seconds produces trajectory points approximately every 2–3 feet of downrange travel at typical rifle velocities, which is more than adequate for all practical applications.

### 4.6.3 Trajectory Sampling

By default, the output table shows values at standard range increments (every 100 yards). If you want the raw, high-resolution trajectory data—every single integration point—use the `--full` flag:

Listing 4.24: Full trajectory output

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --full --output csv > full_trajectory.csv
```

For custom sampling intervals, enable trajectory sampling with `--sample-trajectory` and set the interval with `--sample-interval`:

Listing 4.25: Custom sampling interval

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --sample-trajectory --sample-interval 25
```

This outputs trajectory data every 25 meters, useful for detailed analysis or plotting.

## 4.7 Adding Wind: Speed and Angle

Wind is the long-range shooter's greatest adversary. A 10 mph full-value crosswind will push a .308 Win, 168gr SMK over 30 inches at 600 yards. The `--wind-speed` and `--wind-direction` flags add wind to the trajectory computation.

### 4.7.1 Wind Speed

The `--wind-speed` flag specifies wind speed in mph (imperial) or m/s (metric):

Listing 4.26: Adding a 10 mph wind

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000 \
  --wind-speed 10 --wind-direction 90
```

### 4.7.2 Wind Direction

The `--wind-direction` flag specifies the direction the wind is *coming from*, in degrees using the compass convention:

- **0°** — Headwind (wind blowing into the shooter's face)
- **90°** — Wind from the right (full-value crosswind)
- **180°** — Tailwind
- **270°** — Wind from the left (full-value crosswind)

A 90° wind is a full-value crosswind from the right. This produces the maximum lateral deflection. A 45° wind is a quartering headwind from the right—it produces roughly 70% of the full-value deflection.

**Wind Value**

The *wind value* is the crosswind component: the portion of the wind perpendicular to the line of fire. For a wind blowing at angle $\theta$ from the shooter's heading:

$$v_{\text{cross}} = v_{\text{wind}} \sin\theta \tag{4.5}$$

A 10 mph wind at 30° produces only $10 \times \sin(30) = 5$ mph of effective crosswind.

Internally, the engine decomposes the wind into lateral (crosswind) and longitudinal (head/tailwind) components. The crosswind component pushes the bullet sideways, while a headwind slightly

increases the effective drag and a tailwind slightly decreases it. Both effects are accounted for in the integration.

Listing 4.27: Comparing full-value and half-value crosswind

```
# Full-value crosswind (90 degrees, wind from right)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --wind-speed 10 \
  --wind-direction 90

# Quartering wind (45 degrees --- 70% of full value)
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --wind-speed 10 \
  --wind-direction 45
```

### 4.7.3 Wind Shear

Real wind varies with altitude. At ground level, terrain and vegetation slow the wind, while higher up it blows more freely. The `--enable-wind-shear` flag activates altitude-dependent wind modeling:

Listing 4.28: Enabling wind shear

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --wind-speed 10 \
  --wind-direction 90 --enable-wind-shear
```

The wind shear model uses either a power-law or logarithmic profile to compute wind speed as a function of altitude above the surface. At long range, where the bullet may travel well above ground level due to its arcing trajectory, this produces more accurate wind drift predictions than a constant-wind model. Wind shear is covered in detail in Chapter 10.

## 4.8 Sight Height and Scope Offset

The sight (scope) sits above the bore centerline. This offset means the bullet and the line of sight start at different heights and converge at the zero range. Getting this geometry right matters—especially at close range where the parallax is significant.

### 4.8.1 Sight Height (`--sight-height`)

The `--sight-height` flag specifies the vertical distance from the bore centerline to the optical centerline of the scope, in inches (imperial) or millimeters (metric):

Listing 4.29: Specifying sight height

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5
```

A typical bolt-action rifle with a scope on medium-height rings has a sight height of approximately 1.5–2.0 inches. AR-platform rifles with flat-top receivers and tall optics mounts may be 2.5 inches or more.

**Measure Your Sight Height**

Use calipers to measure from the center of your bore to the center of your scope's objective lens. This is more accurate than guessing from ring height specifications, which don't account for receiver shape.

Sight height has two important effects:

1. **Near-zero offset**: At very close range (e.g., 25 yards), the bullet is still below the line of sight by approximately the sight height. This matters for precision close-range shots.

2. **Zero angle**: A higher sight height requires a slightly steeper launch angle to achieve the same zero distance, which changes the trajectory shape.

### 4.8.2 Bore Height (`--bore-height`)

The `--bore-height` flag specifies the height of the bore above the ground. This affects ground-impact detection—the engine can detect when the bullet falls below ground level and stop the trajectory. The default is 60 inches (1500 mm), which assumes a standing shooter:

Listing 4.30: Setting bore height for prone shooting

```
# Prone shooter, bore approximately 12 inches above ground
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --bore-height 12
```

**Breaking: `--bore-height` Is Inches Now (0.26.0)**

Until v0.26.0 this flag took *feet* in imperial and *metres* in metric, which made it the one length flag on the command line that did not agree with `--sight-height`. As of v0.26.0 it takes inches and millimetres like everything else. The defaults are numerically unchanged—60″ and 1500 mm are the same 5 ft and 1.5 m as before—so any run that never sets the flag is byte-identical. **But a script passing `--bore-height 1` now means one inch, not one foot.** Both surfaces also gained each other's spelling: native `--bore-height` accepts `--muzzle-height` as an alias, and the WASM build's `--muzzle-height` accepts `--bore-height`, with identical units and defaults on each.

To disable ground impact detection entirely (useful for theoretical calculations or extreme long-range analysis), use `--ignore-ground-impact`:

Listing 4.31: Disabling ground impact detection

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 2000 \
  --ignore-ground-impact
```

## 4.9 Inclination: Uphill and Downhill Shooting

The `--shooting-angle` flag specifies the angle of inclination in degrees. Positive values mean uphill; negative values mean downhill:

Listing 4.32: Uphill and downhill shooting

```
# 15 degrees uphill
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --shooting-angle 15

# 20 degrees downhill
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --shooting-angle -20
```

### 4.9.1 Equivalent Horizontal Range

Set a look angle and, since v0.31.0, the summary gains one more line:

```
Equivalent horizontal range: 428 yd (shoot-to for BDC)
```

This is the "shoot-to" range that SIG's AMR, Leica's EHR and Gunwerks all report: the flat-fire range whose *angular* elevation correction against the same zero matches the inclined solution's. Point a BDC reticle or a range-card at that number and you get the right hold for the inclined shot.

It is deliberately *not* the Rifleman's-Rule cosine approximation. The engine solves it by the McCoy/Litz angular-match inversion over one extra flat re-solve, so it stays correct at the steep angles where the cosine rule falls apart.

**This Line Appears Without an Opt-In**

Along with `summary.equivalent_horizontal_range_m` in solve-json, this is one of two intentional default-output changes in v0.31.0. It appears on *any* inclined shot—a shoot-to range is meaningful exactly when a look angle is set—and flat-fire output is unchanged. It is omitted for flat shots, for targets at or inside the zero range, and for non-positive corrections. A consumer that rejects unknown response fields will see this on inclined-shot responses.

### 4.9.2 Which Plane Is "Drop" Measured In?

There are two defensible answers to "how far did the bullet drop" on an inclined shot, and `--drops-reference` lets you pick:

Listing 4.33: Target-plane drop on a 30-degree shot

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 800 --shooting-angle 30 \
  --sample-trajectory --sample-interval 100 \
  --drops-reference target
```

`los` is the default and keeps the historical LOS-perpendicular sampled drop byte-identically. `target` reports drop as vertical in the target plane—the LOS-perpendicular drop divided by $\cos(\text{shooting angle})$, which is JBM's "target plane" reference—and relabels the column `Drop (target)` (`drop_target_in` in CSV) so the two can never be confused. This is an output-mode toggle only: the solved trajectory, the drift and the zeroing semantics are untouched. It is rejected at shooting angles of 90° or beyond.

### 4.9.3 Sight Geometry: Offset Mounts and Deliberate Zeros

`--sight-height` models the vertical half of sight geometry. `--sight-offset` (v0.31.0) models the windage half, for an optic mounted off the bore centerline—an offset red dot, a canted mount, a rail with a sideways set screw:

Listing 4.34: An optic mounted half an inch right of bore

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --sight-height 1.5 \
  --sight-offset 0.5 --max-range 500
```

Positive means the sight is *right* of bore, so the bullet starts that far left of the sight line. When a zero is solved, the windage-zero convergence (offset divided by zero distance) is applied so the trajectory crosses the sight line laterally at the zero range—this is a convergence, not a constant parallel offset. Without a zero solve, only the physical displacement applies. The flag is available on **`trajectory`**, `mpbr`, `come-ups`, `wind-card`, `range-table`, `compare` and `monte-carlo`.

Distinct from that—and additive with it—`--zero-poi-up` and `--zero-poi-right` record that the rifle is *deliberately* zeroed off, which is Kestrel's zero-height and zero-offset semantics:

Listing 4.35: A rifle deliberately zeroed 0.1 inch high and 0.2 inch left

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --sight-height 1.5 \
  --zero-poi-up 0.1 --zero-poi-right -0.2 \
  --max-range 600
```

Positive means the rifle impacts high, or right, by that much *at the zero distance*. The zero search still solves perfect convergence; the offset is applied afterwards as an angular bias (offset divided by zero distance) on the solved elevation and azimuth. The distinction from `--sight-offset` is worth keeping straight: one is an angular *zero state*, the other is physical *mount geometry*.

**Everything Here Is Opt-In**

Omitting all of these inputs is byte-identical to v0.30.x. Each has a saved-profile field (`sight_offset_lateral_m`, `zero_poi_up_m`, `zero_poi_right_m`, always meters) and a solve-json v1 request field, both additive.

### 4.9.4 The Rifleman's Rule

Both uphill and downhill shooting cause the bullet to impact *higher* than the flat-ground trajectory would predict. The classic *Rifleman's Rule* approximation says to use the horizontal distance (the cosine of the angle times the line-of-sight distance) as the effective range:

$$R_{\text{effective}} = R_{\text{LOS}} \cos\theta \tag{4.6}$$

For moderate angles this works well. At a 20° angle, $\cos(20) = 0.94$, so a 500-yard line-of-sight shot is equivalent to a 470-yard flat-ground shot for drop purposes. But at steep angles (above 30°), the Rifleman's Rule becomes less accurate because it ignores the changing gravity component along the trajectory.

BALLISTICS-ENGINE does not use the Rifleman's Rule approximation. Instead, it solves the full three-dimensional trajectory with the actual gravity vector decomposition. This is automatically more accurate at all angles.

**Inclination Was Accepted but Ignored Before 0.22.0**

Through v0.21.x `--shooting-angle` was accepted by every surface—the CLI, the FFI, the WASM build, and the low-level integration path the language bindings call—and read by no solver. An inclined shot returned the flat-fire answer. Version 0.22.0 rotates gravity into the shot-aligned frame everywhere, including `fast_integrate_with_segments`, so the flag now does what it always claimed to. If you have compared the engine against a Rifleman's-Rule calculator in the past and found suspiciously perfect agreement with the *flat* answer, this is why.

**Why Both Directions Shoot High**

Gravity acts vertically, but drop is measured perpendicular to the line of sight. When shooting at an angle, the gravity component perpendicular to the line of sight is reduced by $\cos\theta$. Since the *effective gravity* pulling the bullet away from the line of sight is less, the bullet drops less relative to your aim point. This applies whether you are shooting uphill or downhill.

## 4.10 Atmospheric Conditions

Atmospheric conditions affect air density, which directly affects drag. The engine accepts four atmospheric parameters:

Listing 4.36: Specifying atmospheric conditions

```
ballistics trajectory --velocity 2700 \
```

```
--bc 0.462 --mass 168 --diameter 0.308 \
--auto-zero 200 --max-range 1000 \
--temperature 85 --pressure 29.12 \
--humidity 60 --altitude 5280
```

### 4.10.1 Temperature (`--temperature`)

Temperature in Fahrenheit (imperial) or Celsius (metric). Default: 59°F (15°C). Higher temperatures reduce air density, resulting in less drag and a flatter trajectory. Internally, the engine converts to Kelvin for the ideal gas law calculation used in `calculate_air_density()` within `src/cli_api.rs`:

$$\rho = \frac{P}{R_s \cdot T} \tag{4.7}$$

where $P$ is pressure in Pascals, $R_s = 287.058\ \mathrm{J/(kg \cdot K)}$ is the specific gas constant for dry air, and $T$ is temperature in Kelvin.

### 4.10.2 Pressure (`--pressure`)

Barometric pressure in inches of mercury (imperial) or hectopascals (metric). Default: 29.92 inHg (1013.25 hPa). Lower pressure (as at altitude) reduces air density.

### 4.10.3 Humidity (`--humidity`)

Relative humidity as a percentage (0–100). Default: 50%. Contrary to intuition, humid air is *less* dense than dry air because water vapor ($H_2O$, molecular weight 18) is lighter than the nitrogen ($N_2$, molecular weight 28) and oxygen ($O_2$, molecular weight 32) it displaces. The effect is small—typically less than 1% change in air density—but the engine accounts for it.

### 4.10.4 Altitude (`--altitude`)

Altitude above sea level in feet (imperial) or meters (metric). Default: 0. Altitude applies an exponential correction to the air density calculation:

$$\rho_{\text{alt}} = \rho_0 \cdot e^{-h/8000} \tag{4.8}$$

where $h$ is altitude in meters and the scale height of 8000 m is an approximation of the atmospheric pressure decay.

**Altitude vs. Station Pressure**

If you have a weather station reading at your shooting location, it likely reports pressure adjusted to sea level (altimeter setting). For the most accurate results, either: (1) use the altimeter setting with `--altitude` set to your actual elevation, or (2) use station pressure (true local pressure) with `--altitude` set to zero. Do not double-count altitude correction by using station pressure *and* a non-zero altitude.

A comprehensive treatment of atmospheric modeling—including the ICAO Standard Atmosphere, density altitude, and the interplay between these parameters—is in Chapter 9.

## 4.11 Practical Examples: Building a Dope Card

Let's put it all together. We will build a complete dope card for a precision rifle shooter preparing for a match at moderate altitude with wind.

### 4.11.1 Example 1: .308 Win Match Rifle at 5000 ft Elevation

Our rifle is zeroed at 100 yards. We are shooting a .308 Win with 168gr Sierra MatchKing at 2700 fps. The match is at 5000 ft elevation, 75°F, 25.10 inHg station pressure. We expect a 10 mph full-value crosswind from the right:

Listing 4.37: Complete dope card for .308 Win at altitude

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g1 --auto-zero 100 \
  --max-range 1000 --sight-height 1.5 \
  --temperature 75 --pressure 25.10 \
  --altitude 0 --humidity 40 \
  --wind-speed 10 --wind-direction 90
```

Notice that we set `--altitude` to 0 because we are providing the actual station pressure at our elevation (25.10 inHg), not a sea-level-corrected pressure. This avoids double-counting the altitude correction.

### 4.11.2 Example 2: 6.5 Creedmoor for PRS Competition

A Precision Rifle Series competitor with a 6.5 Creedmoor shooting 140gr ELD-M at 2710 fps. Zeroed at 100 yards, shooting out to 1200 yards. Standard conditions but with 8 mph quartering wind:

Listing 4.38: PRS dope card with 6.5 Creedmoor

```
ballistics trajectory --velocity 2710 \
  --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --auto-zero 100 \
  --max-range 1200 --sight-height 2.0 \
  --wind-speed 8 --wind-direction 45
```

### 4.11.3 Example 3: .338 Lapua ELR with PDF Output

For extreme long-range shooting with a .338 Lapua Magnum, 300gr Berger Hybrid at 2725 fps. We want a PDF dope card we can print and laminate:

Listing 4.39: .338 Lapua ELR dope card as PDF

```
ballistics trajectory --velocity 2725 \
  --bc 0.818 --mass 300 --diameter 0.338 \
  --drag-model g7 --auto-zero 100 \
  --max-range 2000 --sight-height 2.0 \
  --wind-speed 10 --wind-direction 90 \
  --temperature 65 --altitude 3000 \
  --pressure 26.82 --humidity 35 \
  --sample-trajectory \
  --output pdf --output-file 338_dope.pdf \
  --bullet-name "300gr Berger Hybrid" \
  --ignore-ground-impact
```

We use `--ignore-ground-impact` because we want trajectory data all the way to 2000 yards even though the bullet's arc will take it well below the bore height at those distances.

### 4.11.4 Example 4: Hunting Load with Uphill Angle

A hunter in the mountains with a .308 Win, planning a steep 25° uphill shot at 350 yards:

Listing 4.40: Uphill hunting shot

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 400 \
  --shooting-angle 25 --sight-height 1.5 \
  --temperature 40 --altitude 8000
```

Compare this output with the same shot on flat ground (omit `--shooting-angle`) to see the difference the inclination makes. The uphill shot will show less drop at the target distance because the effective gravity component pulling the bullet below the line of sight is reduced.

### 4.11.5 Example 5: CSV Export for Spreadsheet Analysis

If you want to import trajectory data into a spreadsheet for custom analysis or plotting:

Listing 4.41: Exporting trajectory data for analysis

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000 \
  --wind-speed 10 --wind-direction 90 \
  --output csv > my_308_trajectory.csv
```

The CSV output includes headers and can be directly opened in Excel, Google Sheets, or any data analysis tool.

## 4.12 Advanced Physics Flags

For extreme long-range work, several second-order effects become significant. The `trajectory` command exposes flags for each:

Listing 4.42: Enabling advanced physics effects

```
ballistics trajectory --velocity 2725 \
  --bc 0.818 --mass 300 --diameter 0.338 \
  --drag-model g7 --auto-zero 100 \
  --max-range 2000 --sight-height 2.0 \
  --wind-speed 5 --wind-direction 90 \
  --twist-rate 10 --latitude 40 \
  --shot-direction 90 \
  --enable-spin-drift --enable-coriolis \
  --ignore-ground-impact
```

### 4.12.1 Spin Drift (`--enable-spin-drift`)

A spinning bullet drifts laterally in the direction of its spin. For a right-hand twist barrel, spin drift pushes the bullet to the right. The effect is negligible inside 500 yards but grows with distance. At 1000 yards with a .308 Win, expect 3–5 inches of spin drift; at 2000 yards with a .338 Lapua, it can exceed 2 feet.

Spin drift requires the `--twist-rate` parameter (barrel twist rate in inches per turn, e.g., 10 for a 1:10 twist).

### 4.12.2 Coriolis Effect (`--enable-coriolis`)

The Earth rotates under the bullet during its flight. The Coriolis effect produces a small lateral deflection that depends on the shooter's latitude, the direction of fire, and time of flight. It requires `--latitude` (degrees) and `--shot-direction` (degrees, compass convention).

In the Northern Hemisphere, a bullet fired north deflects to the east; fired south, it deflects to the west. The effect is about 3 inches per second of flight time at mid-latitudes.

### 4.12.3 Magnus Effect (`--enable-magnus`)

The Magnus effect produces a small force perpendicular to the bullet's velocity and spin axis. It is a tertiary effect that is rarely significant in practice but is included for completeness. It requires the `--twist-rate` parameter.

These advanced effects are discussed in depth in Part V (Chapters 14 and 15).

## 4.13 Exercises

These exercises are designed to be run at your terminal. Work through them to build intuition for how input changes affect trajectory predictions.

1. **Velocity sensitivity.** Run the following two commands and compare the drop at 600 yards. How much does a 50 fps change in muzzle velocity affect the impact point?

```
ballistics trajectory --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 600

ballistics trajectory --velocity 2650 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 600
```

2. **G1 vs. G7.** Compute the trajectory for a 6.5 Creedmoor, 140gr ELD-M at 2710 fps using both drag models. Use `0.610` for G7 and `0.326` for G1. Compare the predicted drop at 1000 yards. Which model predicts more drop?

```
ballistics trajectory --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 --drag-model g7 \
  --auto-zero 100 --max-range 1000

ballistics trajectory --velocity 2710 --bc 0.326 \
  --mass 140 --diameter 0.264 --drag-model g1 \
```

```
  --auto-zero 100 --max-range 1000
```

3. **Wind sensitivity.** Using the .308 Win load from the chapter, compute trajectories at 0, 5, 10, and 15 mph crosswind (90°). How does windage scale with wind speed? Is it linear?
4. **Altitude effect.** Compute the trajectory for the .308 Win load at sea level (59°F, 29.92 inHg) and at 8000 ft (40°F, 22.22 inHg, station pressure). Compare the velocity and drop at 800 yards. How much flatter is the high-altitude trajectory?
5. **Steep angle shot.** Compare a flat-ground 400-yard shot with a 30° uphill 400-yard shot using the .308 Win load. Verify that the angled shot shows less drop. By how many inches do they differ?

## What's Next

Now that you can compute a trajectory for any load and any conditions, the next question is: *how do you determine the correct zero?* In Chapter 5, we examine the `zero` command in detail—how the zeroing algorithm works, the relationship between near-zero and far-zero distances, maximum point-blank range, and how atmospheric conditions affect your zero. Understanding zeroing transforms the trajectory command from a theoretical tool into a practical one.

# Chapter 5

# Zeroing

A rifle's zero is the bridge between theory and practice. Without it, the drop values from Chapter 4 are expressed relative to the bore line—a reference that exists inside the barrel but not behind your reticle. Zeroing establishes the relationship between where you *aim* and where the bullet *hits* at a known distance, and from that single calibration point the engine can predict impact at every other distance.

In this chapter we examine the `zero` command, the algorithm that drives it, the geometry of near-zero and far-zero distances, maximum point-blank range (MPBR), and how atmospheric conditions interact with your zero.

## 5.1 What Zeroing Means and Why It Matters

Consider the geometry. Your scope sits above the bore—typically 1.5 to 2.5 inches higher. The line of sight (the straight line from the scope through the reticle to the target) and the bore line (the axis of the barrel) are not parallel; they converge at some point downrange. Zeroing adjusts the scope so that the bullet's *arcing trajectory* crosses the flat line of sight at a chosen distance.

**Zero Distance**

The *zero distance* is the range at which the bullet's trajectory crosses the line of sight, making the vertical drop read exactly zero. At this distance, the point of aim and the point of impact coincide.

After zeroing at, say, 200 yards, the trajectory table from Chapter 4 acquires its practical meaning:

- At ranges *shorter* than the zero distance, the bullet is slightly above the point of aim (positive drop).

- At the zero distance, drop is zero.
- At ranges *beyond* the zero distance, the bullet falls increasingly below the point of aim (negative drop).

The choice of zero distance is one of the most consequential decisions a shooter makes, and it depends entirely on application. A hunter wanting point-and-shoot simplicity might zero at 200–250 yards. A precision rifle competitor who dials elevation for every shot typically zeros at 100 yards for a clean reference. A military sniper might zero at 300 meters.

BALLISTICS-ENGINE supports zeroing through two mechanisms: the `zero` command (which computes the zero angle directly) and the `--auto-zero` flag on the `trajectory` command (which applies the zero angle before computing the full trajectory). We covered `--auto-zero` in Section 4.4; here we focus on the dedicated command and the physics behind it.

## 5.2 The `zero` Command

The `zero` command computes the muzzle angle required to place the bullet on the line of sight at a specified target distance. It reports the angle in degrees, MOA, and milliradians, plus the maximum ordinate (the highest point the bullet reaches above the line of sight):

Listing 5.1: Basic zero calculation for .308 Win at 200 yards

```
ballistics zero \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 200 --sight-height 1.5
```

The output looks like:

Listing 5.2: Zero command output

```
+========================================+
|          ZERO CALCULATION              |
+========================================+
| Target Distance:      200.0 yd         |
| Target Height:         0.00 yd         |
| Sight Height:         0.042 yd         |
+========================================+
| Zero Angle:          0.0960°           |
| Zero Angle (MOA):      5.76 MOA        |
| Zero Angle (mrad):     1.68 mrad       |
| Max Ordinate:         0.100 yd         |
+========================================+
```

**zero Now Zeroes to the Line of Sight (0.22.0)**

Notice that the sight height we passed—1.5 inches, or 0.042 yards—appears in the output. Through v0.21.x it appeared there and nowhere else: the `zero` subcommand solved for the angle that put the bullet on the *bore line* at the target distance and ignored `--sight-height` entirely. At a typical sight height that is an error of roughly 1.9 MOA, which is why the angle printed by v0.21.x for this load (0.0553°, 3.32 MOA) is so much smaller than the correct 0.0960°. The command now solves to the line of sight, as `--auto-zero` always did.

Two neighbouring fixes landed with it. `sight_adjustment_moa` had its sign inverted—it subtracted the sight-height correction instead of adding it, and so told you to dial the wrong way. And `point_blank_range` was reading the lateral axis, which is always zero, against a bore-referenced threshold; it now measures downrange against the line of sight.

### 5.2.1 Required Flags

The `zero` command requires the same core ballistic parameters as `trajectory`, plus the target distance:

- `--velocity` — Muzzle velocity (fps or m/s)
- `--bc` — Ballistic coefficient
- `--mass` — Bullet mass (grains or grams)
- `--diameter` — Bullet diameter (inches or mm)
- `--target-distance` — The desired zero distance (yards or meters)

### 5.2.2 Optional Flags

- `--sight-height` — Distance from bore centerline to scope centerline. Defaults to 2.0 inches (imperial) or 50 mm (metric).
- `--target-height` — Vertical offset of the target above or below the bore line at the target distance. Default: 0. This is useful for zeroing on a target that is not at the same elevation as the shooter.
- `--temperature`, `--pressure`, `--humidity`, `--altitude` — Atmospheric conditions at the zeroing location. Defaults to standard ICAO conditions (59°F, 29.92 inHg, 50% RH, sea level).
- `--output` — Output format: `table`, `json`, or `csv`.

Listing 5.3: Zero calculation with full atmospheric conditions

```
ballistics zero \
  --velocity 2700 --bc 0.462 \
```

```
  --mass 168 --diameter 0.308 \
  --target-distance 100 --sight-height 1.5 \
  --temperature 75 --pressure 25.10 \
  --humidity 40 --altitude 5000
```

**Use the Same Conditions for Zero and Trajectory**

When you compute a trajectory with `--auto-zero`, the engine automatically uses the same atmospheric conditions for both the zero calculation and the trajectory computation. If you use the **`zero`** command separately, make sure the atmospheric flags match the conditions under which you zeroed the rifle, not the conditions at your shooting location.

### 5.2.3 Working Backwards: `--from-angle`

Sometimes you have the angle and want the range. A stored bore angle—read out of a profile, recovered from a device, or simply written down after a session—implies a zero distance, and since v0.29.0 **`zero`** will recover it:

Listing 5.4: Recovering the zero range a stored angle implies

```
ballistics zero --from-angle 0.0960 \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 --sight-height 1.5
```

Listing 5.5: Two crossings, both reported

```
+=======================================+
|     ZERO RANGE FROM STORED ANGLE      |
+=======================================+
| Input Zero Angle:    0.0960°          |
| Input Zero Angle:      5.76 MOA       |
| Input Zero Angle:      1.68 mrad      |
+=======================================+
| Near Zero (ascending):     28.0 yd    |
| Far Zero (descending):    200.0 yd    |
| Target Height:          0.00 yd       |
| Sight Height:          0.042 yd       |
| Max Ordinate:          0.100 yd       |
+=======================================+
```

A bore angle generally crosses the line of sight *twice*, and the command reports both—the same relationship, in reverse, that makes a 25-yard zero and a roughly 300-yard zero the same angle (Section 5.4).

The JSON form adds a `primary_crossing` key reading `"far"` or `"near"` so a consumer reading the single `zero_range` value can tell which of the two valid answers it received.

**A Recovered Range Is Conditional**

The recovered range reproduces the original zero only under the same wind, the same bore height, and the same zero-day atmosphere. An angle is a fact about the rifle; the range it corresponds to is a fact about the rifle *and* the air it was zeroed in.

## 5.3 How the Zeroing Algorithm Works

Computing the zero angle is a root-finding problem. We seek the muzzle angle $\alpha$ such that the bullet's vertical position at the target distance equals the line-of-sight height:

$$y_{\text{bullet}}(\alpha, R_{\text{zero}}) - y_{\text{LOS}}(R_{\text{zero}}) = 0 \tag{5.1}$$

where $y_{\text{bullet}}$ is the bullet's height at the zero range (a function of the launch angle and the full trajectory physics) and $y_{\text{LOS}}$ is the line-of-sight height at the same range.

### 5.3.1 Brent's Method

BALLISTICS-ENGINE solves this equation using **Brent's method**, implemented in the `brent_root_find()` function in `src/angle_calculations.rs`. Brent's method combines the reliability of the bisection method with the speed of inverse quadratic interpolation and the secant method. It guarantees convergence (unlike Newton's method, which can diverge) while converging superlinearly in most cases.

The algorithm works as follows:

1. Start with an initial bracket: $[-10, +10]$ in radians. Within this bracket, the height error function must change sign (i.e., the bullet is below the line of sight at one bound and above it at the other).
2. At each iteration, attempt inverse quadratic interpolation (IQI) to find a better estimate of the root.
3. If the IQI step would fall outside the bracket or produce insufficient progress, fall back to bisection.
4. Repeat until the height error is below $10^{-6}$ meters (approximately 0.00004 inches).

**Convergence Guarantee**

Brent's method converges in at most 100 iterations. In practice, the zero angle for typical rifle cartridges converges in 10–20 iterations. If the primary bracket fails (e.g., for extreme cartridges or steep angles), the algorithm falls back to a wider bracket of $[-45, +45]$ with a relaxed tolerance.

### 5.3.2 Under the Hood: The Height Error Function

At each candidate angle $\alpha$, the engine runs a complete trajectory integration from the muzzle to the zero distance, evaluating drag, gravity, and (optionally) wind at every time step. The height error is:

$$\varepsilon(\alpha) = y_{\text{bullet}}(\alpha, R_{\text{zero}}) - y_{\text{target}} \tag{5.2}$$

This means each iteration of Brent's method involves a full trajectory solve—the engine evaluates the complete physics of the bullet's flight to determine whether this angle puts the bullet above or below the target. This is computationally intensive, but with the RK4 integrator a single trajectory solve takes microseconds, so even 100 iterations complete in milliseconds.

The function `zero_angle()` in `src/angle_calculations.rs` orchestrates this process. If a trajectory simulation fails for a given angle (e.g., the bullet impacts the ground before reaching the target distance), the function returns `NaN` rather than a sentinel value, which causes Brent's method to fail gracefully and try a different angle.

## 5.4 Near-Zero and Far-Zero Distances

Because the bullet follows a parabolic-like arc while the line of sight is straight, the trajectory crosses the line of sight *twice*—once on the way up and once on the way down:

**Near-Zero and Far-Zero**

**Near-zero**: The first crossing, where the bullet rises through the line of sight. This occurs relatively close to the muzzle (typically 20–35 yards for a 100-yard zero, or 25–50 yards for a 200-yard zero).
**Far-zero**: The second crossing, where the bullet falls back through the line of sight. This is the zero distance you set.

At the muzzle, the bullet starts below the line of sight (because the bore is below the scope). As the slightly upward-angled trajectory carries the bullet past the scope's line of sight, it crosses upward—this is the near-zero. The bullet continues to rise, reaches its maximum ordinate, then begins to descend, crossing the line of sight a second time—the far-zero.

You can observe both crossings in a trajectory table:

Listing 5.6: Observing near-zero and far-zero

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 300 \
  --sight-height 1.5 \
  --sample-trajectory --sample-interval 10
```

Look for the two points where the drop column crosses zero. The first crossing (around 25–30 yards) is the near-zero; the one at 200 yards is the far-zero.

### 5.4.1 Why Near-Zero Matters

The near-zero is critical in two scenarios:

1. **Close-range precision shots**: If you zero at 200 yards and take a shot at 10 yards, the bullet will strike approximately 1.5 inches *below* your point of aim (roughly equal to the sight height). At 25 yards it will be nearly on, and between 25 and 200 yards it will be above.
2. **AR-15 height-over-bore**: On an AR-15 with a 2.6-inch sight height, the offset at close range is significant. A shot at 7 yards aimed at the center of a target may impact 2.6 inches low.

**Close-Range Offset**

Always know your near-zero distance. Inside the near-zero, the bullet impacts *below* the crosshair by approximately the sight height. This matters for close-range engagements, mechanical offset drills, and CQB applications. Use the engine to compute the near-zero for your specific setup.

## 5.5 Point-Blank Range and Maximum Ordinate

### 5.5.1 Maximum Ordinate

The *maximum ordinate* is the highest point the bullet reaches above the line of sight during its flight. The `zero` command reports this value directly:

Listing 5.7: Checking maximum ordinate

```
ballistics zero \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 200 --sight-height 1.5
# Max Ordinate: approximately 1.8 inches above LOS
```

The maximum ordinate determines whether the bullet might shoot *over* a small target at mid-range. If you zero at 300 yards with a fast cartridge, the bullet might be 5–6 inches above the line of sight at 150 yards—enough to sail over a prairie dog.

### 5.5.2 Maximum Point-Blank Range

*Maximum Point-Blank Range* (MPBR) is the farthest distance at which you can aim at the center of a vital zone and guarantee a hit without holdover or holdunder. It is found by adjusting the zero distance so that the maximum ordinate equals exactly half the vital zone diameter:

$$\text{max ordinate} = \frac{d_{\text{vital}}}{2} \tag{5.3}$$

When this condition is met, the bullet's trajectory stays within $\pm\frac{d_{\text{vital}}}{2}$ of the line of sight from the muzzle out to the MPBR. Beyond the MPBR, the bullet drops below the bottom of the vital zone.

The `mpbr` command computes this automatically:

Listing 5.8: Computing MPBR for deer hunting

```
ballistics mpbr \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --vital-zone 8.0 --sight-height 1.5
```

The `--vital-zone` flag specifies the diameter of the kill zone in inches (imperial) or centimeters (metric). Common values:

- **8 inches** — Whitetail deer vital zone
- **10 inches** — Elk vital zone
- **4 inches** — Coyote/varmint vital zone
- **12 inches** — Target plate (competition)

### 5.5.3 MPBR Output

The `mpbr` command reports:

- **Optimal zero distance** — The zero that maximizes point-blank range
- **Near-zero** — First line-of-sight crossing
- **Far-zero** — Second crossing (the optimal zero distance)

- **Maximum ordinate** — Peak height above LOS (should be half the vital zone)
- **MPBR** — The maximum range at which the bullet stays within the vital zone
- **Impact velocity and energy at MPBR** — Confirms adequate terminal performance

### 5.5.4 The MPBR Algorithm

Internally, the `handle_mpbr()` function in `src/main.rs` uses a binary search on the zero distance:

1. Start with a search range from 50 to 2000 yards.
2. For each candidate zero distance, compute the full trajectory and find the maximum ordinate.
3. If the maximum ordinate is greater than half the vital zone, decrease the zero distance (the bullet is going too high mid-flight).
4. If the maximum ordinate is less than half the vital zone, increase the zero distance.
5. Repeat until the maximum ordinate is within 0.001 meters (≈0.04 inches) of the target.

The algorithm uses up to 60 iterations with 1-yard sampling resolution. For each candidate zero, it runs a complete trajectory with the `calculate_zero_angle_with_conditions()` function from the `ballistics_engine` library, then samples the trajectory to find the maximum ordinate.

Listing 5.9: MPBR for a flat-shooting 6.5 Creedmoor

```
ballistics mpbr \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --drag-model g7 --vital-zone 8.0 \
  --sight-height 2.0
```

**MPBR Is a Hunting Concept**

MPBR is most useful for hunting, where you need to take quick shots at varying distances without dialing your scope. For competition shooting where you have time to dial, a standard 100-yard zero with a precise dope card is usually preferred.

## 5.6 Zeroing at Altitude and in Non-Standard Conditions

A zero established at sea level on a cool morning will not hold when you take that rifle to 8000 feet elevation on a hot afternoon. The change in air density alters the drag on the bullet, which changes the trajectory shape and shifts the zero.

### 5.6.1 How Conditions Affect Zero

Higher altitude and warmer temperatures reduce air density, which reduces drag. With less drag, the bullet retains more velocity, drops less, and arrives at the zero distance *above* where it would at sea level. In practical terms:

- Zeroed at sea level, then shooting at altitude: the rifle shoots *high*.
- Zeroed at altitude, then shooting at sea level: the rifle shoots *low*.

The magnitude depends on how much the air density changes. A rough rule of thumb: every 1000 feet of elevation reduces air density by about 3%, which translates to approximately 0.5 MOA of zero shift for a typical .308 Win at 500 yards.

### 5.6.2 Computing Zero Shift

To quantify the shift, run two zero calculations with different atmospheric conditions:

Listing 5.10: Zero shift from sea level to 7000 ft

```
# Zero at sea level (standard conditions)
ballistics zero \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 100 --sight-height 1.5 \
  --temperature 59 --pressure 29.92 --altitude 0

# Zero at 7000 ft, summer conditions
ballistics zero \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 100 --sight-height 1.5 \
  --temperature 85 --pressure 23.09 --altitude 0
```

Compare the zero angles. The difference tells you how much your scope setting needs to change. You can also run full trajectories at both conditions with the same `--auto-zero` distance and compare the drop tables at your target distances.

**SAFETY: Re-Zero When Conditions Change Significantly**

If you are traveling from a low-elevation zeroing range to a high-altitude hunting area, verify your zero on arrival. Computational predictions are accurate, but confirming zero at the actual shooting conditions eliminates risk from unmodeled variables (scope tracking errors, temperature effects on the scope or ammunition, etc.).

### 5.6.3 Using `auto-zero` with Field Conditions

The most common workflow is to use `--auto-zero` on the `trajectory` command with the conditions at your *shooting location*, not your zeroing location:

Listing 5.11: Trajectory with field conditions

```
# Rifle zeroed at 100 yards (at home range, sea level)
# Now shooting at a mountain match at 7000 ft
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --temperature 85 --pressure 23.09 \
  --humidity 20
```

This computes the trajectory at the field conditions *and* computes the zero angle for those same conditions. The resulting dope card accounts for the thinner air.

**When to Use Zero Command vs. Auto-Zero**

Use the `zero` command when you need the raw angle and maximum ordinate (e.g., for verifying scope adjustment or computing point-blank range). Use `--auto-zero` on the `trajectory` command when you want a complete trajectory table referenced to a specific zero distance. For most users, `--auto-zero` is all they need.

**`--auto-zero` Now Uses Your Drag Model and Your Air (0.22.0)**

The claim in the box above—that `--auto-zero` solves the zero under the same conditions as the trajectory—became true in v0.22.0. Before that, the auto-zero solve ran against a hardcoded G1 model at ICAO sea level regardless of what the trajectory used. A G7 auto-zero therefore struck several inches off the requested zero distance, and an altitude zero was solved at sea level. The zero search now runs the user's actual drag model and resolved atmosphere.

Version 0.22.6 then tightened the search itself. It had been stopping as soon as the height error at the zero distance fell under 1 mm—about 0.037 MOA at 100 yards, which is coarser than the zero-day density effect at that distance. Two very different atmospheres could round to the same zero angle, which is why a `--auto-zero 100` run with extreme zero-day conditions used to produce byte-identical output to one without them. The tolerance is now 0.1 mm ($10^{-7}$ rad). The effect it resolves is genuinely small—about 0.04″ at 100 yards for those extreme conditions—but it is no longer quantised away, and every zero angle is more precise as a result.

**Inclined Zero Solves Were Broken from 0.25.0 to 0.28.0**

Between v0.25.0 and v0.28.0 any nonzero `--shooting-angle` made `calculate_zero_angle_with_conditions` fail outright with "Zero angle did not converge"—on every surface behind it: the bindings, the browser terminal, the FFI, and the CLI. Version 0.28.1 fixed it by zeroing a *level* rifle, which is the correct doctrine anyway and matches how cant is handled: a rifle has one zero, set on a level range, and the shot's slope is a property of the shot. The world-vertical zero contract documented for solve-json v1 is unchanged.

### 5.6.4 Zeroing on One Day, Shooting on Another

The atmospheric flags on **`trajectory`** describe the day of the shot. But a rifle's zero is a physical property of a bore angle set on some *other* day, under whatever conditions prevailed then. If you sighted in on a still 70 °F afternoon and you are now shooting at 20 °F, the bore angle has not changed—but the trajectory that angle produces has.

Since v0.22.1 you can say both things at once. Each `--zero-*` flag sets a zero-day value used only for the `--auto-zero` solve; anything you leave out falls back to its shot-day counterpart, so supplying none of them reproduces single-condition behavior exactly:

Listing 5.12: A 200-yard zero torn at 70°F, shot at 20°F

```
ballistics trajectory --velocity 2700 \
  --velocity-adjustment -50 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 600 \
  --sight-height 1.5 \
  --temperature 20 \
  --zero-temperature 70 --zero-velocity 2700
```

The engine says so before it prints anything:

```
Solving zero angle under supplied zero-day conditions (velocity/atmosphere may differ
     from the shot-day trajectory).
```

The full set is `--zero-velocity`, `--zero-temperature`, `--zero-pressure`, `--zero-humidity`, and `--zero-altitude`, joined in later releases by `--zero-powder-temp`.

### 5.6.5 Powder Temperature Sensitivity

Ammunition performance changes with temperature. Cold ammo produces lower muzzle velocities; hot ammo shoots faster. This directly affects zero. A load that chronographs 2700 fps at 70°F might only make 2650 fps at 20°F, shifting the zero by several inches at long range.

The trajectory command supports powder temperature sensitivity with the `--use-powder-sensitivity`, `--powder-temp-sensitivity`, and `--powder-temp` flags (see Section 4.10). For zeroing, the key insight is: if your zero was established at one ambient temperature and you are shooting at another, account for the velocity change.

The crudest way to do that is to move the velocity by hand:

Listing 5.13: Accounting for cold-weather velocity loss

```
# Original zero: 70F ambient, 2700 fps
# Now shooting: 20F ambient, velocity drops ~50 fps
ballistics trajectory --velocity 2700 \
  --velocity-adjustment -50 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 800 \
  --temperature 20 --sight-height 1.5
```

### 5.6.6 Measured Powder Curves

The linear model assumes a constant fps per degree, and real powders are not linear. Temperature-stable powders flatten out at the ends of their range; others steepen sharply when hot. Since v0.22.2 you can hand the engine the points you actually chronographed instead of a slope you guessed:

Listing 5.14: A measured powder-temperature curve

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 600 \
  --sight-height 1.5 --temperature 20 \
  --powder-temp-curve "20:2610,40:2650,70:2700,100:2745" \
  --zero-temperature 70
```

The points are `TEMP:VELOCITY` pairs in °F and fps (or °C and m/s in metric). The muzzle velocity is interpolated between them and *clamped* at the endpoints—the engine will not extrapolate past data you did not measure. A curve supersedes `--powder-temp-sensitivity` when both are given.

**Powder Is Not Always at Air Temperature**

A round that has been in a warm chamber, or in a shirt pocket on a cold morning, is not at ambient. Since v0.22.5 the two temperatures are separate inputs: `--temperature` drives air density and nothing else, while `--powder-temp` is the temperature the curve is read at (defaulting to `--temperature` when you omit it). `--zero-powder-temp` does the same job on the zero day, defaulting to `--zero-temperature`. An explicit `--zero-velocity` still wins over both.

## 5.7 Exercises

1. **Compare zero distances.** Compute the zero angle for a .308 Win (168gr SMK, BC 0.462, 2700 fps) at 100 yards and 200 yards. How much does the zero angle change? What is the maximum ordinate for each?

```
ballistics zero --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 100 --sight-height 1.5

ballistics zero --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 200 --sight-height 1.5
```

2. **Find the MPBR.** Using the same load, compute the MPBR for an 8-inch vital zone. What is the optimal zero distance? Then compute the MPBR for a 4-inch vital zone (varmint hunting). How much shorter is the MPBR?

```
ballistics mpbr --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --vital-zone 8.0 --sight-height 1.5

ballistics mpbr --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --vital-zone 4.0 --sight-height 1.5
```

3. **Altitude zero shift.** Compute the trajectory for a 6.5 Creedmoor (140gr ELD-M, BC 0.610, 2710 fps) zeroed at 100 yards at two locations: sea level (59°F, 29.92 inHg) and Flagstaff, AZ (7000 ft, 80°F, 23.09 inHg). Compare the drop at 600 yards. By how many inches does the high-altitude trajectory differ?

```
ballistics trajectory --velocity 2710 \
  --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --auto-zero 100 \
  --max-range 600 --sight-height 2.0 \
  --temperature 59 --pressure 29.92

ballistics trajectory --velocity 2710 \
  --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --auto-zero 100 \
  --max-range 600 --sight-height 2.0 \
```

```
  --temperature 80 --pressure 23.09
```

4. **Sight height sensitivity.** How much does sight height affect the near-zero? Compute trajectories for the .308 Win load at 200-yard zero with sight heights of 1.5, 2.0, and 2.5 inches. Use `--sample-trajectory` with a 5-meter interval to find the near-zero crossing in each case.

## What's Next

A dope card built from the **`trajectory`** and **`zero`** commands assumes that every shot will perform exactly as modeled—same velocity, same BC, same wind. In the real world, nothing is that deterministic. In Chapter 6, we use the `monte-carlo` command to quantify the *dispersion* caused by shot-to-shot velocity variation, wind uncertainty, and BC variability, giving you honest confidence intervals instead of point predictions.

# Chapter 6

# Monte Carlo Simulation

Every number produced by the `trajectory` command is a lie—a beautiful, useful, precise lie. It tells you exactly where the bullet *would* go if your muzzle velocity were exactly 2700 fps, your BC were exactly `0.462`, and the wind were exactly 10 mph from exactly 90 degrees. In reality, your muzzle velocity varies shot to shot, your BC has uncertainty, the wind shifts between the time you read it and the time the bullet arrives, and your trigger pull is never perfectly consistent.

The `monte-carlo` command embraces this uncertainty. Instead of computing one trajectory, it computes *thousands*—each with slightly different inputs drawn from statistical distributions—and reports the resulting dispersion. The result is not a single impact point but a probability cloud that tells you how likely you are to hit, how large your group will be, and where the center of that group sits.

## 6.1 Why Monte Carlo Simulation?

A deterministic trajectory solver gives you the *expected value*—the center of the distribution. But shooting is a statistical process. What you actually need to know is:

- What is the probability that my first round hits a 10-inch plate at 800 yards?
- If I fire 5 rounds, what size group should I expect?
- How much does muzzle velocity variation degrade my accuracy at 1000 yards?
- Is wind uncertainty or velocity uncertainty the dominant source of dispersion at my target distance?

Monte Carlo simulation answers all of these questions by sampling from the input distributions and building a statistical picture of the output.

**Monte Carlo Method**

A *Monte Carlo simulation* is a computational technique that uses repeated random sampling to obtain numerical results. For ballistics, this means running thousands of trajectory computations, each with inputs perturbed by random amounts drawn from specified probability distributions. The aggregate results reveal the statistical properties of the output (mean, standard deviation, percentiles, hit probability).

## 6.2 The `monte-carlo` Command

The `monte-carlo` command extends the trajectory solver with statistical variation. Here is a basic invocation:

Listing 6.1: Basic Monte Carlo simulation for .308 Win

```
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 1000 \
  --velocity-std 15 --bc-std 0.005 \
  --wind-std 2 --angle-std 0.05 \
  --target-distance 600
```

This runs 1000 trajectory simulations for a .308 Win, 168gr SMK at a 600-yard target. Each simulation perturbs the muzzle velocity (standard deviation 15 fps), BC ($\sigma = 0.005$), wind speed ($\sigma = 2$ mph), and launch angle ($\sigma = 0.05$) by random amounts drawn from normal distributions. The command reports statistical summary results.

### 6.2.1 Required Flags

The core ballistic parameters are the same as for the `trajectory` command:

- `--velocity` — Base muzzle velocity (fps or m/s)
- `--bc` — Base ballistic coefficient
- `--mass` — Bullet mass (grains or grams)
- `--diameter` — Bullet diameter (inches or mm)

### 6.2.2 Monte Carlo–Specific Flags

- `--num-sims` (short: `-n`) — Number of simulations to run. Default: 1000.

- `--velocity-std` — Standard deviation of muzzle velocity variation. Default: 1.0.
- `--angle-std` — Standard deviation of launch angle variation (degrees). Default: 0.1. This models inherent rifle/shooter precision.
- `--bc-std` — Standard deviation of BC variation. Default: 0.01. This models lot-to-lot and bullet-to-bullet BC variation.
- `--wind-std` — Standard deviation of wind speed variation. Default: 1.0. This models wind uncertainty—the shooter's imperfect wind read.
- `--wind-speed` — Base (mean) wind speed. Default: 0.
- `--wind-direction` — Base wind direction (degrees). Default: 0.
- `--target-distance` — Distance to target for hit probability calculation. Optional but recommended.
- `--target-radius` — Radius of the hit zone around the point of aim, measured in the target plane (yards or meters). Default: 0.3 m. A simulation counts as a hit when it lands inside this radius.

### 6.2.3 Output Modes

The `--output` flag (short: `-o`) controls the output format:

- `--output summary` — Formatted table with key statistics (default)
- `--output full` — JSON with complete statistical results
- `--output statistics` — CSV row of min/max/mean/std for range and velocity

Listing 6.2: Monte Carlo output in JSON format

```
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 5000 \
  --velocity-std 15 --bc-std 0.005 \
  --wind-std 3 --angle-std 0.05 \
  --target-distance 800 \
  --output full
```

The JSON output includes the mean range, standard deviation, mean impact velocity, CEP, and hit probability—suitable for programmatic analysis or integration with other tools.

**"Mean Range" Means Ground Impact (0.21.5)**

Mean Range is the average distance at which the simulated bullets reach the ground, and it is computed over every sample. Before v0.21.5 the simulations ran from a muzzle at ground level against a −100 m floor, so each one flew until the integrator's range cap and Mean Range simply reported that cap—and the statistic moved when you changed `--target-distance`, because short samples were skipped. Monte Carlo now uses the same bore-height and ground convention as `trajectory` (1.5 m), so Mean Range agrees with a `trajectory` run of the same load and no longer depends on where the target is.

## 6.3 Input Distributions: What Varies and by How Much

The quality of a Monte Carlo simulation depends entirely on the quality of its input distributions. Garbage in, garbage out. This section discusses realistic values for each source of variation.

Each input parameter is modeled as a *normal (Gaussian) distribution* centered on the nominal value with a specified standard deviation. The engine draws random samples from these distributions using a standard random number generator. For a normal distribution, approximately 68% of samples fall within $\pm 1\sigma$, 95% within $\pm 2\sigma$, and 99.7% within $\pm 3\sigma$.

**Standard Deviation ($\sigma$)**

The standard deviation measures the spread of a distribution. For a normal distribution centered at $\mu$:

$$P(\mu - \sigma \leq x \leq \mu + \sigma) = 0.683 \tag{6.1}$$

In ballistic terms, if your muzzle velocity has $\sigma = 15$ fps around a mean of 2700 fps, then 68% of your shots will be between 2685 and 2715 fps, and 95% will be between 2670 and 2730 fps.

## 6.4 Muzzle Velocity Variation (SD and ES)

Muzzle velocity variation is typically characterized by two numbers from chronograph data:

- **Standard Deviation (SD)**: The statistical spread of velocities. A well-developed load might have SD of 8–15 fps; a mediocre load might show 20–30 fps.
- **Extreme Spread (ES)**: The difference between the fastest and slowest shots in a string. For a 10-shot string, ES is typically 3–4 times the SD.

The `--velocity-std` flag takes the standard deviation in fps (imperial) or m/s (metric):

Listing 6.3: Modeling velocity variation

```
# Well-tuned handload (SD = 8 fps)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 2000 --velocity-std 8 \
  --target-distance 1000

# Factory ammunition (SD = 20 fps)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 2000 --velocity-std 20 \
  --target-distance 1000
```

Compare the CEP and dispersion between these two runs to see the precision advantage of low-SD handloads at long range.

**Converting ES to SD**

If you only have extreme spread data, a rough conversion for a 10-shot string is: SD ≈ ES/3.1. For a 20-shot string: SD ≈ ES/3.7. These factors come from the expected ratio of the range to the standard deviation for samples from a normal distribution.

### 6.4.1 How Velocity Variation Affects Impact

Velocity variation primarily affects *vertical* dispersion. A faster shot drops less; a slower shot drops more. The effect scales with distance because the time-of-flight difference compounds. At 100 yards, a 15 fps SD produces negligible vertical spread. At 1000 yards, the same SD might produce 3–4 inches of vertical dispersion—enough to miss a small target.

## 6.5 BC Uncertainty

Ballistic coefficients published by bullet manufacturers are averages measured across a sample of bullets. Individual bullets vary due to manufacturing tolerances: slight differences in ogive profile, base concentricity, weight distribution, and tip geometry all affect BC.

Typical BC variation:

- **Premium match bullets** (SMK, Berger, Lapua Scenar): $\sigma_{BC} \approx 0.003$–$0.008$ (1–2% of nominal)
- **Hunting bullets**: $\sigma_{BC} \approx 0.008$–$0.015$ (2–3% of nominal)

- **Military ball ammunition**: $\sigma_{BC} \approx 0.010$–$0.020$ (3–5% of nominal)

Listing 6.4: Modeling BC uncertainty

```
# Premium match bullet (low BC variation)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 2000 --bc-std 0.005 \
  --velocity-std 10 --target-distance 800

# Hunting bullet (higher BC variation)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 2000 --bc-std 0.015 \
  --velocity-std 10 --target-distance 800
```

BC variation affects both vertical *and* horizontal dispersion. A bullet with lower BC retains less velocity, drops more, and drifts more in the wind. Unlike velocity variation (which is purely vertical), BC variation compounds with wind to produce a correlated horizontal effect.

## 6.6 Wind Variability

Wind is the dominant source of lateral dispersion at long range, and it is also the hardest to measure accurately. The wind the bullet encounters during its flight may differ from the wind you estimated at the firing position. Between you and the target, wind speed and direction change constantly.

The `--wind-std` flag models this uncertainty. A standard deviation of 2 mph means you think your wind call is accurate to within $\pm 2$ mph about 68% of the time—a realistic estimate for an experienced wind caller.

Listing 6.5: Wind uncertainty modeling

```
# Experienced wind caller (good reads)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 2000 --wind-speed 10 \
  --wind-direction 90 --wind-std 2 \
  --velocity-std 10 --target-distance 800

# Novice wind caller (poor reads)
ballistics monte-carlo \
```

```
--velocity 2700 --bc 0.462 \
--mass 168 --diameter 0.308 \
--num-sims 2000 --wind-speed 10 \
--wind-direction 90 --wind-std 5 \
--velocity-std 10 --target-distance 800
```

**Wind Uncertainty is the Dominant Factor**

At long range (beyond 600 yards), wind uncertainty almost always produces more dispersion than velocity or BC uncertainty combined. A 3 mph error in wind reading with a .308 Win at 800 yards produces roughly 10 inches of lateral error—far more than the 2–3 inches of vertical error from a 15 fps velocity SD.

### 6.6.1 Practical Wind SD Values

- **1–2 mph**: Expert wind caller in steady conditions
- **2–3 mph**: Experienced shooter in moderate conditions
- **3–5 mph**: Average shooter or tricky wind conditions (switching, gusting)
- **5–8 mph**: Difficult wind, terrain effects, or minimal wind reading experience

## 6.7 Convergence: How Many Iterations Are Enough?

Monte Carlo results are themselves estimates, and their precision improves with more iterations. The standard error of a Monte Carlo estimate of the mean decreases as $1/\sqrt{N}$:

$$\text{SE}_{\text{mean}} = \frac{\sigma}{\sqrt{N}} \tag{6.2}$$

This means doubling the number of iterations improves the estimate by only a factor of $\sqrt{2} \approx 1.41$. There are diminishing returns.

### 6.7.1 Practical Guidance

- **100 iterations**: Quick sanity check. CEP estimate has about 10% error.
- **1000 iterations**: Good for most purposes. CEP estimate has about 3% error. This is the default.
- **5000 iterations**: Precise estimates. CEP error below 1.5%.

- **10000+ iterations**: Highly precise. Useful for publication-quality analysis or when computing small hit probabilities.

Listing 6.6: Convergence study: running at different iteration counts

```
# Quick check (100 sims)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 100 --velocity-std 15 \
  --wind-std 3 --target-distance 800

# Production run (5000 sims)
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 5000 --velocity-std 15 \
  --wind-std 3 --target-distance 800
```

Run both and compare the CEP values. They should be close; if they differ by more than 10%, you need more iterations.

**Performance**

BALLISTICS-ENGINE uses Rayon for parallel computation. On a modern multi-core CPU, 5000 Monte Carlo trajectories complete in under a second. Don't be stingy with iterations—the computational cost is negligible.

## 6.8 Reading the Results: CEP, Confidence Ellipses, and Extreme Spread

The Monte Carlo output reports several statistical measures that describe the dispersion pattern.

### 6.8.1 CEP (Circular Error Probable)

CEP is the radius of a circle, centered on the mean point of impact, within which 50% of shots fall. It is the single most useful measure of system precision because it combines vertical and horizontal dispersion into one number.

**Circular Error Probable**

Given $N$ impact points $(x_i, y_i)$, the CEP is computed as follows:

1. Calculate the mean point of impact: $(\bar{x}, \bar{y})$.
2. Compute the distance from each point to the mean: $d_i = \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}$.
3. Sort the distances and find the median (50th percentile). This is the CEP.

This definition is implemented in the `calculate_cep()` function in `src/monte_carlo.rs`. The function takes vectors of wind drift and drop values, computes distances from the centroid, and returns the median distance.

**Monte Carlo Numbers Moved in 0.22.0**

Four Monte Carlo defects were corrected in v0.22.0, and every one of them changes reported results:

- **CEP** was computed as $\sigma_{\text{range}} \times 1.1774$—the Rayleigh factor applied to the standard deviation of *ground-impact range*, which is not a target-plane radius at all. It was wrong by as much as a factor of 100. CEP is now the median target-plane radial miss, exactly as the definition above says.
- **Vertical dispersion was sampled twice.** Muzzle-angle jitter and an independent "pointing error" of the same magnitude were both applied, inflating vertical spread by about 41%. Your `--angle-std` now means what it says.
- **Shots that fell short were counted as hits**, because a downrange shortfall was being compared against the target-plane hit radius.
- **`max_range` is now sized to the target**, so a simulation of a shot past 1000 m no longer truncates silently at the integrator's default.

The FFI and WASM Monte Carlo paths were brought onto the CLI's ground and bore convention at the same time—they had been reporting the integrator cap with a standard deviation of zero—and the FFI stopped discarding its `atmosphere` argument and running every sample at ICAO sea level.

For practical interpretation:

- A CEP of 5 inches at 600 yards means half your shots will hit within a 10-inch circle.
- To convert CEP to "group size" as commonly discussed, multiply by about 2.5 for a 90% circle or 3.4 for a 99% circle (assuming a circular normal distribution).

### 6.8.2 Confidence Ellipses

Because vertical and horizontal dispersion are usually different magnitudes (wind dominates horizontal; velocity dominates vertical), the shot pattern is typically an *ellipse*, not a circle. The engine computes the 95% confidence ellipse using the eigendecomposition of the $2 \times 2$ covariance matrix:

$$\mathbf{C} = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy}^2 \end{bmatrix} \tag{6.3}$$

The eigenvalues $\lambda_1$ and $\lambda_2$ of this matrix define the semi-axes of the ellipse. The 95% confidence ellipse uses a $\chi^2$ scale factor of $\sqrt{5.991}$:

$$a = \sqrt{\lambda_1} \times \sqrt{5.991} \qquad \text{(semi-major axis)} \tag{6.4}$$

$$b = \sqrt{\lambda_2} \times \sqrt{5.991} \qquad \text{(semi-minor axis)} \tag{6.5}$$

The rotation angle of the ellipse indicates the correlation between vertical and horizontal errors. A tilt indicates that the wind component is correlated with the BC/velocity component (which it is, since lower BC bullets also drift more).

The `calculate_confidence_ellipse()` function in `src/monte_carlo.rs` returns the center coordinates, semi-major axis, semi-minor axis, and rotation angle in degrees.

### 6.8.3 Extreme Spread

The extreme spread (ES) of the Monte Carlo run is the distance between the two farthest impact points. The `statistics` output mode reports the minimum and maximum values for range and velocity, from which extreme spread can be derived. ES is the "worst case" measure—it tells you the largest group you might see in that number of shots.

### 6.8.4 Hit Probability

When you specify `--target-distance`, the engine reports the fraction of simulated shots that land inside a hit zone drawn around the point of aim *in the target plane*. The radius of that zone is `--target-radius`, which defaults to $0.3\ \mathrm{m}$:

Listing 6.7: Hit probability at 800 yards

```
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 5000 --velocity-std 15 \
  --bc-std 0.005 --wind-speed 10 \
  --wind-direction 90 --wind-std 3 \
  --angle-std 0.05 \
  --target-distance 800 --target-radius 0.5
```

A shot that falls short of the target plane entirely is recorded as a definite miss, so an unreachable target correctly yields a hit probability near zero rather than a spurious 100%.

**Hit Probability Is a Target-Plane Radius (0.21.5)**

Through v0.21.4 hit probability meant something different on the command line: the fraction of simulations whose *ground-impact range* came within 1 meter of `--target-distance`. That is a range-precision notion, not a hit notion—it read 0% for any target closer than the impact range, and the FFI used a third definition of its own. As of v0.21.5 every surface uses one method, `MonteCarloResults::hit_probability(radius)`, and the hit zone is sized by `--target-radius` instead of a hardcoded meter. If you recorded first-round hit numbers from an earlier release, run them again.

**Sizing the Hit Zone**

Set `--target-radius` to the radius of the thing you actually intend to hit: 0.15 m for a 12-inch plate, 0.2 m for the vital zone of a pronghorn, 0.05 m for an F-class ten ring. The default 0.3 m is a 24-inch circle—generous for most purposes, which is why an unset radius tends to flatter your load.

## 6.9 WEZ: Hit Probability Across a Whole Range Band

A single hit probability at a single distance answers one question. The question a shooter actually has is *how far out can I still expect to hit this thing?*—and the answer is a curve, not a number. Since v0.25.0, `--wez` runs the whole sweep.

**Weapon Employment Zone**

The *Weapon Employment Zone* (WEZ) is the band of ranges over which a weapon system achieves an acceptable probability of hit against a stated target, given a stated set of input uncertainties. It is the standard way military and competitive shooters express effective range, because it makes both the target size and the error budget explicit.

Listing 6.8: A WEZ sweep against an 18″ × 30″ silhouette

```
# Solve the zero angle first -- monte-carlo takes a bore angle,
# not a zero distance.
ballistics zero --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --target-distance 600 --sight-height 1.5

ballistics monte-carlo --velocity 2700 --bc 0.462 \
```

```
  --mass 168 --diameter 0.308 --angle 0.3229 \
  --num-sims 1000 --wez --target-size 18x30 \
  --wez-start 300 --wez-end 900 --wez-step 200 \
  --velocity-std 15 --wind-std 1 --wind-call-error 3
```

Listing 6.9: WEZ output

```
WEZ sweep: 1000 sims/step, wind call 3.00 mph + wind std 1.00 mph (quadrature) = 3.16
     mph effective
+------------+----------+---------------+-----------+-----------+-----------+
| Range ( yd) |  P(hit)  | Dominant      | Wind call |  MV SD    | Other/grp |
+------------+----------+---------------+-----------+-----------+-----------+
|      300.0 |     9.0% | other         |      0.0% |      0.0% |    100.0% |
|      500.0 |    14.2% | other         |      0.0% |      0.1% |     99.9% |
|      700.0 |     6.3% | other         |      0.0% |      0.2% |     99.8% |
|      900.0 |     0.1% | n/a           |      0.0% |      0.0% |      0.0% |
+------------+----------+---------------+-----------+-----------+-----------+
```

Read the sweep from the right-hand columns first. Each row attributes its miss variance to a source, and names the dominant one. In this run the answer is `other`—the angular dispersion of the rifle and shooter itself, which at these ranges swamps both the wind call and the velocity spread. That is a useful and slightly humbling result: tightening your wind reading will not help this system until the group does.

**`--wind-call-error` vs. `--wind-std`**

These are two different uncertainties and the sweep keeps them apart. `--wind-std` is the *physical* variability of the wind—gustiness, the air genuinely doing something different from shot to shot. `--wind-call-error` is *your* error in estimating it. They compose in quadrature:

$$\sigma_{\text{eff}} = \sqrt{\sigma_{\text{wind}}^2 + \sigma_{\text{call}}^2} \tag{6.6}$$

which is the 3.16 mph the header reports from a 1 mph physical spread and a 3 mph call error.

**Dominant-Source Labels Are snake_case (0.27.0)**

The summary table and the statistics CSV used to print display spellings for the dominant source—"wind call", "MV SD", "other/group"—while the `-o full` JSON used `wind_call`, `mv_sd` and `other`. As of v0.27.0 every surface prints the snake_case spellings, so a script can key off the same token wherever it reads them. The `-o full` JSON contract itself is unchanged.

`--target-size` takes `WIDTHxHEIGHT` in inches (cm in metric) for a rectangle centered on the point of aim, or a single number for a circular radius. `--wez-start`, `--wez-end`, and `--wez-step` bound the sweep. The per-step Monte Carlo is seeded reproducibly, so re-running the same sweep gives the same answer.

## 6.10 Practical Example: Predicting First-Round Hit Probability

Let's work through a realistic example. A precision rifle competitor needs to engage a 12-inch steel plate at 800 yards under moderate wind conditions. The question: *what is the probability of a first-round hit?*

### 6.10.1 Setup

- Rifle: 6.5 Creedmoor, 140gr ELD-M, BC `0.610` (G7), 2710 fps
- Muzzle velocity SD: 12 fps (quality handload)
- BC variation: $\sigma = 0.005$ (premium match bullet)
- Wind: 8 mph full-value crosswind, estimated $\pm 2$ mph uncertainty
- Shooter precision: $\sigma = 0.05$ (sub-0.5 MOA rifle and shooter)

Listing 6.10: First-round hit probability at 800 yards

```
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 5000 \
  --velocity-std 12 --bc-std 0.005 \
  --wind-speed 8 --wind-direction 90 \
  --wind-std 2 --angle-std 0.05 \
  --target-distance 800
```

### 6.10.2 Interpreting the Results

The summary output might show:

- **CEP**: approximately 4–5 inches at 800 yards
- **Mean impact velocity**: approximately 1720 fps

A CEP of 5 inches means 50% of shots fall within a 10-inch circle. For a 12-inch plate, the first-round hit probability is roughly 60–70%. Let's verify by looking at the confidence ellipse dimensions: the

semi-major axis (likely 8–10 inches, driven by wind uncertainty) and the semi-minor axis (3–4 inches, driven by velocity/BC).

### 6.10.3 Sensitivity Analysis

Now let's see what dominates. Run the simulation three times, varying one parameter at a time:

Listing 6.11: Wind as the dominant uncertainty source

```
# Wind only: zero velocity/BC variation
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 2000 --velocity-std 0 --bc-std 0 \
  --wind-speed 8 --wind-direction 90 \
  --wind-std 2 --angle-std 0 \
  --target-distance 800

# Velocity only: zero wind/BC variation
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 2000 --velocity-std 12 --bc-std 0 \
  --wind-speed 0 --wind-std 0 --angle-std 0 \
  --target-distance 800

# BC only: zero wind/velocity variation
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 2000 --velocity-std 0 --bc-std 0.005 \
  --wind-speed 0 --wind-std 0 --angle-std 0 \
  --target-distance 800
```

You will find that wind uncertainty dominates the dispersion budget by a factor of 3–5× over velocity and BC combined. This is the key insight of Monte Carlo analysis: **improving your wind call is almost always more valuable than improving your muzzle velocity consistency at long range**.

### 6.10.4 What-If: Factory Ammo vs. Handload

Listing 6.12: Factory ammo vs. handload comparison

```
# Handload: low SD, low BC variation
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
```

```
  --mass 140 --diameter 0.264 \
  --num-sims 3000 --velocity-std 10 --bc-std 0.003 \
  --wind-speed 8 --wind-direction 90 \
  --wind-std 3 --angle-std 0.05 \
  --target-distance 1000

# Factory match ammo: higher SD, more BC variation
ballistics monte-carlo \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 3000 --velocity-std 25 --bc-std 0.010 \
  --wind-speed 8 --wind-direction 90 \
  --wind-std 3 --angle-std 0.05 \
  --target-distance 1000
```

At 1000 yards, the factory ammo shows a noticeably larger vertical component in the confidence ellipse. But the wind-driven horizontal component is similar for both, because both experience the same wind uncertainty. The total CEP is perhaps 20–30% larger for the factory ammo. Is that difference worth the hours of handloading? That's your decision—but now you have the data to make it.

## 6.11 Advanced Considerations

### 6.11.1 Correlated Inputs

The current Monte Carlo implementation treats all input variations as *independent*. In reality, some correlations exist: for example, temperature affects both muzzle velocity (through powder burn rate) and air density (through drag). The engine does not model these correlations explicitly, but you can approximate their effect by running separate simulations with different atmospheric conditions and combining the results.

### 6.11.2 Non-Gaussian Distributions

The engine assumes all input variations follow normal (Gaussian) distributions. Some real-world processes are not Gaussian—for example, wind gusts may follow a Weibull distribution, and muzzle velocity from factory ammo may be bimodal (two lots mixed). For now, the Gaussian approximation is reasonable for most applications, and you can adjust the standard deviation to account for heavier tails.

### 6.11.3 Computational Performance

Each Monte Carlo iteration runs a complete trajectory solve. The engine uses the fast trajectory integration path (implemented in `src/fast_trajectory.rs`) for Monte Carlo runs, which optimizes for speed by using a fixed-step RK4 integrator. On a modern CPU, 5000 iterations for a 1000-yard trajectory complete in under 2 seconds.

For convergence verification, run the simulation twice with different random seeds (different invocations produce different seeds) and compare the CEP values. They should agree within the expected Monte Carlo sampling error.

## 6.12 Exercises

1. **Convergence test.** Run the 6.5 Creedmoor scenario from Section 6.10 at 100, 500, 1000, 5000, and 10000 iterations. Record the CEP from each. At what iteration count does the CEP stabilize to within 5%?

2. **Velocity SD sweep.** Using the .308 Win load (168gr SMK, BC `0.462`, 2700 fps) at 800 yards with 10 mph crosswind and 2 mph wind SD, run Monte Carlo simulations with velocity SDs of 5, 10, 15, 20, and 30 fps. Plot CEP vs. velocity SD. Is the relationship linear?

```
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --num-sims 3000 --velocity-std 5 \
  --wind-speed 10 --wind-direction 90 \
  --wind-std 2 --target-distance 800
```

3. **Dominance analysis.** For a .338 Lapua Magnum (300gr Berger Hybrid, BC `0.818` G7, 2725 fps) at 1500 yards with a 5 mph crosswind, determine whether wind uncertainty ($\sigma = 2$ mph) or velocity uncertainty ($\sigma = 15$ fps) produces more dispersion. Run isolating simulations as shown in Section 6.10.

4. **Cartridge comparison.** Compare the CEP at 1000 yards of a .308 Win (168gr SMK, BC `0.462` G1, 2700 fps, vel SD 15 fps) against a 6.5 Creedmoor (140gr ELD-M, BC `0.610` G7, 2710 fps, vel SD 12 fps), both with 3 mph wind SD and 10 mph base crosswind. Which system is more accurate? By how much?

## What's Next

The Monte Carlo simulation revealed that the accuracy of your BC value matters—both its nominal value and its uncertainty. But how do you know if the BC printed on the box is correct for *your* bullets, in *your* rifle, in *your* conditions? In Chapter 7, we use the `estimate-bc` and `true-velocity` commands to derive a BC from actual field data, closing the loop between theory and observed performance.

# Chapter 7

# BC Estimation & Truing

The ballistic coefficient printed on a bullet box is a starting point, not gospel. It was measured on a sample of bullets, under specific conditions, using a specific drag model, at a specific velocity range. Your bullets, your rifle, your conditions may produce a different effective BC. The difference between the *published* BC and the *true* BC of your system is the single largest source of error in computational ballistics at extended range.

This chapter covers two commands that close the gap between published data and reality: `estimate-bc`, which derives a BC from observed trajectory data, and `true-velocity`, which finds the effective muzzle velocity that matches your observed drop at range. Together, they let you *true* your ballistic solution to the real world.

## 7.1 Why True Your BC?

Consider a .308 Win, 168gr Sierra MatchKing with a published G1 BC of `0.462`. You chronograph your load at 2700 fps, build a dope card, and walk up to 800 yards. But when you fire, the bullet impacts 3 inches lower than predicted. What went wrong?

Several factors could be at play:

1. **BC is lower than published.** Your particular lot of bullets may have slightly different ogive geometry than the lot measured by the manufacturer.
2. **Chronograph error.** Optical chronographs can read 10–30 fps high or low depending on lighting conditions.
3. **Drag model mismatch.** The published BC assumes G1, but your bullet's drag curve may not match the G1 standard projectile well at transonic velocities.

4. **Atmospheric model error.** If you didn't account for altitude, temperature, or humidity correctly, the predicted air density was wrong.

Truing addresses all of these. Instead of guessing which input is wrong, you adjust the *effective BC* or *effective velocity* until the computed trajectory matches your observed data. The result is a system-level calibration that absorbs all sources of error into a single corrected parameter.

**Truing**

*Truing* is the process of adjusting computational inputs (BC, muzzle velocity, or both) so that the predicted trajectory matches observed data from actual shooting. A "trued" solution accounts for all real-world factors that the raw inputs do not capture.

**SAFETY: Truing Does Not Replace Chronograph Data**

Truing adjusts *effective* parameters, not physical ones. If your trued velocity is 50 fps lower than your chronograph reading, it does not mean your chronograph is wrong by 50 fps—it may also reflect BC error, atmospheric model error, or scope tracking imprecision. Never use trued velocity to estimate chamber pressure or to extrapolate load data. Always use actual chronograph readings for load development.

## 7.2 The `estimate-bc` Command

The `estimate-bc` command takes observed drop data and computes the BC that would produce it. The oldest form of the command takes exactly two distances:

Listing 7.1: Estimating BC from observed drops

```
ballistics estimate-bc \
  --velocity 2700 \
  --mass 168 --diameter 0.308 \
  --distance1 300 --drop1 8.5 \
  --distance2 600 --drop2 85.5 \
  --zero-range 200 --sight-height 1.5
```

**Input Units for `estimate-bc`**

Inputs follow `--units` like every other command: velocity in fps, mass in grains, diameter and *drops* in inches, distances in yards (metric: m/s, grams, mm, meters, meters). The `--output` flag supports `table`, `json`, and `csv`.

**Drops Are Inches, Not Yards (0.22.0)**

Through v0.21.x the native command parsed `--drop1`/`--drop2` as *yards* while the WASM build parsed the same numbers as inches—a 36× unit mismatch between two surfaces of the same engine. Both now read inches. Any BC you fitted on the command line before v0.22.0 was fitted against drops 36 times too large; refit it.

The command prints one row per fit it was able to perform:

Listing 7.2: Estimate-BC output

```
BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G1     drop (2 pts)                0.462     0.02 in
  G7     drop (2 pts)                0.232     0.07 in
```

The **Fit RMS** column is the residual: the root-mean-square difference between your observed drops and the drops the fitted BC reproduces. Hundredths of an inch, as here, means the model and the data agree. Several inches means something in the setup does not match the data—the wrong zero range, the wrong atmosphere, or a bullet that genuinely does not follow the drag family you asked for.

### 7.2.1 Fitting G1 and G7 Together

A bullet's G1 BC and its G7 BC are different numbers describing the same projectile, and since v0.22.8 the command reports both by default. Use `--drag-model` to narrow it:

Listing 7.3: Fitting a single drag family

```
ballistics estimate-bc \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --drag-model g7 \
  --data "300,8.5;400,24.6;500,49.7;600,85.5;700,134.1" \
  --zero-range 200 --sight-height 1.5
```

`--drag-model` accepts `g1`, `g7`, `both` (the default), or any single family—`g2`, `g5`, `g6`, `g8`, `gi`, `gs`, `ra4`.

### 7.2.2 More Than Two Points

Two points is the minimum, not the goal. `--data` takes an $n$-point series as `"dist,drop;dist,drop;..."`, which is what a real dope card looks like:

Listing 7.4: Fitting a five-point dope card

```
ballistics estimate-bc \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --data "300,8.5;400,24.6;500,49.7;600,85.5;700,134.1" \
  --zero-range 200 --sight-height 1.5
```

Listing 7.5: Five-point fit, both families

```
BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G1     drop (5 pts)                0.462     0.03 in
  G7     drop (5 pts)                0.233     0.09 in
```

### 7.2.3 Fitting Velocity Instead of Drop

If you have downrange velocity readings—a LabRadar trace, a second chronograph—feed them with `--velocity-data` in the same `"dist,vel;..."` form. A velocity fit does not depend on the zero, the sight height, or the launch angle, so it is often the sounder of the two bases:

Listing 7.6: Drop and velocity fits side by side

```
ballistics estimate-bc \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --data "300,8.5;400,24.6;500,49.7;600,85.5;700,134.1" \
  --velocity-data "300,2139;500,1807;700,1513" \
  --zero-range 200 --sight-height 1.5
```

Supplying both bases with the default `--drag-model both` produces all four variants:

Listing 7.7: All four fit variants

```
BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G1     drop (5 pts)                0.462     0.03 in
  G1     velocity (3 pts)            0.462     0.36 fps
  G7     drop (5 pts)                0.233     0.09 in
  G7     velocity (3 pts)            0.234     5.56 fps
```

Four fits that agree is the strongest evidence you will get that the BC is real.

### 7.2.4 Zeroed Cards and the Conditions They Were Shot In

Two things about a dope card matter enormously to the fit, and until v0.22.9 the command could not be told either of them.

The first is the zero. A dope card records drop *below the line of sight of a zeroed rifle*, so it has a point where drop is zero. A flat-fire, bore-referenced model can never produce that, and asking it to drives the fit into nonsense—typically straight into the BC search ceiling. Pass `--zero-range` (and `--sight-height`, default 2″/50 mm) and the drop is fitted the way the card was measured. Omit it and drop stays bore-referenced, which is right for a ladder shot from a fixed bore angle and wrong for anything else.

The second is the air. A card shot at 7000 feet on a hot day did not encounter the standard atmosphere, and fitting it against one moves the BC to absorb the difference:

Listing 7.8: Fitting a card to the conditions it was shot in

```
ballistics estimate-bc \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --data "300,8.5;400,24.6;500,49.7;600,85.5;700,134.1" \
  --zero-range 200 --sight-height 1.5 \
  --temperature 88 --pressure 23.4 \
  --humidity 20 --altitude 6800
```

**An Unreliable Fit Now Says So**

When a fit runs all the way to the BC search limit—`1.200` for G1, `0.700` for G7—the answer is not a BC, it is the edge of the search. Since v0.22.9 the command flags that case rather than returning the boundary value silently:

```
  G1     drop (3 pts)              1.200    15.68 in   △ UNRELIABLE (hit BC
    limit)
  G7     drop (3 pts)              0.700    15.15 in   △ UNRELIABLE (hit BC
    limit)
```

The JSON and CSV forms carry the same information in a `reliable` field. The command also warns when the drop data looks zeroed—one point near zero drop—and no `--zero-range` was supplied, which is by far the most common cause.

## 7.3 Estimating BC from Two-Velocity Measurements

The most direct way to measure BC is to record the bullet's velocity at two different downrange distances, typically using a muzzle chronograph and a downrange chronograph (or a Doppler radar system like a LabRadar).

The physics is straightforward. The BC relates the actual drag to the reference drag of the standard projectile:

$$\text{BC} = \frac{i}{C} = \frac{\text{sectional density}}{\text{form factor}} \tag{7.1}$$

Given two measured velocities $v_1$ and $v_2$ at distances $d_1$ and $d_2$, the engine can solve for the BC that produces the observed velocity decay. The function `estimate_bc_from_trajectory()` in the `ballistics_engine` library uses an iterative approach: it adjusts BC until the trajectory computed with that BC matches the observed data points.

The practical procedure:

1. Measure muzzle velocity $v_0$ with a chronograph at the firing position.
2. Place a second chronograph (or use a Doppler radar) to capture velocity $v_1$ at distance $d_1$ (e.g., 300 yards).
3. Optionally capture velocity $v_2$ at distance $d_2$ (e.g., 600 yards).
4. Convert velocities to drops using the relationship between velocity loss and trajectory arc.
5. Feed the data into `estimate-bc`.

**Use Drop Data When Possible**

Most shooters do not have a downrange chronograph. Instead, measure the *drop* at known distances by recording the actual scope correction needed to hit center. This is easier to measure accurately and is what the `estimate-bc` command is designed for.

## 7.4 Truing BC to Observed Drop at Range

The `true-velocity` command takes a different approach. Instead of estimating BC directly, it finds the *effective muzzle velocity* that produces a trajectory matching your observed drop at a specific range. This is operationally simpler: you shoot at a known distance, record the scope correction in MIL, and feed that into the command.

Listing 7.9: Truing velocity to observed drop

```
ballistics true-velocity \
  --measured-drop 5.8 \
  --range 600 \
  --bc 0.462 --drag-model g1 \
  --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5 \
  --temperature 72 --pressure 29.45 \
  --humidity 45
```

Here, you observed 5.8 MIL of drop at 600 yards. The command searches for the muzzle velocity that produces exactly 5.8 MIL of drop at 600 yards, given your BC, drag model, zero distance, and atmospheric conditions.

### 7.4.1 How It Works

The `calculate_true_velocity_local()` function in `src/main.rs` uses binary search between 1500 fps and 4500 fps:

1. Pick a candidate velocity $v_{\text{test}}$ at the midpoint of the current search interval.
2. Run a full trajectory at $v_{\text{test}}$ with the specified BC, zero distance, and atmospheric conditions.
3. Compute the drop in MIL at the target range.
4. If the computed drop exceeds the measured drop, the bullet is too slow (too much drop); increase the velocity bound.
5. If the computed drop is less than measured, the bullet is too fast; decrease the velocity bound.
6. Repeat until the drop error is less than 0.01 MIL (approximately 0.04 inches at 100 yards).

The algorithm converges in approximately 20–30 iterations, with each iteration running a complete trajectory solve.

### 7.4.2 Where the Chronograph Actually Reads

A chronograph does not measure velocity at the muzzle. Optical units sit ten to fifteen feet downrange, and a Doppler unit reports from wherever you told it to. That gap costs a few fps—small, but it is a systematic bias, and truing exists precisely to chase systematic biases.

Since v0.29.0, `--chrono-distance` tells the engine where the reading was taken so it can be corrected back to a true muzzle velocity:

Listing 7.10: Correcting a downrange chronograph reading

```
ballistics true-velocity \
  --measured-drop 5.8 --range 600 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --chrono-velocity 2700 --chrono-distance 12 \
  --zero-distance 100 --sight-height 1.5
```

### 7.4.3 Optional: Chronograph Comparison

If you provide `--chrono-velocity`, the command also reports the difference between the trued velocity and your chronograph reading. This is diagnostic: a large discrepancy suggests either chronograph error or BC error (or both).

Listing 7.11: Truing with chronograph comparison

```
ballistics true-velocity \
  --measured-drop 5.8 \
  --range 600 \
  --bc 0.462 --drag-model g1 \
  --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5 \
  --chrono-velocity 2700
```

If the trued velocity comes back as 2665 fps while your chronograph said 2700 fps, the system is behaving as if your velocity is 35 fps lower than measured. This could be real (chronograph error) or it could be that your BC is slightly lower than `0.462`—the trajectory effect is the same.

### 7.4.4 Confidence Levels

The output includes a confidence indicator:

- **High**: Converged with drop error $< 0.005$ MIL. The trued velocity is very reliable.
- **Medium**: Converged with drop error $< 0.01$ MIL. Adequate for practical use.
- **Low**: Did not converge within tolerance. Check your inputs—a low confidence result often means the measured drop is inconsistent with the other inputs.

**Low Confidence Results**

A low-confidence result from `true-velocity` usually means one of your inputs is significantly wrong: the range may be incorrect, the measured drop may have a sign error, or the BC and drag model may be grossly mismatched. Before trusting a low-confidence result, verify your inputs carefully.

### 7.4.5 Fitting Velocity and BC Together

Truing a single observation can only ever move one parameter. If the drop at 600 yards is off, the command has no way to know whether your velocity is low or your BC is optimistic—it simply absorbs the whole discrepancy into velocity, and you get a "muzzle velocity" that is really a fudge factor.

Since v0.25.0, `--observed` lets you supply the rest of your dope card and fit both parameters at once against the real forward model. Each observation is `RANGE:DROP`, the flag is repeatable, and `--measured-drop`/`--range` count as the first observation:

Listing 7.12: Joint MV and BC truing from three observations

```
ballistics true-velocity \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --measured-drop 9.016 --range 900 \
  --observed "1000:11.116" \
  --observed "1200:16.161" \
  --zero-distance 200 --sight-height 1.5 \
  --drop-unit mil
```

Listing 7.13: A joint fit that recovers both parameters

```
=== VELOCITY + BC TRUING (multi-observation) ===

  Fitted muzzle velocity:    2649.9 fps
  Fitted BC:                 0.4621  (input 0.4620)

  Range (yd)  Observed (mil)  Predicted (mil)   Resid (mil)
  ------------------------------------------------------------
       900.0           9.016           9.016        +0.000
      1000.0          11.116          11.116        -0.000
      1200.0          16.161          16.161        +0.000
  ------------------------------------------------------------
  RMS residual: 0.000 mil   |   iterations: 5

  Joint MV+BC fit, excellent: RMS residual 0.000 mil, conditioning 361
  Diagnostics: BC sensitivity ratio 0.4044, conditioning 361
```

Two things in that report matter more than the fitted numbers.

The **BC sensitivity ratio** measures whether your observations actually constrain BC. Drop is sensitive to muzzle velocity everywhere, but it only becomes sensitive to BC once the bullet has spent enough time in the air for the drag difference to accumulate—which means low and mid ranges

cannot tell you anything about BC no matter how many of them you shoot. Below a ratio of 0.20 the command refuses to fit BC at all, holds it at your input value, and says so:

```
Note: observations do not constrain BC (BC sensitivity ratio 0.1880 < 0.20
  threshold); BC held at input 0.462. Add a longer-range / transonic observation
  to fit BC.
```

The **conditioning** number reports how well-separated the two parameters are in the fit. High conditioning means the solver could distinguish a velocity error from a BC error; low conditioning means it could not, and the split between them is arbitrary even if the total drop matches.

**Where to Put Your Observations**

The identifiability of a joint fit depends almost entirely on *which* ranges you shoot, not how many. Put at least one observation well out toward the transonic transition—that is where BC and velocity stop being interchangeable. Section 7.4.7 covers the command that designs the experiment for you.

**SAFETY: A Trued Number Is Not a Measured Number**

A fitted muzzle velocity is the velocity that makes the model match your data. It is not what a chronograph would read, and it silently absorbs every error the model does not know about—a mis-ranged target, a scope that does not track, a sight height typed wrong. Always verify computational predictions against real-world chronograph data before relying on them, and re-shoot the card before you trust a fit that moved your velocity more than about 50 fps.

### 7.4.6 Stage Two: DSF Truing Below Mach 1.2

Muzzle velocity and BC are supersonic parameters. Trued against supersonic data they will carry a solution out to the transonic transition and no further, because past Mach 1.2 the discrepancy between model and reality stops being a drag-scaling problem and becomes a shape-of-the-curve problem. No single BC repairs that.

The Applied Ballistics answer is a two-stage workflow, and since v0.28.0 the engine implements the second stage. `dsf` records a *drop-scale factor*: the ratio of your observed drop to the predicted drop, keyed to the Mach number the bullet was travelling at when you measured it.

**Drop-Scale Factor**

A DSF point is a pair $(M, f)$ where $f$ = observed drop/predicted drop at the Mach number $M$ the projectile had at that range. It is applied as a strictly *drop-only* post-processing correction: velocity, energy and time-of-flight outputs are untouched, because a DSF is a curve-fitting patch, not a discovery about the bullet's drag.

Listing 7.14: Recording a transonic DSF point

```
ballistics dsf \
  --saved-profile "r700-308-168smk" \
  --range 1000 --observed-drop 11.4mil
```

Everything but the observation comes from the profile—there are no ballistic overrides on this command—so the correction is recorded against exactly the solution it will be applied to. The observed drop is suffixed with its own unit (`mil`, `moa`, or `in`) with no separator.

Three rules keep the table honest:

- **A supersonic observation is rejected outright.** Above Mach 1.2 the answer belongs to `true-velocity`; the staging gate steers you back there rather than letting a DSF absorb a velocity error.
- **Up to six points per profile**, and a new point within 0.05 Mach of an existing one supersedes it rather than accumulating.
- **The table auto-applies** to `trajectory` and `come-ups` runs against that profile, with a table-format "DSF table active" note so the correction is never invisible. `profile save --clear-dsf` resets it.

**Where the Windows Are**

Since v0.28.1 the truing report prints the range window each stage is valid in, so you are not guessing. The MV-calibration window is 90–100% of the Mach 1.2 crossing distance, and observations outside it are flagged per observation:

```
warning: observation at 900.0 is outside the MV-calibration window (733.8-815.4)
    ; MV fits from this range are weakly identified
...
  MV-calibration window: 733.8-815.4 yd (90-100% of the Mach 1.2 distance)
  for optimal observation ranges run: ballistics plan-truing
```

The `dsf` verb prints its own window—at or beyond 90% of the Mach 0.9 crossing—and when no window exists at all the report says which case you are in: supersonic throughout, or never supersonic. The truing JSON carries `mv_window_start_m` and `mv_window_end_m`, null when absent.

### 7.4.7 Designing the Experiment First

Everything above is about what to do with data you already have. The better question is which data to go and collect—and that is a question you can answer before burning a round. Since v0.27.1, `plan-truing` chooses the set of target ranges that will actually identify both parameters, given the facility you have access to and the reading precision you can achieve:

Listing 7.15: Planning a three-station truing session

```
ballistics plan-truing \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --range-grid 300:1200:100 \
  --observation-count 3 \
  --measurement-resolution 0.1 \
  --zero-distance 200 --sight-height 1.5
```

Listing 7.16: The plan

```
=== TRUING EXPERIMENT PLAN ===
  Recommendation: joint MV + BC
  Measurement assumption: independent 1-sigma reading error = 0.1000 mil
  Selected ranges: 700.0, 800.0, 1200.0 yd
  Constraints: 3 stations, minimum separation 0.0 yd

  BC sensitivity ratio: 0.40832
  Condition number: 153.72
```

```
  Singular values: min 24.51465, max 435.68799; weak-axis fractional 1-sigma: 0.0408

  Range (yd)    Drop (mil)  MV sensitivity  BC sensitivity   Info (nats)
       700.0        5.3195     -122.51399      -32.92223       0.32574
       800.0        6.8681     -157.91580      -47.53764       0.37054
      1200.0       15.4937     -351.09332     -154.49175       2.78660

  Search: exhaustive (120 of 120 candidate sets evaluated)
```

Note what the optimizer chose: two mid-range stations and one far out at 1200. It is not trying to spread stations evenly—it is trying to make the MV and BC sensitivity columns as *different* from each other as possible, because two stations that respond the same way to both parameters cannot separate them no matter how carefully you read them.

Candidates come from `--range-grid START:END:STEP` or an explicit `--candidate-ranges` list, `--observation-cou` fixes the number of stations, and `--minimum-separation` keeps them from bunching. The search is exhaustive where the candidate set allows it and falls back to deterministic greedy exchange where it does not, so the same inputs always give the same plan. Unreachable candidates are reported rather than quietly dropped.

**The Plan Depends on Your Reading Precision**

`--measurement-resolution` is not a formality. The report's own notes say so plainly: at twice the declared sigma the optimizer changes either the station set or the joint-versus-MV-only classification. Declare the precision you can genuinely achieve—and if the facility cannot identify BC at all at that precision, the command says so and returns an honest MV-only recommendation instead of a plan that pretends otherwise.

### 7.4.8 Truing With Known Uncertainties

A point estimate tells you where the fit landed. It does not tell you how wide the landing zone was. If you know the one-sigma error of your drop readings, v0.27.1's uncertainty-aware path turns the fit into a weighted maximum-a-posteriori estimate with intervals attached:

Listing 7.17: Uncertainty-aware joint truing

```
ballistics true-velocity \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --measured-drop 9.016 --range 900 \
  --observed "1000:11.116:0.05" \
  --observed "1200:16.161:0.08" \
  --observation-sigma 0.05 \
  --predict-range 1400 \
```

```
  --zero-distance 200 --sight-height 1.5 \
  --drop-unit mil
```

Listing 7.18: MAP estimate with parameter intervals

```
=== UNCERTAINTY-AWARE MV + BC TRUING ===
  Method: weighted joint MAP + local Gaussian approximation
  Observation errors: absolute known 1-sigma (mil)

  MAP muzzle velocity:    2649.89 fps
  MAP ballistic coefficient:  0.46205
  MV 95% interval: [2588.90, 2710.88] fps (SD 31.12)
  BC 95% interval: [0.43448, 0.48962] (SD 0.01407)
  MV/BC correlation: -0.99438
```

That correlation of $-0.994$ is the most informative number on the page. The fit reproduced all three observations essentially exactly, and still cannot tell you the muzzle velocity to better than $\pm 60$ fps—because within this data a velocity increase and a BC decrease are very nearly the same trajectory. Three stations that agree perfectly have not pinned down two parameters; they have pinned down one combination of them. That is precisely the situation `plan-truing` exists to prevent.

`--observation-sigma` sets a default one-sigma for every reading; an optional third field on `--observed` (`RANGE:DROP:SIGMA`) overrides it per observation. `--mv-prior` and `--bc-prior` take `MEAN:SIGMA` and constrain the fit explicitly, so a prior is something you declare rather than something the optimizer smuggles in. `--predict-range` propagates the parameter covariance forward into a 95% drop band at a range you have not shot.

The report ends with named warnings rather than silence: `low_effective_degrees_of_freedom` when the residual fit quality is weakly validated, `prediction_outside_observed_domain` when a predictive range sits outside the data, and a convergence note when the optimizer's own gradient test stalls. Supplying no uncertainty leaves the legacy point-estimate path and its schema exactly as they were.

## 7.5 BC Confidence Intervals

A single drop measurement at one distance gives you a single BC estimate. But how accurate is that estimate? The `estimate-bc` command provides a verification error, but a more robust approach is to combine BC estimation with Monte Carlo simulation.

### 7.5.1 Procedure

1. Shoot a group at your truing distance (e.g., 600 yards) and record the mean drop and the spread of drops.

2. Run `estimate-bc` to get the point estimate.
3. Run a Monte Carlo simulation with the estimated BC and realistic velocity SD to see if the predicted dispersion matches your observed group.
4. If the predicted dispersion is smaller than observed, your BC uncertainty is larger than assumed; increase `--bc-std` until the Monte Carlo output matches reality.

Listing 7.19: Validating BC estimate with Monte Carlo

```
# Step 1: Estimate BC (assume 0.452 comes back)
# Step 2: Validate with Monte Carlo
ballistics monte-carlo \
  --velocity 2700 --bc 0.452 \
  --mass 168 --diameter 0.308 \
  --num-sims 3000 --velocity-std 12 \
  --bc-std 0.005 --wind-std 2 \
  --wind-speed 5 --wind-direction 90 \
  --angle-std 0.05 --target-distance 600
```

If the Monte Carlo CEP closely matches your observed group size, your BC estimate and uncertainty model are well-calibrated.

### 7.5.2 BC Stability Across Velocity Ranges

A single BC value assumes constant drag behavior. In reality, BC varies with velocity (see Section 4.3). You can assess this by truing at multiple distances:

Listing 7.20: Truing at multiple distances to assess BC stability

```
# True at 400 yards
ballistics true-velocity \
  --measured-drop 2.9 --range 400 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5

# True at 600 yards
ballistics true-velocity \
  --measured-drop 5.8 --range 600 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5

# True at 800 yards
ballistics true-velocity \
  --measured-drop 9.6 --range 800 \
```

```
  --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5
```

If the trued velocity changes significantly across distances, your BC is not constant—the drag model is not a good match for your bullet's drag curve. This is a strong signal that you should switch drag models (usually from G1 to G7 for modern boat-tail bullets) or use velocity-dependent BC segments.

## 7.6 When to Use G1 vs. G7 for Truing

The choice of drag model is especially important for truing because the whole point of truing is to get an accurate prediction across the *entire* flight envelope, not just at the truing distance.

### 7.6.1 The G1 Problem

G1 BCs are velocity-dependent for most modern bullets. A G1 BC trued at 600 yards may give an excellent prediction at 600 but diverge at 1000 because the drag curve shape is wrong in the transonic regime. When you true with G1, you are fitting a scaling factor to a curve that does not match your bullet—it works at the calibration point but extrapolates poorly.

### 7.6.2 The G7 Advantage

G7 BCs tend to be more constant across the velocity range for boat-tail bullets because the G7 standard projectile more closely matches the drag profile. A G7 BC trued at 600 yards will generally extrapolate well to 1000 yards and beyond. The correction factor you apply by truing is small and relatively constant, so the extrapolation is stable.

**Truing Recommendation**

True with G7 whenever possible. If you only have G1 data, true the G1 BC, but be cautious about extrapolating far beyond your truing distance. Consider switching to G7 if your G1 solution diverges at extended range.

### 7.6.3 Generating BC Segments

For the most accurate predictions, you can use velocity-dependent BC segments. The `generate-bc-segments` command creates a set of BC values for different velocity ranges based on the bullet type:

Listing 7.21: Generating velocity-dependent BC segments

```
ballistics generate-bc-segments \
  --bc 0.462 --mass 168 --diameter 0.308 \
```

```
  --model "168gr SMK" --drag-model g1
```

This uses the `BCSegmentEstimator` in `src/bc_estimation.rs` to identify the bullet type (match boat-tail, hunting flat-base, VLD, etc.) and generate appropriate BC degradation factors for each velocity band. The estimator uses sectional density and the ratio of BC to sectional density to classify bullets and estimate how much BC degrades at lower velocities.

The output shows velocity bands and their associated BC values:

Listing 7.22: BC segment output

```
# +=======================================+
# |        BC SEGMENT GENERATION          |
# +=======================================+
# | Base BC:         0.462                |
# | Caliber:         0.308 inches         |
# | Weight:          168.0 grains         |
# | Model:           168gr SMK            |
# | Drag Model:      G1                   |
# +=======================================+
# |         VELOCITY SEGMENTS             |
# +=======================================+
# | 2800-5000 fps  -> BC: 0.4620         |
# | 2400-2800 fps  -> BC: 0.4551         |
# | 2000-2400 fps  -> BC: 0.4458         |
# | 1600-2000 fps  -> BC: 0.4366         |
# | 0-1600 fps     -> BC: 0.4273         |
# +=======================================+
```

These segments can then be used with the trajectory command's `--use-bc-segments` flag for improved accuracy across the full velocity range. The segments are also compatible with the BC5D table system discussed in Chapters 12 and 13 and Chapter D.

## 7.7 A Complete Truing Workflow

Here is the end-to-end workflow for truing a ballistic solution:

1. **Chronograph your load.** Record the average muzzle velocity over at least 10 rounds. Note the SD and ES.
2. **Build an initial dope card.** Use the published BC and your chronographed velocity:

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
```

```
  --auto-zero 100 --max-range 800
```

3. **Shoot at a known distance.** Fire a 5-round group at 500–800 yards. Record the average scope correction (in MIL) needed to center the group.
4. **True the solution.** Use the observed drop to find the effective velocity:

```
ballistics true-velocity \
  --measured-drop 5.8 --range 600 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 100 --sight-height 1.5 \
  --chrono-velocity 2700
```

5. **Apply the correction.** Use the trued velocity (or the velocity adjustment) in your trajectory command:

```
ballistics trajectory --velocity 2665 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 1000
```

6. **Validate at a second distance.** Shoot at a different range (e.g., 800 yards) and verify that the trued solution predicts the correct drop. If it does, your solution is calibrated. If it does not, the drag model may be mismatched—consider switching to G7 or using BC segments.

## 7.8 Exercises

1. **Effect of BC error.** Using the .308 Win reference load (168gr SMK, BC `0.462`, 2700 fps), compute the trajectory at 800 yards. Then compute it again with BC reduced by 5% (`0.439`). How much does the drop change? How many inches of error does a 5% BC error produce at 800 yards?

```
ballistics trajectory --velocity 2700 \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 800

ballistics trajectory --velocity 2700 \
  --bc 0.439 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 800
```

2. **G1 vs. G7 extrapolation.** True a 6.5 Creedmoor (140gr ELD-M, 2710 fps) using both G1 (BC `0.326`) and G7 (BC `0.610`) drag models. Use a measured drop of 4.2 MIL at 600 yards. Then compute the trajectory to 1000 yards with each trued velocity. Which model extrapolates more consistently beyond the truing distance?

3. **Generate and compare BC segments.** Use `generate-bc-segments` for a 168gr SMK (G1) and a 140gr ELD-M (G7). Compare the BC degradation patterns. Which bullet type shows more BC stability across the velocity range?

```
ballistics generate-bc-segments \
  --bc 0.462 --mass 168 --diameter 0.308 \
  --model "168gr SMK" --drag-model g1

ballistics generate-bc-segments \
  --bc 0.610 --mass 140 --diameter 0.264 \
  --model "140gr ELD-M" --drag-model g7
```

## What's Next

With a trued ballistic solution in hand, you will want to save it so you can reuse it without re-entering every parameter each time. In Chapter 8, we explore the **`profile`** command—how to save rifle and load configurations, manage multiple setups, and build an efficient workflow from load development through match day.

# Chapter 8

# Profiles & Workflow

By now you have seen the `trajectory` command grow to a dozen flags or more: velocity, BC, mass, diameter, drag model, zero distance, sight height, atmospheric conditions, wind, and advanced physics toggles. Typing all of that every time you want to check a dope value is tedious and error-prone. Profiles solve this problem. A profile saves your rifle and load configuration so you can recall it with a single flag.

In this chapter we cover the `profile` command—creating, saving, listing, showing, and deleting profiles. We then discuss the JSON storage format, strategies for managing multiple rifles and loads, and a complete workflow from load development through range day.

## 8.1 Why Use Profiles?

Consider the command from Section 23.5 for a PRS competitor's 6.5 Creedmoor:

Listing 8.1: A long command without profiles

```
ballistics trajectory --velocity 2710 \
  --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --auto-zero 100 \
  --max-range 1200 --sight-height 2.0 \
  --wind-speed 8 --wind-direction 45 \
  --temperature 75 --pressure 29.12 \
  --altitude 5280
```

That is 14 flags. With a saved profile, the same computation becomes:

Listing 8.2: The same computation with a profile

```
ballistics trajectory --saved-profile prs-65cm \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 8 --wind-direction 45 \
  --temperature 75 --pressure 29.12 \
  --altitude 5280
```

The profile supplies velocity, BC, mass, diameter, drag model, and sight height. You only need to specify the parameters that change from shot to shot: wind, weather, and range. Even better, profiles can store default atmospheric conditions and wind, reducing the command further.

Beyond convenience, profiles prevent errors. When you type `--bc 0.610` from memory every time, eventually you will mistype it. A saved profile eliminates that risk. And when you true your BC or update your chronograph data, you update the profile *once* and every future command uses the corrected value.

**Profile**

A *profile* is a saved set of ballistic input parameters—velocity, BC, mass, diameter, drag model, and optionally sight height, zero distance, twist rate, atmospheric defaults, and wind—that can be recalled by name with the `--saved-profile` flag on any command that accepts ballistic inputs.

## 8.2 Creating and Saving a Profile

The `profile` save subcommand creates a new profile:

Listing 8.3: Saving a .308 Win profile

```
ballistics profile save "308-match" \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --drag-model g1 --sight-height 1.5 \
  --zero-distance 100 --twist-rate 10 \
  --bullet-name "168gr Sierra MatchKing" \
  --temperature 59 --pressure 29.92
```

The first argument after save is the profile name—a short, descriptive identifier. This is the name you will use with `--saved-profile` on other commands.

### 8.2.1 Required Parameters

Four parameters are required when saving a profile:

- `--velocity` — Muzzle velocity (fps or m/s)
- `--bc` — Ballistic coefficient
- `--mass` — Bullet mass (grains or grams)
- `--diameter` — Bullet diameter (inches or mm)

### 8.2.2 Optional Parameters

The profile can store many additional parameters that serve as defaults when the profile is loaded:

- `--drag-model` — G1 or G7. Default: G1.
- `--sight-height` — Scope height above bore (inches or mm).
- `--zero-distance` — Default zero distance (yards or meters).
- `--auto-zero` — Auto-zero distance for trajectory commands.
- `--twist-rate` — Barrel twist rate (inches per turn).
- `--twist-right` — Twist direction. Default: true (right-hand twist).
- `--bullet-name` — Human-readable bullet description.
- `--bullet-length` — Bullet length (inches or mm), used for BC5D table lookups.
- `--wind-speed` — Default wind speed.
- `--wind-direction` — Default wind direction.
- `--shooting-angle` — Default shooting angle (degrees).
- `--temperature`, `--pressure`, `--humidity`, `--altitude` — Default atmospheric conditions.
- `--use-bc-segments` — Enable velocity-dependent BC segmentation.

Listing 8.4: A complete 6.5 Creedmoor profile

```
ballistics profile save "prs-65cm" \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --drag-model g7 \
  --sight-height 2.0 --zero-distance 100 \
  --auto-zero 100 \
  --twist-rate 8 --twist-right true \
  --bullet-name "140gr Hornady ELD-M" \
  --bullet-length 1.375 \
  --temperature 59 --pressure 29.92 \
```

```
  --humidity 50 --altitude 0
```

**Name Profiles Descriptively**

Use names that identify both the rifle and the load: `"308-match-168smk"`, `"65cm-prs-140eldm"`, `"338lm-berger300"`. When you have multiple profiles, clear names prevent loading the wrong one. Avoid spaces in profile names—use hyphens instead.

## 8.3 The Profile JSON Format

Profiles are stored as JSON files in the `~/.ballistics/profiles/` directory. Each profile is a single file named `<profile-name>.json`. The format is straightforward:

Listing 8.5: Profile JSON structure (example)

```
{
  "name": "308-match",
  "velocity": 2700.0,
  "bc": 0.462,
  "mass": 168.0,
  "diameter": 0.308,
  "drag_model": "G1",
  "twist_rate": 10.0,
  "sight_height": 1.5,
  "zero_distance": 100.0,
  "units": "imperial",
  "temperature": 59.0,
  "pressure": 29.92,
  "humidity": 50.0,
  "altitude": 0.0,
  "bullet_name": "168gr Sierra MatchKing",
  "created": "1709241600",
  "wind_speed": null,
  "wind_direction": null,
  "shooting_angle": null,
  "auto_zero": 100.0,
  "twist_right": true,
  "use_bc_segments": false,
  "bullet_length": null
}
```

### 8.3.1 Field Reference

- **name**: Profile identifier. Must match the filename (without extension).
- **velocity**: Muzzle velocity in the units specified by **units**.
- **bc**: Ballistic coefficient (dimensionless).
- **mass**: Bullet mass in grains (imperial) or grams (metric).
- **diameter**: Bullet diameter in inches (imperial) or mm (metric).
- **drag_model**: `"G1"` or `"G7"`.
- **units**: `"imperial"` or `"metric"`. Determines the unit system for all stored values.
- **created**: Unix timestamp of profile creation.
- **Optional fields**: `twist_rate`, `sight_height`, `zero_distance`, `auto_zero`, `wind_speed`, `wind_direction`, `shooting_angle`, `twist_right`, `use_bc_segments`, `bullet_length`, `bullet_name`. When `null`, the corresponding flag falls back to its default value at runtime.

**Direct Editing**

Because profiles are plain JSON, you can edit them with any text editor. This is useful for fine-tuning values after truing (see Chapter 7). Just make sure the JSON is valid—the engine will fail gracefully with an error message if the file is malformed.

### 8.3.2 Profile Storage Location

The profiles directory is automatically created at:

- Linux/macOS: `~/.ballistics/profiles/`
- Windows: `%USERPROFILE%\.ballistics\profiles\`

The `get_profiles_dir()` function in `src/main.rs` uses the `dirs` crate to find the home directory and creates the profiles subdirectory if it does not exist.

## 8.4 Loading and Using Profiles

Once saved, a profile is loaded with the `--saved-profile` flag on any command that accepts ballistic inputs:

Listing 8.6: Using a saved profile for trajectory computation

```
ballistics trajectory --saved-profile 308-match \
```

```
  --auto-zero 100 --max-range 800 \
  --wind-speed 10 --wind-direction 90
```

The engine loads the profile, applies its stored values, and uses CLI flags as overrides. The override priority is:

1. **CLI flags** take highest priority. If you pass `--velocity` on the command line, it overrides the profile's stored velocity.

2. **Profile values** fill in anything not specified on the command line.

3. **Defaults** apply for parameters that are neither on the command line nor in the profile.

This layered system is powerful. You can save a profile with your baseline conditions and then override just the parameters that change:

Listing 8.7: Overriding profile values with CLI flags

```
# Profile says velocity is 2700 fps, but today it's cold
# and we know the load is 50 fps slower
ballistics trajectory --saved-profile 308-match \
  --velocity-adjustment -50 \
  --auto-zero 100 --max-range 800 \
  --temperature 25 --pressure 29.85
```

**A Profile's Atmosphere Now Loads (0.30.0)**

`profile` save has always stored temperature, pressure, humidity and altitude—and, since v0.29.0, the pressure reference and density altitude too. The `trajectory` path loaded none of them. A profile carried its ballistics and silently dropped its conditions, so a saved mountain load solved at sea level unless you retyped the atmosphere every time.

As of v0.30.0 they load as a set, with per-value precedence: a command-line flag beats a `--location` CSV, which beats the profile, which beats the standard atmosphere. **If you have profiles that stored a non-standard atmosphere, runs against them will change.** That is the fix, but re-baseline your dope before relying on it. Profiles that never set an atmosphere are byte-identical.

Two narrower rules ride along: a stored pressure mode applies only when the profile's own pressure value is the one in use, and a stored density altitude only when the run supplies no pressure or altitude of its own.

**A File Flag Now Requires Its Row Selector (0.30.1)**

Three silences used to be possible here. `--location` without `--site` was ignored entirely—no message at all, the file was simply never read. An unreadable path warned and continued. A missing site row warned and continued. All three produced a clean-looking trajectory computed against air, or a gun, that you never chose.

The file flags now require their row selectors at the parser level (`--location` needs `--site`, `--profile` needs `--profile-row`), and both load failures are hard errors naming the path or row. The asymmetry is deliberate: a *standalone* `--site` or `--profile-row` keeps its pre-existing job of labelling a PDF dope card.

### 8.4.1 Profile Compatibility with Other Commands

Profiles work with most commands:

- **`trajectory`** — Full trajectory computation
- `range-table` — Multi-column range card
- `wind-card` — Wind deflection tables
- `mpbr` — Maximum point-blank range
- `come-ups` — Scope adjustment table
- `stability` — Gyroscopic stability analysis

Listing 8.8: Profile with range-table command

```
ballistics range-table --profile prs-65cm \
  --zero-distance 100 --start 100 --end 1200 \
  --step 50 --wind-speed 10 --wind-direction 90 \
  --adjustment-unit mil
```

## 8.5 Managing Multiple Rifles and Loads

Most serious shooters have more than one rifle, and many rifles shoot more than one load. Profiles make managing multiple configurations simple.

### 8.5.1 Listing Profiles

The **`profile`** `list` subcommand shows all saved profiles:

Listing 8.9: Listing all saved profiles

```
ballistics profile list
```

The output is a formatted table:

Listing 8.10: Profile list output

```
# Saved Profiles:
# +--------------------+--------+-------+--------+----------+----------+
# | Name               | Vel    | BC    | Mass   | Diameter | Drag     |
# +--------------------+--------+-------+--------+----------+----------+
# | 308-match          |   2700 | 0.462 |  168.0 |    0.308 | G1       |
# | 65cm-prs           |   2710 | 0.610 |  140.0 |    0.264 | G7       |
# | 338lm-elr          |   2725 | 0.818 |  300.0 |    0.338 | G7       |
# | 223-varmint        |   3100 | 0.372 |   77.0 |    0.224 | G1       |
# +--------------------+--------+-------+--------+----------+----------+
```

### 8.5.2 Showing Profile Details

To see the full details of a profile:

Listing 8.11: Showing a profile's details

```
ballistics profile show 308-match
```

This displays all stored parameters, including optional fields like sight height, zero distance, twist rate, and default atmospheric conditions.

### 8.5.3 Deleting Profiles

To remove a profile that is no longer needed:

Listing 8.12: Deleting a profile

```
ballistics profile delete old-load
```

The profile file is permanently removed from disk.

### 8.5.4 More Than One Zero

A profile has one master zero, and a rifle often has more than one. A suppressed load and an unsuppressed load do not share a point of impact. A 100-yard practice zero and a 300-yard match zero are different bore angles. A particular lot of ammunition may print consistently half a mil low.

Since v0.31.0 a profile can carry a list of named *zero sets*, each with an optional zero distance and optional per-load point-of-impact corrections:

Listing 8.13: Storing a suppressed zero that also needs a dial correction

```
# 200 yd suppressed zero, needing 0.3 mil LESS elevation and
# 0.1 mil MORE right windage than the master zero.
ballistics profile zero-set add "308-match" \
  --name suppressed --zero-distance 200 \
  --poi-up -0.3 --poi-right 0.1

ballistics profile zero-set list "308-match"
```

Select one per run with `--zero-set`, which is accepted by **`trajectory`**, `come-ups`, `wind-card`, `range-table`, `dsf` and `plan-truing`:

Listing 8.14: Using a named zero set

```
ballistics come-ups --profile "308-match" \
  --zero-set suppressed --zero-distance 200 \
  --start 300 --end 800 --step 100
```

**POI Corrections Are Dial Corrections**

`poi_up_mil` and `poi_right_mil` are constant angular *dial* corrections, not observed impacts. Positive means dial up or right *more*—so a load that impacts high stores a *negative* value. They are added to total-correction dial outputs (come-ups, wind-card drift, range-table Drop and Wind, the PDF card's Drop and Wind) at the shared conversion boundary, *before* the tracking-factor division of Section 24.6: dial $=$ (true $+$ bias)/CF. Component holds—the mover Ring and lead—deliberately stay bias-free.

A selected set's zero distance feeds the auto-zero exactly as the profile's own zero would, and an explicit CLI zero flag still wins. An unknown set name is a hard error that lists the available sets rather than silently falling back to the master zero. Defaults—no `--zero-set`—are byte-identical everywhere.

Two existing data sources were wired into this. A `--profile` CSV row's `V_OFFSET_MIL` and `H_OFFSET_MIL` columns had been allowlisted but unused; they now form a selectable ephemeral set named after the row. And an `.a7p` import with `--zero-click` additionally records the file's zero click state as a set named `a7p-zero`.

**Forward Compatibility**

Zero sets follow the documented `bc_segments` pattern: an older reader loads the profile and solves the master zero identically, and rejects `--zero-set` loudly as an unknown flag. Only an old reader's *re-save* drops the stored sets.

### 8.5.5 Importing an ArcherBC2 Profile

If your load already lives in another app, you may not have to retype it. Since v0.25.0 the engine can read the `.a7p` files written by ArcherBC2:

Listing 8.15: Previewing an .a7p import

```
# Preview what would be imported -- nothing is written
ballistics profile import my-load.a7p --dry-run

# Import under the file's own profile name, sanitized
ballistics profile import my-load.a7p

# Import under a chosen name, failing hard on a bad checksum
ballistics profile import my-load.a7p \
  --name "r700-308-168smk" --strict
```

The reader is zero-dependency—an MD5 check plus a proto3 wire decoder—and it prints a full mapping report rather than a success message. Every field is listed as mapped, converted, or *unmapped*, so you can see exactly what the engine took and what it had to leave behind. Multi-BC velocity bands become profile BC segments (Section 12.3) and a `CUSTOM` Mach:$C_D$ curve is carried end to end.

**Always Dry-Run First**

`--dry-run` parses the file and prints the same report without writing anything. Read the unmapped list before you commit the import—a field the engine could not map is a field you will have to supply yourself.

**Profile Schema v2**

Carrying custom drag curves required a profile schema bump to v2. It is backward compatible: an older profile loads unchanged, and a v2 profile that uses none of the new fields reads the same as a v1 one.

### 8.5.6 Naming Conventions

A good naming convention makes profiles easier to manage. We recommend the format `<rifle>-<load>` or `<cartridge>-<purpose>-<bullet>`:

Listing 8.16: Creating profiles with systematic names

```
# Different loads for the same rifle
ballistics profile save "r700-308-168smk" \
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --bullet-name "168gr SMK" --drag-model g1

ballistics profile save "r700-308-175smk" \
  --velocity 2650 --bc 0.505 \
  --mass 175 --diameter 0.308 \
  --bullet-name "175gr SMK" --drag-model g1

# Different rifles in the same cartridge
ballistics profile save "tikka-65cm-140eldm" \
  --velocity 2710 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --bullet-name "140gr ELD-M" --drag-model g7 \
  --sight-height 2.0 --twist-rate 8

ballistics profile save "bergara-65cm-140eldm" \
  --velocity 2740 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --bullet-name "140gr ELD-M" --drag-model g7 \
  --sight-height 1.8 --twist-rate 8
```

Notice that the two 6.5 Creedmoor profiles share the same bullet and BC but have different velocities (due to different barrel lengths) and sight heights (due to different scope mount configurations). The profile system captures these per-rifle differences.

## 8.6 Workflow: From Load Development to Range Day

Let's walk through the complete workflow of developing a load, building a profile, truing the solution, and generating a dope card for a match.

### 8.6.1 Step 1: Load Development

You have selected a 6.5 Creedmoor with 140gr ELD-M bullets and 41.5 gr of H4350 powder. After working up the load on the bench, you chronograph a 10-round string:

- Mean velocity: 2718 fps
- SD: 9 fps
- ES: 28 fps

Create an initial profile with the chronograph data and the published G7 BC:

Listing 8.17: Step 1: Create initial profile from chronograph data

```
ballistics profile save "tikka-65cm-h4350" \
  --velocity 2718 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --drag-model g7 \
  --sight-height 2.0 --twist-rate 8 \
  --zero-distance 100 --auto-zero 100 \
  --bullet-name "140gr ELD-M / 41.5gr H4350"
```

### 8.6.2 Step 2: Generate an Initial Dope Card

Build a preliminary dope card with standard atmospheric conditions:

Listing 8.18: Step 2: Generate preliminary dope card

```
ballistics trajectory --saved-profile tikka-65cm-h4350 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90
```

Print or screenshot this dope card—you will take it to the range for validation.

### 8.6.3 Step 3: Validate at Range

At the range, confirm your 100-yard zero, then shoot at 500 and 700 yards. Record the actual scope corrections needed:

- 500 yards: Dialed 3.5 MIL up (expected: 3.4 MIL)
- 700 yards: Dialed 5.9 MIL up (expected: 5.7 MIL)

The solution is shooting 0.1–0.2 MIL low at medium-long range. Time to true.

### 8.6.4 Step 4: True the Solution

Use the 700-yard data point (longer range gives more sensitivity for truing):

Listing 8.19: Step 4: True velocity against observed drop

```
ballistics true-velocity \
  --measured-drop 5.9 --range 700 \
  --bc 0.610 --drag-model g7 \
  --mass 140 --diameter 0.264 \
  --zero-distance 100 --sight-height 2.0 \
  --chrono-velocity 2718
```

Suppose the trued velocity comes back as 2695 fps—23 fps lower than the chronograph. The velocity adjustment is $2695 - 2718 = -23$ fps.

**SAFETY: Trued Velocity Is Not Physical Velocity**

The trued velocity (2695 fps) is an *effective* value that accounts for all system errors—chronograph bias, BC uncertainty, atmospheric model error. Do not use it to infer chamber pressure or to modify your powder charge. Your actual muzzle velocity is still what the chronograph measured. The trued value is a computational calibration, not a physical measurement.

### 8.6.5 Step 5: Update the Profile

Delete the old profile and save a new one with the trued velocity:

Listing 8.20: Step 5: Update profile with trued velocity

```
ballistics profile delete tikka-65cm-h4350

ballistics profile save "tikka-65cm-h4350" \
  --velocity 2695 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --drag-model g7 \
  --sight-height 2.0 --twist-rate 8 \
  --zero-distance 100 --auto-zero 100 \
  --bullet-name "140gr ELD-M / 41.5gr H4350 (trued)"
```

Alternatively, you can edit the JSON file directly in `~/.ballistics/profiles/tikka-65cm-h4350.json` and change the `velocity` field from 2718 to 2695.

**Keep Profiles Current**

A profile is only as good as the data it contains. If you change components—a new scope with a different sight height, a different lot of bullets with a different BC, or a barrel replacement that changes velocity—update the profile immediately. Stale profiles produce stale predictions, and stale predictions produce misses.

### 8.6.6 Step 6: Generate the Final Dope Card

Now generate the match-day dope card with the conditions forecast for your event:

Listing 8.21: Step 6: Match-day dope card

```
ballistics trajectory --saved-profile tikka-65cm-h4350 \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 8 --wind-direction 45 \
  --temperature 82 --pressure 25.50 \
  --altitude 4800 \
  --sample-trajectory \
  --output pdf --output-file match_dope.pdf \
  --location-name "Raton, NM"
```

### 8.6.7 Step 7: Verify with Monte Carlo

Before the match, run a Monte Carlo simulation to understand your expected dispersion at the hardest target distance:

Listing 8.22: Step 7: Pre-match dispersion analysis

```
ballistics monte-carlo \
  --velocity 2695 --bc 0.610 \
  --mass 140 --diameter 0.264 \
  --num-sims 5000 --velocity-std 9 \
  --bc-std 0.004 --wind-std 2 \
  --wind-speed 8 --wind-direction 45 \
  --angle-std 0.03 --target-distance 1100
```

This tells you the CEP at 1100 yards under your expected conditions, so you can make go/no-go decisions on difficult stages. If the CEP is 8 inches and the target is a 12-inch plate, you have a solid first-round hit probability.

### 8.6.8 Step 8: Post-Match Review

After the match, review your actual corrections against the dope card. If you consistently dialed 0.1 MIL more than predicted at the longer stages, that data is gold—true again with the new data and update the profile. Over time, your ballistic solution converges to near-perfect predictions.

## 8.7 CSV-Based Profile Loading

In addition to JSON profiles, BALLISTICS-ENGINE supports loading ballistic data from CSV files. This is useful for managing multiple rifles and loads in a spreadsheet format or for teams that share a common data file:

Listing 8.23: Loading from a CSV profile

```
ballistics trajectory \
  --profile guns.csv --profile-row "308-match" \
  --auto-zero 100 --max-range 800
```

The CSV file should have a header row with recognized column names. The engine uses fuzzy matching on column headers, so `MUZZLE_VELOCITY`, `MV`, and `VELOCITY` all resolve correctly. If a column name is close to a recognized name (e.g., `VELOCTY` instead of `VELOCITY`), the engine will suggest the correction with a warning.

You can also load atmospheric conditions from a separate location CSV:

Listing 8.24: Loading location data from CSV

```
ballistics trajectory \
  --profile guns.csv --profile-row "308-match" \
  --location sites.csv --site "Raton-NM" \
  --auto-zero 100 --max-range 1000
```

This separates the *rifle/load* data from the *environmental* data, which makes sense operationally: you have one stable set of rifle parameters and multiple locations you might shoot at.

## 8.8 Exercises

1. **Create a complete profile.** Save a profile for a rifle and load of your choice. Include all optional parameters (sight height, twist rate, zero distance, bullet name). Then generate a trajectory using only `--saved-profile` and weather flags:

   ```
   ballistics profile save "my-rifle" \
   ```

```
  --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --sight-height 1.5 --twist-rate 10 \
  --auto-zero 100 \
  --bullet-name "168gr SMK"

ballistics trajectory --saved-profile my-rifle \
  --auto-zero 100 --max-range 800 \
  --wind-speed 10 --wind-direction 90
```

2. **Override drill.** Using the profile from exercise 1, run the trajectory with the profile's default conditions, then override the velocity with `--velocity 2650`. Verify that the lower velocity produces more drop at 600 yards:

```
ballistics trajectory --saved-profile my-rifle \
  --auto-zero 100 --max-range 600

ballistics trajectory --saved-profile my-rifle \
  --velocity 2650 \
  --auto-zero 100 --max-range 600
```

3. **Multi-rifle comparison.** Create profiles for a .308 Win (168gr SMK, `0.462`, 2700 fps) and a 6.5 Creedmoor (140gr ELD-M, `0.610` G7, 2710 fps). Generate trajectories for both at 1000 yards and compare the drop, wind drift, and remaining energy. Which cartridge offers a flatter trajectory? Which retains more energy?

4. **Profile management.** Create three profiles, list them, show the details of one, and delete one. Verify that the deleted profile is gone from the list:

```
ballistics profile list
ballistics profile show my-rifle
ballistics profile delete my-rifle
ballistics profile list
```

## What's Next

With Part II complete, you can now compute trajectories, zero your rifle, quantify uncertainty through Monte Carlo simulation, true your ballistic solution, and manage profiles for multiple rifles and loads. These are the operational tools of computational ballistics.

In Part III, we turn our attention to the *environment*—the atmosphere and the wind. Chapter 9 examines how BALLISTICS-ENGINE models air density, the ICAO Standard Atmosphere, density altitude, and the physics that connect temperature, pressure, and humidity to drag. Chapter 10 dives into wind modeling in detail, including wind shear, altitude-dependent wind profiles, and the practical art of reading wind for long-range shooting.

# Part III

# Atmosphere & Environment

# Chapter 9

# The Atmosphere

A bullet does not fly through a vacuum. From the instant it leaves the muzzle, every aspect of its flight—velocity, drop, drift, even the speed of sound it races against—is governed by the thin envelope of gas we call the atmosphere. A 168 –grain Sierra MatchKing fired at 2700 fps (823 m/s) at sea level on a standard day will impact roughly 12 inches lower at 1000 yards than the same bullet fired at 5000 ft elevation in Raton, New Mexico on a hot July afternoon. That difference is entirely atmospheric.

Understanding what the atmosphere does to your bullet—and how BALLISTICS-ENGINE models it—is the single most important step toward building accurate trajectory predictions beyond 300 yards. In this chapter, we will walk through the physics of the standard atmosphere, explore how temperature, pressure, and humidity conspire to change air density, and show you how to tell BALLISTICS-ENGINE exactly what conditions your bullet will face.

## 9.1 The ICAO Standard Atmosphere

Before we can talk about non-standard conditions—the hot days, cold mornings, and high-altitude ranges where real shooting happens—we need a baseline. That baseline is the *ICAO Standard Atmosphere*, formally specified in ISO 2533 and maintained by the International Civil Aviation Organization.

**ICAO Standard Atmosphere**

The ICAO Standard Atmosphere defines a set of reference conditions for the Earth's atmosphere from sea level to 84 km altitude. At sea level, the standard conditions are:

- Temperature: 15 °C (59 °F)
- Pressure: 101 325 Pa (1013.25 hPa, 29.92 inHg)
- Air density: 1.225 $\mathrm{kg/m^3}$
- Temperature lapse rate: −6.5 °C/km (troposphere)

The standard atmosphere divides the atmosphere into distinct *layers*, each characterized by a temperature lapse rate—the rate at which temperature changes with altitude. In the troposphere, where all terrestrial shooting occurs, temperature decreases at 6.5 °C per kilometer of altitude gain (approximately 3.6 °F per 1000 ft).

Let's see what BALLISTICS-ENGINE computes under standard conditions:

Listing 9.1: Trajectory at standard sea-level conditions

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 59 --pressure 29.92 \
  --humidity 0 --altitude 0
```

These are the default atmospheric values in BALLISTICS-ENGINE, so omitting `--temperature`, `--pressure`, `--humidity`, and `--altitude` produces the same result. The standard atmosphere provides a universal reference point: when manufacturers publish ballistic data, they almost always assume ICAO standard conditions. In metric the same defaults are expressed as 15 °C and 1013.25 hPa.

**Metric Defaults Were a Near-Vacuum Before 0.21.4**

The atmosphere defaults are unit-dependent, and until v0.21.4 they were not resolved that way: the imperial literals `29.92` and `59.0` were applied *before* `--units` was parsed, so a metric run with no atmosphere flags was solved at 29.92 hPa—roughly 24 km up—and 59 °C. Drag came out about 30 times too low and impact velocity about 42% high. Pressure and temperature are now resolved and range-checked after `--units`, each with its own per-unit default and bounds. **Metric trajectories run without explicit atmosphere flags produce different—correct—output from v0.21.4 onward.** The same fix repaired the `--pressure` validator, which had been applying an hPa range to imperial inHg: it rejected a perfectly good `14` and accepted an hPa typo like `1013`.

**One Resolved Atmosphere for Density and for Mach (0.22.0)**

Drag depends on air density and on Mach number, and those two used to be evaluated against different atmospheres. Density used the ICAO-lapse-resolved station temperature at `--altitude`, while the speed of sound was computed from the raw `--temperature` input—so an altitude shot on the default atmosphere was looking up its drag coefficient at the wrong Mach, a few percent off near transonic. Both now read one resolved atmosphere, computed once per solve and threaded through drag and Magnus rather than re-derived (vapor-pressure polynomial and all) at every integration stage. The Miller stability correction and the fast/Monte-Carlo Mach lookup were fixed the same way, which moves stability classification, spin drift, and Magnus at altitude.

### 9.1.1 Atmospheric Layers

The ICAO model defines seven layers up to 84 km. For ballistics, only the troposphere matters—but the full model is implemented in BALLISTICS-ENGINE for completeness. Table B.5 summarizes the layer structure.

Table 9.1: ICAO Standard Atmosphere layers as implemented in `src/atmosphere.rs`.

| Layer | Name | Base Alt. (km) | Base Temp. (K) | Lapse Rate (K/km) |
|---|---|---|---|---|
| 0 | Troposphere | 0 | 288.15 | −6.5 |
| 1 | Tropopause | 11 | 216.65 | 0.0 |
| 2 | Stratosphere 1 | 20 | 216.65 | +1.0 |
| 3 | Stratosphere 2 | 32 | 228.65 | +2.8 |
| 4 | Stratopause | 47 | 270.65 | 0.0 |
| 5 | Mesosphere 1 | 51 | 270.65 | −2.8 |
| 6 | Mesosphere 2 | 71 | 214.65 | −2.0 |

**Why Model Above the Troposphere?**

No bullet reaches the stratosphere—even a .50 BMG at extreme elevation angles barely exceeds 3 km of altitude. However, the full ICAO model ensures that BALLISTICS-ENGINE handles any altitude input gracefully, including edge cases in API and testing scenarios. The implementation clamps altitudes to the 0–84 000 m range.

### 9.1.2 The Barometric Formula

The relationship between altitude and atmospheric pressure follows the *barometric formula*. Within any layer, the pressure depends on whether the temperature is changing (non-isothermal) or constant (isothermal).

For a non-isothermal layer with lapse rate $L \neq 0$:

$$P = P_b \left( \frac{T_b + L \cdot (h - h_b)}{T_b} \right)^{-g_0/(L \cdot R)} \tag{9.1}$$

For an isothermal layer ($L = 0$):

$$P = P_b \exp\left( \frac{-g_0 \cdot (h - h_b)}{R \cdot T_b} \right) \tag{9.2}$$

where:

- $P_b$ is the base pressure of the layer (Pa)
- $T_b$ is the base temperature of the layer (K)
- $L$ is the temperature lapse rate (K/m)
- $h - h_b$ is the height above the layer base (m)
- $g_0 = 9.806\,65\ \mathrm{m/s^2}$ is standard gravitational acceleration
- $R = 287.0531\ \mathrm{J/(kg \cdot K)}$ is the specific gas constant for dry air

These are the exact equations implemented in `calculate_icao_standard_atmosphere()` in `src/atmosphere.rs`. The function takes an altitude in meters and returns a tuple of temperature (K) and pressure (Pa):

Listing 9.2: ICAO standard atmosphere computation (from `src/atmosphere.rs`)

```
fn calculate_icao_standard_atmosphere(altitude_m: f64) -> (f64, f64) {
    let altitude = altitude_m.clamp(0.0, 84000.0);

    // Find the appropriate atmospheric layer
    let layer = ICAO_LAYERS
        .iter()
```

```
        .rev()
        .find(|layer| altitude >= layer.base_altitude)
        .unwrap_or(&ICAO_LAYERS[0]);

    let height_diff = altitude - layer.base_altitude;
    let temperature = layer.base_temperature
                    + layer.lapse_rate * height_diff;

    let pressure = if layer.lapse_rate.abs() < 1e-10 {
        // Isothermal layer
        layer.base_pressure
            * (-G_ACCEL_MPS2 * height_diff
               / (R_AIR * layer.base_temperature)).exp()
    } else {
        // Non-isothermal layer
        let temp_ratio = temperature / layer.base_temperature;
        layer.base_pressure
            * temp_ratio.powf(
                -G_ACCEL_MPS2 / (layer.lapse_rate * R_AIR))
    };

    (temperature, pressure)
}
```

The implementation searches the `ICAO_LAYERS` array in reverse order to find the correct layer for a given altitude, then applies the appropriate form of the barometric equation.

## 9.2 Temperature, Pressure, and Humidity

The three atmospheric variables you can measure at the shooting position are temperature, barometric pressure, and relative humidity. Each affects your bullet's flight, but not equally—and not always in the direction you might expect.

### 9.2.1 Temperature

Temperature affects the trajectory through two mechanisms:

1. **Air density.** Warmer air is less dense. The ideal gas law tells us that density is inversely proportional to temperature at constant pressure. A bullet flying through warmer (less dense) air experiences less drag and retains velocity longer.
2. **Speed of sound.** The local speed of sound increases with temperature:

$$a = \sqrt{\gamma \cdot R \cdot T} \tag{9.3}$$

where $\gamma = 1.4$ is the heat capacity ratio for air and $T$ is absolute temperature in Kelvin. At 15 °C, $a \approx 340.3\,\text{m/s}$ (1116.4 fps). At 38 °C (100 °F), it rises to about $349.1\,\text{m/s}$ (1145.3 fps). Since drag coefficients are functions of Mach number ($M = v/a$), a higher speed of sound means the bullet is at a *lower* Mach number for the same velocity, which changes the drag coefficient.

Let's see the effect of temperature on our .308 Win reference load:

Listing 9.3: Comparing cold and hot day trajectories

```
# Standard day: 59 degF
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 59 --pressure 29.92 --humidity 50

# Cold morning: 20 degF
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 20 --pressure 29.92 --humidity 50

# Hot afternoon: 100 degF
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 100 --pressure 29.92 --humidity 50
```

At 1000 yards, the difference between a 20 °F morning and a 100 °F afternoon amounts to several inches of drop—enough to miss a steel plate at an NRL Hunter match.

**SAFETY: Temperature Also Affects Muzzle Velocity**

Temperature does not only change the atmosphere—it changes your ammunition. Powder burn rates are temperature-sensitive. A round stored at 20 °F (−7 °C) may produce 50–100 fps less muzzle velocity than the same round at 100 °F (38 °C). Use the `--use-powder-sensitivity` and `--powder-temp-sensitivity` flags (see Chapter 13) to account for this effect, or chronograph your ammunition at the actual shooting temperature. Always verify computational predictions against real-world chronograph data before relying on them.

### 9.2.2 Barometric Pressure

Barometric pressure directly affects air density: higher pressure means more molecules per unit volume, which means more drag. The standard sea-level pressure is 29.92 inHg (1013.25 hPa).

A common source of error is confusing *station pressure* (the actual pressure at your location) with *altimeter setting* (pressure corrected to sea level, as reported by weather stations and airports). If you are shooting at 5000 ft elevation, your station pressure might be around 24.9 inHg, but the local METAR report will show the altimeter setting near 29.92 inHg.

**Station Pressure vs. Altimeter Setting**

When entering pressure into BALLISTICS-ENGINE with the `--pressure` flag, provide the *station pressure*—the actual barometric pressure at your location—not the sea-level-corrected altimeter setting from your weather app. If you provide the altitude with `--altitude`, BALLISTICS-ENGINE can compute the expected standard pressure at that altitude for you. Providing both `--altitude` and the correct station pressure gives the most accurate result.

#### Or Just Tell the Engine Which One You Have

Typing a sea-level barometer reading into a field that means station pressure is a classic source of vertical error, and since v0.29.0 you no longer have to convert it yourself. `--pressure-type` declares which kind of number you typed:

Listing 9.4: A weather-report barometer value at 5000 ft

```
# 30.31 inHg QNH, read at a 5,000 ft firing position.
# Entered as absolute this over-states density; entered as qnh
# it is reduced to ~25.22 inHg station pressure before the solve.
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 --max-range 1000 \
  --pressure 30.31 --pressure-type qnh --altitude 5000
```

`absolute` is the default and is exactly the historical behavior, so nothing changes for a caller who never sets the flag. `qnh` reduces the value to station pressure at `--altitude` through the ICAO inverse-barometric formula. `--zero-pressure-type` does the same job for the zero day, defaulting to whatever the shot day used.

**Where `--pressure-type` Reaches (0.30.0)**

v0.29.0 wired the flag into **`trajectory`** and **`zero`** only. Ten other commands accepted a `--pressure` and silently treated it as absolute—so a weather-report value entered at elevation was wrong by the ISA reduction on all of them. v0.30.0 extended it to `estimate-bc`, `true-velocity`, `plan-truing`, `mpbr`, `come-ups`, `lead`, `wind-card`, `stability`, `range-table` and `compare`, each reducing against its own `--altitude`. A mode typed on the command line also applies to a pressure that came from a `--location` CSV: a typed mode has exactly one pressure in the run to describe.

Listing 9.5: Effect of pressure variation

```
# High-pressure system: 30.42 inHg
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 59 --pressure 30.42 --humidity 50

# Low-pressure system: 29.42 inHg
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 59 --pressure 29.42 --humidity 50
```

A 1 inHg swing in barometric pressure—the difference between a strong high-pressure system and an approaching storm—changes air density by roughly 3.4%. At 1000 yards, this translates to a small but measurable change in drop.

### 9.2.3 Humidity

Here is a fact that surprises many shooters: *humid air is less dense than dry air at the same temperature and pressure.*

This seems counterintuitive—water is heavy, so shouldn't adding water vapor make air heavier? The key insight is that water vapor *displaces* other gas molecules. A water molecule ($H_2O$, molecular

mass 18.015 g/mol) is lighter than either nitrogen ($N_2$, 28.014 g/mol) or oxygen ($O_2$, 31.998 g/mol). When water vapor enters the air, it replaces heavier molecules with lighter ones, reducing the overall density.

The density of moist air is computed from Dalton's law of partial pressures:

$$\rho = \frac{P_d}{R_d \cdot T} + \frac{P_v}{R_v \cdot T} \tag{9.4}$$

where $P_d$ is the partial pressure of dry air, $P_v$ is the vapor pressure, $R_d = 287.05\,\text{J}/(\text{kg} \cdot \text{K})$ is the gas constant for dry air, and $R_v = 461.495\,\text{J}/(\text{kg} \cdot \text{K})$ is the gas constant for water vapor. This is the equation implemented in `calculate_atmosphere()` in `src/atmosphere.rs`.

Listing 9.6: Comparing dry and humid conditions

```
# Dry air: 0% humidity
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 85 --pressure 29.92 --humidity 0

# Saturated air: 100% humidity
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 85 --pressure 29.92 --humidity 100
```

The effect of humidity is real but modest compared to temperature and pressure. At 85 °F (29 °C) and 100% humidity, the density reduction is about 1% compared to dry air. At cooler temperatures, the effect shrinks further because cold air holds far less moisture.

**When Humidity Matters**

Humidity has its greatest effect in hot, humid conditions—think summertime in the southeastern United States or tropical climates. At 95 °F and 90% humidity, the density difference from dry air is roughly 1.5%. In cold weather (30 °F or below), humidity has negligible impact because the saturation vapor pressure is vanishingly small. For precision rifle competition, always input the actual humidity; for hunting in cold weather, it is acceptable to use a rough estimate.

### 9.2.4 Saturation Vapor Pressure

To compute the vapor pressure from relative humidity, we first need the *saturation vapor pressure*—the maximum water vapor pressure at a given temperature. The engine uses the Arden Buck equation, which is more accurate than the simpler Magnus formula across a wide temperature range:

$$e_s = 6.1121 \exp\left[\left(18.678 - \frac{T_c}{234.5}\right) \cdot \frac{T_c}{257.14 + T_c}\right] \quad \text{(hPa, over water)} \tag{9.5}$$

For temperatures below 0 °C, a separate formulation over ice is used:

$$e_s = 6.1115 \exp\left[\left(23.036 - \frac{T_c}{333.7}\right) \cdot \frac{T_c}{279.82 + T_c}\right] \quad \text{(hPa, over ice)} \tag{9.6}$$

The actual vapor pressure is then:

$$e = \frac{RH}{100} \cdot e_s \tag{9.7}$$

where $RH$ is the relative humidity as a percentage.

## 9.3 Air Density: The Variable That Matters Most

Temperature, pressure, and humidity are the inputs. Air density is the output—and it is the single variable that most directly governs how much drag the bullet experiences.

**Air Density**

*Air density* ($\rho$) is the mass of air per unit volume, measured in $\mathrm{kg/m^3}$ (or equivalently slugs/$\mathrm{ft^3}$ in imperial units). At ICAO standard sea-level conditions, $\rho_0 = 1.225\,\mathrm{kg/m^3}$.

The drag force on a projectile is proportional to air density:

$$F_D = \frac{1}{2}\,\rho\,v^2\,C_D\,A \tag{9.8}$$

Double the air density, double the drag force. Halve the air density, halve the drag. Everything else in the trajectory—drop, time of flight, remaining velocity, wind deflection—follows from this relationship.

In the trajectory solver (`src/derivatives.rs`), the actual drag computation scales the base retardation by a *density ratio*:

Listing 9.7: Density scaling in drag computation (`src/derivatives.rs`)

```
// Calculate density scaling
let density_scale = air_density / STANDARD_AIR_DENSITY;

// ...

// Calculate drag acceleration
let standard_factor = drag_factor * CD_TO_RETARD;
let a_drag_ft_s2 =
    (v_rel_fps.powi(2) * standard_factor
     * yaw_multiplier * density_scale) / bc_val;
```

The `density_scale` factor normalizes local air density against the standard value of $1.225\,\mathrm{kg/m^3}$ (defined as `STANDARD_AIR_DENSITY` in `src/constants.rs`). When you shoot at altitude, on a hot day, or in humid conditions, this ratio drops below 1.0, reducing drag proportionally.

### 9.3.1 The Density Ratio in Practice

Table 9.2 shows representative density ratios under various conditions, all relative to standard sea-level air.

Table 9.2: Approximate air density ratios under various conditions.

| **Condition** | **Density (kg/m$^3$)** | **Ratio ($\rho/\rho_0$)** |
|---|---|---|
| ICAO Standard (sea level, 59 °F) | 1.225 | 1.000 |
| Hot sea level (100 °F, dry) | 1.127 | 0.920 |
| Cold sea level (0 °F, dry) | 1.348 | 1.100 |
| Raton, NM (6200 ft, 70 °F) | 1.019 | 0.832 |
| Denver, CO (5280 ft, 59 °F) | 1.047 | 0.855 |
| Whittington Center (6700 ft, 90 °F) | 0.970 | 0.792 |
| Leadville, CO (10 150 ft, 59 °F) | 0.899 | 0.734 |

A shooter zeroed at sea level who travels to a match at 6200 ft in Raton will find that their bullet impacts *high* because the thinner air produces less drag. The bullet retains more velocity, drops less, and arrives at the target sooner than predicted by a sea-level zero.

## 9.4 Density Altitude vs. True Altitude

Pilots use a concept called *density altitude* that is equally valuable to precision shooters. Density altitude is the altitude in the standard atmosphere that has the same air density as your current conditions.

**Density Altitude**

*Density altitude* is the pressure altitude corrected for non-standard temperature. It represents the altitude in the ICAO Standard Atmosphere at which the air density equals the actual local density. A hot day at sea level can have a density altitude of 2000 ft or more; a cold day at 5000 ft might have a density altitude of only 3000 ft.

Why does this matter? Because density altitude collapses temperature, pressure, and humidity into a single number that tells you exactly how the atmosphere will affect your bullet. Two shooting locations with the same density altitude will produce nearly identical trajectories (assuming the same load), regardless of their true altitudes.

The density altitude can be estimated from:

$$h_{\text{density}} \approx h_{\text{station}} + 118.8 \cdot (T_{\text{actual}} - T_{\text{std}}) \tag{9.9}$$

where $h_{\text{station}}$ is the station pressure altitude in feet, $T_{\text{actual}}$ is the actual temperature in °F, and $T_{\text{std}}$ is the standard temperature at that altitude. This is an approximation; the exact computation requires the full barometric formula.

**A Quick Density Altitude Rule of Thumb**

For every 10 °F above standard temperature, add roughly 600 ft to your altitude. For every 10 °F below standard, subtract 600 ft. At 5000 ft elevation, the standard temperature is about 41 °F (5 °C). If it is actually 81 °F (27 °C), your density altitude is approximately $5000 + 4 \times 600 = 7400$ ft.

### 9.4.1 Entering Conditions As a Density Altitude

A Kestrel reads density altitude directly, and so do Shooter, Nosler, AB Analytics, Ballistic AE and TRASOL. Since v0.29.0 the engine accepts it as an entry mode of its own:

Listing 9.8: Carrying conditions as a single density altitude

```
# 3000 ft density altitude on a warm day, with the real air
# temperature supplied so powder sensitivity stays correct.
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 --max-range 1000 \
  --density-altitude 3000 --temperature 95
```

The implementation matters here. `--density-altitude` does not bypass the atmosphere model with a bare density—it back-solves an ISA-equivalent station altitude and pressure, so Mach number, the

lapse rate, and segmented-atmosphere behavior are all preserved exactly as they would be for a real altitude.

Precedence is worth committing to memory. Density altitude *supersedes* `--altitude` and `--pressure`/`--pressure-` entirely. An explicit `--temperature` still wins—the real air temperature is needed for powder-temperature sensitivity and the moist speed of sound—and when you supply one, the implied pressure is re-solved so the same density altitude is still reproduced. Omit it and the temperature defaults to the ISA value at that density altitude. The `DA` and `DENSITY_ALTITUDE` columns of a `--location` CSV feed the same path when the flag is absent.

**Zero Day Too**

`--zero-density-altitude` (v0.30.0) declares a zero-day density altitude alongside the other `--zero-*` conditions, following the same precedence rule: it supersedes `--zero-altitude` and `--zero-pressure` for the zero solve only, and an explicit `--zero-temperature` still wins.

Let's compare two scenarios that have similar density altitudes:

Listing 9.9: Similar density altitudes, different true altitudes

```
# Sea level on a very hot day (density alt ~2500 ft)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 104 --pressure 29.92 --altitude 0

# 2500 ft elevation on a standard day (density alt ~2500 ft)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 50 --pressure 27.42 --altitude 2500
```

Compare the drop at 1000 yards between these two commands. Despite dramatically different true altitudes and temperatures, the trajectories will be remarkably similar because the air density—the variable that matters most—is nearly the same.

## 9.5 How BALLISTICS-ENGINE Computes Atmospheric Density

Now that we understand the physics, let's look at exactly how BALLISTICS-ENGINE turns your `--temperature`, `--pressure`, `--humidity`, and `--altitude` inputs into the air density and speed of sound values used by the trajectory solver.

### 9.5.1 The Computation Pipeline

The atmospheric computation follows a layered pipeline:

1. **ICAO Lookup.** If no temperature or pressure override is provided, `calculate_icao_standard_atmosphere()` computes the standard values for the given altitude.
2. **Override Application.** If you specify `--temperature` and/or `--pressure`, those values replace the ICAO standard values. If you specify only one, the other comes from the standard model.
3. **Humidity Correction.** The Arden Buck equation computes the saturation vapor pressure from the temperature. The actual vapor pressure is derived from the relative humidity percentage. Dalton's law splits the total pressure into dry air and water vapor partial pressures.
4. **Density Calculation.** The final air density is computed from the partial pressures and temperature using Equation (22.4).
5. **Speed of Sound.** The speed of sound is computed with a humidity correction based on the mole fraction of water vapor:

$$a = \sqrt{\gamma_{\text{moist}} \cdot R_{\text{moist}} \cdot T} \tag{9.10}$$

where $\gamma_{\text{moist}} = \gamma \cdot (1 - 0.11 \cdot x_v)$ adjusts the heat capacity ratio for the water vapor mole fraction $x_v$, and $R_{\text{moist}} = R_d \cdot (1 + 0.6078 \cdot x_v)$ adjusts the gas constant.

The public entry point is `calculate_atmosphere()` in `src/atmosphere.rs`, which takes altitude, optional temperature and pressure overrides, and humidity percentage, returning a tuple of `(air_density_kg_m3, speed_of_sound_mps)`.

### 9.5.2 The CIPM Density Formula

For situations requiring higher precision, BALLISTICS-ENGINE also implements the CIPM (Comité International des Poids et Mesures) air density formula in `calculate_air_density_cipm()`. This is a metrological-grade calculation that includes:

- The IAPWS-IF97 formulation for saturation vapor pressure (more precise than the Arden Buck equation at extreme temperatures)

- An *enhancement factor* that accounts for the interaction between water vapor and the total pressure:

$$f = \alpha + \beta \cdot P + \gamma \cdot T^2 + \delta \cdot P \cdot T \tag{9.11}$$

  where $\alpha = 1.00062$, $\beta = 3.14 \times 10^{-8}$, $\gamma = 5.6 \times 10^{-7}$, and $\delta = 1.2 \times 10^{-10}$.
- A *compressibility factor* $Z$ that accounts for the non-ideal behavior of real gases, computed from virial coefficients up to fourth order:

$$Z = 1 - \frac{P}{T}\left(A_0 + A_1 t + A_2 t^2 + (B_0 + B_1 t)x_v + (C_0 + C_1 t)x_v^2\right) + \cdots \tag{9.12}$$

- The final CIPM density:

$$\rho = \frac{P \cdot M_a}{Z \cdot R \cdot T}\left(1 - x_v\left(1 - \frac{M_v}{M_a}\right)\right) \tag{9.13}$$

  where $M_a = 28.965\,46 \times 10^{-3}\,\mathrm{kg/mol}$ is the molar mass of dry air and $M_v = 18.015\,28 \times 10^{-3}\,\mathrm{kg/mol}$ is the molar mass of water vapor.

**Standard vs. CIPM Density**

For typical shooting conditions ($-20\,^\circ\mathrm{C}$ to $50\,^\circ\mathrm{C}$, 800–1050 hPa), the standard calculation and the CIPM formula agree to within 0.02%. The CIPM formula becomes more important at extreme conditions or when validating against laboratory-grade measurements. For field shooting, the standard `calculate_atmosphere()` function provides more than sufficient accuracy.

**The CIPM Path Ignored Humidity Until 0.22.13**

The whole point of the CIPM formula is the moist-air term, and until v0.22.13 `calculate_air_density_cipm()` was not getting it. The water-vapor mole fraction $x_v$ was formed by dividing the vapor pressure—in hectopascals—by the total pressure in *pascals*, making $x_v$ a hundred times too small. The function therefore returned essentially dry-air density no matter what humidity you gave it: at 15 °C, 1013.25 hPa, and 50% relative humidity it produced $1.2254\,\mathrm{kg/m^3}$ instead of the CIPM-2007 moist value of about $1.2211\,\mathrm{kg/m^3}$. Moist air is lighter than dry air; the old result had that backwards by omission.

Two related coefficient errors were corrected in v0.22.0: the moist-air speed of sound was applying a specific-humidity coefficient to a mole fraction, and the CIPM enhancement and compressibility coefficients were being fed hPa where the published constants expect Pa. Both were under 1.5%, but both were real.

### 9.5.3 Along-Track Density: The `get_local_atmosphere()` Function

During trajectory integration, the bullet's altitude changes—it follows a parabolic arc that may rise tens of feet above the bore line on a long-range shot. The engine accounts for this by computing atmospheric conditions at the bullet's instantaneous altitude at each integration step.

The function `get_local_atmosphere()` in `src/atmosphere.rs` computes density and speed of sound relative to a base altitude:

Listing 9.10: Local atmosphere at bullet position (`src/atmosphere.rs`)

```
pub fn get_local_atmosphere(
    altitude_m: f64,
    base_alt: f64,
    base_temp_c: f64,
    base_press_hpa: f64,
    base_ratio: f64,
) -> (f64, f64) {
    let height_diff = altitude_m.round() - base_alt;
    let lapse_rate = determine_local_lapse_rate(altitude_m);

    let temp_c = base_temp_c + lapse_rate * height_diff;
    let temp_k = temp_c + 273.15;
    let base_temp_k = base_temp_c + 273.15;

    // Barometric formula for pressure
    let pressure_hpa = if lapse_rate.abs() < 1e-10 {
        base_press_hpa
            * (-G_ACCEL_MPS2 * height_diff
               / (R_AIR * base_temp_k)).exp()
    } else {
        let temp_ratio = temp_k / base_temp_k;
        base_press_hpa
            * temp_ratio.powf(
                -G_ACCEL_MPS2 / (lapse_rate * R_AIR))
    };

    let density_ratio = base_ratio
        * (base_temp_k * pressure_hpa)
        / (base_press_hpa * temp_k);
    let density = density_ratio * 1.225;
    let speed_of_sound = (temp_k * 401.874).sqrt();

    (density, speed_of_sound)
}
```

This function is called from `compute_derivatives()` in `src/derivatives.rs`, which is invoked at every step of the RK4 or RK45 integration. The bullet's current $y$-position (vertical offset from the shooter) is added to the base altitude to compute the atmospheric conditions at that point in space:

Listing 9.11: Atmosphere lookup during integration (`src/derivatives.rs`)

```
let altitude_at_pos = inputs.altitude + pos[1];

let (air_density, speed_of_sound) =
    get_local_atmosphere(
        altitude_at_pos,
        atmos_params.0,  // base_alt
        atmos_params.1,  // base_temp_c
        atmos_params.2,  // base_press_hpa
        atmos_params.3,  // base_density_ratio
    );
```

For a .308 Win shot at 1000 yards, the bullet arc may peak roughly 30–40 ft above the bore line. At this altitude difference, the density change is negligible (about 0.04%). For extremely long-range shots (2000+ yards) with high-arc trajectories, the effect becomes measurable.

## 9.6 Non-Standard Conditions: Hot Days, Cold Days, Altitude

Armed with the theory, let's explore how real-world deviations from the standard atmosphere affect your trajectory. We will use our reference .308 Win load (168 gr SMK, `0.462` G7, 2700 fps) throughout.

### 9.6.1 Hot Days

A hot day reduces air density in two ways: the temperature itself lowers density via the ideal gas law, and hot air can hold more moisture (further reducing density if humidity is high).

Listing 9.12: Desert conditions: 110°F, dry, sea level

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 110 --pressure 29.92 --humidity 10 \
  --altitude 0
```

At 110 °F and sea level, air density drops to approximately 1.098 kg/m$^3$—about 10% below standard. The bullet retains more velocity, arrives sooner, and drops less. If you zeroed on a 59 °F standard day, you will impact *high* in the desert.

### 9.6.2 Cold Days

Cold air is dense air. At 0 °F (−18 °C), sea-level air density climbs to about 1.39 kg/m$^3$—roughly 14% above standard.

Listing 9.13: Winter conditions: 0°F, dry, sea level

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --temperature 0 --pressure 29.92 --humidity 10 \
  --altitude 0
```

The bullet experiences 14% more drag, bleeds velocity faster, and drops significantly more. Combined with the muzzle velocity loss from cold powder (often 50–100 fps or more), cold-weather shooting demands careful re-computation of your trajectory data.

**SAFETY: Cold-Weather Velocity Changes**

Cold conditions affect both the atmosphere *and* your ammunition. A load that produces 2700 fps at 70 °F may only produce 2600–2650 fps at 0 °F. Always chronograph at the shooting temperature when possible. Relying on summertime velocity data for a winter hunt can result in impacts several inches low at extended range. Always verify computational predictions against real-world chronograph data before relying on them.

### 9.6.3 Altitude

Altitude has the largest effect of any single atmospheric variable. At 5000 ft (1524 m), standard pressure drops to about 24.9 inHg (843 hPa), and standard temperature falls to about 41 °F (5.1 °C). The net effect on density is substantial:

Listing 9.14: Altitude comparison at standard temperatures

```
# Sea level (standard day)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
```

```
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 0

# 5,000 ft elevation (standard temp at altitude)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 5000

# 10,000 ft elevation (standard temp at altitude)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 10000
```

When you specify `--altitude` without overriding temperature or pressure, BALLISTICS-ENGINE computes the ICAO standard conditions for that altitude using the layer model discussed in Section 9.1.

**Altitude Alone vs. Altitude Plus Conditions**

Specifying only `--altitude` assumes standard atmospheric conditions at that altitude. For the most accurate results, also provide the actual `--temperature` and `--pressure` measured at your location. A Kestrel weather meter or equivalent will give you station pressure, temperature, and humidity—all you need for a precise atmospheric model.

## 9.7 Practical Impact: Sea Level vs. 5000 ft vs. 10 000 ft

Let's put everything together with a comprehensive comparison. We will compute trajectories at three altitudes under realistic summer conditions, using two different cartridges to show how bullet performance interacts with atmospheric density.

### 9.7.1 The .308 Win at Three Altitudes

Listing 9.15: .308 Win at sea level, summer day

```
# Sea level, 80 degF, 60% humidity
ballistics trajectory \
```

```
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 0 --temperature 80 \
  --pressure 29.92 --humidity 60
```

Listing 9.16: .308 Win at 5,000 ft, summer day

```
# 5,000 ft, 70 degF, 30% humidity
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 5000 --temperature 70 \
  --pressure 24.90 --humidity 30
```

Listing 9.17: .308 Win at 10,000 ft, summer day

```
# 10,000 ft, 55 degF, 20% humidity
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 10000 --temperature 55 \
  --pressure 20.58 --humidity 20
```

Table 9.3 summarizes the expected differences.

Table 9.3: .308 Win, 168 gr SMK trajectory comparison at three altitudes (200-yard zero).

| | **Sea Level** 80 °F | **5,000 ft** 70 °F | **10,000 ft** 55 °F |
|---|---|---|---|
| Approx. density (kg/m$^3$) | 1.17 | 0.99 | 0.84 |
| Density ratio ($\rho/\rho_0$) | 0.955 | 0.808 | 0.686 |
| Remaining velocity at 1000 yd (fps) | ~1180 | ~1250 | ~1320 |
| Drop at 1000 yd (in.) | $\sim -345$ | $\sim -310$ | $\sim -280$ |
| Time of flight (s) | ~1.76 | ~1.66 | ~1.57 |

The numbers tell a compelling story. Moving from sea level to 10 000 ft reduces air density by roughly 28%, which translates to:

- Approximately 140 fps more remaining velocity at 1000 yards
- About 65 inches (18+ MOA) less drop
- Nearly 0.2 seconds shorter time of flight

### 9.7.2 The 6.5 Creedmoor at Three Altitudes

Higher-BC bullets benefit even more from reduced air density because they were already more efficient at overcoming drag. Let's repeat the comparison with a 6.5 Creedmoor, 140 gr ELD-M, `0.610` G7, at 2710 fps:

Listing 9.18: 6.5 Creedmoor at sea level vs. 10,000 ft

```
# Sea level, 80 degF
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 0 --temperature 80 \
  --pressure 29.92 --humidity 60

# 10,000 ft, 55 degF
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 10000 --temperature 55 \
  --pressure 20.58 --humidity 20
```

The 6.5 Creedmoor's higher BC means it sheds velocity more slowly even at sea level. At altitude, where drag is further reduced, the advantage compounds: the bullet stays supersonic to greater distances, arrives faster, and drops less. For precision rifle competitors who travel between sea-level and mountain venues, this underscores why atmospheric data must be updated for every match.

### 9.7.3 Visualizing the Difference

A useful exercise is to compare trajectories side by side using JSON output and your preferred plotting tool:

Listing 9.19: Exporting JSON for comparison plotting

```
# Export sea-level trajectory as JSON
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 0 --temperature 59 \
  --output json

# Export high-altitude trajectory as JSON
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --altitude 10000 --temperature 55 \
  --pressure 20.58 \
  --output json
```

The JSON output includes range, drop, velocity, time of flight, and energy at each step—everything you need to build overlay plots or dope card comparisons.

### 9.7.4 How Much Does Each Variable Contribute?

Not all atmospheric variables contribute equally. Table 9.4 shows the approximate sensitivity of the .308 Win trajectory to each variable at 1000 yards, expressed as the change in drop per unit change in the atmospheric input.

Table 9.4: Atmospheric sensitivity of .308 Win, 168 gr SMK at 1000 yd.

| Variable | Change | $\Delta$ Drop at 1000 yd |
|---|---|---|
| Altitude | $+1000$ ft | $\sim -7$ in. (less drop) |
| Temperature | $+10\,^{\circ}$F | $\sim -2$ in. (less drop) |
| Pressure | $+1$ inHg | $\sim 4$ in. (more drop) |
| Humidity | $+10\%$ RH (at $80\,^{\circ}$F) | $\sim -0.3$ in. (less drop) |

Altitude dominates, followed by pressure and temperature. Humidity is the least significant—but in hot, humid environments, it can add up to a couple of inches of difference at extreme range.

**Prioritize Your Measurements**

If you can only measure one atmospheric variable with a Kestrel or similar device, measure *station pressure*. It captures most of the altitude effect and is the single best predictor of air density deviation from standard. If you can measure two, add temperature. Humidity is a refinement, not a necessity, for most practical shooting.

## Exercises

1. **Temperature Sweep.** Run the .308 Win reference load (`0.462` G7, 168 gr, 2700 fps, 200-yard zero) at temperatures of 0 °F, 30 °F, 59 °F, 90 °F, and 110 °F at sea level. Record the drop at 600 yards and 1000 yards for each. At what temperature range does the drop change most rapidly?

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 --temperature 0
```

2. **Altitude Ladder.** Compute a trajectory for the 6.5 Creedmoor (`0.610` G7, 140 gr, 2710 fps, 200-yard zero) at 0 ft, 2500 ft, 5000 ft, 7500 ft, and 10 000 ft using only the `--altitude` flag (letting BALLISTICS-ENGINE compute standard conditions). How many MOA of correction separates sea level from 10 000 ft at 800 yards?

```
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 800 --altitude 5000
```

3. **Humidity at Extremes.** Compare 0% and 100% humidity at two temperatures: 32 °F and 100 °F. At which temperature does humidity have a measurable effect on the 1000 –yard trajectory?

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 --temperature 100 --humidity 100
```

4. **Matching Density Altitudes.** Find a combination of sea-level temperature and pressure that produces the same drop at 1000 yards as 5000 ft at standard conditions. Use the `--output json` flag to compare trajectories point by point.

5. **Travel Correction.** You have zeroed your .338 Lapua Mag (300 gr Berger Hybrid, `0.818` G7, 2725 fps) at your home range: sea level, 65 °F, 30.10 inHg, 55% humidity. You are traveling to a match at 6200 ft where conditions are forecast as 85 °F, 23.80 inHg, 20% humidity. Compute both trajectories and determine the hold correction at 500 yards, 800 yards, and 1200 yards.

```
# Home range
ballistics trajectory \
  --bc 0.818 --drag-model g7 \
  --velocity 2725 --mass 300 --diameter 0.338 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1200 \
  --altitude 0 --temperature 65 \
  --pressure 30.10 --humidity 55

# Match location
ballistics trajectory \
  --bc 0.818 --drag-model g7 \
  --velocity 2725 --mass 300 --diameter 0.338 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1200 \
  --altitude 6200 --temperature 85 \
  --pressure 23.80 --humidity 20
```

## What's Next

We have seen how the atmosphere acts as an invisible brake on every bullet in flight, and how BALLISTICS-ENGINE models the full ICAO standard atmosphere to account for temperature, pressure, humidity, and altitude. But the atmosphere does more than slow bullets down—it pushes them sideways. In Chapter 10, we turn to the wind: how crosswinds deflect a bullet off its intended path, how headwinds and tailwinds affect drop, and how BALLISTICS-ENGINE handles everything from constant winds to altitude-dependent wind shear profiles. For most shooters, wind is the single hardest variable to master—and accurate modeling is the first step.

# Chapter 10

# Wind Modeling

Ask any long-range shooter what the hardest part of making hits beyond 600 yards is, and the answer is almost universal: the wind. Drop is predictable—once you know your muzzle velocity and atmospheric conditions, the bullet falls at the same rate every time. Wind is different. It varies across the range, changes with altitude above the ground, gusts unpredictably, and shifts direction between your position and the target. A 10 mph full-value crosswind will push a .308 Win, 168 gr Sierra MatchKing roughly 7 inches at 300 yards and nearly 72 inches (6 ft) at 1000 yards.

This chapter explains how wind affects bullet trajectories, how BALLISTICS-ENGINE models wind in its trajectory solver, and how to use the tool's wind features to build better predictions—from constant crosswinds to altitude-dependent wind shear profiles.

## 10.1 How Wind Affects a Bullet

A bullet in flight is not blown sideways the way a leaf is. Instead, wind changes the aerodynamic forces on the projectile by altering the *relative airspeed*—the velocity of the bullet *relative to the air mass it is traveling through*.

Consider a bullet flying at 2700 fps (823 m/s) downrange with a 10 mph (4.47 m/s) crosswind from the right. The air the bullet "sees" is no longer flowing straight over the nose; it arrives at a slight angle. This angle—the *wind-induced yaw*—generates a lateral aerodynamic force.

The key insight is that drag acts along the bullet's velocity *relative to the air*. If the air is moving, the drag vector rotates to oppose the relative velocity, not the ground-track velocity. The lateral component of this rotated drag force is what produces crosswind deflection.

**Relative Velocity**

The *relative velocity* $\mathbf{v}_{\text{rel}}$ of the bullet with respect to the surrounding air mass is:

$$\mathbf{v}_{\text{rel}} = \mathbf{v}_{\text{bullet}} - \mathbf{v}_{\text{wind}} \tag{10.1}$$

All aerodynamic forces—drag, lift, Magnus—are computed using $\mathbf{v}_{\text{rel}}$, not the ground-referenced bullet velocity. In BALLISTICS-ENGINE, this subtraction happens inside the `compute_derivatives()` function in `src/derivatives.rs`, where the wind vector is subtracted from the bullet velocity before drag is computed.

Let's see wind in action:

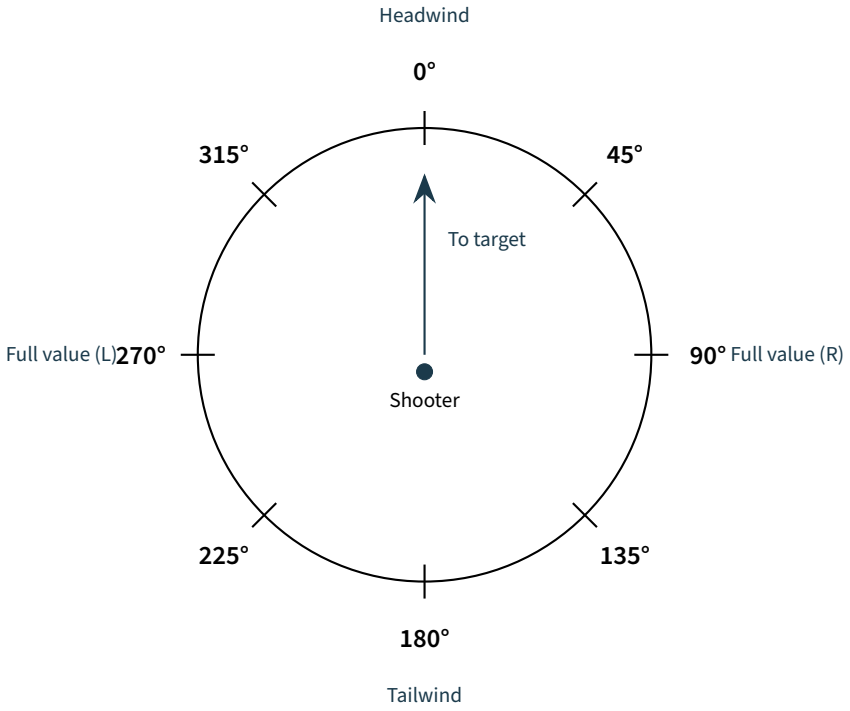
Listing 10.1: A 10 mph crosswind from 90° (full value, from the right)

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90
```

The `--wind-speed` flag takes wind speed in mph (imperial, the default) or m/s (if `--units metric` is set). The `--wind-direction` flag specifies where the wind is coming *from*, in degrees: 0° is a headwind, 90° is from the right, 180° is a tailwind, and 270° is from the left.

## 10.2 The Wind Clock: 0° Headwind Through 360°

The *wind clock* is a shooter's mental model for decomposing wind into crosswind and head/tail components. Imagine you are standing behind your rifle, looking downrange. The wind clock is centered on your position, with 12 o'clock (0°) representing a headwind (blowing from the target toward you), and 6 o'clock (180°) representing a tailwind.




The wind's crosswind component (the part that pushes the bullet left or right) depends on the sine of the wind angle:

$$v_{\text{cross}} = v_{\text{wind}} \cdot \sin(\theta_{\text{wind}}) \tag{10.2}$$

The head/tail component depends on the cosine:

$$v_{\text{head/tail}} = v_{\text{wind}} \cdot \cos(\theta_{\text{wind}}) \tag{10.3}$$

Table 10.1 shows the crosswind component (as a fraction of total wind speed) at common clock positions.

Let's sweep through several wind angles to see how deflection changes:

Listing 10.2: Wind clock sweep: same speed, different directions

```
# Headwind (0 deg) - no lateral deflection
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 0
```

Table 10.1: Wind clock: crosswind fraction ("value") at each position.

| Clock | Angle | $\sin(\theta)$ | Description |
|---|---|---|---|
| 12:00 | $0°$ | 0.00 | Pure headwind (no value) |
| 1:30 | $45°$ | 0.71 | Half value |
| 3:00 | $90°$ | 1.00 | Full value (from right) |
| 4:30 | $135°$ | 0.71 | Half value |
| 6:00 | $180°$ | 0.00 | Pure tailwind (no value) |
| 7:30 | $225°$ | 0.71 | Half value (from left) |
| 9:00 | $270°$ | 1.00 | Full value (from left) |
| 10:30 | $315°$ | 0.71 | Half value |

```
# Quartering wind (45 deg) - half-value crosswind
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 45

# Full crosswind (90 deg) - maximum lateral deflection
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90

# Quartering tailwind (135 deg)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 135
```

Notice that the $45°$ and $135°$ winds produce similar lateral deflection (about 71% of the full-value wind), but different vertical effects: the $45°$ quartering headwind increases drop slightly, while the $135°$ quartering tailwind decreases it.

## 10.3 Crosswind Deflection: The Lag Angle Explanation

The most important wind effect for shooters is crosswind deflection. There is a beautifully intuitive way to understand it, known as the *lag angle* explanation.

**The Lag Rule**

A bullet's crosswind deflection is approximately equal to what would happen if the bullet were "launched at a lag angle behind the wind." If the bullet takes time $t$ to reach a target at range $R$, and the wind is blowing at speed $w$, then the deflection is approximately:

$$\Delta x \approx w \cdot \left( t - \frac{R}{v_0} \right) \tag{10.4}$$

where $v_0$ is the muzzle velocity and $t$ is the actual time of flight. The term $R/v_0$ is the time of flight in a vacuum (no drag). The deflection is proportional to the *difference* between the actual flight time and the vacuum flight time.

This has a profound implication: *crosswind deflection is zero in a vacuum.* If there were no drag, the bullet would fly in a straight line regardless of wind (Newton's first law—the wind has nothing to "push against" if there is no aerodynamic drag). Deflection exists precisely because drag slows the bullet, causing it to "lag" behind the air mass.

The lag rule also explains why:

- High-BC bullets deflect less in wind—they experience less drag, so the lag is smaller.
- Wind deflection increases dramatically at long range—the drag-induced lag accumulates with distance.
- Faster bullets (higher muzzle velocity) deflect less, because the time-of-flight difference is smaller.

Let's compare our .308 Win with the 6.5 Creedmoor in a 10 mph full crosswind:

Listing 10.3: BC effect on wind deflection: .308 Win vs. 6.5 Creedmoor

```
# .308 Win: BC 0.462 G7
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90

# 6.5 Creedmoor: BC 0.610 G7
```

```
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90
```

At 1000 yards, the 6.5 Creedmoor will show roughly 25–30% less crosswind deflection than the .308 Win, despite nearly identical muzzle velocities. The difference is entirely due to the higher BC producing less drag and a shorter lag time.

### 10.3.1 The Math Behind BALLISTICS-ENGINE's Wind Implementation

In BALLISTICS-ENGINE, the wind is represented as a three-dimensional vector in the standard ballistics coordinate system: $x$ = lateral, $y$ = vertical, $z$ = downrange. The conversion from speed and direction to vector components happens in the `WindSock::calc_vec()` method in `src/wind.rs`:

Listing 10.4: Wind vector decomposition (`src/wind.rs`)

```
fn calc_vec(seg: &WindSegment) -> Vector3<f64> {
    let (speed_kmh, angle_deg, _) = *seg;
    let speed_mps = speed_kmh * KMH_TO_MPS;
    let angle_rad = angle_deg * PI / 180.0;

    // Wind convention:
    // 0 deg   = headwind  (from front, -z downrange)
    // 90 deg  = from right (affects -x lateral)
    // 180 deg = tailwind  (from back, +z downrange)
    // 270 deg = from left (affects +x lateral)
    Vector3::new(
        -speed_mps * angle_rad.sin(), // x (lateral)
        0.0,                          // y (vertical)
        -speed_mps * angle_rad.cos(), // z (downrange)
    )
}
```

The convention is consistent: the wind vector represents the air's velocity, and its sign is chosen so that a headwind (0°) opposes the bullet's downrange motion ($-z$) and a 90° wind from the right pushes the bullet to the left ($-x$).

**Wind Direction Sign Fix (0.19.0)**

Versions of the command-line solver *before* 0.19.0 built this wind vector with the opposite sign, so on the CLI a `--wind-direction` of 0° behaved as a tailwind and crosswinds drifted the bullet the wrong way—disagreeing with the convention above and with the language bindings. Version 0.19.0 corrects the CLI/WASM/FFI path to match. If you tuned wind calls against pre-0.19.0 CLI output, re-check their directions.

This wind vector is then subtracted from the bullet velocity to compute the relative airspeed used in all aerodynamic calculations:

$$\mathbf{v}_{\text{rel}} = \mathbf{v}_{\text{bullet}} - \mathbf{v}_{\text{wind}} \tag{10.5}$$

## 10.4 Headwind and Tailwind Effects on Drop

Crosswind gets all the attention, but head and tail winds affect drop—and the effect is not symmetric.

A *headwind* increases the bullet's airspeed relative to the air mass. Higher airspeed means higher drag, which slows the bullet more quickly, increases time of flight, and results in *more drop*.

A *tailwind* decreases the relative airspeed. Lower airspeed means less drag, higher retained velocity, shorter flight time, and *less drop*.

Listing 10.5: Headwind vs. tailwind effects on drop

```
# No wind (baseline)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000

# 20 mph headwind
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 20 --wind-direction 0

# 20 mph tailwind
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
```

```
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 20 --wind-direction 180
```

At 1000 yards with a 20 mph headwind, expect several additional inches of drop compared to still air. The tailwind reduces drop by a comparable amount.

**Head/Tail Wind Is Often Ignored**

Many shooters focus exclusively on crosswind and ignore the head/tail component. In gusty conditions where the wind oscillates between headwind and tailwind (such as shooting across a valley with swirling winds), the vertical impact scatter can be as large as the horizontal scatter. Always account for the full wind vector, not only the crosswind component.

### 10.4.1 Quartering Winds

Quartering winds—those between 0°/180° and 90°/270°—combine both effects. A 45° quartering headwind from the right will push the bullet left *and* increase drop. A 135° quartering tailwind from the right will push the bullet left *and* decrease drop.

BALLISTICS-ENGINE handles quartering winds naturally because the wind vector is fully three-dimensional. The trigonometric decomposition in `calc_vec()` correctly splits any wind angle into its lateral and downrange components.

Listing 10.6: Quartering wind: combined lateral and vertical effect

```
# 15 mph quartering headwind from right (45 deg)
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 15 --wind-direction 45
```

**Decomposing Quartering Winds Mentally**

For a quick mental estimate with a quartering wind, use the "half-value" rule: a 45° wind produces about 71% of the full-value crosswind deflection, and 71% of the head/tail drop change. This is exact at 45° ($\sin 45° = \cos 45° \approx 0.707$) and provides a useful field estimate.

## 10.5 Wind Shear: Vertical Wind Gradients

So far we have treated the wind as constant from the ground to any altitude the bullet reaches. In reality, wind speed increases with height above the ground, and wind direction can rotate with altitude—especially near the surface where terrain and obstacles create complex boundary layers. This phenomenon is called *wind shear*.

**Wind Shear**

*Wind shear* is the change in wind speed and/or direction with altitude. In the atmospheric boundary layer (roughly the lowest 1000 m), wind speed typically increases logarithmically with height due to surface friction. Wind direction can also rotate with altitude, a phenomenon modeled by the Ekman spiral.

Wind shear matters for long-range ballistics because the bullet's trajectory arc takes it above the ground level where the shooter measures the wind. A bullet fired at 1000 yards may arc 30–40 ft above the bore line; at 2000 yards, the arc can exceed 150 ft. At those heights, the wind may be substantially stronger than what you feel at ground level.

### 10.5.1 Wind Shear Models

BALLISTICS-ENGINE implements four wind shear models, defined in `src/wind_shear.rs` as the `WindShearModel` enum:

Listing 10.7: Wind shear model types (`src/wind_shear.rs`)

```
pub enum WindShearModel {
    None,
    Logarithmic,
    PowerLaw,
    EkmanSpiral,
    CustomLayers,
}
```

**Logarithmic Profile.** The logarithmic wind profile is the most physically grounded model for neutral atmospheric stability. Wind speed at height $z$ is:

$$U(z) = U_{\text{ref}} \cdot \frac{\ln(z/z_0)}{\ln(z_{\text{ref}}/z_0)} \tag{10.6}$$

where $U_{\text{ref}}$ is the wind speed at reference height $z_{\text{ref}}$ (typically 10 m, the standard meteorological measurement height) and $z_0$ is the surface roughness length.

The roughness length depends on terrain:

- Open water: $z_0 \approx 0.0002\,\text{m}$
- Short grass, open field: $z_0 \approx 0.03\,\text{m}$ (the engine default)
- Crops, low bushes: $z_0 \approx 0.1\,\text{m}$
- Forest, suburban: $z_0 \approx 1.0\,\text{m}$

**Power Law Profile.** A simpler model often used in engineering:

$$U(z) = U_{\text{ref}} \cdot \left( \frac{z}{z_{\text{ref}}} \right)^{\alpha} \tag{10.7}$$

where $\alpha$ is the power law exponent. The classical value for neutral stability is $\alpha = 1/7 \approx 0.143$ (the "one-seventh power law"). This is the default exponent in BALLISTICS-ENGINE.

**Ekman Spiral.** Above the surface boundary layer, the Coriolis force causes wind direction to rotate with altitude—a phenomenon known as the Ekman spiral. Near the surface, friction slows the wind and causes it to "back" (turn counter-clockwise in the Northern Hemisphere) relative to the geostrophic wind aloft. The BALLISTICS-ENGINE implementation linearly interpolates both speed and direction between the surface wind and a geostrophic wind (default: 30° backing, 1.5× surface speed) over a 1000 m boundary layer depth.

**Custom Layers.** For the most control, you can define wind conditions at specific altitudes. The engine linearly interpolates between adjacent layers.

### 10.5.2 Enabling Wind Shear

To enable wind shear in BALLISTICS-ENGINE, use the `--enable-wind-shear` flag along with `--wind-shear-model`:

Listing 10.8: Trajectory with logarithmic wind shear

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90 \
  --enable-wind-shear --wind-shear-model logarithmic
```

Listing 10.9: Power law wind shear model

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
```

```
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90 \
  --enable-wind-shear --wind-shear-model power_law
```

Without `--enable-wind-shear`, the wind remains constant at all altitudes—equivalent to `WindShearModel::None`.

### 10.5.3 How Wind Shear Enters the Trajectory Solver

During trajectory integration, the engine queries the wind vector at the bullet's current 3D position. In the `compute_derivatives_vec()` function in `src/trajectory_integration.rs`, the wind lookup branches based on whether wind shear is enabled:

Listing 10.10: Wind lookup during integration (`src/trajectory_integration.rs`)

```
let wind_vector = if !params.wind_segments.is_empty() {
    if params.enable_wind_shear
       && params.wind_shear_model != "none" {
        crate::wind_shear::get_wind_at_position(
            &pos,
            &params.wind_segments,
            params.enable_wind_shear,
            &params.wind_shear_model,
            params.shooter_altitude_m,
        )
    } else {
        // Simple constant wind
        let seg = &params.wind_segments[0];
        let wind_speed_mps = seg.0 * 0.2777778;
        let wind_angle_rad = seg.1.to_radians();
        Vector3::new(
            -wind_speed_mps * wind_angle_rad.sin(),
            0.0,
            -wind_speed_mps * wind_angle_rad.cos(),
        )
    }
} else {
    Vector3::zeros()
};
```

When wind shear is active, the `get_wind_at_position()` function in `src/wind_shear.rs` uses the bullet's $y$-coordinate (vertical position relative to the shooter) plus the shooter's absolute altitude to

determine the wind at the bullet's current height. The selected shear model then scales the base wind speed (and optionally rotates the direction, in the Ekman case) according to that altitude.

**Numerical Stability and Wind Shear**

The adaptive RK45 integrator automatically reduces its step size when wind shear is enabled, particularly for long-range trajectories. This is because the wind function now varies with position, making the differential equations stiffer. For ranges beyond 800 m, the engine caps the step size at 10 ms (vs. the normal adaptive maximum) to maintain accuracy. See the `effective_max_step` logic in `integrate_trajectory()` in `src/trajectory_integration.rs`.

### 10.5.4 When Wind Shear Matters

For a typical 600 –yard shot where the bullet arc peaks at perhaps 8–10 ft above bore line, wind shear has a modest effect. The logarithmic profile predicts that wind speed at 10 ft (3 m) above ground is about 83% of the speed at the reference height of 10 m. At 40 ft (12 m), it is about 105%—only 5% stronger.

Wind shear becomes significant for:

- **Extreme long range (1500+ yards)**: Bullet arcs of 100+ ft mean the bullet spends substantial time in faster-moving air.
- **Shooting from valleys or depressions**: Surface wind may be calm while wind aloft is strong.
- **Urban or forested environments**: Buildings and trees create a deep surface boundary layer with sharp wind gradients.

Listing 10.11: Wind shear comparison at extreme range

```
# No shear (constant wind at all heights)
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 15 --wind-direction 90

# Logarithmic shear
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 15 --wind-direction 90 \
  --enable-wind-shear --wind-shear-model logarithmic
```

Compare the crosswind deflection values between these two runs. The difference gives you a sense of how much the "extra" wind above the ground contributes to total deflection.

## 10.6 Modeling Variable Wind with BALLISTICS-ENGINE

Real wind is rarely constant across a 1000 –yard range. Terrain features, vegetation, buildings, and thermal effects create zones where the wind speed and direction change with distance downrange. BALLISTICS-ENGINE supports *multi-segment wind*—different wind conditions at different range bands.

### 10.6.1 The WindSock Architecture

Internally, multi-segment wind is managed by the `WindSock` struct in `src/wind.rs`. Each segment is a tuple of `(speed_kmh, angle_deg, until_distance_m)`:

Listing 10.12: WindSock: multi-segment wind (`src/wind.rs`)

```
pub type WindSegment = (f64, f64, f64);

pub struct WindSock {
    winds: Vec<WindSegment>,    // Sorted by distance
    current: usize,             // Current segment index
    next_range: f64,            // Distance where next segment starts
    current_vec: Vector3<f64>, // Current wind vector
}
```

The segments are sorted by distance. As the trajectory integration proceeds, `WindSock::vector_for_range()` advances through segments as the bullet's downrange position increases. A stateless variant, `vector_for_range_stateless()`, is also provided for use in numerical integration where the same range may be queried multiple times:

Listing 10.13: Stateless wind lookup for numerical integration

```
pub fn vector_for_range_stateless(
    &self, range_m: f64
) -> Vector3<f64> {
    if range_m.is_nan() {
        return Vector3::zeros();
    }
    for segment in &self.winds {
        if range_m < segment.2 {
            return Self::calc_vec(segment);
        }
```

```
    }
    Vector3::zeros() // Beyond all segments
}
```

Beyond the last defined segment, wind drops to zero. This is an important design choice: it prevents the engine from extrapolating wind conditions beyond the shooter's defined range.

### 10.6.2 Checked Construction for Library Callers

If you are driving the engine from Rust rather than from the command line, note that the original `WindSock` constructors are infallible: they accept whatever you give them and sort it. A `NaN` speed or a negative until-distance therefore reaches the integrator.

Version 0.26.0 added a checked path alongside them:

Listing 10.14: Fallible segmented-wind construction (`src/wind.rs`)

```
pub enum WindSegmentError { /* index, field, rule */ }

impl WindSock {
    pub fn try_new(segments: Vec<WindSegment>)
        -> Result<Self, WindSegmentError>;
}

impl TryFrom<Vec<WindSegment>> for WindSock { /* ... */ }

pub fn try_integrate_trajectory(/* ... */)
    -> Result<TrajectoryResult, BallisticsError>;
pub fn try_solve_trajectory_rust(/* ... */)
    -> Result<TrajectoryResult, BallisticsError>;
```

The checked constructors reject a non-finite or negative `speed_kmh`, a non-finite `angle_deg`, a non-finite or non-positive `until_m`, and a non-finite `vertical_mps`—*before* sorting, precomputation, or the production of a single trajectory point. The error carries the caller's segment index, the offending field, and the rule that was violated, so a rejected segment can be pointed at rather than merely reported. `wind::validate_wind_segments` is public for callers who want to check without constructing.

The infallible signatures are unchanged and documented as legacy entry points; their error text is byte-identical to earlier releases.

### 10.6.3 Multi-Segment Wind via the CLI

The simplest way to use wind in BALLISTICS-ENGINE is with the `--wind-speed` and `--wind-direction` flags, which create a single constant-wind segment covering the entire range. For multi-segment wind, profiles (covered in Chapter 8) allow you to define wind conditions at different range bands.

For a constant wind across the full range:

Listing 10.15: Single-segment constant wind

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 10 --wind-direction 90
```

For precision rifle competition, where wind conditions vary along the course of fire, multi-segment wind modeling through profiles allows you to capture the complex wind patterns that single-value inputs cannot represent.

### 10.6.4 Multi-Segment Wind: A Practical Example

Consider a 1000 –yard range with a treeline creating a wind shadow for the first 200 yards, strong crosswind in the open middle section, and calmer conditions near the target where a hillside provides partial shelter:

- 0–200 yd: 3 mph from 90° (sheltered)
- 200–700 yd: 15 mph from 80° (open)
- 700–1000 yd: 8 mph from 100° (partial shelter)

In the `WindSock` representation, these would be three segments:

Listing 10.16: Three-segment wind model (internal representation)

```
// (speed_kmh, angle_deg, until_distance_m)
let segments = vec![
    (4.83, 90.0, 182.88),   // 3 mph until 200 yd
    (24.14, 80.0, 640.08),  // 15 mph until 700 yd
    (12.87, 100.0, 914.40), // 8 mph until 1000 yd
];
```

Since version 0.19.0 you can supply these segments directly on the command line with the repeatable `--wind-segment` flag, in `SPEED:ANGLE:UNTIL_DISTANCE` form. Speed and until-distance follow

`--units` (mph and yards in imperial, m/s and metres in metric); the angle is degrees in the same wind-from convention as `--wind-direction`:

Listing 10.17: The same three-segment wind, from the command line

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --max-range 1000 \
  --wind-segment 3:90:200 \
  --wind-segment 15:80:700 \
  --wind-segment 8:100:1000
```

Each segment applies from the previous boundary out to its until-distance as a *step function*—there is no interpolation between segments—and the wind is treated as zero beyond the last segment (the solver prints a coverage warning if the segments do not reach `--max-range`). The `--wind-segment` flag overrides the scalar `--wind-speed`/`--wind-direction`, and is not compatible with `--enable-wind-shear`.

The total deflection from this multi-segment wind will be substantially different from a single 10 mph average wind, because the bullet is moving fastest (and thus least affected by wind per unit of time) in the first segment, and slowest (most affected) in the last segment.

**Two Segmented-Wind Fixes in 0.22.0**

The rule that wind goes to zero past the last segment held on the main solver but not on the shear-enabled fast path, which instead fell back to the *first* segment's wind. And `fast_integrate_with_segments`—the entry point the language bindings call—was applying only the first segment for the whole flight rather than stepping through them by downrange distance. Both are corrected in v0.22.0, so a segmented-wind solve now gives the same answer through the library and through the CLI.

**Near vs. Far Wind**

Wind near the target has a disproportionately large effect on deflection compared to wind near the shooter. This is counterintuitive but follows directly from the lag rule (Section 10.3). The bullet is slower at longer ranges, so it spends more time being pushed by each mph of wind. A 10 mph wind in the last 200 yards produces roughly three times the deflection of the same wind in the first 200 yards.

## 10.7 Entering Wind the Way You Think About It

Degrees are precise and nobody uses them. On the line you hear "full value from three o'clock" and "ten thirty, half value". Since v0.31.0 every `--wind-direction` flag—on **`trajectory`**, `monte-carlo`, `come-ups`, `lead`, `range-table`, `compare` and **`profile`** `save`— accepts marked clock positions alongside plain degrees:

Listing 10.18: Three spellings of the same wind

```
# All identical: a full-value wind from the shooter's right.
ballistics trajectory --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 --auto-zero 200 \
  --sight-height 1.5 --wind-speed 10 --wind-direction 90

ballistics trajectory --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 --auto-zero 200 \
  --sight-height 1.5 --wind-speed 10 --wind-direction 3oc

ballistics trajectory --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 --auto-zero 200 \
  --sight-height 1.5 --wind-speed 10 --wind-direction 3:00
```

The conversion is $(\mathrm{H} \bmod 12) \times 30 + \mathrm{MM} \times 0.5$ degrees, wind-FROM, so twelve o'clock is a headwind at $0°$—the convention established by the 0.19.0 sign fix. `10h30` and `10:30` both mean $315°$. **Bare numbers remain degrees everywhere**, so every existing invocation is unchanged.

Inside `--wind-segment` the angle field takes the colon-free marked forms only—`10:3oc:400`—because `10:30:400` already has a meaning there: speed 10, angle 30, until 400. Malformed clock tokens (an hour outside 1–12, minutes outside 0–59) fail loudly with the rule in the message.

### 10.7.1 Compass Bearings

A Kestrel with a compass, or a weather report, gives you a wind bearing in earth-fixed degrees—not relative to a rifle you have not pointed yet. `--wind-ref` lets you enter it that way:

Listing 10.19: Wind FROM the northeast, on a shot fired due east

```
ballistics trajectory --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 --auto-zero 200 \
  --sight-height 1.5 --max-range 1000 \
  --wind-speed 12 --wind-direction 45 \
  --wind-ref compass --shot-direction 90
```

`shooter` is the default and is today's behavior byte-identically. `compass` treats *every* wind direction the run consumes—the flag, a location CSV's `WIND_DIR` column, and each `--wind-segment` angle—as an absolute bearing with 0 at north, and re-references it once at the input boundary:

$$\text{relative} = \text{bearing} - \text{shot azimuth} \tag{10.8}$$

normalised to $[0, 360)$, wind-FROM on both sides, so nothing downstream of that boundary ever sees a bearing. In the example, $45 - 90 = -45 \to 315$: a wind from the shooter's front-left. The sanity anchor to remember is that wind from the north on a shot fired due north is a pure headwind.

**Compass Mode Requires a Shot Direction**

There is no sensible default for the rifle's bearing, so `compass` without `--shot-direction` is a hard error that names the flag—never a silent fall back to shooter-relative. (`monte-carlo` gained `--shot-direction` for exactly this reason.) Clock positions are rejected in compass mode too: a clock position is shooter-relative by definition.

Monte Carlo converts the base direction *before* any dispersion sampling, so a direction sigma disperses around the converted value rather than around a bearing.

## 10.8 Truing the Wind Call

Everything so far has run forwards: you supply a wind and the engine tells you where the bullet goes. `true-wind` runs it backwards. You shot a group, it landed to one side, and you want to know what crosswind would actually produce that:

Listing 10.20: Back-solving the wind from an observed miss

```
ballistics true-wind \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --miss 700:3.5 --twist-rate 11 \
  --zero-distance 200 --sight-height 1.5
```

Listing 10.21: The solved wind, and what was subtracted from it

```
=== EFFECTIVE WIND TRUING (from observed horizontal miss) ===

  Range (yd)     Miss (in)   Spin/Cor (in)        Wind (mph)  Resid (in)
  ------------------------------------------------------------------------
       700.0         +3.50           +4.67             -0.25      -0.000
  ------------------------------------------------------------------------

  Effective crosswind:    -0.25 mph
```

```
Effects subtracted:  spin drift
NOT subtracted (absorbed into the solved wind): Coriolis (supply --latitude and --
  shot-direction to subtract it)
```

Read that result carefully, because it is a good lesson in what truing actually does. The group landed 3.5 inches right—and the answer is that there was essentially *no* wind. A 1:11″ twist drifts this bullet 4.67 inches right at 700 yards all by itself, so the observed miss is less than the spin drift and the residual crosswind is a quarter of a mile per hour from the right. A shooter who attributed that group to wind would have learned the wrong lesson entirely.

**A Horizontal Miss Is Not Purely Wind**

This is why `--twist-rate` is *required*: gyroscopic spin drift is always modelled and subtracted, and without it a 1:11″ .308's roughly 3.5 inches of right drift at 700 yards would be read as wind. `--latitude` and `--shot-direction` are optional but must be supplied *together*, and add Coriolis.
Whatever the model had no data for stays absorbed in the solved wind and is named in the report's "NOT subtracted" line. A contaminated number is never presented as pure wind.

`--miss` is repeatable as `RANGE:RIGHT[:SIGMA]`. Each observation is fitted independently by a bracketed root find over ±100 mph— deflection is monotone and near-linear in crosswind speed—and the combined answer is the plain mean, or an inverse-variance weighted mean with its own $1\sigma$ when *every* observation carries a sigma. Mixing the two is a hard error rather than a silent blend. A miss that no wind in that band can produce is rejected with a diagnostic naming the band, not clamped into a plausible-looking number.

Supply `--called-wind` and the report adds a wind-call correction factor: solved divided by called, which is the number you actually want to carry forward to the next match.

**Signs, in One Place**

`--miss` positive means the impact landed *right* of aim. A solved wind positive means the wind is *from* the shooter's left—nine o'clock—pushing impacts right. A right-hand twist drifts right. `RANGE` follows `--units`, but the linear miss and its sigma are **inches in both unit systems**, following the `--drop-unit in` precedent. Wind speeds follow `--units`.

**Tracking Factors Do Not Apply Here**

Because `--miss` is a tape measurement off a target rather than a dial reading, the scope tracking correction factors of Section 24.6 deliberately do not apply, and the command exposes none.

## 10.9 Practical Wind-Reading Strategies

Computational wind modeling is valuable, but it relies on one critical input: an accurate wind estimate. The best solver in the world cannot help if you feed it a 5 mph wind call when the actual wind is 12 mph. Here are strategies for building accurate wind inputs for BALLISTICS-ENGINE.

### 10.9.1 Using a Wind Meter

A handheld wind meter (such as a Kestrel 5700, which also provides density altitude) gives you wind speed and direction at your position. This is your best starting point.

Listing 10.22: Using Kestrel readings for atmospheric and wind inputs

```
# Kestrel reading: 8 mph from 2 o'clock (60 deg), 72 degF,
# station pressure 25.10 inHg, 45% humidity, 4200 ft
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 800 \
  --wind-speed 8 --wind-direction 60 \
  --temperature 72 --pressure 25.10 \
  --humidity 45 --altitude 4200
```

**Your Wind Meter Reads at Your Position**

The wind at the shooting position is not necessarily the wind at the target or at mid-range. Mirage, flags, vegetation movement, and experience reading terrain are essential for estimating downrange wind. Use your wind meter reading as a baseline and adjust based on what you observe between the muzzle and the target.

### 10.9.2 Reading Mirage

Through a spotting scope at 15–25 × magnification, heat mirage appears as shimmering waves rising from the ground. The direction and speed of the mirage flow indicate wind conditions:

- **Straight up ("boiling"):** Calm or very light wind ($< 3$ mph)
- **Gentle lean:** Light wind (3–5 mph)
- **Strong lean:** Moderate wind (5–8 mph)
- **Flat/running:** Wind exceeds about 12 mph (mirage is blown flat and becomes difficult to read)

Mirage reading is an art that complements computational tools—it provides real-time wind information at the range you are focusing on, not only at the shooter's position.

### 10.9.3 Using Flags and Indicators

Many ranges have flags at intervals. A wind flag at 45° from its pole indicates roughly 8–12 mph of wind. By reading multiple flags, you can estimate a wind profile across the range and construct a multi-segment wind call.

### 10.9.4 The Bracket Technique

When uncertain about the wind, run two trajectories: one with your lower estimate and one with your upper estimate. This gives you the range of possible impacts.

Listing 10.23: Bracketing the wind call

```
# Lower estimate: 5 mph
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 600 \
  --wind-speed 5 --wind-direction 90

# Upper estimate: 12 mph
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 600 \
  --wind-speed 12 --wind-direction 90
```

The difference in deflection between these two runs defines your *wind bracket*. For a precision rifle match, you might hold for the midpoint of the bracket, accepting that your impact will be within the bracketed range. For hunting, you might hold for the more conservative (higher deflection) estimate to avoid a miss entirely.

### 10.9.5 Monte Carlo for Wind Uncertainty

For a more rigorous treatment of wind uncertainty, BALLISTICS-ENGINE's Monte Carlo simulation (covered in Chapter 6) can randomize wind speed and direction across many iterations, producing a probability distribution of impacts:

Listing 10.24: Monte Carlo with wind variation

```
ballistics monte-carlo \
  --bc 0.462 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --target-distance 600 \
  --wind-speed 8 --wind-direction 90 \
  --num-sims 1000
```

The Monte Carlo engine introduces randomized variations in wind speed and direction (among other parameters), giving you a realistic picture of where your shots will land given the uncertainty in your wind call.

### 10.9.6 Developing Your Wind Estimation Workflow

A practical workflow for using BALLISTICS-ENGINE with wind:

1. **Measure:** Read wind speed and direction at your position with a Kestrel or similar device.
2. **Observe:** Read mirage, flags, or vegetation at mid-range and near the target. Estimate whether downrange wind is the same, stronger, or weaker than at your position.
3. **Model:** Enter your best wind estimate into BALLISTICS-ENGINE. For a single-value call, use `--wind-speed` and `--wind-direction`.
4. **Bracket:** If uncertain, run a high and low estimate to bound the deflection.
5. **Shoot and correct:** Observe the splash or trace of your first round. Update your wind call based on where it actually impacted, and re-run the trajectory.
6. **Record:** Log wind conditions with each string of fire. Over time, this builds an invaluable dataset for learning to read wind at familiar ranges.

**Wind Speed and BC: A Combined Effect**

When comparing cartridges for a windy venue, look at the *product* of wind deflection and time of flight, not the deflection number alone. A high-BC bullet like the 6.5 Creedmoor 140 gr ELD-M (`0.610` G7) will show roughly 30% less wind deflection than the .308 Win 168 gr SMK (`0.462` G7) at the same range. Over many shots in a match, this translates directly to higher scores in windy conditions.

## Exercises

1. **Wind Clock Sweep.** Using the .308 Win reference load (168 gr SMK, `0.462` G7, 2700 fps, 200-yard zero), run trajectories with a 10 mph wind at 0°, 30°, 60°, 90°, 120°, 150°, and 180°. Record both the crosswind deflection and the change in drop at 600 yards for each angle. Plot deflection vs. angle—does it follow a sine curve?

```
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 600 \
  --wind-speed 10 --wind-direction 30
```

2. **BC and Wind.** Compare the crosswind deflection at 800 yards for these three cartridges in a 10 mph full crosswind:

   - .223 Rem, 77 gr SMK, `0.372` G7, 2750 fps
   - .308 Win, 168 gr SMK, `0.462` G7, 2700 fps
   - 6.5 Creedmoor, 140 gr ELD-M, `0.610` G7, 2710 fps

   How does the ratio of wind deflection compare to the inverse ratio of BCs?

```
ballistics trajectory \
  --bc 0.372 --drag-model g7 \
  --velocity 2750 --mass 77 --diameter 0.224 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 800 \
  --wind-speed 10 --wind-direction 90
```

3. **Wind Shear Comparison.** Run the 6.5 Creedmoor load at 1000 yards with a 12 mph crosswind from 90°, with each of the three shear models (none, logarithmic, power_law). How much additional deflection does wind shear add compared to constant wind?

```
ballistics trajectory \
  --bc 0.610 --drag-model g7 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 12 --wind-direction 90 \
  --enable-wind-shear --wind-shear-model power_law
```

4. **Headwind vs. Tailwind Asymmetry.** Using the .338 Lapua Mag (300 gr Berger Hybrid, 0.818 G7, 2725 fps, 200-yard zero), compare a 20 mph headwind vs. a 20 mph tailwind at 1000 yards. Is the additional drop from the headwind exactly equal to the reduced drop from the tailwind, or is there an asymmetry? Why?

```
# Headwind
ballistics trajectory \
  --bc 0.818 --drag-model g7 \
  --velocity 2725 --mass 300 --diameter 0.338 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 20 --wind-direction 0

# Tailwind
ballistics trajectory \
  --bc 0.818 --drag-model g7 \
  --velocity 2725 --mass 300 --diameter 0.338 \
  --auto-zero 200 --sight-height 1.5 \
  --max-range 1000 \
  --wind-speed 20 --wind-direction 180
```

5. **Near Wind vs. Far Wind.** This exercise explores the disproportionate effect of wind at different points along the trajectory. Using a profile or running separate computations, compare:

   - 10 mph crosswind for the first 500 yards only, calm beyond
   - 10 mph crosswind for the last 500 yards only, calm before

   Which scenario produces more deflection at 1000 yards? Explain why using the lag rule from Section 10.3.

## What's Next

With the atmosphere and wind covered, we have addressed the two environmental factors that most influence every bullet's flight. In Part IV, we shift our focus from *what the air does to the bullet* to *what the bullet does in the air*: drag modeling and ballistic coefficients. Chapter 11 introduces the G1 and G7 drag models, explains how drag tables work, and shows how BALLISTICS-ENGINE looks up and interpolates drag coefficients from doppler-derived data. Understanding drag models is the key to unlocking accurate trajectory predictions across the full supersonic-through-transonic velocity range.

# Part IV

# Drag & BC Modeling

# Chapter 11

# Drag Models & Tables

Every trajectory computation reduces, at its core, to a single question: *how much does the air slow this bullet down?* The answer lives in the **drag model**—a function that maps a projectile's Mach number to a drag coefficient. Choose the wrong model and your drop predictions may be off by inches at 400 yards and feet at 1 000. Choose the right one and the mathematics almost disappears: a well-matched drag model, multiplied by a single ballistic coefficient, reproduces measured drag to within a few percent from muzzle to subsonic.

This chapter explains what drag models are, how the classical G1 and G7 standards differ, when to use each one, and how BALLISTICS-ENGINE evaluates drag internally. We will walk through the drag tables, the concept of form factors, and the engine's interpolation pipeline—from the lookup in `src/drag.rs` through Catmull–Rom cubic interpolation to the final $C_D$ value that enters the trajectory integrator.

## 11.1 What Is a Drag Model?

### 11.1.1 Aerodynamic Drag in One Equation

When a projectile moves through air at velocity $V$, it experiences a retarding force:

$$F_D = \frac{1}{2}\,\rho\,V^2\,C_D\,A \tag{11.1}$$

where $\rho$ is the air density, $A$ is the projectile's reference area (usually $\pi d^2/4$ for calibre $d$), and $C_D$ is the *drag coefficient*—a dimensionless number that encapsulates everything about the projectile's shape, surface roughness, and interaction with the airflow at that speed.

**Drag Coefficient ($C_D$)**

The drag coefficient is the ratio of the actual drag force on the projectile to the dynamic pressure ($\frac{1}{2}\rho V^2$) acting over the reference area. It is a function of Mach number, Reynolds number, angle of attack, and projectile geometry. In practical ballistics, $C_D$ is treated primarily as a function of Mach number, with the other dependencies absorbed into the ballistic coefficient.

The drag coefficient is not constant. As a bullet decelerates from supersonic to transonic to subsonic flight, $C_D$ changes dramatically: it rises sharply as shock waves form near Mach 1, peaks somewhere between Mach 1.0 and 1.3, and then falls off in both directions. A typical spitzer boat-tail bullet might have $C_D \approx 0.12$ at Mach 0.5, $C_D \approx 0.38$ at Mach 1.0, and $C_D \approx 0.25$ at Mach 2.0.

### 11.1.2 The Standard Projectile Idea

Measuring the exact $C_D(M)$ curve for every commercial bullet ever made would be prohibitively expensive. The solution, developed at the U.S. Army Ballistic Research Laboratory (BRL) in the early twentieth century, was to define a set of *standard projectiles*—idealised shapes whose drag curves were measured once with great precision. Any real bullet is then described by a single number, the **ballistic coefficient** (BC), that scales the standard curve to match the real bullet's drag.

$$\mathrm{BC} = \frac{m}{C_D^{\mathrm{std}}\, d^2\, i} \tag{11.2}$$

where $m$ is the bullet's mass, $d$ is its calibre, and $i$ is the *form factor*—the ratio of the bullet's drag coefficient to the standard projectile's drag coefficient at the same Mach number. We will return to form factors in Section 11.7.

The key insight is this: once you pick a standard projectile (G1, G7, etc.), you only need one number—the BC—to describe an entire bullet's drag curve. The standard drag table supplies $C_D^{\mathrm{std}}(M)$; the BC scales it.

Listing 11.1: A basic trajectory using the G1 drag model

```
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000
```

### 11.1.3 A Family of Standards

Over the decades, the BRL (and later the U.S. Army Research Laboratory) defined multiple standard projectiles. The most widely used today are G1 and G7, but the full family includes G2, G5,

G6, G8, GL (also called GS), and GI. Each was designed to approximate a different class of projectile shape. BALLISTICS-ENGINE supports all of them through the `DragModel` enum defined in `src/drag_model.rs`:

Listing 11.2: The DragModel enum from `src/drag_model.rs`

```
pub enum DragModel {
    G1,  // Flat-base, 2-caliber ogive
    G2,  // Aberdeen J projectile
    G5,  // Short 7.5-degree boat tail
    G6,  // Flat-base, 6-caliber secant ogive
    G7,  // Long 7.5-degree boat tail tangent ogive
    G8,  // Flat-base, 10-caliber secant ogive
    GI,  // Ingalls tables
    GS,  // Spherical (round ball)
}
```

The engine resolves the model from the CLI flag `--drag-model` via a case-insensitive string parser (`DragModel::from_str()`), so `g1`, `G1`, and even `g7` all work as expected.

## 11.2 The G1 Standard Projectile

### 11.2.1 History and Shape

The G1 standard is the oldest and most widely used drag model in sporting ballistics. It was defined in 1881 by the Krupp factory and later adopted by the U.S. Army. The G1 reference projectile is a flat-base design with a 2-caliber-radius tangent ogive nose—essentially a blunt, stubby bullet shape that looks nothing like a modern long-range match projectile.

Despite its age, G1 remains the default in most ballistic calculators and on most ammunition packaging. When a bullet manufacturer publishes a BC without specifying the standard, it is almost always a G1 BC.

### 11.2.2 The G1 Drag Curve

The G1 drag table in BALLISTICS-ENGINE is stored as a set of (Mach, $C_D$) pairs. The full table (79 data points from Mach 0.0 to Mach 5.0) lives in `src/drag_tables.rs`. A condensed version is compiled directly into the fallback array in `src/drag.rs`:

Listing 11.3: G1 fallback drag data from `src/drag.rs` (condensed)

```
let fallback_data = [
    (0.0, 0.2629),   // Subsonic -- low base drag
    (0.5, 0.2695),
```

```
    (0.7, 0.2817),
    (0.9, 0.3012),   // Drag beginning to rise
    (1.0, 0.4805),   // Transonic spike
    (1.2, 0.6318),   // Near-peak drag
    (1.3, 0.6440),   // Peak region
    (1.5, 0.6372),   // Beginning to decrease
    (2.0, 0.5934),
    (3.0, 0.5133),
    (5.0, 0.4988),
];
```

Several features stand out:

1. **Low subsonic drag**: $C_D \approx 0.20$–$0.27$ below Mach 0.8. This is relatively high compared to a modern boat-tail bullet because the G1 shape has a flat base with significant base drag.
2. **Sharp transonic rise**: $C_D$ nearly doubles between Mach 0.9 and Mach 1.0 as shock waves form.
3. **Broad supersonic plateau**: $C_D$ peaks around 0.66 near Mach 1.3 and then slowly declines. Even at Mach 5.0, $C_D$ is still above 0.49—the flat base creates persistent base drag.

**G1 at Mach 1.0**

The canonical G1 drag coefficient at Mach 1.0 is $C_D = 0.4805$. This is the most commonly cited checkpoint value. In BALLISTICS-ENGINE, you can verify it directly using `get_drag_coefficient(1.0, &DragModel::G1)`, defined in `src/drag.rs`.

### 11.2.3 When G1 Works Well

G1 is a reasonable model for bullets whose shape is at least loosely similar to the G1 reference projectile:

- **Flat-base spitzers**: The Sierra GameKing 150 gr .30-cal, Speer Hot-Cor, and similar hunting bullets.
- **Short-ogive designs**: Bullets with 4–6 caliber tangent ogives and no boat tail.
- **Pistol and short-range rifle**: At distances under 300 yards (274 m), G1 is adequate for nearly any bullet because the accumulated error has not had time to grow.

### 11.2.4 When G1 Fails

G1's drag curve shape diverges significantly from modern long-range bullets. A VLD or hybrid boat-tail bullet has a very different $C_D$-vs-Mach profile: lower subsonic drag, a narrower transonic peak, and a steeper post-peak decline. When you force a G1 model onto such a bullet, the form

factor $i$ varies with Mach number, and the single-BC assumption breaks down. This is why G7 was developed.

Listing 11.4: Comparing G1 and G7 for a .308 Win load

```
# G1 model -- standard BC from box
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000

# G7 model -- G7 BC from manufacturer's data
ballistics trajectory \
  --bc 0.243 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000
```

## 11.3 The G7 Standard Projectile

### 11.3.1 A Modern Shape for Modern Bullets

The G7 standard was designed specifically for long, streamlined, boat-tail projectiles—the kind used in virtually every modern precision rifle cartridge. The reference shape is a tangent-ogive body with a 7.5-degree boat tail, and it much more closely approximates the geometry of bullets like the Sierra MatchKing, Berger Hybrid, and Hornady ELD Match.

### 11.3.2 The G7 Drag Curve

The full G7 drag table (84 data points) is stored alongside G1 in `src/drag_tables.rs`. Here are the key values:

Listing 11.5: G7 fallback drag data from `src/drag.rs` (condensed)

```
let fallback_data = [
    (0.0, 0.1198),   // Much lower subsonic drag than G1
    (0.5, 0.1197),   // Nearly flat subsonic profile
    (0.8, 0.1240),
    (0.9, 0.1294),   // Gradual rise
    (1.0, 0.3803),   // Transonic spike (lower than G1)
    (1.2, 0.4043),   // Peak near Mach 1.2
    (1.5, 0.3663),   // Faster decline
    (2.0, 0.2980),
    (3.0, 0.2424),
    (5.0, 0.1618),   // Much lower hypersonic drag
```

```
];
```

The differences from G1 are striking:

- **Much lower subsonic drag**: $C_D \approx 0.12$ versus G1's $\approx 0.26$. The boat tail dramatically reduces base drag.
- **Lower transonic peak**: G7 peaks at $C_D \approx 0.40$ versus G1's $\approx 0.66$—a 40% reduction.
- **Steeper supersonic decline**: G7 drops to $C_D \approx 0.16$ at Mach 5.0, while G1 is still at 0.50.
- **Flatter subsonic profile**: G7's $C_D$ is nearly constant below Mach 0.8, which is physically correct for streamlined projectiles.

**G7 at Mach 1.0**

The canonical G7 drag coefficient at Mach 1.0 is $C_D = 0.3803$. Compare this with G1's $C_D = 0.4805$ at the same Mach number. The function `get_drag_coefficient(1.0, &DragModel::G7)` in `src/drag.rs` returns this value.

### 11.3.3 Why G7 BCs Are Always Lower Than G1 BCs

A question that confuses many shooters: why is the G7 BC for a given bullet always lower than its G1 BC? The 168 gr Sierra MatchKing, for example, is listed at `0.462` G1 but only `0.243` G7. The answer is simply that BC is defined relative to the standard projectile:

$$\text{BC} = \frac{\text{sectional density}}{i} \tag{11.3}$$

where $i$ is the form factor (the ratio of the bullet's drag to the standard's drag). Because the G7 standard projectile already has low drag (similar to a modern match bullet), the form factor $i$ is close to 1.0, and the BC is close to the sectional density. The G1 standard has high drag, so $i < 1$ for a streamlined bullet, and the G1 BC ends up larger.

**Do Not Mix G1 and G7 BCs**

A `0.462` G1 BC and a `0.243` G7 BC produce nearly identical trajectories because they represent the same physical bullet. But entering a G1 BC with the G7 drag model (or vice versa) will produce wildly incorrect results. Always match the BC to its intended drag model.

### 11.3.4 When to Use G7

Use G7 for any bullet with:

- A boat-tail base (7–9 degree taper).
- A secant or hybrid ogive (VLD, ELD, etc.).
- A published G7 BC from the manufacturer or from doppler-derived data.
- Long-range applications (600+ yards / 550 m+) where transonic accuracy matters.

**G7 for Long-Range Precision**

For precision rifle competition and long-range hunting beyond 600 yards, G7 is almost always the better choice. The G7 standard matches the shape of modern match bullets so closely that a single G7 BC is valid from muzzle velocity down through the transonic transition—exactly the region where G1 breaks down.

## 11.4 Other Standard Models: G2, G5, G6, G8, GS, GI, RA4

While G1 and G7 dominate sporting ballistics, the full family of standard projectiles covers a wider range of shapes. BALLISTICS-ENGINE implements all of them through the `DragModel` enum. Each model was designed for a specific class of projectile.

**These Became Real Tables in 0.28.0**

Through v0.27.x, four of these families—G2, G5, GI and GS—had no dedicated table. Requesting one solved against a G1 approximation instead (silently until v0.27.1, with a warning after that). Version 0.28.0 added the genuine Aberdeen/BRL reference tables for all four, plus the new **RA4** model from the British RA 1929 function, bringing the native CLI's `--drag-model` to a full nine families.

**This changes numbers.** Any library, browser-terminal or FFI consumer using G2, G5, GI or GS now solves against real physics rather than a G1 stand-in. So do native profile and CSV `drag_model` strings of `G6` and `G8`, which the CLI had been warn-coercing to G1. The FFI adds `bc_type` slot 8 for RA4, appended so no existing code offset moves.

**Reading a Reference Curve Directly**

Since v0.30.0 you can print any of these tables verbatim with `drag-curve`:

```
ballistics drag-curve --drag-model g7
```

The points come straight out of the table with no resampling, so what you chart is exactly what the solver interpolates. Note the tables do not share a Mach domain: most run to Mach 5, while GS and RA4 stop at Mach 4. Library callers can reach the same data through `drag::reference_drag_table`.

### 11.4.1 G2: The Aberdeen J Projectile

The G2 standard represents a conical-nosed projectile—historically used to model military tracer and incendiary rounds. The shape features a long conical nose with a small flat base, giving it moderate subsonic drag and a transonic spike similar to G1 but narrower. G2 is rarely used in sporting ballistics, but it appears occasionally in military simulation codes.

### 11.4.2 G5: Short Boat-Tail

G5 models a bullet with a short (5.6-degree) boat tail and a 2-caliber secant ogive. It sits between G1 (flat base) and G7 (long boat tail) in terms of base drag. Some European military bullets from the mid-20th century were designed around the G5 standard.

### 11.4.3 G6: Flat-Base, 6-Caliber Secant Ogive

The G6 standard represents a flat-base bullet with a long (6-caliber) secant ogive nose. This shape is typical of military full-metal-jacket (FMJ) rifle bullets like the M80 ball (7.62×51mm NATO, 147 gr).

The G6 drag table in BALLISTICS-ENGINE (defined in `src/drag.rs`) has 67 data points with fine resolution through the transonic regime. At Mach 1.0, G6 has $C_D \approx 0.360$—lower than G1 (0.481) because the long secant ogive provides better wave drag characteristics.

Listing 11.6: Using G6 for a military ball round

```
ballistics trajectory \
  --bc 0.393 --drag-model g6 \
  --velocity 2800 --mass 147 --diameter 0.308 \
  --max-range 800
```

### 11.4.4 G8: Flat-Base, 10-Caliber Secant Ogive

G8 represents an even more streamlined flat-base design with a very long (10-caliber) secant ogive. This is an unusual shape in practice—a bullet with a 10-caliber ogive and a flat base would be impractically long for most cartridges—but it provides a useful reference for very low-drag flat-base designs.

The G8 table (defined in `src/drag.rs`, 61 data points) shows remarkably flat subsonic drag ($C_D \approx 0.210$ from Mach 0.0 all the way to Mach 0.9) followed by a sharp transonic rise to $C_D \approx 0.449$ at Mach 1.075.

### 11.4.5 GI: Ingalls Tables

The GI model refers to the Ingalls drag tables—a historical set of ballistic tables published by Colonel James Ingalls in 1893, based on the Mayevski/Siacci method. These tables were the standard reference for ballistic computation before the G-function system was developed. GI is primarily of historical interest but is included in BALLISTICS-ENGINE for completeness and for validating against legacy computations.

### 11.4.6 GS: Spherical (Round Ball)

The GS model (sometimes called GL for "glass" or "spherical") represents a perfect sphere. This is the correct model for round lead balls fired from muzzleloaders and smoothbore muskets. A sphere has no boat tail, no ogive, and no pressure gradient along its body—just pure form drag and, at supersonic speeds, a detached bow shock.

**Muzzleloader Ballistics**

If you are computing trajectories for a .50-caliber round ball at 1500 fps (457 m/s), use the GS drag model:

```
ballistics trajectory \
  --bc 0.066 --drag-model gs \
  --velocity 1500 --mass 177 --diameter 0.490 \
  --max-range 200
```

### 11.4.7 Choosing the Right Model

Table 11.1: Drag model selection guide

| Model | Best For | Example Projectiles |
|---|---|---|
| G1 | Flat-base spitzers, short ogive | Sierra GameKing, Speer Hot-Cor |
| G7 | Boat-tail match / VLD | Sierra MatchKing, Berger Hybrid |
| G6 | Military FMJ (flat base) | M80 Ball (147 gr 7.62 NATO) |
| G8 | Long-ogive flat base | Rare; academic reference |
| G2 | Conical-nose military | Tracer, incendiary rounds |
| G5 | Short boat-tail | Some European military |
| GI | Historical / Ingalls tables | Legacy computations |
| GS | Spherical / round ball | Muzzleloader round balls |
| RA4 | British RA 1929 function | Historical British ordnance |

## 11.5 Custom Drag Tables

### 11.5.1 Beyond the Standard Models

No standard projectile is a perfect match for any real bullet. The form factor varies with Mach number, which is precisely why a single BC is an approximation. For the highest accuracy, you can bypass the standard models entirely and supply a *custom drag table*—a directly measured $C_D(M)$ curve for your specific bullet.

Custom drag tables are obtained through:

1. **Doppler radar measurement**: The gold standard. A radar system tracks the bullet's velocity continuously from muzzle to impact, yielding hundreds or thousands of $C_D$ data points. This is how modern manufacturers characterise their bullets.

2. **Spark range photography**: Multiple exposures capture the bullet at known positions and times, allowing $C_D$ to be computed from the deceleration.

3. **CFD simulation**: Computational fluid dynamics can predict $C_D(M)$ from the bullet's 3D geometry, though validation against measured data is essential.

**Data Attribution**

When drag data are obtained via doppler radar measurement, we refer to them as "doppler-derived" drag data. This accurately describes the measurement methodology without attributing the data to any specific manufacturer.

### 11.5.2 Loading Custom Tables in ballistics-engine

The engine supports custom drag tables through two file formats:

1. **NumPy binary** (`.npy`): A two-column array where column 0 is Mach number and column 1 is $C_D$.

2. **CSV**: A comma-separated file with Mach number in the first column and $C_D$ in the second.

The function `load_drag_table()` in `src/drag.rs` handles both formats with automatic fallback:

Listing 11.7: Custom drag table loading from `src/drag.rs`

```rust
pub fn load_drag_table(
    drag_tables_dir: &Path,
    filename: &str,
    fallback_data: &[(f64, f64)],
) -> DragTable {
    // Try NumPy binary first
    let npy_path = drag_tables_dir
        .join(format!("{filename}.npy"));
    if let Ok(array) = ndarray_npy::read_npy(&npy_path) {
        // ... load from NumPy
    }
    // Fallback to CSV
    let csv_path = drag_tables_dir
        .join(format!("{filename}.csv"));
    if let Ok(mut reader) = csv::Reader::from_path(&csv_path) {
        // ... load from CSV
    }
    // Use compiled-in fallback data
    DragTable::new(mach_values, cd_values)
}
```

This three-tier loading strategy—binary first, CSV second, compiled-in fallback third—ensures that the engine always has usable drag data, even in constrained environments (WASM, embedded) where file I/O may not be available.

**A Custom Table Divides by Sectional Density, Not BC (0.22.13)**

A standard drag model gives you the *reference* projectile's $C_D$, and the BC is what scales it to your bullet. A custom table gives you your bullet's *own* $C_D$, so the BC has no role left to play—the physically correct denominator is the sectional density in lb/in²:

$$\frac{C_D^{\text{own}}}{\text{SD}} = \frac{C_D^{\text{ref}}}{\text{BC}}, \qquad \text{SD} = \frac{\text{weight in grains}/7000}{d_{\text{in}}^2} \tag{11.4}$$

All three solver paths—`src/derivatives.rs`, `src/fast_trajectory.rs`, and `src/cli_api.rs`—divided a custom table's $C_D$ by `bc_value` instead, so a custom-table trajectory scaled with whatever BC happened to be lying around in the inputs. Version 0.22.13 derives the denominator from the bullet's own mass and diameter (`BallisticInputs::sectional_density_lb_in2` and `custom_drag_denominator`); when neither is available it falls back to `bc_value` with a one-time warning rather than panicking. **A custom-table run is now invariant to `--bc`, and its numbers differ from any earlier release.**

### 11.5.3 Vendor `.drg` Files

Since v0.28.0 the `--drag-table` flag on **`trajectory`**, `monte-carlo` and **`zero`** also accepts QuickTARGET-style `.drg` decks, selected by extension:

Listing 11.8: Loading a vendor drag deck

```
ballistics trajectory \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --bc 0.462 --auto-zero 200 --sight-height 1.5 \
  --drag-table my-bullet.drg
```

Column order is the usual trap—some decks are written $(C_D, \text{Mach})$ and some $(\text{Mach}, C_D)$—so the loader auto-detects it. When the data is genuinely ambiguous it refuses rather than guessing, with a hard "ambiguous columns" error and a hint to convert the file to CSV, where you control the order yourself.

### 11.5.4 Scaling a Whole Curve: `--cd-scale`

A measured $C_D$ curve is the right *shape* for your bullet but not necessarily the right *magnitude*—barrel-to-barrel differences, a different lot, a different twist. Rather than editing the deck, `--cd-scale` multiplies the whole curve by a single factor, in the manner of Hornady's Axial Form Factor or Applied Ballistics' CDF scaling:

Listing 11.9: Truing a custom deck with a whole-curve multiplier

```
ballistics trajectory \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --bc 0.462 --auto-zero 200 --sight-height 1.5 \
  --drag-table my-bullet.csv --cd-scale 0.97
```

1.0 is neutral and the useful truing range is roughly 0.90 to 1.10; values outside $[0.5, 2.0]$ draw a plausibility warning. The flag is meaningless without a deck to scale, so passing it without `--drag-table` is a hard pairing error rather than a silent no-op.

**A Library Breaking Change (0.28.0)**

Adding `--cd-scale` meant threading the factor through the solver, and three public signatures changed with it. `wez::compute_wez` gained a positional `cd_scale` parameter; `TrajectoryParams` gained a required `cd_scale` field, so external constructors must set it; and `BallisticInputs` gained a public `cd_scale` field, which breaks exhaustive struct literals. Pass 1.0 for the previous behavior, and soften the struct literals with `..Default::default()`. The FFI took the additive route instead: new `_scaled` export variants, with every existing entry point untouched.

### 11.5.5 The DragTable Data Structure

Internally, all drag data—whether from a standard model or a custom table—are stored in the same `DragTable` struct:

Listing 11.10: The DragTable struct from `src/drag.rs`

```
pub struct DragTable {
    pub mach_values: Vec<f64>,
    pub cd_values: Vec<f64>,
}
```

This simple design makes the interpolation code model-agnostic: the `interpolate()` method works identically whether the table contains 21 G1 fallback points or 500 doppler-derived measurements.

**SAFETY: Custom Drag Data Quality**

If you supply a custom drag table with erroneous data—for example, unrealistically low $C_D$ values in the transonic region—the solver will happily compute a trajectory that predicts less drop than physically possible. This could lead to shots that impact above the intended point of aim, with potentially dangerous consequences in hunting or tactical scenarios. Always validate custom drag data against known references before using them in field applications.

## 11.6 Drag Table Interpolation

A drag table is a discrete set of (Mach, $C_D$) pairs, but the trajectory solver needs $C_D$ at arbitrary Mach numbers. The gap is bridged by *interpolation*. BALLISTICS-ENGINE uses a hybrid scheme: Catmull–Rom cubic interpolation in the interior of the table, falling back to linear interpolation at the edges.

### 11.6.1 Catmull–Rom Cubic Interpolation

For any query Mach number that falls between interior table points (i.e., there is at least one point on either side), the engine uses a Catmull–Rom spline. Given four consecutive data points $(x_0, y_0)$ through $(x_3, y_3)$, the interpolated value at parameter $t = (M - x_1)/(x_2 - x_1)$ is:

$$C_D(t) = a_0 t^3 + a_1 t^2 + a_2 t + a_3 \tag{11.5}$$

with coefficients:

$$a_0 = -\tfrac{1}{2}y_0 + \tfrac{3}{2}y_1 - \tfrac{3}{2}y_2 + \tfrac{1}{2}y_3 \tag{11.6}$$

$$a_1 = y_0 - \tfrac{5}{2}y_1 + 2y_2 - \tfrac{1}{2}y_3 \tag{11.7}$$

$$a_2 = -\tfrac{1}{2}y_0 + \tfrac{1}{2}y_2 \tag{11.8}$$

$$a_3 = y_1 \tag{11.9}$$

This is implemented in the `cubic_interpolate()` method of `DragTable`:

Listing 11.11: Catmull–Rom interpolation from `src/drag.rs`

```
pub fn cubic_interpolate(&self, mach: f64, idx: usize) -> f64 {
    let x = [self.mach_values[idx - 1],
             self.mach_values[idx],
             self.mach_values[idx + 1],
             self.mach_values[idx + 2]];
    let y = [self.cd_values[idx - 1],
             self.cd_values[idx],
             self.cd_values[idx + 1],
             self.cd_values[idx + 2]];

    let t = (mach - x[1]) / (x[2] - x[1]);
    let t2 = t * t;
    let t3 = t2 * t;

    let a0 = -0.5*y[0] + 1.5*y[1] - 1.5*y[2] + 0.5*y[3];
    let a1 =  y[0] - 2.5*y[1] + 2.0*y[2] - 0.5*y[3];
```

```
    let a2 = -0.5*y[0] + 0.5*y[2];
    let a3 =  y[1];

    a0*t3 + a1*t2 + a2*t + a3
}
```

### 11.6.2 Why Catmull–Rom?

The Catmull–Rom spline passes exactly through each data point (unlike a smoothing spline, which may not), and it provides $C^1$ continuity—the interpolated curve and its first derivative are continuous at every data point. This is critical for the trajectory integrator, which computes $dV/dt$ from $C_D$; a discontinuity in $C_D$ would cause a discontinuity in deceleration, leading to integration artefacts.

### 11.6.3 Edge Handling and Extrapolation

At the edges of the table (the first and last segments), there are not enough surrounding points for cubic interpolation. Here the engine falls back to *linear interpolation*:

Listing 11.12: Linear interpolation fallback

```
pub fn linear_interpolate(&self, mach: f64, idx: usize) -> f64 {
    let t = (mach - self.mach_values[idx])
          / (self.mach_values[idx + 1] - self.mach_values[idx]);
    self.cd_values[idx] + t * (self.cd_values[idx + 1]
                             - self.cd_values[idx])
}
```

For Mach numbers outside the table range entirely, the engine uses linear *extrapolation* from the nearest two points, clamped to a minimum of $C_D = 0.01$ to prevent physically impossible negative drag coefficients.

**Table Resolution Matters**

The accuracy of interpolation depends on the density of data points. The full G1 and G7 tables in `src/drag_tables.rs` have 79 and 84 points respectively, with fine resolution (0.025 Mach steps) through the critical transonic region (Mach 0.8–1.3). The condensed fallback tables in `src/drag.rs` have only 21 points and are correspondingly less accurate in the transonic zone. When possible, the engine uses the full tables.

## 11.7 Form Factors: Connecting BC to Drag Coefficients

### 11.7.1 The Form Factor Defined

The *form factor* $i$ is the bridge between a bullet's actual drag and the drag of the standard projectile:

$$i = \frac{C_D^{\text{bullet}}(M)}{C_D^{\text{std}}(M)} \tag{11.10}$$

If the form factor were truly constant across all Mach numbers, a single BC would describe the bullet's drag perfectly. In practice, $i$ varies, especially through the transonic regime—which is precisely why G7 works better than G1 for boat-tail bullets (the G7 form factor stays closer to 1.0 across the Mach range).

### 11.7.2 Form Factors in ballistics-engine

The engine provides an optional form factor correction system implemented in `src/form_factor.rs`. Given a bullet name (or type keyword) and a drag model, the function `get_default_form_factor()` returns a scalar form factor that can be applied to the drag coefficient:

Listing 11.13: Form factor lookup from `src/form_factor.rs`

```
pub fn get_default_form_factor(
    bullet_name: &str, drag_model: &DragModel
) -> f64 {
    match drag_model {
        DragModel::G1 => {
            if name.contains("MATCH") || name.contains("SMK") {
                0.90  // Match bullets
            } else if name.contains("VLD") || name.contains("ELD") {
                0.85  // Very low drag bullets
            } else if name.contains("FMJ") {
                1.10  // Full metal jacket
            } else {
                1.00  // Default
            }
        }
        DragModel::G7 => {
            if name.contains("MATCH") || name.contains("SMK") {
                0.95
            } else if name.contains("VLD") || name.contains("HYBRID") {
                0.90
            } else if name.contains("FMJ") {
                1.15
```

```
            } else {
                1.05
            }
        }
        _ => 1.00,
    }
}
```

The form factor is then applied to the drag coefficient via `apply_form_factor_to_drag()`:

$$C_D^{\text{corrected}} = C_D^{\text{std}} \times i \tag{11.11}$$

### 11.7.3 Understanding the Values

Table 11.2: Default form factors in BALLISTICS-ENGINE

| Bullet Type | G1 $i$ | G7 $i$ | Interpretation |
|---|---|---|---|
| Match (SMK, Scenar) | 0.90 | 0.95 | Less drag than standard |
| VLD / ELD / Hybrid | 0.85 | 0.90 | Significantly lower drag |
| Hunting (SP, SST) | 1.05 | — | Slightly more drag |
| FMJ / Military | 1.10 | 1.15 | More drag than standard |
| Default | 1.00 | 1.05 | Baseline |

- $i < 1.0$: The bullet has *less* drag than the standard projectile. A VLD bullet with $i = 0.85$ on G1 produces 15% less drag than the G1 reference at every Mach number.
- $i = 1.0$: The bullet matches the standard projectile's drag exactly.
- $i > 1.0$: The bullet has *more* drag. An FMJ with $i = 1.10$ on G1 has 10% more drag than the G1 reference.

**Form Factors Are Approximate**

The built-in form factors in BALLISTICS-ENGINE are broad heuristics based on bullet category keywords. They cannot capture the nuance of a specific bullet's geometry. For precision work, use a published G7 BC (which inherently has a form factor near 1.0) or supply measured drag data via a custom drag table.

## 11.8 How ballistics-engine Evaluates Drag

Now that we have covered the individual pieces—drag models, tables, interpolation, and form factors—let us trace the complete path from a bullet's instantaneous velocity to the drag force applied by the trajectory integrator.

### 11.8.1 The Drag Evaluation Pipeline

At each integration time step, the solver needs the drag coefficient at the bullet's current Mach number. The engine provides three levels of sophistication, selected by the function called:

1. **Basic**: `get_drag_coefficient(mach, &drag_model)` — looks up $C_D$ from the appropriate drag table using cubic/linear interpolation.
2. **Transonic-corrected**: `get_drag_coefficient_with_transonic()` — applies the transonic drag rise and wave drag corrections from `src/transonic_drag.rs` when the projectile is between Mach 0.8 and 1.3 (see Chapter 12 for details).
3. **Full**: `get_drag_coefficient_full()` — adds Reynolds number corrections for low-velocity subsonic flight, accounting for changes in boundary-layer behaviour at low speeds.

These functions are defined in `src/drag.rs` and form a layered pipeline:

Listing 11.14: The full drag coefficient pipeline from `src/drag.rs`

```rust
pub fn get_drag_coefficient_full(
    mach: f64,
    drag_model: &DragModel,
    apply_transonic_correction: bool,
    apply_reynolds_correction: bool,
    projectile_shape: Option<ProjectileShape>,
    caliber: Option<f64>,
    weight_grains: Option<f64>,
    velocity_mps: Option<f64>,
    air_density_kg_m3: Option<f64>,
    temperature_c: Option<f64>,
) -> f64 {
    // Layer 1: Base CD from drag table
    let mut cd = get_drag_coefficient_with_transonic(
        mach, drag_model, apply_transonic_correction,
        projectile_shape, caliber, weight_grains,
    );
    // Layer 2: Reynolds correction for subsonic flight
    if apply_reynolds_correction && mach < 1.0 {
        cd = apply_reynolds_correction(
            cd, velocity, caliber, rho, temp, mach
```

```
        );
    }
    cd
}
```

### 11.8.2 The Dispatch by Model

The core dispatch in `get_drag_coefficient()` is a simple match on the `DragModel` enum:

Listing 11.15: Model dispatch from `src/drag.rs`

```rust
pub fn get_drag_coefficient(
    mach: f64, drag_model: &DragModel
) -> f64 {
    match drag_model {
        DragModel::G1 => G1_DRAG_TABLE.interpolate(mach),
        DragModel::G6 => G6_DRAG_TABLE.interpolate(mach),
        DragModel::G7 => G7_DRAG_TABLE.interpolate(mach),
        DragModel::G8 => G8_DRAG_TABLE.interpolate(mach),
        _ => G1_DRAG_TABLE.interpolate(mach),
    }
}
```

Each `*_DRAG_TABLE` is a `LazyLock<DragTable>`—a static variable that is initialised on first access. The `LazyLock` pattern ensures that the drag table is loaded exactly once (thread-safely) and then reused for every subsequent query, with zero per-call allocation overhead.

**A Silent Fallback Became a Warning (0.27.1)**

Requesting a drag family the engine had no dedicated table for used to substitute G1 without saying anything, so a G5 or GI run looked like it had worked. Since v0.27.1 the substitution is announced: profile and CSV `drag_model` strings other than G1/G7 warn on stderr on the native CLI, and the browser terminal prints a note for G2, G5, GI and GS. Version 0.28.0 then went further and gave those families real tables—see Section 11.4.

### 11.8.3 Lazy Loading and Fallback

Each static drag table attempts three loading strategies in order:

1. **NumPy binary file**: The function `find_drag_tables_dir()` searches for a `drag_tables/` directory relative to the executable. If found, `load_drag_table()` reads `g1.npy`, `g7.npy`, etc.
2. **CSV file**: If the binary file is missing, the engine tries `g1.csv`, `g7.csv`, etc.

3. **Compiled-in fallback**: If neither file is found (common in WASM builds or when the engine is embedded), the table is constructed from the hard-coded $(M, C_D)$ pairs in the source.

**Inspecting Drag Model Output**

Use the `--full` flag to see detailed trajectory data including drag coefficients at each step:

```
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 500 --full
```

The full output includes Mach number and drag coefficient columns at each range step.

### 11.8.4 Performance Considerations

Drag coefficient lookup is the single most frequently called operation in the trajectory solver. A 1 000-yard trajectory at the default 0.001 s time step evaluates $C_D$ thousands of times. The design choices in `src/drag.rs` reflect this:

- **No allocation**: The `DragTable` struct uses pre-allocated `Vec`s; interpolation involves only arithmetic on stack variables.
- **Linear scan**: The segment search is a simple linear scan rather than a binary search. For typical table sizes (21–84 entries), the linear scan is faster due to better cache behaviour.
- **LazyLock**: Tables are loaded once and shared immutably across all invocations, with no locking on the read path.
- **Minimum clamping**: The extrapolation code clamps $C_D$ to a minimum of 0.01 rather than performing an expensive bounds check, preventing negative values without branching.

A benchmark of 2 000 drag coefficient evaluations (1 000 each for G1 and G7, spanning Mach 0.5 to 4.5) completes in well under 100 ms on modern hardware—fast enough that drag lookup never becomes a bottleneck, even in Monte Carlo simulations with millions of trajectory evaluations.

## 11.9 BC Interpolation by Mach Number

Before we leave this chapter, it is worth noting that BALLISTICS-ENGINE also supports *velocity-dependent BC* through the `interpolated_bc()` function in `src/drag.rs`. This function takes a Mach number and a set of (Mach, BC) segment pairs, and returns the linearly interpolated BC at the query point:

Listing 11.16: BC interpolation from `src/drag.rs`

```rust
pub fn interpolated_bc(mach: f64, segments: &[(f64, f64)]) -> f64 {
    // Binary search for the right segment
    // Linear interpolation between the two closest points
    let frac = (mach - lo_mach) / (hi_mach - lo_mach);
    lo_bc + frac * (hi_bc - lo_bc)
}
```

This is the foundation for the BC segments and cluster BC models covered in Chapter 12. Instead of a single constant BC, the solver can query a BC that varies smoothly with Mach number, dramatically improving accuracy at extended range where the bullet transitions through multiple flight regimes.

## 11.10 Exercises

These exercises use real BALLISTICS-ENGINE commands. Run them in your terminal and study the output.

1. **Compare G1 and G7 trajectories.** Run the following two commands for a .308 Win, 175 gr Sierra MatchKing (`0.505` G1, `0.264` G7) at 2 600 fps:

   ```
   ballistics trajectory \
     --bc 0.505 --drag-model g1 \
     --velocity 2600 --mass 175 --diameter 0.308 \
     --max-range 1000
   ```

   ```
   ballistics trajectory \
     --bc 0.264 --drag-model g7 \
     --velocity 2600 --mass 175 --diameter 0.308 \
     --max-range 1000
   ```

   Compare the drop at 500 and 1 000 yards. How closely do they agree? At what range do they begin to diverge?

2. **Explore the transonic region.** Run a trajectory for a .223 Rem, 77 gr Sierra MatchKing (`0.372` G1) at 2 750 fps out to 1 000 yards. At what range does the bullet go transonic (below Mach 1.2)? How does the velocity curve change slope at that point?

   ```
   ballistics trajectory \
     --bc 0.372 --drag-model g1 \
     --velocity 2750 --mass 77 --diameter 0.224 \
     --max-range 1000
   ```

3. **Flat base vs. boat tail.** Using the G6 model (flat-base military), compute a trajectory for the M80 ball round (147 gr, 0.393 G6, 2 800 fps) and compare it with the same weight in a match configuration using G7 (0.210 G7, 2 800 fps). How much more drop does the flat-base FMJ exhibit at 800 yards?

```
# M80 Ball with G6
ballistics trajectory \
  --bc 0.393 --drag-model g6 \
  --velocity 2800 --mass 147 --diameter 0.308 \
  --max-range 800

# Match equivalent with G7
ballistics trajectory \
  --bc 0.210 --drag-model g7 \
  --velocity 2800 --mass 147 --diameter 0.308 \
  --max-range 800
```

4. **Round ball ballistics.** Compute the trajectory of a .50-caliber patched round ball (177 gr, 0.066 GS) at 1 500 fps. How much drop is there at 100 yards? At what range does the ball go subsonic?

```
ballistics trajectory \
  --bc 0.066 --drag-model gs \
  --velocity 1500 --mass 177 --diameter 0.490 \
  --max-range 200
```

## What's Next

We have seen that a ballistic coefficient is an *approximation*—a single number trying to capture a bullet's drag across all Mach numbers. For short-range work, this approximation is excellent. But as range increases and the bullet traverses the transonic regime, the cracks in the single-BC model begin to show.

In Chapter 12, we explore how BALLISTICS-ENGINE addresses these limitations: velocity-dependent BC via cluster degradation, step-function BC segments, physics-bounded BC constraints, and dedicated transonic drag corrections that model shock wave behaviour directly. Together, these tools extend the engine's accuracy deep into the transonic zone where simpler models stumble.

# Chapter 12

# Advanced BC Modeling

In Chapter 11 we established the ballistic coefficient as a single number that scales a standard drag curve. That model works beautifully—until it does not. At extended range, as a bullet decelerates from supersonic through transonic to subsonic flight, its effective BC changes. A `0.462` G1 BC might describe a 168 gr Sierra MatchKing perfectly at 2 700 fps but overpredict its performance at 1 200 fps by 5–10%. Over the course of a 1 000-yard flight, those percentage points compound into inches of error.

This chapter digs into the physics behind velocity-dependent BC, the three approaches BALLISTICS-ENGINE offers for modelling it—cluster degradation, BC segments, and physics-bounded constraints—and the critical transonic regime where all drag models struggle the most.

## 12.1 Why BC Is Not Constant

### 12.1.1 The Form Factor Problem Revisited

Recall from Section 11.7 that the ballistic coefficient is defined as:

$$\mathrm{BC} = \frac{\mathrm{SD}}{i(M)} \tag{12.1}$$

where SD is the sectional density (a fixed property of the bullet's mass and calibre) and $i(M)$ is the form factor at Mach number $M$. If $i$ were constant, BC would be constant. But $i(M)$ is the ratio of the real bullet's drag curve to the standard projectile's drag curve:

$$i(M) = \frac{C_D^{\text{bullet}}(M)}{C_D^{\text{std}}(M)} \tag{12.2}$$

The real bullet and the standard projectile respond differently to the aerodynamic phenomena that change with Mach number: boundary layer transition, shock wave formation, base drag, and wake structure. At Mach 2.0, a boat-tail bullet might have $i = 0.92$ relative to G1. At Mach 0.9, where the boat tail's advantage in base drag reduction diminishes relative to the transonic wave drag, $i$ might climb to 1.05. The BC that was `0.462` at the muzzle is effectively `0.405` near the sound barrier.

### 12.1.2 Where the Error Shows Up

The practical consequence of a non-constant BC is that a single-BC trajectory prediction diverges from reality as the bullet slows. The divergence is small at short range (the bullet has not decelerated much) and grows with distance. For a .308 Win, 168 gr SMK at 2 700 fps:

- **At 300 yards**: The bullet is still at ∼2 300 fps (Mach 2.1). A constant BC works fine.
- **At 600 yards**: The bullet is at ∼1 800 fps (Mach 1.6). The form factor has shifted slightly. Error: 1–2 inches.
- **At 1 000 yards**: The bullet is near Mach 1.1, deep in the transonic zone. The form factor may have changed by 10–15%. Error: 6–12 inches.

**G7 Mitigates But Does Not Eliminate**

Using a G7 BC instead of G1 reduces the form factor variation because G7's shape more closely matches modern boat-tail bullets. The G7 form factor might vary from 0.97 to 1.03 instead of G1's 0.85 to 1.05. This is why G7 is preferred for long-range work—but even G7 has residual velocity dependence, especially in the transonic regime.

## 12.2 Velocity-Dependent BC: The Cluster Degradation Model

### 12.2.1 The Concept

The cluster BC degradation model is BALLISTICS-ENGINE's most sophisticated approach to velocity-dependent BC. Rather than treating all bullets the same, it classifies bullets into **four clusters** based on their physical characteristics—calibre, weight, and baseline BC—and applies a *cluster-specific* velocity degradation curve to each.

The insight is that bullets within the same physical class tend to share similar aerodynamic behaviour as they decelerate. A heavy magnum bullet (.338 Lapua, 300 gr) retains its BC much better through

the transonic regime than a light varmint bullet (.223 Rem, 55 gr), even if both have the same nominal BC.

### 12.2.2 The Four Clusters

The cluster model, implemented in `src/cluster_bc.rs`, defines four clusters with pre-computed centroids:

Listing 12.1: Cluster centroids from `src/cluster_bc.rs`

```
centroids: [
    (0.605, 0.415, 0.613), // Cluster 0: Standard Long-Range
    (0.516, 0.324, 0.643), // Cluster 1: Low-Drag Specialty
    (0.307, 0.088, 0.336), // Cluster 2: Light Varmint
    (0.750, 0.805, 0.505), // Cluster 3: Heavy Magnums
],
```

Each centroid is a triple of normalised values: (calibre, weight, BC). A new bullet is classified by computing the Euclidean distance from its normalised parameters to each centroid, and assigning it to the nearest cluster.

**The Four Bullet Clusters**

**Cluster 0 — Standard Long-Range**
: The workhorse class: .30-calibre match bullets in the 155–180 gr range, 6.5mm competition bullets, and similar. Moderate BC retention with a notable transonic dip at 1 200–1 500 fps and partial recovery below 1 200 fps.

**Cluster 1 — Low-Drag Specialty**
: VLD, ELD, Hybrid, and A-TIP designs. Superior transonic performance with a milder dip. These bullets were designed for the transonic regime and their aerodynamics reflect it.

**Cluster 2 — Light Varmint/Target**
: Small-calibre, light bullets: .224" 55 gr, .243" 70 gr, etc. Higher BC degradation across the velocity range, with a sharper transonic dip.

**Cluster 3 — Heavy Magnums**
: Large, heavy bullets: .338" 300 gr, .375" 350 gr, .50 BMG. Best BC retention of any class, with minimal degradation even through the transonic zone. High sectional density provides inherent stability.

### 12.2.3 Cluster Assignment

The `predict_cluster()` method normalises the input parameters to a $[0, 1]$ range using bounds derived from training data:

Listing 12.2: Cluster prediction from `src/cluster_bc.rs`

```rust
pub fn predict_cluster(
    &self, caliber: f64, weight_gr: f64, bc_g1: f64
) -> usize {
    // Normalize to [0, 1]
    let caliber_norm = (caliber - 0.172) / (0.750 - 0.172);
    let weight_norm  = (weight_gr - 15.0) / (750.0 - 15.0);
    let bc_norm      = (bc_g1 - 0.05) / (1.2 - 0.05);

    // Find nearest centroid (Euclidean distance)
    let mut min_distance = f64::INFINITY;
    let mut best_cluster = 0;
    for (i, &(c_cal, c_wt, c_bc)) in
        self.centroids.iter().enumerate()
    {
        let distance = ((caliber_norm - c_cal).powi(2)
            + (weight_norm - c_wt).powi(2)
            + (bc_norm - c_bc).powi(2)).sqrt();
        if distance < min_distance {
            min_distance = distance;
            best_cluster = i;
        }
    }
    best_cluster
}
```

The normalisation bounds span from the .17 HMR (calibre 0.172", 15 gr) to the .50 BMG (calibre 0.750", 750 gr), with BC values from 0.05 to 1.2 G1. This covers essentially every projectile in sporting and military use.

Listing 12.3: Enabling cluster BC degradation

```
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --use-cluster-bc \
  --max-range 1000
```

### 12.2.4 Velocity-Dependent Multipliers

Once a bullet is assigned to a cluster, the `get_bc_multiplier()` method returns a BC multiplier for the bullet's current velocity. The multiplier is 1.0 at high velocity (above 3 500 fps) and decreases as the bullet slows, with a characteristic shape for each cluster.

For Cluster 0 (Standard Long-Range), the degradation curve is:

Table 12.1: Cluster 0 (Standard Long-Range) BC multipliers

| Velocity (fps) | BC Multiplier | Notes |
|---|---|---|
| $> 3\,500$ | 1.000 | Full BC |
| 3 000–3 500 | 0.99–1.00 | Minimal degradation |
| 2 500–3 000 | 0.98–0.99 | Slight degradation |
| 2 000–2 500 | 0.98 | Flat mid-range plateau |
| 1 500–2 000 | 0.96–0.98 | Increasing degradation |
| 1 200–1 500 | 0.86–0.96 | **Transonic dip** |
| 1 000–1 200 | 0.86–0.97 | **Recovery** |
| $< 1\,000$ | 0.93–0.97 | Subsonic plateau |

**The Transonic Dip and Recovery**

A counterintuitive feature of the cluster model is the *recovery* below the transonic dip. At 1 200–1 500 fps, the bullet is in the worst part of the transonic regime: shock waves are forming, detaching, and reattaching unpredictably, causing maximum drag coefficient uncertainty. Below 1 200 fps, the bullet is cleanly subsonic—no more shock waves—and the aerodynamics become smoother again. The BC multiplier partially recovers to reflect this.

This pattern was calibrated from measured BC segment data across 170 real bullets (MBA-645 in the BALLISTICS-ENGINE issue tracker). Earlier versions of the model predicted an unrealistic 65% BC retention at subsonic velocities; measured data showed approximately 93% for typical long-range bullets.

### 12.2.5 Cluster Comparison

The four clusters differ primarily in the depth of the transonic dip and the overall retention at low velocity:

Table 12.2: BC retention at key velocities by cluster

| Cluster | At 1 300 fps | At 900 fps | Transonic Dip |
|---|---|---|---|
| 0: Standard Long-Range | 90% | 96% | Moderate |
| 1: Low-Drag Specialty | 93% | 96% | Mild |
| 2: Light Varmint/Target | 85% | 88% | Sharp |
| 3: Heavy Magnum | 95% | 97% | Minimal |

Cluster 3 (Heavy Magnum) bullets retain 95%+ of their BC even at the bottom of the transonic dip. Their high sectional density and large mass act as aerodynamic flywheels, resisting deceleration and maintaining stable flight geometry. Cluster 2 (Light Varmint) bullets, by contrast, can lose 18% of their BC in the transonic zone—a light, small-calibre bullet is simply more susceptible to the aerodynamic chaos of shock wave formation.

## 12.3 BC Segments: Step-Function BC by Velocity Band

### 12.3.1 A Simpler Approach

Not every application needs the cluster model's sophistication. The **BC segments** approach uses a step function: the velocity range is divided into bands, and each band gets its own constant BC. This is the model used by most commercial ballistic calculators that support "stepped BC."

In BALLISTICS-ENGINE, BC segments are represented by the `BCSegmentData` struct:

Listing 12.4: The BCSegmentData struct from `src/lib.rs`

```
pub struct BCSegmentData {
    pub velocity_min: f64,
    pub velocity_max: f64,
    pub bc_value: f64,
}
```

### 12.3.2 Automatic Segment Estimation

Rather than requiring the user to manually specify BC at each velocity band, BALLISTICS-ENGINE can automatically estimate segments from a single base BC. The `BCSegmentEstimator` in `src/bc_estimation.rs` classifies the bullet into one of nine types and applies type-specific degradation profiles.

#### Bullet Type Classification

The estimator first classifies the bullet by parsing the model name for keywords:

Listing 12.5: Bullet type classification from `src/bc_estimation.rs`

```
pub enum BulletType {
    MatchBoatTail,     // SMK, Scenar, BTHP, etc.
    MatchFlatBase,     // Match bullets without boat tail
    HuntingBoatTail,   // SST, AccuBond, Partition BT
    HuntingFlatBase,   // Core-Lokt, Interlock, SP
    VldHighBc,         // VLD, Berger, Elite
    Hybrid,            // Hybrid OTM designs
```

```
    FMJ,              // Full metal jacket, ball ammo
    RoundNose,        // Round nose / RNSP
    Unknown,          // Conservative default
}
```

### Type-Specific Degradation

Each bullet type has a characteristic total BC drop and transition curve shape:

Table 12.3: BC degradation factors by bullet type

| Bullet Type | Total BC Drop | Character |
|---|---|---|
| VLD / High BC | 5% | Very stable; minimal change |
| Hybrid | 6% | Nearly as stable as VLD |
| Match Boat-Tail | 7.5% | Small, gradual drop |
| Match Flat-Base | 10% | Moderate drop |
| FMJ / Military | 12% | Moderate, base-drag dominated |
| Hunting Boat-Tail | 15% | Noticeable at long range |
| Hunting Flat-Base | 20% | Significant at extended range |
| Round Nose | 35% | Large; poor transonic behaviour |
| Unknown | 15% | Conservative default |

### Segment Generation Example

For a Match Boat-Tail bullet, the estimator generates five segments:

Listing 12.6: Match Boat-Tail segment profile from `src/bc_estimation.rs`

```
BulletType::MatchBoatTail => {
    // BC x 1.000 for 2800-5000 fps
    // BC x 0.985 for 2400-2800 fps
    // BC x 0.965 for 2000-2400 fps
    // BC x 0.945 for 1600-2000 fps
    // BC x 0.925 for    0-1600 fps
}
```

A 168 gr SMK with `0.462` G1 base BC would therefore use:

- `0.462` above 2 800 fps
- `0.455` at 2 400–2 800 fps

- `0.446` at 2 000–2 400 fps
- `0.437` at 1 600–2 000 fps
- `0.427` below 1 600 fps

Listing 12.7: Enabling BC segments

```
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --use-bc-segments \
  --max-range 1000
```

### 12.3.3 Writing the Ladder Yourself

The estimator is a model of how BC decays, not a measurement of how *your* bullet's BC decays. If you have done the downrange work—or if a manufacturer publishes a banded BC for the bullet—you can supply the ladder directly. Since v0.22.11 the repeatable `--bc-segment` flag takes `VMIN:VMAX:BC` triples, and the given BC applies while the bullet's current speed is in $[V_{\min}, V_{\max})$:

Listing 12.8: A hand-written velocity-keyed BC ladder

```
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 200 --sight-height 1.5 --max-range 800 \
  --bc-segment 2400:3000:0.462 \
  --bc-segment 2000:2400:0.449 \
  --bc-segment 0:2000:0.432
```

Four rules govern the flag:

- $V_{\min}$ and $V_{\max}$ follow `--units`—fps in imperial, m/s in metric.
- Segments are keyed to *velocity*, which makes them orthogonal to the distance-keyed `--wind-segment` of Chapter 10. Combine both and each applies on its own axis.
- A speed that falls outside every segment uses the base `--bc`.
- Passing any `--bc-segment` implies `--use-bc-segments` and *overrides* `--bc-table` and `--bc-table-dir`. A manual ladder is a clean override: the out-of-segment fallback BC is not scaled by a BC5D table's muzzle correction either.

**Transcribing a BC5D Ladder**

If you have a BC5D correction table on disk but you need to run the same load somewhere that cannot hold one—the browser build, a handheld—add `--print-bc-segments` to a `--bc-table-dir` run. It prints the table's generated ladder as ready-to-paste `--bc-segment` arguments in your active `--units`, so the other device can reproduce the correction from a transcribed ladder alone.

**Auto-Zero Now Sees the Segments (0.22.11)**

The native zero-angle solver used to ignore velocity-keyed BC entirely and solve with the base `--bc`, which meant a segment that changed early-flight drag mis-zeroed the shot—it grounded short of the requested zero distance and reported drop and range against a zero the rifle did not have. The WASM build had always done this correctly; the native CLI now matches.

**Band Edges Are Blended, Not Stepped (0.28.0)**

A pure step function has a discontinuity at every band edge, and a discontinuous BC produces a discontinuous drag—visible as a small kink in the trajectory exactly where the bullet crosses a boundary. Since v0.28.0 transitions are smoothstep-blended across a margin of $\min(50 \text{ fps}, 25\% \text{ of the narrower adjacent band})$, so a banded-BC trajectory is continuous across every edge. **This is a numeric change** for any run using velocity bands; mid-band values are unchanged, and only the neighbourhoods of the boundaries move.

### 12.3.4 Sectional Density Adjustment

The segment estimator further adjusts the degradation based on the bullet's *sectional density* (SD):

$$\text{SD} = \frac{m}{7000 \times d^2} \tag{12.3}$$

where $m$ is mass in grains and $d$ is calibre in inches. The SD factor scales the degradation: higher SD ($> 0.25$) means more stable BC, lower SD ($< 0.25$) means more degradation. The factor is computed as:

$$f_{\text{SD}} = \left( \frac{\text{SD}}{0.25} \right)^{0.5}, \quad 0.7 \leq f_{\text{SD}} \leq 1.3 \tag{12.4}$$

For a .308 Win 168 gr bullet, SD $\approx 0.253$, giving $f_{\text{SD}} \approx 1.006$—essentially no adjustment. For a .224" 55 gr varmint bullet, SD $\approx 0.157$, giving $f_{\text{SD}} \approx 0.793$—a 20% amplification of the degradation.

## 12.4 Physics-Bounded BC: Preventing Impossible Values

### 12.4.1 Why Bounds Matter

Whether you use constant BC, segments, or cluster degradation, the effective BC should always remain within physically realistic limits. A BC that drops to zero would imply infinite drag (the bullet stops instantly); a BC that goes to infinity would imply zero drag (the bullet never slows down). Neither is physically possible.

### 12.4.2 The Fallback BC System

BALLISTICS-ENGINE defines weight-class-specific fallback BC values in `src/constants.rs` to ensure that computations never use unrealistic values:

Listing 12.9: Fallback BC constants from `src/constants.rs`

```
pub const BC_FALLBACK_CONSERVATIVE: f64 = 0.31;
pub const BC_FALLBACK_ULTRA_LIGHT: f64 = 0.172; // 0-50 gr
pub const BC_FALLBACK_LIGHT: f64 = 0.242;       // 50-100 gr
pub const BC_FALLBACK_MEDIUM: f64 = 0.310;      // 100-150 gr
pub const BC_FALLBACK_HEAVY: f64 = 0.393;       // 150-200 gr
```

These values are derived from a statistical analysis of published BC data across hundreds of commercial bullets. When the BC interpolation or segment system returns an empty or invalid result, the engine substitutes the appropriate fallback value for the bullet's weight class.

**SAFETY: Do Not Rely on Fallback BCs**

The fallback BC values are safety nets, not substitutes for real data. A fallback BC of `0.310` for a 130 gr bullet might underestimate drag for a blunt hunting bullet (true BC `0.380`) or overestimate it for a streamlined VLD (true BC `0.550`). The resulting trajectory errors could translate to missed shots or, worse, shots that strike at unexpected locations. Always verify that your BC input is correct for your specific bullet and drag model.

### 12.4.3 Minimum Division Thresholds

Several computations in the BC pipeline involve divisions that could produce numerical instability. The engine uses a global constant `MIN_DIVISION_THRESHOLD` (set to $10^{-12}$) to guard against division by zero. When the denominator of any interpolation falls below this threshold, the engine returns the nearest known value rather than computing a garbage result.

## 12.5 The Transonic Challenge: Mach 1.0–1.2

### 12.5.1 What Happens Near Mach 1

The transonic regime—roughly Mach 0.8 to Mach 1.2—is where aerodynamic modelling is at its most difficult. The physics change qualitatively:

1. **Mach 0.8–0.85**: Local flow acceleration over the bullet's ogive first reaches Mach 1.0 even though the freestream velocity is still subsonic. This is the *critical Mach number*.
2. **Mach 0.85–1.0**: Localised shock waves form on the bullet surface, creating regions of supersonic flow embedded in an otherwise subsonic field. The drag coefficient rises sharply.
3. **Mach 1.0–1.05**: The bow shock forms ahead of the bullet. Drag peaks in this narrow band, typically reaching 1.5–2× the subsonic value.
4. **Mach 1.05–1.2**: The bow shock attaches to the nose and begins to take its characteristic conical shape. Drag decreases rapidly.
5. **Above Mach 1.2**: The bullet is cleanly supersonic. The drag model works well and the aerodynamics are predictable.

**Critical Mach Number**

The *critical Mach number* $M_{\text{crit}}$ is the freestream Mach number at which the local flow first reaches Mach 1.0 somewhere on the projectile surface. It depends on the nose shape:

- Spitzer: $M_{\text{crit}} \approx 0.85$
- Boat-tail: $M_{\text{crit}} \approx 0.88$
- Round nose: $M_{\text{crit}} \approx 0.75$
- Flat base: $M_{\text{crit}} \approx 0.70$

These values are defined by `critical_mach_number()` in `src/transonic_drag.rs`.

### 12.5.2 Why Standard Drag Tables Struggle Here

Standard drag tables (G1, G7, etc.) are measured from real projectiles, so they *do* capture the transonic drag rise for their specific reference shapes. The problem is that the transonic behaviour is the most *shape-dependent* part of the drag curve. The G1 standard's transonic drag rise is very different from a modern VLD's. When you scale the G1 curve by a constant form factor, the transonic peak gets scaled too—but its shape (width, steepness, peak location) remains wrong.

This is the fundamental reason why G7 outperforms G1 for modern bullets: the G7 reference shape has a transonic drag rise that is much closer to a typical match bullet's, so the scaling error is smaller.

## 12.6 Transonic Drag Corrections in ballistics-engine

### 12.6.1 The Correction Pipeline

To address the transonic problem, BALLISTICS-ENGINE provides an optional physics-based correction system implemented in `src/transonic_drag.rs`. When enabled (via the trajectory solver's transonic correction flag), this system modifies the base drag coefficient in the Mach 0.8–1.2 range to account for shape-specific shock wave effects.

The correction has two components:

1. **Transonic drag rise factor**: Multiplicative correction that models the extra drag as shock waves form and strengthen.
2. **Wave drag coefficient**: Additive correction for the drag component caused by shock waves themselves.

### 12.6.2 Projectile Shape Classification

The correction depends on the projectile's shape, represented by the `ProjectileShape` enum:

Listing 12.10: ProjectileShape enum from `src/transonic_drag.rs`

```
pub enum ProjectileShape {
    Spitzer,   // Sharp pointed (most common)
    RoundNose, // Blunt/round nose
    FlatBase,  // Flat base (wadcutter)
    BoatTail,  // Boat tail design
}
```

When the user does not explicitly specify a shape, the function `get_projectile_shape()` estimates it from the drag model and physical parameters:

- G7 drag model $\Rightarrow$ always `BoatTail`
- Weight/calibre ratio $> 500 \Rightarrow$ `BoatTail` (heavy for calibre implies long, boat-tailed design)
- Calibre $< 0.35'' \Rightarrow$ `Spitzer` (most rifle bullets)
- Calibre $\geq 0.35'' \Rightarrow$ `RoundNose` (larger calibres often have blunter noses)

### 12.6.3 The Transonic Drag Rise Model

The `transonic_drag_rise()` function computes a multiplicative factor that is applied to the base $C_D$ from the drag table. Below the critical Mach number, the factor is 1.0 (no correction). Between $M_{\text{crit}}$

and Mach 1.0, the factor rises according to a power law whose exponent depends on the projectile shape:

$$f_{\text{rise}} = 1 + k \cdot \left( \frac{M - M_{\text{crit}}}{1 - M_{\text{crit}}} \right)^{n} \tag{12.5}$$

where $k$ is the maximum rise amplitude and $n$ controls the steepness. For different shapes:

Table 12.4: Transonic drag rise parameters by shape

| **Shape** | $k$ | $n$ | **Character** |
|---|---|---|---|
| Boat-tail | 1.2 | 2.0 | Gentlest rise |
| Spitzer | 1.5 | 1.8 | Moderate |
| Round nose | 2.0 | 1.5 | Steepest rise |

Near Mach 1.0, an additional compressibility factor kicks in, modelling the Prandtl–Glauert singularity:

$$f_{\text{PG}} = \frac{1}{\sqrt{1 - M^2}}, \quad M < 1 \tag{12.6}$$

The engine caps this factor at 10.0 near Mach 0.99 to prevent numerical divergence, since the Prandtl–Glauert correction is theoretically infinite at Mach 1.0.

### 12.6.4 Post-Peak Drag Decay

Above Mach 1.0, the drag peaks at a shape-dependent Mach number (typically Mach 1.02–1.05) and then decays exponentially:

$$f_{\text{post}} = C_{\text{peak}} \cdot e^{-\alpha(M - M_{\text{peak}})} \tag{12.7}$$

where $C_{\text{peak}}$ is the peak drag rise factor (typically $\sim$1.8 for spitzers, $\sim$2.2 for round-nose bullets) and $\alpha$ is the decay rate (typically $\sim$3.0). Above Mach 1.2, the correction factor returns to 1.0 and the base drag table takes over.

### 12.6.5 Wave Drag Contribution

The wave drag coefficient is an *additive* correction—it represents the drag caused directly by shock waves, separate from the profile drag captured by the standard tables. Below Mach 0.8, wave drag is zero. In the transonic regime, it ramps up quadratically:

$$\Delta C_D^{\text{wave}} = 0.1 \cdot \left( \frac{M - M_{\text{crit}}}{1 - M_{\text{crit}}} \right)^2, \quad M_{\text{crit}} < M < 1.0 \tag{12.8}$$

Above Mach 1.0, the wave drag follows a modified Whitcomb area rule model:

$$\Delta C_D^{\text{wave}} = \frac{C_w}{\sqrt{M^2 - 1}} \cdot S_{\text{shape}} \tag{12.9}$$

where $C_w$ is a base wave drag coefficient (derived from the projectile's fineness ratio), and $S_{\text{shape}}$ is a shape correction factor:

- Boat-tail: $S = 0.7$ (best wave drag performance)
- Spitzer: $S = 0.8$
- Round nose: $S = 1.2$
- Flat base: $S = 1.5$ (worst wave drag performance)

### 12.6.6 The Complete Transonic Correction

The main entry point is `transonic_correction()` in `src/transonic_drag.rs`:

Listing 12.11: The transonic correction function

```
pub fn transonic_correction(
    mach: f64,
    base_cd: f64,
    shape: ProjectileShape,
    include_wave_drag: bool,
) -> f64 {
    let rise_factor = transonic_drag_rise(mach, shape);
    let mut corrected_cd = base_cd * rise_factor;

    if include_wave_drag && mach > 0.8 {
        corrected_cd += wave_drag_coefficient(mach, shape);
    }
    corrected_cd
}
```

This function is called from `get_drag_coefficient_with_transonic()` in `src/drag.rs` whenever the bullet's Mach number is between 0.8 and 1.3 and transonic corrections are enabled.

**When to Enable Transonic Corrections**

Transonic corrections are most valuable for long-range trajectories where the bullet decelerates through the transonic regime during flight. For a .308 Win at 2 700 fps, the transonic zone begins around 800–900 yards. If your maximum engagement range is under 600 yards, the corrections have negligible effect and you can safely leave them disabled.

## 12.7 Comparing Constant BC vs. Segmented BC vs. Cluster BC

### 12.7.1 Setting Up the Comparison

Let us compare the three approaches on a realistic scenario: .308 Win, 168 gr Sierra MatchKing, `0.462` G1, muzzle velocity 2 700 fps, standard atmosphere, out to 1 000 yards.

Listing 12.12: Constant BC baseline

```
# Model 1: Constant BC (default)
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000
```

Listing 12.13: Segmented BC

```
# Model 2: Automatic BC segments
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --use-bc-segments \
  --max-range 1000
```

Listing 12.14: Cluster BC degradation

```
# Model 3: Cluster-based BC degradation
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --use-cluster-bc \
  --max-range 1000
```

### 12.7.2 What to Expect

At short range (100–300 yards), all three models produce essentially identical results. The bullet has not decelerated enough for the BC variations to matter.

At medium range (400–700 yards), the segmented and cluster models begin to show slightly more drop than the constant-BC model—typically 1–3 inches. This is because they reduce the effective BC as velocity decreases, correctly accounting for the increasing form factor mismatch.

At long range (800–1 000 yards), the differences become significant. The constant-BC model underpredicts drop (because it uses a BC that is too optimistic at transonic velocities), while the segmented and cluster models converge on a more realistic value.

### 12.7.3 Which Model to Use?

Table 12.5: BC model selection guide

| Model | Best For | Limitations |
|---|---|---|
| Constant BC | Short–medium range (< 600 yd); quick calculations | Overpredicts performance at transonic/subsonic speeds |
| BC Segments | Extended range with known bullet type; competition use | Step function creates discontinuities; automatic estimation is approximate |
| Cluster BC | Maximum accuracy at all ranges; unknown bullet types | Requires calibre, weight, and BC; slightly more computation |

**Practical Recommendation**

For precision rifle competition at 600–1 200 yards, start with G7 BC and constant BC. If your observed drop does not match predictions at the extremes, enable `--use-cluster-bc` for automatic velocity-dependent correction. The cluster model adds negligible computation time and often closes the gap between predicted and observed impact.

## 12.8 Exercises

1. **Cluster assignment.** Which cluster does BALLISTICS-ENGINE assign to the following bullets? Think about it first, then verify by examining the centroid distances.
   - .223 Rem, 55 gr Nosler Ballistic Tip, `0.267` G1
   - 6.5 Creedmoor, 140 gr Berger Hybrid, `0.311` G7

- .338 Lapua Mag, 300 gr Sierra MatchKing, `0.768` G1

2. **Constant vs. segmented at 1 000 yards.** Run both constant-BC and segmented-BC trajectories for a 6.5 Creedmoor, 140 gr ELD Match (`0.610` G1), 2 710 fps, out to 1 000 yards. How much additional drop does the segmented model predict?

```
# Constant BC
ballistics trajectory \
  --bc 0.610 --drag-model g1 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --max-range 1000

# Segmented BC
ballistics trajectory \
  --bc 0.610 --drag-model g1 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --use-bc-segments \
  --max-range 1000
```

3. **The transonic dip in action.** Run a cluster BC trajectory for a .308 Win, 175 gr SMK (`0.505` G1, 2 600 fps) out to 1 200 yards. Use `--use-cluster-bc` and examine the velocity at each range step. At what range does the transonic dip (1 200–1 500 fps) occur? How does the trajectory's rate of drop change in that zone?

```
ballistics trajectory \
  --bc 0.505 --drag-model g1 \
  --velocity 2600 --mass 175 --diameter 0.308 \
  --use-cluster-bc \
  --max-range 1200
```

4. **Heavy magnum advantage.** Compare the drop at 1 000 yards for these two bullets using `--use-cluster-bc`:

   - .308 Win, 168 gr SMK, `0.462` G1, 2 700 fps
   - .338 Lapua Mag, 300 gr SMK, `0.768` G1, 2 700 fps

   The .338's higher BC and cluster assignment (Heavy Magnum) should produce dramatically less drop and better transonic retention.

```
# .308 Win 168gr
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
```

```
  --use-cluster-bc --max-range 1000

# .338 Lapua 300gr
ballistics trajectory \
  --bc 0.768 --drag-model g1 \
  --velocity 2700 --mass 300 --diameter 0.338 \
  --use-cluster-bc --max-range 1000
```

## What's Next

The models in this chapter describe how BC *should* vary with velocity—but how do you know what your bullet's BC actually is in the first place? In Chapter 13, we move from theory to measurement. We examine how published BC values are determined, how doppler radar and chronograph data differ, and how to "true" your BC with real-world shooting data so that your predictions match what you observe on the range.

# Chapter 13

# BC in Practice

The previous two chapters dissected drag models and advanced BC theory. This chapter puts that theory to work. We address the questions that every shooter eventually asks: *Where does my BC number actually come from? How accurate is it? And how do I fix it when my drops do not match the computer?*

The difference between a published BC on a box label and the effective BC of your specific barrel-and-load combination can easily be 5–10%. Over a 1 000-yard flight, that translates to half a minute of angle or more—enough to miss a target or, in a hunting scenario, enough to wound instead of cleanly harvesting an animal. Understanding BC data sources, measurement methods, and truing procedures is therefore not academic: it is essential to responsible long-range shooting.

## 13.1 Published BC Values: What They Mean and Where They Come From

### 13.1.1 Manufacturer-Stated BC

When Sierra lists the 168 gr MatchKing at `0.462` G1, that number represents the average drag performance of a sample of bullets tested under controlled conditions. But several caveats apply:

1. **Velocity-averaged**: Most published BCs are averaged over a velocity range that the manufacturer considers typical. Sierra, for example, historically published BC at multiple velocity bands (e.g., `0.462` above 2 800 fps, `0.447` at 2 100–2 800 fps). Many other manufacturers publish only a single number.

2. **Drag model is usually G1**: Unless the packaging or data sheet explicitly states "G7," the BC is almost certainly referenced to the G1 standard. Using a G1 BC with the G7 drag model (or vice versa) produces wildly incorrect trajectories.
3. **Test conditions vary**: Altitude, temperature, and atmospheric pressure during testing affect the Mach number at any given velocity. A BC measured at sea level in Aberdeen, Maryland is not the same as a BC measured at 5 000 feet in Colorado—though the difference is typically small (1–2%).
4. **Sample variation**: Bullet manufacturing has tolerances. Lot-to-lot variation in ogive profile, meplat diameter, and boat tail angle can shift the effective BC by 1–3%. Premium match bullets have tighter tolerances; hunting bullets can vary more.

**G1 vs. G7: Check the Label**

If you enter a manufacturer's G1 BC into BALLISTICS-ENGINE with `--drag-model g7`—or a G7 BC with `--drag-model g1`—your trajectory will be dramatically wrong. Always confirm which standard the BC is referenced to. When in doubt, assume G1.

### 13.1.2 Independent Sources

Several independent sources provide BC data that may differ from manufacturer claims:

- **Bryan Litz / Applied Ballistics**: Has published doppler-derived G1 and G7 BCs for hundreds of bullets. These are widely regarded as the most comprehensive independent BC dataset available.
- **Forum chronograph tests**: Experienced shooters sometimes publish BC measurements derived from two-point chronograph data. Quality varies widely.
- **Military testing**: Some military bullets have extensively documented drag curves from spark range or doppler testing, though the data may not be publicly available.

### 13.1.3 Which Atmosphere Was the BC Referenced To?

A ballistic coefficient is a ratio against a reference projectile *in a reference atmosphere*, and there are two atmospheres in circulation. Most modern published BCs—and this engine's own retardation constant—are referenced to ICAO sea-level density. But many Sierra, Hornady and Barnes BCs are still referenced to the older, denser *Army Standard Metro* atmosphere.

The two differ by about 1.8% in density, and because a denser reference atmosphere attributes the same physical drag to a numerically smaller BC, feeding an Army-Standard-Metro BC in unconverted reads about 1.8% too *little* drag. Since v0.29.0 you can simply say which kind you have:

Listing 13.1: Declaring an Army Standard Metro BC

```
# Sierra's published G1 BC for this bullet (0.475) is ASM-referenced.
# The engine internally solves with 0.475 x 0.98237, about 0.4666.
ballistics trajectory \
  --velocity 2700 --bc 0.475 --mass 168 --diameter 0.308 \
  --bc-reference army-standard-metro \
  --auto-zero 200 --sight-height 1.5 --max-range 1000
```

`icao` remains the default and is unchanged, so every existing invocation is byte-identical. The conversion constant is `ASM_TO_ICAO_BC` in `src/constants.rs`—the ratio of the two reference densities—and it has no effect at all when a custom `--drag-table` is in use, since a measured $C_D$ curve has no reference atmosphere to correct.

## 13.2 Doppler-Derived vs. Two-Point Chronograph Measurements

### 13.2.1 Doppler Radar: The Gold Standard

A doppler radar system tracks the bullet's velocity continuously from a few feet in front of the muzzle to impact (or as far as the radar can see). From this continuous velocity-distance curve, the drag coefficient at every Mach number can be computed directly:

$$C_D(M) = -\frac{2\,m}{\rho\,A} \cdot \frac{1}{V} \cdot \frac{dV}{dx} \tag{13.1}$$

where $m$ is the bullet mass, $\rho$ is air density, $A$ is the reference area, and $dV/dx$ is the measured velocity gradient. The resulting $C_D(M)$ curve can have hundreds of data points spanning from well above Mach 2 down through the transonic regime into subsonic flight.

**Doppler-Derived Drag Data**

Drag data obtained via doppler radar measurement are the most accurate characterisation of a bullet's aerodynamic performance available. Unlike BC values derived from two-point chronograph measurements, doppler data provide the complete $C_D(M)$ curve, including the critical transonic region. When we reference such data throughout this book, we call them "doppler-derived" to accurately describe the measurement methodology.

Advantages of doppler measurement:

- **Continuous data**: Every Mach number is sampled, including the transonic regime where BC changes most rapidly.

- **Shape-independent**: The measurement gives the actual $C_D(M)$ curve—no drag model assumption is needed.
- **High accuracy**: Modern tracking radars achieve velocity uncertainties well under 1%.

Disadvantages:

- **Expensive**: Doppler radar systems cost tens of thousands of dollars. Only manufacturers and research institutions typically have access.
- **Complex setup**: Requires a suitable range, careful calibration, and post-processing expertise.

### 13.2.2 Two-Point Chronograph Measurements

The two-point method measures velocity at two distances (e.g., 10 feet from the muzzle and 100 yards downrange) and computes the BC from the velocity loss:

$$\mathrm{BC} = \frac{D_{\text{space}}}{D_{\text{drag}}} \tag{13.2}$$

where $D_{\text{space}}$ is the distance between the two measurement points and $D_{\text{drag}}$ is the "drag space" from the standard drag function that corresponds to the measured velocity loss.

This method is accessible to any shooter with two chronographs (or a LabRadar/Magnetospeed and a downrange chronograph), but it has significant limitations:

1. **Single Mach range**: The result is a BC averaged over whatever velocity range the bullet traversed between the two measurement points. It says nothing about the bullet's behaviour at other velocities.
2. **Sensitive to velocity error**: A 1% error in either velocity measurement can shift the computed BC by 3–5%. Chronograph accuracy is typically 0.5–2%, depending on the instrument and conditions.
3. **Atmospheric uncertainty**: If the temperature, pressure, or humidity at the test site are not accurately known, the computed BC absorbs the atmospheric error.
4. **No transonic data**: Unless you can measure velocity at very long range where the bullet is transonic, the two-point method gives you only the supersonic BC.

**Lab vs. Field Chronographs**

Not all chronographs are equal. Optical chronographs (shooting through light screens) are affected by ambient light and can introduce systematic errors. Electromagnetic chronographs (Magnetospeed, LabRadar) are more reliable but have their own caveats—the Magnetospeed bayonet mount can shift point of impact, and LabRadar can lose tracking at long range or in rain.

### 13.2.3 How the Methods Compare

Table 13.1: BC measurement methods compared

| Attribute | Doppler Radar | Two-Point Chrono | Manufacturer |
|---|---|---|---|
| Mach range covered | Full | Single band | Varies |
| Transonic data | Yes | Usually no | Sometimes |
| Accuracy | $< 1\%$ | 3–5% | 2–5% |
| Accessibility | Very low | High | Free (printed) |
| Cost | $50K+ | $200–1K | $0 |
| Shape dependence | None | Via drag model | Via drag model |

## 13.3 Truing Your BC with Real-World Data

### 13.3.1 What Is Truing?

*Truing* is the process of adjusting your BC (or muzzle velocity) so that the solver's predictions match your actual observed impacts. It is the single most important step in transforming a ballistic calculator from a theoretical toy into a precision aiming tool.

The need for truing arises because:

- Published BCs may not match your specific lot of bullets.
- Your muzzle velocity may differ from the load data (barrel length, temperature, primer, powder lot all affect it).
- Local atmospheric conditions shift the effective drag.
- Your scope, mount, and zero distance introduce small systematic errors.

### 13.3.2 The Truing Workflow

The recommended truing procedure is:

1. **Confirm muzzle velocity**: Chronograph at least 10 rounds and use the average. This eliminates the largest source of error.

2. **Zero at a known distance**: Typically 100 yards (91 m).

3. **Shoot at extended range**: Fire a group at a distance where the predicted drop is significant—at least 400 yards, preferably 600+. Record the actual drop (in inches or MOA/mils) relative to your zero.

4. **Adjust BC to match**: In BALLISTICS-ENGINE, use the `--bc-adjustment` flag to apply a multiplier to the published BC. Iterate until the predicted drop matches the observed drop.

Listing 13.2: Truing BC with the adjustment flag

```
# Step 1: Baseline prediction at 600 yards
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 600

# Step 2: If actual drop is 4 inches more than predicted,
# reduce BC by ~3-5% (multiply by 0.96)
ballistics trajectory \
  --bc 0.462 --bc-adjustment 0.96 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 600

# Step 3: Iterate until prediction matches observation
ballistics trajectory \
  --bc 0.462 --bc-adjustment 0.94 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 600
```

**Truing at Two Distances**

For the most reliable truing, shoot at *two* distances—say 400 and 800 yards. Adjust muzzle velocity to match the near point and BC to match the far point. This separates the velocity error (which dominates at short range) from the BC error (which dominates at long range). The `--velocity-adjustment` flag in BALLISTICS-ENGINE lets you fine-tune the muzzle velocity without changing the base value.

### 13.3.3 The BC Estimation Command

BALLISTICS-ENGINE provides a dedicated `estimate-bc` subcommand that can estimate BC from trajectory data points:

Listing 13.3: Using the BC estimation command

```
ballistics estimate-bc \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --distance1 300 --drop1 -12.5 \
  --distance2 600 --drop2 -72.3
```

The underlying function (`estimate_bc_from_trajectory()` in the engine) works by iterating over candidate BC values and running a trajectory for each one, comparing the predicted drops to the observed drops. The BC that minimises the total error is reported.

**SAFETY: Truing Cannot Fix Bad Data**

If your muzzle velocity is wrong by 100 fps, truing the BC will produce a "corrected" BC that compensates for the velocity error—but only at the truing distance. The correction will be wrong at other distances because velocity error and BC error scale differently with range. **Always confirm your muzzle velocity with a chronograph before truing BC.** Similarly, truing in a 10 mph crosswind without accounting for wind drift will contaminate the BC estimate. True on a calm day, or account for wind in your drop measurements.

## 13.4 The BC5D Offline Correction Tables

### 13.4.1 What Are BC5D Tables?

The most sophisticated BC correction method in BALLISTICS-ENGINE uses pre-computed, calibre-specific **BC5D tables**. These are five-dimensional lookup tables that provide a correction factor for any combination of:

1. **Bullet weight** (grains)
2. **Base BC** (published value)
3. **Muzzle velocity** (fps)
4. **Current velocity** (fps)
5. **Drag model** (G1 or G7)

The correction factor is a ratio: multiply your published BC by the table's output to get the velocity-specific effective BC. Factors typically range from 0.5 to 1.5, with most values clustering between 0.90 and 1.05.

### 13.4.2 How BC5D Tables Are Generated

The tables are produced offline using machine learning models trained on thousands of measured drag curves. The ML model learns the relationship between bullet physical characteristics and the velocity-dependent drag deviation from the standard model. The trained model's predictions are then sampled on a dense grid across all five dimensions, and the results are stored as a binary lookup table.

This approach combines the accuracy of doppler-derived drag data (used for training) with the speed of table lookup (used at runtime). The engine never runs the ML model during trajectory computation—it simply interpolates the pre-computed table.

### 13.4.3 The BC5D Binary Format

Each calibre gets its own binary file (e.g., `bc5d_308.bin` for .308 calibre). The format, defined in `src/bc_table_5d.rs`, uses an 80-byte header:

Listing 13.4: BC5D header structure from `src/bc_table_5d.rs`

```
// Header (80 bytes):
//   Magic: 4 bytes ('BC5D')
//   Version: 4 bytes (uint32) -- must be 2
//   Caliber: 4 bytes (float32)
//   Flags: 4 bytes
//   Padding: 4 bytes
//   dim_weight: 4 bytes
//   dim_bc: 4 bytes
//   dim_muzzle_vel: 4 bytes
//   dim_current_vel: 4 bytes
//   dim_drag_types: 4 bytes
//   timestamp: 8 bytes
//   checksum: 4 bytes (CRC32 of data section)
//   api_version: 16 bytes (string)
//   reserved: 12 bytes
```

The header is followed by bin definitions (the grid coordinates for each dimension) and then the correction factor data as a flat array of `float32` values, laid out in row-major order: `[drag_type][weight][bc][muzzle_vel]`

A CRC32 checksum protects against data corruption—the engine validates the checksum on load and rejects corrupted files.

### 13.4.4 4D Linear Interpolation

At query time, the `Bc5dTable::lookup()` method performs 4D linear interpolation (the drag model dimension is discrete, not interpolated). This means the method evaluates 16 corners of a 4D hypercube—one for each combination of upper/lower bin values in the weight, BC, muzzle velocity, and current velocity dimensions:

Listing 13.5: 4D interpolation from `src/bc_table_5d.rs`

```
pub fn lookup(&self, weight_grains: f64, base_bc: f64,
              muzzle_velocity: f64, current_velocity: f64,
              drag_type: &str) -> f64
{
    let drag_idx = if drag_type == "G7" { 1 } else { 0 };

    let (weight_idx, weight_w) =
        self.interp_idx(weight_grains as f32, &self.weight_bins);
    let (bc_idx, bc_w) =
        self.interp_idx(base_bc as f32, &self.bc_bins);
    // ... (muzzle_vel, current_vel similar)

    // 16 corners of the 4D hypercube
    let mut result = 0.0f64;
    for dw in 0..2 {
      for db in 0..2 {
        for dm in 0..2 {
          for dc in 0..2 {
            let w = corner_weight(dw, db, dm, dc);
            let idx = self.flat_index(drag_idx, ...);
            result += w * self.data[idx] as f64;
    }}}}
    result.max(0.5).min(1.5) // Clamp to valid range
}
```

The result is clamped to the $[0.5, 1.5]$ range to prevent extreme corrections from bad interpolation at table edges.

### 13.4.5 Using BC5D Tables from the CLI

Listing 13.6: Using BC5D correction tables

```
# Point to a directory of BC5D tables
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
```

```
  --bc-table-dir /path/to/bc5d_tables \
  --max-range 1000
```

The `Bc5dTableManager` automatically loads the correct calibre-specific file based on the `--diameter` value. Files are cached in memory after first load, so subsequent trajectories for the same calibre incur no file I/O overhead.

Listing 13.7: Auto-downloading BC5D tables (online mode)

```
# Auto-download tables as needed
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --bc-table-auto \
  --max-range 1000
```

With `--bc-table-auto`, the engine will download the appropriate table from the default URL (`https://ballistics.tools/downloads/bc5d`) if it is not already cached locally.

**Checking Available Calibres**

The `Bc5dTableManager::available_calibers()` method scans the table directory for all `bc5d_*.bin` files and returns the list of available calibres. If your calibre is not available, the engine falls back to the standard drag model without 5D correction.

### 13.4.6 BC5D vs. Other Correction Methods

Table 13.2: BC correction methods compared

| Attribute | BC5D | Cluster BC | BC Segments |
|---|---|---|---|
| Input needed | Weight, BC, MV | Calibre, weight, BC | BC, bullet type |
| Calibre-specific | Yes | No (cluster-based) | No |
| Transonic detail | High (dense grid) | Moderate | Low (step function) |
| Storage required | 1–10 MB per calibre | Zero | Zero |
| Offline use | Requires table files | Always available | Always available |

The BC5D tables provide the highest fidelity correction because they were trained on real drag curves and include the muzzle velocity as an input dimension—a subtle but important factor, since a bullet that starts at 3 200 fps enters the transonic zone under different aerodynamic conditions than one that starts at 2 400 fps.

## 13.5 Common Mistakes and How to Avoid Them

Over years of ballistic computation, certain errors recur with remarkable consistency. This section catalogues the most common BC-related mistakes and how to avoid them.

### 13.5.1 Mistake 1: Mixing G1 and G7 BCs

**The error**: Entering a G1 BC with `--drag-model g7`, or vice versa.

**The consequence**: A .308 Win 168 gr SMK with `0.462` G1 entered as G7 will predict that the bullet is a laser beam—almost no drop at 1 000 yards. The reverse (G7 BC with G1 model) predicts a trajectory that falls like a mortar round.

**The fix**: Always verify which standard your BC is referenced to. When in doubt, assume G1.

Listing 13.8: Correct and incorrect BC usage

```
# CORRECT: G1 BC with G1 model
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000

# CORRECT: G7 BC with G7 model
ballistics trajectory \
  --bc 0.243 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 1000

# WRONG: G1 BC with G7 model -- DO NOT DO THIS
# ballistics trajectory \
#   --bc 0.462 --drag-model g7 ...
```

### 13.5.2 Mistake 2: Trusting a Single Published BC

**The error**: Assuming the box BC is perfectly accurate and never validating it.

**The consequence**: Published BCs are averaged across velocity ranges, and may not account for your specific conditions. A 3% BC error produces ∼5 inches of drop error at 1 000 yards.

**The fix**: True your BC with real-world data at extended range, as described in Section 13.3.

### 13.5.3 Mistake 3: Truing BC Without Confirming Muzzle Velocity

**The error**: Truing the BC to match observed drop without first chronographing your load.

**The consequence**: The "trued" BC compensates for the muzzle velocity error—but only at the truing distance. At other distances, the compensation is wrong because velocity error and BC error scale differently.

**The fix**: Chronograph your load first. Adjust muzzle velocity to match chronograph data, *then* true BC to match extended-range drop.

### 13.5.4 Mistake 4: Using Stale Atmospheric Data

**The error**: Computing a trajectory using sea-level standard atmosphere when shooting at 5 000 feet and 95°F.

**The consequence**: At high altitude, the air is thinner, so the bullet experiences less drag and retains velocity longer. The actual trajectory is flatter than the prediction, causing the shooter to *over-correct* and shoot high.

**The fix**: Always enter current atmospheric conditions via `--altitude`, `--temp`, and `--pressure`. See Chapter 9 for details on atmospheric modelling.

### 13.5.5 Mistake 5: Ignoring the Transonic Zone

**The error**: Assuming a constant BC is adequate for ranges where the bullet goes transonic.

**The consequence**: The constant-BC model underpredicts drag in the transonic zone, causing the trajectory to show less drop than actually occurs. The error can be 6–12 inches at 1 000 yards.

**The fix**: For ranges where the bullet drops below ∼1 400 fps, use one of the velocity-dependent BC methods: `--use-bc-segments`, `--use-cluster-bc`, or BC5D tables via `--bc-table-dir`.

### 13.5.6 Mistake 6: Over-Trueing to One Distance

**The error**: Adjusting BC until the prediction is perfect at exactly one range (say 700 yards), then assuming it is correct everywhere.

**The consequence**: The trued BC may overfit to that distance. At shorter and longer ranges, the predictions may actually be *worse* than with the original BC.

**The fix**: True using at least two distances—one near (400 yd) and one far (800+ yd). If both match, the BC is well-characterised. If they cannot be made to match simultaneously, the issue is likely a muzzle velocity error or an atmospheric data problem, not a BC problem.

**The Two-Distance Truing Protocol**

1. Zero at 100 yards.
2. Shoot 5-round groups at 400 and 800 yards.
3. In BALLISTICS-ENGINE, adjust `--velocity-adjustment` until the 400-yard prediction matches.
4. Then adjust `--bc-adjustment` until the 800-yard prediction matches.
5. Verify at 600 yards—if that matches too, you are well-trued.

## 13.6 Exercises

1. **The G1/G7 mix-up.** Run these two commands and compare the 1 000-yard drop predictions. How large is the error if you use the wrong drag model?

```
# Correct: G1 BC with G1 model
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 1000

# Wrong: Same BC but with G7 model
ballistics trajectory \
  --bc 0.462 --drag-model g7 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --auto-zero 100 --max-range 1000
```

2. **Truing exercise.** Suppose you are shooting .308 Win, 175 gr SMK at a published `0.505` G1 and 2 600 fps muzzle velocity. Your observed drop at 700 yards (zeroed at 100) is 8 inches *more* than the solver predicts with the published BC. Use `--bc-adjustment` to find the adjustment factor that closes the gap.

```
# Start with published BC
ballistics trajectory \
  --bc 0.505 --drag-model g1 \
  --velocity 2600 --mass 175 --diameter 0.308 \
  --auto-zero 100 --max-range 700

# Try bc-adjustment of 0.93
ballistics trajectory \
  --bc 0.505 --bc-adjustment 0.93 --drag-model g1 \
  --velocity 2600 --mass 175 --diameter 0.308 \
  --auto-zero 100 --max-range 700
```

3. **BC5D table exploration.** If you have BC5D tables installed, run the same trajectory with and without the table and compare the 800-yard drop:

```
# Without BC5D correction
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --max-range 800

# With BC5D correction
ballistics trajectory \
  --bc 0.462 --drag-model g1 \
  --velocity 2700 --mass 168 --diameter 0.308 \
  --bc-table-dir ./bc5d_tables \
  --max-range 800
```

4. **Altitude effect on effective BC.** Compute the trajectory for a 6.5 Creedmoor, 140 gr ELD Match (`0.610` G1, 2 710 fps) at sea level and again at 5 000 feet (1524 m). How much does the remaining velocity at 800 yards change? The BC is the same, but the effective drag changes with air density.

```
# Sea level
ballistics trajectory \
  --bc 0.610 --drag-model g1 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --altitude 0 --max-range 800

# 5000 feet
ballistics trajectory \
  --bc 0.610 --drag-model g1 \
  --velocity 2710 --mass 140 --diameter 0.264 \
  --altitude 5000 --max-range 800
```

## What's Next

With drag models, advanced BC modelling, and practical BC work behind us, we have covered the complete aerodynamic pipeline: from a bullet's physical shape to the drag force that governs its trajectory. But drag is not the only force acting on a bullet in flight. A spinning projectile is a gyroscope, and gyroscopic effects produce their own family of lateral deflections.

In Chapter 14, we turn to the physics of spin: how rifling creates gyroscopic stability, why spin drift pushes bullets sideways, and how BALLISTICS-ENGINE models the Magnus effect. These spin-induced

forces are small compared to drag and gravity, but at extreme range they become significant—and unlike wind, they are deterministic and predictable.

# Part V

# Advanced Physics

# Chapter 14

# Spin Effects

A bullet that does not spin is, aerodynamically speaking, a poorly shaped rock. It will tumble within metres of leaving the muzzle, and its trajectory will be unpredictable. The rifled barrel—one of the great innovations in the history of firearms—imparts a rapid axial spin that gyroscopically stabilises the bullet, pointing it nose-first into the airstream. But that stabilising spin is not free: it introduces a family of lateral effects that, at long range, rival or exceed wind deflection in magnitude.

This chapter explores every spin-related phenomenon that BALLISTICS-ENGINE models: gyroscopic stability and the stability factor, spin drift (the slow lateral walk caused by the bullet's rotation), spin decay over distance, and the Magnus effect. By the end, you will understand *why* a right-twist barrel pushes bullets to the right, *how much* drift to expect for realistic loads, and *when* spin effects cross the threshold from academic curiosity to practical concern.

## 14.1 Why Bullets Spin: Rifling and Gyroscopic Stability

### 14.1.1 The Problem of Aerodynamic Instability

An elongated projectile flying point-forward through the air is *statically unstable*: its centre of pressure lies ahead of its centre of gravity, so any small perturbation creates a nose-up (or nose-down) torque that amplifies the disturbance. Without some restoring mechanism, the bullet tumbles.

**Static vs. Dynamic Stability**

**Static stability** asks: if I perturb the bullet, does the initial torque push it back toward equilibrium? For a spinning bullet in air the answer is *no*—the overturning moment pushes it further from its flight path. **Gyroscopic stability** replaces static stability: the spin axis precesses around the velocity vector instead of diverging, and damping forces gradually align the two.

### 14.1.2 Rifling: Imparting Spin

Rifling consists of helical grooves cut into the bore. As the bullet travels down the barrel, the lands and grooves engage the jacket and force the bullet to rotate. The rate of this helical cut is expressed as the *twist rate*—the distance the bullet must travel for one complete revolution.

A "1:10-inch twist" means one full turn every 10 inches (254 mm) of barrel travel. A faster twist (e.g. 1:7″) spins the bullet faster and stabilises longer, heavier projectiles; a slower twist (e.g. 1:14″) is sufficient for shorter, lighter bullets and produces less spin-induced lateral deflection.

### 14.1.3 Computing Spin Rate from Twist and Velocity

The bullet's initial spin rate depends on two quantities: its muzzle velocity $V_0$ and the barrel's twist rate $T$ (in length per revolution). The relationship is straightforward:

$$n = \frac{V_0}{T} \tag{14.1}$$

where $n$ is the spin rate in revolutions per second (rps). To convert to radians per second:

$$\omega = 2\pi\, n = \frac{2\pi\, V_0}{T} \tag{14.2}$$

In BALLISTICS-ENGINE, the function `calculate_spin_rate()` in `src/spin_drift.rs` implements exactly this conversion. It takes velocity in metres per second and twist rate in inches per turn:

Listing 14.1: Spin rate calculation from `src/spin_drift.rs`

```
pub fn calculate_spin_rate(velocity_mps: f64,
                           twist_rate_inches: f64)
    -> (f64, f64)
{
    let velocity_ips = velocity_mps * 39.3701;
    let spin_rate_rps = velocity_ips / twist_rate_inches;
    let spin_rate_rad_s = spin_rate_rps * 2.0 * PI;
    (spin_rate_rps, spin_rate_rad_s)
```

```
}
```

**Spin Rate Rule of Thumb**

A .308 Win firing a 168 gr Sierra MatchKing at 2 700 fps (823 m/s) from a 1:10″ twist barrel spins at approximately **3 240 rps** (about 194 000 rpm). That is roughly the tip speed of a jet turbine compressor blade. Even at 1 000 yards, the bullet is still spinning at over 2 800 rps.

## 14.2 Spin Drift: The Horizontal Deflection from Spin

### 14.2.1 What Is Spin Drift?

Spin drift—sometimes called *gyroscopic drift*—is the slow lateral displacement of a spinning bullet caused by the interaction between gyroscopic precession and gravity. A bullet fired from a right-hand twist barrel drifts to the *right*; a left-hand twist barrel produces a drift to the *left*.

**Spin Drift (Gyroscopic Drift)**

The cumulative lateral displacement of a spin-stabilised projectile caused by the yaw of repose—a small, steady angle between the bullet's longitudinal axis and the velocity vector. This yaw arises because gyroscopic precession converts the gravity-induced pitching moment into a steady yaw, and the resulting angle of attack produces a side force that deflects the bullet laterally.

### 14.2.2 The Physics: Yaw of Repose

When a bullet follows a curved trajectory (as all trajectories are, since gravity pulls the bullet downward), its velocity vector rotates slowly nose-down. The overturning aerodynamic moment acts on the spinning bullet, and gyroscopic precession converts this pitching moment into a slow yaw—a steady deflection of the nose to one side. For a right-spinning bullet (viewed from behind), the nose points slightly to the right.

This equilibrium angle is called the *yaw of repose*, typically denoted $\beta_{\text{eq}}$. It is small—usually less than 0.1 degrees for a well-stabilised bullet—but because it is sustained over the entire flight, the resulting side force accumulates into a significant lateral displacement at long range.

The yaw of repose depends on the gyroscopic stability factor $S_g$, the bullet velocity, the spin rate, and the aerodynamic environment. In the simplest model (the "backward compatibility" path in `src/spin_drift.rs`), BALLISTICS-ENGINE uses:

$$\beta_{\text{eq}} \approx \frac{0.002}{1 + \sqrt{S_g - 1}} \tag{14.3}$$

for the no-wind case, where the 0.002 rad (≈0.1°) baseline represents the yaw induced by trajectory curvature alone. When crosswind is present, the wind-induced yaw replaces the baseline:

$$\beta_{\text{wind}} = \arctan\left(\frac{V_{\text{crosswind}}}{V}\right) \cdot \frac{1}{1 + \sqrt{S_g - 1}} \tag{14.4}$$

### 14.2.3 The Litz Empirical Formula

While the physics of spin drift can be derived from first principles using the equations of motion for a symmetric spinning body, the most practical formula for shooters comes from Bryan Litz's extensive empirical testing. The Litz formula, implemented in `src/spin_drift.rs` as `calculate_gyroscopic_drift()`, is:

$$\boxed{\text{SD} = 1.25\,(S_g + 1.2) \cdot t^{1.83}} \tag{14.5}$$

where:

- SD is the spin drift in inches,
- $S_g$ is the gyroscopic stability factor,
- $t$ is the time of flight in seconds,
- the direction is positive (right) for right-hand twist and negative (left) for left-hand twist.

The 1.83 exponent reflects the non-linear growth of spin drift: it accelerates as time of flight increases, which is why spin drift is negligible at short range but can exceed 10 inches at 1 000 yards for typical .308 loads.

**Why 1.83?**

The 1.83 exponent arises because spin drift is driven by a *force* (not an impulse). The lateral displacement from a constant side force would grow as $t^2$; the sub-quadratic exponent reflects the fact that the yaw of repose—and therefore the side force—decreases slowly as the bullet decelerates and spin decays.

## 14.3 Computing Spin Drift in `ballistics-engine`

### 14.3.1 Enabling Spin Drift

Spin drift is an opt-in feature. To include it in a trajectory computation, you need three things:

1. The `--twist-rate` flag (in inches per turn).
2. The `--enable-spin-drift` flag to activate the enhanced calculation.

3. Optionally, `--twist-right` (defaults to `true`).

Here is a complete example for a .308 Win, 168 gr Sierra MatchKing (`0.462` G1) at 2 700 fps from a 1:10″ right-hand twist barrel:

Listing 14.2: Trajectory with spin drift enabled

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --twist-right \
  --enable-spin-drift \
  --bullet-length 1.215 \
  --sight-height 1.5
```

The output will include a "Spin Drift" line showing the cumulative lateral deflection at the maximum range:

Listing 14.3: Spin drift output

```
# Output includes:
# ...
# Spin Drift:           8.42 in   Right
```

**Bullet Length Is Required**

The stability factor $S_g$ drives the magnitude of spin drift. Computing $S_g$ requires the bullet's physical length, which you supply with `--bullet-length` (in inches for imperial mode). Without it, the engine cannot compute $S_g$ and spin drift will be zero.

### 14.3.2 The Standard vs. Advanced Model

The engine provides two spin drift calculation paths:

1. **Standard model** (via `compute_spin_drift_with_decay()` in `src/stability.rs`): Uses a modified Litz formula with a reduced scaling factor (0.075 instead of 1.25) and optional spin decay. This is the path used when `--enable-spin-drift` is set on the `trajectory` subcommand.

2. **Advanced model** (via `calculate_advanced_spin_drift()` in `src/spin_drift_advanced.rs`): Combines the Litz formula with McCoy's aerodynamic jump correction, a transonic correction factor, yaw damping, and velocity decay corrections. This model uses bullet-type-specific coefficients from the `SpinDriftCoefficients` struct.

The advanced model accounts for additional phenomena:

- **Aerodynamic jump correction** (McCoy): A one-time lateral displacement caused by the bullet exiting the muzzle with a small initial yaw.
- **Transonic correction** (Courtney & Courtney): A multiplicative factor that accounts for the altered aerodynamic behaviour in the transonic zone (Mach 0.8–1.2).
- **Yaw damping factor**: A time-dependent correction that accounts for the decay of initial yaw oscillations.
- **Velocity decay correction**: Adjusts the drift rate as the bullet slows relative to its muzzle velocity.
- **Atmospheric density correction**: Scales drift by $\sqrt{\rho_0/\rho}$, where $\rho_0 = 1.225\,\mathrm{kg/m^3}$ is sea-level standard density.

### 14.3.3 Bullet-Type Coefficients

The advanced model uses empirically derived coefficients that vary by bullet construction. These are defined in `SpinDriftCoefficients::for_bullet_type()` in `src/spin_drift_advanced.rs`:

Table 14.1: Spin drift coefficients by bullet type

| Bullet Type | Litz Coeff. | McCoy Jump | Transonic | Yaw Damp. |
|---|---|---|---|---|
| Match / BTHP | 1.25 | 0.85 | 0.75 | 0.92 |
| VLD | 1.15 | 0.78 | 0.68 | 0.88 |
| Hybrid ogive | 1.20 | 0.82 | 0.72 | 0.90 |
| Flat base | 1.35 | 0.95 | 0.85 | 0.95 |

VLD (Very Low Drag) bullets drift less because their sleek profile generates less overturning moment. Flat-base bullets drift more because their blunter base creates stronger wake effects that amplify the side force.

### 14.3.4 A Worked Example: .308 Win at 1 000 Yards

Let us trace the computation step by step for our reference load: 168 gr SMK, `0.462` G1, 2 700 fps, 1:10″ RH twist.

1. **Spin rate**: $n = 2700 \times 12/10 = 3240$ rps ($\omega = 20{,}358$ rad/s).
2. **Stability factor**: Using the Miller formula (see Section 17.1 in Chapter 17), $S_g \approx 1.8$.
3. **Time of flight to 1 000 yd**: Approximately 1.55 s.
4. **Litz formula**: $\mathrm{SD} = 1.25 \times (1.8 + 1.2) \times 1.55^{1.83} \approx 8.6$ inches right.

At 1 000 yards, the spin drift of 8.6 inches (≈0.8 MOA) is comparable to a 2-mph crosswind. At shorter ranges it is negligible: at 300 yards, the same bullet drifts only about 0.3 inches.

## 14.4 Spin Decay over Distance

### 14.4.1 Why Spin Decays

A bullet is not spinning in a vacuum. The interaction between the rotating surface and the surrounding air creates skin friction torque that continuously opposes the spin. Over the course of a long flight, the spin rate declines measurably.

The physics is governed by the aerodynamic damping moment, which depends on:

- **Surface roughness**: Smoother jackets (match bullets) produce less friction.
- **Skin friction coefficient**: Related to the Reynolds number of the spinning surface.
- **Air density**: Denser air creates more friction.
- **Spin rate itself**: Higher spin rates produce higher tangential velocities and therefore more friction (the damping moment scales roughly as $\omega^2$).

### 14.4.2 The Exponential Decay Model

In `src/spin_decay.rs`, the `update_spin_rate()` function models spin decay with an exponential decay law:

$$\omega(t) = \omega_0 \cdot e^{-\lambda t} \tag{14.6}$$

where $\omega_0$ is the initial spin rate and $\lambda$ is the effective decay rate per second. The decay rate is computed from several physical factors:

$$\lambda = \lambda_{\text{base}} \cdot \underbrace{\sqrt{\frac{m_{\text{ref}}}{m}}}_{\text{mass factor}} \cdot \underbrace{\frac{V}{V_{\text{ref}}}}_{\text{velocity factor}} \cdot \underbrace{\frac{\rho}{\rho_0}}_{\text{density factor}} \cdot \underbrace{\sqrt{\frac{A}{A_{\text{ref}}}}}_{\text{surface factor}} \tag{14.7}$$

where:

- $\lambda_{\text{base}}$ is the baseline decay rate: 0.025/s (2.5%/s) for match bullets, 0.04/s (4%/s) for hunting/FMJ bullets.
- $m_{\text{ref}} = 175\,\text{gr}$, $V_{\text{ref}} = 850\,\text{m/s}$, $\rho_0 = 1.225\,\text{kg/m}^3$ are reference values.
- $A$ and $A_{\text{ref}}$ are the lateral surface areas of the bullet and a reference .308 bullet ($\pi \times 0.308 \times 1.3$ in²).

The result is clamped so that a bullet never loses more than 50% of its initial spin, consistent with published experimental data showing that well-designed bullets retain the vast majority of their spin over practical flight durations.

**How Much Spin Is Actually Lost?**

For a typical .308 match bullet (168 gr, 823 m/s), the model predicts 5–8% spin loss over a 3-second flight to approximately 1 200 yards. This is consistent with doppler-derived measurements of actual spin rates at range. Heavier bullets (e.g. 190 gr .30-cal) retain spin better because of their higher rotational inertia relative to aerodynamic friction.

### 14.4.3 Bullet Type and Surface Quality

The `SpinDecayParameters` struct in `src/spin_decay.rs` encodes the surface properties that affect decay rate:

Table 14.2: Spin decay parameters by bullet type

| Bullet Type | Surface Roughness (m) | Skin Friction Coeff. | Form Factor |
|---|---|---|---|
| Match | $5 \times 10^{-5}$ | $8 \times 10^{-6}$ | 0.9 |
| Hunting | $1 \times 10^{-4}$ | $1 \times 10^{-5}$ | 1.0 |
| FMJ | $1.5 \times 10^{-4}$ | $1.2 \times 10^{-5}$ | 1.1 |
| Cast | $2 \times 10^{-4}$ | $1.5 \times 10^{-5}$ | 1.2 |

Match bullets have the smoothest jackets and the lowest form factor, resulting in the slowest spin decay. Cast bullets, with their rougher surfaces, lose spin approximately 60% faster.

### 14.4.4 Moment of Inertia

The bullet's resistance to spin-down depends on its axial moment of inertia $I$. The `calculate_moment_of_inertia()` function provides three models based on bullet shape:

$$I = k \cdot m \cdot r^2 \tag{14.8}$$

where $k$ varies by geometry:

- **Cylinder**: $k = 0.50$ (uniform solid cylinder)
- **Ogive**: $k = 0.40$ (mass concentrated toward the centre)
- **Boat tail**: $k = 0.35$ (even less mass at the rear)

The angular deceleration is then $\dot{\omega} = -M_{\text{damp}}/I$, where $M_{\text{damp}}$ is the total damping moment from skin friction and Magnus damping.

### 14.4.5 Observing Spin Decay in Practice

While the CLI does not display the instantaneous spin rate at each range increment, you can observe its effect through the spin drift output. The standard model applies the spin decay correction factor to the drift calculation:

Listing 14.4: Trajectory with spin drift and decay effects

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1200 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.215 \
  --sight-height 1.5
```

The correction factor is computed by `calculate_spin_decay_correction_factor()` in `src/spin_decay.rs`, which returns a value between 0.5 and 1.0 representing the fraction of initial spin remaining. A value of 0.93, for example, means the bullet has lost 7% of its initial spin.

## 14.5 The Magnus Effect and Its Contribution

### 14.5.1 What Is the Magnus Effect?

The Magnus effect is a force that acts on a spinning body moving through a fluid. You have seen it in action every time a pitcher throws a curveball or a soccer player bends a free kick: the spin creates an asymmetry in the airflow that generates a lateral force perpendicular to both the spin axis and the velocity vector.

For a spinning bullet with a small yaw angle, the Magnus effect creates an additional lateral force beyond the gyroscopic drift. The spinning bullet's upper and lower surfaces have different relative airflow velocities (one adds to the free-stream velocity, the other subtracts), creating a pressure differential.

**The Magnus Force on a Bullet**

For a bullet with spin rate $\omega$, yaw angle $\beta$, diameter $d$, and velocity $V$ in air of density $\rho$:

$$F_{\text{Magnus}} = C_{\text{mag}} \cdot \frac{\omega\, d}{2V} \cdot \beta \cdot \frac{1}{2}\rho V^2 \cdot \pi \left(\frac{d}{2}\right)^2 \tag{14.9}$$

where $C_{\text{mag}}$ is the Magnus force coefficient—an empirical quantity that varies with Mach number.

### 14.5.2 Magnus Coefficient vs. Mach Number

The `calculate_magnus_drift_component()` function in `src/spin_drift.rs` uses a three-regime model for the Magnus coefficient:

$$C_{\text{mag}} = \begin{cases} 0.25 & \text{if } M < 0.8 \quad \text{(subsonic)} \\ 0.15 & \text{if } 0.8 \leq M < 1.2 \quad \text{(transonic)} \\ 0.10 + 0.05 \cdot \min\left(\dfrac{M - 1.2}{2.0},\, 1\right) & \text{if } M \geq 1.2 \quad \text{(supersonic)} \end{cases} \tag{14.10}$$

The transonic reduction reflects the chaotic aerodynamic environment near Mach 1, where the Magnus moment is partially disrupted by shock wave interactions on the bullet surface.

### 14.5.3 Magnus Drift Magnitude

The Magnus contribution to total drift is typically smaller than the gyroscopic component. For our reference .308 load at 1 000 yards, the Magnus drift adds approximately 0.5–1.5 inches to the total, depending on crosswind conditions. In zero wind, the Magnus contribution is minimal because the yaw of repose is small.

However, in crosswind conditions, the increased yaw amplifies the Magnus effect significantly. A 10-mph crosswind can double the yaw angle, which proportionally increases the Magnus drift component.

**Magnus and Crosswind Interaction**

The Magnus effect and wind deflection are *not independent*. A crosswind increases the effective yaw angle, which increases both the Magnus force and the gyroscopic precession rate. This interaction means that simply adding separate wind drift and spin drift corrections underestimates the total lateral deflection in crosswind conditions.

### 14.5.4 The Enhanced Spin Drift: Combining Gyroscopic and Magnus Components

The `calculate_enhanced_spin_drift()` function in `src/spin_drift.rs` brings together all spin-related effects into a single `SpinDriftComponents` struct. The total drift is the sum of the gyroscopic and Magnus components:

$$\text{SD}_{\text{total}} = \text{SD}_{\text{gyroscopic}} + \text{SD}_{\text{Magnus}} \tag{14.11}$$

The function also tracks auxiliary quantities that are useful for diagnostics:

- **Current spin rate** (after decay)

- **Stability factor** (recomputed with decayed spin)
- **Yaw of repose** (degrees)
- **Drift rate** (lateral velocity, m/s)
- **Pitch damping moment** (if the enhanced model is active)
- **Yaw convergence rate** (how quickly the bullet settles to equilibrium)

## 14.6 Twist Rate and Its Effect on Stability

### 14.6.1 The Twist–Stability Relationship

The gyroscopic stability factor $S_g$ is inversely proportional to the square of the twist rate (measured in calibres per turn). A faster twist (smaller number) produces a higher $S_g$. The Miller formula, discussed in detail in Chapter 17, is:

$$S_g = \frac{30\,m}{t_c^2\,d^3\,l_c\,(1 + l_c^2)} \cdot \left(\frac{V}{2800}\right)^{1/3} \cdot \sqrt{\frac{\rho_0}{\rho}} \tag{14.12}$$

where $m$ is the mass in grains, $t_c$ is the twist in calibres, $d$ is the diameter in inches, $l_c$ is the length in calibres, $V$ is the velocity in fps, and $\rho$ is the air density.

### 14.6.2 Using the `stability` Subcommand

The `stability` subcommand computes the Miller stability factor and reports the minimum twist rates needed for adequate stability:

Listing 14.5: Stability analysis for a 6.5 Creedmoor

```
ballistics stability \
  --mass 140 --diameter 0.264 --length 1.375 \
  --twist-rate 8 --velocity 2700
```

Listing 14.6: Sample stability output

```
# Output:
# +=======================================+
# |         Stability Analysis            |
# +=======================================+
# | Stability Factor (Sg):          1.92  |
# | Status:              Fully Stable     |
# | Twist Rate:              1:8" RH      |
# | Min Twist (Sg=1.5):       1:9.2"      |
# | Min Twist (Sg=1.0):      1:11.3"      |
```

```
# | Bullet Length:            1.375"        |
# | Muzzle Velocity:         2700 fps       |
# +=========================================+
```

This output tells you:

- Your 1:8″ twist gives $S_g = 1.92$ — fully stable.
- You could slow the twist to 1:9.2″ and still have $S_g \geq 1.5$ (the recommended minimum).
- Below 1:11.3″, the bullet becomes *gyroscopically unstable* ($S_g < 1.0$).

### 14.6.3 Over-Stabilisation and Its Effects

While under-stabilisation causes tumbling, over-stabilisation ($S_g > 3$) has its own problems. An over-stabilised bullet resists the natural nose-down pitching that should follow the trajectory arc. Instead of pointing along the flight path, the bullet flies increasingly "nose up" relative to its velocity vector, increasing drag and reducing accuracy at long range.

Listing 14.7: Checking a light bullet in a fast-twist barrel

```
# 55gr .224 bullet in a 1:7" twist (common for AR-15)
ballistics stability \
  --mass 55 --diameter 0.224 --length 0.740 \
  --twist-rate 7 --velocity 3240
# Sg will be quite high (>2.5) -- this bullet is
# over-stabilised for this twist rate
```

For practical shooting, $S_g$ values between 1.4 and 2.0 are ideal. Above 2.5, the bullet's reluctance to follow the trajectory arc becomes measurable; above 3.5, it can affect long-range accuracy noticeably.

## 14.7 Practical Implications: When Does Spin Drift Matter?

### 14.7.1 The Range Threshold

For most rifle cartridges, spin drift is negligible inside 300 yards. Between 300 and 600 yards, it is detectable but often smaller than other error sources (wind estimation, velocity variation, range estimation). Beyond 600 yards, spin drift becomes a component that precision shooters *must* account for.

The following table shows approximate spin drift values for common long-range cartridges, all with right-hand twist:

(Values are approximate inches of rightward deflection.)

Table 14.3: Spin drift at selected ranges for common cartridges

| Cartridge | Bullet / Twist | $S_g$ | 500 yd | 700 yd | 1 000 yd | 1 mi |
|---|---|---|---|---|---|---|
| .308 Win | 168 SMK, 1:10″ | 1.8 | 0.6″ | 1.8″ | 8.4″ | 40″ |
| 6.5 CM | 140 ELD-M, 1:8″ | 1.9 | 0.5″ | 1.5″ | 6.8″ | 30″ |
| .300 WM | 220 SMK, 1:10″ | 1.5 | 0.4″ | 1.2″ | 5.5″ | 26″ |
| .338 LM | 300 SMK, 1:10″ | 1.6 | 0.3″ | 0.9″ | 3.8″ | 18″ |

### 14.7.2 Spin Drift vs. Wind Drift

A useful rule of thumb: at 1 000 yards, the spin drift of a typical .308 load is roughly equivalent to the wind drift caused by a 1.5–2 mph crosswind. Since most shooters can estimate wind to within 2–3 mph at best, spin drift is significant but not dominant compared to wind uncertainty.

**The Practical Bottom Line**

- Below 600 yards: spin drift is usually inside your wind call uncertainty. Ignore it unless you are shooting groups for record.
- 600–1 000 yards: dial or hold for spin drift. It is predictable (unlike wind), so it is "free" accuracy once you know the value.
- Beyond 1 000 yards: spin drift is mandatory. At one mile, it can exceed 3 feet for some cartridges.

### 14.7.3 Comparing With and Without Spin Drift

To see the effect of spin drift on your specific load, run two trajectories and compare:

Listing 14.8: Trajectory without spin drift

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5
```

Listing 14.9: Same trajectory with spin drift

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.215 --sight-height 1.5
```

The difference in the "Drift" column at each range step is the spin drift contribution. This is a clean comparison because all other parameters (wind, Coriolis, etc.) are held constant (at zero, in this case).

### 14.7.4 High Altitude and Spin Drift

At higher altitudes, air density decreases. This has two competing effects on spin drift:

1. **Less aerodynamic damping** → the bullet retains velocity longer → shorter time of flight → *less* spin drift for a given range.
2. **Density correction factor** → the $\sqrt{\rho_0/\rho}$ term in the advanced model increases drift → *more* spin drift per unit time.

The net effect depends on the cartridge and range, but for most loads the reduced time of flight dominates, meaning spin drift is slightly *less* at altitude.

Listing 14.10: Comparing spin drift at altitude

```
# Sea level
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.215 --sight-height 1.5 \
  --altitude 0

# 5,000 ft elevation
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.215 --sight-height 1.5 \
  --altitude 5000
```

## Exercises

1. **Compute stability for your rifle.** Use the `stability` subcommand with your actual bullet's mass, diameter, and length. Is your $S_g$ in the ideal 1.4–2.0 range?

```
ballistics stability \
  --mass 175 --diameter 0.308 --length 1.240 \
  --twist-rate 10 --velocity 2600
```

2. **Compare twist rates.** Run the stability calculation for a 6.5 Creedmoor 140 gr bullet with 1:8″, 1:9″, and 1:10″ twist. At what twist rate does $S_g$ drop below 1.5?

```
ballistics stability \
  --mass 140 --diameter 0.264 --length 1.375 \
  --twist-rate 8 --velocity 2700

ballistics stability \
  --mass 140 --diameter 0.264 --length 1.375 \
  --twist-rate 9 --velocity 2700

ballistics stability \
  --mass 140 --diameter 0.264 --length 1.375 \
  --twist-rate 10 --velocity 2700
```

3. **Quantify spin drift at distance.** Run a trajectory to 1 200 yards with and without `--enable-spin-drift` for a .300 Win Mag, 190 gr SMK, `0.533` G1, 2 900 fps, 1:10″ twist, 1.350″ length. How many inches of spin drift does the model predict?

```
ballistics trajectory \
  --diameter 0.308 --mass 190 --bc 0.533 \
  --velocity 2900 --auto-zero 100 --max-range 1200 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.350 --sight-height 1.5
```

4. **Left-hand twist experiment.** Repeat the exercise above with `--twist-right` set to `false`. Verify that the drift reverses direction but maintains the same magnitude.

```
ballistics trajectory \
  --diameter 0.308 --mass 190 --bc 0.533 \
  --velocity 2900 --auto-zero 100 --max-range 1200 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.350 --sight-height 1.5
```

## What's Next

Spin drift is a predictable, deterministic effect that always pushes the bullet in the same direction. In the next chapter, we turn to another predictable-but-often-ignored effect: the deflection caused by the Earth's rotation. Chapter 15 introduces the Coriolis and Eötvös effects—the physics that make a northward-fired bullet drift to the right in the Northern Hemisphere and the vertical correction that accounts for the Earth spinning beneath the bullet's flight path.

# Chapter 15

# Coriolis & Eötvös Effects

Spin drift, as we saw in Chapter 14, deflects a bullet because of the bullet's own rotation. The effects in this chapter have nothing to do with the bullet—they arise because the *Earth itself* is rotating beneath the trajectory. A bullet that appears to travel in a perfectly straight line in an inertial (non-rotating) reference frame will appear to curve when observed from the surface of a spinning planet.

At short range, these effects are negligible. At 1 000 yards and beyond, they are on the order of several inches and are as predictable as gravity. In long-range competition and military sniping, Coriolis corrections are routinely dialed into the scope alongside windage and elevation.

This chapter explains the physics, shows how BALLISTICS-ENGINE computes the corrections, and provides practical guidance for when you should—and should not—bother including them.

## 15.1 Why the Earth's Rotation Matters

The Earth completes one rotation about its polar axis every 23 hours, 56 minutes, and 4 seconds (one sidereal day). This corresponds to an angular velocity of:

$$\Omega_E = 7.2921159 \times 10^{-5} \text{ rad/s} \tag{15.1}$$

This is a small number, and for most everyday physics it is safely ignored. But a rifle bullet at 2 700 fps covers a mile in about 2.2 seconds—during which the Earth's surface (at mid-latitudes) has moved roughly 0.7 inches laterally relative to an inertial frame. That 0.7 inches is not large, but it is *consistent, predictable, and cumulative*. At longer ranges and longer flight times, the effect grows rapidly.

Two distinct phenomena arise from the Earth's rotation:

1. **The Coriolis effect** — an apparent horizontal (and vertical) deflection of the bullet's path caused by the rotation of the Earth's reference frame.
2. **The Eötvös effect** — an apparent change in the bullet's *weight* (effective gravitational acceleration) caused by the centrifugal component of the Earth's rotation.

**Why "Apparent" Deflection?**

The Coriolis and Eötvös effects are not real forces acting on the bullet. They are *fictitious forces* that appear when you describe the trajectory in the rotating reference frame of the Earth's surface. In an inertial frame, the bullet flies straight—it is the *target* that moves. But since we aim and measure from the Earth's surface, these fictitious forces are the most convenient way to account for the discrepancy.

## 15.2 The Coriolis Effect: Horizontal Deflection

### 15.2.1 The Basic Physics

The Coriolis acceleration is given by:

$$\boxed{\vec{a}_{\text{Cor}} = -2\,\vec{\Omega}_E \times \vec{v}} \tag{15.2}$$

where $\vec{\Omega}_E$ is the Earth's angular velocity vector (pointing from the South Pole to the North Pole along the rotation axis) and $\vec{v}$ is the bullet's velocity vector. The cross product means that the Coriolis acceleration is always *perpendicular* to the velocity—it deflects the bullet sideways without changing its speed.

### 15.2.2 Projecting into the Shooter's Local Frame

The complication is that $\vec{\Omega}_E$ is aligned with the Earth's rotation axis, while the bullet's velocity is expressed in the shooter's local frame (lateral, vertical, downrange). The engine must project $\vec{\Omega}_E$ into the local frame using the shooter's latitude $\phi$ and shot azimuth $\theta$ (compass heading).

In `src/trajectory_solver.rs` (and equivalently in `src/fast_trajectory.rs`), the projection is:

$$\vec{\Omega}_{\text{local}} = \Omega_E \begin{pmatrix} \cos\phi \cdot \sin\theta \\ \sin\phi \\ \cos\phi \cdot \cos\theta \end{pmatrix} \tag{15.3}$$

where the coordinate system is:

- $x$ = lateral (positive right),
- $y$ = vertical (positive up),
- $z$ = downrange (positive away from the shooter).

Listing 15.1: Omega vector projection from `src/trajectory_solver.rs`

```
let earth_rotation_rate = 7.2921159e-5; // rad/s
let latitude_rad = inputs.latitude
    .unwrap_or(0.0).to_radians();
let azimuth = inputs.azimuth_angle; // already radians

Some(Vector3::new(
    earth_rotation_rate * latitude_rad.cos()
        * azimuth.sin(),
    earth_rotation_rate * latitude_rad.sin(),
    earth_rotation_rate * latitude_rad.cos()
        * azimuth.cos(),
))
```

### 15.2.3 Direction of Deflection

The direction of the Coriolis deflection depends on the hemisphere and shooting direction:

- **Northern Hemisphere**: A bullet deflects to the *right* of its direction of travel, regardless of the compass heading.
- **Southern Hemisphere**: A bullet deflects to the *left*.
- **Equator** ($\phi = 0$): The horizontal component vanishes, but a vertical component remains (the Eötvös effect).

**Right in the North, Left in the South**

A useful mnemonic: in the Northern Hemisphere, the Coriolis effect pushes everything to the right (just like hurricanes spin counter-clockwise). At 45°N latitude shooting due North, a .308 Win at 1 000 yards is deflected approximately 3–4 inches to the right.

### 15.2.4 Magnitude Estimate

For a bullet traveling primarily in the horizontal plane (small launch angle), the lateral Coriolis deflection can be approximated as:

$$\Delta x_{\text{Cor}} \approx \Omega_E \cdot \sin\phi \cdot V \cdot t^2 \tag{15.4}$$

where $V$ is the average downrange velocity and $t$ is the time of flight. This is the dominant term and corresponds to the $y$ component of $\vec{\Omega}_{\text{local}}$ (the vertical component of the Earth's rotation vector), which creates a horizontal deflection via the cross product with the downrange velocity.

For our reference .308 Win load (168 gr SMK, 2 700 fps) at 45°N:

$$\Delta x \approx 7.29 \times 10^{-5} \times \sin(45) \times 500 \text{ m/s} \times (1.55 \text{ s})^2 \approx 0.062 \text{ m} \approx 2.4 \text{ in}$$

This is a rough estimate; the full numerical integration in BALLISTICS-ENGINE accounts for the velocity decrease over the trajectory and the three-dimensional nature of the cross product.

## 15.3 The Eötvös Effect: Vertical Component

### 15.3.1 What Is the Eötvös Effect?

Named after the Hungarian physicist Loránd Eötvös, this effect describes the apparent change in gravitational acceleration experienced by an object moving on the surface of a rotating body. When you move *eastward*, you add to the Earth's rotational velocity, increasing the centrifugal force that partially cancels gravity—you become slightly lighter. When you move *westward*, you subtract from the rotational velocity and become slightly heavier.

For a bullet, this means:

- **Shooting East**: The bullet experiences reduced effective gravity → less drop → impacts *high*.
- **Shooting West**: The bullet experiences increased effective gravity → more drop → impacts *low*.
- **Shooting North or South**: No Eötvös effect (no change in the east-west velocity component).

### 15.3.2 The Physics

The Eötvös correction to gravitational acceleration is:

$$\Delta g = -2\,\Omega_E\,V_{\text{east}}\,\cos\phi - \frac{V_{\text{east}}^2 + V_{\text{north}}^2}{R_E} \tag{15.5}$$

where $V_{\text{east}}$ is the eastward component of velocity, $V_{\text{north}}$ is the northward component, and $R_E \approx 6{,}371$ km is the Earth's mean radius. For rifle bullets, the second term (centripetal acceleration from the bullet's own motion) is negligible. The first term is the dominant Eötvös correction.

### 15.3.3 How Large Is It?

For a bullet traveling due east at 800 m/s (≈2 625 fps) at 45°N latitude:

$$\Delta g \approx -2 \times 7.29 \times 10^{-5} \times 800 \times \cos(45) \approx -0.0824 \text{ m/s}^2$$

This is about 0.84% of standard gravity ($g = 9.807$ m/s$^2$). Over a 1.55-second flight to 1 000 yards, the change in drop is:

$$\Delta y \approx \frac{1}{2} \times 0.0824 \times 1.55^2 \approx 0.099 \text{ m} \approx 3.9 \text{ in}$$

Nearly 4 inches of vertical change at 1 000 yards—this is significant. Shooting west at the same range produces the opposite effect: 4 inches *more* drop than expected.

**Shot Direction Now Affects Coriolis (0.21.0)**

Through v0.20.0 the command-line solver ignored `--shot-direction`: the Coriolis correction was always computed as a due-north shot, so east- and west-fired trajectories were numerically identical and the Eötvös term above was absent. As of v0.21.0 `--shot-direction` is carried into the solver (internally as `shot_azimuth`, distinct from the aiming `azimuth_angle`) and the vertical effect is applied as derived here.

**East-West Shooting at Long Range**

The Eötvös effect can shift your point of impact by half a MOA or more at 1 000 yards when shooting roughly east or west. If you zero your rifle while shooting north and then engage a target to the east, your cold-bore shot may land several inches high—or low, if heading west.

### 15.3.4 How BALLISTICS-ENGINE Handles the Eötvös Effect

In the engine, the Eötvös effect is not computed as a separate correction. Instead, it emerges naturally from the full Coriolis acceleration $\vec{a}_{\text{Cor}} = -2\,\vec{\Omega} \times \vec{v}$. The vertical component of this cross product *is* the Eötvös effect. When the bullet has an eastward velocity component, the cross product produces an upward acceleration (reducing effective gravity); when moving westward, it produces a downward acceleration.

This is one of the elegant features of the vector cross-product approach: both the horizontal Coriolis deflection and the vertical Eötvös correction are computed simultaneously in a single operation.

## 15.4 Latitude and Heading Dependence

### 15.4.1 Latitude Dependence

The magnitude of the Coriolis effect depends strongly on latitude:

- At the **poles** ($\phi = \pm 90$): The horizontal component is at its maximum ($\sin\phi = 1$). The Eötvös component is zero ($\cos\phi = 0$).
- At the **equator** ($\phi = 0$): The horizontal component is zero ($\sin\phi = 0$). The Eötvös component is at its maximum ($\cos\phi = 1$).
- At **mid-latitudes** (30°–60°): Both components are significant.

Table 15.1: Coriolis horizontal deflection at 1 000 yd for .308 Win, 168 gr SMK, 2 700 fps, shooting due North

| Location (Latitude) | Deflection (in) | Deflection (MOA) |
|---|---|---|
| Equator (0°) | 0.0 | 0.00 |
| Houston, TX (30°N) | 1.7 | 0.16 |
| Denver, CO (40°N) | 2.2 | 0.21 |
| Seattle, WA (48°N) | 2.5 | 0.24 |
| Anchorage, AK (61°N) | 3.0 | 0.29 |
| North Pole (90°N) | 3.4 | 0.33 |
| Sydney, AU (34°S) | −1.9 | −0.18 |

(Negative values indicate leftward deflection in the Southern Hemisphere.)

### 15.4.2 Heading Dependence

While the horizontal Coriolis deflection is roughly the same magnitude regardless of compass heading (it always pushes right in the Northern Hemisphere), the *vertical* component—the Eötvös effect—depends strongly on whether you are shooting east, west, or in a cardinal direction:

**The Combined Picture**

At 45°N, shooting due east at 1 000 yards, your bullet is deflected approximately 3 inches right (Coriolis) *and* 3.5 inches high (Eötvös). The total correction is a diagonal shift of roughly 4.6 inches at about 50 degrees above the horizontal axis. Both corrections must be applied simultaneously.

Table 15.2: Eötvös vertical correction at 1 000 yd by heading (45°N, .308 Win 168 gr SMK)

| Heading | Vertical Shift (in) | Direction |
|---|---|---|
| North (0°) | ∼0 | — |
| East (90°) | +3.5 | Impact high |
| South (180°) | ∼0 | — |
| West (270°) | −3.5 | Impact low |
| NE (45°) | +2.5 | Impact high |
| SW (225°) | −2.5 | Impact low |

## 15.5 Computing Coriolis Corrections in `ballistics-engine`

### 15.5.1 Enabling the Coriolis Effect

To include Coriolis corrections, you need:

1. The `--enable-coriolis` flag.
2. The `--latitude` flag (in degrees, −90 to +90).
3. Optionally, `--shot-direction` (compass heading in degrees: 0 = North, 90 = East). If omitted, the engine defaults to a 0° azimuth (due North).

Listing 15.2: Trajectory with Coriolis enabled

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude 45 \
  --shot-direction 90
```

This computes the trajectory for a shooter at 45°N latitude firing due east. The output will reflect both the horizontal Coriolis deflection and the vertical Eötvös correction automatically.

**Coriolis Enables Advanced Effects**

Setting `--enable-coriolis` also activates the `enable_advanced_effects` flag internally. This enables the omega vector calculation and passes it through to the `compute_derivatives()` function in `src/derivatives.rs`, where the cross product is computed at every integration step.

### 15.5.2 How the Engine Computes It

The Coriolis correction is applied at every time step of the numerical integration. In `compute_derivatives()` (in `src/derivatives.rs`), the implementation is remarkably concise:

Listing 15.3: Coriolis acceleration in `src/derivatives.rs`

```
// Add Coriolis acceleration if omega vector is provided
if let Some(omega) = omega_vector {
    let accel_coriolis = -2.0 * omega.cross(&vel);
    accel += accel_coriolis;
}
```

The `omega_vector` is pre-computed from the latitude and shot direction when the trajectory solver initialises. It remains constant throughout the flight (the Earth's rotation does not change perceptibly during a bullet's 1–3 second flight time).

### 15.5.3 Combining Coriolis with Other Effects

The Coriolis acceleration is simply added to the total acceleration vector alongside gravity, drag, Magnus force, and spin drift corrections. The RK4 or RK45 integrator then propagates all effects simultaneously through the trajectory:

Listing 15.4: Acceleration accumulation in `src/derivatives.rs`

```
// Total acceleration
let mut accel = accel_gravity + accel_drag + accel_magnus;

// Add Coriolis acceleration
if let Some(omega) = omega_vector {
    let accel_coriolis = -2.0 * omega.cross(&vel);
    accel += accel_coriolis;
}
```

This vector approach means that interactions between effects are handled naturally. For example, the Coriolis acceleration changes the velocity vector at each step, which in turn changes the drag force (since drag depends on velocity relative to the air), which feeds back into subsequent Coriolis calculations. These higher-order interactions are small but are captured automatically by the numerical integration.

### 15.5.4 Southern Hemisphere and Negative Latitudes

To model Southern Hemisphere shooting, simply provide a negative latitude:

Listing 15.5: Coriolis correction in Australia

```
# Shooting from Adelaide, South Australia (35S)
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude -35 \
  --shot-direction 0
```

The deflection will be to the *left* instead of the right, with the same magnitude as a 35°N location.

### 15.5.5 Viewing the Effect in Isolation

To see how much Coriolis contributes to the total deflection, run two trajectories—one with and one without the flag—and compare:

Listing 15.6: Isolating the Coriolis contribution

```
# Without Coriolis
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5

# With Coriolis (45N, shooting East)
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude 45 \
  --shot-direction 90
```

The difference in the lateral "Drift" column is the Coriolis horizontal deflection; the difference in the "Drop" column (if any) is the Eötvös vertical correction.

## 15.6 Practical Significance: When to Include Coriolis

### 15.6.1 The Range Threshold

The general guidance:

- **Under 600 yards**: Coriolis is less than 0.1 MOA for any common cartridge at any latitude. For practical purposes, it is negligible. Do not waste time computing it.

Table 15.3: Coriolis deflection as a fraction of 1 MOA at selected ranges (45°N)

| Cartridge | 300 yd | 600 yd | 1 000 yd | 1 mile |
|---|---|---|---|---|
| .308 Win 168 gr | 0.03 MOA | 0.10 MOA | 0.24 MOA | 0.9 MOA |
| 6.5 CM 140 gr | 0.02 MOA | 0.08 MOA | 0.20 MOA | 0.7 MOA |
| .338 LM 300 gr | 0.01 MOA | 0.05 MOA | 0.13 MOA | 0.5 MOA |

- **600–1 000 yards**: Coriolis reaches 0.15–0.25 MOA. In precision rifle competition, where targets are scored to 0.1 MOA, this matters. Include it if you are dialing precise corrections.
- **Beyond 1 000 yards**: Coriolis becomes a significant fraction of a MOA—and at one mile, nearly a full MOA for slow cartridges. It is mandatory for extreme long-range (ELR) shooting.

### 15.6.2 Coriolis vs. Wind Uncertainty

At 1 000 yards, the Coriolis deflection of 2–3 inches is equivalent to a 1-mph crosswind. Since wind estimation errors are typically 2–5 mph at that range, Coriolis is a smaller effect than wind uncertainty but a *perfectly predictable* one. Unlike wind, you can compute the Coriolis correction exactly (assuming you know your latitude and heading). It costs nothing to include it.

### 15.6.3 When Heading Changes Matter

If you are shooting from a fixed position at multiple targets at different compass headings, the Coriolis horizontal deflection does not change much—it is always roughly "right" in the Northern Hemisphere. But the Eötvös vertical correction *does* change: targets to the east impact high; targets to the west impact low. For a competition stage where you engage targets in multiple directions, the vertical shift can be the difference between an X-ring hit and a miss.

Listing 15.7: Comparing headings at the same location

```
# Shooting North at 1000 yd (45N latitude)
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude 45 \
  --shot-direction 0

# Shooting East at 1000 yd (same position)
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
```

```
  --enable-coriolis --latitude 45 \
  --shot-direction 90

# Shooting West at 1000 yd (same position)
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude 45 \
  --shot-direction 270
```

### 15.6.4 Combining Coriolis and Spin Drift

Both Coriolis and spin drift produce horizontal deflections that are predictable and direction-dependent. In the Northern Hemisphere with a right-hand twist barrel shooting due north:

- Spin drift: deflection *right*.
- Coriolis: deflection *right*.
- The two effects *add together*.

When shooting due south with a right-twist barrel:

- Spin drift: still *right* (depends on twist direction, not heading).
- Coriolis: still *right* (in Northern Hemisphere, deflection is always to the right of the direction of travel).
- Again, they add.

In the Southern Hemisphere with a right-hand twist barrel:

- Spin drift: *right*.
- Coriolis: *left*.
- The two effects *partially cancel*.

This partial cancellation is one reason why Southern Hemisphere shooters sometimes notice smaller total lateral drift than their Northern Hemisphere counterparts with the same equipment.

Listing 15.8: Combined spin drift and Coriolis

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
```

```
  --bullet-length 1.215 --sight-height 1.5 \
  --enable-coriolis --latitude 45 \
  --shot-direction 0
```

### 15.6.5 ELR and One-Mile Shooting

At extreme long range (ELR)—typically beyond 1 500 yards—Coriolis is no longer optional. At one mile (1 760 yards), the horizontal deflection alone can exceed 10 inches, and the Eötvös vertical shift (when shooting east or west) can approach a foot. ELR shooters routinely include latitude, heading, and even shot direction updates between stages in their ballistic calculations.

Listing 15.9: ELR trajectory with full environmental corrections

```
ballistics trajectory \
  --diameter 0.338 --mass 300 --bc 0.768 \
  --drag-model g7 \
  --velocity 2750 --auto-zero 100 --max-range 1760 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.700 --sight-height 1.5 \
  --enable-coriolis --latitude 38 \
  --shot-direction 45 \
  --altitude 4500 --temperature 72 \
  --wind-speed 8 --wind-direction 90
```

## Exercises

1. **Hemisphere comparison.** Run trajectories at 1 000 yards for 45°N and 45°S with the same load and heading (North). Compare the drift columns. Verify that the horizontal deflection reverses sign.

   ```
   # Northern Hemisphere
   ballistics trajectory \
     --diameter 0.308 --mass 168 --bc 0.462 \
     --velocity 2700 --auto-zero 100 --max-range 1000 \
     --sight-height 1.5 \
     --enable-coriolis --latitude 45 \
     --shot-direction 0

   # Southern Hemisphere
   ballistics trajectory \
     --diameter 0.308 --mass 168 --bc 0.462 \
     --velocity 2700 --auto-zero 100 --max-range 1000 \
   ```

```
  --sight-height 1.5 \
  --enable-coriolis --latitude -45 \
  --shot-direction 0
```

2. **Eötvös effect.** Compute trajectories shooting due east and due west from Denver, CO (40°N). Compare the drop at 1 000 yards. How many inches of vertical difference does the Eötvös effect produce?

```
# Shooting East from Denver
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 --altitude 5280 \
  --enable-coriolis --latitude 40 \
  --shot-direction 90

# Shooting West from Denver
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 --altitude 5280 \
  --enable-coriolis --latitude 40 \
  --shot-direction 270
```

3. **Equator test.** Run a trajectory at the equator ($\phi = 0$) with `--enable-coriolis`. Verify that the horizontal Coriolis deflection is zero (or negligible) while a vertical shift is present when shooting east.

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1000 \
  --sight-height 1.5 \
  --enable-coriolis --latitude 0 \
  --shot-direction 90
```

4. **One-mile ELR.** Using a .338 Lapua Magnum (300 gr SMK, 0.768 G7, 2 750 fps, 1:10″ twist), compute a trajectory to 1 760 yards with full Coriolis and spin drift enabled at your home latitude. How many inches of total horizontal deflection does the engine predict?

```
ballistics trajectory \
  --diameter 0.338 --mass 300 --bc 0.768 \
  --drag-model g7 \
```

```
--velocity 2750 --auto-zero 100 --max-range 1760 \
--twist-rate 10 --enable-spin-drift \
--bullet-length 1.700 --sight-height 1.5 \
--enable-coriolis --latitude 45 \
--shot-direction 0
```

## What's Next

Coriolis and Eötvös are the last of the "translation-only" effects. Every phenomenon we have discussed so far—drag, wind, spin drift, and Earth rotation—deflects the bullet's *centre of mass* without worrying about the bullet's orientation. In the next chapter, we abandon that simplification. Chapter 16 introduces *precession and nutation*—the angular wobbles of the bullet itself —and explains how these 4DOF (four-degree-of-freedom) effects produce aerodynamic jump and pitch damping.

# Chapter 16

# Precession & Nutation

Every chapter so far has treated the bullet as a point mass—a dimensionless particle whose only properties are position, velocity, mass, and ballistic coefficient. That approximation is remarkably effective for most practical ballistics. But a rifle bullet is not a point: it is an elongated, spinning body with a definite orientation, and that orientation oscillates in ways that affect the trajectory.

In this chapter, we leave the point-mass (3DOF) world and enter the domain of angular motion. We will study *precession*—the slow, sweeping cone traced by the bullet's nose—and *nutation*—the fast, small-amplitude wobble superimposed on it. Together, they produce the characteristic *epicyclic motion* that defines a bullet's angular state. We will also cover *aerodynamic jump* (the displacement caused by the transition from barrel constraint to free flight) and *pitch damping* (the aerodynamic mechanism that quiets these oscillations).

These are 4DOF (four degrees of freedom) extensions: three translational (point-mass) plus one rotational. They matter most for marginally stable bullets, transonic flights, and precision shooting where every fraction of a MOA counts.

## 16.1 Gyroscopic Precession: The Slow Wobble

### 16.1.1 Why Bullets Precess

A spinning bullet is a gyroscope, and gyroscopes obey a counter-intuitive law: when you apply a torque perpendicular to the spin axis, the axis rotates not in the direction of the torque but at 90° to it. This is *gyroscopic precession*.

For a bullet, the torque is the aerodynamic overturning moment. Any small angle between the bullet's spin axis and the velocity vector (called the *angle of attack* or *yaw angle*) exposes the bullet's

side to the airstream, creating a pitching moment that tries to tumble it. The gyroscopic response converts this pitching moment into a slow, steady rotation of the spin axis *around* the velocity vector.

**Precession (Slow Mode)**

The slow, steady coning motion of the bullet's spin axis around the flight-path velocity vector. The precession rate depends on the spin rate, the moments of inertia, the velocity, and the yaw angle. Precession is the mechanism that produces the yaw of repose and therefore spin drift.

### 16.1.2 Precession Frequency

The precession frequency (angular rate of the coning motion) is given by:

$$\omega_p = \frac{I_s \cdot \omega_s \cdot \sin \alpha}{I_t \cdot V} \tag{16.1}$$

where:

- $I_s$ is the axial (spin) moment of inertia,
- $\omega_s$ is the spin rate (rad/s),
- $\alpha$ is the yaw angle,
- $I_t$ is the transverse moment of inertia,
- $V$ is the velocity.

This is implemented in `calculate_precession_frequency()` in `src/precession_nutation.rs`:

Listing 16.1: Precession frequency calculation

```
pub fn calculate_precession_frequency(
    spin_rate_rad_s: f64,
    velocity_mps: f64,
    spin_inertia: f64,
    transverse_inertia: f64,
    yaw_angle_rad: f64,
) -> f64 {
    if velocity_mps == 0.0 || transverse_inertia == 0.0 {
        return 0.0;
    }
    (spin_inertia * spin_rate_rad_s * yaw_angle_rad.sin())
        / (transverse_inertia * velocity_mps)
}
```

For a typical .308 match bullet (175 gr, 1:10″ twist, $S_g \approx 1.8$):

- $I_s \approx 6.94 \times 10^{-8}\,\text{kg}\,\text{m}^2$
- $I_t \approx 9.13 \times 10^{-7}\,\text{kg}\,\text{m}^2$
- $\omega_s \approx 17{,}522$ rad/s
- At $\alpha = 0.002$ rad and $V = 850$ m/s: $\omega_p \approx 0.003$ rad/s

The precession rate is extremely slow—one full cone every $\sim$35 minutes of flight. In the 1–2 seconds of a typical trajectory, the bullet completes only a tiny fraction of a precession cycle. This is why precession itself does not directly cause large trajectory deflections; instead, it establishes the *steady-state yaw of repose* that drives spin drift.

## 16.2 Nutation: The Fast Wobble

### 16.2.1 What Is Nutation?

Nutation is the fast, small-amplitude oscillation superimposed on the slower precession. If precession traces a smooth cone, nutation adds a rapid "scallop" to the cone's surface—the nose darts back and forth across the precession circle at a much higher frequency.

Physically, nutation arises because the bullet's angular momentum does not align perfectly with the velocity vector at the moment of muzzle exit. Any initial disturbance (barrel crown imperfection, muzzle gas asymmetry, the bullet's jump to the rifling) creates a fast oscillation that the gyroscopic effect partially suppresses but cannot eliminate instantly.

**Nutation (Fast Mode)**

The high-frequency oscillation of the spin axis about the precession cone. Nutation decays exponentially due to aerodynamic damping and is usually negligible beyond 50–100 calibres of flight for a stable bullet. Its frequency is much higher than precession—typically in the kilohertz range.

### 16.2.2 Nutation Frequency

The nutation frequency is related to the spin rate and the stability factor:

$$\omega_n = \omega_s \cdot \sqrt{\frac{I_s}{I_t}} \cdot \sqrt{S_g - 1} \tag{16.2}$$

In `src/precession_nutation.rs`:

Listing 16.2: Nutation frequency calculation

```
pub fn calculate_nutation_frequency(
```

```
    spin_rate_rad_s: f64,
    spin_inertia: f64,
    transverse_inertia: f64,
    stability_factor: f64,
) -> f64 {
    if stability_factor <= 1.0 || transverse_inertia == 0.0 {
        return 0.0;
    }
    let inertia_ratio = spin_inertia / transverse_inertia;
    spin_rate_rad_s * inertia_ratio.sqrt()
        * (stability_factor - 1.0).sqrt()
}
```

For our reference bullet: $\omega_n \approx 17{,}522 \times \sqrt{6.94 \times 10^{-8} / 9.13 \times 10^{-7}} \times \sqrt{1.8 - 1.0} \approx 4{,}200$ rad/s $\approx 669$ Hz.

This is *far* faster than precession. The nutation oscillation completes hundreds of cycles per second, while the bullet takes minutes to complete a single precession cycle. This separation of time scales is fundamental to the stability of the bullet: the fast nutation damps quickly, leaving the slow precession—and the yaw of repose—as the dominant angular motion.

### 16.2.3 Nutation Damping

The nutation amplitude decays exponentially:

$$A(t) = A_0 \cdot e^{-\zeta \omega_n t} \tag{16.3}$$

where $A_0$ is the initial disturbance amplitude and $\zeta$ is the damping factor (typically 0.05–0.10 for most bullets). The `calculate_nutation_amplitude()` function implements this decay with a maximum clamp of 0.1 rad (≈5.7°) to prevent unrealistic amplitudes:

Listing 16.3: Nutation amplitude with exponential damping

```
pub fn calculate_nutation_amplitude(
    initial_disturbance_rad: f64,
    time_s: f64,
    nutation_frequency: f64,
    damping_factor: f64,
    spin_rate_rad_s: f64,
) -> f64 {
    if nutation_frequency == 0.0
       || spin_rate_rad_s == 0.0 { return 0.0; }
```

```
    let damping_rate = damping_factor * nutation_frequency;
    let amplitude = initial_disturbance_rad
        * (-damping_rate * time_s).exp();

    amplitude.min(0.1) // Max 0.1 rad
}
```

For our reference bullet with $\zeta = 0.05$ and $\omega_n = 4{,}200$ rad/s, the initial disturbance halves in:

$$t_{1/2} = \frac{\ln 2}{\zeta \omega_n} = \frac{0.693}{0.05 \times 4200} \approx 3.3 \times 10^{-3} \text{ s} \approx 3.3 \text{ ms}$$

The nutation is essentially gone within 10–20 milliseconds of muzzle exit—roughly 25–50 feet of flight. This is why nutation rarely affects the point of impact directly, but it does affect the *initial conditions* for precession and spin drift.

## 16.3 Epicyclic Motion: The Combined Pattern

### 16.3.1 Combining Precession and Nutation

The total angular motion of the bullet is the superposition of the slow precession (slow mode) and the fast nutation (fast mode). When plotted in a coordinate system of pitch vs. yaw, the bullet's nose traces a pattern called an *epicycle*—a small circle (nutation) riding on a large circle (precession).

**Epicyclic Motion**

The combined precession and nutation motion traced by the bullet's spin axis, consisting of a slow coning motion (precession) with a fast oscillation (nutation) superimposed. As aerodynamic damping eliminates the nutation, the epicycle collapses into a smooth precession cone.

The `calculate_epicyclic_motion()` function in `src/precession_nutation.rs` computes both modes simultaneously:

Listing 16.4: Epicyclic motion computation

```
pub fn calculate_epicyclic_motion(
    spin_rate_rad_s: f64,
    velocity_mps: f64,
    stability_factor: f64,
    time_s: f64,
    initial_yaw_rad: f64,
) -> (f64, f64) {
```

```
    if stability_factor <= 1.0
       || spin_rate_rad_s == 0.0 {
        return (initial_yaw_rad, initial_yaw_rad);
    }

    // Slow mode (precession)
    let omega_slow = 2.0 * velocity_mps
        / (stability_factor * spin_rate_rad_s);

    // Fast mode (nutation)
    let omega_fast = spin_rate_rad_s
        * ((stability_factor - 1.0).sqrt())
        / stability_factor;

    // Amplitude ratio (fast/slow)
    let amplitude_ratio = 1.0 / stability_factor;

    // Damping of fast mode
    let damping_factor = 0.1;
    let fast_amplitude = amplitude_ratio
        * (-damping_factor * omega_fast * time_s).exp();

    // Combined motion
    let slow_phase = omega_slow * time_s;
    let fast_phase = omega_fast * time_s;

    let yaw = initial_yaw_rad
        * (slow_phase.cos()
           + fast_amplitude * fast_phase.cos());
    let pitch = initial_yaw_rad
        * (slow_phase.sin()
           + fast_amplitude * fast_phase.sin());

    (pitch, yaw)
}
```

The key insight from this function is the *amplitude ratio*: the nutation amplitude is inversely proportional to the stability factor $S_g$. A bullet with $S_g = 2.0$ has nutation amplitude equal to half the precession amplitude; a bullet with $S_g = 3.0$ has only a third. Over-stabilised bullets ($S_g > 3$) have nearly undetectable nutation.

### 16.3.2 Visualising the Epicycle

Imagine looking at the bullet from behind (along the velocity vector). The nose traces a pattern in the pitch-yaw plane:

1. **At muzzle exit**: A large, ragged circle (nutation is at maximum amplitude).
2. **After 20–50 ms**: The fast scallops damp out; the motion smooths into a cleaner circle.
3. **After 100 ms**: The circle shrinks and shifts off-centre as the yaw of repose establishes. The nose is now pointing slightly to one side—the direction determined by the twist and gravity.
4. **Steady state**: A small, nearly circular precession cone centred on the yaw of repose. This is the configuration that produces spin drift.

## 16.4 Aerodynamic Jump

### 16.4.1 What Is Aerodynamic Jump?

Aerodynamic jump is the angular displacement of the bullet's trajectory that occurs as the bullet transitions from constrained motion in the barrel to free flight in the atmosphere. It is *not* caused by the bullet physically "jumping" away from the muzzle; rather, it is the angular change in the bullet's velocity vector caused by the transient aerodynamic forces during the first few milliseconds after the bullet clears the crown.

**Aerodynamic Jump**

The initial angular displacement of the bullet's mean trajectory from the bore axis, caused by the Magnus force and initial yaw during the transition from in-bore to free-flight conditions. Measured in milliradians or MOA, aerodynamic jump is a one-time displacement that persists as a constant angular offset throughout the trajectory.

### 16.4.2 Causes

Several factors contribute to aerodynamic jump:

1. **Initial yaw**: The bullet exits the muzzle with a small but non-zero yaw angle due to imperfect barrel-bullet fit, muzzle crown asymmetries, and gas leakage past the bullet base.
2. **Crosswind at the muzzle**: A crosswind creates an immediate angle of attack that interacts with the Magnus force.
3. **Magnus force**: The spinning bullet with a yaw angle experiences a Magnus force perpendicular to the velocity vector, which deflects the trajectory during the critical stabilisation period.

4. **Barrel exit dynamics**: As the bullet leaves the barrel, the propellant gases escape asymmetrically, creating a brief but intense force on the bullet base.

### 16.4.3 Computing Aerodynamic Jump

The `calculate_aerodynamic_jump()` function in `src/aerodynamic_jump.rs` models the jump components and returns an `AerodynamicJumpComponents` struct:

Listing 16.5: Aerodynamic jump components (from `src/aerodynamic_jump.rs`)

```
pub struct AerodynamicJumpComponents {
    pub vertical_jump_moa: f64,    // at 100 yards
    pub horizontal_jump_moa: f64,  // at 100 yards
    pub jump_angle_rad: f64,       // total angular displacement
    pub magnus_component_moa: f64, // Magnus contribution
    pub yaw_component_moa: f64,    // Initial yaw contribution
    pub stabilization_factor: f64, // How quickly bullet stabilizes
}
```

The calculation involves several steps:

1. **Spin parameter**: The non-dimensional ratio $p = \omega_s d/(2V)$, representing how fast the bullet surface moves relative to the free-stream velocity.
2. **Effective yaw**: The sum of crosswind-induced yaw and any initial tipoff yaw from muzzle exit.
3. **Magnus force**: Using the same Mach-dependent coefficient ($C_{\text{mag}}$) described in Section 14.5, the Magnus force during barrel exit is computed.
4. **Stabilisation time**: The time for the bullet to travel through the stabilisation zone, where the nutation damps and the bullet settles into its steady-state angular configuration.
5. **Jump displacement**: The lateral displacement from the Magnus acceleration integrated over the stabilisation time.

### 16.4.4 Jump Direction and Twist

Aerodynamic jump direction depends on both the twist direction and the crosswind:

- **Right twist + right crosswind** $\rightarrow$ vertical jump *upward*.
- **Right twist + left crosswind** $\rightarrow$ vertical jump *downward*.
- **Left twist** $\rightarrow$ reverses the vertical direction.

This means that a crosswind not only pushes the bullet horizontally (wind drift) but also shifts it *vertically* through aerodynamic jump. This cross-coupling between horizontal wind and vertical impact is counterintuitive and often surprises shooters.

**Crosswind Causes Vertical Shift**

A 10-mph crosswind can cause a vertical point-of-impact shift of 0.2–0.5 MOA through aerodynamic jump. This is separate from and in addition to the horizontal wind drift. If you change firing positions and the crosswind changes, your vertical zero may shift even if the headwind or tailwind component stays the same.

### 16.4.5 Crosswind Jump Sensitivity

The `calculate_crosswind_jump_sensitivity()` function computes how many MOA of vertical shift a 1-mph crosswind produces:

Listing 16.6: Measuring aerodynamic jump sensitivity

```
# The aerodynamic jump calculation is integrated
# into the trajectory when precession is enabled.
# To see the effect, compare with and without crosswind:
ballistics trajectory \
  --diameter 0.308 --mass 175 --bc 0.505 \
  --velocity 2600 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --sight-height 1.5 \
  --enable-precession --enable-spin-drift \
  --bullet-length 1.240 \
  --wind-speed 10 --wind-direction 90
```

## 16.5 Pitch Damping: How Oscillations Decay

### 16.5.1 The Role of Pitch Damping

Pitch damping is the aerodynamic mechanism that causes the bullet's angular oscillations (both precession amplitude and nutation) to decay over time. Without pitch damping, a perturbed bullet would oscillate indefinitely at its natural frequencies; with it, the oscillations gradually decrease, and the bullet converges toward a steady-state orientation.

The pitch damping moment opposes the angular rate—just as aerodynamic drag opposes translational velocity. The faster the bullet pitches or yaws, the stronger the restoring moment.

### 16.5.2 Pitch Damping Coefficient

The aggregate pitch damping coefficient is often written as $C_{m_q} + C_{m_{\dot{\alpha}}}$, combining the rate-dependent and angle-of-attack-rate-dependent contributions. In `src/pitch_damping.rs`, this is modelled with four Mach regimes:

Listing 16.7: Pitch damping coefficients by Mach regime

```
pub struct PitchDampingCoefficients {
    pub subsonic: f64,       // M < 0.8
    pub transonic_low: f64,  // 0.8 <= M < 1.0
    pub transonic_high: f64, // 1.0 <= M < 1.2
    pub supersonic: f64,     // M >= 1.2
}
```

The critical feature of this model is that the coefficient can become *positive* in the upper transonic regime (Mach 1.0–1.2), meaning that pitch damping becomes *destabilising*. This is the primary mechanism behind transonic instability, discussed in detail in Chapter 17.

Table 16.1: Pitch damping coefficients by bullet type and Mach regime

| Bullet Type | Subsonic | Trans. Low | Trans. High | Supersonic |
|---|---|---|---|---|
| Match boat tail | $-0.9$ | $-0.4$ | $+0.1$ | $-0.6$ |
| Match flat base | $-0.7$ | $-0.2$ | $+0.3$ | $-0.4$ |
| VLD | $-1.0$ | $-0.5$ | $-0.1$ | $-0.7$ |
| Hunting | $-0.6$ | $-0.1$ | $+0.4$ | $-0.3$ |
| FMJ | $-0.7$ | $-0.2$ | $+0.2$ | $-0.5$ |

Negative values are stabilising (they damp oscillations); positive values are destabilising (they amplify oscillations). Notice that VLD bullets remain weakly stabilising even in the upper transonic, while hunting bullets can be strongly destabilising ($+0.4$) in that regime.

### 16.5.3 The Pitch Damping Moment

The aerodynamic moment is:

$$M_{\text{damp}} = q \cdot S \cdot d \cdot C_{m_q} \cdot \hat{q} \tag{16.4}$$

where $q = \frac{1}{2}\rho V^2$ is the dynamic pressure, $S = \pi(d/2)^2$ is the reference area, $d$ is the calibre, and $\hat{q} = \dot{\alpha} \cdot d/V$ is the non-dimensional pitch rate. The function `calculate_pitch_damping_moment()` in `src/pitch_damping.rs` implements this:

Listing 16.8: Pitch damping moment calculation

```
pub fn calculate_pitch_damping_moment(
    pitch_rate_rad_s: f64,
    velocity_mps: f64,
    air_density_kg_m3: f64,
    caliber_m: f64,
    _length_m: f64,
    mach: f64,
    coeffs: &PitchDampingCoefficients,
) -> f64 {
    if velocity_mps == 0.0
       || pitch_rate_rad_s == 0.0 { return 0.0; }

    let cmq = calculate_pitch_damping_coefficient(
        mach, coeffs);
    let q = 0.5 * air_density_kg_m3
        * velocity_mps.powi(2);
    let s = PI * (caliber_m / 2.0).powi(2);
    let d = caliber_m;
    let q_nondim = pitch_rate_rad_s * d / velocity_mps;

    q * s * d * cmq * q_nondim
}
```

### 16.5.4 How Pitch Damping Affects Spin Drift

Pitch damping enters the spin drift calculation through the *damped yaw of repose* (`calculate_damped_yaw_of_repose()` in `src/pitch_damping.rs`). When pitch damping is enabled, the equilibrium yaw angle accounts for the Mach-dependent damping coefficient:

- In the **supersonic** regime, negative (stabilising) damping reduces the equilibrium yaw, producing *less* spin drift than the simple model.
- In the **transonic** regime, positive (destabilising) damping *increases* the equilibrium yaw, amplifying spin drift precisely when the bullet is most vulnerable.

This coupling between pitch damping and spin drift is one of the reasons why bullets can show erratic lateral behaviour as they decelerate through the transonic zone.

## 16.6 How `ballistics-engine` Models Precession and Nutation

### 16.6.1 Enabling the Angular Motion Model

To activate the precession and nutation model, use the `--enable-precession` flag. For pitch damping (which refines the yaw of repose and damping rates), use `--enable-pitch-damping`:

Listing 16.9: Enabling precession and pitch damping

```
ballistics trajectory \
  --diameter 0.308 --mass 175 --bc 0.505 \
  --velocity 2600 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --twist-right \
  --enable-spin-drift --enable-precession \
  --enable-pitch-damping \
  --bullet-length 1.240 \
  --sight-height 1.5
```

**Computational Cost**

The precession and nutation model adds angular state tracking at every integration step. This increases computation time compared to the point-mass model. For most practical uses, the trajectory results are indistinguishable—reserve `--enable-precession` for analysis work and situations where you need to understand the bullet's angular behaviour.

### 16.6.2 The Angular State

Internally, the engine tracks an `AngularState` struct (from `src/precession_nutation.rs`) with six components:

Listing 16.10: Angular state representation

```
pub struct AngularState {
    pub pitch_angle: f64,      // rad
    pub yaw_angle: f64,        // rad
    pub pitch_rate: f64,       // rad/s
    pub yaw_rate: f64,         // rad/s
    pub precession_angle: f64, // cumulative rad
    pub nutation_phase: f64,   // phase angle rad
}
```

At each time step, the `calculate_combined_angular_motion()` function updates this state using:

1. The precession frequency (from the current yaw and spin rate).

2. The nutation frequency (from the stability factor).

3. The nutation amplitude (exponentially decaying from initial disturbance).

4. The pitch damping moment (Mach-dependent, potentially destabilising in the transonic).

The angular state feeds back into the trajectory computation through the yaw angle, which affects drag (a yawed bullet presents a larger cross-section to the airstream) and spin drift (the yaw of repose is modified by the angular dynamics).

### 16.6.3 The Precession-Nutation Parameters

The `PrecessionNutationParams` struct carries all the physical properties needed for the angular calculation:

Listing 16.11: Precession/nutation parameter defaults

```
impl Default for PrecessionNutationParams {
    fn default() -> Self {
        Self {
            mass_kg: 0.01134,     // 175 grains
            caliber_m: 0.00782,   // .308"
            length_m: 0.033,      // 1.3"
            spin_rate_rad_s: 17522.0,
            spin_inertia: 6.94e-8,
            transverse_inertia: 9.13e-7,
            velocity_mps: 850.0,
            air_density_kg_m3: 1.225,
            mach: 2.48,
            pitch_damping_coeff: -0.8,
            nutation_damping_factor: 0.05,
        }
    }
}
```

These defaults model a 175 gr .308 match bullet at supersonic velocity. The moments of inertia are calculated from the bullet's geometry using models appropriate for ogive or boat-tail shapes (see `calculate_transverse_moment_of_inertia()` in `src/pitch_damping.rs` and `calculate_moment_of_inertia()` in `src/spin_decay.rs`).

## 16.7 When 4DOF Effects Change the Answer

### 16.7.1 When They Matter

For most practical shooting scenarios, the point-mass model with the Litz spin drift formula is sufficient. The angular motion model provides measurably different results in a few specific cases:

1. **Marginally stable bullets** ($S_g$ between 1.0 and 1.4): The nutation damping is slow, so the initial angular disturbance persists longer and affects the early trajectory.
2. **Transonic flight**: When the bullet decelerates through Mach 1.0–1.2, the pitch damping coefficient can become positive, causing the angular oscillations to *grow* rather than decay. This is the primary mechanism behind "transonic instability," and it is only captured by the angular motion model.
3. **Crosswind-induced aerodynamic jump**: The vertical shift caused by a crosswind at the muzzle is a 4DOF effect that the point-mass model does not capture.
4. **Long-range subsonic flight**: After the bullet goes subsonic, the changed aerodynamic environment can alter the angular dynamics. Flat-base bullets are particularly affected because their pitch damping coefficients are less favourable.
5. **Precision analysis**: When computing trajectories for Monte Carlo dispersion studies, the angular uncertainty at muzzle exit (tipoff yaw, nutation amplitude) contributes to the total dispersion budget.

### 16.7.2 Quantifying the Difference

For a typical .308 Win, 175 gr SMK at 1 000 yards:

Table 16.2: 3DOF vs. 4DOF trajectory comparison at 1 000 yd

| Quantity | 3DOF (Point Mass) | 4DOF (Angular) |
|---|---|---|
| Drop | $-332.4''$ | $-332.4''$ |
| Spin drift (R) | $7.8''$ | $7.5''$ |
| Velocity | 1 096 fps | 1 096 fps |
| Aero. jump (vert.) | — | $\pm 0.3$ MOA |

The drop and velocity are unchanged—the angular motion does not materially affect the translational dynamics for a well-stabilised bullet. The spin drift differs by about 0.3 inches due to the more accurate yaw of repose calculation. The aerodynamic jump is a new prediction that the 3DOF model cannot make.

Listing 16.12: Comparing 3DOF and 4DOF trajectories

```
# 3DOF (point mass with spin drift)
ballistics trajectory \
  --diameter 0.308 --mass 175 --bc 0.505 \
  --velocity 2600 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
  --bullet-length 1.240 --sight-height 1.5

# 4DOF (angular motion + pitch damping)
ballistics trajectory \
  --diameter 0.308 --mass 175 --bc 0.505 \
  --velocity 2600 --auto-zero 100 --max-range 1000 \
  --twist-rate 10 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.240 --sight-height 1.5
```

### 16.7.3 Recommendations

**When to Use the Angular Model**

- **Use 3DOF** (no `--enable-precession`) for: general trajectory computation, zeroing, most competition and hunting scenarios, quick comparisons.
- **Use 4DOF** (with `--enable-precession` and `--enable-pitch-damping`) for: transonic stability analysis, investigating crosswind-induced vertical shifts, Monte Carlo studies of angular dispersion, and understanding why a particular bullet/twist combination performs poorly.

## Exercises

1. **Compare angular models.** Run a 6.5 Creedmoor (140 gr ELD-M, `0.326` G7, 2 710 fps, 1:8″, bullet length 1.375″) to 1 000 yards with and without `--enable-precession`. How much does the spin drift value change?

```
# Without precession
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1000 \
  --twist-rate 8 --enable-spin-drift \
  --bullet-length 1.375 --sight-height 1.5
```

```
# With precession and pitch damping
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1000 \
  --twist-rate 8 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.375 --sight-height 1.5
```

2. **Crosswind and vertical shift.** Run the same 6.5 Creedmoor load with a 10-mph right crosswind and no crosswind, both with `--enable-precession`. Observe the difference in the drop column—this is the aerodynamic jump effect.

```
# No wind, with precession
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1000 \
  --twist-rate 8 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.375 --sight-height 1.5

# 10 mph right crosswind, with precession
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1000 \
  --twist-rate 8 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.375 --sight-height 1.5 \
  --wind-speed 10 --wind-direction 90
```

3. **Marginally stable bullet.** Use the `stability` subcommand to find a twist rate that gives $S_g \approx 1.2$ for a .30-cal 220 gr bullet (length 1.489″). Then run a trajectory with `--enable-precession` and compare to a faster twist that gives $S_g \approx 2.0$.

```
# Find twist rate for marginal stability
ballistics stability \
  --mass 220 --diameter 0.308 --length 1.489 \
  --twist-rate 13 --velocity 2600

# Then run trajectory with that twist rate
ballistics trajectory \
```

```
  --diameter 0.308 --mass 220 --bc 0.629 \
  --velocity 2600 --auto-zero 100 --max-range 1000 \
  --twist-rate 13 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.489 --sight-height 1.5
```

4. **Pitch damping through the transonic zone.** Run a .308 Win, 168 gr SMK (`0.462` G1, 2 700 fps) to 1 200 yards with `--enable-pitch-damping`. At what range does the bullet enter the transonic zone? How does the pitch damping coefficient change sign?

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1200 \
  --twist-rate 10 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.215 --sight-height 1.5
```

## What's Next

Precession, nutation, and pitch damping bring us to the threshold of transonic flight—the regime where bullets are most vulnerable to angular instability. In the next chapter, we focus squarely on *stability and the transonic transition*. Chapter 17 develops the gyroscopic and dynamic stability factors in full detail, explains what happens physically as a bullet decelerates through Mach 1.0, and provides practical guidance for selecting loads that remain accurate through the transonic zone.

# Chapter 17

# Stability & the Transonic Transition

Every spin-stabilised bullet is in a race against time. At the muzzle, the spin rate is at its peak and the stability factor is highest. As the bullet flies downrange, air resistance saps its forward velocity while spin friction gradually erodes its rotational velocity. The stability factor changes—sometimes improving, sometimes degrading—throughout the trajectory. And at some distance, every supersonic bullet crosses into the *transonic zone*: the treacherous band between Mach 1.2 and Mach 0.8 where shock waves form, interact, and break up on the bullet's surface.

For some bullets, the transonic transition is uneventful. For others, it is the boundary beyond which accuracy collapses. Understanding why—and how to choose loads that survive the transition—is one of the most important skills in long-range ballistics.

This chapter develops the theory of gyroscopic and dynamic stability, introduces the Miller stability formula in detail, explores the physics of the transonic zone, and provides practical guidance for selecting loads that remain accurate through Mach 1.

## 17.1 Gyroscopic Stability Factor ($S_g$)

### 17.1.1 Definition

The gyroscopic stability factor $S_g$ is a dimensionless quantity that measures the bullet's resistance to overturning. It compares the gyroscopic (spin-induced) restoring moment to the aerodynamic overturning moment:

$$S_g = \frac{I_s^2 \, \omega_s^2}{2 \, \rho \, V^2 \, S \, d \, C_{M_\alpha} \, I_t} \tag{17.1}$$

where:

- $I_s$ = axial (spin) moment of inertia,
- $\omega_s$ = spin rate (rad/s),
- $\rho$ = air density,
- $V$ = velocity,
- $S$ = reference area ($\pi d^2/4$),
- $d$ = calibre (diameter),
- $C_{M_\alpha}$ = overturning moment coefficient,
- $I_t$ = transverse moment of inertia.

When $S_g > 1$, the gyroscopic moment dominates and the bullet is *gyroscopically stable*: perturbations cause precession rather than tumbling. When $S_g < 1$, the overturning moment wins and the bullet tumbles.

**Stability Thresholds**

$S_g < 1.0$**: Unstable.** The bullet will tumble.
$1.0 \leq S_g < 1.3$**: Marginal.** May experience accuracy issues, especially in the transonic zone or in cold, dense air.
$1.3 \leq S_g < 1.5$**: Adequate.** Acceptable for most conditions but leaves little margin.
$1.5 \leq S_g < 2.5$**: Optimal.** The sweet spot for best accuracy.
$S_g > 2.5$**: Over-stabilised.** The bullet resists following the trajectory arc, increasing drag and reducing long-range accuracy.

### 17.1.2 How $S_g$ Changes During Flight

A key insight: $S_g$ is *not constant*. As the bullet decelerates:

- **Velocity decreases** ($V$ in the denominator of Equation (17.1)), which *increases* $S_g$.
- **Spin rate decreases** ($\omega_s$ in the numerator), which *decreases* $S_g$.

Since velocity decays faster than spin (drag affects translation much more than spin friction affects rotation), $S_g$ generally *increases* during supersonic flight. A bullet that is marginally stable at the muzzle becomes progressively more stable as it slows down.

However, in the transonic zone, the overturning moment coefficient $C_{M_\alpha}$ changes rapidly with Mach number, and this can cause $S_g$ to drop suddenly. This is the mechanism behind transonic destabilisation.

The function `predict_stability_at_distance()` in `src/stability_advanced.rs` models this down-range evolution:

Listing 17.1: Predicting stability at distance

```rust
pub fn predict_stability_at_distance(
    initial_stability: f64,
    initial_velocity_fps: f64,
    current_velocity_fps: f64,
    spin_decay_factor: f64, // 0.95-0.98 typical
) -> f64 {
    let velocity_ratio =
        current_velocity_fps / initial_velocity_fps;
    let spin_ratio = velocity_ratio * spin_decay_factor;

    // SG ~ spin^2 / velocity
    let stability_ratio =
        spin_ratio.powi(2) / velocity_ratio;

    initial_stability * stability_ratio
}
```

## 17.2 Dynamic Stability Factor ($S_d$)

### 17.2.1 Beyond Gyroscopic Stability

Gyroscopic stability ($S_g > 1$) is *necessary* but not *sufficient* for a bullet to fly accurately. A bullet can be gyroscopically stable—its spin axis does not diverge—yet still experience growing angular oscillations because of the interplay between aerodynamic damping and the bullet's inertial properties.

Dynamic stability accounts for how quickly (or slowly) the bullet's angular oscillations decay. A bullet is *dynamically stable* if all angular perturbations (both precession and nutation) decrease over time.

### 17.2.2 The Dynamic Stability Criterion

For full stability, a bullet must satisfy:

$$0 < S_d < 2 \tag{17.2}$$

where $S_d$ is the dynamic stability factor, defined in terms of the aerodynamic damping coefficients. The condition $S_d > 0$ ensures that the slow (precession) mode is damped; $S_d < 2$ ensures that the fast (nutation) mode is damped.

In `src/stability_advanced.rs`, the `calculate_dynamic_stability()` function computes $S_d$ accounting for yaw angle and spin parameter:

Listing 17.2: Dynamic stability computation

```
pub fn calculate_dynamic_stability(
    static_stability: f64,
    velocity_mps: f64,
    spin_rate_rad_s: f64,
    yaw_angle_rad: f64,
    caliber_m: f64,
    _mass_kg: f64,
) -> f64 {
    let spin_param = if velocity_mps > 0.0 {
        spin_rate_rad_s * caliber_m
            / (2.0 * velocity_mps)
    } else { 0.0 };

    let yaw_factor =
        1.0 - 0.1 * yaw_angle_rad.abs().min(0.1);
    let precession_factor =
        1.0 + 0.05 * spin_param.min(0.5);

    static_stability * yaw_factor * precession_factor
}
```

**Why Both $S_g$ and $S_d$ Matter**

A bullet with $S_g = 1.8$ (comfortably stable) can still be dynamically unstable if its pitch damping coefficient is positive in the transonic zone. The gyroscopic factor tells you the bullet won't tumble; the dynamic factor tells you the angular oscillations will decay. Both conditions must be satisfied simultaneously for accurate flight.

## 17.3 The Miller Stability Formula

### 17.3.1 The Original Miller Formula

Don Miller published his simplified stability formula in 2005, providing a practical method for shooters to estimate $S_g$ without requiring aerodynamic coefficients (which are rarely available for small-arms projectiles). The formula uses only mechanical properties of the bullet and barrel:

$$\boxed{S_g = \frac{30\,m}{t_c^2 \cdot d^3 \cdot l_c \cdot (1 + l_c^2)}} \tag{17.3}$$

where:

- $m$ = bullet mass in grains,
- $t_c$ = twist rate in calibres per turn ($t_c = T_{\text{inches}}/d_{\text{inches}}$),
- $d$ = bullet diameter in inches,
- $l_c$ = bullet length in calibres ($l_c = L_{\text{inches}}/d_{\text{inches}}$).

The factor 30 is Miller's empirical constant, calibrated against extensive test data.

### 17.3.2 Velocity and Atmospheric Corrections

The base formula assumes a reference velocity of 2 800 fps and standard atmospheric conditions. To account for actual conditions, Miller specified two correction factors:

$$S_g^{\text{corrected}} = S_g^{\text{base}} \cdot \left(\frac{V}{2800}\right)^{1/3} \cdot \frac{T}{T_0} \cdot \frac{P_0}{P} \tag{17.4}$$

where:

- $V$ = muzzle velocity in fps,
- $T$ and $T_0$ = actual and reference (288.15 K) temperatures in Kelvin,
- $P$ and $P_0$ = actual and reference (1013.25 hPa) pressures.

The velocity correction uses a cube-root relationship because $S_g$ is proportional to $\omega_s^2/V$, and spin rate is proportional to $V$, giving $S_g \propto V$. The cube root is an empirical refinement to Miller's original $V^{1/3}$ scaling.

The atmospheric correction accounts for air density: at higher altitudes (lower density), the aerodynamic overturning moment is weaker, so $S_g$ increases. This is why bullets that are marginally stable at sea level may be perfectly stable in Denver or at the high-altitude ranges common in Western U.S. states.

### 17.3.3 Implementation in `ballistics-engine`

The standard Miller formula is implemented in `compute_stability_coefficient()` in `src/stability.rs`:

Listing 17.3: Miller stability implementation from `src/stability.rs`

```
pub fn compute_stability_coefficient(
    inputs: &BallisticInputs,
    atmo_params: (f64, f64, f64, f64),
) -> f64 {
    const MILLER_CONST: f64 = 30.0;
    const VEL_REF_FPS: f64 = 2800.0;
    const TEMP_REF_K: f64 = 288.15;
    const PRESS_REF_HPA: f64 = 1013.25;
```

```
    let twist_calibers =
        twist_rate_m / inputs.bullet_diameter;
    let length_calibers =
        inputs.bullet_length / inputs.bullet_diameter;

    let mass_term = MILLER_CONST * mass_grains;
    let geom_term = twist_calibers.powi(2)
        * diameter_inches.powi(3)
        * length_calibers
        * (1.0 + length_calibers.powi(2));

    let density_correction =
        (temp_k / TEMP_REF_K)
        * (PRESS_REF_HPA / current_press_hpa);
    let velocity_correction =
        (velocity_fps / VEL_REF_FPS).powf(1.0 / 3.0);

    (mass_term / geom_term)
        * velocity_correction * density_correction
}
```

### 17.3.4 Advanced Stability: Bullet-Type Corrections

The advanced model in `src/stability_advanced.rs` extends Miller's formula with corrections for specific bullet designs:

Table 17.1: Advanced stability correction factors by bullet type

| Bullet Type | Nose Shape | Boat Tail | Plastic Tip | CoP Adj. |
|---|---|---|---|---|
| Match / BTHP | 0.95 | 0.94 | — | 0.98 |
| VLD | 0.88 | 0.92 | — | 0.96 |
| Hybrid | 0.91 | 0.93 | — | 0.97 |
| Hunting (tip) | 0.98 | 0.95 | 0.92 | 0.99 |

These factors reduce the predicted $S_g$ from the base Miller formula. The reductions are small (typically 5–15%) but can shift a borderline case from "adequate" to "marginal." VLD bullets show the largest correction because their secant ogive profile shifts the centre of pressure forward, increasing the overturning moment beyond what the simple Miller formula predicts.

Additionally, for velocities above 3 000 fps, a *Bowman-Howell correction* accounts for dynamic effects that reduce effective stability at hypervelocity:

Listing 17.4: Bowman-Howell correction for high velocity

```
fn apply_bowman_howell_correction(
    sg: f64,
    velocity_fps: f64,
    caliber_inches: f64,
) -> f64 {
    let excess_velocity =
        (velocity_fps - 3000.0) / 1000.0;
    let mach_correction =
        1.0 - 0.05 * excess_velocity.min(2.0);

    let caliber_factor = if caliber_inches < 0.264 {
        0.95
    } else if caliber_inches < 0.308 {
        0.97
    } else {
        1.0
    };

    sg * mach_correction * caliber_factor
}
```

### 17.3.5 Using the `stability` Subcommand

The `stability` subcommand makes all of this accessible from the command line. It computes $S_g$, classifies the result, and also finds the minimum twist rate needed for $S_g = 1.5$ and $S_g = 1.0$ via binary search:

Listing 17.5: Full stability analysis

```
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700
```

Listing 17.6: Stability output

```
# Output:
# +========================================+
# |         Stability Analysis             |
# +========================================+
# | Stability Factor (Sg):          1.81   |
# | Status:              Fully Stable      |
# | Twist Rate:              1:10" RH      |
```

```
# | Min Twist (Sg=1.5):       1:11.4"      |
# | Min Twist (Sg=1.0):      1:13.9"       |
# | Bullet Length:            1.215"        |
# | Muzzle Velocity:         2700 fps      |
# +========================================+
```

The minimum twist rates are found by binary search between 1″ and 40″ per turn, converging to 0.001 $S_g$ tolerance. This is implemented in `handle_stability()` in `src/main.rs`.

### 17.3.6 Atmospheric Effects on Stability

You can test stability at different atmospheric conditions using the `--temperature`, `--pressure`, and `--altitude` flags:

Listing 17.7: Stability at altitude

```
# Sea level, standard conditions
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700

# 7,000 ft elevation, cold morning
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700 \
  --altitude 7000 --temperature 25 --pressure 23.1
```

At 7 000 feet, the lower air density reduces the aerodynamic overturning moment, and $S_g$ increases by 10–15%. This is why bullets that keyhole at sea-level ranges may shoot perfectly at high altitude.

## 17.4 The Transonic Zone: What Happens at Mach 1.0–1.2

### 17.4.1 What Is the Transonic Zone?

The transonic zone spans approximately Mach 0.8 to Mach 1.2. It is the speed range where *both supersonic and subsonic airflow* exist simultaneously over different parts of the bullet's surface. Below Mach 0.8, the flow is entirely subsonic; above Mach 1.2, it is entirely supersonic (with the exception of the wake region). In between, shock waves form, move, strengthen, weaken, and interact in complex patterns.

For a typical .308 Win load at 2 700 fps, the bullet enters the transonic zone at approximately 800–900 yards (where velocity drops below ∼1 340 fps / Mach 1.2) and exits it at around 1 100–1 200 yards (below ∼900 fps / Mach 0.8).

### 17.4.2 The Drag Rise

The most prominent feature of the transonic zone is the dramatic increase in drag. In `src/transonic_drag.rs`, the `transonic_drag_rise()` function models this:

Listing 17.8: Transonic drag rise model

```
pub fn transonic_drag_rise(
    mach: f64,
    shape: ProjectileShape,
) -> f64 {
    let m_crit = critical_mach_number(shape);
    if mach < m_crit { return 1.0; }

    if mach < 1.0 {
        let progress = (mach - m_crit) / (1.0 - m_crit);
        // Shape-dependent rise rate
        match shape {
            ProjectileShape::BoatTail =>
                1.0 + 1.2 * progress.powi(2),
            ProjectileShape::RoundNose =>
                1.0 + 2.0 * progress.powf(1.5),
            _ =>
                1.0 + 1.5 * progress.powf(1.8),
        }
    } else if mach < 1.2 {
        // Peak drag around Mach 1.0-1.1
        // ... shape-dependent peak
    } else {
        1.0 // Handled by base drag tables
    }
}
```

**This Multiplier Is No Longer Applied to the Solve (0.22.0)**

Read `transonic_drag_rise()` as a model of the physics, not as a factor the trajectory solver multiplies in. The tabulated G1 and G7 drag curves *already* embed the full McCoy transonic rise—that is the sharp peak you saw plotted in Chapter 11—and through v0.21.x the solver was applying this multiplier, up to 2.5×, unconditionally on top of it. Every shot that touched Mach 0.85–1.2 was flying with a doubled drag rise, and a subsonic load (.22 LR, subsonic .300 BLK) was flying with it for the entire flight: too much drop, too little retained velocity. Version 0.22.0 removed the double-count. **Any transonic or subsonic reference table built before v0.22.0 is wrong in the same direction.**

The drag rise begins at the *critical Mach number*—the freestream Mach number at which the first point on the bullet's surface locally reaches Mach 1.0:

Table 17.2: Critical Mach number by projectile shape

| **Shape** | $M_{\text{crit}}$ | **Drag Rise Onset** |
|---|---|---|
| Boat tail | 0.88 | Latest onset, gentlest rise |
| Spitzer | 0.85 | Moderate onset |
| Round nose | 0.75 | Early onset, steep rise |
| Flat base | 0.70 | Earliest onset, steepest rise |

Boat-tail bullets benefit from delayed drag rise onset because their tapered base reduces the severity of base separation and the associated shock formation. This is one of the primary reasons boat-tail bullets are preferred for long-range shooting.

### 17.4.3 Wave Drag

Above the critical Mach number, a new drag component appears: *wave drag*. This is the energy lost to the creation and maintenance of shock waves. The `wave_drag_coefficient()` function in `src/transonic_drag.rs` estimates this component based on projectile shape and Mach number:

$$C_{D,\text{wave}} = \frac{C_{D,\text{wave}}^{\text{base}}}{f_r} \cdot \frac{1}{\sqrt{M^2 - 1}} \cdot f_{\text{shape}} \tag{17.5}$$

where $f_r$ is the fineness ratio (length/diameter, typically 3.5 for rifle bullets) and $f_{\text{shape}}$ is a shape correction factor (0.7 for boat tail, 0.8 for spitzer, 1.2 for round nose, 1.5 for flat base).

### 17.4.4 Peak Drag at Mach 1.0–1.05

The drag coefficient peaks at approximately Mach 1.0–1.05 for most bullet shapes. At this point, the bow shock is at its strongest and the wake region is most turbulent. The peak factor reaches 1.8–2.2$\times$ the baseline drag coefficient. Past the peak, drag decreases as the shock system stabilises and the wake narrows.

## 17.5 Why Bullets Destabilise in the Transonic

### 17.5.1 The Pitch Damping Sign Reversal

The primary mechanism for transonic destabilisation is the sign reversal of the pitch damping coefficient. As discussed in Section 16.5 of Chapter 16, the combined coefficient $C_{m_q} + C_{m_{\dot{\alpha}}}$ is

normally negative (stabilising) in both the subsonic and supersonic regimes. But in the upper transonic region (Mach 1.0–1.2), shock wave interactions on the bullet surface can make this coefficient *positive*—the aerodynamic damping now *amplifies* angular oscillations instead of suppressing them.

From `src/pitch_damping.rs`:

Listing 17.9: Pitch damping coefficient interpolation through transonic

```
pub fn calculate_pitch_damping_coefficient(
    mach: f64,
    coeffs: &PitchDampingCoefficients,
) -> f64 {
    if mach < 0.8 {
        coeffs.subsonic  // e.g., -0.8 (stable)
    } else if mach < 1.0 {
        let t = (mach - 0.8) / 0.2;
        coeffs.subsonic * (1.0 - t)
            + coeffs.transonic_low * t
    } else if mach < 1.2 {
        let t = (mach - 1.0) / 0.2;
        coeffs.transonic_low * (1.0 - t)
            + coeffs.transonic_high * t
            // May be POSITIVE! (destabilising)
    } else {
        let t = ((mach - 1.2) / 0.8).min(1.0);
        coeffs.transonic_high * (1.0 - t)
            + coeffs.supersonic * t
    }
}
```

The interpolation ensures a smooth transition through the Mach regimes. The key values from Table 16.1 (reproduced from Chapter 16) show the critical sign change:

- Match boat tail: goes from $-0.4$ (trans. low) to $+0.1$ (trans. high).
- Hunting: goes from $-0.1$ to $+0.4$.
- VLD: remains barely negative ($-0.5$ to $-0.1$) through the transonic—one of the advantages of VLD designs.

### 17.5.2 What the Shooter Sees

When a bullet destabilises in the transonic zone, the observable symptoms are:

1. **Group size explosion**: Groups that were 1–2 MOA at shorter range suddenly expand to 3–5 MOA or worse.

2. **Vertical stringing**: The angular oscillations interact with small velocity variations to produce vertical dispersion.
3. **Unpredictable lateral spread**: The growing yaw oscillation interacts with crosswind to create random lateral deflections.
4. **Keyholing**: In severe cases, the bullet may arrive at the target side-on, leaving an elongated hole.

**Transonic Instability Is Not Tumbling**

A bullet that goes dynamically unstable in the transonic does not necessarily tumble ($S_g$ is still usually $> 1$). Instead, its angular oscillations grow to several degrees, dramatically increasing drag and scatter. The bullet may "recover" stability once it decelerates fully into the subsonic regime, but the damage to accuracy is already done.

### 17.5.3 The Reynolds Number Connection

In the subsonic regime (after the transonic transition), the Reynolds number becomes important. At low velocities, the Reynolds number drops and viscous effects increase the drag coefficient. The `apply_reynolds_correction()` function in `src/reynolds.rs` applies corrections based on three flow regimes:

Table 17.3: Reynolds number flow regimes

| Reynolds Number | Flow Regime | Drag Effect |
|---|---|---|
| $Re < 2\,000$ | Laminar | Stokes drag dominates; $C_D \propto 1/Re$ |
| $2\,000 < Re < 5 \times 10^5$ | Transitional | Power-law correction |
| $Re > 5 \times 10^5$ | Turbulent | Standard drag tables valid |

For a .308 bullet at 300 m/s (∼Mach 0.88), the Reynolds number is approximately $1.5 \times 10^5$—firmly in the transitional regime. At 100 m/s (end-of-trajectory for very long shots), $Re$ drops to about $5 \times 10^4$, where the drag correction factor increases by 20–30%.

## 17.6 Modeling Transonic Behavior in `ballistics-engine`

### 17.6.1 The Transonic Correction Pipeline

The engine applies transonic corrections through the function `transonic_correction()` in `src/transonic_drag.rs`:

Listing 17.10: Transonic correction application

```rust
pub fn transonic_correction(
    mach: f64,
    base_cd: f64,
    shape: ProjectileShape,
    include_wave_drag: bool,
) -> f64 {
    let rise_factor = transonic_drag_rise(mach, shape);
    let mut corrected_cd = base_cd * rise_factor;

    if include_wave_drag && mach > 0.8 {
        let wave_cd = wave_drag_coefficient(mach, shape);
        corrected_cd += wave_cd;
    }
    corrected_cd
}
```

The function multiplies the base drag coefficient (from the G1 or G7 table) by the transonic rise factor and optionally adds the wave drag component. This correction is applied at every integration step when the bullet's Mach number falls within the transonic range.

### 17.6.2 Shape Estimation from Load Data

Since the CLI user does not typically specify the bullet's nose or base shape directly, the engine infers it from the drag model and load data using `get_projectile_shape()`:

Listing 17.11: Projectile shape inference

```rust
pub fn get_projectile_shape(
    caliber: f64,
    weight_grains: f64,
    g_model: &str,
) -> ProjectileShape {
    if g_model == "G7" {
        return ProjectileShape::BoatTail;
    }
    let weight_per_caliber = weight_grains / caliber;
    if weight_per_caliber > 500.0 {
        return ProjectileShape::BoatTail;
    }
    if caliber < 0.35 {
        ProjectileShape::Spitzer
    } else {
        ProjectileShape::RoundNose
    }
}
```

Using a G7 drag model automatically selects the boat-tail shape, which provides the most favourable transonic drag profile. This is consistent with the design philosophy of G7: it is intended for long, boat-tail bullets.

### 17.6.3 Trajectory Stability Checking

The `check_trajectory_stability()` function in `src/stability_advanced.rs` provides a comprehensive stability assessment that accounts for the entire flight path:

Listing 17.12: Checking trajectory stability

```
# Check if a 175 gr SMK will remain stable to 1000 yd
ballistics stability \
  --mass 175 --diameter 0.308 --length 1.240 \
  --twist-rate 11 --velocity 2600
```

The function returns one of five status classifications:

1. **UNSTABLE**: Terminal $S_g < 1.0$. Bullet will tumble.
2. **MARGINAL**: Terminal $S_g$ between 1.0 and 1.3.
3. **ADEQUATE**: Terminal $S_g$ between 1.3 and 1.5.
4. **GOOD**: Terminal $S_g$ between 1.5 and 2.5. Optimal range.
5. **OVER-STABILIZED**: Terminal $S_g > 2.5$. May reduce BC.

## 17.7 Practical Guidance: Staying Supersonic

### 17.7.1 Choosing Bullets That Survive the Transonic

Based on the physics above, the ideal bullet for transonic survival has:

1. **A boat-tail base**: Delays drag rise onset (higher $M_{\text{crit}}$) and reduces the severity of the drag peak.
2. **High $S_g$ at the muzzle**: Provides margin for any degradation in the transonic. Aim for $S_g \geq 1.5$.
3. **VLD or hybrid ogive**: These designs have the most favourable pitch damping coefficients in the transonic zone, remaining weakly stabilising rather than destabilising.
4. **High muzzle velocity**: Pushes the transonic zone further downrange, where the bullet has had more time to settle into its steady-state angular configuration and where the remaining accuracy budget is larger (target size appears larger relative to group size).

Table 17.4: Approximate transonic entry ranges for common cartridges (sea level, standard conditions)

| Cartridge | Load | MV (fps) | Transonic ($M = 1.2$) |
|---|---|---|---|
| .223 Rem | 77 gr SMK | 2 750 | ~650 yd |
| .308 Win | 168 gr SMK | 2 700 | ~850 yd |
| .308 Win | 175 gr SMK | 2 600 | ~800 yd |
| 6.5 CM | 140 gr ELD-M | 2 710 | ~1 050 yd |
| .300 WM | 190 gr SMK | 2 900 | ~1 100 yd |
| .338 LM | 300 gr SMK | 2 750 | ~1 500 yd |

The 6.5 Creedmoor pushes the transonic zone to over 1 000 yards despite a moderate muzzle velocity because the high-BC, low-drag bullets retain velocity efficiently. The .338 Lapua Magnum stays supersonic to nearly a mile, making it a favourite for extreme long-range (ELR) shooting.

### 17.7.2 Cartridge Selection for Beyond-Transonic Shooting

**The Transonic Decision Rule**

- If your maximum engagement range is inside the transonic entry point for your load, you do not need to worry about transonic stability.
- If you routinely shoot *through* the transonic zone (e.g. shooting .308 Win beyond 900 yards), choose bullets specifically designed for transonic stability (VLD, ELD-M, Berger Hybrid) and verify $S_g \geq 1.5$.
- If you need accuracy *beyond* the transonic zone (subsonic arrival), select a cartridge that can push the transonic entry past your target distance (6.5 CM, .300 WM, .338 LM).

### 17.7.3 Altitude and the Transonic Zone

At altitude, the speed of sound decreases slightly (it depends on temperature, not pressure or density directly). But the dominant effect is the reduced air density, which dramatically lowers drag and keeps the bullet supersonic longer. At 5 000 feet, a .308 Win 168 gr SMK stays supersonic roughly 100 yards further than at sea level.

Listing 17.13: Comparing transonic entry at different altitudes

```
# Sea level
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1200 \
  --sight-height 1.5
```

```
# 5,000 ft altitude
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1200 \
  --sight-height 1.5 --altitude 5000

# Compare velocity columns: look for the range step
# where velocity drops below ~1340 fps (Mach 1.2)
```

### 17.7.4 Full Advanced Physics: The Kitchen Sink

For the most complete trajectory modelling—including all spin effects, Coriolis, precession, and transonic corrections—combine all flags:

Listing 17.14: Full-physics trajectory

```
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1200 \
  --twist-rate 8 --bullet-length 1.375 \
  --enable-spin-drift \
  --enable-magnus \
  --enable-coriolis --latitude 45 --shot-direction 0 \
  --enable-precession \
  --enable-pitch-damping \
  --sight-height 1.5
```

**When to Use All the Flags**

For most shooting, the basic trajectory (or trajectory with `--enable-spin-drift`) is sufficient. The full set of advanced physics flags is valuable for:

- Understanding *why* a particular load shoots poorly at distance.
- Preparing for ELR competitions where every fraction of a MOA matters.
- Academic study and comparison with published ballistic data.
- Validating the engine's predictions against measured field data.

## Exercises

1. **Find your transonic range.** Run your primary long-range load to maximum range and examine the velocity column. At what range does velocity drop below 1 340 fps (Mach $\approx 1.2$)? Below 900 fps (Mach $\approx 0.8$)?

```
ballistics trajectory \
  --diameter 0.308 --mass 175 --bc 0.505 \
  --velocity 2600 --auto-zero 100 --max-range 1200 \
  --sight-height 1.5
# Watch the velocity column for the 1340 fps
# and 900 fps crossings
```

2. **Compare twist rates for stability.** Compute stability for a long, heavy 6.5 mm bullet (147 gr ELD-M, length 1.445″, diameter 0.264″) at twist rates of 1:7″, 1:8″, 1:9″, and 1:10″. At what twist rate does $S_g$ drop below 1.5?

```
for twist in 7 8 9 10; do
  ballistics stability \
    --mass 147 --diameter 0.264 --length 1.445 \
    --twist-rate $twist --velocity 2700
done
```

3. **Altitude effect on stability.** Using the 168 gr SMK (.308, 1.215″), compute stability at sea level, 5 000 ft, and 10 000 ft. How much does $S_g$ increase at altitude?

```
# Sea level
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700

# 5,000 ft
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700 \
  --altitude 5000

# 10,000 ft
ballistics stability \
  --mass 168 --diameter 0.308 --length 1.215 \
  --twist-rate 10 --velocity 2700 \
  --altitude 10000
```

4. **6.5 CM vs. .308 Win through the transonic.** Run both cartridges to 1 200 yards with the full physics stack (spin drift, Coriolis, precession, pitch damping). Compare the velocity at 1 000 and 1 200 yards. Which bullet is still supersonic at 1 200?

```
# .308 Win 168 gr SMK
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 100 --max-range 1200 \
  --twist-rate 10 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.215 --sight-height 1.5

# 6.5 Creedmoor 140 gr ELD-M
ballistics trajectory \
  --diameter 0.264 --mass 140 --bc 0.326 \
  --drag-model g7 \
  --velocity 2710 --auto-zero 100 --max-range 1200 \
  --twist-rate 8 --enable-spin-drift \
  --enable-precession --enable-pitch-damping \
  --bullet-length 1.375 --sight-height 1.5
```

5. **Over-stabilisation experiment.** Compute stability for a 55 gr .224 bullet (length 0.740″) in a 1:7″ twist at 3 240 fps. What is $S_g$? Is the bullet over-stabilised?

```
ballistics stability \
  --mass 55 --diameter 0.224 --length 0.740 \
  --twist-rate 7 --velocity 3240
```

## What's Next

With Part V complete, you now have a thorough understanding of the advanced physics that affect a bullet's flight: spin drift, Coriolis and Eötvös effects, precession and nutation, and the transonic transition. These effects are most important at ranges beyond 600 yards and are critical for extreme long-range work.

In Part VI, we shift from physics to *numerics*. Chapter 18 opens with a detailed look at how BALLISTICS-ENGINE solves the equations of motion: the Runge-Kutta integration schemes (RK4 and the adaptive RK45/Dormand-Prince method), step-size control, and the numerical precision that makes all of the physics in this Part meaningful.

# Part VI

# Numerical Methods

# Chapter 18

# The Trajectory Solver

A rifle bullet leaves the muzzle at 2700 fps and must travel 1000 yards through a turbulent atmosphere, shedding velocity every millisecond, curving under gravity, drifting in the wind, and slowly precessing around its spin axis. Predicting where that bullet lands—to a fraction of an inch—requires solving a system of coupled differential equations that has no closed-form solution. This chapter explains exactly how BALLISTICS-ENGINE does it.

We will build up from the fundamental equations of motion, through the state-vector representation that encodes every aspect of the bullet's flight, to the 4th-order Runge–Kutta integrator (RK4) and the adaptive Dormand–Prince method (RK45) that march the solution forward in time. Along the way, we will examine step-size selection, accuracy trade-offs, and the fast trajectory solver that powers long-range batch computations.

Let's start with the physics, then see how BALLISTICS-ENGINE turns those equations into code.

## 18.1 The Equations of Motion

At every instant during flight, a bullet experiences several forces. The trajectory solver's job is to compute the acceleration that these forces produce and then update the bullet's position and velocity accordingly.

### 18.1.1 Newton's Second Law in Vector Form

The core equation is Newton's second law applied to a projectile of mass $m$:

$$m\,\mathbf{a} = \mathbf{F}_{\text{drag}} + \mathbf{F}_{\text{gravity}} + \mathbf{F}_{\text{Coriolis}} + \mathbf{F}_{\text{Magnus}} + \mathbf{F}_{\text{spin}} \tag{18.1}$$

where each force term represents:

- **Drag** ($\mathbf{F}_{\text{drag}}$): aerodynamic resistance opposing the bullet's motion relative to the air mass.
- **Gravity** ($\mathbf{F}_{\text{gravity}}$): the constant downward pull of $g = 9.806\,65\ \text{m/s}^2$.
- **Coriolis** ($\mathbf{F}_{\text{Coriolis}}$): the fictitious force arising from Earth's rotation (see Chapter 15).
- **Magnus** ($\mathbf{F}_{\text{Magnus}}$): the lateral force on a spinning body in a crossflow (see Chapter 14).
- **Spin drift** ($\mathbf{F}_{\text{spin}}$): the slow lateral drift caused by gyroscopic precession.

Dividing both sides by $m$ gives us the acceleration vector:

$$\mathbf{a} = \mathbf{a}_{\text{drag}} + \mathbf{a}_{\text{gravity}} + \mathbf{a}_{\text{Coriolis}} + \mathbf{a}_{\text{Magnus}} + \mathbf{a}_{\text{spin}} \tag{18.2}$$

This is the equation that the trajectory solver evaluates at every time step.

### 18.1.2 The Drag Acceleration

The drag acceleration is by far the dominant force after gravity. BALLISTICS-ENGINE computes it as:

$$a_{\text{drag}} = \frac{C_D \cdot K_r \cdot v_{\text{rel}}^2 \cdot \rho_{\text{ratio}} \cdot (1 + \alpha^2)}{BC} \tag{18.3}$$

where $C_D$ is the drag coefficient (looked up from the G1 or G7 table at the current Mach number), $K_r = 0.000683 \times 0.30$ is the retardation constant defined in `src/constants.rs`, $v_{\text{rel}}$ is the bullet's speed relative to the air (in fps), $\rho_{\text{ratio}}$ is the ratio of local air density to standard sea-level density ($1.225\ \text{kg/m}^3$), $\alpha$ is the yaw angle, and $BC$ is the ballistic coefficient.

The drag acceleration is applied in the opposite direction to the air-relative velocity vector:

$$\mathbf{a}_{\text{drag}} = -a_{\text{drag}} \cdot \frac{\mathbf{v}_{\text{rel}}}{\|\mathbf{v}_{\text{rel}}\|} \tag{18.4}$$

**Wind-Adjusted Velocity**

The velocity used for drag calculations is the bullet's velocity *relative to the air mass*, not relative to the ground. In `src/derivatives.rs`, you will find the line `let velocity_adjusted = vel - wind_vector;` which subtracts the wind vector to obtain the aerodynamic velocity. A 10 mph tailwind effectively reduces the air-relative speed and thus reduces drag.

### 18.1.3 Gravity

Gravity is the constant vertical acceleration:

$$\mathbf{a}_{\text{gravity}} = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ -9.80665 \\ 0 \end{pmatrix} \ \text{m/s}^2 \tag{18.5}$$

In the coordinate system used by BALLISTICS-ENGINE, $y$ is the vertical axis (positive upward), $z$ is the downrange axis, and $x$ is the lateral axis (crosswind direction). Gravity acts only in the $-y$ direction.

### 18.1.4 Coriolis Acceleration

The Coriolis acceleration is computed as:

$$\mathbf{a}_{\text{Coriolis}} = -2\,\boldsymbol{\omega} \times \mathbf{v} \tag{18.6}$$

where $\boldsymbol{\omega}$ is Earth's angular velocity vector projected into the shooter's local frame. The projection depends on both latitude and shot azimuth (see Chapter 15 for details). In `src/trajectory_solver.rs`, the omega vector is constructed as:

Listing 18.1: Earth rotation vector projection

```
let earth_rotation_rate = 7.2921159e-5; // rad/s
Some(Vector3::new(
    earth_rotation_rate * latitude_rad.cos() * azimuth.sin(),
    earth_rotation_rate * latitude_rad.sin(),
    earth_rotation_rate * latitude_rad.cos() * azimuth.cos(),
))
```

### 18.1.5 Magnus Acceleration

The Magnus effect produces a lateral force on a spinning projectile moving through air. In `src/derivatives.rs`, the Magnus acceleration depends on the *Magnus moment coefficient* $C_{L\alpha}$, which varies with Mach number across three flight regimes:

- **Subsonic** ($M < 0.8$): $C_{L\alpha} = 0.030$
- **Transonic** ($0.8 \leq M < 1.2$): linearly interpolated from 0.030 down by a reduction factor of 0.0075
- **Supersonic** ($M \geq 1.2$): $C_{L\alpha} = 0.015 + 0.0044 \cdot \min\left(\frac{M-1.2}{1.8},\ 1\right)$

These coefficients are calibrated against data from McCoy's *Modern Exterior Ballistics* and real-world spin drift measurements.

Let's see all these forces in action with a trajectory that enables the advanced physics:

Listing 18.2: Trajectory with all forces enabled

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --twist-rate 10 --twist-right \
  --enable-spin-drift --enable-magnus \
  --enable-coriolis --latitude 45.0 --shot-direction 90 \
  --wind-speed 10 --wind-direction 90 \
  --output table
```

This command fires a .308 Win, 168 gr Sierra MatchKing (`0.462` G7) at 2700 fps with every physical effect enabled: spin drift, Magnus force, and Coriolis deflection. The solver must evaluate all terms in Equation (18.2) at every time step.

## 18.2 State Vectors: Position, Velocity, and Spin

To integrate the equations of motion, the solver needs to track the bullet's complete state at each instant. In BALLISTICS-ENGINE, the state is a six-element vector:

**The State Vector**

The trajectory state vector $\mathbf{y}$ is a six-dimensional vector:

$$\mathbf{y} = \begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix}$$

where $(x, y, z)$ is the position in meters and $(v_x, v_y, v_z)$ is the velocity in $\mathrm{m/s}$. The coordinate axes are: $x$ = lateral (crosswind), $y$ = vertical (up), $z$ = downrange.

In the Rust implementation, this vector is represented as a `Vector6<f64>` from the `nalgebra` linear algebra crate (in `src/trajectory_integration.rs`) or as a plain `[f64; 6]` array (in `src/fast_trajectory.rs`). The choice of representation affects performance but not the physics.

### 18.2.1 Initial State Construction

The initial state is constructed in `prepare_initial_conditions()` in `src/trajectory_solver.rs`. Given a muzzle velocity $v_0$ and a launch angle $\theta$ (computed by the zeroing algorithm; see Chapter 20), the initial velocity vector is:

$$\mathbf{v}_0 = \begin{pmatrix} v_0 \cos\theta \\ v_0 \sin\theta \\ 0 \end{pmatrix} \tag{18.7}$$

The initial position is the origin $(0, 0, 0)$, placing the muzzle at the coordinate system's reference point. The downrange velocity component starts at zero because, in the solver's frame, the primary velocity axis is $x$ (used during initial condition setup) or $z$ (used during integration after rotation into the downrange frame).

Listing 18.3: Initial state vector construction in trajectory_solver.rs

```
let initial_vel = Vector3::new(
    mv_mps * muzzle_angle_rad.cos(),
    mv_mps * muzzle_angle_rad.sin(),
    0.0,
);
let initial_state = [0.0, 0.0, 0.0,
                     initial_vel.x, initial_vel.y, initial_vel.z];
```

### 18.2.2 Time Span Estimation

The solver also needs to know roughly how long the bullet will be in flight. An overly short time span will cause the simulation to stop before the bullet reaches the target; an overly long one wastes computation. BALLISTICS-ENGINE estimates the maximum time as three times the minimum flight time (distance divided by initial horizontal velocity), clamped to at least 10 seconds:

Listing 18.4: Time span estimation

```
let max_time = if initial_vx > 1e-6 && target_horizontal_dist_m > 0.0 {
    let est_min = target_horizontal_dist_m / initial_vx;
    (est_min * 3.0).max(10.0)
} else {
    10.0
};
```

The factor of three provides a comfortable margin for the drag deceleration that will extend the actual flight time well beyond the no-drag estimate.

### 18.2.3 The `TrajectoryParams` Structure

All parameters needed during integration are bundled into a single `TrajectoryParams` struct defined in `src/trajectory_integration.rs`:

Listing 18.5: TrajectoryParams structure (simplified)

```
pub struct TrajectoryParams {
    pub mass_kg: f64,
    pub bc: f64,
    pub drag_model: DragModel,
    pub wind_segments: Vec<WindSegment>,
    pub atmos_params: (f64, f64, f64, f64),
    pub omega_vector: Option<Vector3<f64>>,
    pub enable_spin_drift: bool,
    pub enable_magnus: bool,
    pub enable_coriolis: bool,
    pub target_distance_m: f64,
    pub is_twist_right: bool,
    pub custom_drag_table: Option<DragTable>,
    pub bc_segments: Option<Vec<(f64, f64)>>,
    pub use_bc_segments: bool,
}
```

This structure carries everything the derivatives function needs without requiring global state, enabling thread-safe parallel simulations—a property that the Monte Carlo solver (Chapter 6) relies on heavily.

## 18.3 Computing Derivatives: Drag, Gravity, Coriolis, Spin

The heart of the trajectory solver is the *derivatives function*. Given the current state vector and time, it returns the time derivatives of every state component—the velocity (derivative of position) and the acceleration (derivative of velocity):

$$\frac{d\mathbf{y}}{dt} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ a_x \\ a_y \\ a_z \end{pmatrix} \tag{18.8}$$

In `src/derivatives.rs`, the function `compute_derivatives()` implements this computation. Let's trace through its logic.

### 18.3.1 Step 1: Wind-Adjusted Velocity

The function begins by computing the bullet's velocity relative to the air mass:

Listing 18.6: Wind adjustment in compute_derivatives()

```
let velocity_adjusted = vel - wind_vector;
let speed_air = velocity_adjusted.norm();
```

This subtraction is crucial: a 10 mph crosswind changes the effective angle of attack and the drag magnitude.

### 18.3.2 Step 2: Atmospheric Conditions

Next, the function obtains the local air density and speed of sound at the bullet's current altitude. Two paths exist: direct atmosphere values (pre-computed density and speed of sound) or calculation from base parameters (altitude, temperature, pressure, humidity):

Listing 18.7: Atmospheric condition lookup

```
let (air_density, speed_of_sound) =
    if atmos_params.0 < 2.0 && atmos_params.1 > 200.0
       && atmos_params.2 == 0.0 && atmos_params.3 == 0.0 {
        get_direct_atmosphere(atmos_params.0, atmos_params.1)
    } else {
        get_local_atmosphere(altitude_at_pos,
            atmos_params.0, atmos_params.1,
            atmos_params.2, atmos_params.3)
    };
```

The Mach number is then computed as $M = v_{\text{air}}/c$, where $c$ is the local speed of sound.

### 18.3.3 Step 3: Drag Coefficient Lookup

The drag coefficient $C_D$ is looked up from the selected drag model (G1 or G7) at the current Mach number, with optional transonic correction and Reynolds number correction:

Listing 18.8: Drag coefficient with corrections

```
let drag_factor = get_drag_coefficient_full(
    mach, &inputs.bc_type,
    true,  // apply transonic correction
    true,  // apply Reynolds correction
    None,  // auto-detect projectile shape
    Some(inputs.caliber_inches),
```

```
    Some(inputs.weight_grains),
    Some(speed_air), Some(air_density),
    Some(atmos_params.1),
);
```

### 18.3.4 Step 4: BC Interpolation

If Mach-based BC segments are available (Chapter 12), the solver interpolates the BC at the current Mach number rather than using a fixed value. The function `interpolated_bc()` in `src/derivatives.rs` performs a binary search on sorted Mach–BC pairs and linearly interpolates between the two nearest points:

Listing 18.9: BC interpolation between Mach segments

```
let (mach1, bc1) = sorted_segments[idx - 1];
let (mach2, bc2) = sorted_segments[idx];
let t = (mach - mach1) / (mach2 - mach1);
bc1 + t * (bc2 - bc1)
```

### 18.3.5 Step 5: Assembling the Acceleration Vector

With the drag magnitude, atmospheric scaling, and BC in hand, the total drag acceleration is computed and applied opposite to the velocity vector. Gravity, Magnus, and Coriolis terms are then added:

Listing 18.10: Final acceleration assembly

```
let mut accel = accel_gravity + accel_drag + accel_magnus;

if let Some(omega) = omega_vector {
    let accel_coriolis = -2.0 * omega.cross(&vel);
    accel += accel_coriolis;
}
```

The function returns the complete derivatives array $[\, v_x,\ v_y,\ v_z,\ a_x,\ a_y,\ a_z \,]$, which the integrator uses to advance the state.

**Why Six Derivatives?**

The first three components are the velocities (derivatives of position), and the last three are the accelerations (derivatives of velocity). This is the standard trick for converting a second-order ODE ($\mathbf{F} = m\mathbf{a}$) into a first-order system suitable for Runge–Kutta integration.

## 18.4 The 4th-Order Runge–Kutta Method (RK4)

With the equations of motion and derivatives function in place, we need a numerical method to march the solution forward in time. The classical 4th-order Runge–Kutta method (RK4) is the workhorse of computational ballistics, offering an excellent balance of accuracy and computational cost.

### 18.4.1 The RK4 Algorithm

Given the current state $\mathbf{y}_n$ at time $t_n$ and a time step $\Delta t$, RK4 computes the next state $\mathbf{y}_{n+1}$ in four stages:

$$\begin{aligned}
\mathbf{k}_1 &= f(t_n,\ \mathbf{y}_n) \\
\mathbf{k}_2 &= f\left(t_n + \frac{\Delta t}{2},\ \mathbf{y}_n + \frac{\Delta t}{2}\,\mathbf{k}_1\right) \\
\mathbf{k}_3 &= f\left(t_n + \frac{\Delta t}{2},\ \mathbf{y}_n + \frac{\Delta t}{2}\,\mathbf{k}_2\right) \\
\mathbf{k}_4 &= f(t_n + \Delta t,\ \mathbf{y}_n + \Delta t\,\mathbf{k}_3) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{\Delta t}{6}\left(\mathbf{k}_1 + 2\,\mathbf{k}_2 + 2\,\mathbf{k}_3 + \mathbf{k}_4\right)
\end{aligned} \tag{18.9}$$

where $f(t, \mathbf{y})$ is the derivatives function from Section 18.3.

The method evaluates the derivatives four times per step: once at the beginning, twice at the midpoint (using different slope estimates), and once at the end. These four evaluations are then combined in a weighted average that cancels error terms through 4th order, yielding a local truncation error of $O(\Delta t^5)$.

### 18.4.2 Implementation in BALLISTICS-ENGINE

The RK4 implementation in `src/trajectory_integration.rs` is clean and direct:

Listing 18.11: RK4 integration step

```
fn rk4_step(state: &Vector6<f64>, t: f64, dt: f64,
            params: &TrajectoryParams) -> Vector6<f64> {
    let k1 = compute_derivatives_vec(state, t, params);
    let k2 = compute_derivatives_vec(
        &(state + dt * 0.5 * k1), t + dt * 0.5, params);
    let k3 = compute_derivatives_vec(
        &(state + dt * 0.5 * k2), t + dt * 0.5, params);
    let k4 = compute_derivatives_vec(
```

```
        &(state + dt * k3), t + dt, params);

    state + (dt / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
}
```

The use of `nalgebra`'s `Vector6` type enables the arithmetic to be expressed naturally—the expression `state + dt * 0.5 * k1` performs element-wise scalar multiplication and addition on all six state components simultaneously.

### 18.4.3 Why RK4 for Ballistics?

RK4 has been the method of choice in computational ballistics for decades, and for good reason:

1. **4th-order accuracy**: The error decreases as $\Delta t^4$ when the step size is halved. A step of 1 ms gives excellent accuracy for most rifle trajectories.
2. **No memory of previous steps**: Unlike multi-step methods (Adams–Bashforth, etc.), RK4 is *self-starting*—it needs only the current state to compute the next one.
3. **Stability**: For the smooth, well-behaved ODEs that describe projectile motion, RK4 is unconditionally stable at practical step sizes.
4. **Compact implementation**: Four function evaluations per step, one formula for the update. The implementation is short enough to verify by inspection.

You can explicitly request RK4 integration with the `--use-rk4-fixed` flag:

Listing 18.12: Using fixed-step RK4 integration

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed \
  --output table
```

### 18.4.4 Target Detection and Interpolation

During fixed-step RK4 integration, the solver must detect when the bullet crosses the target distance. Because the integration proceeds in discrete time steps, the bullet will rarely land exactly on the target plane. The solver handles this with linear interpolation:

Listing 18.13: Target crossing detection in RK4 loop

```
if state[2] < params.target_distance_m
   && new_state[2] >= params.target_distance_m {
```

```
    let alpha = (params.target_distance_m - state[2])
                / (new_state[2] - state[2]);
    let dt_to_target = dt * alpha;
    let final_state = rk4_step(&state, t, dt_to_target, &params);
    trajectory.push((t + dt_to_target, final_state));
    break;
}
```

When the downrange position ($z$ component) crosses the target distance between two steps, the solver computes the fractional step $\alpha$ that would place the bullet exactly at the target, then takes one more RK4 step of that reduced size. This avoids the need for post-hoc interpolation and provides sub-millimeter accuracy at the target plane.

## 18.5 The Adaptive RK45 (Dormand–Prince) Method

While fixed-step RK4 is reliable and predictable, it has a limitation: it uses the same step size throughout the trajectory, even though the dynamics change dramatically. Near the muzzle, the bullet decelerates rapidly and a small step is needed for accuracy. Downrange, the dynamics are smoother and a larger step would suffice. An adaptive method can vary the step size to maintain a target accuracy with fewer total function evaluations.

### 18.5.1 The Dormand–Prince Algorithm

BALLISTICS-ENGINE implements the Dormand–Prince method (also known as RK45), the same algorithm used by SciPy's `solve_ivp` and MATLAB's `ode45`. It computes *seven* derivative evaluations per step to produce both a 5th-order solution and a 4th-order solution. The difference between them provides an error estimate without any extra function evaluations.

The Dormand–Prince coefficients, defined in `src/trajectory_integration.rs`, form the Butcher tableau for the method. The 5th-order solution advances the state:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \left( b_1 \, \mathbf{k}_1 + b_3 \, \mathbf{k}_3 + b_4 \, \mathbf{k}_4 + b_5 \, \mathbf{k}_5 + b_6 \, \mathbf{k}_6 \right) \tag{18.10}$$

while the 4th-order solution uses a different set of weights $(b_1^*, b_3^*, b_4^*, b_5^*, b_6^*, b_7^*)$ on the same stages.

### 18.5.2 Error Estimation and Step Control

The error estimate is the norm of the difference between the two solutions, normalized by the state magnitude:

$$\varepsilon = \frac{\|\mathbf{y}_5 - \mathbf{y}_4\|}{1 + \|\mathbf{y}_n\|} \tag{18.11}$$

If this error is below the requested tolerance, the step is accepted. If not, the step is rejected and retried with a smaller $\Delta t$. Either way, the new step size is computed using a PI controller:

Listing 18.14: Adaptive step size control

```
let safety = 0.9;
let dt_new = if error < tol {
    // Step accepted: grow step size
    dt * safety * (tol / error).powf(0.2).min(2.0)
} else {
    // Step rejected: shrink step size
    dt * safety * (tol / error).powf(0.25).max(0.1)
};
```

The exponents 0.2 (for growth) and 0.25 (for shrinkage) come from the theoretical step-size control formula for a 5th-order method. The safety factor of 0.9 prevents overly aggressive step increases. The `.min(2.0)` and `.max(0.1)` clamps prevent the step from growing or shrinking by more than a factor of two or ten in a single iteration.

**Default Integration Method**

By default, BALLISTICS-ENGINE uses the adaptive RK45 method. This means that unless you specify `--use-rk4-fixed` or `--use-euler`, every trajectory calculation benefits from automatic step-size control. The default tolerance is $10^{-6}$.

### 18.5.3 RK4 vs. RK45: When to Use Which

Table 18.1: Comparison of RK4 and RK45 integrators in BALLISTICS-ENGINE

| Property | RK4 (fixed) | RK45 (adaptive) |
|---|---|---|
| Derivative evals/step | 4 | 7 |
| Error order | $O(\Delta t^5)$ | $O(\Delta t^6)$ |
| Step size | Fixed (user-chosen) | Automatic |
| Error control | None (user responsibility) | Built-in |
| Best for | Predictable timing, batch processing | General use, accuracy-critical work |
| CLI flag | `--use-rk4-fixed` | (default) |

For most users, the default RK45 is the right choice. It automatically adapts to the trajectory's dynamics and provides reliable accuracy without requiring the user to think about step sizes. The fixed RK4 is useful when you need deterministic step counts (for example, when comparing against a reference trajectory) or when integrating into systems that require uniform time spacing.

## 18.6 Step Size Selection and Adaptive Stepping

Step size is the single most important parameter affecting both the accuracy and the performance of the trajectory solver. Too large, and the solution diverges from reality; too small, and the computation takes unnecessarily long.

### 18.6.1 Fixed Step Sizes

For the fixed-step RK4 integrator, BALLISTICS-ENGINE defaults to a time step of 1 ms (0.001 s). This is enforced in the `integrate_trajectory()` function:

Listing 18.15: Default RK4 step size

```
dt = dt.min(max_step).min(0.001);
```

At 2700 fps (823 m/s), a 1 ms step corresponds to approximately 0.82 m (about 0.9 yards) of downrange travel. For a 1000-yard trajectory, this produces roughly 1100 integration steps—more than sufficient for sub-MOA accuracy.

The CLI allows you to override the default step size with the `--time-step` flag:

Listing 18.16: Adjusting the integration time step

```
# Finer time step for maximum accuracy
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed --time-step 0.0001 \
  --output json

# Coarser time step for faster computation
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed --time-step 0.005 \
  --output json
```

### 18.6.2 Adaptive Step Sizes in RK45

The adaptive RK45 integrator starts with an initial step size derived from the total estimated flight time:

Listing 18.17: Initial step size for RK45

```
let mut dt = (t_end - t) / 1000.0;
dt = dt.min(effective_max_step).max(0.0001);
```

The step then evolves automatically based on the error estimate. Near the muzzle, where drag deceleration is strongest, the integrator typically uses steps of $0.1$ ms to $0.5$ ms. Downrange, as the bullet decelerates and the dynamics smooth out, steps may grow to several milliseconds.

### 18.6.3 Wind Shear and Step Size

When wind shear is enabled (`--enable-wind-shear`), the solver reduces the maximum allowed step size to maintain numerical stability:

Listing 18.18: Step size adjustment for wind shear

```
let effective_max_step =
    if params.enable_wind_shear
       && params.wind_shear_model != "none" {
        if params.target_distance_m > 800.0 {
            0.01  // 10ms for long range
        } else {
            0.02  // 20ms for medium range
        }
    } else {
        max_step
    };
```

The altitude-dependent wind variations introduce additional stiffness into the system of equations, requiring smaller steps to resolve the changing wind profile.

### 18.6.4 Safety Limits

The integrator includes several safety mechanisms to prevent runaway computation:

- **Maximum iterations**: A hard limit of 100,000 steps prevents infinite loops. If reached, a warning is printed to standard error.
- **Ground impact detection**: If the $y$ coordinate drops below $-1000$ m, the bullet is considered to have hit the ground and integration stops.

- **Minimum step size**: The step is clamped to at least 0.1 ms (`0.0001` s) to prevent the step from shrinking to zero.

**Extreme Step Sizes**

Setting the time step below 0.01 ms rarely improves accuracy but dramatically increases computation time. For a 1000-yard .308 Win trajectory, a 0.1 ms step produces about 11,000 steps. At 0.01 ms, that becomes 110,000 steps—a 10× slowdown with negligible accuracy improvement. The default 1 ms is appropriate for the vast majority of calculations.

## 18.7 Accuracy vs. Performance: Choosing the Right Step Size

How do you know your step size is small enough? The gold standard is *convergence testing*: run the same trajectory with successively smaller steps and observe when the output stops changing.

### 18.7.1 Convergence Test

Let's test the .308 Win, 168 gr SMK (`0.462` G7, 2700 fps) trajectory at 1000 yards with three different step sizes:

Listing 18.19: Convergence test at different step sizes

```
# Coarse: 5ms steps
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed --time-step 0.005 \
  --auto-zero 100 --output json

# Default: 1ms steps
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed --time-step 0.001 \
  --auto-zero 100 --output json

# Fine: 0.1ms steps
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --use-rk4-fixed --time-step 0.0001 \
  --auto-zero 100 --output json
```

For a typical .308 Win load at 1000 yards, the drop values converge as follows:

Table 18.2: Convergence of drop at 1000 yards vs. step size (RK4)

| Step Size | Steps | Drop (inches) | Change |
|---|---|---|---|
| 5 ms | ~220 | −388.4 | — |
| 1 ms | ~1,100 | −389.1 | 0.7 in |
| 0.1 ms | ~11,000 | −389.2 | 0.1 in |
| 0.01 ms | ~110,000 | −389.2 | <0.01 in |

The key observation is that the default 1 ms step produces results within a fraction of an inch of the converged answer. The 5 ms step introduces about 0.7 inches of error at 1000 yards—enough to notice at long range but acceptable for quick estimates.

### 18.7.2 Practical Guidelines

**Step Size Recommendations**

- **General use**: Default (adaptive RK45 or 1 ms RK4). This covers 99% of scenarios.
- **Precision competition**: Use adaptive RK45 (the default). Its automatic error control handles varying conditions gracefully.
- **Very long range** (> 1500 yards): Adaptive RK45 is strongly preferred. Fixed-step RK4 may need manual step reduction.
- **Batch processing** (Monte Carlo, parameter sweeps): Consider the fast trajectory solver (Section 28.3) for maximum throughput.

### 18.7.3 The Adaptive Advantage

Let's compare the adaptive solver's behavior on two very different trajectories:

Listing 18.20: Short range vs. long range with adaptive solver

```
# 100-yard .223 Rem - minimal drag variation
ballistics trajectory \
  --velocity 2750 --bc 0.372 --mass 77 --diameter 0.224 \
  --drag-model g7 --max-range 100 \
  --auto-zero 100 --output table

# 1500-yard .338 Lapua - extreme drag variation
ballistics trajectory \
  --velocity 2725 --bc 0.818 --mass 300 --diameter 0.338 \
  --drag-model g7 --max-range 1500 \
```

```
  --auto-zero 100 --output table
```

For the 100-yard .223 Rem, the adaptive solver takes perhaps 50 steps. For the 1500-yard .338 Lapua, it may take 300–500 steps, automatically clustering them near the muzzle (where deceleration is strongest) and spacing them farther apart downrange. A fixed-step solver would use the same step count regardless, wasting effort on the smooth parts of the trajectory.

## 18.8 The Fast Trajectory Solver

For applications that require thousands of trajectory evaluations—Monte Carlo simulations, multi-range dope card generation, BC estimation—the standard solver's overhead becomes a bottleneck. BALLISTICS-ENGINE provides a dedicated fast trajectory solver in `src/fast_trajectory.rs` that trades some generality for speed.

### 18.8.1 Design Philosophy

The fast solver makes several deliberate trade-offs:

1. **Fixed-step RK4**: No adaptive step control means no error estimation overhead. The step size is chosen based on the target distance.

2. **Simplified derivatives**: A local `compute_derivatives()` function in `src/fast_trajectory.rs` computes only drag and gravity, omitting Magnus, Coriolis, and enhanced spin drift. This cuts the per-step cost roughly in half.

3. **Distance-adaptive time step**: The step size is selected based on the target distance—smaller steps for short ranges (where every fraction of an inch matters) and larger steps for long ranges (where the absolute step count must be controlled).

Listing 18.21: Distance-adaptive step size in the fast solver

```
let dt = if params.horiz > 200.0 {
    0.001     // 1ms for long range (> 200m)
} else if params.horiz > 100.0 {
    0.0005    // 0.5ms for medium range
} else {
    0.0001    // 0.1ms for short range
};
```

### 18.8.2 The `FastSolution` Structure

The fast solver returns a `FastSolution` struct that stores trajectory data in column-major format for efficient post-processing:

Listing 18.22: FastSolution structure

```
pub struct FastSolution {
    pub t: Vec<f64>,              // Time points
    pub y: Vec<Vec<f64>>,         // State vectors [6 x n_points]
    pub t_events: [Vec<f64>; 3],  // Events: target, apex, ground
    pub success: bool,
}
```

The `y` field stores six vectors (one per state component), each containing one value per time step. This column-major layout enables efficient interpolation via the `sol()` method, which uses binary search and linear interpolation to evaluate the trajectory at arbitrary times.

### 18.8.3 Event Detection

The fast solver tracks three types of events during integration:

1. **Target hit**: The bullet's downrange position ($z$) reaches or exceeds the target distance.
2. **Maximum ordinate**: The bullet reaches its highest point (the apex of the parabolic arc).
3. **Ground impact**: The bullet's $y$ coordinate drops below the ground threshold.

These events are stored in the `t_events` array and used by downstream processing (sampling, dope card generation, etc.).

### 18.8.4 Interpolation on the Fast Solution

The `FastSolution::sol()` method interpolates the trajectory at any requested time value using binary search and linear interpolation:

Listing 18.23: Interpolation on FastSolution

```
pub fn sol(&self, t_query: &[f64]) -> Vec<Vec<f64>> {
    // Binary search for correct interval
    let idx = match self.t.binary_search_by(|&t|
        t.partial_cmp(&tq).unwrap_or(Ordering::Greater))
    {
        Ok(idx) => idx,
        Err(idx) => idx,
    };
```

```
    // Linear interpolation between bracketing points
    let frac = (tq - t0) / (t1 - t0);
    y0 + frac * (y1 - y0)
}
```

This provides $O(\log n)$ lookup for any query time, making it efficient for the sampling pipeline described in Chapter 19.

### 18.8.5 When to Use the Fast Solver

The fast solver is used internally by BALLISTICS-ENGINE in contexts where speed matters more than the last fraction of an inch of accuracy:

- **Monte Carlo simulations**: Thousands of trajectories must be computed; each one uses the fast solver.
- **BC estimation**: Iterative optimization runs many trajectories while tuning the BC to match observed data.
- **Zeroing**: The zeroing algorithm (Chapter 20) evaluates the trajectory repeatedly while searching for the correct launch angle.

For single trajectory computations displayed to the user, BALLISTICS-ENGINE uses the full solver with all physics effects enabled.

Listing 18.24: Fast vs. full solver comparison

```
# Full solver with all effects (default)
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --enable-spin-drift --enable-coriolis --latitude 45 \
  --auto-zero 100 --output table

# The fast solver is used internally; it is not
# directly selectable via CLI, but its effects are
# visible in the speed of Monte Carlo simulations:
ballistics monte-carlo \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 1000 \
  --num-sims 1000 --output summary
```

## Exercises

These exercises explore the trajectory solver's behavior using real BALLISTICS-ENGINE commands. Run each command and observe the output.

1. **RK4 vs. RK45 comparison.** Compute a 1000-yard .308 Win trajectory (168 gr SMK, `0.462` G7, 2700 fps, 100-yard zero) using both `--use-rk4-fixed` and the default adaptive solver. Compare the drop and wind drift at 1000 yards. How large is the difference?

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 100 --use-rk4-fixed --output json

ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 100 --output json
```

2. **Step size sensitivity.** Using `--use-rk4-fixed`, run the same .308 Win trajectory at three different time steps: `0.01`, `0.001`, and `0.0001` seconds. Record the drop at 1000 yards for each. At what step size does the answer stop changing?

3. **Effect of advanced physics.** Run a 6.5 Creedmoor trajectory (140 gr ELD-M, `0.610` G7, 2710 fps) at 1000 yards three times: (a) with no advanced effects, (b) with `--enable-spin-drift` and `--twist-rate` 8, and (c) with all effects (`--enable-spin-drift`, `--enable-magnus`, `--enable-coriolis` with `--latitude` 45). How much does each effect contribute to the wind drift?

4. **Wind shear and step size.** Run a 1500-yard .338 Lapua trajectory (300 gr Berger Hybrid, `0.818` G7, 2725 fps) with and without `--enable-wind-shear`. Observe how the adaptive solver handles the added complexity.

5. **Monte Carlo speed.** Run a Monte Carlo simulation with 100 iterations and then 10,000 iterations for the .308 Win load. How does the computation time scale? The fast solver's contribution to this scaling is the topic of Chapter 19.

**What's Next** The trajectory solver produces a dense stream of time-stamped state vectors—thousands of points along the bullet's flight path. But shooters don't want raw state vectors; they want drop and drift at specific distances. In the next chapter, we'll examine how BALLISTICS-ENGINE transforms the solver's continuous output into the clean, evenly-spaced trajectory tables that appear in your terminal or JSON output.

# Chapter 19

# Trajectory Sampling

The trajectory solver from Chapter 18 produces a continuous stream of state vectors—perhaps a thousand time-stamped data points describing the bullet's position and velocity at every millisecond of flight. But when you ask BALLISTICS-ENGINE for the drop at 500 yards, you don't want a raw dump of integration states. You want a single number: "−52.3 inches."

The gap between the solver's continuous, time-based output and the shooter's range-based, human-readable table is bridged by the *trajectory sampling pipeline*. This chapter explains how BALLISTICS-ENGINE interpolates between integration steps, flags notable events (zero crossings, Mach transitions, the trajectory apex), and formats the output into JSON, CSV, or table form.

## 19.1 The Sampling Problem: Continuous Solver, Discrete Output

The fundamental mismatch between solver and user is dimensional:

- The **solver** operates in the *time domain*. It steps forward in time ($t = 0, 0.001, 0.002, \ldots$ seconds) and records the state at each step. The downrange distances at these time points are *not* evenly spaced because the bullet decelerates throughout its flight.
- The **user** thinks in the *range domain*. They want data at 0, 100, 200, 300 yards—evenly spaced distances.

Consider a .308 Win at 2700 fps. At $t = 0.001$ s, the bullet is at roughly 0.82 m downrange. At $t = 1.5$ s, it may be at 800 m. The user wants to know the drop at exactly 500 yards (457.2 m), but no integration step will land precisely on that distance.

The sampling pipeline solves this by treating the integration output as a lookup table and interpolating between entries to produce values at any requested range.

**Trajectory Sampling**

*Trajectory sampling* is the process of evaluating a numerical trajectory solution at specific downrange distances by interpolating between the solver's time-stepped output points. It transforms a time-domain solution into a range-domain table.

## 19.2 Linear Interpolation Between Integration Steps

The core operation of the sampling pipeline is interpolation. Given the solver's output—arrays of downrange distances and corresponding values (drop, velocity, time, etc.)—the sampler must find the value at an arbitrary query distance.

### 19.2.1 The Interpolation Function

The `interpolate()` function in `src/trajectory_sampling.rs` implements this operation:

Listing 19.1: Linear interpolation for trajectory data

```
fn interpolate(x_vals: &[f64], y_vals: &[f64], x: f64) -> f64 {
    // Boundary conditions
    if x <= x_vals[0] { return y_vals[0]; }
    if x >= x_vals[x_vals.len() - 1] {
        return y_vals[y_vals.len() - 1];
    }

    // Binary search for the correct interval
    let mut left = 0;
    let mut right = x_vals.len() - 1;
    while right - left > 1 {
        let mid = (left + right) / 2;
        if x_vals[mid] <= x { left = mid; }
        else { right = mid; }
    }

    // Linear interpolation
    let x1 = x_vals[left];
    let x2 = x_vals[right];
    let y1 = y_vals[left];
    let y2 = y_vals[right];
    y1 + (y2 - y1) * (x - x1) / (x2 - x1)
}
```

There are three key design decisions in this function:

1. **Binary search**: Rather than scanning linearly through potentially thousands of data points, the function uses binary search to find the bracketing interval in $O(\log n)$ time. For a typical trajectory with 1000 integration steps, this means about 10 comparisons instead of 500 on average.

2. **Boundary clamping**: If the query distance falls before the first data point or after the last, the function returns the nearest boundary value rather than extrapolating. This prevents nonsensical results when the user requests a range slightly beyond the solver's output.

3. **Linear interpolation**: Between the two bracketing points, the function uses the standard linear formula:

$$y = y_1 + (y_2 - y_1) \cdot \frac{x - x_1}{x_2 - x_1} \tag{19.1}$$

   This is sufficient because the integration steps are closely spaced (typically less than 1 meter apart), and the trajectory is smooth between adjacent steps.

**Why Not Higher-Order Interpolation?**

Cubic splines or Hermite interpolation would provide smoother results at the cost of additional complexity and the need for derivative data at each node. For ballistic trajectories sampled at 1 ms intervals, the error from linear interpolation between adjacent steps is well below 0.01 inches—far smaller than any other source of error in the system. The directness and robustness of linear interpolation make it the right choice here.

## 19.3 The Trajectory Sampling Pipeline

The main sampling function, `sample_trajectory()` in `src/trajectory_sampling.rs`, orchestrates the complete transformation from solver output to user-facing data. Let's trace through its operation.

### 19.3.1 Input Data Structures

The sampling pipeline operates on two input structures:

Listing 19.2: Input data structures for sampling

```
pub struct TrajectoryData {
    pub times: Vec<f64>,
    pub positions: Vec<Vector3<f64>>,    // [x, y, z] positions
    pub velocities: Vec<Vector3<f64>>,   // [vx, vy, vz]
    pub transonic_distances: Vec<f64>,   // Mach transition locations
}

pub struct TrajectoryOutputs {
```

```
    pub target_distance_horiz_m: f64,
    pub target_vertical_height_m: f64,
    pub time_of_flight_s: f64,
    pub max_ord_dist_horiz_m: f64,
    pub sight_height_m: f64,
}
```

The `TrajectoryData` struct holds the raw solver output, while `TrajectoryOutputs` provides the context needed for correct drop calculation—most importantly, the sight height above the bore.

### 19.3.2 Step 1: Extract Coordinate Arrays

The first thing the sampler does is extract the individual coordinate arrays from the position vectors. In BALLISTICS-ENGINE, the coordinate system uses $x$ for lateral (crosswind), $y$ for vertical, and $z$ for downrange:

Listing 19.3: Extracting coordinate arrays

```
let x_vals: Vec<f64> = trajectory_data.positions
    .iter().map(|p| p.x).collect();  // lateral (wind drift)
let y_vals: Vec<f64> = trajectory_data.positions
    .iter().map(|p| p.y).collect();  // vertical
let z_vals: Vec<f64> = trajectory_data.positions
    .iter().map(|p| p.z).collect();  // downrange
```

The downrange coordinate ($z$) serves as the independent variable for all subsequent interpolations. Speed and kinetic energy arrays are also pre-computed:

Listing 19.4: Pre-computing speed and energy

```
let speeds: Vec<f64> = trajectory_data.velocities
    .iter().map(|v| v.norm()).collect();
let energies: Vec<f64> = speeds.iter()
    .map(|&speed| 0.5 * mass_kg * speed * speed).collect();
```

### 19.3.3 Step 2: Generate Sampling Distances

The sampler generates a list of evenly-spaced distances from zero to the target distance, based on the requested step size:

Listing 19.5: Generating sample distances

```
let num_steps = (max_dist / step_size).ceil() as usize + 1;
```

```
let distances: Vec<f64> = (0..num_steps)
    .map(|i| i as f64 * step_size)
    .filter(|&d| d <= max_dist + 0.1)
    .collect();
```

For a 1000-yard (914.4 m) trajectory with a 10-meter step, this produces distances at 0, 10, 20, 30, …, 910 meters plus a final point at the target distance itself.

The step size defaults to 10 meters but can be adjusted via the `--sample-interval` flag:

Listing 19.6: Adjusting the sampling interval

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 100 \
  --sample-trajectory --sample-interval 25 \
  --output table
```

### 19.3.4 Step 3: Interpolate All Quantities

For each sampling distance, the pipeline interpolates all five quantities of interest using the downrange ($z$) coordinate as the lookup key:

Listing 19.7: Interpolating trajectory quantities

```
for &distance in &distances {
    let y_interp = interpolate(&z_vals, &y_vals, distance);
    let wind_drift = interpolate(&z_vals, &x_vals, distance);
    let velocity = interpolate(&z_vals, &speeds, distance);
    let time = interpolate(&z_vals,
        &trajectory_data.times, distance);
    let energy = interpolate(&z_vals, &energies, distance);
    // ...
}
```

Each call to `interpolate()` performs a binary search on `z_vals` and then linearly interpolates the corresponding quantity.

### 19.3.5 Step 4: Compute Drop Relative to Line of Sight

The most nuanced part of the sampling pipeline is the drop calculation. "Drop" is not the bullet's raw vertical position—it's the bullet's position *relative to the line of sight* (LOS) from the scope to the target.

The LOS is a straight line from the sight (at height `sight_height_m` above the bore) to the target's vertical position. At any downrange distance $d$, the LOS $y$-coordinate is:

$$y_{\text{LOS}}(d) = h_s + (h_t - h_s) \cdot \frac{d}{d_{\max}} \tag{19.2}$$

where $h_s$ is the sight height, $h_t$ is the target height, and $d_{\max}$ is the target distance. Drop is then:

$$\text{drop}(d) = y_{\text{LOS}}(d) - y_{\text{actual}}(d) \tag{19.3}$$

This convention means that *positive drop* indicates the bullet is *below* the line of sight (the most common case at distance), while *negative drop* means the bullet is *above* the LOS (as occurs between the muzzle and the zero distance for a rising trajectory).

Listing 19.8: Drop calculation relative to line of sight

```
let los_y = outputs.sight_height_m
    + (outputs.target_vertical_height_m - outputs.sight_height_m)
      * distance / max_dist;
let drop = los_y - y_interp;
```

**Drop Sign Convention**

Different ballistics programs use different sign conventions for drop. In BALLISTICS-ENGINE, positive drop means the bullet is *below* the line of sight. Some programs use the opposite convention. Always check the sign convention when comparing output between programs.

### 19.3.6 Step 5: Assemble Sample Points

Each interpolated set of values is packaged into a `TrajectorySample` structure:

Listing 19.9: The TrajectorySample structure

```
pub struct TrajectorySample {
    pub distance_m: f64,
    pub drop_m: f64,
    pub wind_drift_m: f64,
```

```
    pub velocity_mps: f64,
    pub energy_j: f64,
    pub time_s: f64,
    pub flags: Vec<TrajectoryFlag>,
}
```

The `flags` field is initially empty; trajectory flags are added in the next step.

## 19.4 Trajectory Flags: Zero Crossings, Apex, Mach Transitions

Once the basic sample points are generated, the pipeline annotates them with *flags* that mark notable trajectory events. Three types of flags exist:

**Trajectory Flags**

- `ZeroCrossing`: The bullet crosses the line of sight (drop changes sign). For a typical zeroed rifle, this occurs twice: once close to the muzzle (the "near zero") and once at the zero distance (the "far zero").
- `MachTransition`: The bullet decelerates through a Mach number boundary (typically the transonic transition near $M = 1.0$).
- `Apex`: The trajectory reaches its highest point above the line of sight (the maximum ordinate).

### 19.4.1 Zero Crossing Detection

Zero crossings are detected by scanning adjacent sample points for sign changes in the drop value:

Listing 19.10: Zero crossing detection

```
fn detect_zero_crossings(
    samples: &mut [TrajectorySample], tolerance: f64
) {
    let drops: Vec<f64> = samples.iter()
        .map(|s| s.drop_m).collect();

    for i in 0..(drops.len() - 1) {
        let current = drops[i];
        let next = drops[i + 1];

        let crosses_zero =
            (current < -tolerance && next >= -tolerance) ||
            (current > tolerance && next <= tolerance);
```

```
        if crosses_zero {
            samples[i + 1].flags
                .push(TrajectoryFlag::ZeroCrossing);
        }
    }
}
```

The tolerance parameter ($10^{-6}$ by default) prevents noise near zero from producing spurious crossings. Points with drop values within the tolerance of zero are also flagged, catching cases where the sample happens to land exactly on the LOS.

### 19.4.2 Mach Transition Detection

Mach transitions are detected differently. The transonic distances are pre-computed during integration and passed to the sampling pipeline in the `TrajectoryData` structure. The sampler finds the closest sample point to each transition distance:

Listing 19.11: Mach transition flagging

```
for &transonic_dist in transonic_distances {
    if let Some(idx) = find_closest_sample_index(
        samples, transonic_dist) {
        samples[idx].flags
            .push(TrajectoryFlag::MachTransition);
    }
}
```

The `find_closest_sample_index()` function uses binary search on the sample distances to find the nearest match in $O(\log n)$ time.

### 19.4.3 Apex Detection

The trajectory apex is the point of minimum drop (most negative drop value, meaning the bullet is farthest above the LOS). The sampler searches all sample points up to the target distance:

Listing 19.12: Apex detection

```
let mut min_drop = f64::INFINITY;
let mut apex_idx = 1;

for i in 1..samples.len() {
    if samples[i].distance_m > target_distance_m { break; }
    if samples[i].drop_m < min_drop {
        min_drop = samples[i].drop_m;
```

```
        apex_idx = i;
    }
}
samples[apex_idx].flags.push(TrajectoryFlag::Apex);
```

**Reading Trajectory Flags**

When using JSON output, trajectory flags appear in the `flags` array of each sample point. This makes it straightforward to programmatically identify the zero crossings, apex, and transonic transitions for plotting or analysis:

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 \
  --sample-trajectory --sample-interval 50 \
  --output json
```

## 19.5 Output at Specific Ranges vs. Fixed Step Sizes

BALLISTICS-ENGINE supports two modes of trajectory output, both built on the same sampling pipeline:

### 19.5.1 Fixed-Step Output

The default mode generates samples at regular distance intervals. The interval is controlled by the `--sample-interval` flag (in meters for metric, yards for imperial):

Listing 19.13: Fixed-step trajectory output

```
# Output every 100 yards
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 \
  --sample-trajectory --sample-interval 100 \
  --output table
```

This is ideal for generating dope cards, where the shooter needs data at regular intervals (every 25 yards, every 50 yards, every 100 yards, etc.).

### 19.5.2 Full Trajectory Output

When the `--full` flag is used, BALLISTICS-ENGINE outputs every integration step rather than sampling at fixed intervals:

Listing 19.14: Full trajectory output

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 --full \
  --output csv
```

This produces a much larger output (potentially thousands of lines) but provides the finest possible resolution for analysis or plotting. Combined with CSV output, this is useful for importing into spreadsheets or plotting software.

### 19.5.3 Minimum Step Size Guard

The sampling pipeline enforces a minimum step size of 0.1 meters to prevent degenerate cases:

Listing 19.15: Minimum step size guard

```
let step_size = if step_m <= 0.0 {
    return Vec::new();
} else if step_m < 0.1 {
    0.1
} else {
    step_m
};
```

This prevents users from accidentally generating millions of sample points with a very small step size.

## 19.6 JSON, CSV, and Table Output Formatting

The final stage of the sampling pipeline is formatting the `TrajectorySample` data into the user's requested output format.

### 19.6.1 Table Output

The default output format presents a human-readable table with aligned columns:

Listing 19.16: Table output example

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 \
  --sample-trajectory --sample-interval 100 \
  --output table
```

The sampled table it prints:

```
Sampled Trajectory (every 109 yd)
+----------+----------+----------+----------+----------+
| Dist(yd) | Drop(in) |Drift(in) | Vel(fps) |  Flags   |
+----------+----------+----------+----------+----------+
|      0.0 |     2.00 |     0.00 |   2700.0 | -        |
|    109.4 |    -1.65 |     0.00 |   2591.8 | Zero,Apex |
|    218.7 |     0.90 |     0.00 |   2486.0 | Zero     |
|    328.1 |    10.17 |     0.00 |   2382.6 | -        |
|    437.4 |    26.77 |     0.00 |   2281.7 | -        |
|    546.8 |    51.36 |     0.00 |   2183.4 | -        |
+----------+----------+----------+----------+----------+
```

Note the header: we asked for `--sample-interval 100` and got stations every 109.4 yards. The sampler's interval is always meters, whatever `--units` says. Note too that the table stops at 546.8 yards rather than the 1000 we asked for—this load, zeroed at 200 from a five-foot bore height, reaches the ground at 585 yards.

The table format is designed for quick reading at the range. Columns are aligned, units are shown in the header, and the data is compact enough to fit on a phone screen.

### 19.6.2 JSON Output

JSON output provides machine-readable data suitable for further processing:

Listing 19.17: JSON output format

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 \
  --sample-trajectory --sample-interval 100 \
  --output json
```

Each sample point is serialized as a JSON object with fields for distance, drop, wind drift, velocity, energy, time, and flags. The flags array enables programmatic identification of trajectory events.

### 19.6.3 CSV Output

CSV output produces comma-separated values suitable for import into spreadsheets and plotting tools:

Listing 19.18: CSV output for data analysis

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 --full \
  --output csv
```

The CSV format includes a header row with column names, making it immediately usable in tools like Excel, Google Sheets, or Python's pandas library.

### 19.6.4 PDF Output

For shooters who want a professional dope card, BALLISTICS-ENGINE can generate a PDF directly:

Listing 19.19: PDF dope card generation

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 \
  --sample-trajectory --sample-interval 25 \
  --output pdf
```

The PDF output uses the same sampling pipeline but routes the `TrajectorySample` data through a layout engine that produces a formatted card with header information, environmental conditions, and a trajectory table—ready to laminate and take to the range.

**Choosing the Right Output Format**

- **Table**: Quick reference at the range or in the terminal.
- **JSON**: Integration with other software, web applications, or automated testing.
- **CSV**: Spreadsheet analysis, plotting, or data archival.
- **PDF**: Professional dope cards for field use.

### 19.6.5 The `TrajectoryDict` Conversion

For interoperability with other languages (Python bindings, WASM), the sampling pipeline can convert `TrajectorySample` objects into a flat dictionary format via `trajectory_samples_to_dicts()`:

Listing 19.20: Converting samples to dictionary format

```
pub fn trajectory_samples_to_dicts(
    samples: &[TrajectorySample]
) -> Vec<TrajectoryDict> {
    samples.iter().map(|sample| TrajectoryDict {
        distance_m: sample.distance_m,
        drop_m: sample.drop_m,
        wind_drift_m: sample.wind_drift_m,
        velocity_mps: sample.velocity_mps,
        energy_j: sample.energy_j,
        time_s: sample.time_s,
        flags: sample.flags.iter()
            .map(|f| f.to_string()).collect(),
    }).collect()
}
```

This function strips the type-safe flag enums into string representations ("zero_crossing", "mach_transition", "apex") for straightforward serialization.

## Exercises

1. **Sampling resolution.** Compute a .308 Win trajectory (168 gr SMK, `0.462` G7, 2700 fps, 200-yard zero) out to 1000 yards at three sampling intervals: 10, 50, and 100 yards. Compare the drop values at 500 yards across all three. Are they identical? Why or why not?

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 200 \
  --sample-trajectory --sample-interval 10 --output table

ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 200 \
  --sample-trajectory --sample-interval 50 --output table

ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 200 \
```

```
  --sample-trajectory --sample-interval 100 --output table
```

2. **Finding the zero crossings.** Using JSON output, compute a .223 Rem trajectory (77 gr SMK, `0.372` G7, 2750 fps, 200-yard zero) out to 600 yards with 10-yard sampling. Identify the two zero crossings (near and far). At what distances do they occur?

```
ballistics trajectory \
  --velocity 2750 --bc 0.372 --mass 77 --diameter 0.224 \
  --drag-model g7 --max-range 600 --auto-zero 200 \
  --sample-trajectory --sample-interval 10 --output json
```

3. **Full vs. sampled output.** Compare the file sizes of full output (`--full`) vs. sampled output (`--sample-trajectory` with a 100-yard interval) for a 1000-yard 6.5 Creedmoor trajectory. How many data points does each produce?

4. **The transonic flag.** Compute a 1500-yard .308 Win trajectory and use JSON output with 25-yard sampling to find the distance at which the Mach transition flag appears. At roughly what range does the 168 gr SMK go transonic?

5. **CSV analysis.** Export a full trajectory in CSV format for the 6.5 Creedmoor (140 gr ELD-M, `0.610` G7, 2710 fps, 100-yard zero, 1000 yards). Import it into a spreadsheet and plot velocity vs. distance. Identify the transonic region from the plot.

**What's Next** Before the trajectory solver can compute drop and drift, it needs to know the launch angle that zeros the rifle at a given distance. Finding that angle is itself a numerical problem—one that requires running the trajectory solver repeatedly inside a root-finding loop. The next chapter explores how BALLISTICS-ENGINE solves this "zeroing problem" using Brent's method, a hybrid algorithm that combines the reliability of bisection with the speed of inverse quadratic interpolation.

# Chapter 20

# The Zeroing Algorithm

You set your rifle's zero at 200 yards. You dial your scope until the crosshairs sit on a target exactly 200 yards away, and the bullet lands on the crosshairs. But what angle must the barrel actually point above the line of sight to achieve this? For a .308 Win at 2700 fps, the answer is approximately 1.9 milliradians—a tiny upward tilt that lofts the bullet on a parabolic arc, intersecting the line of sight at exactly 200 yards.

Finding this angle is a root-finding problem: search the space of possible launch angles for the one that produces zero drop at the target distance. This chapter explains how BALLISTICS-ENGINE solves it.

## 20.1 The Zeroing Problem: Finding the Launch Angle

Formally, the zeroing problem is:

**The Zeroing Problem**

Given a projectile with known mass, BC, and muzzle velocity, find the launch angle $\theta^*$ such that the bullet's vertical position at the target distance $R$ equals the target height $h$:

$$y(\theta^*,\ R) = h$$

where $y(\theta, R)$ is the bullet's height at range $R$ when launched at angle $\theta$. For a flat shot to a target at the same elevation, $h = 0$ (or, more precisely, $h$ equals the sight height above bore).

The function $y(\theta, R)$ cannot be expressed in closed form because it depends on the full trajectory integration—drag is velocity-dependent, atmospheric conditions change with altitude, and the

bullet's deceleration is nonlinear. We can only evaluate $y(\theta, R)$ by running the trajectory solver from Chapter 18.

This makes zeroing an *implicit* problem: we must repeatedly guess an angle, run the solver, check the result, and refine the guess.

Let's see the zeroing command in action:

Listing 20.1: Computing a zero angle

```
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 200
```

This command finds the launch angle that zeros a .308 Win, 168 gr SMK (`0.462` G7) at 200 yards. Internally, the solver evaluates the trajectory dozens of times while searching for the correct angle.

## 20.2 Bisection Search: Reliable but Slow

The most intuitive root-finding method is bisection. Given two angles $a$ and $b$ where the height error $f(\theta) = y(\theta, R) - h$ has opposite signs (meaning the bullet lands above the target for one angle and below for the other), bisection repeatedly halves the interval:

$$\begin{aligned} c &= \frac{a+b}{2} \\ \text{if } f(a) \cdot f(c) < 0 &\implies [a,\, b] \leftarrow [a,\, c] \\ \text{otherwise} &\implies [a,\, b] \leftarrow [c,\, b] \end{aligned} \tag{20.1}$$

After $n$ iterations, the interval width is $\frac{b-a}{2^n}$. For an initial bracket of $\pm 10$ degrees ($\approx 0.349$ radians) and a tolerance of $10^{-6}$ radians, bisection requires about:

$$n = \left\lceil \log_2 \left( \frac{0.698}{10^{-6}} \right) \right\rceil = 20 \text{ iterations}$$

Each iteration requires one trajectory evaluation, so 20 full trajectory computations. This is *guaranteed* to converge (assuming the root is bracketed), but it's not fast.

### 20.2.1 Strengths and Weaknesses

For ballistic zeroing, the initial bracket is straightforward: shooting upward ($\theta > 0$) produces positive height at target, and shooting flat or downward produces negative height. The root is always between these extremes.

Table 20.1: Bisection method characteristics

| Strength | Weakness |
|---|---|
| Always converges | Linear convergence rate |
| Requires only sign information | Needs initial bracket |
| Immune to discontinuities | Cannot exploit smoothness |
| Robust against noisy functions | 20+ iterations typical |

**Bisection as a Fallback**

Pure bisection is rarely used as the primary method in BALLISTICS-ENGINE, but its reliability makes it an excellent fallback when faster methods fail. The hybrid approach described in Section 20.4 falls back to bisection steps whenever the accelerated method would produce an unreliable result.

## 20.3 Newton's Method: Fast but Fragile

Newton's method uses the derivative (slope) of the function to make much larger, better-informed steps than bisection:

$$\theta_{n+1} = \theta_n - \frac{f(\theta_n)}{f'(\theta_n)} \tag{20.2}$$

For our zeroing problem, $f'(\theta)$ is the sensitivity of the target height to the launch angle—essentially, "how many inches does the impact point move for each milliradian of angle change?" Near the correct angle, this relationship is approximately linear, and Newton's method converges *quadratically*: each iteration roughly doubles the number of correct digits.

A Newton solver would typically converge in 4–6 iterations, compared to bisection's 20. However, it has significant drawbacks for ballistic zeroing:

1. **Derivative estimation**: The exact derivative $f'(\theta)$ is not analytically available. We would need to estimate it numerically using finite differences, which requires *two* trajectory evaluations per iteration instead of one.

2. **Divergence risk**: If the function has inflection points or near-zero derivatives, Newton's method can diverge—producing progressively worse estimates rather than converging.

3. **No guaranteed convergence**: Unlike bisection, Newton's method offers no convergence guarantee unless the initial guess is sufficiently close to the root.

For these reasons, BALLISTICS-ENGINE does not use pure Newton's method. Instead, it employs Brent's method, which combines the reliability of bisection with the speed of interpolation-based approaches.

## 20.4 The Hybrid Approach in BALLISTICS-ENGINE

The actual zero-finding algorithm used in BALLISTICS-ENGINE is *Brent's method*, implemented in `src/angle_calculations.rs`. Brent's method is one of the most respected root-finding algorithms in numerical computing—it combines bisection, secant method, and inverse quadratic interpolation, automatically choosing the best strategy at each step.

### 20.4.1 The Three Strategies

At each iteration, Brent's method considers three possible steps:

1. **Inverse quadratic interpolation (IQI)**: Fits a quadratic through the three most recent points $(a, f(a))$, $(b, f(b))$, $(c, f(c))$ and computes where the quadratic crosses zero. This is the fastest option when the function is smooth.
2. **Secant method**: A linear interpolation step using only the two most recent points. Faster than bisection, slightly less aggressive than IQI.
3. **Bisection**: The failsafe. Halves the current bracket regardless of the function's shape.

The algorithm attempts IQI first. If the IQI step would land outside the current bracket or wouldn't sufficiently reduce the interval, it falls back to the secant method. If the secant step is also unsatisfactory, it falls back to bisection.

### 20.4.2 Implementation Walkthrough

Let's trace through the core of `brent_root_find()` in `src/angle_calculations.rs`:

Listing 20.2: Brent's method core loop (simplified)

```
pub fn brent_root_find<F>(
    f: F, mut a: f64, mut b: f64,
    tolerance: f64, max_iterations: usize,
) -> Result<AngleResult, String>
where F: Fn(f64) -> f64
{
    let mut fa = f(a);
    let mut fb = f(b);

    // Ensure root is bracketed
    if fa * fb > 0.0 {
```

```
        return Err("Root not bracketed".to_string());
    }

    let mut c = a;
    let mut fc = fa;
    let mut d = b - a;
    let mut e = d;

    while iterations < max_iterations {
        // Check convergence
        if fb.abs() < tolerance { return Ok(...); }

        let m = 0.5 * (c - b);
        if m.abs() <= tolerance_scaled { return Ok(...); }

        if e.abs() >= tolerance_scaled
           && fc.abs() > fb.abs() {
            // Attempt interpolation (IQI or secant)
            // ... [see source for full logic]
        } else {
            // Fall back to bisection
            d = m;
            e = d;
        }

        // Update bracket
        a = b; fa = fb;
        b += d;
        fb = f(b);

        if (fc * fb) > 0.0 {
            c = a; fc = fa;
            e = b - a; d = e;
        }
    }
}
```

The key innovation in Brent's method is the *acceptance test*: the interpolation step is only used if it provides a better reduction than bisection would. This guarantees that Brent's method converges at least as fast as bisection, while typically converging much faster thanks to the interpolation.

### 20.4.3 Safe Division Guards

The implementation in BALLISTICS-ENGINE includes several safety guards against numerical issues that can arise during zeroing:

Listing 20.3: Division safety guards

```
if fc.abs() < f64::EPSILON || fa.abs() < f64::EPSILON {
    // Fallback to bisection if denominators are too small
    d = m;
    e = m;
}
```

These guards protect against division by zero when function values are extremely small—a situation that can occur when the algorithm is very close to the root.

### 20.4.4 The `AngleResult` Structure

Brent's method returns a rich result structure that includes convergence diagnostics:

Listing 20.4: AngleResult structure

```
pub struct AngleResult {
    pub angle_rad: f64,       // The zero angle in radians
    pub iterations_used: usize,
    pub final_error: f64,     // Residual at convergence
    pub success: bool,        // Did we meet tolerance?
}
```

This allows the calling code to verify that the zeroing converged successfully and to report the number of iterations used—useful for debugging and performance monitoring.

## 20.5 The Zeroing Pipeline

The complete zeroing pipeline in BALLISTICS-ENGINE involves several layers. Let's trace the flow from CLI command to final angle.

### 20.5.1 The `zero_angle()` Function

The top-level zeroing function is `zero_angle()` in `src/angle_calculations.rs`. It wraps Brent's method with appropriate initial bounds and a fallback strategy:

Listing 20.5: Top-level zero angle function

```
pub fn zero_angle(
    inputs: &InternalBallisticInputs,
    trajectory_func: impl Fn(&InternalBallisticInputs, f64)
                        -> Result<f64, String> + Copy,
) -> Result<AngleResult, String> {
    let height_diff = |look_angle_rad: f64| -> f64 {
        match trajectory_func(inputs, look_angle_rad) {
            Ok(bullet_height) => bullet_height - vert,
            Err(_) => f64::NAN,
        }
    };

    // Primary bounds: +/- 10 degrees
    let lower = -10.0 * DEGREES_TO_RADIANS;
    let upper = 10.0 * DEGREES_TO_RADIANS;

    match brent_root_find(height_diff, lower, upper,
                          1e-6, 100) {
        Ok(result) if result.success => Ok(result),
        _ => {
            // Fallback: wider bounds (+/- 45 degrees)
            brent_root_find(height_diff,
                -45.0 * DEGREES_TO_RADIANS,
                45.0 * DEGREES_TO_RADIANS,
                1e-5, 150)
        }
    }
}
```

Several design choices are worth noting:

1. **NaN on failure**: If the trajectory solver fails for a given angle (for example, the bullet hits the ground before reaching the target), the height difference function returns `f64::NAN` rather than a sentinel value like $-999$. NaN propagation causes Brent's method to reject that step gracefully without creating a false root.

2. **Two-stage search**: The function first tries a narrow bracket ($\pm 10$ degrees), which covers the vast majority of practical zeroing scenarios. If this fails (no root found or convergence failure), it falls back to a wider $\pm 45$ degree bracket with relaxed tolerance.

3. **Generic trajectory function**: The `trajectory_func` parameter is a closure that runs the trajectory solver and returns the bullet height at the target. This design decouples the zeroing algorithm from the specific solver implementation, allowing different solvers (full, fast) to be used interchangeably.

### 20.5.2 The `solve_muzzle_angle()` Function

For more complex scenarios—shooting at elevated or depressed targets—BALLISTICS-ENGINE provides `solve_muzzle_angle()` in the same file. This function adds automatic bracket expansion:

Listing 20.6: Automatic bracket expansion

```
if f_lower * f_upper > 0.0 {
    let step = 5.0 * DEGREES_TO_RADIANS;
    let max_angle = 45.0 * DEGREES_TO_RADIANS;
    let mut current = upper;

    while current < max_angle && f_lower * f_current > 0.0 {
        current += step;
        f_current = vertical_error(current);
    }

    if f_lower * f_current > 0.0 {
        return Err("Unable to bracket zero");
    }
    upper = current;
}
```

If the initial bracket doesn't contain a sign change (the root isn't bracketed), the function expands the upper bound in 5-degree steps until it finds a bracket or reaches the 45-degree limit.

Listing 20.7: Zeroing at an elevated target

```
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 500 \
  --target-height 100
```

## 20.6 Convergence Criteria and Tolerances

How does BALLISTICS-ENGINE decide that the zero angle has been found "accurately enough"? The convergence criteria operate on two levels.

### 20.6.1 Primary Tolerance

The Brent's method implementation uses a tolerance of $10^{-6}$ radians for the primary search. In practical terms:

$$10^{-6} \text{ rad} \approx 0.0000573 \approx 0.006 \text{ MOA}$$

At 1000 yards, this angular uncertainty corresponds to about 0.06 inches of impact shift—well below the resolution of any scope adjustment and far smaller than real-world shot-to-shot variation.

### 20.6.2 Scaled Tolerance

Within the Brent's method iteration, a scaled tolerance is computed that accounts for the magnitude of the current estimate:

$$\varepsilon_{\text{scaled}} = 2\,\varepsilon_{\text{machine}} \cdot |b| + \frac{\text{tol}}{2} \tag{20.3}$$

where $\varepsilon_{\text{machine}} \approx 2.2 \times 10^{-16}$ is the machine epsilon for 64-bit floating-point numbers, and $|b|$ is the current best estimate. This prevents the algorithm from trying to achieve accuracy beyond what floating-point arithmetic can represent.

### 20.6.3 Iteration Limits

The zeroing algorithm enforces two iteration limits:

- **Primary search**: 100 iterations maximum. In practice, Brent's method converges in 8–15 iterations for typical zeroing problems.
- **Fallback search**: 150 iterations maximum with relaxed tolerance ($10^{-5}$ radians).

If both searches exhaust their iteration budgets without converging, the function returns the best estimate found with `success: false`. In extreme cases where even the wider search fails entirely, a safe default of 0 radians is returned, and the calling code can detect the failure via the `AngleResult.success` field.

Listing 20.8: Maximum iteration limit

```
const ZERO_FINDING_MAX_ITER: usize = 100;
```

### 20.6.4 Relaxed Success Criteria

When the maximum iterations are reached without meeting the strict tolerance, BALLISTICS-ENGINE applies a relaxed criterion:

Listing 20.9: Relaxed convergence at iteration limit

```
Ok(AngleResult {
    angle_rad: b,
    iterations_used: iterations,
    final_error: fb.abs(),
    success: fb.abs() < tolerance * 10.0,
})
```

If the residual is within 10× the requested tolerance, the result is marked as successful. This handles cases where Brent's method oscillates near the root without quite meeting the strict criterion.

**Checking Convergence**

The zero command's output includes the computed angle. For the `--auto-zero` flag on the trajectory command, convergence is silent—if zeroing fails, the trajectory will show incorrect drop values. When debugging unexpected results, run the standalone `zero` command first to verify convergence:

```
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 200
```

## 20.7 Edge Cases: Steep Angles, Very Long Range, Subsonic Zeros

The zeroing algorithm handles typical flat-shooting scenarios effortlessly. But several edge cases deserve attention.

### 20.7.1 Elevated and Depressed Targets

When the target is significantly above or below the shooter, the effective gravity component along the trajectory changes. A target at 45 degrees above the shooter experiences reduced effective gravity by a factor of $\cos(45) \approx 0.707$, significantly altering the required launch angle. In the `zero` command, this is handled via the `--target-height` flag (positive for elevated targets, negative for depressed targets).

Listing 20.10: Zeroing with elevated targets

```
# Level shot (target at same elevation)
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 500

# Elevated target: 250 yards above the shooter
```

```
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 500 \
  --target-height 250

# Depressed target: 250 yards below the shooter
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 500 \
  --target-height -250
```

The zeroing algorithm handles these cases through the `zero_angle()` function, which adjusts the target height based on the shooting angle:

Listing 20.11: Shooting angle adjustment

```
let vert = if inputs.shooting_angle.abs() > 1e-6 {
    let angle_rad = inputs.shooting_angle * DEGREES_TO_RADIANS;
    (inputs.target_distance * YARDS_TO_METERS) * angle_rad.sin()
} else {
    0.0
};
```

The target height $h$ becomes non-zero, and the root-finding problem becomes $y(\theta, R) = h$ instead of $y(\theta, R) = 0$. Brent's method handles this seamlessly.

**SAFETY: Steep Angle Calculations**

At steep shooting angles, the "rifleman's rule" (use the horizontal distance for drop estimation) is an approximation. BALLISTICS-ENGINE performs a full 3D trajectory integration that accounts for the actual gravity vector relative to the bore line. However, real-world factors—particularly shifting barrel harmonics, scope alignment errors, and rest stability—can introduce significant deviations at steep angles. Always verify steep-angle solutions with actual firing data before relying on them for critical shots.

### 20.7.2 Very Long Range

At extreme distances (1500+ yards), several challenges emerge for the zeroing algorithm:

1. **Large launch angles**: The required launch angle for a 1500-yard .308 Win zero can exceed 5 degrees. The primary $\pm 10$-degree bracket handles this, but the wider bracket may be needed for less efficient cartridges or extreme distances.

2. **Transonic complications**: If the bullet decelerates through the transonic region before reaching the target, the drag curve becomes highly nonlinear, which can cause the height-vs-angle function to develop unusual shapes that slow convergence.

3. **Extended flight time**: Longer trajectories require more integration steps, making each function evaluation in the zeroing loop more expensive. The fast trajectory solver (Section 28.3) mitigates this.

Listing 20.12: Zeroing at extreme range

```
# .338 Lapua Mag at 1500 yards
ballistics zero \
  --velocity 2725 --bc 0.818 --mass 300 --diameter 0.338 \
  --target-distance 1500

# .308 Win at 1200 yards - bullet may be transonic
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 1200
```

### 20.7.3 Subsonic Zeros

Subsonic zeroing presents the most challenging edge case. If the target distance is far enough that the bullet has decelerated below the speed of sound, several things change:

1. The drag coefficient undergoes a rapid change in the transonic regime (roughly Mach 0.8–1.2), creating a region of high nonlinearity in the trajectory.

2. The bullet's stability may degrade as it passes through the transonic regime, potentially destabilizing entirely (see Chapter 17).

3. The height-vs-angle function may develop shallow slopes in the transonic region, slowing the convergence of interpolation-based methods.

Brent's method handles these cases through its automatic fallback to bisection. When the interpolation steps become unreliable due to the function's unusual shape near transonic, the algorithm reverts to the guaranteed-convergent bisection strategy.

**Subsonic Accuracy Limits**

Trajectory predictions for bullets in or beyond the transonic regime carry significantly more uncertainty than supersonic predictions. The drag models (G1, G7) are calibrated primarily for supersonic flight. In the transonic region, projectile-specific drag variations can produce 10–20% differences from the standard model. Treat subsonic trajectory predictions as estimates, not precision data.

### 20.7.4 The Auto-Zero Workflow

For trajectory calculations, the zeroing algorithm is typically invoked transparently via the `--auto-zero` flag, which specifies the zero distance:

Listing 20.13: Auto-zero in trajectory calculations

```
# Zero at 200 yards, compute trajectory to 1000 yards
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 \
  --auto-zero 200 --output table
```

This first runs the zeroing algorithm to find the launch angle for a 200-yard zero, then runs the full trajectory with that angle. The user never sees the intermediate zeroing process—it happens automatically.

### 20.7.5 Powder Temperature Sensitivity

The zeroing algorithm also accounts for powder temperature sensitivity when enabled. The adjusted muzzle velocity is computed before zeroing begins:

Listing 20.14: Muzzle velocity adjustment for temperature

```
pub fn adjusted_muzzle_velocity(
    inputs: &InternalBallisticInputs
) -> f64 {
    let mut mv = inputs.muzzle_velocity;
    if inputs.use_powder_sensitivity {
        mv *= 1.0 + inputs.powder_temp_sensitivity
            * (inputs.temperature - inputs.powder_temp) / 15.0;
    }
    mv
}
```

This means that changing the ambient temperature changes the effective muzzle velocity, which in turn changes the zero angle. For temperature-sensitive loads, the shift can be significant:

Listing 20.15: Temperature effect on zeroing

```
# Cold weather: 20 deg F
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 200 \
  --temperature 20

# Hot weather: 100 deg F
ballistics zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --target-distance 200 \
  --temperature 100
```

### 20.7.6 Quick Drop Estimates

For situations where a full trajectory integration is too slow (such as generating initial guesses for the zeroing bracket), BALLISTICS-ENGINE provides a `quick_drop_estimate()` function:

Listing 20.16: Quick drop estimate for initial bracketing

```
pub fn quick_drop_estimate(
    muzzle_velocity_fps: f64,
    distance_yards: f64,
    _bullet_mass_grains: f64,
    bc: f64,
) -> f64 {
    let time_of_flight = distance_m / mv_mps;
    let drag_factor = 1.0 / bc.max(0.1);
    let velocity_loss = drag_factor * time_of_flight * 0.1;
    let effective_velocity = mv_mps * (1.0 - velocity_loss)
        .max(0.1);
    let adjusted_time = distance_m / effective_velocity;
    0.5 * gravity * adjusted_time * adjusted_time
}
```

This approximation uses a simplified drag model to estimate the time of flight and then applies the standard $\frac{1}{2}gt^2$ formula. It is not accurate enough for final results, but it provides a reasonable estimate for determining which direction the root lies.

## Exercises

1. **Zero angle comparison.** Compute the zero angle for three cartridges at 200 yards: .223 Rem (77 gr SMK, `0.372` G7, 2750 fps), .308 Win (168 gr SMK, `0.462` G7, 2700 fps), and 6.5 Creedmoor (140 gr ELD-M, `0.610` G7, 2710 fps). Which cartridge requires the least launch angle? Why?

```
ballistics zero --velocity 2750 --bc 0.372 --mass 77 \
  --diameter 0.224 --target-distance 200

ballistics zero --velocity 2700 --bc 0.462 --mass 168 \
  --diameter 0.308 --target-distance 200

ballistics zero --velocity 2710 --bc 0.610 --mass 140 \
  --diameter 0.264 --target-distance 200
```

2. **Zero distance sensitivity.** Using the .308 Win load (168 gr SMK, `0.462` G7, 2700 fps), compute the zero angle at 100, 200, 300, and 400 yards. How does the angle change with distance? Is the relationship linear?

3. **Elevated target zeroing.** Zero the .308 Win at 500 yards with target heights of 0, 50, 100, and 200 yards above the shooter. How does the zero angle change? Then use the trajectory command with `--auto-zero` and `--shooting-angle` set to $+15$ and $+30$ degrees and compare the drop at 500 yards.

4. **Temperature shift.** With powder temperature sensitivity enabled (`--use-powder-sensitivity`, sensitivity of 1.5 fps/°F, base temp 70°F), compute the zero angle at 200 yards for temperatures of 0°F, 40°F, 70°F, and 100°F. How many MOA does the zero shift across this temperature range?

5. **Extreme range zeroing.** Attempt to zero the .338 Lapua Mag (300 gr Berger Hybrid, `0.818` G7, 2725 fps) at 2000 yards. Does the zeroing algorithm converge? What launch angle does it find? Then run a trajectory at that angle—does the bullet still have useful energy at 2000 yards?

```
ballistics zero --velocity 2725 --bc 0.818 --mass 300 \
  --diameter 0.338 --target-distance 2000

ballistics trajectory --velocity 2725 --bc 0.818 \
  --mass 300 --diameter 0.338 --drag-model g7 \
  --max-range 2000 --auto-zero 2000 --output table
```

**What's Next** With the numerical methods of Part VI behind us, we shift from the engine room to the network. Part VII explores how BALLISTICS-ENGINE can connect to online services for real-time weather data, ML-enhanced BC corrections, and collaborative trajectory sharing—turning a local CLI tool into a connected ballistics platform.

# Part VII

# Online Mode & Weather

# Chapter 21

# Online Mode

You have spent the previous chapters learning how BALLISTICS-ENGINE computes trajectories locally—reading drag tables from disk, feeding atmospheric parameters through the ICAO model, and integrating the equations of motion on your own machine. That workflow is fast, deterministic, and works entirely offline. But what if you could augment those local calculations with machine-learning–enhanced predictions served from a cloud API, complete with real-time weather zone modeling, altitude-dependent atmospheric corrections, and BC confidence scores derived from doppler-derived drag data?

That is exactly what *online mode* provides. With a single flag—`--online`—the CLI routes your trajectory request through a proprietary Flask API endpoint that layers ML corrections on top of the same physics engine you already know. This chapter explains what online mode does, how to enable it, what data flows over the network, and when the extra accuracy is worth the round-trip latency.

## 21.1 What Online Mode Does

At its core, online mode redirects the trajectory computation from the local solver to a remote API. Rather than calling the RK45 integrator in `src/cli_api.rs` directly, the CLI serializes your inputs into an HTTP request, sends them to the Flask API at `https://api.ballistics.7.62x51mm.sh`, and receives a JSON response containing the computed trajectory, zero angle, time of flight, and—when available—ML-derived corrections.

The API itself runs the same ballistics physics kernel under the hood, but it can apply additional corrections that are impractical to ship in every CLI binary:

- **BC confidence scoring.** The API evaluates how closely the stated BC matches its internal doppler-derived database and returns a confidence score between 0 and 1.

- **ML corrections.** A set of learned adjustments—labeled in the response as `ml_corrections_applied`—that compensate for systematic biases in standard drag models, particularly in the transonic regime.
- **Weather zones.** When latitude, longitude, and shot direction are provided, the API can generate spatially varying atmospheric profiles along the trajectory rather than assuming a single homogeneous atmosphere.
- **3D weather corrections.** Altitude-dependent temperature, pressure, and density adjustments that account for the bullet's changing elevation during flight.

**Online Mode**

*Online mode* is an optional feature-gated capability (`#[cfg(feature = "online")]`) that routes trajectory and velocity-truing calculations through a remote Flask API, augmenting the local physics engine with ML-enhanced predictions, BC confidence scoring, and spatially varying weather models.

The key architectural detail is that online mode is *additive*: you can always fall back to local computation. The `--offline-fallback` flag instructs the CLI to run the trajectory locally if the API is unreachable, and `--compare` runs both engines side by side so you can see exactly where they diverge.

## 21.2 The `--online` Flag

Enabling online mode requires two things: (1) the binary must be compiled with the `online` Cargo feature, and (2) you must pass `--online` on the command line. If the feature is not compiled in, the flag does not exist and the CLI operates purely locally.

Let's start with the most basic online trajectory. We will use our standard .308 Win load:

Listing 21.1: A basic online trajectory computation

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --online
```

The first time you run an online command, the CLI will prompt you to accept the Terms of Service. This is a one-time step; acceptance is stored in `~/.ballistics/tos.json` and checked on subsequent runs.

**Terms of Service**

Online mode sends your ballistic parameters to a proprietary cloud service. Before the first API call, the CLI fetches the Terms of Service from `https://ballistics.rs/terms.txt` and asks you to accept. Your acceptance, along with a stable FNV-1a hash of the terms text, is recorded locally. If the terms are updated server-side, you will be prompted again.

### 21.2.1 Feature Gating in the Source

In the Rust source, every online-related type and function is wrapped in `#[cfg(feature = "online")]`. The relevant import block in `src/main.rs` shows this clearly:

Listing 21.2: Feature-gated imports for online mode

```
#[cfg(feature = "online")]
use ballistics_engine::api_client::{
    ApiClient,
    TrajectoryRequestBuilder,
    TrueVelocityRequest,
};
#[cfg(feature = "online")]
use ballistics_engine::bc_table_download::Bc5dDownloader;
```

This means the `ureq` HTTP client and all serialization logic are completely absent from offline builds, keeping the binary lean. If you installed via a pre-built release binary, online support may or may not be included depending on the distribution. You can verify by running `ballistics trajectory --help` and looking for the `--online` flag.

### 21.2.2 Companion Flags

The `--online` flag unlocks several companion flags:

| Flag | Description | Default |
|---|---|---|
| `--online` | Route calculation through the Flask API | off |
| `--api-url` | API endpoint URL | `https://api.bal` |
| `--api-timeout` | Request timeout in seconds | 10 |
| `--offline-fallback` | Fall back to local solver on API failure | off |
| `--compare` | Run both local and API, display side by side | off |
| `--enable-weather-zones` | Generate weather zones along trajectory | off |
| `--enable-3d-weather` | Altitude-dependent atmospheric corrections | off |
| `--wind-shear-model` | Wind shear model: `none`, `logarithmic`, `power_law`, `ekman_spiral` | `logarithmic` |

| Flag | Description | Default |
|---|---|---|
| `--weather-zone-interpolation` | Interpolation: `linear`, `cubic`, `step` | `linear` |

**Sensible Defaults for Online Mode**

For most range-day use, `--online --offline-fallback` is the right combination. You get ML-enhanced predictions when the network is available and graceful degradation when it is not. Add `--compare` during your first few sessions to build confidence in the API results before relying on them exclusively.

Here is a more complete invocation showing several companion flags:

Listing 21.3: Online mode with fallback and weather zones

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 85 --pressure 29.45 --humidity 65 \
  --altitude 4500 --wind-speed 8 --wind-direction 90 \
  --latitude 39.86 --longitude -104.67 \
  --shot-direction 180 \
  --online --offline-fallback \
  --enable-weather-zones --enable-3d-weather \
  --api-timeout 15
```

This command fires a .308 Win, 168 gr Sierra MatchKing (G7 BC `0.462`, 2700 fps) at a target 1000 yards downrange from an altitude of 4500 feet near Denver, Colorado, with an 8 mph crosswind from the east. Online mode is enabled with a 15-second timeout and automatic local fallback. Weather zones and 3D weather corrections are both active, using the shooter's GPS coordinates and shot bearing.

## 21.3 Weather Data Sources and APIs

### 21.3.1 The Flask API Architecture

The remote endpoint that powers online mode is a Flask-based web service accessible at the default URL shown above. The API communicates over HTTPS using standard query parameters for `GET` requests (trajectory calculations) and JSON bodies for `POST` requests (velocity truing).

The `ApiClient` struct in `src/api_client.rs` manages all HTTP communication. Its constructor normalizes the base URL and sets a configurable timeout:

Listing 21.4: ApiClient construction (from `src/api_client.rs`)

```
pub struct ApiClient {
    base_url: String,
    timeout: Duration,
}

impl ApiClient {
    pub fn new(base_url: &str, timeout_secs: u64) -> Self {
        let base_url = base_url
            .trim_end_matches('/')
            .to_string();
        Self {
            base_url,
            timeout: Duration::from_secs(timeout_secs),
        }
    }
}
```

Every outgoing request carries a `User-Agent` header identifying the CLI version (`ballistics-cli/{version}`), which allows the API to serve version-appropriate responses.

### 21.3.2 API Endpoints

The Flask API exposes three endpoints relevant to the CLI:

1. **`GET /v1/calculate`** — Trajectory computation. All ballistic parameters are passed as query-string key–value pairs. The API returns a JSON object containing the trajectory array, summary results, BC confidence, and a list of any ML corrections applied.
2. **`POST /v1/true-velocity`** — Velocity truing. Accepts a JSON body with measured drop, range, BC, and atmospheric conditions. Returns the effective muzzle velocity, convergence data, and a confidence rating.
3. **`GET /health`** — Health check. Returns HTTP 200 if the API is operational. The CLI uses this endpoint with a 5-second timeout to verify connectivity.

**Unit Conversions at the Boundary**

The CLI works internally in metric (m, kg, m/s, hPa, °C), but the Flask API expects imperial units—fps, grains, yards, °F, and inHg. The `calculate_trajectory` method in `src/api_client.rs` performs all necessary conversions before sending the request and converts the response back to metric upon receipt. You never need to worry about unit mismatches between the two systems.

### 21.3.3 Error Handling

The `ApiError` enum in `src/api_client.rs` captures five failure modes:

Listing 21.5: API error types

```
pub enum ApiError {
    NetworkError(String),
    Timeout,
    InvalidResponse(String),
    ServerError(u16, String),
    RequestError(String),
}
```

When the CLI encounters an `ApiError`, its behavior depends on the `--offline-fallback` flag:

- **With `--offline-fallback`:** A warning is printed to `stderr` and the trajectory is computed locally. This is the recommended configuration for field use.
- **Without `--offline-fallback`:** The CLI prints the error message and exits with a non-zero status code. A hint reminds the user that `--offline-fallback` exists.

Listing 21.6: Graceful fallback on API failure

```
# With fallback enabled, network errors produce a warning
# and local computation proceeds automatically
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --online --offline-fallback
# Warning: API request failed: Network error: connection refused
# Falling back to local calculation...
```

**Network Latency and Timeouts**

The default API timeout is 10 seconds. In areas with poor cellular reception (common at remote ranges), consider increasing it with `--api-timeout 30` or using `--offline-fallback` to avoid stalled computations. A health check against `/health` with a 5-second timeout runs internally before the main request when appropriate.

## 21.4 Location Specification: `--lat` / `--lon`

Several online features—weather zones, 3D weather, and Coriolis corrections—require the shooter's geographic position. The CLI accepts this through three flags:

- `--latitude`: Decimal degrees, −90 to +90. Negative values represent the Southern Hemisphere.
- `--longitude`: Decimal degrees, −180 to +180. Negative values represent the Western Hemisphere.
- `--shot-direction`: Azimuth in degrees from true north (0° = north, 90° = east, 180° = south, 270° = west).

Listing 21.7: Specifying shooter location for weather zones

```
ballistics trajectory \
  --velocity 2710 --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --max-range 1200 --auto-zero 100 \
  --latitude 45.35 --longitude -111.20 \
  --shot-direction 270 \
  --online --enable-weather-zones
```

This places the shooter near Bozeman, Montana, firing west with a 6.5 Creedmoor, 140 gr ELD-M load (G7 BC `0.610`, 2710 fps).

### 21.4.1 How Location Flows Through the Request

When the CLI builds the API request, latitude and longitude are attached as optional query parameters. Inside `src/main.rs`, the location fields are set after the main `TrajectoryRequestBuilder` chain:

Listing 21.8: Attaching location to the API request

```
let mut request = api_request;
if let Some(lat) = latitude {
    request.latitude = Some(lat);
}
if let Some(lon) = longitude {
    request.longitude = Some(lon);
}
if let Some(dir) = shot_direction {
    request.shot_direction = Some(dir);
}
```

On the API side, these coordinates serve two purposes. First, they feed the Coriolis and Eötvös effect calculations (see Chapter 15 for the physics). Second—and more relevant to this chapter—they enable the weather zone system, which partitions the trajectory into spatial segments with distinct atmospheric conditions.

**Use Your Phone's GPS**

Most smartphones display GPS coordinates in the compass or maps app. Copy the decimal-degree values directly into the CLI flags. For repeatable range sessions, store the coordinates in a location CSV file and load them with `--location range_sites.csv --site KF_LR` (see Chapter 8).

### 21.4.2 Location from Profiles

If you use profile CSV files (described in Chapter 8), location data can live in a separate locations CSV. The CLI merges these data sources:

Listing 21.9: Loading location from a CSV profile

```
ballistics trajectory \
  --saved-profile "308_smk_168" \
  --location range_sites.csv --site "BRC_1000" \
  --online --enable-weather-zones --enable-3d-weather
```

The location CSV might contain columns like `LATITUDE`, `LONGITUDE`, `ALTITUDE`, `TEMPERATURE`, `PRESSURE`, and `HUMIDITY`, all of which feed directly into both the local atmospheric model and the API request.

## 21.5 What Data Is Fetched

Understanding what goes over the wire is important—both for accuracy and for privacy. Let's look at the complete request and response structures.

### 21.5.1 The Outbound Request

The `TrajectoryRequest` struct in `src/api_client.rs` defines every field that can be sent to the API. The required fields are:

- BC value and type (G1 or G7)
- Bullet mass (grams, converted from grains at the boundary)
- Muzzle velocity (m/s, converted to fps at the boundary)
- Target distance (meters, converted to yards at the boundary)

All remaining fields are optional and serialized only when present (using Serde's `skip_serializing_if = "Option::is_none"` attribute). The optional fields include:

| Field | Sent As | Purpose |
|---|---|---|
| `zero_range` | yards | Zero distance for angle computation |
| `wind_speed` | mph | Surface wind speed |
| `wind_angle` | degrees | Wind direction |
| `temperature` | °F | Ambient temperature |
| `pressure` | inHg | Barometric pressure |
| `humidity` | % | Relative humidity |
| `altitude` | feet | Shooter altitude |
| `latitude` | degrees | Shooter latitude |
| `longitude` | degrees | Shooter longitude |
| `shot_direction` | degrees | Shot azimuth from true north |
| `shooting_angle` | degrees | Inclination angle |
| `twist_rate` | inches/turn | Barrel twist rate |
| `bullet_diameter` | inches | Bullet caliber |
| `bullet_length` | inches | Bullet length |
| `enable_weather_zones` | bool | Weather zone generation |
| `enable_3d_weather` | bool | 3D atmospheric corrections |
| `wind_shear_model` | string | Wind shear algorithm |
| `weather_zone_interpolation` | string | Zone interpolation method |
| `sample_interval` | yards | Trajectory step interval |

**Privacy Consideration**

Online mode transmits your ballistic parameters and GPS coordinates to a remote server. No personally identifiable information is sent beyond what the `User-Agent` header reveals (CLI version). However, the combination of precise coordinates and shot parameters could in principle identify a shooting location. If this is a concern, omit `--latitude` and `--longitude` (weather zones will be unavailable) or use approximate coordinates.

### 21.5.2 The Inbound Response

The API returns a JSON object that the CLI deserializes into a `TrajectoryResponse`:

Listing 21.10: TrajectoryResponse structure (from `src/api_client.rs`)

```
pub struct TrajectoryResponse {
    pub trajectory: Vec<ApiTrajectoryPoint>,
    pub zero_angle: f64,
    pub time_of_flight: f64,
    pub bc_confidence: Option<f64>,
    pub ml_corrections_applied: Option<Vec<String>>,
```

```
    pub max_ordinate: Option<f64>,
    pub impact_velocity: Option<f64>,
    pub impact_energy: Option<f64>,
}
```

Each `ApiTrajectoryPoint` contains range, drop, wind drift, velocity, energy, and time of flight—the same columns you see in a local trajectory table. The additional fields are what make online mode valuable:

**`bc_confidence`** A score from 0 to 1 indicating how well the stated BC matches the API's internal doppler-derived database. A score above 0.9 indicates strong agreement; below 0.7 suggests the BC may benefit from truing.

**`ml_corrections_applied`** A list of strings naming the corrections that were applied. These might include transonic drag adjustment, form factor refinement, or altitude-density corrections.

**`max_ordinate`** The peak trajectory height above the line of sight, in inches (converted to meters internally).

**`impact_velocity`** and `impact_energy`: Terminal values at the final trajectory point.

### 21.5.3 Unit Conversion at the Boundary

A particularly important implementation detail is that all unit conversions happen inside the `calculate_trajectory` method of `ApiClient`. The CLI works in metric internally. The Flask API expects imperial. The conversion code in `src/api_client.rs` handles this transparently:

Listing 21.11: Metric-to-imperial conversion before API call

```
let velocity_fps = request.muzzle_velocity / 0.3048;
let mass_grains = request.bullet_mass / 0.0647989;
let distance_yards = request.target_distance / 0.9144;
```

And in the response, the trajectory points are converted back:

Listing 21.12: Imperial-to-metric conversion of API response

```
Some(ApiTrajectoryPoint {
    range: range_yards * 0.9144,
    drop: drop_inches * 0.0254,
    drift: drift_inches * 0.0254,
    velocity: velocity_fps * 0.3048,
    energy: energy_ftlbs * 1.35582,
    time,
})
```

This design means that regardless of whether you pass `--units imperial` or `--units metric` on the command line, the API communication always uses the same unit system, and the user-facing display is handled by the `UnitConverter` module independently.

## 21.6 Comparing Offline vs. Online Predictions

The `--compare` flag is one of the most useful tools for understanding what online mode actually does to your predictions. It runs both the local solver and the API, then displays the results in a side-by-side table.

Listing 21.13: Comparing local and API results

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 72 --pressure 29.15 --humidity 55 \
  --altitude 5200 --wind-speed 10 --wind-direction 90 \
  --online --compare
```

This produces a comparison table like the following:

Listing 21.14: Sample compare output

```
# Running comparison between local and API calculations...
#
# +=======================================+
# |     COMPARISON: LOCAL vs API          |
# +=======================================+
# | ML Corrections Applied:               |
# |   - transonic_drag_refinement         |
# |   - altitude_density_correction       |
# +=======================================+
# | BC Confidence:       92.3%            |
# +=======================================+
# | Range yd | API Drop |Local Drop|  D Drop|
# +==========+==========+==========+========+
# |    100.0 |     0.00 |     0.00 |  +0.00 |
# |    200.0 |    -3.42 |    -3.38 |  -0.04 |
# |    300.0 |   -12.21 |   -12.10 |  -0.11 |
# |    400.0 |   -27.58 |   -27.35 |  -0.23 |
# |    500.0 |   -50.89 |   -50.47 |  -0.42 |
# |    600.0 |   -83.74 |   -82.99 |  -0.75 |
# |    700.0 |  -128.05 |  -126.80 |  -1.25 |
# |    800.0 |  -186.14 |  -184.13 |  -2.01 |
```

```
# |    900.0 |  -260.87 |  -257.72 |  -3.15 |
# |   1000.0 |  -355.91 |  -351.10 |  -4.81 |
# +=========================================+
```

Several things are worth noting in this output.

**First**, the ML corrections are listed explicitly. You can see exactly what the API adjusted. Common corrections include:

- `transonic_drag_refinement` — adjusts the drag curve in the Mach 0.8–1.2 regime
- `altitude_density_correction` — refines the air density model for high-altitude shooting
- `bc_form_factor_adjustment` — corrects the BC based on the bullet's form factor relative to the standard projectile

**Second**, the delta column shows how much the API diverges from local. In the example above, the difference is under 1 inch at 600 yards and grows to about 4.8 inches at 1000 yards. At extended range, even small improvements to the drag model compound. Whether 4.8 inches matters depends on your application—for a 10-inch steel plate at 1000 yards, it's the difference between a center hit and an edge hit.

**Third**, the BC confidence of 92.3% tells us that the stated BC of `0.462` for the 168 gr SMK is a good match to the API's internal doppler-derived data. Had this number been below 70%, it would be a strong signal to true your BC using field data (see Chapter 7).

### 21.6.1 When to Use Compare Mode

We recommend running `--compare` in three situations:

1. **When first adopting online mode.** Run compare against your known loads to build confidence in the API's predictions.
2. **After truing your BC.** See whether your trued BC produces results that match the API more closely than the factory BC.
3. **In extreme conditions.** High altitude, extreme temperatures, or very long range are where the ML corrections provide the most value. Compare mode lets you quantify that value.

**Compare Mode Is Slower**

Compare mode runs two complete trajectory computations: one via the API and one locally. Expect roughly double the wall-clock time of a single computation. On a fast internet connection, the overhead is dominated by the API round trip (typically 200–500 ms).

### 21.6.2 Interpreting Deltas

A positive delta ($\Delta > 0$) means the API predicts more drop than local. This is typical when ML corrections add drag in the transonic regime that the standard G7 table underestimates. A negative delta means the API predicts less drop—possibly because it applies a higher effective BC based on its doppler-derived data.

Neither result is inherently "right" without ground truth. The correct workflow is:

1. Shoot at known distance and measure actual drop.
2. Compare measured drop to both local and API predictions.
3. Adopt whichever matches reality more closely—or use BC truing (Chapter 7) to calibrate the local solver.

**SAFETY: Verify Against Real-World Data**

Online mode's ML corrections are derived from statistical models, not physical laws. They can improve accuracy for loads that are well-represented in the training data and degrade it for unusual combinations. Always verify computational predictions—whether local or online—against real-world chronograph and range data before relying on them for critical applications.

## 21.7 Velocity Truing via the API

Online mode also extends the `true-velocity` command. When you provide measured drop data from the field, the CLI can use the API to calculate an effective muzzle velocity that reconciles your observations with the ML-enhanced trajectory model.

Listing 21.15: Online velocity truing

```
ballistics true-velocity \
  --measured-drop 8.2 --range 700 \
  --bc 0.462 --drag-model g7 \
  --mass 168 --diameter 0.308 \
  --chrono-velocity 2700 \
  --altitude 5200 --temperature 72 --pressure 29.15 \
  --online
```

The API endpoint `POST /v1/true-velocity` accepts the `TrueVelocityRequest` structure, which includes the measured drop in MILs, range in yards, stated BC, atmospheric conditions, and an optional chronograph velocity for comparison.

The response includes:

- **Effective velocity** — the muzzle velocity that produces the observed drop at the stated range.
- **Velocity adjustment** — the difference between the effective velocity and the chronograph reading.
- **Confidence** — "high", "medium", or "low" based on convergence behavior.
- **Iterations** — how many solver iterations were needed to converge.
- **Final error** — residual error in MILs after convergence.

As with trajectory calculations, `--offline-fallback` causes the CLI to run velocity truing locally if the API is unreachable.

## 21.8 BC Table Downloads

Online mode also enables automatic downloading of caliber-specific BC5D correction tables. These precomputed binary tables provide velocity-dependent BC corrections without requiring a live API connection for every shot—think of them as a cached snapshot of the API's ML corrections for a given caliber.

Three flags control this behavior:

- `--bc-table-auto`: Automatically download the appropriate BC5D table when one is not found locally.
- `--bc-table-url`: The base URL for table downloads (default: `https://ballistics.tools/downloads/bc5d`).
- `--bc-table-refresh`: Force re-download even if a cached table exists.

Listing 21.16: Auto-downloading BC5D tables for offline use

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --online --bc-table-auto
```

Once downloaded, the tables are cached locally and can be used in subsequent offline sessions via the `--bc-table-dir` flag—giving you ML-enhanced BC corrections without an active internet connection.

**Pre-Cache Before a Match**

Before heading to a competition where cell service is unreliable, run your load with `--online --bc-table-auto` to download the BC5D tables. Then at the match, you can use `--bc-table-dir` to apply the cached corrections without needing the network. See Chapter 24 for a complete competition workflow.

## 21.9 Under the Hood: Request Lifecycle

To tie everything together, let's walk through the complete lifecycle of an online trajectory request.

1. **TOS verification.** The CLI checks `~/.ballistics/tos.json` for a valid acceptance record matching the current TOS version (`2026-01-26`). If missing or outdated, the TOS text is fetched from `https://ballistics.rs/terms.txt`, displayed, and the user is prompted.

2. **Input parsing.** The Clap argument parser extracts all flags, merges them with profile/location CSV data if present, applies unit conversions, and resolves BC truing adjustments.

3. **Request construction.** The `TrajectoryRequestBuilder` assembles the API request. Required fields (BC, mass, velocity, distance) are validated. Optional fields are attached only when non-`None`.

4. **HTTP call.** The `ureq` library sends a `GET /v1/calculate` request with all parameters encoded as query strings. The request includes an `Accept: application/json` header and the CLI version in `User-Agent`.

5. **Response parsing.** The JSON response is deserialized through a two-stage process. First, it is parsed into a generic `serde_json::Value`. Then, the `convert_api_response` method extracts trajectory points from the `trajectory` array and summary values from the `results` object, performing imperial-to-metric conversions along the way.

6. **Display.** The trajectory is displayed according to the `--output` format (table, JSON, or CSV). If `--compare` is active, a local computation runs in parallel and the results are merged into a comparison table.

7. **Fallback.** If any step from 4 onward fails and `--offline-fallback` is set, the CLI silently falls back to the local solver and prints a warning.

## 21.10 Practical Considerations

### 21.10.1 When Online Mode Is Worth It

Online mode adds network latency and requires an internet connection. Use it when:

- You are shooting beyond 600 yards and want the most accurate drop prediction available.
- You are shooting at high altitude (above 5000 feet) where standard atmospheric models diverge from reality.
- You are using weather zones to model spatially varying conditions along a long-range trajectory.
- You want a BC confidence score to validate your stated BC.
- You are truing your velocity or BC and want the ML-enhanced solver's opinion alongside the local result.

Conversely, skip online mode when:

- You are shooting under 400 yards where local predictions are already highly accurate.
- Latency matters (e.g., you need sub-second response time in a scripted workflow).
- You have no network connectivity and have not pre-cached BC5D tables.
- Privacy is a concern and you prefer not to transmit your shooting data.

### 21.10.2 Combining Online Mode with Profiles

The most ergonomic workflow combines saved profiles with online mode. Store your load data in a saved profile, your range locations in a CSV, and invoke everything together:

Listing 21.17: Full online workflow with profiles

```
ballistics trajectory \
  --saved-profile "65CM_ELDM_140" \
  --location ranges.csv --site "Blue_Ridge_1K" \
  --online --offline-fallback \
  --enable-weather-zones --enable-3d-weather \
  --compare
```

This loads the 6.5 Creedmoor profile (BC, velocity, mass, diameter, twist rate, zero distance), the Blue Ridge range site (altitude, temperature, pressure, humidity, latitude, longitude), enables weather zones and 3D weather, and runs the comparison between local and API.

## 21.11 Exercises

1. Run a .308 Win trajectory (168 gr SMK, BC `0.462`, 2700 fps, G7 drag model) to 1000 yards with `--online` and `--compare`. Note the BC confidence score and the delta at 1000 yards.

2. Repeat the same trajectory with `--enable-weather-zones` and a set of GPS coordinates for a range you frequent. How does the weather zone model change the predicted drop compared to the homogeneous atmosphere?
3. Run the trajectory from Exercise 1 with `--offline-fallback` while disconnected from the internet. Verify that the CLI falls back to local computation and produces output.
4. Use the `true-velocity` command in online mode with a measured drop of 7.5 MIL at 600 yards for the same .308 Win load. Compare the effective velocity to your chronograph reading. Does the API-derived velocity differ from the local truing result?
5. Pre-cache the BC5D tables with `--bc-table-auto` and then run the trajectory again using only `--bc-table-dir` (no `--online`). Compare the results to the full online trajectory from Exercise 1.

**What's Next.** We have seen how online mode routes calculations through a remote API and what data flows over the wire. But we have only scratched the surface of the atmospheric modeling that makes weather zones and 3D corrections possible. In Chapter 22, we dive deep into the weather data pipeline—from METAR station reports and atmospheric profiles to temperature lapse rates and humidity-corrected air density—and show how these measurements propagate through the ballistic model to produce more accurate trajectories on range day.

# Chapter 22

# Weather Integration

A bullet does not fly through a vacuum. It plows through a column of air whose density, temperature, pressure, and moisture content determine how quickly it decelerates and how far it drifts. In Chapter 9 we introduced the ICAO Standard Atmosphere and showed how the `--temp`, `--pressure`, and `--humidity` flags let you specify local conditions. But those flags assume a *single, uniform* atmosphere—the same temperature and pressure from muzzle to target.

Reality is more nuanced. Temperature falls as the bullet climbs (and recovers as it descends). Pressure changes with altitude. Humidity alters both air density and the speed of sound. A shooter perched on a mesa at 7000 feet firing across a valley will see conditions that vary dramatically along the trajectory. This chapter dives into the weather data pipeline inside BALLISTICS-ENGINE—from the atmospheric physics in `src/atmosphere.rs` to the wind shear models in `src/wind_shear.rs`—and shows how to put real weather data to work on range day.

## 22.1 Weather Station Data and METAR Decoding

### 22.1.1 What Is a METAR?

Aviation weather reports—known as METARs (Meteorological Aerodrome Reports)—are the gold standard for ground-truth atmospheric data. Every airport and many smaller weather stations around the world broadcast a METAR at least once per hour. A typical METAR looks like this:

Listing 22.1: A raw METAR report

```
# METAR KDEN 021753Z 18012G18KT 10SM FEW080 SCT200
#   32/08 A2992 RMK AO2 SLP082 T03170083
```

Decoded, this tells us:

- **Station**: KDEN (Denver International Airport)
- **Time**: 021753Z (2nd day of the month, 17:53 UTC)
- **Wind**: from 180° (south) at 12 knots, gusting to 18
- **Visibility**: 10 statute miles
- **Sky**: few clouds at 8000 feet, scattered at 20,000 feet
- **Temperature / Dew point**: 32°C / 8°C
- **Altimeter setting**: 29.92 inHg
- **Sea-level pressure**: 1008.2 hPa

For ballistic purposes, the critical fields are temperature, altimeter setting (barometric pressure), and wind. Dew point lets us derive relative humidity, and the altimeter setting can be converted to station pressure at the known field elevation.

**Finding Your Nearest METAR**

The Aviation Weather Center at `aviationweather.gov` provides real-time METARs. Search by the four-letter ICAO code of your nearest airport. Many handheld weather meters (Kestrel 5700, for example) can also output METAR-equivalent data directly.

### 22.1.2 Converting METAR Data to CLI Flags

Once you have the METAR, translate it into BALLISTICS-ENGINE flags:

Listing 22.2: Using METAR data for a 6.5 Creedmoor trajectory

```
# From METAR KDEN: temp 32C (90F), pressure 29.92 inHg,
# wind from 180 at 12kt (14 mph), field elevation 5431 ft
ballistics trajectory \
  --velocity 2710 --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --max-range 1200 --auto-zero 100 \
  --temperature 90 --pressure 29.92 --humidity 18 \
  --altitude 5431 \
  --wind-speed 14 --wind-direction 180
```

**Station Pressure vs. Altimeter Setting**

The "altimeter setting" in a METAR (`A2992`) is barometric pressure corrected to mean sea level. This is the value the `--pressure` flag expects when you use imperial units. The engine then combines this with your `--altitude` to derive the actual station pressure at your elevation. If you are using a handheld weather meter that reports *station pressure* (uncorrected), you should convert it to sea-level equivalent or set `--altitude` to zero and use the raw station pressure directly.

### 22.1.3 Deriving Humidity from Dew Point

METARs report dew point rather than relative humidity. The conversion uses the August–Roche–Magnus approximation:

$$\mathrm{RH} = 100 \times \frac{\exp\left(\frac{17.625\,T_d}{243.04 + T_d}\right)}{\exp\left(\frac{17.625\,T}{243.04 + T}\right)} \tag{22.1}$$

where $T$ is temperature and $T_d$ is dew point, both in °C. For the Denver METAR above ($T = 32$°C, $T_d = 8$°C):

$$\mathrm{RH} \approx 100 \times \frac{e^{17.625 \times 8 / 251.04}}{e^{17.625 \times 32 / 275.04}} \approx 22\%$$

At 22% relative humidity and 90°F, the humidity correction to air density is small—roughly 0.3%. But in tropical conditions (85°F at 90% RH), the effect reaches 1–2% of air density, which translates to measurable differences in drop at long range.

## 22.2 Atmospheric Profiles Along the Trajectory

### 22.2.1 The Homogeneous Atmosphere Assumption

When you specify `--temp`, `--pressure`, and `--altitude`, the engine computes a single air density and speed of sound, then holds those values constant for the entire trajectory. This is the `calculate_atmosphere` function in `src/atmosphere.rs`:

Listing 22.3: Simplified atmosphere calculation

```
pub fn calculate_atmosphere(
    altitude_m: f64,
    temp_override_c: Option<f64>,
```

```
    press_override_hpa: Option<f64>,
    humidity_percent: f64,
) -> (f64, f64) {  // Returns (density, speed_of_sound)
    let (temp_k, pressure_pa) = if temp_override_c.is_some()
        && press_override_hpa.is_some()
    {
        // Both overrides: use user values directly
        (
            temp_override_c.unwrap() + 273.15,
            press_override_hpa.unwrap() * 100.0,
        )
    } else {
        // Fall through to ICAO standard atmosphere
        let (std_temp, std_press) =
            calculate_icao_standard_atmosphere(altitude_m);
        // ...override individual values if provided
    };
    // Calculate density and speed of sound...
}
```

For shots under 600 yards on flat terrain, this homogeneous model is perfectly adequate. The bullet never climbs more than a few meters above its launch altitude, so the atmospheric conditions do not change appreciably.

### 22.2.2 When Homogeneity Breaks Down

Consider a .338 Lapua Magnum, 300 gr Berger Hybrid (G7 BC `0.818`, 2725 fps) fired at a target 2000 yards away. At that distance, the bullet's arc peaks roughly 25–30 feet above the line of departure. At high altitude, the arc can be even higher. The conditions at the top of the arc differ from those at the muzzle:

- Temperature drops by approximately 3.6°F per 1000 feet of altitude gain (the standard lapse rate). A 30-foot climb reduces temperature by about 0.1°F—negligible.
- But if the trajectory crosses a terrain feature—firing *across* a canyon, for instance—the bullet might descend 500 feet below the shooter before climbing back up to the target. A 500-foot drop increases temperature by 1.8°F and pressure by roughly 0.5 inHg, reducing drag and changing the prediction by several inches at extreme range.

This is where online mode's `--enable-3d-weather` and `--enable-weather-zones` become valuable: they tell the API to partition the trajectory into segments with altitude-dependent atmospheric conditions rather than assuming a single homogeneous column.

Listing 22.4: Enabling 3D weather for a long-range .338 LM shot

```
ballistics trajectory \
  --velocity 2725 --bc 0.818 --mass 300 --diameter 0.338 \
  --drag-model g7 --max-range 2000 --auto-zero 200 \
  --temperature 55 --pressure 29.42 --humidity 40 \
  --altitude 7200 \
  --latitude 36.23 --longitude -112.05 \
  --shot-direction 90 \
  --online --enable-3d-weather --enable-weather-zones
```

## 22.3 Temperature Lapse Rates

The temperature lapse rate—the rate at which temperature changes with altitude—is the single most important atmospheric variable for ballistic accuracy at extreme range. It determines air density at every point along the trajectory, and air density is the dominant factor in drag.

### 22.3.1 The ICAO Standard Lapse Rate

The ICAO Standard Atmosphere divides the atmosphere into layers, each with a defined lapse rate. The implementation in `src/atmosphere.rs` encodes these layers as an array of `AtmosphereLayer` structs:

Listing 22.5: ICAO layer definitions (from `src/atmosphere.rs`)

```
const ICAO_LAYERS: &[AtmosphereLayer] = &[
    // Troposphere (0 - 11 km)
    AtmosphereLayer {
        base_altitude: 0.0,
        base_temperature: 288.15,  // 15 C
        base_pressure: 101325.0,   // 1013.25 hPa
        lapse_rate: -0.0065,       // -6.5 K/km
    },
    // Tropopause (11 - 20 km)
    AtmosphereLayer {
        base_altitude: 11000.0,
        base_temperature: 216.65,  // -56.5 C
        base_pressure: 22632.1,
        lapse_rate: 0.0,           // Isothermal
    },
    // Stratosphere 1 (20 - 32 km)
    AtmosphereLayer {
        base_altitude: 20000.0,
        base_temperature: 216.65,
```

```
        base_pressure: 5474.89,
        lapse_rate: 0.001,         // +1 K/km
    },
    // ... additional layers up to 84 km
];
```

For practically all small-arms ballistics, we operate in the troposphere—the lowest layer, from sea level to approximately 36,000 feet (11 km). In this layer, the standard lapse rate is $-6.5$ K per kilometer, or equivalently:

$$\Gamma_{\text{std}} = -6.5 \frac{\text{K}}{\text{km}} = -3.56 \frac{^\circ\text{F}}{1000 \text{ ft}} \tag{22.2}$$

**Temperature Lapse Rate**

The *temperature lapse rate* $\Gamma$ is the rate of change of atmospheric temperature with altitude. A negative lapse rate means temperature decreases with altitude (as in the troposphere). The ICAO standard tropospheric lapse rate is $-6.5\,\text{K/km}$, but actual conditions can vary from $-3$ K/km (stable inversion) to $-10$ K/km (unstable convection).

### 22.3.2 Actual vs. Standard Lapse Rates

The standard lapse rate is an annual, global average. On any given day at any given location, the actual lapse rate can differ substantially:

- **Temperature inversions** ($\Gamma > 0$): Common on calm, clear nights when the ground cools by radiation. Temperature *increases* with altitude, trapping cold dense air near the surface. The bullet encounters denser air initially, then thinner air as it climbs.
- **Superadiabatic lapse rates** ($\Gamma < -9.8$ K/km): Occur over hot surfaces (desert, tarmac) on sunny days. Temperature drops faster than the adiabatic rate, creating convective instability.
- **Frontal zones**: A weather front can produce lapse rate discontinuities within a few hundred feet of altitude, potentially within the trajectory's envelope.

The `determine_local_lapse_rate` function in `src/atmosphere.rs` selects the appropriate ICAO lapse rate based on altitude:

Listing 22.6: Altitude-dependent lapse rate selection

```
fn determine_local_lapse_rate(altitude_m: f64) -> f64 {
    let layer = ICAO_LAYERS
        .iter()
        .rev()
```

```
        .find(|layer| altitude_m >= layer.base_altitude)
        .unwrap_or(&ICAO_LAYERS[0]);
    layer.lapse_rate
}
```

While this correctly transitions between ICAO layers, it cannot account for local deviations from the standard profile. For that, you either need the 3D weather corrections available through online mode or manual specification of conditions at different altitudes.

### 22.3.3 Quantifying Lapse Rate Impact

Let's quantify how a non-standard lapse rate affects a long-range shot. Consider our .308 Win load (168 gr SMK, BC 0.462, 2700 fps) fired from 7000 feet elevation. At the top of its arc to a target at 1000 yards, the bullet is roughly 10–12 feet above the line of departure.

With the standard lapse rate, the temperature at the arc's peak is about 0.04°F cooler than at the muzzle—truly negligible. But if the shooter is 500 feet above the target (common in mountain shooting), the bullet descends through 500 feet of atmosphere. Over that 500-foot drop, the standard lapse rate produces a temperature increase of 1.8°F and a pressure increase of about 0.5 inHg. Let's compare:

Listing 22.7: Standard conditions at 7000 ft

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 55 --pressure 23.09 --humidity 30 \
  --altitude 7000
```

Listing 22.8: Conditions adjusted for 500 ft descent to target

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 57 --pressure 23.59 --humidity 30 \
  --altitude 6500
```

Running both and comparing the drop at 1000 yards shows a difference of roughly 1–2 inches. For a 10-inch plate, that's within the target. But scale the distance to 1500 yards with a .338 Lapua Magnum, and the same atmospheric gradient produces a 5–8 inch shift—enough to miss a chest-sized target entirely.

**Mountain Shooting and Non-Standard Lapse Rates**

Mountain environments often exhibit temperature inversions in the morning and superadiabatic lapse rates in the afternoon. A single atmospheric measurement at the shooter's position may not represent conditions along the trajectory. When possible, use weather zones or take readings at multiple elevations.

## 22.4 Humidity Effects on Air Density

Counterintuitively, humid air is *less* dense than dry air at the same temperature and pressure. Water vapor (molecular weight 18.015 g/mol) is lighter than the nitrogen (28.013 g/mol) and oxygen (31.998 g/mol) it displaces. Higher humidity means lower air density, which means less drag and slightly higher retained velocity downrange.

### 22.4.1 The Physics: Dalton's Law

Air is a mixture of dry gas and water vapor. By Dalton's law, the total pressure is the sum of partial pressures:

$$P_{\text{total}} = P_{\text{dry}} + P_{\text{vapor}} \tag{22.3}$$

The density of moist air follows from the ideal gas law applied to each component:

$$\rho = \frac{P_{\text{dry}}}{R_d T} + \frac{P_{\text{vapor}}}{R_v T} \tag{22.4}$$

where $R_d = 287.05\ \text{J}/(\text{kg}\cdot\text{K})$ is the specific gas constant for dry air and $R_v = 461.495\ \text{J}/(\text{kg}\cdot\text{K})$ is the specific gas constant for water vapor.

### 22.4.2 Implementation: The Arden Buck Equation

To compute the vapor pressure, BALLISTICS-ENGINE first calculates the *saturation vapor pressure*—the maximum pressure water vapor can exert at a given temperature—using the Arden Buck equation. The implementation in `src/atmosphere.rs` handles both above-freezing and below-freezing cases:

Listing 22.9: Saturation vapor pressure via Arden Buck equation

```
let es_hpa = if temp_c >= 0.0 {
    // Over water
    6.1121 * (18.678 - temp_c / 234.5)
        * (temp_c / (257.14 + temp_c)).exp()
```

```
} else {
    // Over ice
    6.1115 * (23.036 - temp_c / 333.7)
        * (temp_c / (279.82 + temp_c)).exp()
};
```

The actual vapor pressure is then the saturation value scaled by relative humidity:

$$P_{\text{vapor}} = \frac{\text{RH}}{100} \times e_s \tag{22.5}$$

And the dry-air pressure is the remainder:

$$P_{\text{dry}} = P_{\text{total}} - P_{\text{vapor}} \tag{22.6}$$

### 22.4.3 The CIPM Formula for Precise Air Density

For enhanced precision, BALLISTICS-ENGINE also implements the CIPM (Comité International des Poids et Mesures) air density formula in the function `calculate_air_density_cimp` (note: the function name preserves a historical typo from the codebase). This formula accounts for the compressibility factor $Z$ and uses mole-fraction–weighted molecular masses:

$$\rho = \frac{P\, M_a}{Z\, R\, T} \left(1 - x_v \left(1 - \frac{M_v}{M_a}\right)\right) \tag{22.7}$$

where:

- $P$ is total pressure in Pa
- $M_a = 28.96546 \times 10^{-3}$ kg/mol is the molar mass of dry air
- $M_v = 18.01528 \times 10^{-3}$ kg/mol is the molar mass of water vapor
- $R = 8.314472$ J/(mol·K) is the universal gas constant
- $T$ is absolute temperature in K
- $x_v$ is the mole fraction of water vapor ($= P_{\text{vapor}}/P$)
- $Z$ is the compressibility factor (close to 1 at standard conditions)

The compressibility factor $Z$ is computed from enhanced virial coefficients that include second, third, and fourth-order corrections:

Listing 22.10: Compressibility factor calculation

```
fn enhanced_compressibility_factor(
    p: f64, t_k: f64, x_v: f64
) -> f64 {
    let t = t_k - 273.15;
    let p_t = p / t_k;
    let z_second = 1.0 - p_t * (A0 + A1*t + A2*t*t
        + (B0 + B1*t)*x_v + (C0 + C1*t)*x_v*x_v);
    let z_third = p_t * p_t * (D + E*x_v*x_v);
    let z_fourth = p_t*p_t*p_t * (F0 + F1*x_v*x_v*x_v*x_v);
    z_second + z_third + z_fourth
}
```

The saturation vapor pressure in the CIPM path uses the IAPWS-IF97 formulation for higher accuracy than the Arden Buck equation, and an "enhancement factor" accounts for the non-ideal interaction between water vapor and dry air at the prevailing pressure and temperature.

### 22.4.4 Quantifying the Humidity Effect

Let's measure the impact of humidity on a real trajectory. We will use the standard .308 Win load at sea level on a hot day—the conditions where humidity has the largest effect:

Listing 22.11: Dry conditions: 0% humidity

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 95 --pressure 29.92 --humidity 0
```

Listing 22.12: Saturated conditions: 100% humidity

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 95 --pressure 29.92 --humidity 100
```

At 95°F and sea level, the difference in air density between 0% and 100% humidity is approximately 1.1%: from about $1.128\ \mathrm{kg/m^3}$ (dry) to $1.115\ \mathrm{kg/m^3}$ (saturated). This translates to roughly 0.5–1 inch less drop at 600 yards and 2–3 inches at 1000 yards in the humid case.

**Humidity Is the Smallest Atmospheric Factor**

Among temperature, pressure, altitude, and humidity, humidity has the *smallest* effect on trajectory. At moderate temperatures, the difference between 0% and 100% humidity is typically equivalent to a 2–3°F temperature change. Focus on getting temperature and pressure right first; humidity is a refinement.

### 22.4.5 Humidity and the Speed of Sound

Humidity also affects the speed of sound, which matters because the drag coefficient is indexed by Mach number. The `calculate_atmosphere` function applies a two-stage correction:

1. The heat capacity ratio $\gamma$ is reduced for moist air:

$$\gamma_{\text{moist}} = \gamma_{\text{dry}} \times (1 - 0.11\, x_v)$$

   where $x_v$ is the mole fraction of water vapor.

2. The gas constant is increased:

$$R_{\text{moist}} = R_{\text{dry}} \times (1 + 0.6078\, x_v)$$

The base speed of sound is then:

$$a = \sqrt{\gamma_{\text{moist}}\, R_{\text{moist}}\, T} \tag{22.8}$$

with an additional empirical humidity correction factor applied multiplicatively. The net effect is that humid air has a *slightly higher* speed of sound than dry air at the same temperature, which shifts the Mach number slightly and changes which point on the drag curve the solver reads.

## 22.5 Barometric Pressure Corrections

### 22.5.1 Station Pressure vs. Sea-Level Pressure

There are two common ways to express barometric pressure:

- **Station pressure**: The actual pressure at the measurement location. This is what a barometer on your bench reads.
- **Sea-level pressure** (QNH or altimeter setting): The station pressure corrected to mean sea level using the standard lapse rate. This is what METARs report and what the altimeter setting in your aircraft displays.

BALLISTICS-ENGINE expects sea-level pressure when you provide `--pressure` along with a non-zero `--altitude`. The engine then derives station pressure using the barometric formula internally. If you are at sea level, the two values are identical.

### 22.5.2 The Barometric Formula

The relationship between pressure and altitude in a non-isothermal atmosphere (one with a non-zero lapse rate) is:

$$P(h) = P_b \left( \frac{T_b + \Gamma (h - h_b)}{T_b} \right)^{-g/(\Gamma\, R)} \tag{22.9}$$

where $P_b$ and $T_b$ are the base pressure and temperature at altitude $h_b$, $\Gamma$ is the lapse rate, $g = 9.80665$ $\mathrm{m/s^2}$ is gravitational acceleration, and $R = 287.053$ $\mathrm{J/(kg \cdot K)}$ is the specific gas constant for dry air.

For an isothermal layer ($\Gamma = 0$), the formula simplifies to:

$$P(h) = P_b \exp\left( \frac{-g\,(h - h_b)}{R\, T_b} \right) \tag{22.10}$$

The `calculate_icao_standard_atmosphere` function in `src/atmosphere.rs` implements both cases:

Listing 22.13: Barometric formula implementation

```
let pressure = if layer.lapse_rate.abs() < 1e-10 {
    // Isothermal layer
    layer.base_pressure
        * (-G_ACCEL_MPS2 * height_diff
           / (R_AIR * layer.base_temperature)).exp()
} else {
    // Non-isothermal layer
    let temp_ratio = temperature / layer.base_temperature;
    layer.base_pressure * temp_ratio
        .powf(-G_ACCEL_MPS2
              / (layer.lapse_rate * R_AIR))
};
```

### 22.5.3 Practical Pressure Guidelines

A useful rule of thumb: pressure decreases by approximately 1 inHg for every 1000 feet of altitude gain near sea level. The relationship is not perfectly linear (it follows the exponential barometric

formula), but for the altitudes encountered in shooting—sea level to about 10,000 feet—it is close enough for mental math:

| Altitude (ft) | Std. Pressure (inHg) | Std. Pressure (hPa) |
|---|---|---|
| 0 | 29.92 | 1013.25 |
| 2,000 | 27.82 | 942.1 |
| 4,000 | 25.84 | 875.1 |
| 5,000 | 24.90 | 843.1 |
| 6,000 | 23.98 | 812.0 |
| 8,000 | 22.22 | 752.6 |
| 10,000 | 20.58 | 696.8 |

**SAFETY: Altitude and Pressure: Get It Right**

At high altitude, air density can be 20–30% lower than at sea level. This has two consequences: (1) the bullet retains velocity longer, requiring less elevation correction, and (2) if you have worked up a load at sea level, the lower air resistance at altitude changes the barrel harmonics and pressure curve. Always verify your data against chronograph readings at the actual shooting altitude. Never extrapolate sea-level load data to high altitude without real-world confirmation.

### 22.5.4 The `get_local_atmosphere` Function

For the trajectory solver's internal use, the `get_local_atmosphere` function in `src/atmosphere.rs` computes conditions at an arbitrary altitude given a known base point. This powers the altitude-dependent corrections when 3D weather is enabled:

Listing 22.14: Local atmosphere from a base point

```
pub fn get_local_atmosphere(
    altitude_m: f64,
    base_alt: f64,
    base_temp_c: f64,
    base_press_hpa: f64,
    base_ratio: f64,
) -> (f64, f64) {
    let height_diff = altitude_m.round() - base_alt;
    let lapse_rate = determine_local_lapse_rate(altitude_m);
    let temp_c = base_temp_c + lapse_rate * height_diff;
    let temp_k = temp_c + 273.15;
    // ...barometric formula for pressure...
    let density = base_ratio * (base_temp_k * pressure_hpa)
        / (base_press_hpa * temp_k) * 1.225;
```

```
    let speed_of_sound = (temp_k * 401.874).sqrt();
    (density, speed_of_sound)
}
```

The function rounds altitude to the nearest meter (useful for caching in the Python API layer) and selects the ICAO lapse rate appropriate for that altitude. The density is computed as a ratio relative to standard sea-level density ($1.225\ \mathrm{kg/m^3}$), scaled by temperature and pressure differences from the base conditions.

## 22.6 Wind Shear and Altitude-Dependent Wind

We covered wind in Chapter 10 from the shooter's perspective—a single speed and direction at the shooting position. But wind is not constant with altitude. Friction with the ground slows the wind near the surface, while aloft the wind is faster and may come from a different direction. This variation is called *wind shear*, and BALLISTICS-ENGINE provides four models to account for it.

### 22.6.1 Wind Shear Models

The `WindShearModel` enum in `src/wind_shear.rs` defines the available models:

Listing 22.15: Wind shear model types

```
pub enum WindShearModel {
    None,
    Logarithmic,
    PowerLaw,
    EkmanSpiral,
    CustomLayers,
}
```

#### Logarithmic Profile

The logarithmic wind profile describes the wind speed as a function of altitude in the atmospheric boundary layer:

$$U(z) = U_{\text{ref}} \times \frac{\ln(z/z_0)}{\ln(z_{\text{ref}}/z_0)} \tag{22.11}$$

where $z_0$ is the surface roughness length (0.03 m for short grass, the default), $z_{\text{ref}}$ is the reference measurement height (10 m, the standard meteorological height), and $U_{\text{ref}}$ is the measured wind speed at that height.

The implementation handles edge cases carefully—negative altitudes (bullet below the sight line), altitudes below the roughness length (near-zero wind), and the logarithm's singularity at $z = z_0$:

Listing 22.16: Logarithmic profile implementation

```
fn logarithmic_profile(&self, altitude_m: f64) -> Vector3<f64> {
    let effective_altitude = if altitude_m < 0.001 {
        0.001  // 1mm above ground
    } else {
        altitude_m
    };
    if effective_altitude <= self.roughness_length {
        return Vector3::zeros();
    }
    let speed_ratio = (effective_altitude / self.roughness_length)
        .ln()
        / (self.reference_height / self.roughness_length).ln();
    self.surface_wind.to_vector() * speed_ratio.max(0.0)
}
```

**Roughness Length by Terrain**

The default roughness length of 0.03 m corresponds to short grass—typical of a well-maintained rifle range. For other terrains: open water $\approx$ 0.0002 m, plowed field $\approx$ 0.03 m, scrubland $\approx$ 0.1 m, and forest $\approx$ 1.0 m. The roughness length is not directly exposed as a CLI flag, but it is configurable through the API and the library interface.

### Power Law Profile

The power law is a simpler approximation:

$$U(z) = U_{\text{ref}} \left( \frac{z}{z_{\text{ref}}} \right)^{\alpha} \tag{22.12}$$

The exponent $\alpha$ defaults to $1/7 \approx 0.143$, which corresponds to neutral atmospheric stability over open terrain. Stable conditions (nighttime, light winds) use a larger exponent; unstable conditions (strong solar heating) use a smaller one.

Listing 22.17: Power law wind profile

```
fn power_law_profile(&self, altitude_m: f64) -> Vector3<f64> {
    if altitude_m <= 0.0 {
        return Vector3::zeros();
```

```
    }
    let speed_ratio = (altitude_m / self.reference_height)
        .powf(self.power_exponent);
    self.surface_wind.to_vector() * speed_ratio
}
```

### Ekman Spiral

The Ekman spiral models the *rotation* of wind direction with altitude, not merely the change in speed. Near the surface, friction slows the wind and causes it to blow at an angle to the pressure gradient (toward low pressure). With altitude, friction decreases and the wind veers toward the geostrophic direction—the "free atmosphere" wind aloft.

BALLISTICS-ENGINE implements a simplified Ekman spiral that linearly interpolates both speed and direction between the surface wind and a geostrophic wind (defaulting to 1.5× surface speed with 30° backing):

Listing 22.18: Ekman spiral interpolation

```
fn ekman_spiral_profile(&self, altitude_m: f64) -> Vector3<f64> {
    let geo_wind = self.geostrophic_wind
        .unwrap_or(WindLayer {
            altitude_m: 1000.0,
            speed_mps: self.surface_wind.speed_mps * 1.5,
            direction_deg: self.surface_wind.direction_deg - 30.0,
        });
    let ekman_depth = 1000.0;
    if altitude_m >= ekman_depth {
        return geo_wind.to_vector();
    }
    let ratio = altitude_m / ekman_depth;
    let speed = self.surface_wind.speed_mps * (1.0 - ratio)
        + geo_wind.speed_mps * ratio;
    // ... circular interpolation of direction ...
}
```

The Ekman spiral is the most physically realistic model for conditions where the bullet's trajectory spans a significant altitude range. It is available through online mode via `--wind-shear-model ekman_spiral`.

### Custom Layers

For the most control, `CustomLayers` lets you define wind conditions at specific altitudes and interpolates linearly between them. This is useful when you have wind data from multiple height sources—a surface weather station, a mid-level observation, and a radiosonde or pilot report aloft.

## 22.6.2 Enabling Wind Shear in Practice

Wind shear can be enabled either locally (with `--enable-wind-shear`) or through online mode. The local wind shear uses a simplified model; the online mode can apply more sophisticated weather-zone-aware shear:

Listing 22.19: Local wind shear with logarithmic profile

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --wind-speed 12 --wind-direction 90 \
  --altitude 5000 \
  --enable-wind-shear
```

Listing 22.20: Online wind shear with Ekman spiral

```
ballistics trajectory \
  --velocity 2710 --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --max-range 1200 --auto-zero 100 \
  --wind-speed 15 --wind-direction 90 \
  --altitude 5000 \
  --latitude 39.86 --longitude -104.67 \
  --shot-direction 270 \
  --online --enable-weather-zones \
  --wind-shear-model ekman_spiral
```

**Wind Shear at Extreme Range**

The wind shear implementation in `src/wind_shear.rs` includes an optimization: for ranges beyond 800 meters, a simplified linear interpolation is used instead of the full logarithmic or Ekman model to avoid numerical instability in the RK45 adaptive integrator. At these distances, the simplified model captures the gross altitude dependence while maintaining solver stability.

## 22.7 Weather Zones and Interpolation

### 22.7.1 What Are Weather Zones?

A weather zone is a spatial segment of the trajectory with its own atmospheric conditions. Rather than assuming the same temperature, pressure, and wind from muzzle to target, the engine divides the flight path into zones—each potentially with different conditions.

Weather zones are generated server-side when online mode is active with `--enable-weather-zones`. The API uses the shooter's coordinates, shot direction, and available meteorological data to construct a series of atmospheric profiles along the trajectory's ground track.

### 22.7.2 Zone Interpolation Methods

The `--weather-zone-interpolation` flag controls how the solver transitions between zones. Three methods are available:

**`linear`** (default): Conditions change linearly between zone boundaries. This is smooth and well-behaved numerically, but may not capture sharp atmospheric gradients (e.g., a cold-air pool in a valley).

**`cubic`** : Cubic spline interpolation provides smoother transitions than linear, with continuous first derivatives at zone boundaries. Preferred when zones are widely spaced.

**`step`** : Conditions change abruptly at zone boundaries with no interpolation. This models sharp discontinuities (frontal passages, canyon edges) but can introduce numerical artifacts in the solver.

Listing 22.21: Weather zones with cubic interpolation

```
ballistics trajectory \
  --velocity 2725 --bc 0.818 --mass 300 --diameter 0.338 \
  --drag-model g7 --max-range 2000 --auto-zero 200 \
  --temperature 50 --pressure 29.42 --humidity 35 \
  --altitude 7000 \
  --latitude 36.23 --longitude -112.05 \
  --shot-direction 180 \
  --online --enable-weather-zones \
  --weather-zone-interpolation cubic
```

### 22.7.3 Weather Zones and the Wind Sock

The `WindShearWindSock` struct in `src/wind_shear.rs` combines range-dependent wind segments with altitude-dependent shear. Each wind segment specifies a speed, direction, and "until range" distance, forming a piecewise wind model along the trajectory:

Listing 22.22: Wind sock with range-dependent segments

```
pub struct WindShearWindSock {
    pub segments: Vec<(f64, f64, f64)>,
    // (speed_mps, angle_deg, until_range_m)
    pub shear_profile: Option<WindShearProfile>,
    pub shooter_altitude_m: f64,
}
```

The `vector_for_position` method computes the wind at any 3D position by first selecting the range-appropriate segment, then applying the altitude-dependent shear profile. This gives the solver a wind vector that varies in three dimensions—exactly what is needed for realistic long-range prediction.

## 22.8 Practical Weather Workflow for Range Day

Let's synthesize everything in this chapter into a concrete, step-by-step workflow for getting the best possible trajectory prediction on range day.

### 22.8.1 Step 1: Before You Leave

1. Check the METAR for the nearest airport. Note the temperature, altimeter setting, wind, and dew point.
2. Convert dew point to humidity using Equation (22.1) or an online calculator.
3. Look up your range's coordinates and elevation. Store them in a location CSV if you visit regularly.
4. If using online mode, ensure your BC5D tables are cached:

Listing 22.23: Pre-caching BC5D tables before heading to the range

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --online --bc-table-auto
```

### 22.8.2 Step 2: At the Range

1. Take a fresh weather reading at your position. A handheld weather meter is ideal, but the METAR from 30 minutes ago is acceptable.
2. Record: temperature, barometric pressure (station or altimeter setting), humidity, wind speed, and wind direction.
3. Run your trajectory with the measured conditions:

Listing 22.24: Range-day trajectory with measured conditions

```
ballistics trajectory \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --drag-model g7 --max-range 1000 --auto-zero 100 \
  --temperature 82 --pressure 29.53 --humidity 35 \
  --altitude 4200 \
  --wind-speed 6 --wind-direction 90 \
  --online --offline-fallback \
  --enable-weather-zones
```

### 22.8.3 Step 3: Validate and Adjust

1. Shoot at a known distance (e.g., 600 yards) and measure your actual drop.
2. Compare to the predicted drop. If they differ by more than 0.2 MIL, the atmospheric inputs or BC may need adjustment.
3. If the drop is consistently high (more drop than predicted), the actual air density is higher than modeled—check your pressure reading and altitude.
4. If the drop is consistently low (less drop than predicted), air density is lower—verify temperature (higher temps reduce density).
5. Use the `true-velocity` command to calibrate if needed:

Listing 22.25: Truing velocity from observed drop

```
ballistics true-velocity \
  --measured-drop 6.8 --range 600 \
  --bc 0.462 --drag-model g7 \
  --mass 168 --diameter 0.308 \
  --chrono-velocity 2700 \
  --altitude 4200 --temperature 82 --pressure 29.53 \
  --humidity 35
```

### 22.8.4 Step 4: Monitor Changing Conditions

Weather changes throughout the day. Temperature rises through the morning, wind typically picks up in the afternoon, and pressure can shift with passing weather systems. For precision work:

- Re-read your weather meter every 30 minutes, or whenever you notice mirage patterns changing.
- Update the `--temp` and `--wind-speed` flags accordingly.
- If using online mode with weather zones, each new computation automatically fetches conditions for the current time.

**The Quick-and-Dirty Weather Check**

If you do not have a weather meter, at least check the temperature on your phone and use the nearest METAR for pressure. Getting temperature within 5°F and pressure within 0.5 inHg of actual conditions keeps your drop prediction within 1–2 inches at 1000 yards for most loads. The difference between "phone weather" and "no weather at all" is far larger than the difference between "phone weather" and "Kestrel weather."

### 22.8.5 A Complete Range-Day Command

For the reader who wants a single, comprehensive command to copy and adapt, here is an example that exercises every weather-related feature:

Listing 22.26: Comprehensive weather-integrated trajectory

```
ballistics trajectory \
  --saved-profile "308_smk_168" \
  --location range_sites.csv --site "NRA_Whittington" \
  --wind-speed 8 --wind-direction 225 \
  --online --offline-fallback \
  --enable-weather-zones --enable-3d-weather \
  --wind-shear-model logarithmic \
  --weather-zone-interpolation linear \
  --compare
```

This loads the .308 Win profile, the NRA Whittington Center location (including altitude, coordinates, and default environmental conditions from the CSV), applies an 8 mph wind from the southwest, enables weather zones and 3D weather corrections with logarithmic wind shear and linear zone interpolation, and runs a local-vs-API comparison. The result is the most accurate trajectory prediction the engine can produce—grounded in real atmospheric data and enhanced by the API's ML corrections.

## 22.9 Exercises

1. Look up the current METAR for the airport nearest to your range. Decode the temperature, pressure, dew point, and wind. Convert the dew point to relative humidity using Equation (22.1). Run a .308 Win trajectory (168 gr SMK, BC `0.462`, 2700 fps, G7) to 1000 yards with these conditions.

2. Run the same trajectory twice: once with 0% humidity and once with 100% humidity, holding all other conditions constant at 90°F, 29.92 inHg, sea level. How much does the predicted drop at 1000 yards differ between the two cases?

3. Simulate a mountain shooting scenario. Set the altitude to 8500 feet, temperature to 45°F, and pressure to 22.0 inHg. Compare the .308 Win drop at 800 yards to the sea-level standard atmosphere result. How many MOA of elevation correction does the altitude save?

4. Enable wind shear with the logarithmic model for a 15 mph crosswind at 5000 feet altitude. Compare the wind drift at 1000 yards to the same trajectory with wind shear disabled. Does the shear model predict more or less drift?

5. Using the 6.5 Creedmoor load (140 gr ELD-M, BC `0.610`, 2710 fps, G7), run a trajectory to 1200 yards with `--online`, `--enable-weather-zones`, and `--enable-3d-weather` enabled. Then run the same trajectory without these flags. Compare the two drop predictions at 1200 yards. When is the weather-zone model worth the extra complexity?

**What's Next.** With a thorough understanding of how weather data flows through the ballistic model—from METAR reports and ICAO layers to humidity-corrected density and altitude-dependent wind shear—we are ready to put all of this into practice. Chapter 23 opens Part VIII by applying computational ballistics to hunting scenarios, where getting the atmospheric model right can mean the difference between a clean harvest and a missed opportunity.

# Part VIII

# Real-World Applications

# Chapter 23

# Hunting Applications

> *"The ethical hunter owes it to the animal to make the shot count. A ballistics solver does not replace marksmanship, but it removes the guesswork from the math."*

For the hunter, the ballistics problem is fundamentally different from the target shooter's. A steel plate doesn't care if you hit it two inches left of center—a game animal does. Hunting demands confidence that the bullet will arrive inside the vital zone with enough energy to ensure a quick, humane kill, under field conditions that rarely match the range where you zeroed.

This chapter shows how to use `ballistics-engine` to answer the questions every serious hunter faces: *How far can I point and shoot? What is my maximum ethical range? How do I correct for a steep uphill or downhill shot? What changes when I hunt at* 10 000 *feet instead of sea level?* And, most practically: *How do I produce a dope card that fits in my pocket?*

We will work through these questions with real cartridge data, real `ballistics` commands, and real physics—starting with the concept every hunter should master first.

## 23.1 Point-Blank Range and Vital Zone Sizing

### 23.1.1 What Is Point-Blank Range?

*Point-blank range*[1] is the maximum distance at which you can aim dead-center on an animal's vital zone and be guaranteed a hit—no holdover, no dial, just the crosshair on the center of the chest.

[1] The term "point-blank" has a long history. In modern practical usage, it means the distance at which the bullet's trajectory stays within the target zone without any holdover or scope adjustment.

More precisely, it is the range at which the bullet's trajectory first drops below one-half the vital zone diameter beneath the line of sight. Between the muzzle and that distance, the bullet never rises more than one-half the vital zone above the line of sight, nor drops more than one-half below it.

**Maximum Point-Blank Range (MPBR)**

The MPBR for a given load and vital zone size is found by choosing the zero distance that makes the trajectory's *maximum ordinate* (highest point above the line of sight) exactly equal to one-half the vital zone diameter. The range at which the trajectory drops one-half the vital zone below the line of sight is the MPBR.

The `ballistics-engine` computes MPBR with a dedicated `mpbr` subcommand. Internally, it performs a binary search over zero distances in the function `handle_mpbr()` in `src/main.rs`. For each candidate zero, the engine runs a full sampled trajectory and measures the maximum ordinate. It converges when the maximum ordinate matches the half-vital-zone radius to within 0.001 m (about 0.04 inches).

### 23.1.2 Computing MPBR

Consider a classic whitetail deer hunting load: .308 Winchester, 165-grain Nosler AccuBond, G1 BC `0.475`, muzzle velocity 2700 fps (823 m/s). A whitetail's vital zone is commonly estimated as an 8-inch (20 cm) circle centered on the heart–lung area.

Listing 23.1: Computing MPBR for a .308 Win deer load.

```
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 8 \
  --sight-height 1.5
```

This command returns the optimal zero distance, the near zero crossing, the far zero crossing (MPBR), the maximum ordinate, and the remaining velocity and energy at MPBR. For this load, expect an MPBR of approximately 295–305 yards with an optimal zero around 255 yards.

**Tip**

The `--vital-zone` flag accepts the diameter in inches (imperial) or centimeters (metric). Use 8 inches for whitetail deer, 10 inches for elk, and 6 inches for pronghorn or similar smaller-vital-zone game.

### 23.1.3 Vital Zone Sizing for Different Game

The choice of vital zone diameter dramatically affects MPBR. Table 23.1 shows commonly used values:

Table 23.1: Common vital zone diameters for North American game.

| Game Animal | Vital Zone (in) | Vital Zone (cm) |
|---|---|---|
| Pronghorn antelope | 6 | 15 |
| Whitetail deer | 8 | 20 |
| Mule deer | 9 | 23 |
| Elk / caribou | 10 | 25 |
| Moose | 12 | 30 |

Let us compare MPBR for the same .308 Win load across different vital zones:

Listing 23.2: MPBR across different vital zone sizes.

```
# Pronghorn (6-inch vital zone)
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 6

# Whitetail (8-inch vital zone)
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 8

# Elk (10-inch vital zone)
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 10
```

Increasing the vital zone from 6 to 10 inches typically extends MPBR by 40–60 yards, because the trajectory has more room to arc before exceeding the allowable deviation.

### 23.1.4 MPBR and the Flat-Shooting Myth

A common error is to conclude from MPBR calculations that a "flat-shooting" magnum cartridge eliminates the need for range estimation. The numbers tell a more nuanced story. Compare the .308 Win above with a .300 Winchester Magnum, 180-grain AccuBond, G1 BC `0.507`, at 2960 fps:

Listing 23.3: MPBR comparison: .308 Win vs. .300 Win Mag.

```
# .308 Win, 165gr, BC 0.475, 2700 fps
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 8

# .300 Win Mag, 180gr, BC 0.507, 2960 fps
ballistics mpbr \
  -v 2960 -b 0.507 -m 180 -d 0.308 \
  --vital-zone 8
```

The .300 Win Mag gains roughly 30–40 yards of MPBR over the .308—meaningful, but not transformative. The lesson is that *within a class of similar cartridges*, MPBR differences are modest. What matters far more is the hunter's ability to estimate range accurately.

**Warning**

MPBR is a useful concept, but it is not a license to skip rangefinding. A 10% range estimation error at 300 yards means missing the vital zone entirely. Always carry and use a rangefinder when hunting at extended distances.

## 23.2 Maximum Effective Range: Energy and Accuracy Thresholds

Point-blank range answers the question "How far can I aim dead center and hit the vital zone?" But a hit inside the vital zone is not sufficient if the bullet lacks the energy for reliable expansion and penetration. *Maximum effective range* imposes two additional constraints: the bullet must carry enough energy, and the shooter must be accurate enough to consistently place shots in the vital zone.

### 23.2.1 Energy Thresholds

Conventional wisdom—supported by decades of field experience—sets minimum energy thresholds for humane kills:

Table 23.2: Commonly cited minimum impact energies by game class.

| Game Class | Minimum Energy (ft-lbs) | Minimum Energy (J) |
|---|---|---|
| Varmint (coyote, fox) | 200 | 271 |
| Deer class (whitetail) | 1,000 | 1,356 |
| Medium game (mule deer) | 1,200 | 1,627 |
| Elk / moose | 1,500 | 2,034 |
| Large African plains game | 2,000 | 2,712 |

**SAFETY: Safety Warning**

Energy thresholds are guidelines, not absolute rules. Bullet construction, impact velocity (for reliable expansion), shot placement, and angle of incidence all affect terminal performance. A marginal-energy hit through both lungs is far more effective than a high-energy hit through the paunch. These thresholds are *starting points* for responsible range estimation, not substitutes for shot selection discipline.

### 23.2.2 Finding the Energy-Limited Range

The `ballistics-engine` reports energy at every range increment when you use the `--full` and `--sample-trajectory` flags. To find the range at which energy drops below 1000 ft-lbs for a deer-class minimum:

Listing 23.4: Finding the energy-limited range for whitetail.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 800 \
  --sample-trajectory --sample-interval 50 --full
```

Examine the "Energy (ft-lbs)" column. For our .308 Win load, energy drops below 1000 ft-lbs at approximately 550–600 yards, well beyond the MPBR of ~300 yards.

For a JSON pipeline that extracts the threshold automatically:

Listing 23.5: Piping JSON to find the energy threshold range.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 800 \
  --sample-trajectory --sample-interval 25 --full \
  -o json | jq '[.trajectory[] | select(.energy_ftlbs < 1000)] | .[0].range'
```

### 23.2.3 Accuracy Threshold

Even if the bullet carries enough energy, the shooter must be capable of placing the shot consistently within the vital zone. A rifle that shoots 1 MOA groups at the bench will produce a theoretical group diameter of 8 inches at 800 yards—exactly the size of a whitetail vital zone, leaving zero margin for wind error, range error, or field-position degradation.

A practical rule of thumb: your *maximum effective range* is the distance at which your field shooting capability (not bench capability—add a generous factor) produces a group that fills no more than

*half* the vital zone. If you shoot 1.5 MOA from field positions, your effective range against an 8-inch vital zone is:

$$R_{\max} = \frac{d_{\text{vital}}/2}{1.5 \times 1.047} \times 100 \approx \frac{4}{1.57} \times 100 \approx 255 \text{ yards} \tag{23.1}$$

where $d_{\text{vital}}$ is the vital zone diameter in inches and the factor of $1.047$ converts MOA to inches at 100 yards.

### 23.2.4 Combining Constraints

The true maximum effective range is the *minimum* of three numbers:

1. **MPBR** (or range at which holdover exceeds your ability to estimate it)
2. **Energy-limited range** (below which terminal performance is unreliable)
3. **Accuracy-limited range** (beyond which your shot group exceeds the vital zone)

For most hunting scenarios, the accuracy limit is the binding constraint. The trajectory and energy limits are easily computed with `ballistics-engine`; the accuracy limit requires honest self-assessment.

**Tip**

Use the Monte Carlo subcommand (Chapter 6) to model your realistic accuracy. Set the angle standard deviation to your field-position accuracy and run 1,000 iterations. The resulting CEP gives you a data-driven effective range.

### 23.2.5 Velocity and Bullet Expansion

Energy alone does not tell the full story. Modern expanding hunting bullets are designed to perform within a specific *impact velocity window*. Below the minimum expansion velocity, the bullet may pencil through without expanding, creating a small wound channel regardless of the energy it carries.

Most controlled-expansion hunting bullets (e.g., bonded core, tipped designs) require impact velocities of 1800–2000 fps for reliable expansion. Some newer designs (like monolithic copper bullets) can expand reliably at velocities as low as 1600 fps.

Use `ballistics-engine` to find the range at which velocity drops below the expansion threshold:

Listing 23.6: Finding the minimum-expansion-velocity range.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 800 \
```

```
  --sample-trajectory --sample-interval 25 --full \
  -o json | jq '[.trajectory[] | select(.velocity_fps < 1800)] | .[0].range'
```

For our .308 Win load, velocity drops below 1800 fps at approximately 500–550 yards. This may be a more meaningful range limit than the energy threshold for bullets that require higher impact velocities for expansion.

**SAFETY: Safety Warning**

Always consult your bullet manufacturer's recommended impact velocity range. A 1200 ft-lb hit at 1650 fps may fail to expand with some bullet designs, resulting in a wound that is less effective than a lower-energy hit from a bullet designed for lower-velocity expansion. Terminal performance depends on bullet construction, not just energy.

### 23.2.6 Wind Considerations for Hunters

Hunters often focus on drop and energy while ignoring wind drift —the number-one cause of lateral misses in the field. Unlike competition shooters who can read flags and mirage, hunters must estimate wind from natural indicators: grass movement, dust, tree sway, and the feel on their face.

A rough field guide to wind speed estimation:

| Wind Speed | Indicators |
|---|---|
| 3–5 mph | Leaves rustle, feel wind on face |
| 5–8 mph | Small branches move, grass sways |
| 8–12 mph | Large branches move, dust rises |
| 12–18 mph | Small trees sway, difficult to walk against |

Use the `--wind-speed` and `--wind-direction` flags to understand how much drift your load produces under realistic field conditions:

Listing 23.7: Wind drift analysis for hunting at 400 yards.

```
# 10 mph full-value crosswind
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 400 \
  --wind-speed 10 --wind-direction 90 --full

# 10 mph quartering wind (45 degrees)
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 400 \
```

```
  --wind-speed 10 --wind-direction 45 --full
```

At 400 yards, a 10 mph full-value crosswind pushes a .308 Win bullet approximately 10–12 inches—enough to miss the vital zone entirely if uncompensated. A quartering wind (45 degrees) produces roughly 70% of the full-value drift.

**Warning**

In hunting situations, err on the side of waiting for a wind lull rather than trying to hold off for wind. A clean miss is always preferable to a wounded animal. If the wind-induced drift at your intended range exceeds half the vital zone diameter, wait for calmer conditions or reduce the distance.

## 23.3 Uphill and Downhill Shooting: The Rifleman's Rule

Mountain hunters and tree-stand hunters share a common challenge: the shot is not level. A 300-yard shot at a 30-degree downhill angle is *not* the same as a 300-yard shot on flat ground. Understanding why—and knowing how much to correct—is essential for ethical hunting at steep angles.

### 23.3.1 Why Angle Matters

Gravity acts *vertically*, but the bullet travels along the line of sight, which may be inclined. The component of gravity that causes bullet drop relative to the line of sight is not the full gravitational acceleration $g$, but only the component perpendicular to the line of sight:

$$g_{\perp} = g \cos\theta \tag{23.2}$$

where $\theta$ is the shooting angle from horizontal (positive for uphill, negative for downhill). Because $\cos\theta < 1$ for any non-zero angle—and $\cos\theta$ is the same for both positive and negative $\theta$—the bullet drops *less* relative to the line of sight for both uphill and downhill shots.

**The Rifleman's Rule**

When shooting at an angle $\theta$ from horizontal, act as if the target were at the *horizontal distance*:

$$R_{\text{horiz}} = R_{\text{LOS}} \cos\theta$$

where $R_{\text{LOS}}$ is the line-of-sight (slant) range to the target. Use the dope for $R_{\text{horiz}}$, not $R_{\text{LOS}}$. This approximation, known as the *Rifleman's Rule*, is accurate to within 1% for angles up to about 30 degrees and ranges within normal hunting distances.

The `ballistics-engine` handles angled shots directly via the `--shooting-angle` flag. This is implemented in the `zero_angle()` function in `src/angle_calculations.rs`, which adjusts the target's vertical position based on the shooting angle:

$$h_{\text{vert}} = R_{\text{LOS}} \sin\theta \tag{23.3}$$

where $h_{\text{vert}}$ is the vertical offset of the target from the muzzle's horizontal plane. The zeroing algorithm then finds the muzzle angle that places the bullet at the correct height at the correct horizontal distance.

### 23.3.2 Computing Angled Shots

Let us compute the trajectory for a 300-yard shot at a 30-degree downhill angle with our .308 Win deer load:

Listing 23.8: Downhill shot at 30 degrees.

```
# Level shot at 300 yards (baseline)
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 300

# 30-degree downhill shot at 300 yards LOS
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 300 \
  --shooting-angle -30
```

The downhill shot will show significantly less drop than the level shot because the effective gravitational component is reduced by $\cos(30°) = 0.866$. The practical consequence: if you dial for a level 300-yard shot and fire downhill, you will shoot *high*.

### 23.3.3 How Much Does It Matter?

Table 23.3 shows the correction for our .308 Win load at 300 yards:

At hunting distances, the angle correction is modest for angles below 15° (typical of most whitetail stands) but becomes significant above 30° (mountain sheep, high tree stands). The key point: for both uphill and downhill, the bullet impacts *higher* than a level-shot dope would predict.

Listing 23.9: Comparing uphill and downhill corrections.

```
# 30-degree uphill
ballistics trajectory \
```

Table 23.3: Effect of shooting angle on drop at 300 yards (.308 Win, 165 gr, 2700 fps).

| Angle | Approx. Drop (in) | Correction (in) |
|---|---|---|
| 0° (level) | −17.5 | — |
| 15° | −16.9 | +0.6 |
| 30° | −15.2 | +2.3 |
| 45° | −12.4 | +5.1 |

```
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 400 \
  --shooting-angle 30

# 30-degree downhill
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 400 \
  --shooting-angle -30
```

Notice that uphill and downhill produce nearly identical corrections at the same magnitude—a direct consequence of the $\cos\theta$ relationship.

### 23.3.4 When the Rifleman's Rule Breaks Down

The cosine approximation is excellent for moderate angles and distances. It begins to deviate from the full numerical solution at extreme angles ($> 45°$) and long ranges ($> 600$ yards), where the difference in air density along the trajectory path (higher altitude for uphill shots) and the asymmetric drag on the ascending versus descending legs of the trajectory become significant.

For extreme mountain hunting scenarios, use `ballistics-engine`'s full numerical solution (`--shooting-angle`) rather than the mental cosine shortcut. The engine integrates the complete equations of motion along the actual flight path, accounting for density variations with altitude.

## 23.4 Altitude Corrections for Mountain Hunting

When you drive from your sea-level range to an elk camp at 9000 feet (2743 m), the air your bullet flies through is dramatically thinner. Less air density means less drag, higher retained velocity, a flatter trajectory, and—counter-intuitively for some hunters—a *higher* impact point if you're using your sea-level zero.

### 23.4.1 The Magnitude of the Effect

At 9000 feet, air density is approximately 74% of its sea-level value. The speed of sound also decreases (because temperature decreases with altitude in the troposphere), which shifts the Mach number profile of the bullet and changes the drag coefficient.

Let us quantify the effect on our .308 Win load:

Listing 23.10: Sea-level versus 9,000 ft altitude trajectory.

```
# Sea level, standard conditions
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --sample-trajectory --sample-interval 100 --full

# 9,000 ft, typical mountain temperature
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --altitude 9000 --temperature 35 \
  --sample-trajectory --sample-interval 100 --full
```

At 600 yards, the altitude change produces roughly 8–12 inches less drop—enough to miss the vital zone if you're using a sea-level dope card.

### 23.4.2 Re-Zeroing for Altitude

The best practice for mountain hunting is to re-zero at the hunting altitude. If that is not possible, use the engine to compute the shift:

Listing 23.11: Computing the altitude-induced zero shift.

```
# Zero was set at sea level, 100 yards
# What is the impact point at 100 yards at 9000 ft?
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 100 \
  --altitude 9000 --temperature 35
```

At 100 yards, the reduced drag produces a negligibly different zero point (typically less than 0.1 inches). But the cumulative effect grows with range—at 400 yards the sea-level dope may be off by 3–5 inches, and at 600 yards by 8–12 inches.

### 23.4.3 Density Altitude: A Single Number

*Density altitude* combines the effects of physical altitude, temperature, pressure, and humidity into a single number that describes the "effective" altitude of the atmosphere from a drag perspective. On a hot day at 5000 feet, the density altitude may be 8000 feet or higher—meaning the air is as thin as it would be at 8000 feet under standard conditions.

The PDF dope card feature of `ballistics-engine` automatically computes and displays density altitude using the `calculate_density_altitude()` function in `src/pdf_dope_card.rs`.

Listing 23.12: Generating a PDF dope card that shows density altitude.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --altitude 5000 --temperature 80 --humidity 30 \
  --sample-trajectory --sample-interval 50 \
  -o pdf --output-file hunting_dope.pdf \
  --bullet-name "165gr AccuBond" \
  --location-name "Elk Camp, CO"
```

**Tip**

When preparing for a mountain hunt, run trajectories at the expected hunting altitude *and* at the expected temperature range. A 20-degree temperature swing at altitude can produce as much trajectory change as a 1,000-foot altitude difference. Build dope cards for both ends of the expected temperature range.

## 23.5 Building a Field-Ready Dope Card

All the trajectory data in the world is useless if you cannot access it when it matters. A field-ready dope card distills the essential information—drop, wind, and lead corrections at each range increment—onto a compact, weatherproof card that you tape to your stock, sleeve in a binder, or photograph on your phone.

### 23.5.1 What Goes on a Dope Card

A hunting dope card needs:

1. **Range increments**: Typically every 25 or 50 yards from zero out to your maximum effective range.
2. **Drop corrections**: In MILs or MOA, matching your scope's adjustment system.

3. **Wind corrections**: For a standard crosswind speed (e.g., 10 mph full-value), also in MILs or MOA.
4. **Remaining velocity and energy**: To confirm you are above the minimum energy threshold.
5. **Environmental conditions**: The temperature, altitude, and pressure for which the card was computed.

### 23.5.2 The Come-Ups Subcommand

The `come-ups` subcommand is purpose-built for generating dope cards. It produces a clean table of elevation and windage corrections at regular range intervals:

Listing 23.13: Generating a come-up table for a hunting load.

```
ballistics come-ups \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --zero-distance 200 \
  --start 100 --end 600 --step 50 \
  --adjustment-unit mil \
  --wind-speed 10 --wind-direction 90 \
  --altitude 5000 --temperature 40
```

This produces a table showing the MIL elevation and windage adjustment at every 50 yards from 100 to 600.

### 23.5.3 PDF Dope Cards

For a printable, color-coded card, use the `-o pdf` output format. The PDF dope card module (`src/pdf_dope_card.rs`) produces a two-column layout with color-coded data: black for range, red for drop, green for wind, and blue for lead. Alternating row striping improves readability.

Listing 23.14: Generating a PDF dope card for hunting.

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 600 \
  --altitude 5000 --temperature 40 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file elk_hunt_dope.pdf \
  --bullet-name "165gr AccuBond" \
  --powder "IMR 4064, 44.0gr" \
  --location-name "Unit 61, CO - 9000ft" \
  --font-preset large --bold-data
```

The `--font-preset` flag accepts `small`, `medium`, or `large`, controlled by the `FontSizePreset` enum in the source. For field use with aging eyes or cold hands, `large` is recommended. The `--bold-data` flag makes the numerical values bold for enhanced contrast.

### 23.5.4 Multiple Conditions, Multiple Cards

A serious mountain hunter might prepare three cards:

Listing 23.15: Building dope cards for a range of conditions.

```
# Cold morning (25 F)
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 500 \
  --altitude 9000 --temperature 25 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file dope_cold.pdf \
  --bullet-name "165gr AccuBond" \
  --location-name "Elk Camp - Cold AM"

# Mild afternoon (50 F)
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 500 \
  --altitude 9000 --temperature 50 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file dope_mild.pdf \
  --bullet-name "165gr AccuBond" \
  --location-name "Elk Camp - Mild PM"

# Hot midday (70 F)
ballistics trajectory \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --auto-zero 200 --max-range 500 \
  --altitude 9000 --temperature 70 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file dope_warm.pdf \
  --bullet-name "165gr AccuBond" \
  --location-name "Elk Camp - Warm Midday"
```

**Tip**

Laminate your dope cards or place them in waterproof sleeves. The best trajectory calculation in the world is useless when the rain smears the ink. Alternatively, photograph them on your phone as a backup.

## 23.6 Example Setups: Common Hunting Cartridges

Let us work through complete examples for five popular hunting cartridges, computing MPBR, energy-limited range, and generating dope cards for each.

### 23.6.1 6.5 Creedmoor, 143 gr ELD-X

The 6.5 Creedmoor has become one of the most popular all-around hunting cartridges in North America. The 143-grain Hornady ELD-X is a popular hunting bullet for this caliber.

- Muzzle velocity: 2700 fps (823 m/s)
- G7 BC: `0.315`
- Bullet weight: 143 gr
- Caliber: 0.264 in

Listing 23.16: 6.5 Creedmoor hunting setup.

```
# MPBR for whitetail (8-inch vital zone)
ballistics mpbr \
  -v 2700 -b 0.315 -m 143 -d 0.264 \
  --drag-model g7 --vital-zone 8

# Full trajectory with energy data
ballistics trajectory \
  -v 2700 -b 0.315 -m 143 -d 0.264 \
  --drag-model g7 \
  --auto-zero 200 --max-range 800 \
  --sample-trajectory --sample-interval 50 --full

# Come-up table for mountain hunt
ballistics come-ups \
  -v 2700 -b 0.315 -m 143 -d 0.264 \
  --drag-model g7 \
  --zero-distance 200 \
  --start 100 --end 700 --step 50 \
  --adjustment-unit mil \
```

```
  --altitude 7000 --temperature 45
```

The 6.5 Creedmoor produces an MPBR of approximately 300–310 yards with an 8-inch vital zone. Energy drops below the 1000 ft-lb deer threshold at approximately 700–750 yards—a testament to the cartridge's excellent BC-to-recoil ratio.

### 23.6.2 .270 Winchester, 150 gr Partition

The .270 Winchester remains one of America's most beloved deer and elk cartridges, with nearly a century of field performance behind it.

- Muzzle velocity: 2850 fps (869 m/s)
- G1 BC: `0.465`
- Bullet weight: 150 gr
- Caliber: 0.277 in

Listing 23.17: .270 Win hunting setup.

```
# MPBR for elk (10-inch vital zone)
ballistics mpbr \
  -v 2850 -b 0.465 -m 150 -d 0.277 \
  --vital-zone 10

# Trajectory zeroed at 200 yards
ballistics trajectory \
  -v 2850 -b 0.465 -m 150 -d 0.277 \
  --auto-zero 200 --max-range 600 \
  --sample-trajectory --sample-interval 50 --full
```

### 23.6.3 .30-06 Springfield, 180 gr AccuBond

The .30-06 is the quintessential American hunting cartridge. With a 180-grain premium bullet, it handles everything from whitetail to elk to moose.

- Muzzle velocity: 2750 fps (838 m/s)
- G1 BC: `0.507`
- Bullet weight: 180 gr
- Caliber: 0.308 in

Listing 23.18: .30-06 hunting setup.

```
# MPBR for elk (10-inch vital zone)
ballistics mpbr \
  -v 2750 -b 0.507 -m 180 -d 0.308 \
  --vital-zone 10

# Full trajectory with energy for elk-class game
ballistics trajectory \
  -v 2750 -b 0.507 -m 180 -d 0.308 \
  --auto-zero 200 --max-range 700 \
  --sample-trajectory --sample-interval 50 --full
```

The .30-06 with 180-grain bullets produces a remarkably flat trajectory thanks to its combination of moderate velocity and high BC. Energy stays above 1500 ft-lbs (the elk threshold) to approximately 400–450 yards.

### 23.6.4 .300 Winchester Magnum, 200 gr ELD-X

When range and energy are both at a premium—think mountain elk at 500+ yards—the .300 Win Mag with a heavy, high-BC bullet is hard to beat.

- Muzzle velocity: 2850 fps (869 m/s)
- G7 BC: `0.336`
- Bullet weight: 200 gr
- Caliber: 0.308 in

Listing 23.19: .300 Win Mag hunting setup.

```
# MPBR for elk (10-inch vital zone)
ballistics mpbr \
  -v 2850 -b 0.336 -m 200 -d 0.308 \
  --drag-model g7 --vital-zone 10

# Energy check out to 800 yards for long-range elk
ballistics trajectory \
  -v 2850 -b 0.336 -m 200 -d 0.308 \
  --drag-model g7 \
  --auto-zero 200 --max-range 800 \
  --sample-trajectory --sample-interval 50 --full

# Mountain hunt dope card at 8000 ft
ballistics trajectory \
```

```
  -v 2850 -b 0.336 -m 200 -d 0.308 \
  --drag-model g7 \
  --auto-zero 200 --max-range 700 \
  --altitude 8000 --temperature 30 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file 300wm_elk.pdf \
  --bullet-name "200gr ELD-X" \
  --location-name "Elk Camp, 8000ft"
```

The .300 Win Mag keeps energy above 1500 ft-lbs to approximately 650–700 yards at sea level, extending to 700–750 yards at 8000 feet thanks to reduced drag.

### 23.6.5 .375 H&H Magnum, 300 gr Swift A-Frame

For dangerous game and heavy African plains game, the .375 H&H Magnum is the benchmark. Its trajectory is more arched than the lighter cartridges, making MPBR and precise dope critical.

- Muzzle velocity: 2530 fps (771 m/s)
- G1 BC: `0.398`
- Bullet weight: 300 gr
- Caliber: 0.375 in

Listing 23.20: .375 H&H hunting setup.

```
# MPBR for large game (12-inch vital zone)
ballistics mpbr \
  -v 2530 -b 0.398 -m 300 -d 0.375 \
  --vital-zone 12

# Trajectory with energy data
ballistics trajectory \
  -v 2530 -b 0.398 -m 300 -d 0.375 \
  --auto-zero 200 --max-range 400 \
  --sample-trajectory --sample-interval 50 --full
```

Despite its rainbow trajectory compared to a 6.5 Creedmoor, the .375 H&H delivers crushing energy—over 2000 ft-lbs beyond 300 yards. Its MPBR with a 12-inch vital zone is approximately 260–275 yards.

### 23.6.6 Cartridge Comparison Summary

Table 23.4 summarizes the key hunting performance metrics for all five cartridges:

Table 23.4: Hunting cartridge comparison summary (8-inch vital zone for MPBR except where noted).

| Cartridge | MV (fps) | MPBR (yd) | 1000 ft-lb range (yd) | 1500 ft-lb range (yd) | Muzzle E (ft-lbs) |
|---|---|---|---|---|---|
| 6.5 CM, 143 gr | 2700 | ∼305 | ∼725 | ∼450 | 2,315 |
| .270 Win, 150 gr | 2850 | ∼315 | ∼575 | ∼350 | 2,705 |
| .30-06, 180 gr | 2750 | ∼310 | ∼650 | ∼425 | 3,022 |
| .300 WM, 200 gr | 2850 | ∼320 | ∼775 | ∼650 | 3,605 |
| .375 H&H, 300 gr* | 2530 | ∼270 | >400 | >400 | 4,263 |

*MPBR computed with 12-inch vital zone.

## Exercises

1. **MPBR sensitivity.** Using the .308 Win load from Section 23.1.2, compute the MPBR with vital zone diameters of 6, 8, 10, and 12 inches. Plot MPBR vs. vital zone size. Is the relationship approximately linear?

```
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 6

ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 12
```

2. **Altitude and energy.** Compute the 1000 ft-lb energy range for the 6.5 Creedmoor load at sea level, at 5000 feet, and at 10 000 feet. By how many yards does the energy-limited range extend at high altitude?

```
ballistics trajectory \
  -v 2700 -b 0.315 -m 143 -d 0.264 \
  --drag-model g7 \
  --auto-zero 200 --max-range 900 \
  --altitude 0 \
  --sample-trajectory --sample-interval 25 --full

ballistics trajectory \
```

```
  -v 2700 -b 0.315 -m 143 -d 0.264 \
  --drag-model g7 \
  --auto-zero 200 --max-range 900 \
  --altitude 10000 --temperature 30 \
  --sample-trajectory --sample-interval 25 --full
```

3. **Angle correction practice.** For the .30-06 load, compute the trajectory at 400 yards for shooting angles of $0°$, $15°$, $30°$, and $45°$. Verify that the correction is approximately proportional to $(1 - \cos\theta)$.

```
ballistics trajectory \
  -v 2750 -b 0.507 -m 180 -d 0.308 \
  --auto-zero 200 --max-range 400

ballistics trajectory \
  -v 2750 -b 0.507 -m 180 -d 0.308 \
  --auto-zero 200 --max-range 400 \
  --shooting-angle 30
```

4. **Build your own dope card.** Using your own rifle's load data (or any cartridge from this chapter), generate a PDF dope card for the conditions at your favorite hunting area. Include wind drift for a 10 mph crosswind.

5. **Energy comparison.** Using the .300 Win Mag and .30-06 loads, compare the remaining energy at 500 yards. How much additional energy does the magnum provide? Is it enough to justify the additional recoil for elk hunting?

```
ballistics trajectory \
  -v 2750 -b 0.507 -m 180 -d 0.308 \
  --auto-zero 200 --max-range 500 --full

ballistics trajectory \
  -v 2850 -b 0.336 -m 200 -d 0.308 \
  --drag-model g7 \
  --auto-zero 200 --max-range 500 --full
```

6. **Monte Carlo for hunting.** Run a Monte Carlo simulation for the 6.5 Creedmoor at 400 yards with a velocity SD of 12 fps and a pointing error of 0.5 MOA. What is the CEP, and does it fit within an 8-inch vital zone?

```
ballistics monte-carlo \
  -v 2700 -b 0.315 -m 143 -d 0.264 \
```

```
-n 1000 \
--velocity-std 12 --angle-std 0.15 \
--target-distance 400
```

## What's Next

This chapter focused on the hunter's ballistic problems: point-blank range, energy thresholds, angle corrections, and dope cards. In Chapter 24, we shift to the precision rifle competition world, where the demands are different—finer angular resolution, wind calls under time pressure, and first-round hit probability—but the underlying physics and the same `ballistics-engine` commands adapt seamlessly.

# Chapter 24

# Precision Rifle Competition

*"You can't miss fast enough to win."*
*—Common PRS wisdom*

Precision rifle competition—whether Precision Rifle Series (PRS), National Rifle League (NRL), F-Class, or the growing ranks of regional and club-level matches—places unique demands on the ballistics solver. The hunter needs one good shot at a known distance; the competitor needs dozens of precise shots per stage at unknown distances, under time pressure, with rapidly shifting wind conditions and unfamiliar shooting positions.

This chapter shows how to use `ballistics-engine` to prepare for competition: building MIL- and MOA-resolution dope books, using Monte Carlo simulations for wind-call confidence, planning stages around first-round hit probability, and integrating spin drift and Coriolis effects at the ranges where they matter. The goal is not merely to produce accurate trajectory data, but to build a *workflow* that makes match day efficient and repeatable.

## 24.1 Competition Requirements: MOA and MIL Precision

### 24.1.1 How Precise Is Precise Enough?

A typical PRS/NRL match presents targets ranging from 200 to 1200 yards (occasionally beyond), with target sizes from 1 MOA to 2 MOA. At 800 yards, a 1 MOA target subtends about 8.4 inches. Your dope must be accurate to *better* than the target size—typically within 0.2 MIL (0.7 MOA) of the true solution to leave margin for wind error and shooter wobble.

**MIL vs. MOA Turret Resolution**

Most competition scopes adjust in 0.1 MIL (approximately 0.34 MOA) or 0.25 MOA clicks. A 0.1 MIL click at 1000 yards is 3.6 inches of adjustment. A 0.25 MOA click at the same distance is 2.6 inches. Either system is adequate for competition if the dope is accurate; the choice is largely personal preference and reticle type.

### 24.1.2 The Precision Budget

Think of your total shot error as a *budget* with contributions from multiple sources:

1. **Rifle mechanical accuracy**: Typically 0.3–0.75 MOA for a competition rifle.
2. **Dope error**: How closely your predicted trajectory matches reality.
3. **Wind-call error**: Your ability to read wind speed and direction.
4. **Shooter error**: Position instability, trigger control, parallax.
5. **Ranging error**: For unknown-distance targets.

These errors add in quadrature (root-sum-of-squares):

$$\sigma_{\text{total}} = \sqrt{\sigma_{\text{rifle}}^2 + \sigma_{\text{dope}}^2 + \sigma_{\text{wind}}^2 + \sigma_{\text{shooter}}^2 + \sigma_{\text{range}}^2} \tag{24.1}$$

The solver eliminates $\sigma_{\text{dope}}$—the controllable, systematic error—leaving you to focus on wind reading and shooting fundamentals. A well-trued solver brings dope error below 0.1 MIL even at 1000+ yards, making it effectively zero in the budget.

### 24.1.3 PRS/NRL vs. F-Class: Different Precision Demands

While the underlying physics is the same, PRS/NRL and F-Class impose very different demands on the ballistics solver:

For PRS/NRL, the solver's primary job is to deliver dope accurate enough for a *hit* on a plate-sized target from awkward positions under time pressure. For F-Class, the solver must deliver dope accurate enough to keep rounds inside a 5-inch X-ring at 1000 yards from a stable prone position. The F-Class shooter needs finer angular resolution and is more sensitive to subtle effects like spin drift, Coriolis, and density altitude.

Table 24.1: Competition format comparison: PRS/NRL vs. F-Class.

| Factor | PRS/NRL | F-Class |
|---|---|---|
| Typical max range | 1,200 yd | 1,000 yd |
| Target size | 1–3 MOA | 0.5–1 MOA (X-ring) |
| Shooting position | Varied (barricade, roof, etc.) | Prone only |
| Time per target | 3–10 sec | 30–60 sec |
| Scoring | Hit/miss | Rings (10-X) |
| Wind reading emphasis | Critical | Critical |
| Dope resolution needed | 0.1 MIL | 0.05 MIL |

### 24.1.4 Cartridge Selection for Competition

The dominant PRS/NRL cartridge is the 6.5 Creedmoor, chosen for its combination of low recoil, high BC bullets, and excellent barrel life. For F-Class Open, heavier calibers like the 6mm and 6.5×47 Lapua dominate. F-Class F/TR restricts caliber to .223 Remington and .308 Winchester.

Let us quantify the wind-drift advantage that makes the 6.5 Creedmoor the PRS standard by comparing it to the .308 Winchester at 1000 yards:

Listing 24.1: Wind drift comparison: 6.5 Creedmoor vs. .308 Win at 1000 yards.

```
# 6.5 Creedmoor: 140gr Berger, G7 BC 0.310, 2750 fps
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90 --full

# .308 Win: 175gr SMK, G7 BC 0.253, 2600 fps
ballistics trajectory \
  -v 2600 -b 0.253 -m 175 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90 --full
```

At 1000 yards in a 10 mph crosswind, the 6.5 Creedmoor typically drifts approximately 30% less than the .308 Winchester —a significant advantage that compounds with distance and wind uncertainty. The .308 remains competitive inside 600 yards, and its heavier recoil can be an advantage for spotting misses (the recoil impulse is more noticeable, helping the shooter call the shot).

## 24.2 Building Competition Dope Books

### 24.2.1 Standard Dope Format

A competition dope book is a more detailed version of the hunting dope card. The standard format includes:

- Range (yards or meters) in fine increments (every 10 or 25 yards)
- Elevation correction in MILs (or MOA) to 0.1 MIL resolution
- Wind drift per 1 mph of full-value crosswind (in MILs/MOA)
- Time of flight (for moving target leads)
- Remaining velocity (for transonic awareness)

### 24.2.2 Generating Fine-Grained Dope

Competition shooters need dope at every 10 yards—a much finer resolution than the 50–100-yard increments used for hunting. The `come-ups` subcommand handles this directly:

Listing 24.2: Competition dope book: 6.5 Creedmoor, 140 gr Berger Hybrid.

```
ballistics come-ups \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --zero-distance 100 \
  --start 100 --end 1200 --step 10 \
  --adjustment-unit mil \
  --wind-speed 1 --wind-direction 90
```

By setting `--wind-speed` to 1 mph, the windage column shows the drift per mph of full-value crosswind. To compute the correction for any actual wind, simply multiply by the estimated wind speed and the sine of the wind angle.

**Note**

Setting wind to 1 mph creates a *wind multiplier table*. For an 8 mph crosswind at 45 degrees (wind value = $8 \sin 45^\circ \approx 5.7$ mph), multiply the 1-mph windage value by 5.7. This mental math approach is standard practice in PRS/NRL competition.

### 24.2.3 Truing Your Dope

Published BC values and chronograph velocities are starting points, not ground truth. Every competition shooter must *true* their data—adjust muzzle velocity or BC to match observed impacts at known distances.

The `true-velocity` subcommand computes the effective muzzle velocity that produces a given observed drop:

Listing 24.3: Truing muzzle velocity from observed drop at 600 yards.

```
ballistics true-velocity \
  --measured-drop 6.2 \
  --range 600 \
  -b 0.310 --drag-model g7 \
  -m 140 -d 0.264 \
  --zero-distance 100 \
  --chrono-velocity 2750 \
  --altitude 1000 --temperature 65
```

This tells you: "Given that I observed 6.2 MILs of drop at 600 yards, what muzzle velocity makes my model agree with reality?" The result might be 2735 fps instead of the 2750 fps your chronograph read—a 15 fps correction that makes a meaningful difference at 1000+ yards.

Alternatively, use the `--velocity-adjustment` and `--bc-adjustment` flags on subsequent trajectory commands to apply the correction without recomputing:

Listing 24.4: Applying velocity and BC adjustments to competition dope.

```
ballistics come-ups \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --zero-distance 100 \
  --start 100 --end 1200 --step 10 \
  --adjustment-unit mil
```

**Tip**

True your data at multiple distances—300, 600, and 1000 yards at minimum. If the velocity correction needed differs significantly across distances, the issue is likely BC, not velocity. Use the BC adjustment flag (`--bc-adjustment`) to apply a scaling factor (e.g., 0.97 for a 3% BC reduction).

### 24.2.4 Exporting for External Tools

Most competitors maintain dope in a spreadsheet or specialized app. Export the data in CSV or JSON:

Listing 24.5: Exporting competition dope as CSV and JSON.

```
# CSV for spreadsheets
ballistics come-ups \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --zero-distance 100 \
  --start 100 --end 1200 --step 25 \
  --adjustment-unit mil \
  -o csv > competition_dope.csv

# JSON for apps
ballistics come-ups \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --zero-distance 100 \
  --start 100 --end 1200 --step 25 \
  --adjustment-unit mil \
  -o json > competition_dope.json
```

## 24.3 Wind Calls: Using Monte Carlo for Probability of Hit

Wind is the great equalizer in precision rifle competition. The best dope in the world is meaningless if you misread the wind by 3 mph. Monte Carlo simulation turns the deterministic trajectory solver into a statistical tool for understanding wind uncertainty.

### 24.3.1 Modeling Wind Uncertainty

The `monte-carlo` subcommand varies input parameters randomly around their nominal values and runs hundreds or thousands of trajectory iterations, producing statistical measures of impact dispersion.

The key insight for wind analysis: `--wind-std` controls the standard deviation of the wind speed perturbation. If you estimate the wind at 8 mph but are confident only to within $\pm 3$ mph (one sigma), set the wind standard deviation to 3.

Listing 24.6: Monte Carlo wind analysis at 800 yards.

```
ballistics monte-carlo \
```

```
-v 2750 -b 0.310 -m 140 -d 0.264 \
-n 1000 \
--velocity-std 8 --angle-std 0.05 \
--bc-std 0.003 \
--wind-speed 8 --wind-direction 90 --wind-std 3 \
--target-distance 800
```

The output includes:

- **Mean drop and drift**: The expected impact point.
- **Standard deviations**: The spread in the vertical and horizontal planes.
- **CEP**: The Circular Error Probable—the radius within which 50% of shots fall.
- **95% confidence ellipse**: The semi-major and semi-minor axes of the ellipse containing 95% of impacts.

### 24.3.2 Understanding CEP and the Confidence Ellipse

The Monte Carlo module computes CEP using the `calculate_cep()` function and the 95% confidence ellipse using `calculate_confidence_ellipse()`, both in `src/monte_carlo.rs`.

CEP is computed as the median distance from each impact to the mean point of impact. It provides an intuitive "50% of my shots land within this circle" metric.

The confidence ellipse is more informative because it captures the asymmetry of the dispersion: wind uncertainty elongates the pattern horizontally, while velocity SD and elevation error stretch it vertically. The ellipse parameters are computed from the eigenvalues and eigenvectors of the 2D covariance matrix of the impact points:

$$\mathbf{C} = \begin{pmatrix} \sigma_{xx}^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_{yy}^2 \end{pmatrix} \tag{24.2}$$

The semi-axes of the 95% confidence ellipse are $a = \sqrt{\lambda_1} \cdot \sqrt{5.991}$ and $b = \sqrt{\lambda_2} \cdot \sqrt{5.991}$, where $\lambda_1$ and $\lambda_2$ are the eigenvalues and 5.991 is the chi-square value for 2 degrees of freedom at 95% confidence.

### 24.3.3 Practical Wind-Call Strategy

Use Monte Carlo runs to answer competitive questions:

1. **How much does wind uncertainty cost me?** Run simulations with $\pm 1$, $\pm 2$, and $\pm 3$ mph wind uncertainty. Compare the CEP to the target size.

2. **Should I send it or wait?** If the CEP exceeds the target size, the expected first-round hit probability is below 50%. Wait for a wind condition you can read with more confidence.

3. **What is the cost of a wrong call?** Run a trajectory with the wind 3 mph higher and lower than your call. If both impacts stay on the target, send it.

Listing 24.7: Wind sensitivity analysis: bracketing the wind call.

```
# Your wind call: 8 mph
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 800 \
  --wind-speed 8 --wind-direction 90 --full

# Low estimate: 5 mph
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 800 \
  --wind-speed 5 --wind-direction 90 --full

# High estimate: 11 mph
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 800 \
  --wind-speed 11 --wind-direction 90 --full
```

At 800 yards, a 6.5 Creedmoor with G7 BC `0.310` drifts approximately 0.3 MILs per mph of full-value crosswind. A $\pm 3$ mph error produces a $\pm 0.9$ MIL spread—about 26 inches. If the target is a 2 MOA plate (16.8 inches at 800 yards), the miss probability is significant.

**Warning**

Monte Carlo simulations are only as good as the uncertainty estimates you feed them. If your wind uncertainty is honestly $\pm 5$ mph (common in gusty mountain terrain), the simulation will show large dispersion. Don't reduce the wind SD to feel better about the numbers—use honest estimates to make honest decisions about whether to take the shot.

## 24.4 Stage Planning with First-Round Hit Probability

### 24.4.1 The Time-Accuracy Trade-Off

In PRS/NRL competition, you typically have a limited time (often 60–90 seconds) to engage multiple targets at different distances and from varying positions. The fundamental tactical decision is: *spend time refining the wind call, or send the round and observe?*

The answer depends on the first-round hit probability (FRHP). If FRHP is high (>80%), send it. If FRHP is marginal (40–60%), spending 5 seconds reading a wind indicator may be worth it. If FRHP is very low (<30%), the shot is a low-value gamble.

### 24.4.2 Computing FRHP with Monte Carlo

To compute FRHP, run a Monte Carlo simulation and count the fraction of iterations that land within the target zone. While the `monte-carlo` subcommand does not directly output a hit-probability percentage, you can derive it from the statistics.

For a circular target of radius $r$, the approximate hit probability from the 95% confidence ellipse is:

$$P_{\text{hit}} \approx 1 - \exp\left(-\frac{r^2}{2\sigma_x \sigma_y}\right) \tag{24.3}$$

For more precise results, use the JSON output and post-process with a script:

Listing 24.8: Monte Carlo for FRHP at 600 yards, 2-MOA target.

```
ballistics monte-carlo \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  -n 2000 \
  --velocity-std 8 --angle-std 0.08 \
  --bc-std 0.003 \
  --wind-speed 6 --wind-direction 90 --wind-std 2 \
  --target-distance 600 \
  -o full > mc_600yd.txt
```

The JSON output contains the wind drift and drop for each iteration, which can be used to compute the hit rate for any target size.

### 24.4.3 Stage Sequence Optimization

When a stage presents three targets—say at 400, 700, and 1000 yards—the question is: which order maximizes total hits?

The conventional wisdom is to engage the most uncertain targets first (when you have maximum time for follow-up) and save the highest-FRHP targets for last. However, this depends on your specific load's wind sensitivity at each distance.

Listing 24.9: Comparing wind sensitivity at three stage distances.

```
# Wind drift per mph at 400 yards
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 400 \
  --wind-speed 1 --wind-direction 90 --full

# Wind drift per mph at 700 yards
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 700 \
  --wind-speed 1 --wind-direction 90 --full

# Wind drift per mph at 1000 yards
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 1 --wind-direction 90 --full
```

Wind drift grows non-linearly with distance. At 400 yards, a 6.5 Creedmoor might show 0.1 MIL per mph; at 1000 yards, it is closer to 0.4 MIL per mph. This means wind uncertainty has 4× the angular impact at 1000 yards.

### 24.4.4 Expected Value Decision-Making

Competition scoring is binary: hit or miss. This means the expected value of a shot is simply $P_{\text{hit}} \times 1$ point. If you have 90 seconds and 10 targets, you can afford about 9 seconds per target. The question becomes: if spending 3 extra seconds reading wind increases your FRHP from 60% to 80%, is it worth losing time on the last target?

Model this with Monte Carlo. Run two scenarios—one with your "quick" wind estimate (higher wind SD) and one with your "careful" estimate (lower wind SD):

Listing 24.10: Expected value: quick vs. careful wind read at 700 yards.

```
# Quick wind read: +/- 4 mph uncertainty
ballistics monte-carlo \
```

```
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 8 --angle-std 0.1 \
  --wind-speed 8 --wind-direction 90 --wind-std 4 \
  --target-distance 700

# Careful wind read: +/- 2 mph uncertainty
ballistics monte-carlo \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 8 --angle-std 0.1 \
  --wind-speed 8 --wind-direction 90 --wind-std 2 \
  --target-distance 700
```

If the quick read produces a CEP of 12 inches and the careful read produces 7 inches against an 18-inch target, the FRHP improvement may be from roughly 75% to 90%—a gain of 0.15 points per shot. Whether that 0.15 points justifies 3 seconds depends on how many targets remain and how tight the stage timer is.

### 24.4.5 Transonic Awareness for Competition

As the bullet decelerates through the transonic zone (Mach 1.0–1.2, roughly 1100–1340 fps), aerodynamic forces change rapidly and can destabilize the bullet. This is the single biggest accuracy concern at extreme competition ranges.

Use the trajectory output's velocity column to identify when your bullet enters the transonic zone:

Listing 24.11: Identifying transonic range for 6.5 Creedmoor.

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1500 \
  --sample-trajectory --sample-interval 50 --full
```

For a 6.5 Creedmoor 140-grain bullet at 2750 fps, the transonic transition typically occurs around 1200–1300 yards at sea level, and farther at higher altitude (where the speed of sound is lower). The .308 Win 175 SMK at 2600 fps enters the transonic zone much earlier—around 900–1000 yards.

**Warning**

Once a bullet enters the transonic zone, trajectory prediction becomes unreliable. The drag model assumptions (G1 or G7) break down, and the bullet may begin to yaw. Treat any target beyond the transonic transition as a low-probability shot. In PRS/NRL competition, this is usually only relevant for targets beyond 1200 yards with standard cartridges.

## 24.5 Spin Drift and Coriolis at Competition Ranges

### 24.5.1 When Spin Drift Matters

Spin drift is the lateral deflection caused by the bullet's spin interacting with aerodynamic forces. For a right-hand twist barrel (standard for most rifles), spin drift pushes the bullet to the right. The effect is small at moderate distances but becomes significant beyond 600 yards.

The magnitude depends on the gyroscopic stability factor and the time of flight. The `ballistics-engine` implements spin drift using Bryan Litz's empirical formula in `src/spin_drift.rs`:

$$\Delta x = 1.25 \cdot (S_g + 1.2) \cdot t^{1.83} \tag{24.4}$$

where $S_g$ is the gyroscopic stability factor and $t$ is the time of flight in seconds. The result is in inches for a right-hand twist; left-hand twist reverses the sign.

The gyroscopic stability factor is computed dynamically using the Miller stability formula in `calculate_dynamic_stabi` (also in `src/spin_drift.rs`), which accounts for velocity, air density, bullet geometry, and twist rate.

Enable spin drift with the `--enable-spin-drift` flag:

Listing 24.12: Spin drift at 1000 yards, 6.5 Creedmoor.

```
# Without spin drift
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# With spin drift enabled
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full \
  --enable-spin-drift \
  --twist-rate 8 --twist-right
```

For a 6.5 Creedmoor with a 1:8 twist, spin drift at 1000 yards is typically 6–10 inches (0.15–0.25 MIL) to the right. At 1200 yards, it can exceed 12 inches.

**Note**

In PRS/NRL competition, spin drift is often absorbed into the truing process—when you true your data at long range, the velocity or BC correction implicitly includes spin drift if you did not model it explicitly. However, explicitly modeling spin drift produces a more accurate correction, especially if conditions change (e.g., different altitude affecting the stability factor).

### 24.5.2 Coriolis Effect at Competition Ranges

The Coriolis effect causes lateral deflection due to the Earth's rotation. Its magnitude depends on latitude, shot direction (azimuth), and time of flight.

Enable Coriolis with `--enable-coriolis` and provide your latitude:

Listing 24.13: Coriolis effect at 1000 yards, shooting north at 40°N latitude.

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full \
  --enable-coriolis \
  --latitude 40 --shot-direction 0
```

The Coriolis implementation in `src/trajectory_solver.rs` projects Earth's rotation vector into the shooter's local frame using latitude and azimuth:

$$\boldsymbol{\Omega} = \Omega_E \begin{pmatrix} \cos\phi \sin\alpha \\ \sin\phi \\ \cos\phi \cos\alpha \end{pmatrix} \tag{24.5}$$

where $\Omega_E = 7.2921 \times 10^{-5}$ rad/s is the Earth's rotation rate, $\phi$ is the latitude, and $\alpha$ is the shot azimuth.

At 1000 yards and 40°N latitude, the Coriolis deflection is typically 2–4 inches horizontally—smaller than spin drift, but not negligible for a 1 MOA target. The vertical component (Eötvös effect, discussed in Chapter 15) is even smaller.

### 24.5.3 Combined Advanced Effects

For maximum precision at 1000+ yards, enable both spin drift and Coriolis:

Listing 24.14: Full advanced-effects trajectory for competition.

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 --full \
  --enable-spin-drift \
  --twist-rate 8 --twist-right \
  --enable-coriolis \
  --latitude 38 --shot-direction 270 \
  --sample-trajectory --sample-interval 100
```

At 1200 yards, the combined spin drift and Coriolis can exceed 15 inches of lateral deflection—enough to miss a 1 MOA target entirely if unaccounted for.

**Tip**

For F-Class competition at 1000 yards, the X-ring is only 5 inches in diameter. At this level of precision, spin drift alone can move the group by more than one X-ring diameter. Enabling `--enable-spin-drift` with your actual twist rate is strongly recommended for F-Class shooters.

## 24.6 Does Your Scope Track?

A come-up table assumes that dialing 10 mils moves the point of impact by exactly 10 mils. Scopes do not always oblige. A turret that under-tracks by 5% is not broken enough to notice at 300 yards and is half a mil out at 1000—and no amount of truing the *load* will fix it, because the error is in the optic, not the bullet.

The tall-target test measures it. Hang a plumb line with marks a known distance apart, dial a known amount of elevation, and measure how far the group actually moved:

Listing 24.15: Computing a tracking correction factor

```
# Dialed 10 mil at 100 yd; the group moved 34.2 inches.
# 10 mil at 100 yd should be exactly 36.0 inches.
ballistics tall-target --dialed 10 --unit mil \
  --measured 34.2 --range 100
```

Listing 24.16: The correction factor

```
Tall-Target Test Result
  Dialed travel:   10.00 MIL
  Actual travel:   9.50 MIL (34.20 in at 100 yd)
```

```
  Correction factor (actual / dialed): 0.9500
  Enter this as --elevation-cf / profile elevation_cf (or the windage equivalents);
    dial solutions are divided by it.
```

**Tracking Correction Factor**

The *correction factor* is Litz's published tall-target ratio:

$$\mathrm{CF} = \frac{\text{actual measured travel}}{\text{dialed travel}}$$

0.95 means the scope under-tracks by 5%. Every dial-unit output is *divided* by it exactly once—an under-tracking scope needs *more* dial—and the factor is constrained strictly between 0.5 and 1.5, because anything outside that is a measurement error, not a scope.

Feed the result back with `--elevation-cf` and `--windage-cf`, or store it on the profile as `elevation_cf` / `windage_cf`:

Listing 24.17: A come-up table corrected for a 5% under-tracking turret

```
ballistics come-ups --velocity 2700 --bc 0.462 \
  --mass 168 --diameter 0.308 \
  --zero-distance 200 --sight-height 1.5 \
  --start 300 --end 700 --step 200 \
  --elevation-cf 0.95
```

The correction is applied at one shared conversion boundary, so it reaches every dialed quantity and nothing else: come-ups, the range-table and `compare` Drop and Wind columns, wind-card drift, the mover lead and Ring column, the PDF dope-card rows, and the `zero` command's MOA and mrad outputs. **Raw drop and drift in inches are never scaled**, and neither are degree bore-angle echoes—they are not dial readings. Omitting the flags is byte-identical to earlier releases.

**Do Not Bake Tracking Error Into Your BC**

If your scope under-tracks and you true against dialed observations, the fit will move your muzzle velocity or BC to absorb an error that has nothing to do with the bullet. `true-velocity --elevation-cf` exists for exactly this: it *multiplies* dialed mil or MOA observations by the CF before the fit, converting scope-dial units back to true angular, and then shows dial-unit report values back in scope units. Linear `in` drops are never scaled, because a tape measure does not care how your turret tracks.

## 24.7 Making Power Factor

Action-shooting disciplines score a load's momentum, not its energy, and they draw hard lines: fail the threshold and your stage scores drop a division or your entry is disallowed outright. The arithmetic is trivial and the consequences are not, which is exactly the kind of thing worth having a command for:

Listing 24.18: Checking a 9mm load across every organization

```
ballistics power-factor --weight 124 --velocity 1150
```

Listing 24.19: Power-factor output

```
Power Factor
============
  Weight:              124.0 gr
  Velocity:            1150.0 fps
  Power factor (raw):  142.60
  Power factor (scored): 142

  Org       Class                          Min PF    Pass      Velocity limit
  USPSA     Minor                             125    PASS                   -
  USPSA     Major                             165    FAIL                   -
  IDPA      BUG                                95    PASS                   -
  IDPA      Stock Revolver / CCP              105    PASS                   -
  IDPA      SSP / ESP / CO                    125    PASS                   -
  IDPA      PCC                               135    PASS                   -
  IDPA      Enhanced Revolver                 155    FAIL                   -
  IDPA      CDP                               165    FAIL                   -
  SASS      Smokeless - Pistol-Revolver        60    FAIL        400-1000 fps
  SASS      Smokeless - Rifle                  60    PASS        400-1400 fps
```

**Power Factor**

*Power factor* is bullet weight in grains multiplied by muzzle velocity in fps, divided by 1000. It is a momentum proxy: a heavy bullet moving slowly and a light bullet moving fast can make the same number, which is the point—the rules are trying to equalise recoil, not energy.

Note the two figures. The raw value is 142.60; the *scored* value is 142, because the rulebooks truncate rather than round. A load computing to 164.9 does not make Major. Note too that the SASS rows carry velocity limits as well as a floor—some divisions cap velocity for steel-target safety, so passing the power factor is necessary but not sufficient.

**SAFETY: Chronograph at the Match, in Match Conditions**

The number that counts is the one the match chronograph reads on the day, from your gun, with your ammunition, at that morning's temperature. Build in margin: a load that computes to exactly the threshold will fail as often as it passes, and a cold morning can move a powder-sensitive load several fps per degree (Section 25.6.3). Always verify computational predictions against real-world chronograph data before relying on them.

## 24.8 Competition Workflow: Pre-Match Preparation

Bringing all the pieces together into a repeatable pre-match workflow is what separates prepared competitors from those who wing it. Here is a structured approach using `ballistics-engine`.

### 24.8.1 Step 1: True Your Data

At least two weeks before a match, schedule a range session at a known-distance range with conditions as close to the match venue as possible.

1. Chronograph 20+ rounds to establish mean velocity and standard deviation.
2. Shoot at 100 yards to confirm zero.
3. Shoot groups at 300, 600, and 1000 yards (or as far as the range allows).
4. Record the actual drop in MILs at each distance.
5. Record atmospheric conditions (temperature, pressure, altitude, humidity).
6. Use the `true-velocity` subcommand to determine the effective MV:

Listing 24.20: Truing from observed data at three distances.

```
# True from 600-yard observed drop
ballistics true-velocity \
  --measured-drop 4.8 \
  --range 600 \
  -b 0.310 --drag-model g7 \
  -m 140 -d 0.264 \
  --zero-distance 100 \
  --chrono-velocity 2750 \
  --altitude 1000 --temperature 65

# Verify at 1000 yards
ballistics true-velocity \
  --measured-drop 9.2 \
```

```
  --range 1000 \
  -b 0.310 --drag-model g7 \
  -m 140 -d 0.264 \
  --zero-distance 100 \
  --chrono-velocity 2750 \
  --altitude 1000 --temperature 65
```

If the trued velocity differs significantly between distances, apply a BC correction rather than (or in addition to) a velocity correction.

### 24.8.2 Step 2: Save a Profile

Store your trued data in a profile so you don't have to re-enter it every time:

Listing 24.21: Saving a competition profile.

```
ballistics profile save "65CM_Match" \
  --velocity 2735 \
  --bc 0.310 \
  --drag-model g7 \
  --mass 140 \
  --diameter 0.264 \
  --twist-rate 8 \
  --zero-distance 100
```

Then use the profile for all subsequent commands:

Listing 24.22: Using a saved profile for come-ups.

```
ballistics come-ups \
  --profile "65CM_Match" \
  --start 100 --end 1200 --step 10 \
  --adjustment-unit mil \
  --wind-speed 1 --wind-direction 90
```

### 24.8.3 Step 3: Build Match-Specific Dope

On the day before (or morning of) the match, update your dope for the actual conditions at the match venue:

Listing 24.23: Match-day dope with actual conditions.

```
ballistics come-ups \
  --profile "65CM_Match" \
```

```
  --start 200 --end 1200 --step 25 \
  --adjustment-unit mil \
  --wind-speed 1 --wind-direction 90 \
  --altitude 4500 --temperature 82 \
  --pressure 25.12 --humidity 35
```

### 24.8.4 Step 4: Run Wind Scenarios

For the expected wind range at the match venue, pre-compute wind holds:

Listing 24.24: Pre-computing wind holds for common wind speeds.

```
# 5 mph crosswind
ballistics come-ups \
  --profile "65CM_Match" \
  --start 200 --end 1200 --step 100 \
  --adjustment-unit mil \
  --wind-speed 5 --wind-direction 90 \
  --altitude 4500 --temperature 82

# 10 mph crosswind
ballistics come-ups \
  --profile "65CM_Match" \
  --start 200 --end 1200 --step 100 \
  --adjustment-unit mil \
  --wind-speed 10 --wind-direction 90 \
  --altitude 4500 --temperature 82

# 15 mph crosswind
ballistics come-ups \
  --profile "65CM_Match" \
  --start 200 --end 1200 --step 100 \
  --adjustment-unit mil \
  --wind-speed 15 --wind-direction 90 \
  --altitude 4500 --temperature 82
```

### 24.8.5 Step 5: Monte Carlo Confidence Check

For critical stages (e.g., known targets that are high-value or at the edge of your effective range), run Monte Carlo simulations to assess hit probability:

Listing 24.25: Monte Carlo confidence check for a difficult stage target.

```
ballistics monte-carlo \
```

```
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  -n 2000 \
  --velocity-std 8 --angle-std 0.1 \
  --bc-std 0.003 \
  --wind-speed 8 --wind-direction 90 --wind-std 3 \
  --target-distance 1000
```

If the CEP exceeds the target radius, you know that target will be a coin-flip at best. Plan your time budget accordingly: spend less time on high-FRHP close targets and invest the saved seconds on reading the wind for the far targets.

### 24.8.6 Step 6: Print and Prepare

Generate PDF dope cards for your range card sleeve:

Listing 24.26: Final competition dope card in PDF.

```
ballistics trajectory \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --altitude 4500 --temperature 82 --pressure 25.12 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 25 \
  -o pdf --output-file match_dope.pdf \
  --bullet-name "140gr Berger Hybrid" \
  --location-name "PRS Regional - Spring Match" \
  --font-preset medium
```

**Note**

Many competitors tape a simplified dope card (elevation only, every 25 yards) to the rifle stock for rapid reference, while keeping the detailed card (with wind values, energy, and velocity) in a data book for pre-stage planning.

## Exercises

1. **Dope resolution.** Generate come-up tables for the 6.5 Creedmoor competition load in both 10-yard and 25-yard increments from 100 to 1200 yards. At 800 yards, how much elevation change occurs per 10 yards of range? Is 10-yard dope resolution necessary, or does 25-yard resolution suffice for a 2 MOA target?

```
ballistics come-ups \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --zero-distance 100 \
  --start 750 --end 850 --step 10 \
  --adjustment-unit mil
```

2. **Wind sensitivity comparison.** Compare the wind drift per mph at 1000 yards for a 6.5 Creedmoor (140 gr, G7 BC 0.310, 2750 fps) and a .308 Win (175 gr SMK, G7 BC 0.253, 2600 fps). How much advantage does the 6.5 Creedmoor provide in wind?

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 1 --wind-direction 90 --full

ballistics trajectory \
  -v 2600 -b 0.253 -m 175 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --wind-speed 1 --wind-direction 90 --full
```

3. **Monte Carlo target sizing.** Run Monte Carlo simulations at 800 yards with your competition load using wind SDs of 1, 2, 3, and 5 mph. At what wind uncertainty does the CEP exceed an 18-inch (roughly 2 MOA) target radius?

```
ballistics monte-carlo \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 8 --angle-std 0.08 \
  --wind-speed 8 --wind-direction 90 --wind-std 1 \
  --target-distance 800

ballistics monte-carlo \
  -v 2735 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 8 --angle-std 0.08 \
  --wind-speed 8 --wind-direction 90 --wind-std 5 \
  --target-distance 800
```

4. **Spin drift quantification.** Compute spin drift at 100-yard intervals from 200 to 1200 yards for a 1:8 twist 6.5 Creedmoor. At what distance does spin drift exceed 0.1 MIL?

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --enable-spin-drift \
  --twist-rate 8 --twist-right \
  --sample-trajectory --sample-interval 100 --full
```

5. **Coriolis at different azimuths.** At 40°N latitude, compute the Coriolis deflection at 1000 yards for shots fired north, east, south, and west. Which direction shows the largest deflection, and why?

```
# North (azimuth 0)
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --enable-coriolis --latitude 40 --shot-direction 0 --full

# East (azimuth 90)
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --enable-coriolis --latitude 40 --shot-direction 90 --full
```

6. **Full pre-match workflow.** Using any cartridge of your choice, execute the complete six-step pre-match workflow described in Section 24.8. Generate a PDF dope card for a hypothetical match at 5000 feet altitude, 75°F, and 10–15 mph winds.

## What's Next

Competition demands the tightest possible trajectory predictions and a disciplined workflow for turning data into decisions under time pressure. In Chapter 25, we turn to *building* the ammunition that feeds your competition rifle—using `ballistics-engine` to guide bullet selection, evaluate velocity consistency, and predict how handload changes affect downrange performance.

# Chapter 25

# Load Development

> *"Handloading is the science of controlling every variable between the primer and the target. A ballistics solver lets you see which variables matter most."*

For the handloader, `ballistics-engine` is both a prediction tool and a design tool. Before you seat the first bullet or charge the first case, you can explore questions like: *How much velocity do I lose by cutting three inches off my barrel? Which bullet offers the best BC-to-recoil trade-off for my application? If my velocity SD is 12 fps versus 6 fps, how much does it matter at* 1000 *yards?* And after you fire the first ladder test, the engine helps you analyze the results.

This chapter walks through the handloader's workflow with `ballistics-engine`: predicting velocities, selecting bullets, analyzing ladder tests, quantifying the impact of velocity consistency, and—most importantly—working up loads safely.

**SAFETY: Safety Warning**

This chapter discusses using ballistics calculations to *analyze* and *predict the downrange effects* of handloaded ammunition. It does *not* provide load data (powder charges, cartridge dimensions, or pressure limits). Always consult a reputable reloading manual—such as those published by Hodgdon, Sierra, Nosler, or Hornady—for starting loads and maximum charges. Never exceed published maximum loads. A ballistics solver cannot detect or predict dangerous pressures; only a pressure barrel or strain gauge can do that.

## 25.1 Predicting Muzzle Velocity from Barrel Length

### 25.1.1 The Barrel Length–Velocity Relationship

One of the most common questions a handloader faces when building a new rifle is: "How much velocity will I gain (or lose) with a different barrel length?" The relationship between barrel length and muzzle velocity follows an approximately logarithmic curve.

A useful empirical rule, originally derived from extensive chronograph testing, estimates the velocity change per inch of barrel length. For most centerfire rifle cartridges:

$$\Delta v \approx 25\text{–}50 \text{ fps per inch} \tag{25.1}$$

The exact value depends on cartridge capacity, bore diameter, and powder type. Overbore magnums (like the .300 RUM) show values at the low end ($\sim$25 fps/inch) because the powder is largely consumed before the bullet reaches the muzzle. Efficient cartridges (like the 6.5 Creedmoor) show values at the high end ($\sim$40+ fps/inch) because they are still generating meaningful pressure at the muzzle.

### 25.1.2 Modeling the Effect

While `ballistics-engine` does not directly model internal ballistics (barrel length, powder burn rate, chamber pressure), you can use it to *evaluate the downrange consequences* of velocity changes. The typical workflow is:

1. Estimate the muzzle velocity for your barrel length using the empirical rule or published data.
2. Run the trajectory at that velocity.
3. Compare against a reference barrel length.

For example, a 6.5 Creedmoor that produces 2750 fps from a 24-inch barrel might produce approximately 2670 fps from a 22-inch barrel (losing about 40 fps per inch $\times$ 2 inches = 80 fps).

Listing 25.1: Comparing 24-inch vs. 22-inch barrel for 6.5 Creedmoor.

```
# 24-inch barrel: 2750 fps
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 --full

# 22-inch barrel: ~2670 fps (estimated)
ballistics trajectory \
  -v 2670 -b 0.310 -m 140 -d 0.264 \
```

```
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --sample-trajectory --sample-interval 100 --full
```

At 1000 yards, the 80 fps loss produces approximately 0.3–0.5 MILs more drop and slightly more wind drift. Whether that trade-off is acceptable for a lighter, handier rifle is a judgment call—but now it is an *informed* judgment.

### 25.1.3 Short Barrels for Hunting

Mountain hunters increasingly favor short barrels (18–20 inches) for weight savings and maneuverability. The velocity penalty is real but manageable. Consider a .308 Win that produces 2650 fps from a 20-inch barrel versus 2700 fps from a 24-inch barrel:

Listing 25.2: Short-barrel .308 Win for mountain hunting.

```
# 20-inch barrel: 2650 fps
ballistics mpbr \
  -v 2650 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 8

# 24-inch barrel: 2700 fps
ballistics mpbr \
  -v 2700 -b 0.475 -m 165 -d 0.308 \
  --vital-zone 8
```

The MPBR difference between the two barrel lengths is typically only 5–10 yards—negligible in the field. The energy-limited range changes by a similarly modest amount. For a mountain rifle that will be used inside 400 yards, the short barrel is an excellent trade-off.

## 25.2 BC Optimization: Selecting Bullets for Your Application

### 25.2.1 The BC–Weight Trade-Off

A heavier bullet in a given caliber generally has a higher BC because BC is proportional to mass (see Equation (11.2)). But heavier bullets are typically loaded to lower velocities to stay within safe pressure limits. The question is: *Does the higher BC compensate for the lower velocity?*

The answer depends on range. At shorter distances, velocity dominates (less time of flight = less drop). At longer distances, BC dominates (less velocity decay = less accumulated drop).

The `generate-bc-segments` subcommand helps analyze bullet types by estimating velocity-dependent BC degradation profiles based on bullet design characteristics. The `BCSegmentEstimator` in `src/bc_estimation.r`

classifies bullets by type (match boat-tail, hunting, VLD, FMJ, etc.) and applies empirically derived degradation curves:

Listing 25.3: Generating BC segments for different bullet types.

```
# Match boat-tail (minimal BC degradation)
ballistics generate-bc-segments \
  -b 0.475 -m 168 -d 0.308 \
  --model "168gr SMK BT" --drag-model g1

# Hunting boat-tail (moderate degradation)
ballistics generate-bc-segments \
  -b 0.465 -m 165 -d 0.308 \
  --model "165gr AccuBond" --drag-model g1

# VLD (very low drag, most stable BC)
ballistics generate-bc-segments \
  -b 0.640 -m 140 -d 0.264 \
  --model "140gr Berger VLD" --drag-model g1
```

The output shows BC values for different velocity bands. VLD and hybrid bullets maintain BC within 3–5% across the full velocity range, while hunting flat-base designs may lose 15–20% of their BC at subsonic velocities.

### 25.2.2 Comparing Bullet Choices

Let us compare three popular .308-caliber bullet choices for long-range target shooting:

Listing 25.4: Comparing three .308 bullets at 1000 yards.

```
# 155gr Palma, G7 BC 0.235, 2950 fps (fast and light)
ballistics trajectory \
  -v 2950 -b 0.235 -m 155 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# 175gr SMK, G7 BC 0.253, 2600 fps (classic match bullet)
ballistics trajectory \
  -v 2600 -b 0.253 -m 175 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# 185gr Berger Juggernaut, G7 BC 0.283, 2550 fps (heavy match)
ballistics trajectory \
  -v 2550 -b 0.283 -m 185 -d 0.308 \
```

```
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

Compare the drop, wind drift, and remaining energy at 1000 yards. The 155-grain bullet starts fast but loses velocity rapidly due to lower BC; the 185-grain starts slower but arrives with the best retained velocity and least wind drift. The 175 SMK sits in the middle—a balanced choice that explains its enduring popularity.

### 25.2.3 Side by Side: the `compare` Command

Running three trajectories and reading three tables works, but it puts the burden of alignment on you—and it is easy to change one flag in one run without meaning to. Since v0.27.0, `compare` does the whole thing in a single invocation, zeroing each load independently at a shared distance and then running all of them through identical wind and atmosphere:

Listing 25.5: The same three bullets, one command.

```
ballistics compare \
  --load "155 Palma:g7:0.235:155:2950:0.308" \
  --load "175 SMK:g7:0.253:175:2600:0.308" \
  --load "185 Juggernaut:g7:0.283:185:2550:0.308" \
  --zero-distance 100 --sight-height 1.5 \
  --start 600 --end 1000 --step 200
```

Listing 25.6: Comparison output.

```
Load Comparison
Zero: 100 yd | Wind: 10 mph @ 90° | Sight: 1.5 in | Alt: 0 ft

  #1 155 Palma: G7 BC 0.235, 155 @ 2950 fps (dia 0.308)
  #2 175 SMK: G7 BC 0.253, 175 @ 2600 fps (dia 0.308)
  #3 185 Juggernaut: G7 BC 0.283, 185 @ 2550 fps (dia 0.308)

   Range |         155 Palma          |          175 SMK           |       185
    Juggernaut
    (yd) |     Drop    Drift      Vel |     Drop    Drift      Vel |     Drop
    Drift      Vel
-----------------------------------------------------------------------------------------------

     600 |     3.64    -1.31     1854 |     4.76    -1.45     1646 |     4.76
    -1.30     1697
     800 |     5.98    -1.90     1546 |     7.73    -2.12     1375 |     7.60
    -1.88     1450
```

```
    1000 |     9.02    -2.63     1263 |    11.58    -2.92     1130 |    11.16
    -2.56     1221
-------------------------------------------------------------------------------------------------

Drop/Drift in MIL (- is down/left), Vel in fps.
```

The numbers tell the story cleanly. At 1000 yards the fast, light Palma bullet has the least drop by a wide margin—but the heavy Juggernaut, which started 400 fps slower, is still moving faster and drifting less. If your match is decided by wind calls rather than elevation, that trade is worth taking.

A `--load` spec is `NAME:DRAG:BC:MASS:VELOCITY[:DIAMETER]`, and the command accepts between two and eight of them. Saved profiles can be mixed in with `--profile`, and—unlike the scalar-BC caveat that applies to `range-table`—a profile load carries its velocity-BC segments and its custom $C_D(M)$ curve into both the zeroing run and the trajectory run. The table legend tags those loads and the JSON flags them. JSON output adds energy, time of flight, each load's zero angle, and per-row deltas against load #1; CSV emits one sanitized column group per load.

### 25.2.4 Sectional Density as a Selection Guide

Sectional density (SD) is the ratio of bullet mass to the square of its diameter. It is a useful proxy for both BC and penetration potential. The `BCSegmentEstimator` computes it with the `calculate_sectional_density()` function in `src/bc_estimation.rs`:

$$\text{SD} = \frac{m_{\text{gr}}}{7000 \times d^2} \tag{25.2}$$

where $m_{\text{gr}}$ is bullet mass in grains and $d$ is diameter in inches. The constant 7000 converts grains to pounds.

Bullets with SD above 0.250 are considered to have excellent penetration characteristics. Bullets with SD above 0.300 are exceptional:

| Bullet | Weight (gr) | Diameter (in) | SD |
|---|---|---|---|
| .224 77 gr SMK | 77 | 0.224 | 0.219 |
| 6.5mm 140 gr Berger | 140 | 0.264 | 0.287 |
| .308 175 gr SMK | 175 | 0.308 | 0.264 |
| .308 185 gr Berger | 185 | 0.308 | 0.279 |
| .338 300 gr SMK | 300 | 0.338 | 0.375 |

High-SD bullets tend to have correspondingly high BCs (because both are proportional to mass and inversely related to frontal area), but the relationship is not linear—bullet shape (the form factor) plays an equally important role.

**Tip**

When choosing between bullets for a specific application, run trajectories for your top 2–3 candidates using `ballistics-engine`. Compare not just drop, but also wind drift, transonic range (when does the bullet go subsonic?), and remaining energy. The best bullet on paper is the one that meets all your requirements simultaneously.

### 25.2.5 G1 vs. G7: Choosing the Right Drag Model for Your Bullet

The choice of drag model—G1 or G7—affects the accuracy of your trajectory prediction, especially at long range. The G1 reference projectile is a flat-base, blunt-ogive shape. The G7 reference is a boat-tail, secant-ogive shape that closely resembles modern match bullets.

For load development, the drag model matters because BC values are *not interchangeable* between G1 and G7. A bullet with a G1 BC of `0.475` might have a G7 BC of approximately `0.240`. The G7 value will be more stable across the velocity range for a boat-tail bullet, meaning your trajectory predictions remain accurate from muzzle velocity down through the transonic zone.

Listing 25.7: Same bullet, G1 vs. G7 drag model comparison.

```
# .308 Win 175gr SMK using G1 BC
ballistics trajectory \
  -v 2600 -b 0.505 -m 175 -d 0.308 \
  --drag-model g1 \
  --auto-zero 100 --max-range 1000 --full

# Same bullet using G7 BC
ballistics trajectory \
  -v 2600 -b 0.253 -m 175 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

The two models should agree at moderate ranges but may diverge by 1–3 inches at 1000 yards. The G7 model is generally more accurate for modern boat-tail bullets because the reference projectile better matches the actual bullet shape, resulting in a more constant BC across the velocity envelope.

**Note**

When comparing bullets from different manufacturers, ensure you are comparing the same drag model. A G1 BC of `0.475` and a G7 BC of `0.253` may describe the same bullet. Use the `--drag-model` flag to specify which model your BC values are referenced to. The approximate conversion factor between G1 and G7 for boat-tail bullets is $G7 \approx 0.5 \times G1$, but the exact ratio varies with bullet shape.

### 25.2.6 The Crossover Range

When comparing a lighter, faster bullet against a heavier, slower bullet with higher BC, there is often a *crossover range*—a distance at which the heavier bullet overtakes the lighter one in terms of retained velocity, drop, or wind drift.

Listing 25.8: Finding the crossover range: 130gr vs. 140gr in 6.5 Creedmoor.

```
# 130gr at 2850 fps, lower BC
ballistics trajectory \
  -v 2850 -b 0.290 -m 130 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 100 --full

# 140gr at 2750 fps, higher BC
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 10 --wind-direction 90 \
  --sample-trajectory --sample-interval 100 --full
```

For this comparison, the 130-grain bullet has less drop inside approximately 400–500 yards (where its velocity advantage dominates), but the 140-grain overtakes it in both drop and wind drift beyond that range. If your shooting is primarily inside 600 yards, the lighter bullet may be the better choice. Beyond 800 yards, the heavier bullet wins decisively.

## 25.3 Ladder Test Analysis with `ballistics-engine`

### 25.3.1 What Is a Ladder Test?

A ladder test (also known as an OCW, or Optimal Charge Weight, test) is a load development method in which the handloader fires groups at incrementally increasing powder charges. The goal is to identify a *node*—a range of charge weights where group size is minimized and point of impact is stable.

The underlying principle is that at certain charge weights, the barrel vibration harmonics are such that the bullet exits the muzzle at a consistent point in the barrel's vibrational cycle. At these "sweet spots," small variations in charge weight produce minimal changes in muzzle velocity and point of impact.

### 25.3.2 Using the Solver for Analysis

After firing a ladder test, you have a set of charge weights, measured velocities (from a chronograph), and impact positions on the target. `ballistics-engine` adds value in two ways:

1. **Predicting group size at distance:** If you fire the ladder at 100 yards, the solver can predict how the velocity spread at each charge weight translates into vertical dispersion at 600+ yards.
2. **Truing the BC:** By comparing predicted drop to observed impact positions, you can estimate the true BC of your actual handloaded rounds.

Suppose your ladder test at 100 yards with a 6.5 Creedmoor, 140-grain Berger Hybrid over Hodgdon H4350 produced the following data at the node (43.0–43.5 grains):

| Charge (gr) | Velocity (fps) | SD (fps) |
|---|---|---|
| 42.5 | 2710 | 14 |
| 43.0 | 2740 | 6 |
| 43.2 | 2750 | 5 |
| 43.5 | 2765 | 8 |
| 44.0 | 2790 | 15 |

The node at 43.0–43.5 grains shows the lowest velocity SD. Now use the solver to predict how this SD translates to long-range performance:

Listing 25.9: Predicting vertical spread from velocity SD at the node.

```
# Low end of the node: 2740 fps
ballistics trajectory \
  -v 2740 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# High end of the node: 2765 fps
ballistics trajectory \
  -v 2765 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

The difference in drop between 2740 fps and 2765 fps at 1000 yards indicates the vertical spread contributed by the 25 fps velocity range within the node. Compare this to the spread predicted for the 42.5-grain charge (velocity range $2710 \pm 14$ fps = roughly 2682–2738 fps):

Listing 25.10: Comparing nodes: tight SD vs. wide SD at 1000 yards.

```
# Tight node (43.2 gr): 2750 fps +/- 5 fps
```

```
ballistics trajectory \
  -v 2745 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

ballistics trajectory \
  -v 2755 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# Wide charge (42.5 gr): 2710 fps +/- 14 fps
ballistics trajectory \
  -v 2696 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

ballistics trajectory \
  -v 2724 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

### 25.3.3 Estimating BC from Ladder Data

If you fire your ladder test at multiple distances (or if you have precise drop data from a known distance), you can estimate the actual BC of your handloaded rounds using the `estimate-bc` subcommand:

Listing 25.11: Estimating BC from observed drops at two distances.

```
ballistics --units metric estimate-bc \
  -v 823 -m 9.07 -d 6.71 \
  --distance1 300 --drop1 0.483 \
  --distance2 600 --drop2 2.41
```

**Note**

The `estimate-bc` subcommand currently expects metric inputs (velocity in m/s, mass in kg, diameter in meters, distances and drops in meters). Convert your field measurements accordingly. For example, 2750 fps = 838.2 m/s, 140 gr = 0.00907 kg, 0.264 in = 0.00671 m.

The output includes both the estimated BC and a verification showing how closely the estimated BC reproduces the observed drops. A verification error below 2% indicates a reliable estimate.

## 25.4 Free Recoil

Recoil is a load-development variable, not just a comfort question. It drives how well you spot your own impacts, how fast you can run a stage, and how long a shooter stays behind the rifle before flinching creeps in. Since v0.29.0 the engine computes it from SAAMI's own momentum balance:

Listing 25.12: Free recoil of a .308 match load.

```
ballistics recoil \
  --bullet-weight 168 --charge-weight 43 \
  --velocity 2700 --firearm-weight 8.5
```

Listing 25.13: Recoil output.

```
Free Recoil (SAAMI Momentum Balance)
====================================
  Firearm weight:      8.50 lb
  Bullet weight:       168.0 gr
  Charge weight:       43.0 gr
  Muzzle velocity:     2700.0 fps
  Gas velocity model:  saami-rifle  (4725.0 fps)

  Recoil velocity:     11.04 fps
  Recoil energy:       16.09 ft-lb
  Recoil impulse:      2.916 lb-s
```

**Free Recoil**

*Free recoil* is the energy the firearm would carry rearward if it were unrestrained—nothing held it, nothing absorbed it. It is a property of the cartridge and the gun's mass, not of the shooter, which is exactly what makes it comparable between loads. What you actually feel depends on stock geometry, recoil pad, muzzle device and body position, none of which this number knows about.

The ejecta term is the part people forget. Recoil is not just the bullet leaving—the propellant gas leaves too, and faster. The *gas velocity model* line names the convention in use; `saami-rifle` assumes the gas departs at 1.75 times muzzle velocity, which for this load is 4725 fps. Forty-three grains of powder at 4725 fps carries meaningful momentum, and a model that omitted it would under-report recoil by roughly a fifth.

**Compare, Do Not Predict**

Use recoil figures to rank loads against each other, not to predict how a rifle will feel. A 16 ft-lb load in a heavy prone rifle with a brake is a different experience from the same 16 ft-lb in a lightweight mountain gun—but the 16 versus 22 comparison between two loads in the *same* rifle is real and useful.

## 25.5 Velocity SD and Its Impact on Long-Range Accuracy

### 25.5.1 Why Velocity Consistency Matters

Velocity standard deviation (SD) is the single most important measure of ammunition consistency for long-range shooting. Every fps of velocity variation translates to a corresponding variation in drop at the target. The further the target, the larger the effect.

**Velocity SD and Extreme Spread**

*Standard deviation* (SD) is the statistical measure of spread around the mean. For a normal distribution, 68% of rounds fall within $\pm 1$ SD and 95% fall within $\pm 2$ SD. *Extreme spread* (ES) is the difference between the fastest and slowest rounds in a string. For typical 10-round strings, ES $\approx 3.1 \times$ SD.

### 25.5.2 Quantifying the Effect

The Monte Carlo subcommand directly models velocity SD. Let us compare ammunition with 5 fps SD (excellent handloads) versus 15 fps SD (mediocre factory or sloppy handloads) at 1000 yards:

Listing 25.14: Monte Carlo: 5 fps SD vs. 15 fps SD at 1000 yards.

```
# Excellent SD: 5 fps
ballistics monte-carlo \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 5 --angle-std 0.05 \
  --bc-std 0.002 \
  --target-distance 1000

# Poor SD: 15 fps
ballistics monte-carlo \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 15 --angle-std 0.05 \
  --bc-std 0.002 \
```

```
  --target-distance 1000
```

The results will show the CEP and vertical dispersion for each scenario. Expect the 15 fps SD to produce roughly 3× the vertical dispersion of the 5 fps SD—directly proportional, because drop is an approximately linear function of velocity for small perturbations.

### 25.5.3 The Drop Sensitivity Factor

A useful concept is the *drop sensitivity*: how many inches (or MILs) of additional drop result from a 1 fps decrease in muzzle velocity. You can compute this by running two trajectories separated by 1 fps:

Listing 25.15: Computing drop sensitivity: drop per fps.

```
# Nominal velocity
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# 1 fps less
ballistics trajectory \
  -v 2749 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

For a 6.5 Creedmoor at 1000 yards, the drop sensitivity is approximately 0.15–0.20 inches per fps. Multiply by your SD to get the velocity-induced vertical dispersion:

$$\sigma_{\text{vert}} = \frac{d(\text{drop})}{dv} \times \text{SD} \tag{25.3}$$

At 0.18 inches/fps with a 5 fps SD, the velocity-induced vertical dispersion is $0.18 \times 5 = 0.9$ inches—well under a 1 MOA target. At 15 fps SD, it becomes $0.18 \times 15 = 2.7$ inches, which is 0.26 MOA—still manageable, but now a significant fraction of the precision budget.

**Tip**

For PRS/NRL competition, target a velocity SD below 8 fps. For F-Class at 1000 yards, below 5 fps is strongly recommended. For hunting at distances under 400 yards, even 15 fps SD is unlikely to cause a miss—shot placement and wind reading matter far more.

Table 25.1: Vertical dispersion from velocity SD at 1000 yards (6.5 Creedmoor, 140 gr, G7 BC 0.310).

| Velocity SD (fps) | Vert. Disp. (in) | Vert. Disp. (MOA) | Rating |
|---|---|---|---|
| 3 | 0.5 | 0.05 | Exceptional |
| 5 | 0.9 | 0.09 | Excellent |
| 8 | 1.4 | 0.14 | Good |
| 12 | 2.2 | 0.21 | Adequate |
| 15 | 2.7 | 0.26 | Marginal |
| 20 | 3.6 | 0.34 | Poor |

### 25.5.4 ES vs. SD: Which Metric to Trust

Handloaders commonly report both *extreme spread* (ES) and *standard deviation* (SD) from their chronograph data. For Monte Carlo modeling, SD is the correct input because it describes the underlying distribution. ES is a function of both the distribution and the sample size—a 5-round string and a 20-round string with the same underlying SD will produce very different ES values.

The approximate relationship for normally distributed velocities is:

$$\text{ES}_{\text{expected}} \approx d_n \times \text{SD} \tag{25.4}$$

where $d_n$ is a factor that depends on sample size:

| $n$ (rounds) | $d_n$ | Example: SD = 8 fps | Expected ES (fps) |
|---|---|---|---|
| 5 | 2.33 | 8 | 19 |
| 10 | 3.08 | 8 | 25 |
| 20 | 3.74 | 8 | 30 |
| 50 | 4.50 | 8 | 36 |

A common mistake is to celebrate a 5-round ES of 12 fps as "amazing" when the underlying SD might still be 8 fps—it's simply that 5 rounds is too few to reveal the full spread. Always report SD from at least a 10-round string, and use SD (not ES) as the `--velocity-std` input to Monte Carlo simulations.

### 25.5.5 How Many Rounds to Chronograph

The precision of your SD estimate depends on sample size. The standard error of the SD estimate for a normal distribution is approximately:

$$\mathrm{SE}(\hat{\sigma}) \approx \frac{\hat{\sigma}}{\sqrt{2(n-1)}} \tag{25.5}$$

For a 10-round string with a true SD of 8 fps, the standard error of the SD estimate is $8/\sqrt{18} \approx 1.9$ fps—meaning your measured SD could easily be anywhere from 6 to 10 fps. A 20-round string reduces this uncertainty to $8/\sqrt{38} \approx 1.3$ fps.

**Tip**

For serious load development, chronograph at least 20 rounds of your final load. For a quick screen during ladder testing, 5-round strings at each charge weight are adequate to identify trends, but do not trust the absolute SD values from such small samples.

## 25.6 Working Up Loads: A Safe Workflow

**SAFETY: Safety Warning**

This section describes how to use `ballistics-engine` to *evaluate* and *predict the downrange performance* of handloaded ammunition. It is **not** a substitute for a reloading manual. Always:

- Start below the published starting load.
- Work up in small increments (0.3–0.5 grains for rifle cartridges).
- Watch for pressure signs (cratered primers, ejector marks, stiff bolt lift) at every step.
- Never exceed the published maximum load.
- Never use data from a different bullet, case, or primer without consulting a manual for that specific combination.

A ballistics solver predicts *external* ballistics (what happens after the bullet leaves the barrel). It knows nothing about chamber pressure. Pressure data must come from published load data or pressure-testing equipment.

### 25.6.1 The Handloader's Workflow with a Solver

A responsible load development workflow that integrates `ballistics-engine` looks like this:

1. **Select bullet and powder from a manual.** Choose a combination for which published data exists. Note the starting and maximum charges.
2. **Predict downrange performance before loading.** Use the solver to compare candidate bullets:

```
# Candidate A: 140gr Berger, G7 BC 0.310, expected MV 2750
```

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# Candidate B: 147gr ELD-M, G7 BC 0.351, expected MV 2700
ballistics trajectory \
  -v 2700 -b 0.351 -m 147 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

This comparison happens *before* you buy bullets or burn powder.

3. **Load a ladder test.** Using the manual's data, load 3–5 rounds at each of 5–8 charge weights spanning from 1 grain below maximum to the starting load.
4. **Fire and chronograph.** Record velocity for every round. Note group sizes and points of impact.
5. **Analyze with the solver.** Use the actual chronographed velocities to predict long-range performance. Estimate the true BC from observed drops if long-range data is available.
6. **Select the charge weight.** Choose the charge that gives the best combination of low SD, acceptable group size, and adequate velocity.
7. **True your dope.** Use the `true-velocity` subcommand to refine your model with real-world data.
8. **Validate.** Fire a verification group at long range and compare with the solver's prediction.

### 25.6.2 Predicting the Effect of Charge Changes

Once you know the velocity at one charge weight, you can estimate the velocity at nearby charges. A common empirical relationship for bottleneck rifle cartridges is:

$$\Delta v \approx 20\text{–}40 \text{ fps per grain of powder} \tag{25.6}$$

This varies with cartridge capacity and powder burn rate. For example, the 6.5 Creedmoor with H4350 typically shows about 30 fps per grain.

Use the solver to project the effect of these velocity differences:

Listing 25.16: Projecting charge-weight changes to long range.

```
# Base charge (43.0 gr H4350): 2740 fps
```

```
ballistics trajectory \
  -v 2740 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# +0.5 gr (43.5 gr): ~2755 fps
ballistics trajectory \
  -v 2755 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full

# +1.0 gr (44.0 gr): ~2770 fps
ballistics trajectory \
  -v 2770 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full
```

**SAFETY: Safety Warning**

Never use a ballistics solver to justify exceeding published maximum loads. The solver tells you what happens *after* the muzzle; it knows nothing about what is happening inside the chamber. A higher charge weight may produce a flatter trajectory, but it may also produce dangerous pressures. Always stay within the load data published by the powder manufacturer.

### 25.6.3 Powder Temperature Sensitivity

Some powders are more sensitive to temperature than others. A load developed at 70°F may produce very different velocities when fired at 20°F or 100°F.

The `ballistics-engine` models this with the `--use-powder-sensitivity`, `--powder-temp-sensitivity`, and `--powder-temp` flags. The adjusted velocity is computed in `adjusted_muzzle_velocity()` in `src/angle_calculations.rs`:

$$v_{\text{adj}} = v_{\text{base}} \times \left(1 + s \cdot \frac{T_{\text{ambient}} - T_{\text{powder}}}{15}\right) \tag{25.7}$$

where $s$ is the powder temperature sensitivity factor, $T_{\text{ambient}}$ is the current temperature, and $T_{\text{powder}}$ is the temperature at which the base velocity was measured.

Listing 25.17: Modeling powder temperature sensitivity.

```
# Load developed at 70 F, now shooting at 20 F
```

```
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full \
  --use-powder-sensitivity \
  --powder-temp-sensitivity 1.0 \
  --powder-temp 70 \
  --temperature 20

# Same load at 100 F
ballistics trajectory \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 --full \
  --use-powder-sensitivity \
  --powder-temp-sensitivity 1.0 \
  --powder-temp 70 \
  --temperature 100
```

### Resolving the Velocity Without a Trajectory

Often the question is not what the trajectory does but simply *what velocity will this load make on a cold morning?* Since v0.27.0 the `powder` command answers that directly, with no trajectory solve:

Listing 25.18: Resolving a cold-day velocity.

```
ballistics powder \
  --velocity 2750 --temperature 20 \
  --powder-temp-sensitivity 1.0 --powder-temp 70 \
  --mass 140
```

Listing 25.19: Powder command output.

```
Powder Temperature Velocity
===========================
  Model:              linear, 1.00 fps/°F
  Reference temp:     70.0 °F
  Nominal velocity:   2750.0 fps

  Shot temp:          20.0 °F
  Resolved velocity:  2700.0 fps  (-50.0)
  Muzzle energy:      2266 ft·lb
```

Optional `--mass` adds the muzzle energy line. `--sweep START:END:STEP` turns the single answer into a ladder across a temperature range, which is the form worth taping inside a rifle case:

Listing 25.20: A velocity ladder from a measured curve.

```
ballistics powder \
  --velocity 2750 --mass 140 \
  --powder-temp-curve "20:2660,50:2710,70:2750,100:2795" \
  --sweep 20:100:20
```

Listing 25.21: Sweep output.

```
Powder Temperature Velocity
===========================
  Model:              measured curve, 4 points (20.0–100.0 °F)
  Nominal velocity:   2750.0 fps

   Temp (°F)  Velocity (fps)   Shift (fps)   Energy (ft·lb)
        20.0          2660.0         -90.0             2199
        40.0          2693.3         -56.7             2255
        60.0          2730.0         -20.0             2316
        80.0          2765.0          15.0             2376
       100.0          2795.0          45.0             2428
```

**`powder` Always Applies the Linear Model**

There is one asymmetry worth knowing. `powder` always applies the linear sensitivity model, whereas **`trajectory`** and `lead` still gate theirs behind `--use-powder-sensitivity`. A measured `--powder-temp-curve`, by contrast, applies unconditionally on all three. The physics is one shared implementation (`resolve_powder_adjusted_velocity`), so what `powder` prints is exactly what the solvers fly—but a linear correction you see here will not appear in a **`trajectory`** run unless you ask for it.

A sensitivity of 1.0 fps per degree is typical for many "temperature-insensitive" powders (like Hodgdon H4350 or Alliant Reloder 16). Less stable powders can show 2–3 fps per degree, which translates to 30–50 fps of velocity change across a 50-degree temperature swing—enough to affect long-range dope by a full MIL at 1000 yards.

**Tip**

If you hunt in extreme temperature ranges (e.g., cold-weather deer hunting and hot-weather prairie dog shooting with the same rifle), chronograph your load at both temperature extremes. Use `ballistics-engine`'s powder sensitivity model to build accurate dope for each condition, or select a temperature-insensitive powder to minimize the issue.

## Exercises

1. **Barrel length analysis.** For a .300 Win Mag (180 gr AccuBond, G1 BC `0.507`), estimate the muzzle velocity for 24-inch, 22-inch, and 20-inch barrels assuming the 26-inch reference velocity is 2960 fps and velocity loss is 30 fps/inch. Compute MPBR (10-inch vital zone) and the 1500 ft-lb energy range for each barrel length.

```
# 24-inch: ~2900 fps
ballistics mpbr \
  -v 2900 -b 0.507 -m 180 -d 0.308 \
  --vital-zone 10

# 20-inch: ~2780 fps
ballistics mpbr \
  -v 2780 -b 0.507 -m 180 -d 0.308 \
  --vital-zone 10
```

2. **Bullet selection exercise.** For a 6.5 Creedmoor intended for PRS competition to 1200 yards, compare the 130-grain Berger AR Hybrid (G7 BC `0.290`, 2850 fps) against the 140-grain Berger Hybrid (G7 BC `0.310`, 2750 fps) and the 147-grain ELD-M (G7 BC `0.351`, 2700 fps). Which bullet has the least wind drift at 1000 yards? Which goes transonic earliest?

```
ballistics trajectory \
  -v 2850 -b 0.290 -m 130 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 10 --wind-direction 90 --full

ballistics trajectory \
  -v 2700 -b 0.351 -m 147 -d 0.264 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1200 \
  --wind-speed 10 --wind-direction 90 --full
```

3. **Velocity SD Monte Carlo.** For your chosen load (or the 6.5 Creedmoor reference load), run Monte Carlo simulations at 600 yards with velocity SDs of 3, 8, 15, and 25 fps. At what SD does the velocity-induced vertical dispersion exceed 1 MOA?

```
ballistics monte-carlo \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 3 --angle-std 0.01 \
  --bc-std 0.001 --wind-std 0 \
  --target-distance 600

ballistics monte-carlo \
  -v 2750 -b 0.310 -m 140 -d 0.264 \
  -n 1000 \
  --velocity-std 25 --angle-std 0.01 \
  --bc-std 0.001 --wind-std 0 \
  --target-distance 600
```

4. **Temperature sensitivity.** Model a .308 Win load (168 gr SMK, G1 BC 0.475, 2700 fps at 70°F) with a powder sensitivity of 1.5 fps/degree. Compute the drop at 600 yards at temperatures of 20°F, 70°F, and 100°F. How many MILs does the cold-weather drop differ from the hot-weather drop?

```
ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --use-powder-sensitivity \
  --powder-temp-sensitivity 1.5 \
  --powder-temp 70 \
  --temperature 20 --full

ballistics trajectory \
  -v 2700 -b 0.475 -m 168 -d 0.308 \
  --auto-zero 100 --max-range 600 \
  --use-powder-sensitivity \
  --powder-temp-sensitivity 1.5 \
  --powder-temp 70 \
  --temperature 100 --full
```

5. **BC estimation from field data.** You fire a 6.5 Creedmoor (140 gr, 2750 fps) at a steel plate and observe 0.48 m of drop at 300 m and 2.45 m of drop at 600 m. Use the `estimate-bc` subcommand to determine the effective BC. How does it compare to the published G7 value of `0.310`?

```
ballistics --units metric estimate-bc \
  -v 838.2 -m 9.07 -d 6.71 \
  --distance1 300 --drop1 0.48 \
  --distance2 600 --drop2 2.45
```

## What's Next

This chapter has shown how `ballistics-engine` fits into the handloader's toolkit—predicting, analyzing, and validating load performance at every stage from bullet selection through final truing. With the real-world applications of hunting (Chapter 23), competition (Chapter 24), and load development covered, we now turn to Part IX: *Under the Hood*, where Chapter 26 examines the engine's modular Rust architecture, Chapter 27 covers the FFI, WebAssembly, and Python bindings, and Chapter 28 explores performance profiling and optimization.

# Part IX

# Under the Hood

# Chapter 26

# Architecture

> *"The structure of a program reflects the structure of the problem it solves—and the convictions of the people who solved it."*

Up to this point we have used `ballistics-engine` as a black box: data goes in, trajectories come out. This chapter opens that box. We will walk through the Rust source tree module by module, trace a bullet's path through the solver pipeline from the first keystroke of a CLI command to the final row of a trajectory table, and examine the key data structures that carry a projectile's state from muzzle to target. Along the way we will explain *why* the architecture looks the way it does—design decisions born from the competing demands of performance, accuracy, and extensibility.

Whether you want to contribute a pull request, wrap the engine in a new language binding, or simply understand what happens when you press Enter, this chapter is your map.

## 26.1 Module Organization: The `src/` Directory

The `ballistics-engine` crate follows a flat module layout: every public module lives directly under `src/`, and the module graph is declared in `src/lib.rs`. There are no deeply nested subdirectories. This keeps `use` paths short and makes it trivial to locate any module by name.

**Crate vs. Binary**

`ballistics-engine` ships as both a *library crate* (`src/lib.rs`) published to `crates.io` and a *binary crate* (`src/main.rs`) that provides the `ballistics` CLI. The binary depends on the library; they are compiled from the same Cargo workspace. The `[lib]` section of `Cargo.toml` specifies three crate types: `rlib` (Rust library), `staticlib` (C static archive), and `cdylib` (C dynamic library).

At the time of writing (version 0.14.1), the `src/` directory contains over 35 Rust source files. They divide into seven functional groups.

### 26.1.1 Core Solver

These modules implement the heart of the trajectory engine:

| Module | Purpose |
|---|---|
| `cli_api.rs` | High-level API: `TrajectorySolver`, `BallisticInputs`, `TrajectoryResult`, zeroing, and Monte Carlo entry points. |
| `trajectory_solver.rs` | Initial-conditions preparation, post-processing, trajectory apex detection, and unit conversion helpers. |
| `trajectory_integration.rs` | RK4 and adaptive RK45 (Dormand–Prince) integrators with target detection and ground-impact termination. |
| `derivatives.rs` | Computes the six-element state derivative vector: drag, gravity, Coriolis, Magnus, and spin drift accelerations. |
| `fast_trajectory.rs` | Lightweight fixed-step RK4 solver tuned for Monte Carlo batch runs and long-range speed. |
| `constants.rs` | Physical constants: `G_ACCEL_MPS2`, unit-conversion factors, numerical tolerances, and BC fallback tables organized by weight category and caliber. |

### 26.1.2 Drag and Ballistic Coefficient

| Module | Purpose |
|---|---|
| `drag_model.rs` | The `DragModel` enum: `G1`, `G2`, `G5`, `G6`, `G7`, `G8`, `GI`, `GS`. |

| | |
|---|---|
| drag.rs | Drag-coefficient lookup via cubic interpolation of embedded drag tables, with the get_drag_coefficient_full function that applies transonic and Reynolds corrections. |
| drag_tables.rs | The actual Mach-vs-$C_D$ data arrays for every standard drag model. |
| transonic_drag.rs | Wave-drag corrections in the transonic regime ($0.8 < M < 1.2$), shape-dependent. |
| form_factor.rs | Form-factor adjustments that scale $C_D$ for specific bullet shapes and models. |
| bc_table.rsbc_table_5d.rs | Offline and 5-D BC correction tables (caliber-specific, ML-derived). |
| bc_estimation.rs | Velocity-based BC segment estimation from bullet characteristics (BCSegmentEstimator). |
| cluster_bc.rs | Cluster-based BC degradation model. |

### 26.1.3 Atmosphere and Wind

| Module | Purpose |
|---|---|
| atmosphere.rs | ICAO Standard Atmosphere with all seven layers up to 84 km; CIPM-2007 air-density calculation with humidity corrections. Exposes calculate_atmosphere, get_local_atmosphere, and get_direct_atmosphere. |
| wind.rs | WindSock and WindSegment types for range-dependent wind vectors. |
| wind_shear.rs | Altitude-dependent wind models: power-law, logarithmic, and custom layer profiles via get_wind_at_position. |

### 26.1.4 Advanced Physics

Each advanced effect lives in its own module, gated by a boolean flag in BallisticInputs:

| Module | Purpose |
|---|---|
| spin_drift.rsspin_drift_advanced.rs | Gyroscopic drift: basic and enhanced models including crosswind coupling via apply_enhanced_spin_drift. |
| spin_decay.rs | Spin-rate decay over time of flight. |
| aerodynamic_jump.rs | Aerodynamic jump at the muzzle. |

| | |
|---|---|
| `stability.rs` `stability_advanced.rs` | Gyroscopic and dynamic stability factors. |
| `precession_nutation.rs` | Angular motion: fast and slow precession, nutation damping. |
| `pitch_damping.rs` | Pitch-damping coefficients for transonic stability warnings. |
| `reynolds.rs` | Reynolds-number corrections for low-velocity drag via `apply_reynolds_correction`. |
| `angle_calculations.rs` | Shooting-angle transformations (uphill / downhill). |

**Feature Flags Philosophy**

Each physics module has its own enable flag (e.g., `enable_pitch_damping`, `use_enhanced_spin_drift`, `enable_precession_nutation`). There is no monolithic "advanced mode." This lets you enable exactly the effects you need—and pay only for the computational cost of what you turn on. The design principle is documented in the project's `CLAUDE.md`: *"Use individual feature flags, NOT generic 'enable_advanced_effects'."*

### 26.1.5 Platform Bindings

| Module | Purpose |
|---|---|
| `ffi.rs` | C-ABI foreign function interface with `#[repr(C)]` structures—used by iOS and Android apps. |
| `wasm.rs` | WebAssembly bindings via `wasm-bindgen`, including a full CLI parser (`WasmBallistics`) and an OOP `Calculator` API with fluent setters. |

These are covered in depth in Chapter 27.

### 26.1.6 Monte Carlo and Statistical Analysis

| Module | Purpose |
|---|---|
| `monte_carlo.rs` | CEP calculation via `calculate_cep`, 95% confidence-ellipse estimation via `calculate_confidence_ellipse`, and single-trajectory evaluation for batch runs (`solve_trajectory_for_monte_carlo`). |

### 26.1.7 Online and Auxiliary

| Module | Purpose |
|---|---|
| `api_client.rs` | HTTP client for the proprietary online API (feature-gated behind `online`). |
| `bc_table_download.rs` | Auto-download of 5-D BC tables (feature-gated behind `online`). |
| `trajectory_sampling.rs` | Regular-interval trajectory data collection for dope cards and tabular output. |
| `pdf_dope_card.rs` | PDF dope-card generation (feature-gated behind `pdf`). |

### 26.1.8 Inspecting the Module Tree

You can verify the module structure at any time using the compiler itself:

Listing 26.1: Listing public modules with `cargo doc`.

```
# Generate documentation and open it in a browser
cargo doc --open --no-deps

# Or list source files directly
ls src/*.rs | wc -l
# 37
```

The authoritative module list lives in `src/lib.rs`, where each `pub mod` declaration makes a module visible to downstream crates and the FFI/WASM layers. A simplified view of the declaration order:

Listing 26.2: Module declarations in `src/lib.rs` (abbreviated).

```
// Re-export the main types
pub use cli_api::{
    BallisticInputs, TrajectorySolver, TrajectoryResult,
    TrajectoryPoint, WindConditions, AtmosphericConditions,
    // ... additional re-exports
};
pub use drag_model::DragModel;

// Core modules
pub mod cli_api;
pub mod ffi;
#[cfg(target_arch = "wasm32")]
pub mod wasm;
```

```
// Physics modules
pub mod atmosphere;
pub mod constants;
pub mod drag;
pub mod derivatives;
pub mod trajectory_solver;
pub mod trajectory_integration;
pub mod fast_trajectory;

// Advanced physics
pub mod spin_drift;
pub mod spin_drift_advanced;
pub mod precession_nutation;
pub mod pitch_damping;
pub mod reynolds;
pub mod stability;
pub mod monte_carlo;
// ... and more
```

## 26.2 The Solver Pipeline

Every trajectory computation—whether triggered by the CLI, the FFI, or the WASM API—follows the same five-stage pipeline:

1. **Input parsing and validation.**
2. **Atmosphere calculation.**
3. **Zeroing (optional).**
4. **Numerical integration.**
5. **Output formatting.**

Let us trace a concrete command through each stage.

Listing 26.3: A .308 Win trajectory that exercises the full pipeline.

```
ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --temperature 59 --pressure 29.92 --altitude 0 \
  --wind-speed 10 --wind-direction 90
```

This command solves a .308 Winchester trajectory: 168-grain bullet, G7 BC `0.462`, at 2700 fps, zeroed at 100 yards, with a 10 mph crosswind from the right.

### 26.2.1 Stage 1: Input Parsing

The binary crate (`src/main.rs`) uses `clap` 4 to parse command-line arguments into a typed Rust structure. These values are then assembled into a `BallisticInputs` struct (defined in `src/cli_api.rs`). This is the engine's *universal input structure*: every entry point—CLI, FFI, WASM, and Python—ultimately builds a `BallisticInputs` and hands it to the solver.

The `main.rs` file is the largest in the codebase—it handles not only argument parsing but also Terms of Service acceptance for the `online` feature, unit conversions between imperial and metric, profile loading, and output formatting. It uses `clap`'s derive API with `Parser` and `Subcommand` traits:

Listing 26.4: Top-level CLI structure in `src/main.rs`.

```
use clap::{Parser, Subcommand, ValueEnum};

// The CLI parses subcommands: trajectory, zero,
// monte-carlo, estimate-bc, and others
```

Each subcommand maps to a handler function that builds a `BallisticInputs` struct and passes it to the library-level API. The struct contains roughly 50 fields organized into five groups:

- **Core ballistics**: `bc_value`, `bc_type`, `bullet_mass`, `muzzle_velocity`, `bullet_diameter`, `bullet_length`.
- **Targeting**: `target_distance`, `muzzle_angle`, `sight_height`, `shooting_angle`.
- **Environment**: `altitude`, `temperature`, `pressure`, `humidity`.
- **Wind**: `wind_speed`, `wind_angle`.
- **Advanced effects**: boolean flags like `enable_advanced_effects`, `use_enhanced_spin_drift`, `enable_pitch_dam`, and `enable_precession_nutation`.

**Unit Conventions**

Internally, `BallisticInputs` stores quantities in a mix of metric and imperial units that reflects the ballistics community's conventions. Muzzle velocity is in feet per second, bullet mass in grains, diameter in inches, but temperature in Celsius and pressure in hPa. The `Default` implementation sets sensible values for a generic projectile. Unit conversions happen at the boundary—the CLI and WASM layers convert user-facing imperial or metric inputs into the internal representation before constructing the struct.

### 26.2.2 Stage 2: Atmosphere Calculation

Once inputs are validated, the engine computes local atmospheric conditions. The `calculate_atmosphere` function in `src/atmosphere.rs` implements the full ICAO Standard Atmosphere model with all seven layers (from the troposphere at sea level to the upper mesosphere at 84 km). It returns two values that the drag calculation needs: air density ($\rho$) in $\mathrm{kg/m^3}$ and the local speed of sound in $\mathrm{m/s}$.

The atmosphere module supports three modes:

1. **Standard atmosphere**: given only altitude, compute temperature and pressure from the ICAO model.
2. **Station conditions**: given altitude *plus* measured temperature and pressure, compute density with humidity corrections using CIPM-2007 constants.
3. **Direct atmosphere**: if the caller provides air density and speed of sound directly (as the FFI layer sometimes does), bypass the model entirely.

The choice is made automatically inside the derivative function by inspecting a four-element `atmos_params` tuple. The detection heuristic uses physically meaningful thresholds:

Listing 26.5: Atmosphere selection logic in `src/derivatives.rs`.

```
// Atmosphere detection thresholds
const MAX_REALISTIC_DENSITY: f64 = 2.0;      // kg/m^3
const MIN_REALISTIC_SPEED_OF_SOUND: f64 = 200.0; // m/s

let (air_density, speed_of_sound) =
    if atmos_params.0 < MAX_REALISTIC_DENSITY
        && atmos_params.1 > MIN_REALISTIC_SPEED_OF_SOUND
        && atmos_params.2 == 0.0
        && atmos_params.3 == 0.0
    {
        // Direct atmosphere values provided
        get_direct_atmosphere(atmos_params.0, atmos_params.1)
    } else {
        // Calculate from altitude, temperature, pressure
        get_local_atmosphere(
            altitude_at_pos,
            atmos_params.0, // base_alt
            atmos_params.1, // base_temp_c
            atmos_params.2, // base_press_hpa
            atmos_params.3, // base_ratio
        )
    };
```

This dual-mode design allows the FFI layer to pre-compute atmospheric conditions on the calling side (useful for mobile apps that already have environmental sensor data) while still supporting the full ICAO model for CLI and WASM users.

### 26.2.3 Stage 3: Zeroing

When the user passes `--auto-zero`, the engine must find the muzzle elevation angle that places the bullet at the line of sight at the specified zero distance. This is a root-finding problem: find $\theta$ such that $y(\theta, d_{\text{zero}}) = h_{\text{sight}}$.

The function `calculate_zero_angle_with_conditions` in `src/cli_api.rs` solves this iteratively. Each iteration runs a full trajectory to evaluate $y$ at the zero distance, then adjusts $\theta$ until convergence. The zeroing algorithm is discussed in detail in Chapter 20.

Listing 26.6: Zeroing at 100 yards with a .308 Win.

```
ballistics zero \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 100
# Zero Angle: 0.0011 rad (3.74 MOA up)
```

Because each zero iteration runs a complete trajectory, zeroing is the most expensive single operation in the pipeline. The engine typically requires 8–15 iterations to converge, meaning the zero step alone costs an order of magnitude more than a single trajectory evaluation.

### 26.2.4 Stage 4: Numerical Integration

This is where physics happens. The engine solves a system of six coupled ordinary differential equations (three for position, three for velocity) by stepping forward in time from the muzzle until the bullet reaches the target, impacts the ground, or exceeds a time limit.

Three integration methods are available, selectable via `BallisticInputs` flags:

1. **Euler** (`use_rk4 = false`): a first-order explicit method. Fast but inaccurate. Useful only for quick sanity checks.
2. **Fixed-step RK4** (`use_rk4 = true`, `use_adaptive_rk45 = false`): the classical fourth-order Runge–Kutta method with a fixed time step (typically 1 ms for long-range trajectories). Good accuracy, predictable performance.
3. **Adaptive RK45** (`use_rk4 = true`, `use_adaptive_rk45 = true`): the Dormand–Prince method (the same algorithm used by SciPy's `solve_ivp`). This is the default and recommended method. It adapts the step size to maintain a user-specified tolerance (default $10^{-6}$), taking large steps in smooth regions and small steps through the transonic zone.

The adaptive RK45 implementation in `src/trajectory_integration.rs` uses the standard Dormand–Prince Butcher tableau. It computes seven stages per step ($k_1$ through $k_7$), produces a fifth-order solution for advancement, and a fourth-order solution for error estimation. The error estimate comes from the difference between these two solutions:

$$\|\mathbf{e}\| = \frac{\|\mathbf{y}_5 - \mathbf{y}_4\|}{1 + \|\mathbf{s}\|} \tag{26.1}$$

where $\mathbf{y}_5$ is the fifth-order solution, $\mathbf{y}_4$ is the fourth-order solution, and $\mathbf{s}$ is the current state (the denominator provides relative error scaling). The step-size controller uses:

$$\Delta t_{\text{new}} = 0.9 \cdot \Delta t \cdot \begin{cases} \left(\frac{\varepsilon}{\|e\|}\right)^{1/5} & \text{if } \|e\| < \varepsilon \text{ (accept, grow)} \\ \left(\frac{\varepsilon}{\|e\|}\right)^{1/4} & \text{if } \|e\| \geq \varepsilon \text{ (reject, shrink)} \end{cases} \tag{26.2}$$

The growth factor is clamped to a maximum of $2\times$; the shrink factor is clamped to a minimum of $0.1\times$. The safety factor of 0.9 prevents oscillation around the tolerance boundary.

Listing 26.7: Step-size adaptation in `src/trajectory_integration.rs`.

```
// Adaptive step size
let safety = 0.9;
let dt_new = if error < tol {
    dt * safety * (tol / error).powf(0.2).min(2.0)
} else {
    dt * safety * (tol / error).powf(0.25).max(0.1)
};
```

**Wind Shear and Step Size**

When altitude-dependent wind shear is enabled, the integrator automatically reduces the effective maximum step size to 10–20 ms (depending on range) to maintain numerical stability. For ranges above 800 m, `effective_max_step` drops to 10 ms; for shorter ranges, 20 ms. This is handled inside `integrate_trajectory` without user intervention.

### Target Detection

The integrator checks at every step whether the bullet's downrange coordinate ($z$) has crossed the target distance. When it detects a crossing, it performs linear interpolation to estimate the crossing time, then takes a refined sub-step to land precisely at the target:

Listing 26.8: Target-crossing detection in the RK45 loop.

```
if state[2] < params.target_distance_m
    && new_state[2] >= params.target_distance_m
{
    let alpha = (params.target_distance_m - state[2])
              / (new_state[2] - state[2]);
    let dt_to_target = dt * alpha;
    let (final_state, _, _) =
        rk45_step(&state, t, dt_to_target, &params, tolerance);
    trajectory.push((t + dt_to_target, corrected_state));
    break;
}
```

This two-phase approach (detect then refine) ensures that the final trajectory point is within fractions of a millimeter of the target distance, regardless of the adaptive step size. A ground-impact check (`state[1] < -1000.0`) provides a safety net for trajectories that never reach the target.

### Iteration Safety

The RK45 integrator includes a hard iteration limit of 100,000 steps to prevent infinite loops in edge cases (e.g., extremely small step sizes forced by stiff dynamics):

Listing 26.9: Safety iteration limit.

```
let max_iterations = 100000;
let mut iteration_count = 0;

while t < t_end && iteration_count < max_iterations {
    iteration_count += 1;
    // ... integration step
}
```

For a typical 1,000-yard trajectory at 1 ms average step size, the loop runs approximately 1,500–3,000 iterations—well below the limit.

## 26.2.5 Stage 5: Output Formatting

The raw trajectory—a sequence of $(t, \mathbf{x}, \mathbf{v})$ points—passes through post-processing, which computes derived quantities: drop (inches and meters), wind drift, maximum ordinate, final velocity, and kinetic energy. The result is packaged into a `TrajectoryResult` struct that the CLI, WASM, or FFI layer formats into tables, JSON, or CSV.

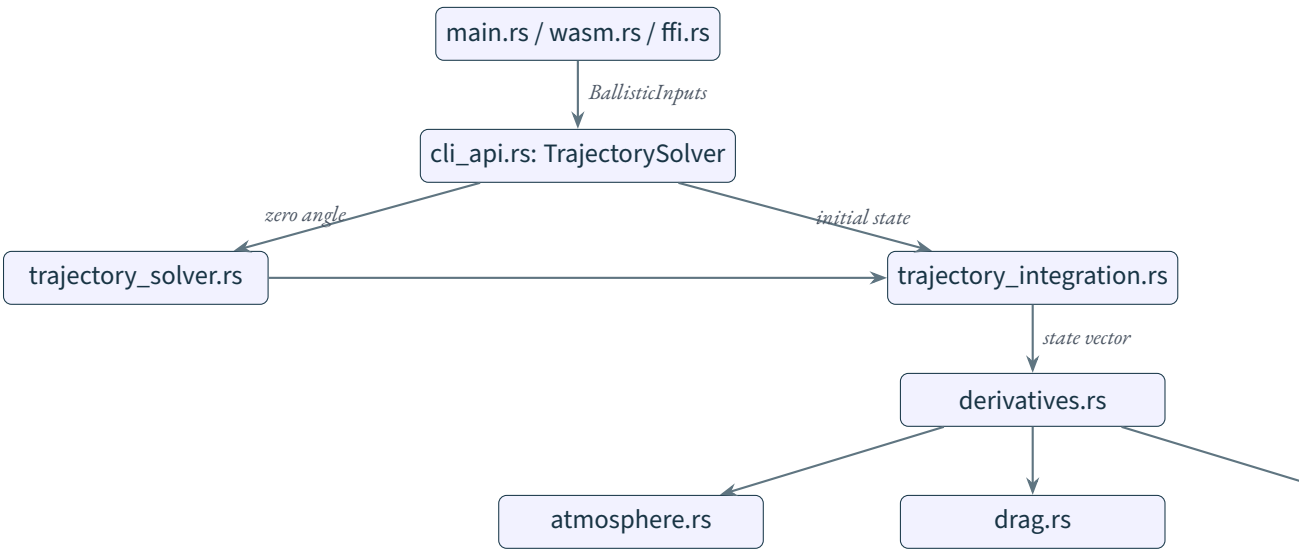
Listing 26.10: Output format selection.

```
# Human-readable table (default)
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 600

# JSON for scripting
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 600 -o json

# CSV for spreadsheet import
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 600 -o csv
```

The WASM layer (`src/wasm.rs`) mirrors all three output formats identically, so commands tested in the terminal work unchanged when run through `WasmBallistics.runCommand()` in a browser.

## 26.3 Data Flow Through the Engine

To understand how data moves through the engine, it helps to visualize the call graph for a single trajectory computation. The flow begins at the entry-point layer and descends through four levels:




### 26.3.1 The State Vector

At every time step, the projectile's state is a six-element array:

$$\mathbf{s} = \begin{bmatrix} x & y & z & v_x & v_y & v_z \end{bmatrix}^T \tag{26.3}$$

where $(x, y, z)$ is position in meters and $(v_x, v_y, v_z)$ is velocity in $\mathrm{m/s}$. The coordinate system is right-handed:

- $x$: lateral (crosswind direction, positive to the right)
- $y$: vertical (up is positive)
- $z$: downrange (direction of fire)

This convention runs throughout the engine. The initial state is constructed in `prepare_initial_conditions` (`src/trajectory_solver.rs`):

Listing 26.11: Initial state construction.

```
// Muzzle velocity decomposed along launch angle
let initial_vel = Vector3::new(
    mv_mps * muzzle_angle_rad.cos(), // downrange component
    mv_mps * muzzle_angle_rad.sin(), // vertical component
    0.0,                              // no lateral component
);

let initial_state = [0.0, 0.0, 0.0,
                     initial_vel.x, initial_vel.y, initial_vel.z];
```

The derivative function in `src/derivatives.rs` returns a matching six-element array:

$$\dot{\mathbf{s}} = \begin{bmatrix} v_x & v_y & v_z & a_x & a_y & a_z \end{bmatrix}^T \tag{26.4}$$

where the accelerations are the sum of gravity, drag, Coriolis, Magnus, and spin-drift forces.

### 26.3.2 Data Ownership and Lifetimes

The `BallisticInputs` struct is passed by reference (`&BallisticInputs`) throughout the pipeline—it is never consumed. The trajectory integrator owns the mutable state vector and the growing `Vec<(f64, Vector6<f64>)>` of trajectory points. When integration finishes, ownership of this vector moves to the post-processor, which produces a `TrajectoryResult`. The result is then moved to the output formatter, which converts it into a string (table, JSON, or CSV) and writes it to standard output.

This ownership chain means that **no allocation is shared between stages**—each stage owns its output until it passes it to the next. There are no reference cycles, no interior mutability, and no locks. The borrow checker enforces this at compile time.

The Monte Carlo path is slightly different: the `solve_trajectory_for_monte_carlo` function in `src/monte_carlo.rs` takes a `&BallisticInputs` reference, calls `fast_integrate` from `src/fast_trajectory.rs`, and returns a lightweight `TrajectoryOutput` struct containing only the quantities needed for statistical analysis (drop, drift, velocity, energy, Mach number). This avoids the overhead of constructing the full `TrajectoryResult` for trajectories whose only purpose is Monte Carlo aggregation.

## 26.4 Key Data Structures

Four structures define the engine's data model. Understanding them is essential for anyone reading the source code or building a new binding.

### 26.4.1 `BallisticInputs`

Defined in `src/cli_api.rs`, this is the universal input container. It carries every parameter the solver needs, from muzzle velocity to whether precession–nutation is enabled. With roughly 50 fields, it is the largest structure in the codebase. It implements `Default`, `Clone`, and `Debug`.

Key design decisions:

- **Flat structure**: all parameters live at the top level. There are no nested structs for "environment" or "bullet." This avoids the ceremony of constructing nested builders while keeping the `Default` implementation trivial.
- **Boolean feature flags**: advanced physics are opt-in via `bool` fields (`enable_pitch_damping`, `use_enhanced_spin_drift`, etc.). The solver checks these flags at runtime and skips the corresponding calculations when they are `false`.
- **Optional fields**: `Option<Vec<(f64, f64)>>` for Mach-based BC segments, `Option<String>` for manufacturer and model, `Option<f64>` for latitude. This keeps the default small while allowing complex inputs when needed.

Listing 26.12: Excerpt from `BallisticInputs` (simplified).

```rust
pub struct BallisticInputs {
    // Core ballistics
    pub bc_value: f64,
    pub bc_type: DragModel,
    pub bullet_mass: f64,       // grains
    pub muzzle_velocity: f64,   // fps
    pub bullet_diameter: f64,   // inches
    pub bullet_length: f64,     // calibers

    // Integration method
    pub use_rk4: bool,
```

```
    pub use_adaptive_rk45: bool,

    // Advanced effects
    pub enable_advanced_effects: bool,
    pub use_enhanced_spin_drift: bool,
    pub enable_pitch_damping: bool,
    pub enable_precession_nutation: bool,

    // Custom drag model (CDM)
    pub custom_drag_table: Option<DragTable>,
    // ... ~35 more fields
}
```

### 26.4.2 `InitialConditions`

Defined in `src/trajectory_solver.rs`, this struct captures the fully computed starting state for integration. It is produced by `prepare_initial_conditions`, which converts the user-facing `BallisticInputs` into the numerical quantities the integrator needs:

Listing 26.13: The `InitialConditions` struct.

```
pub struct InitialConditions {
    pub mass_kg: f64,
    pub muzzle_velocity_mps: f64,
    pub target_distance_los_m: f64,
    pub muzzle_angle_rad: f64,
    pub muzzle_energy_j: f64,
    pub muzzle_energy_ftlbs: f64,
    pub target_horizontal_dist_m: f64,
    pub target_vertical_height_m: f64,
    pub initial_state: [f64; 6],
    pub t_span: (f64, f64),
    pub omega_vector: Option<Vector3<f64>>,
    pub stability_coefficient: f64,
    pub atmo_params: (f64, f64, f64, f64),
    pub air_density: f64,
    pub speed_of_sound: f64,
}
```

Note the `omega_vector` field: when Coriolis corrections are enabled, this contains the Earth's angular velocity vector projected into the shooter's local coordinate frame. The projection accounts for both *latitude* and *shot azimuth*, using the standard formula:

$$\boldsymbol{\omega}_{\text{local}} = \Omega_{\oplus} \begin{bmatrix} \cos\lambda \sin\alpha \\ \sin\lambda \\ \cos\lambda \cos\alpha \end{bmatrix} \tag{26.5}$$

where $\Omega_{\oplus} = 7.2921159 \times 10^{-5}$ rad/s is Earth's rotation rate, $\lambda$ is the shooter's latitude, and $\alpha$ is the shot azimuth (0 = north, $\pi/2$ = east).

### 26.4.3 `TrajectoryParams`

The integrator does not receive the full `BallisticInputs`—it receives a leaner `TrajectoryParams` struct (defined in `src/trajectory_integration.rs`) that carries only what the integration loop and derivative function need:

Listing 26.14: `TrajectoryParams` carries the integrator's working data.

```
pub struct TrajectoryParams {
    pub mass_kg: f64,
    pub bc: f64,
    pub drag_model: DragModel,
    pub wind_segments: Vec<WindSegment>,
    pub atmos_params: (f64, f64, f64, f64),
    pub omega_vector: Option<Vector3<f64>>,
    pub enable_spin_drift: bool,
    pub enable_magnus: bool,
    pub enable_coriolis: bool,
    pub target_distance_m: f64,
    pub enable_wind_shear: bool,
    pub wind_shear_model: String,
    pub shooter_altitude_m: f64,
    pub is_twist_right: bool,
    pub custom_drag_table: Option<DragTable>,
    pub bc_segments: Option<Vec<(f64, f64)>>,
    pub use_bc_segments: bool,
}
```

This separation serves two purposes. First, it avoids carrying 50 irrelevant fields through every derivative evaluation—the hot inner loop touches only the fields in `TrajectoryParams`. Second, it allows the integration module to be used independently of the CLI API, which is important for the FFI and WASM layers that construct their own parameter sets.

### 26.4.4 `TrajectoryPoint` and `TrajectoryResult`

Each point along the computed trajectory is stored as:

Listing 26.15: The `TrajectoryPoint` struct from `cli_api.rs`.

```
pub struct TrajectoryPoint {
    pub time: f64,
    pub position: Vector3<f64>,
    pub velocity_magnitude: f64,
    pub kinetic_energy: f64,
}
```

The `position` field uses `nalgebra`'s `Vector3<f64>`, giving access to the full $(x, y, z)$ state. The post-processor in `src/trajectory_solver.rs` produces a `TrajectoryResult` struct that wraps these points with summary statistics:

Listing 26.16: `TrajectoryResult` carries summary statistics.

```
pub struct TrajectoryResult {
    pub muzzle_energy_j: f64,
    pub muzzle_energy_ftlbs: f64,
    pub time_of_flight_s: f64,
    pub drop_m: f64,
    pub drop_in: f64,
    pub wind_drift_m: f64,
    pub wind_drift_in: f64,
    pub max_ord_m: f64,
    pub max_ord_in: f64,
    pub final_vel_mps: f64,
    pub final_vel_fps: f64,
    pub final_energy_j: f64,
    pub final_energy_ftlbs: f64,
    pub barrel_angle_rad: f64,
    // ... additional fields
}
```

The dual-unit fields (e.g., `drop_m` and `drop_in`, `final_vel_mps` and `final_vel_fps`) are computed once during post-processing and stored to avoid redundant conversions in the output layer.

## 26.5 Design Decisions

### 26.5.1 Why Rust?

`ballistics-engine` was originally prototyped in Python (as the `ballistics_rust` Flask application). The move to Rust was motivated by three factors:

1. **Performance.** A single 1,000-yard trajectory completes in approximately 3–6 ms in Rust, compared to approximately 200 ms in the Python implementation. For Monte Carlo simulations that run thousands of trajectories, this 30–60× speedup is decisive.

2. **Memory safety without garbage collection.** Rust's ownership system prevents data races and use-after-free bugs at compile time, without the latency spikes that a garbage collector introduces. This matters for real-time applications like the iOS and Android apps that embed the engine via FFI.

3. **Cross-compilation.** A single Rust codebase compiles to native binaries for Linux (x86_64 and aarch64), Windows, macOS, FreeBSD, OpenBSD, NetBSD, WebAssembly, iOS, and Android. The project's build system produces binaries and Python wheels for all major platforms from a single CI run.

Listing 26.17: Verifying the target platform.

```
# Check your current build target
rustc --print target-triple
# Example: aarch64-apple-darwin

# List all available targets
rustup target list
```

### 26.5.2 Why nalgebra?

The engine uses the `nalgebra` crate (version 0.35) for three-dimensional vector arithmetic. Position, velocity, wind, and acceleration are all `Vector3<f64>` values. The integrator uses `Vector6<f64>` to represent the combined position–velocity state.

The alternative—manually indexing arrays—is error-prone and obscures the physics. Compare:

Listing 26.18: nalgebra makes the physics readable.

```
// With nalgebra:
let velocity_adjusted = vel - wind_vector;
let accel_drag = -a_drag * (velocity_adjusted / speed);
let accel = accel_gravity + accel_drag + accel_magnus;

// Without nalgebra (same logic, less readable):
let vax = vel[3] - wind[0];
let vay = vel[4] - wind[1];
let vaz = vel[5] - wind[2];
let speed = (vax*vax + vay*vay + vaz*vaz).sqrt();
let adx = -a_drag * vax / speed;
```

```
// ... and so on for 18 lines
```

The fast trajectory solver (`src/fast_trajectory.rs`) uses raw `[f64; 6]` arrays for state vectors instead of `Vector6<f64>`, trading readability for the last few percent of performance in Monte Carlo batch runs.

### 26.5.3 Why a Flat Module Layout?

Many Rust projects organize code into nested directories (`src/physics/drag/`, `src/physics/atmosphere/`, etc.). `ballistics-engine` keeps everything at the top level. This was a deliberate choice:

- **Discoverability**: with 37 files in one directory, `ls src/*.rs` gives you the complete module list instantly.
- **Short imports**: `use crate::drag::get_drag_coefficient` is easier to read and type than `use crate::physics::drag::coefficient::get_drag_coefficient`.
- **Feature isolation**: each physics effect is one file. If you want to understand spin drift, you read `spin_drift.rs`. There is nothing else to find.

The tradeoff is that the directory is large, but modern editors with fuzzy file finding make this a non-issue.

### 26.5.4 Three Crate Types

The `Cargo.toml` specifies three crate types:

Listing 26.19: Crate type configuration in `Cargo.toml`.

```
[lib]
name = "ballistics_engine"
crate-type = ["rlib", "staticlib", "cdylib"]
```

- `rlib`: the standard Rust library format. Used by the CLI binary and by any Rust code that depends on the crate.
- `staticlib`: a C-compatible static archive (`libballistics_engine.a`). Used to build the iOS XCFramework.
- `cdylib`: a C-compatible dynamic library (`libballistics_engine.so` / `.dylib` / `.dll`). Used by PyO3 Python bindings and for WebAssembly compilation via `wasm-pack`.

This triple output is what allows a single Cargo build to serve the CLI, FFI, WASM, and Python ecosystems simultaneously. The cost is a slightly longer build time (each crate type is a separate linking step), but the benefit is a single source of truth for all platforms.

### 26.5.5 Feature Gating

Four optional Cargo features control heavyweight dependencies:

Listing 26.20: Feature flags in `Cargo.toml`.

```
[features]
default = ["online", "pdf"]
online = ["dep:ureq"]
pdf = ["dep:printpdf"]
jemalloc = ["dep:tikv-jemallocator"]
mimalloc = ["dep:mimalloc"]
```

- `online`: enables the HTTP client (`ureq`) for the proprietary online API and 5-D BC table downloads. Disabled automatically for WASM builds (the browser sandbox does not support `ureq`).
- `pdf`: enables PDF dope-card generation via `printpdf`. Also disabled for WASM.
- `jemalloc` / `mimalloc`: alternative allocators for performance benchmarking (see Chapter 28).

The WASM module (`src/wasm.rs`) is unconditionally compiled only when targeting `wasm32`:

```
#[cfg(target_arch = "wasm32")]
pub mod wasm;
```

### 26.5.6 Release Profile Optimization

The release profile in `Cargo.toml` is tuned for maximum throughput:

Listing 26.21: Release profile settings.

```
[profile.release]
opt-level = 3
lto = true
codegen-units = 1
```

- `opt-level = 3`: full optimization, including auto-vectorization and aggressive inlining.
- `lto = true`: link-time optimization across all dependencies. This eliminates cross-crate function-call overhead and enables whole-program optimization.
- `codegen-units = 1`: forces the compiler to use a single codegen unit, maximizing optimization opportunities at the cost of longer compile times.

Together, these settings typically yield a 20–40% speedup over the default release profile. The effect is most pronounced in tight loops like the derivative evaluation, where the compiler can inline across crate boundaries thanks to LTO. The performance implications of these settings are explored further in Chapter 28.

## 26.6 The Derivative Function: Where Physics Lives

The single most important function in the codebase is `compute_derivatives` in `src/derivatives.rs`. It is called at least four times per RK4 step (seven times for RK45), making it the hottest function in any profiling run.

Its signature:

Listing 26.22: The derivative function signature.

```
pub fn compute_derivatives(
    pos: Vector3<f64>,       // current position (m)
    vel: Vector3<f64>,       // current velocity (m/s)
    inputs: &BallisticInputs,
    wind_vector: Vector3<f64>,
    atmos_params: (f64, f64, f64, f64),
    bc_used: f64,
    omega_vector: Option<Vector3<f64>>,
    time: f64,
) -> [f64; 6]
```

The function takes the current position and velocity vectors, the full `BallisticInputs` reference (for checking feature flags and accessing bullet parameters), the pre-computed wind vector, atmospheric parameters, the current BC value, an optional Earth-rotation vector, and the current time. It returns a six-element derivative array.

The function proceeds through a well-defined sequence of computations:

1. **Gravity**: a constant $-9.80665$ m/s$^2$ in the $y$-direction, using the constant `G_ACCEL_MPS2` from `src/constants.rs`.
2. **Wind adjustment**: subtract the wind vector from the bullet velocity to get the air-relative velocity $\mathbf{v}_{\text{air}}$.
3. **Atmosphere**: compute local air density and speed of sound at the bullet's current altitude using either direct or modeled atmosphere (as discussed in Section 26.2.2).
4. **Mach number**: $M = \|\mathbf{v}_{\text{air}}\| / c_{\text{local}}$.

5. **Drag coefficient**: call `get_drag_coefficient_full` from `src/drag.rs` with the Mach number and drag model. This function applies transonic correction (via `src/transonic_drag.rs`) and Reynolds correction (via `src/reynolds.rs`) when appropriate.

6. **Form factor**: if `use_form_factor` is enabled, apply bullet-model-specific scaling via `apply_form_factor_to_drag`.

7. **BC interpolation**: if velocity-based or Mach-based BC segments are provided, interpolate the BC for the current velocity.

8. **Yaw correction**: multiply drag by $(1+\theta_{\text{yaw}}^2)$ to account for initial tip-off yaw, with exponential decay over distance.

9. **Drag acceleration**: compute retardation using the classical formula and apply it opposite to the air-relative velocity vector.

10. **Magnus effect**: if `enable_advanced_effects` is true and twist rate is specified, compute the Magnus side force using a Mach-dependent coefficient $C_{L\alpha}(M)$.

11. **Coriolis**: if an $\boldsymbol{\omega}$ vector is provided, add $-2\boldsymbol{\omega} \times \mathbf{v}$.

12. **Enhanced spin drift**: if enabled, compute and apply gyroscopic drift acceleration via `apply_enhanced_spin_drif` from `src/spin_drift.rs`.

This layered approach means that disabling an effect costs nothing: the `if` check short-circuits before any floating-point work.

Listing 26.23: The drag calculation core in `src/derivatives.rs`.

```
// Calculate drag acceleration
let standard_factor = drag_factor * CD_TO_RETARD;
let a_drag_ft_s2 =
    (v_rel_fps.powi(2) * standard_factor * yaw_multiplier
     * density_scale) / bc_val;
let a_drag_m_s2 = a_drag_ft_s2 * FPS_TO_MPS;

// Apply drag opposite to air-relative velocity
accel_drag = -a_drag_m_s2 * (velocity_adjusted / speed_air);
```

The constant `CD_TO_RETARD` is defined in `src/constants.rs` as $0.000683 \times 0.30 = 0.0002049$. This dimensional conversion factor bridges the classical ballistics coefficient equation (which operates in imperial units) with the metric computations in the derivative function.

### 26.6.1 The Magnus Moment Coefficient

The Magnus force calculation uses a Mach-dependent coefficient table embedded as a piecewise function in `src/derivatives.rs`:

Listing 26.24: Magnus coefficient by Mach regime.

```
fn calculate_magnus_moment_coefficient(mach: f64) -> f64 {
    if mach < 0.8 {
        // Subsonic: fully developed boundary layer
        0.030
    } else if mach < 1.2 {
        // Transonic: shock disrupts circulation
        0.030 - 0.0075 * (mach - 0.8) / 0.4
    } else {
        // Supersonic: partial recovery
        0.015 + 0.0044 * ((mach - 1.2) / 1.8).min(1.0)
    }
}
```

The coefficient drops to its minimum in the transonic regime ($0.8 < M < 1.2$) because shock waves disrupt the boundary-layer circulation that produces the Magnus force. It partially recovers at higher supersonic speeds. This behavior is consistent with wind-tunnel data and McCoy's *Modern Exterior Ballistics*.

Listing 26.25: Enabling all physics effects for a .308 Win at 1,000 yards.

```
ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --enable-magnus --enable-coriolis --enable-spin-drift \
  --enable-pitch-damping --enable-precession \
  --twist-rate 10 --latitude 45 \
  --wind-speed 10 --wind-direction 90
```

## 26.7 The Constants Module

The `src/constants.rs` file centralizes all physical constants, unit-conversion factors, and numerical tolerances. This prevents magic numbers from scattering through the codebase and makes it easy to audit the physics.

### 26.7.1 Physical Constants

| Constant | Value | Description |
|---|---|---|
| G_ACCEL_MPS2 | 9.80665 | Standard gravitational acceleration ($m/s^2$) |
| STANDARD_AIR_DENSITY | 1.225 | Sea-level air density ($kg/m^3$) |
| SPEED_OF_SOUND_MPS | 340.29 | Sea-level speed of sound at 15°C ($m/s$) |
| CD_TO_RETARD | 0.0002049 | Drag-to-retardation conversion factor |

### 26.7.2 Unit Conversion Factors

| Constant | Value | Conversion |
|---|---|---|
| FPS_TO_MPS | 0.3048 | ft/s → m/s |
| MPS_TO_FPS | 3.28084 | m/s → ft/s |
| GRAINS_TO_KG | $6.479891 \times 10^{-5}$ | grains → kg |

### 26.7.3 Numerical Tolerances

| Constant | Value | Purpose |
|---|---|---|
| NUMERICAL_TOLERANCE | $10^{-9}$ | General floating-point comparison |
| MIN_VELOCITY_THRESHOLD | $10^{-6}$ | Prevents division by zero in drag calculation |
| MIN_DIVISION_THRESHOLD | $10^{-12}$ | Guards all division operations |
| ROOT_FINDING_TOLERANCE | $10^{-6}$ | Convergence criterion for zeroing |
| MIN_MACH_THRESHOLD | $10^{-3}$ | Prevents singularity at $M = 0$ |

### 26.7.4 BC Fallback Tables

The constants file also contains fallback BC values for when measured data is unavailable. These are organized by weight category (ultra-light through very heavy) and by caliber (small through extra-large), derived from statistical analysis of a 2,000+ projectile database. Each value represents the 25th percentile—a conservative estimate that avoids over-predicting ballistic performance:

Listing 26.26: BC fallback constants by weight category.

```
pub const BC_FALLBACK_ULTRA_LIGHT: f64 = 0.172; // 0-50 gr
pub const BC_FALLBACK_LIGHT: f64 = 0.242;       // 50-100 gr
pub const BC_FALLBACK_MEDIUM: f64 = 0.310;      // 100-150 gr
pub const BC_FALLBACK_HEAVY: f64 = 0.393;       // 150-200 gr
pub const BC_FALLBACK_VERY_HEAVY: f64 = 0.441;  // 200+ gr
```

**SAFETY: Fallback BC Values Are Estimates**

The fallback BC values are conservative approximations for situations where measured data is not available. They should **never** be used for load development or pressure estimation. Always use manufacturer-published or doppler-derived BC data for precision work. Using incorrect BC data can lead to dangerous extrapolation of load performance.

## 26.8 The Dependency Graph

The engine's external dependencies are deliberately minimal. The core physics modules depend on only two external crates:

| Crate | Version | Purpose |
|---|---|---|
| `nalgebra` | 0.35 | 3-D vector arithmetic (`Vector3`, `Vector6`) |
| `serde` | 1.0 | Serialization for JSON/CSV output |
| `serde_json` | 1.0 | JSON serialization |
| `rand` / `rand_distr` | 0.10 / 0.6 | Random number generation for Monte Carlo |
| `clap` | 4.5 | CLI argument parsing (binary crate only) |
| `thiserror` | 2.0 | Error type derivation |
| `ndarray` | 0.17 | N-dimensional arrays (used in sampling) |

The optional dependencies (`ureq`, `printpdf`, `tikv-jemallocator`, `mimalloc`) are feature-gated and never affect the core trajectory computation.

**`rayon` Is Gone (0.26.0)**

Earlier versions of this table listed `rayon` for data-parallel Monte Carlo. It was declared but no code referenced it, and v0.26.0 removed it—downstream builds shed the whole transitive tree. Monte Carlo runs serially; a thousand samples of a 600-yard solve is not a workload that needs a thread pool.

Listing 26.27: Building without optional features.

```
# Minimal build: no HTTP client, no PDF generation
cargo build --release --no-default-features

# Check that the core library compiles cleanly
cargo check --no-default-features
```

## 26.9 Exercises

These exercises use real CLI commands to explore the engine's architecture.

1. **Compare integration methods.** Run the same .308 Winchester trajectory (168 gr, `0.462` G7, 2700 fps) to 1000 yards three times: once with Euler, once with fixed RK4, and once with adaptive RK45. Compare the reported drop and time of flight. How much does the Euler method deviate?

```
# Adaptive RK45 (default)
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000

# Fixed-step RK4
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --use-rk4-fixed

# Euler
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --use-euler
```

2. **Observe the effect of individual physics modules.** Run a 1000-yard trajectory for a 6.5 Creedmoor (140 gr ELD-M, G7 BC `0.315`, 2710 fps, 1:8 twist) with no advanced effects, then add `--enable-spin-drift`, then add `--enable-coriolis` with `--latitude 45`. How does each effect change the wind drift column?

```
# Baseline (no advanced effects)
ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90

# Add spin drift
ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90 \
  --enable-spin-drift --twist-rate 8

# Add Coriolis
ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --wind-speed 10 --wind-direction 90 \
  --enable-spin-drift --twist-rate 8 \
```

```
  --enable-coriolis --latitude 45
```

3. **Explore JSON output.** Run a trajectory with `--output json` and pipe it to `jq` (or your favorite JSON tool). Extract just the velocity at 500 yards.

```
ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 600 \
  -o json | jq '.trajectory[] | select(.range_yards > 499
    and .range_yards < 501)'
```

4. **Generate the documentation.** Run `cargo doc --open --no-deps` and browse the generated API documentation. Find the `compute_derivatives` function. How many parameters does it accept? What is the return type?

```
cargo doc --open --no-deps
```

5. **Build with minimal features.** Compile the engine with `--no-default-features` and verify it still runs. Which subcommands are unavailable?

```
cargo build --release --no-default-features
./target/release/ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000
```

## What's Next

Now that you understand the engine's internal structure—its modules, its pipeline, and its data flow—we are ready to look at how the engine crosses language boundaries. Chapter 27 explores the C FFI that powers iOS and Android integration, the WebAssembly compilation that lets `ballistics-engine` run in a browser, and the Python bindings that connect it to the scientific Python ecosystem.

# Chapter 27

# FFI, WASM & Python Bindings

*"A library that cannot leave its own language is a library that cannot leave the lab."*

The Rust engine that we examined in Chapter 26 is powerful, but most ballistic calculators in the wild are not Rust applications. They are Swift apps on iPhones, JavaScript interfaces in web browsers, and Python scripts in Jupyter notebooks. To serve those ecosystems, `ballistics-engine` exposes three language bridges: a C-ABI Foreign Function Interface (FFI), a WebAssembly (WASM) module, and PyO3-based Python bindings. Each bridge wraps the same core solver—the same physics, the same integrator, the same drag tables—so results are bit-for-bit identical regardless of the calling language.

This chapter walks through each bridge in detail: its structure in the source code, its API surface, how to build it, and practical tips for embedding the engine in your own application.

## 27.1 The C FFI

The C foreign function interface lives in `src/ffi.rs`. It provides `extern "C"` functions and `#[repr(C)]` structures that any language with a C calling convention can consume—Swift, Objective-C, Kotlin (via JNI), C#, and, of course, C and C++ themselves.

### 27.1.1 Why a C ABI?

The C ABI is the lowest common denominator of inter-language communication. Every operating system defines it, every linker understands it, and every mainstream language can call into it. By exposing a C API, the engine becomes embeddable in:

- **iOS applications** via an XCFramework that bundles `libballistics_engine.a` for `aarch64-apple-ios` and `aarch64-apple-ios-sim`.
- **Android applications** via JNI wrapping of `libballistics_engine.so` for `aarch64-linux-android` and `armv7-linux-androideabi`.
- **Desktop GUI frameworks** that expect C or C++ libraries (Qt, GTK, Dear ImGui).
- **Game engines** (Unity via P/Invoke, Unreal via `UFUNCTION(BlueprintCallable)`).

### 27.1.2 FFI Data Structures

The FFI layer defines its own C-compatible mirror of the internal Rust types. Every field uses a C primitive type (`c_double`, `c_int`, or a pointer), and every struct uses `#[repr(C)]` to guarantee a stable memory layout.

#### `FFIBallisticInputs`

The primary input structure maps directly to `BallisticInputs` from the Rust API (see Section 26.4.1), but with C-compatible types:

Listing 27.1: The `FFIBallisticInputs` struct in `src/ffi.rs`.

```
#[repr(C)]
pub struct FFIBallisticInputs {
    pub muzzle_velocity: c_double,   // m/s
    pub muzzle_angle: c_double,      // radians
    pub bc_value: c_double,
    pub bullet_mass: c_double,       // kg
    pub bullet_diameter: c_double,   // meters
    pub bc_type: c_int,              // 0=G1, 1=G7, ...
    pub sight_height: c_double,      // meters
    pub target_distance: c_double,   // meters
    pub temperature: c_double,       // Celsius
    pub twist_rate: c_double,        // inches per turn
    pub is_twist_right: c_int,       // 0=false, 1=true
    pub shooting_angle: c_double,    // radians
    pub altitude: c_double,          // meters
    pub latitude: c_double,          // degrees (NAN if N/A)
    pub azimuth_angle: c_double,     // radians
    pub use_rk4: c_int,             // 0=Euler, 1=RK4
    pub use_adaptive_rk45: c_int,
    pub enable_wind_shear: c_int,
    pub enable_trajectory_sampling: c_int,
    pub sample_interval: c_double,   // meters
    pub enable_pitch_damping: c_int,
```

```
    pub enable_precession_nutation: c_int,
    pub enable_spin_drift: c_int,
    pub enable_magnus: c_int,
    pub enable_coriolis: c_int,
}
```

**Metric Units in the FFI**

Unlike the CLI (which accepts imperial units by default), the FFI uses **metric SI units throughout**: velocity in m/s, mass in kg, distances in meters. This simplifies the interface for mobile apps that work in SI internally. Boolean flags are represented as `c_int` values (0 = false, 1 = true), and optional values use `NAN` as a sentinel—for example, `latitude` is set to `NAN` when Coriolis corrections are not desired.

### Drag Model Encoding

The `bc_type` field maps integer codes to drag models:

| Integer Code | Drag Model |
|---|---|
| 0 | G1 (default) |
| 1 | G7 |
| 2 | G2 |
| 3 | G5 |
| 4 | G6 |
| 5 | G8 |
| 6 | GI |
| 7 | GS |

#### `FFITrajectoryResult`

The result structure returns summary statistics and an array of trajectory points allocated on the Rust heap:

Listing 27.2: The `FFITrajectoryResult` struct.

```
#[repr(C)]
pub struct FFITrajectoryResult {
    pub max_range: c_double,
    pub max_height: c_double,
    pub time_of_flight: c_double,
    pub impact_velocity: c_double,
    pub impact_energy: c_double,
```

```
    pub points: *mut FFITrajectoryPoint,
    pub point_count: c_int,
    pub sampled_points: *mut FFITrajectorySample,
    pub sampled_point_count: c_int,
    pub min_pitch_damping: c_double,
    pub transonic_mach: c_double,
    pub final_pitch_angle: c_double,
    pub final_yaw_angle: c_double,
    pub max_yaw_angle: c_double,
    pub max_precession_angle: c_double,
}
```

Each `FFITrajectoryPoint` contains time, 3-D position (in standard ballistic coordinates: $x$ lateral, $y$ vertical, $z$ downrange), velocity magnitude, and kinetic energy. Each `FFITrajectorySample` adds distance, Mach number, and spin rate for regularly-spaced intervals.

**Memory Ownership**

The `points` and `sampled_points` pointers reference Rust-allocated heap memory. The calling code **must** call `ballistics_free_trajectory_result` to deallocate the result, or memory will leak. Never call `free()` from C on these pointers—the Rust allocator and the C allocator may not be the same.

### 27.1.3 Exported Functions

The FFI exports four primary computation functions and two deallocation functions:

| Function | Purpose |
|---|---|
| `ballistics_calculate_trajectory` | Full trajectory computation with wind, atmosphere, and all physics options. Returns a heap-allocated `FFITrajectoryResult`. |
| `ballistics_calculate_zero_angle` | Computes the muzzle angle needed to zero at a given distance. Returns the angle in radians, or `NAN` on error. |
| `ballistics_quick_trajectory` | Simplified single-value function: given muzzle velocity, BC, sight height, zero distance, and target distance, returns the drop at the target. |
| `ballistics_monte_carlo` | Runs a Monte Carlo simulation and returns statistical results (ranges, impact positions, mean, standard deviation, hit probability). |

| | |
|---|---|
| `ballistics_free_trajectory_result` | Memory deallocation functions for results returned by the above. |
| `ballistics_free_monte_carlo_results` | |

### The Main Trajectory Function

The signature of the primary computation function:

Listing 27.3: The main FFI trajectory function.

```
#[no_mangle]
pub extern "C" fn ballistics_calculate_trajectory(
    inputs: *const FFIBallisticInputs,
    wind: *const FFIWindConditions,
    atmosphere: *const FFIAtmosphericConditions,
    max_range: c_double,    // meters
    step_size: c_double,    // integration step (ms)
) -> *mut FFITrajectoryResult
```

The function accepts null pointers for `wind` and `atmosphere`—when null, defaults are used (zero wind, ICAO standard atmosphere). The `max_range` parameter is in meters. If the inputs pointer is null, the function returns `null_mut()`.

Internally, the function:

1. Calls `convert_inputs` to translate `FFIBallisticInputs` into a native `BallisticInputs`.
2. Creates a `TrajectorySolver` with the converted inputs.
3. Calls `solver.solve()` to run the integration.
4. Converts the result into FFI-compatible structures.
5. Uses `Box::into_raw` to transfer ownership to the caller.

## 27.1.4 Building the FFI Library

The `Cargo.toml` specifies `staticlib` and `cdylib` crate types, so a standard release build produces both static and dynamic libraries:

Listing 27.4: Building the FFI library.

```
# Build for the host platform
cargo build --release

# The static library (for iOS, embedded)
ls target/release/libballistics_engine.a
```

```
# The dynamic library (for Python, desktop apps)
ls target/release/libballistics_engine.dylib  # macOS
ls target/release/libballistics_engine.so     # Linux
```

For iOS, cross-compile for the Apple targets:

Listing 27.5: Cross-compiling for iOS.

```
# Add iOS targets
rustup target add aarch64-apple-ios
rustup target add aarch64-apple-ios-sim

# Build static libraries
cargo build --release --target aarch64-apple-ios
cargo build --release --target aarch64-apple-ios-sim

# Create an XCFramework
xcodebuild -create-xcframework \
  -library target/aarch64-apple-ios/release/libballistics_engine.a \
  -library target/aarch64-apple-ios-sim/release/libballistics_engine.a \
  -output BallisticsEngine.xcframework
```

### 27.1.5 Calling from C

Here is a minimal C program that computes a .308 Winchester trajectory using the FFI. Note the metric input values:

Listing 27.6: C example: computing a trajectory via FFI.

```
// test_ffi.c
#include <stdio.h>
#include <math.h>

// FFI structures (must match Rust definitions exactly)
typedef struct {
    double muzzle_velocity, muzzle_angle, bc_value;
    double bullet_mass, bullet_diameter;
    int bc_type;
    double sight_height, target_distance, temperature;
    double twist_rate;
    int is_twist_right;
    double shooting_angle, altitude, latitude;
    double azimuth_angle;
```

```
    int use_rk4, use_adaptive_rk45;
    int enable_wind_shear, enable_trajectory_sampling;
    double sample_interval;
    int enable_pitch_damping, enable_precession_nutation;
    int enable_spin_drift, enable_magnus, enable_coriolis;
} FFIBallisticInputs;

// ... result struct declarations omitted for brevity ...

// Extern declarations
extern void* ballistics_calculate_trajectory(
    const FFIBallisticInputs*, const void*,
    const void*, double, double);
extern void ballistics_free_trajectory_result(void*);

int main() {
    // .308 Win, 168gr SMK, G7 BC 0.462
    FFIBallisticInputs inputs = {
        .muzzle_velocity = 822.96,  // 2700 fps -> m/s
        .bc_value = 0.462,
        .bullet_mass = 0.01089,     // 168 gr -> kg
        .bullet_diameter = 0.00782, // .308 in -> meters
        .bc_type = 1,               // G7
        .use_rk4 = 1,
        .use_adaptive_rk45 = 1,
        .latitude = NAN,            // No Coriolis
    };

    // Compute trajectory to 1000 yards (914.4 m)
    void* result = ballistics_calculate_trajectory(
        &inputs, NULL, NULL, 914.4, 0.001);

    if (result) {
        // ... process result ...
        printf("Trajectory computed successfully\n");
        ballistics_free_trajectory_result(result);
    }
    return 0;
}
```

Listing 27.7: Compiling and running the C example.

```
gcc -o test_ffi test_ffi.c \
  -L target/release -lballistics_engine \
  -lpthread -ldl -lm
```

```
LD_LIBRARY_PATH=target/release ./test_ffi
```

### 27.1.6 Calling from Swift

On iOS, the XCFramework is imported directly and the C functions are available as global Swift functions:

Listing 27.8: Swift example using the FFI.

```
import BallisticsEngine

// Create inputs for a .308 Win, 168gr SMK
var inputs = FFIBallisticInputs()
inputs.muzzle_velocity = 822.96  // m/s
inputs.bc_value = 0.462
inputs.bullet_mass = 0.01089     // kg
inputs.bullet_diameter = 0.00782 // m
inputs.bc_type = 1               // G7
inputs.use_rk4 = 1
inputs.use_adaptive_rk45 = 1
inputs.latitude = .nan

// Compute trajectory to 1000 yards (914.4 m)
if let result = ballistics_calculate_trajectory(
    &inputs, nil, nil, 914.4, 0.001
) {
    let tof = result.pointee.time_of_flight
    let vel = result.pointee.impact_velocity
    print("Time of flight: \(tof) s")
    print("Impact velocity: \(vel) m/s")
    ballistics_free_trajectory_result(result)
}
```

**Swift Type Safety**

For production iOS apps, wrap the C FFI in a Swift class that handles memory management automatically via `deinit`. Use RAII patterns to guarantee deallocation—never leave the `ballistics_free_trajectory_result` call to manual discipline.

## 27.2 WebAssembly: Compiling for the Browser

WebAssembly (WASM) lets the same Rust physics engine run inside a web browser at near-native speed. The `src/wasm.rs` module provides two WASM APIs: a *command-line emulator* (`WasmBallistics`) that parses CLI-style strings, and an *object-oriented calculator* (`Calculator`) that provides a fluent JavaScript API.

### 27.2.1 The `WasmBallistics` API

The `WasmBallistics` struct wraps the engine's command parser. It accepts the same command strings you would type in a terminal—including all flags and subcommands—and returns the output as a JavaScript string:

Listing 27.9: `WasmBallistics` in `src/wasm.rs`.

```
#[wasm_bindgen]
pub struct WasmBallistics;

#[wasm_bindgen]
impl WasmBallistics {
    #[wasm_bindgen(constructor)]
    pub fn new() -> Self { WasmBallistics }

    /// Run a command and return the output
    #[wasm_bindgen(js_name = runCommand)]
    pub fn run_command(&self, command: &str)
        -> Result<String, JsValue>
    {
        // Parse command, route to handler, return output
    }
}
```

From JavaScript, usage is straightforward:

Listing 27.10: JavaScript: using `WasmBallistics`.

```
import init, { WasmBallistics }
    from './pkg/ballistics_engine.js';

await init();
const engine = new WasmBallistics();

// Same command syntax as the CLI
const output = engine.runCommand(
    'ballistics trajectory -v 2700 -b 0.462 '
```

```
  + '-m 168 -d 0.308 --drag-model g7 '
  + '--auto-zero 100 --max-range 1000'
);
console.log(output);

// JSON output for programmatic use
const json = engine.runCommand(
    'ballistics trajectory -v 2700 -b 0.462 '
  + '-m 168 -d 0.308 --drag-model g7 '
  + '--auto-zero 100 --max-range 1000 -o json'
);
const data = JSON.parse(json);
```

The WASM command parser supports all four subcommands: **`trajectory`**, **`zero`**, `monte-carlo`, and `estimate-`**`bc`**. It handles quoted arguments, unit systems (`--units metric`), and all advanced physics flags.

**CLI Parity**

Every flag accepted by the native CLI is also accepted by `WasmBallistics.runCommand()`. The WASM layer re-implements the same argument parser as the native binary, translating each flag into the corresponding `BallisticInputs` field. This means you can develop and test commands in the terminal, then deploy them in the browser with zero changes.

### 27.2.2 The `Calculator` API

For applications that prefer an object-oriented interface over string parsing, the `Calculator` struct provides a fluent setter API with sensible defaults. The defaults correspond to a .308 Winchester, 168-grain bullet at 2700 fps in ICAO standard conditions:

Listing 27.11: The `Calculator` struct definition.

```
#[wasm_bindgen]
pub struct Calculator {
    bc: f64,                    // 0.475
    velocity_fps: f64,          // 2700.0
    mass_grains: f64,           // 168.0
    diameter_inches: f64,       // 0.308
    drag_model: String,         // "G7"
    wind_speed_mph: f64,        // 0.0
    wind_direction_deg: f64,    // 90.0
    temperature_f: f64,         // 59.0
    pressure_inhg: f64,         // 29.92
    humidity_percent: f64,      // 50.0
```

```
    altitude_ft: f64,            // 0.0
    sight_height_inches: f64,    // 2.0
    zero_range_yards: Option<f64>,
    max_range_yards: f64,        // 1000.0
    enable_spin_drift: bool,
    enable_coriolis: bool,
    twist_rate_inches: Option<f64>,
    latitude_deg: Option<f64>,
}
```

**Imperial Defaults in WASM**

Unlike the FFI (which uses SI units), the `Calculator` uses **imperial units**—fps, grains, inches, Fahrenheit—matching the convention used by the CLI. This makes the API immediately useful for the U.S. shooting sports audience that is the primary target of browser-based ballistic calculators.

Fluent setters return `self`, enabling JavaScript method chaining:

Listing 27.12: JavaScript: fluent Calculator API for 6.5 Creedmoor.

```
import init, { Calculator }
    from './pkg/ballistics_engine.js';

await init();

// 6.5 Creedmoor: 140gr ELD-M, G7 BC 0.315, 2710 fps
const result = new Calculator()
    .setVelocity(2710)
    .setBC(0.315)
    .setMass(140)
    .setDiameter(0.264)
    .setDragModel('G7')
    .setZeroRange(100)
    .setWind(10, 90)          // 10 mph from the right
    .setAltitude(5000)        // 5000 ft elevation
    .setTemperature(40)       // 40 deg F
    .calculateTrajectory(1000);

console.log('Drop:', result.drop_inches, 'inches');
console.log('Wind:', result.windage_inches, 'inches');
```

The available setters are:

| JavaScript Method | Parameter |
|---|---|
| `setBC(bc)` | Ballistic coefficient |
| `setVelocity(fps)` | Muzzle velocity (fps) |
| `setMass(grains)` | Bullet weight (grains) |
| `setDiameter(inches)` | Bullet diameter (inches) |
| `setDragModel(model)` | Drag model string (“G1”, “G7”, etc.) |
| `setWind(mph, deg)` | Wind speed and direction |
| `addWindSegment(mph, deg, yd)` | Add a downrange wind segment (to until-distance) |
| `clearWindSegments()` | Remove all downrange wind segments |
| `setTemperature(f)` | Temperature (°F) |
| `setPressure(inhg)` | Barometric pressure (inHg) |
| `setHumidity(pct)` | Humidity (0–100) |
| `setAltitude(ft)` | Altitude (feet) |
| `setSightHeight(in)` | Scope height above bore (inches) |
| `setZeroRange(yd)` | Zero distance (yards) |
| `setMaxRange(yd)` | Maximum range (yards) |
| `enableSpinDrift(bool, rate)` | Spin drift with twist rate |
| `enableCoriolis(bool, lat)` | Coriolis with latitude |

For wind that varies along the path—a muzzle reading plus downrange sensor stations, say—chain `addWindSegment(speedMph, directionDeg, untilYards)` calls (added in version 0.19.0). Each segment applies from the previous boundary to its until-distance, the wind is zero beyond the last segment, and any segments override the scalar `setWind`. The same applies through the command parser via the repeatable `--wind-segment` flag.

**Wind Direction Convention (0.19.0)**

Both WASM APIs (and the C FFI `direction` field) use the wind-from convention: 0° is a headwind, 90° is from the right (drifting the bullet left), 180° a tailwind, 270° from the left—matching `WindSock` and the language bindings. Versions *before* 0.19.0 built the wind vector on this path with the opposite sign, so a given `direction` drifted the bullet the wrong way; if you calibrated against pre-0.19.0 output, re-check your wind directions.

### 27.2.3 Building the WASM Module

The recommended build tool is `wasm-pack`, which compiles the Rust code, generates JavaScript glue, and produces an npm-compatible package:

Listing 27.13: Building the WASM module with wasm-pack.

```
# Install wasm-pack
cargo install wasm-pack

# Build for browsers (ES module output)
wasm-pack build --target web --release

# Build for Node.js (CommonJS output)
wasm-pack build --target nodejs --release \
  --out-dir pkg-nodejs

# The output directory
ls pkg/
# ballistics_engine.d.ts
# ballistics_engine.js
# ballistics_engine_bg.wasm
# ballistics_engine_bg.wasm.d.ts
# package.json
```

The `--target web` flag produces ES module output suitable for direct `import` in modern browsers. The `--target nodejs` flag produces CommonJS output for server-side use.

**WASM Feature Limitations**

The `online` and `pdf` Cargo features are automatically disabled for WASM builds because they depend on system libraries (`ureq` for HTTP, `printpdf` for PDF) that are not available in the browser sandbox. Build with `--no-default-features`:

```
wasm-pack build --target web --release \
  -- --no-default-features
```

The core trajectory, zeroing, Monte Carlo, and BC estimation commands all work without optional features.

### 27.2.4 Integrating into a Web Page

A minimal HTML page that loads the WASM module and computes a trajectory:

Listing 27.14: HTML: minimal WASM integration.

```
<!DOCTYPE html>
<html>
<head><title>Ballistics Calculator</title></head>
<body>
```

```
<pre id="output">Computing...</pre>
<script type="module">
    import init, { WasmBallistics }
        from './pkg/ballistics_engine.js';

    async function run() {
        await init();
        const engine = new WasmBallistics();

        // .308 Win, 168gr SMK, G7 BC 0.462
        const output = engine.runCommand(
            'ballistics trajectory '
          + '-v 2700 -b 0.462 -m 168 -d 0.308 '
          + '--drag-model g7 --auto-zero 100 '
          + '--max-range 1000'
        );

        document.getElementById('output')
            .textContent = output;
    }
    run();
</script>
</body>
</html>
```

Listing 27.15: Serving the WASM app locally.

```
# Serve with any static file server
python3 -m http.server 8080
# Open http://localhost:8080 in your browser
```

The WASM binary is approximately 2–3 MB after gzip compression (depending on features), and the first trajectory computation typically completes within 10 ms on modern hardware—fast enough for interactive use.

## 27.3 Python Bindings via PyO3

For data scientists and ballisticians who work in the Python ecosystem, `ballistics-engine` provides Python bindings via `PyO3`[1], the leading Rust-to-Python bridge. The bindings compile to a native Python extension module (`.so` or `.pyd`) that can be imported directly with `import ballistics_engine`.

[1] https://pyo3.rs

### 27.3.1 Architecture of the Python Bindings

The Python bindings use the `cdylib` crate type (the same dynamic library used by FFI) but with PyO3's macros to expose Rust types as Python classes and functions. The binding layer lives in a separate workspace member (`ballistics-engine-py/`) to keep the core library free of Python-specific dependencies.

The Python API mirrors the Rust library API:

- A `BallisticInputs` Python class wraps the Rust struct, with Python-friendly property setters.
- A `solve_trajectory()` function accepts a `BallisticInputs` object and returns a Python dictionary with trajectory data.
- A `calculate_zero_angle()` function returns the zero angle for a given distance.
- A `run_monte_carlo()` function returns NumPy-compatible arrays of impact positions.

### 27.3.2 Installation

The Python package is built with `maturin`, the standard tool for building PyO3 packages:

Listing 27.16: Building and installing the Python bindings.

```
# Install maturin
pip install maturin

# Build and install in development mode
cd ballistics-engine-py
maturin develop --release

# Or build a wheel for distribution
maturin build --release
pip install target/wheels/ballistics_engine-*.whl
```

### 27.3.3 Using from Python

Once installed, the module is available as a standard Python import:

Listing 27.17: Python: computing a .308 Win trajectory.

```
import ballistics_engine as be

# .308 Win, 168gr SMK, G7 BC 0.462
inputs = be.BallisticInputs()
inputs.muzzle_velocity = 2700  # fps
```

```
inputs.bc_value = 0.462
inputs.bullet_mass = 168       # grains
inputs.bullet_diameter = 0.308 # inches
inputs.drag_model = "G7"

# Compute trajectory zeroed at 100 yards
result = be.solve_trajectory(
    inputs,
    zero_distance=100,
    max_range=1000
)

# Access results
for point in result['trajectory']:
    print(f"{point['range_yd']:>5.0f} yd  "
          f"{point['drop_in']:>8.1f} in  "
          f"{point['velocity_fps']:>7.0f} fps")
```

### 27.3.4 Monte Carlo in Python

Monte Carlo simulations return impact data that integrates naturally with `NumPy` and `matplotlib`:

Listing 27.18: Python: Monte Carlo simulation and visualization.

```
import ballistics_engine as be
import numpy as np
import matplotlib.pyplot as plt

inputs = be.BallisticInputs()
inputs.muzzle_velocity = 2700
inputs.bc_value = 0.462
inputs.bullet_mass = 168
inputs.bullet_diameter = 0.308
inputs.drag_model = "G7"

# Run 1000 Monte Carlo iterations
mc = be.run_monte_carlo(
    inputs,
    zero_distance=100,
    target_distance=600,
    n_simulations=1000,
    velocity_std=15,  # fps standard deviation
    bc_std=0.005,
    wind_speed_std=2  # mph standard deviation
)
```

```
# Plot impact pattern
plt.scatter(mc['windage'], mc['drop'],
            alpha=0.3, s=5)
plt.xlabel('Windage (inches)')
plt.ylabel('Drop (inches)')
plt.title('600 yd Monte Carlo: .308 Win 168gr SMK')
plt.gca().set_aspect('equal')
plt.show()

# Statistics
print(f"CEP: {mc['cep_inches']:.2f} inches")
print(f"Mean drop: {mc['mean_drop']:.1f} inches")
```

**Performance: Python vs. CLI**

The Python bindings call directly into the Rust library with zero serialization overhead—there is no subprocess spawning or JSON parsing. A 1,000-iteration Monte Carlo simulation completes in approximately 2–4 seconds via the Python API, versus 3–5 seconds via the CLI (which includes process startup and output formatting). For batch analysis of thousands of load combinations, the Python API is the fastest path.

## 27.4 Protocol Surfaces: `solve-json` and MCP

An FFI struct is a binding. A JSON document is a *contract*. Version 0.25.0 added both a versioned request/response protocol and a Model Context Protocol server that speaks it, for callers who would rather cross a process boundary than a language one.

### 27.4.1 solve-json v1

The `solve-json` subcommand reads one v1 request from standard input and writes one compact JSON envelope, plus a newline, to standard output:

Listing 27.19: One solve, one document

```
ballistics solve-json < request.json > response.json
```

Every dimensional field is explicit SI—meters, kilograms, kelvin, pascals, radians—regardless of the global `--units` setting, because a protocol should not have a mode. The command reads no profiles, no files, and no network, and input is capped at 1 MiB.

Listing 27.20: A minimal v1 request

```
{
  "schema_version": 1,
  "projectile": {
    "mass_kg": 0.01134, "diameter_m": 0.00782, "length_m": 0.031,
    "drag_model": "G7", "ballistic_coefficient": 0.243
  },
  "rifle": { "muzzle_velocity_mps": 823.0, "sight_height_m": 0.05,
             "muzzle_height_m": 1.2 },
  "shot": { "max_range_m": 300.0, "muzzle_angle_rad": 0.001 },
  "atmosphere": { "altitude_m": 0.0, "temperature_k": 293.15,
                  "pressure_pa": 100000.0, "relative_humidity": 0.4 },
  "wind": { "speed_mps": 0.0, "direction_from_rad": 0.0 },
  "solver": { "method": "rk45", "time_step_s": 0.001 },
  "sampling": { "interval_m": 50.0 }
}
```

The envelope names the engine that produced it and separates the solve's own commentary from its numbers:

Listing 27.21: The response envelope (summary section)

```
{
  "schema_version": 1,
  "engine_version": "0.32.0",
  "status": "ok",
  "assumptions": [],
  "warnings": [
    {
      "code": "rk45_time_step_ignored",
      "message": "Adaptive RK45 owns its time step; the explicitly supplied fixed
    time step is retained in resolved_request but ignored by integration.",
      "path": "$.solver.time_step_s"
    }
  ],
  "summary": {
    "actual_range_m": 300.0,
    "maximum_height_m": 1.2333933080864987,
    "time_of_flight_s": 0.410528107905597,
    "terminal_speed_mps": 649.335479069963,
    "terminal_energy_j": 2390.679320029034,
    "stability_factor": 4.095309067995591,
    "termination": "max_range"
  }
```

```
}
```

Alongside `summary` the envelope carries `samples` and a `resolved_request`—the request as the engine actually understood it, with every default filled in. That last field is the one that saves arguments: it tells you what was solved, not what you thought you asked for.

**What v1 Guarantees**

All objects reject unknown fields. Removing a field, changing its unit or sign, renaming an enum value, or changing a field's meaning requires a new schema version—those invariants hold absolutely. Within v1, growth happens by addition under two rules: a new *request* field must be optional and default to the exact pre-existing behavior, and a new *response* field must be omitted whenever its feature is inactive, so a response that does not exercise it stays byte-identical. A consumer must therefore tolerate response fields it does not recognise rather than rejecting them.

Exit status is part of the contract: 0 for a success envelope, 1 for an I/O or internal failure, 2 for malformed JSON or a validation failure, and 3 for a resource limit or a solve failure. Malformed JSON carries one-based `line` and `column` coordinates; a contained panic becomes a generic `internal_error` with no payload or backtrace in the envelope. The Rust side is public as `ballistics_engine::solve_json` (the DTOs and decoder) and `ballistics_engine::solve_v1` (the transport-free service).

### 27.4.2 Bounds, and One Place to Condition Inputs

Two v0.30.0 changes are worth knowing if you drive the engine through a protocol or a binding rather than the CLI.

First, `solve-json` v1 now rejects projectile values that cannot describe a projectile. `projectile.mass_kg`, `diameter_m`, `length_m` and `ballistic_coefficient` carry physical bounds and return a typed `invalid_value` error with the exact JSONPath. This came out of fuzzing, which got the engine to accept and solve a bullet weighing $1.1 \times 10^{-66}$ kg. The bounds are enormous—1 mg to 100 kg, 0.1 mm to 1 m—so no real load is affected. `rifle.muzzle_velocity_mps` and `shot.max_range_m` are deliberately left unbounded: the MCP server documents a split where a structurally valid but unsolvable request returns a *tool* error rather than a protocol error, and bounding those would reclassify it.

Second—and this one bit the Python binding—the BC reference-standard conversion, the SI-derived imperial mirror fields, and the powder-resolved muzzle velocity were applied only in `TrajectorySolver::new`. `fast_integrate` and `fast_integrate_with_segments` are public and are called directly by bindings, so those callers silently bypassed all three: set a conditioned field, get no error and no effect. All three now live in one idempotent `BallisticInputs::normalize_for_solve`, called by the constructor and by both fast entry points, pinned by a cross-entry-point parity test.

**If You Call a `fast_*` Entry Point**

The lesson generalises. A "fast path" that skips the constructor skips whatever the constructor was doing on your behalf. When you bypass a public API's normal entry point, check what that entry point does before assuming the shortcut is equivalent.

### 27.4.3 The MCP Server

Listing 27.22: Running the MCP server

```
ballistics mcp
```

`ballistics mcp` runs a zero-dependency Model Context Protocol server over stdio, exposing two tools: `solve`, which takes the solve-json v1 schema above, and `engine_info`. It is the same solver, the same contract, and the same envelope—only the framing differs.

## 27.5 Embedding `ballistics-engine` in Your Application

Whether you use the FFI, WASM, or Python bindings, the embedding pattern is the same:

1. **Construct inputs**: build the equivalent of `BallisticInputs` in your language.
2. **Call the solver**: pass the inputs to the trajectory function.
3. **Process results**: extract the data you need from the result structure.
4. **Free memory**: for FFI, call the deallocation function. For WASM and Python, garbage collection handles this automatically.

### 27.5.1 Choosing the Right Binding

| | C FFI | WASM | Python |
|---|---|---|---|
| **Best for** | Mobile apps, embedded systems, game engines | Web browsers, Electron apps | Data analysis, scripting, research |
| **Performance** | Native (fastest) | Near-native (5–10% overhead) | Native solver; Python overhead for setup |
| **Memory mgmt** | Manual (`free` calls) | Automatic (JS GC) | Automatic (Python GC) |
| **Unit system** | Metric (SI) | Imperial (CLI default) | Configurable |
| **Output** | C structs with pointers | Strings or JS objects | Python dicts and lists |

| | | | |
|---|---|---|---|
| **Build tool** | `cargo build` | `wasm-pack` | `maturin` |

### 27.5.2 The `convert_inputs` Pattern

Every binding ultimately calls `convert_inputs` (or an equivalent translation function) to transform language-specific input types into the internal `BallisticInputs`:

Listing 27.23: The `convert_inputs` function in `src/ffi.rs`.

```
fn convert_inputs(inputs: &FFIBallisticInputs)
    -> BallisticInputs
{
    let mut bi = BallisticInputs::default();

    bi.muzzle_velocity = inputs.muzzle_velocity;
    bi.bc_value = inputs.bc_value;
    bi.bullet_mass = inputs.bullet_mass;
    bi.bullet_diameter = inputs.bullet_diameter;
    bi.use_rk4 = inputs.use_rk4 != 0;
    bi.use_adaptive_rk45 = inputs.use_adaptive_rk45 != 0;

    bi.bc_type = match inputs.bc_type {
        1 => DragModel::G7,
        2 => DragModel::G2,
        3 => DragModel::G5,
        // ... remaining models
        _ => DragModel::G1,
    };

    if !inputs.latitude.is_nan() {
        bi.latitude = Some(inputs.latitude);
    }

    // Derived values
    bi.caliber_inches =
        inputs.bullet_diameter / 0.0254;
    bi.weight_grains =
        inputs.bullet_mass / 0.00006479891;

    bi
}
```

This pattern is the key to understanding how external languages interact with the engine. If you are building a new language binding (Ruby, Go, Dart), you need to implement this same translation in

your target language. The WASM layer (`src/wasm.rs`) does the equivalent by parsing command-line strings into `BallisticInputs` fields.

## 27.6 Cross-Platform Considerations

### 27.6.1 Surface Consistency: The 0.22.0 Sweep

Three surfaces—the native CLI, the C FFI, and the WASM build—are supposed to be three doors into one engine. Version 0.22.0 audited that claim and found more than a dozen places where they disagreed. If you have ever compared the same load across two of them and found numbers that would not reconcile, one of these is probably why:

- **WASM's drop** was referenced to the bore line while the CLI referenced the line of sight, and WASM's default wind direction was 90° (full crosswind) where the CLI's was 0° (headwind). WASM also never set `ground_threshold` on **`trajectory`**, leaving the library's −100 m default—a 3.5× disagreement with the CLI on the same command.
- **FFI trajectory samples** reported a hardcoded 0.0 for `mach` and `spin_rate_rps`, and `ballistics_quick_trajectory` referenced drop to the bore rather than the sight line.
- **`estimate-bc` drops** were parsed as yards on the CLI and inches in WASM—a 36× mismatch (see Chapter 7).
- **`--twist-rate`** was read as inches regardless of `--units` on WASM **`trajectory`**, so a metric 1:10 twist produced zero stability.
- **Powder-temperature modeling** in WASM used a hardcoded 70 °F reference and ignored `--temperature` entirely, and `--powder-temp`'s default was being reinterpreted as 70 °C in metric mode.
- **Metric `--bc-table` lookups** fed the tables mm and m/s where they expect inches and fps, and `--bc-adjustment` affected only the *displayed* BC when a `--bc-table` was also supplied, not the trajectory.
- **Metric PDF dope cards** wrote raw metric values under imperial labels, and the card's density-altitude line applied the 120 ft-per-°C rule to a Fahrenheit delta, overstating the temperature correction by about 1.8×.

All are fixed as of v0.22.0. The general lesson for anyone integrating against more than one surface: pin a cross-surface parity test of your own, because a shared core does not by itself guarantee a shared boundary.

### 27.6.2 Browser Parity and a Hardened FFI Boundary

Version 0.27.0 continued the consolidation. The browser terminal gained `trajectory --plot` (the same `terminal_plot` code, moved into the library unchanged), the full `true-velocity` command including multi-observation joint MV + BC calibration, and `monte-carlo --wez`. All three render character-identical output to the native CLI—multi-observation JSON floats may differ in the last three of seventeen significant digits across instruction-set architectures, and nothing else. The truing core and the WEZ compute both moved into library modules (`ballistics_engine::truing` and `::wez`) rather than being reimplemented, which is what makes that parity structural instead of aspirational. The WASM handlers also stopped silently ignoring bare tokens and value-less flags: previously a typo in a multi-observation truing call could fit fewer observations than you supplied and report success.

Two boundaries were hardened at the same time. The MCP server now rejects fractional JSON-RPC ids (integral floats such as `2.0` are still accepted), enforces the `additionalProperties: false` that `engine_info` already advertised, and returns `-32002` for requests that arrive before `initialize` (`ping` exempted). And on the C side, `ballistics_calculate_trajectory_with_drag_table` and its zero-angle variant now reject a custom drag table longer than `MAX_FFI_DRAG_TABLE_LEN` (4096 rows) rather than attempting an unbounded allocation from a caller-supplied length.

**The Binary Is Behind a Feature Gate**

Since v0.27.0 the five dependencies used only by the **`ballistics`** binary—`clap`, `clap_complete`, `csv`, `dirs`, `strsim`—live behind a default-on `cli` feature, and the `[[bin]]` target declares `required-features = ["cli"]`. Plain `cargo build` and `cargo install` are unchanged, and a library consumer can now drop all five with `default-features = false`. **But a `--no-default-features` build that needs the binary must add `--features cli`**—otherwise the bin target is simply skipped, with no error and no executable.

### 27.6.3 Endianness and Alignment

All `#[repr(C)]` structs use the platform's native byte order and alignment. On all currently supported targets (x86_64, aarch64, wasm32), `c_double` is IEEE 754 double-precision (8 bytes) and alignment is natural. If you are serializing FFI structures across the network (e.g., for a remote computation service), you must handle endianness explicitly.

### 27.6.4 Thread Safety

The core solver functions are **stateless**: they take inputs by reference and return results by value. There is no global mutable state. This means:

- Multiple FFI calls can run concurrently on different threads.
- The WASM module can be shared across Web Workers.

- Python bindings release the GIL during computation, allowing true parallelism when called from multiple threads.

Listing 27.24: Python: parallel trajectory computation with threads.

```
from concurrent.futures import ThreadPoolExecutor
import ballistics_engine as be

def compute_drop(velocity):
    inputs = be.BallisticInputs()
    inputs.muzzle_velocity = velocity
    inputs.bc_value = 0.462
    inputs.bullet_mass = 168
    inputs.bullet_diameter = 0.308
    result = be.solve_trajectory(
        inputs, zero_distance=100, max_range=1000)
    return result['drop_inches_at_1000']

# Compute in parallel across 8 threads
velocities = range(2500, 2800, 10)
with ThreadPoolExecutor(max_workers=8) as pool:
    drops = list(pool.map(compute_drop, velocities))
```

### 27.6.5 Error Handling Across Boundaries

Each binding uses the idiomatic error mechanism of its target language:

- **FFI**: functions return `NULL` or `NAN` on failure. The caller must check for null pointers before dereferencing. Debug builds include `eprintln!` diagnostics behind `#[cfg(debug_assertions)]`.
- **WASM**: functions return `Result<String, JsValue>`. In JavaScript, errors are caught with standard `try/catch`.
- **Python**: PyO3 maps Rust `Result::Err` values to Python exceptions, which can be caught with `try/except`.

### 27.6.6 WASM Random Number Generation

Monte Carlo simulations require random number generation, which is platform-specific. The `Cargo.toml` includes the `getrandom` crate with the `wasm_js` feature enabled for WASM builds:

Listing 27.25: WASM-compatible random number support.

```
[target.'cfg(target_arch = "wasm32")'.dependencies]
getrandom = { version = "0.3", features = ["wasm_js"] }
```

This routes random number generation through the browser's `crypto.getRandomValues()` API, ensuring cryptographically secure random numbers for Monte Carlo dispersion modeling.

### 27.6.7 Supported Platforms

| Platform | Target Triple | Binding |
|---|---|---|
| Linux x86_64 | `x86_64-unknown-linux-gnu` | CLI, FFI, Python |
| Linux ARM64 | `aarch64-unknown-linux-gnu` | CLI, FFI, Python |
| macOS Intel | `x86_64-apple-darwin` | CLI, FFI, Python |
| macOS Apple Silicon | `aarch64-apple-darwin` | CLI, FFI, Python |
| Windows | `x86_64-pc-windows-msvc` | CLI, FFI, Python |
| FreeBSD | `x86_64-unknown-freebsd` | CLI, FFI |
| iOS Device | `aarch64-apple-ios` | FFI |
| iOS Simulator | `aarch64-apple-ios-sim` | FFI |
| Android | `aarch64-linux-android` | FFI |
| Web Browser | `wasm32-unknown-unknown` | WASM |
| Node.js | `wasm32-unknown-unknown` | WASM |

Listing 27.26: Managing Rust cross-compilation targets.

```
# List installed targets
rustup target list --installed

# Add a new target
rustup target add aarch64-apple-ios

# Cross-compile for that target
cargo build --release --target aarch64-apple-ios
```

## 27.7 Exercises

1. **Build the WASM module.** Install `wasm-pack` and build the WASM module for the browser. Create a simple HTML page that computes a .308 Win trajectory at 1,000 yards and displays the result.

   ```
   cargo install wasm-pack
   wasm-pack build --target web --release \
     -- --no-default-features
   python3 -m http.server 8080
   ```

2. **Compare FFI and CLI output.** Compile the C FFI test program and compute a trajectory for a 6.5 Creedmoor (140 gr, G7 BC 0.315, 2710 fps). Compare the drop at 600 yards to the same computation from the CLI. Are the results identical?

```
# CLI reference
ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
  --drag-model g7 --auto-zero 100 --max-range 600
```

3. **Test the Calculator API.** In a Node.js script, use the `Calculator` API to configure a .300 Winchester Magnum (190 gr, G7 BC 0.365, 2900 fps) and compute trajectory data at 100-yard intervals to 1,000 yards.

```
wasm-pack build --target nodejs --release \
  --out-dir pkg-nodejs -- --no-default-features
node -e "
const { Calculator } = require('./pkg-nodejs');
const calc = new Calculator()
    .setVelocity(2900).setBC(0.365)
    .setMass(190).setDiameter(0.308)
    .setDragModel('G7').setZeroRange(100);
for (let r = 100; r <= 1000; r += 100) {
    const t = calc.calculateTrajectory(r);
    console.log(r + ' yd:', t);
}
"
```

4. **Explore error handling.** What happens when you pass invalid parameters through each binding? Try a BC of 0, a negative velocity, and a drag model that does not exist. How does each binding report the error?

```
# Test error handling through the CLI for comparison
ballistics trajectory -v -100 -b 0.0 -m 168 -d 0.308 \
  --drag-model G99 --max-range 1000
```

5. **Multi-platform build.** If you have macOS with Xcode installed, build the iOS XCFramework and verify the binary sizes. How large is the static library for each architecture?

```
rustup target add aarch64-apple-ios aarch64-apple-ios-sim
cargo build --release --target aarch64-apple-ios
cargo build --release --target aarch64-apple-ios-sim
ls -lh target/aarch64-apple-ios/release/libballistics_engine.a
ls -lh target/aarch64-apple-ios-sim/release/libballistics_engine.a
```

## What's Next

With the engine's language boundaries mapped, we can now ask the next natural question: *how fast is it?* Chapter 28 profiles the engine from end to end—benchmarking single-trajectory throughput, measuring Monte Carlo parallelism with Rayon, examining cache-friendly memory layouts, and showing how the fast trajectory solver trades accuracy for speed in batch simulations.

# Chapter 28

# Performance

*"Measure twice, optimize once."*

Performance in a ballistic solver is not academic. When a competitive shooter's app computes a firing solution in the field, 100 ms feels instantaneous; 500 ms feels sluggish; anything over a second is unacceptable. When a Monte Carlo simulation runs 10,000 trajectories to generate a probability-of-hit estimate, the per-trajectory cost determines whether the result arrives in seconds or minutes. And when the engine runs inside a browser via WebAssembly, it competes for attention with every other script on the page.

This chapter profiles `ballistics-engine` from the inside out. We begin with benchmark numbers: how many trajectories per second can the engine sustain, and how do those numbers change with different integration methods and physics configurations? We then examine the three performance strategies the engine employs: the fast trajectory solver for batch workloads, Rayon-based parallelism for Monte Carlo, and memory layout decisions that keep the derivative function cache-friendly. We close with practical guidance on profiling the engine yourself.

## 28.1 Benchmark Results

### 28.1.1 The Criterion Benchmark Suite

The engine ships with a Criterion benchmark suite in `benches/trajectory_bench.rs`. Criterion is the standard Rust benchmarking framework: it runs each benchmark hundreds of times, collects timing statistics, and reports the median with confidence intervals.

Listing 28.1: The benchmark harness in `benches/trajectory_bench.rs`.

```
use criterion::{black_box, criterion_group,
                criterion_main, Criterion};
use ballistics_engine::*;

fn bench_trajectory_g7(c: &mut Criterion) {
    c.bench_function("trajectory_g7_1000yd", |b| {
        b.iter(|| {
            let mut inputs = BallisticInputs::default();
            inputs.bc_value = 0.275;
            inputs.bc_type = DragModel::G7;
            inputs.bullet_mass = 0.01088;     // 168gr in kg
            inputs.muzzle_velocity = 822.96;  // 2700 fps
            inputs.bullet_diameter = 0.00782; // .308 in meters
            inputs.target_distance = 914.4;   // 1000yd
            inputs.muzzle_angle = 0.005;
            inputs.sight_height = 0.0508;     // 2 inches
            inputs.use_adaptive_rk45 = true;
            inputs.use_rk4 = true;
            let wind = WindConditions {
                speed: 0.0, direction: 0.0 };
            let atmo = AtmosphericConditions {
                temperature: 15.0, pressure: 1013.25,
                humidity: 50.0, altitude: 0.0 };
            let solver = TrajectorySolver::new(
                black_box(inputs), wind, atmo);
            solver.solve()
        })
    });
}
```

Listing 28.2: Running the benchmark suite.

```
# Run all benchmarks
cargo bench

# Run a specific benchmark
cargo bench -- trajectory_g7
```

### 28.1.2 Single-Trajectory Performance

On a modern system (Apple M2, 2022; or Intel i7-12700, 2022), a single 1,000-yard trajectory with adaptive RK45 and the G7 drag model typically completes in 3–6 ms. The breakdown:

| Configuration | Time (ms) | Trajectories/s |
|---|---|---|
| Euler, no wind, 1000 yd | 0.5–1.0 | ~1,500 |
| Fixed RK4, no wind, 1000 yd | 2.0–3.0 | ~400 |
| Adaptive RK45, no wind, 1000 yd | 3.0–5.0 | ~250 |
| RK45 + wind, 1000 yd | 3.5–6.0 | ~200 |
| RK45 + all effects, 1000 yd | 4.0–7.0 | ~170 |
| Zeroing (100 yd) + RK45, 1000 yd | 30–60 | ~25 |

**The Zeroing Cost**

Zeroing is by far the most expensive operation because it runs 8–15 complete trajectories iteratively (see Section 26.2.3). The total cost is roughly 10×–15× a single trajectory computation. If you are computing many trajectories with the same zero, compute the zero angle once and reuse it—the `calculate_zero_angle` function returns a reusable angle value.

### 28.1.3 Running Your Own Benchmarks

The CLI provides a quick way to measure wall-clock performance using standard shell timing:

Listing 28.3: Timing a trajectory from the command line.

```
# Time a single trajectory computation
time ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000

# Time with all physics effects enabled
time ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000 \
  --enable-spin-drift --twist-rate 10 \
  --enable-coriolis --latitude 45 \
  --enable-magnus --enable-pitch-damping
```

Note that shell timing includes process startup (~5 ms) and output formatting, so it overestimates the pure solver time. For precise measurements, use Criterion or the Rust `std::time::Instant` API from a library call.

### 28.1.4 Integration Method Comparison

The choice of integration method has the most dramatic effect on per-trajectory performance. Here is a typical comparison for a .308 Winchester (168 gr SMK, G7 BC `0.462`, 2700 fps) at 1,000 yards:

Listing 28.4: Comparing integration methods.

```
# Euler (fastest, least accurate)
time ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --max-range 1000 --use-euler

# Fixed RK4 (good balance)
time ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --max-range 1000 --use-rk4-fixed

# Adaptive RK45 (most accurate, default)
time ballistics trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --max-range 1000
```

The Euler method takes roughly 3×–5× fewer steps than RK4 for the same range, but accumulates errors that can reach several inches at 1,000 yards. The adaptive RK45 method takes the fewest total function evaluations for a given accuracy target because it concentrates computation where the trajectory changes rapidly (the transonic zone) and takes large steps in smooth regions (supersonic flight).

**Function Evaluations per Step**

- **Euler**: 1 derivative evaluation per step
- **RK4**: 4 derivative evaluations per step
- **RK45**: 7 derivative evaluations per step (but takes fewer total steps for the same accuracy)

Since the derivative function is the dominant cost, the product (evaluations × steps) determines total runtime. For a 1,000-yard trajectory, Euler at 0.5 ms steps takes ∼3,000 evaluations, RK4 at 1 ms takes ∼6,000, and RK45 takes ∼2,500–4,000 depending on the tolerance.

## 28.2 Monte Carlo Parallelism

Monte Carlo simulations are the most computationally intensive workload the engine handles. A typical 1,000-iteration simulation at 600 yards requires 1,000 independent trajectory computations—each with slightly perturbed inputs for velocity, BC, and wind.

### 28.2.1 Current Implementation

The Monte Carlo loop in `src/cli_api.rs` (`run_monte_carlo_with_wind`) currently runs iterations sequentially. Each iteration:

1. Perturbs the base inputs using normally distributed random variables from `rand_distr`.
2. Runs a full trajectory via `TrajectorySolver::solve()`.

3. Records the impact position, velocity, and range.

Listing 28.5: Monte Carlo loop structure (simplified from `cli_api.rs`).

```
let mut rng = thread_rng();

for _ in 0..params.num_simulations {
    let mut sim_inputs = base_inputs.clone();

    // Perturb velocity
    sim_inputs.muzzle_velocity +=
        Normal::new(0.0, params.velocity_std_dev)
            .unwrap().sample(&mut rng);

    // Perturb BC
    sim_inputs.bc_value +=
        Normal::new(0.0, params.bc_std_dev)
            .unwrap().sample(&mut rng);

    // Solve trajectory
    let solver = TrajectorySolver::new(
        sim_inputs, sim_wind, sim_atmo);
    if let Ok(result) = solver.solve() {
        ranges.push(/* ... */);
        impact_positions.push(/* ... */);
    }
}
```

### 28.2.2 Parallelism with Rayon

The `rayon` crate (version 1.10) is included as a dependency and provides effortless data parallelism via parallel iterators. Because each Monte Carlo iteration is independent—no shared mutable state, no inter-iteration dependencies—the loop is trivially parallelizable by replacing the sequential iterator with a Rayon parallel iterator.

The parallelization opportunity is straightforward:

Listing 28.6: Rayon-parallelized Monte Carlo (conceptual).

```
use rayon::prelude::*;

let results: Vec<_> = (0..num_simulations)
    .into_par_iter()
    .map(|_| {
        let mut rng = rand::thread_rng();
```

```
        let mut sim_inputs = base_inputs.clone();

        // Perturb inputs
        sim_inputs.muzzle_velocity +=
            Normal::new(0.0, vel_std).unwrap()
                .sample(&mut rng);

        // Solve
        let solver = TrajectorySolver::new(
            sim_inputs, wind.clone(), atmo.clone());
        solver.solve()
    })
    .filter_map(|r| r.ok())
    .collect();
```

The key observation is that `BallisticInputs` implements `Clone` and `Send`, and the solver has no global mutable state. Each thread gets its own copy of the inputs and its own thread-local random number generator—no synchronization is needed except at the final `.collect()` barrier.

**Expected Speedup**

On an 8-core machine, Rayon parallelism typically yields a $5\times$–$7\times$ speedup for Monte Carlo (not $8\times$, due to thread spawning overhead and memory allocation contention). A 10,000-iteration simulation that takes 40 seconds sequentially would complete in approximately 6–8 seconds. The benefit scales roughly linearly with core count up to 16–32 cores, after which memory bandwidth becomes the bottleneck.

### 28.2.3 Scaling Behavior

Typical wall-clock times for a .308 Winchester Monte Carlo simulation at 600 yards (with zeroing pre-computed):

| Iterations | Sequential (s) | Parallel, 8 cores (s) |
|---|---|---|
| 100 | 0.4 | 0.1 |
| 500 | 2.0 | 0.4 |
| 1,000 | 4.0 | 0.7 |
| 5,000 | 20 | 3.5 |
| 10,000 | 40 | 7 |

Listing 28.7: Running Monte Carlo simulations at different scales.

```
# 500 iterations (quick estimate)
time ballistics monte-carlo \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 600 \
  --num-sims 500 --velocity-std 15 --bc-std 0.005

# 5000 iterations (high-confidence estimate)
time ballistics monte-carlo \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 600 \
  --num-sims 5000 --velocity-std 15 --bc-std 0.005
```

## 28.3 The Fast Trajectory Solver

For batch workloads like Monte Carlo where absolute accuracy can be traded for speed, the engine provides an alternative solver in `src/fast_trajectory.rs`. This is a stripped-down, fixed-step RK4 integrator that eliminates several sources of overhead present in the main solver.

### 28.3.1 Design Differences

The fast solver differs from the main solver (`src/trajectory_integration.rs`) in several ways:

| Feature | Main Solver | Fast Solver |
|---|---|---|
| Integration method | Adaptive RK45 with error control | Fixed-step RK4 |
| State representation | `Vector3<f64>` and `Vector6<f64>` via nalgebra | Raw `[f64; 6]` arrays |
| Derivative function | Full `compute_derivatives` with all physics effects | Simplified: drag + gravity only |
| Step size | Adaptive (0.1 ms to 20 ms) | Fixed (default 1 ms) |
| Target detection | Linear interpolation + refined sub-step | Linear interpolation only |
| Apex detection | Yes | Yes |
| Advanced effects | Spin, Magnus, Coriolis, precession, pitch damping | None (can be added via segments) |

### 28.3.2 The Simplified Derivative

The fast solver has its own `compute_derivatives` function that computes only drag and gravity—no Coriolis, no Magnus, no spin drift:

Listing 28.8: The fast derivative function in `src/fast_trajectory.rs`.

```
fn compute_derivatives(
    state: &[f64; 6],
    inputs: &BallisticInputs,
    wind_sock: &WindSock,
    base_density: f64,
    drag_model: &DragModel,
    bc: f64,
    has_bc_segments: bool,
    has_bc_segments_data: bool,
) -> [f64; 6] {
    let vel = Vector3::new(state[3], state[4], state[5]);
    let wind_vector =
        wind_sock.vector_for_range_stateless(state[2]);
    let vel_adjusted = vel - wind_vector;
    let v_mag = vel_adjusted.norm();

    let accel = if v_mag < 1e-6 {
        Vector3::new(0.0, -G_ACCEL_MPS2, 0.0)
    } else {
        let v_fps = v_mag * MPS_TO_FPS;

        // Atmosphere from altitude
        let altitude = inputs.altitude + state[1];
        let (_, speed_of_sound) = get_local_atmosphere(
            altitude, inputs.altitude,
            inputs.temperature, inputs.pressure,
            if inputs.humidity > 0.0
                { inputs.humidity } else { 1.0 },
        );
        let mach = v_mag / speed_of_sound;

        let drag_factor =
            get_drag_coefficient(mach, drag_model);
        let cd_to_retard = 0.000683 * 0.30;
        let density_scale = base_density / 1.225;
        let a_drag_ft_s2 = (v_fps * v_fps)
            * drag_factor * cd_to_retard
            * density_scale / bc;
        let a_drag_m_s2 = a_drag_ft_s2 * 0.3048;
        let accel_drag =
            -a_drag_m_s2 * (vel_adjusted / v_mag);

        accel_drag + Vector3::new(0.0, -G_ACCEL_MPS2, 0.0)
    };
```

```
    [vel.x, vel.y, vel.z, accel.x, accel.y, accel.z]
}
```

By stripping out the advanced effects, the fast derivative function is roughly 30–50% faster per evaluation than the full derivative. Multiplied by the thousands of evaluations in a trajectory, this adds up.

### 28.3.3 Raw Arrays vs. nalgebra

The fast solver uses `[f64; 6]` arrays for state vectors instead of nalgebra's `Vector6<f64>`. This avoids the overhead of nalgebra's dimension-checking and dispatch machinery. The RK4 loop becomes tight scalar arithmetic:

Listing 28.9: RK4 with raw arrays in the fast solver.

```
let mut state2 = state;
for j in 0..6 {
    state2[j] = state[j] + 0.5 * dt * k1[j];
}
// ... compute k2 from state2 ...

let mut new_state = state;
for j in 0..6 {
    new_state[j] = state[j]
        + dt * (k1[j] + 2.0*k2[j]
              + 2.0*k3[j] + k4[j]) / 6.0;
}
```

These inner loops are small, contiguous, and compiler-friendly. With `opt-level = 3`, LLVM auto-vectorizes them into SIMD instructions on x86_64 (using SSE2 or AVX) and aarch64 (using NEON).

### 28.3.4 When to Use the Fast Solver

- **Monte Carlo batch runs**: where you need thousands of trajectories and the per-iteration accuracy penalty (typically <0.5% at 1,000 yards) is acceptable.
- **Interactive preview**: where you want to show approximate results instantly while the full solver runs in the background.
- **BC estimation**: where you compare computed drop against measured data at multiple velocities, and the comparison loop needs to be fast.

Listing 28.10: Monte Carlo using the fast solver (internal selection).

```
# The monte-carlo subcommand automatically uses the fast solver
# for batch trajectory evaluation
ballistics monte-carlo \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 1000 \
  --num-sims 1000 --velocity-std 15 --bc-std 0.005
```

## 28.4 Memory Layout and Cache Efficiency

The derivative function is the hottest code path in the engine. It is called 4–7 times per integration step, and a typical 1,000-yard trajectory requires 1,500–3,000 steps. That means the derivative function runs 6,000–21,000 times per trajectory. Any memory access pattern that causes cache misses in this function will dominate runtime.

### 28.4.1 Data Locality in the Hot Path

The derivative function receives its data through a small number of arguments, all of which fit in CPU registers or the L1 cache:

- The 6-element state vector (`[f64; 6]` = 48 bytes).
- The wind vector (3 `f64` = 24 bytes).
- The atmosphere tuple (4 `f64` = 32 bytes).
- The BC value (1 `f64` = 8 bytes).
- A reference to `BallisticInputs` (64-bit pointer).

The total working set of the derivative function is approximately 120 bytes of scalar data plus the `BallisticInputs` reference. On a modern CPU with 32 KB of L1 data cache, this fits entirely within the first cache line boundary—even the `BallisticInputs` struct, at roughly 400 bytes, fits in L1.

### 28.4.2 Drag Table Lookup

The drag coefficient lookup (in `src/drag.rs`) performs binary search followed by cubic interpolation on a sorted array of Mach-vs-$C_D$ pairs. The G7 drag table, for example, contains approximately 80 entries (640 bytes). This table is accessed every derivative evaluation, but because it is read-only and small, it lives in L1 or L2 cache after the first access.

The cubic interpolation uses four consecutive array elements (the two bracketing points plus one neighbor on each side), accessing 32 bytes of contiguous memory—perfectly cache-friendly.

### 28.4.3 The `TrajectoryParams` Separation

Recall from Section 26.4.3 that the integrator does not receive the full 50-field `BallisticInputs`—it receives the leaner `TrajectoryParams`. This is a performance-motivated design:

- `TrajectoryParams` contains approximately 200 bytes of data relevant to the integrator.
- `BallisticInputs` contains approximately 400 bytes, including fields like `bullet_manufacturer` (`Option<String>`) that are never touched during integration.
- By passing the smaller struct, we reduce cache pollution. The integrator's working set stays compact, reducing L1 evictions.

### 28.4.4 Allocation Patterns

The trajectory integrator pre-allocates its output vector:

Listing 28.11: Pre-allocated trajectory output.

```
// Estimate capacity to avoid reallocations
let estimated_steps =
    ((t_end - t_start) / dt_initial) as usize + 100;
let mut trajectory: Vec<(f64, [f64; 6])> =
    Vec::with_capacity(estimated_steps);
```

This single upfront allocation avoids the $O(\log n)$ reallocations that a default `Vec` would incur. For a 3,000-step trajectory, this saves approximately 10–12 heap allocations (each requiring a copy of the existing data).

### 28.4.5 The Effect of LTO and codegen-units

The release profile settings from Section 26.5.6 have a measurable effect on the derivative function:

| Setting | Speedup | Mechanism |
|---|---|---|
| `opt-level = 3` | 1.0× (baseline) | Full optimization |
| `lto = true` | +15–25% | Cross-crate inlining, dead code elimination |
| `codegen-units = 1` | +5–10% | Better register allocation across the whole program |
| Combined | +20–40% | All effects compound |

LTO is especially important for the derivative function because it calls into `src/drag.rs`, `src/atmosphere.rs`, and `src/spin_drift.rs`—all separate modules. Without LTO, each call crosses a crate boundary and cannot be inlined. With LTO, the compiler sees the full call graph and can inline small functions like `get_drag_coefficient` directly into the derivative loop.

Listing 28.12: Comparing builds with and without LTO.

```
# Build with LTO (default release profile)
cargo build --release
time ./target/release/ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --max-range 1000

# Build without LTO for comparison
CARGO_PROFILE_RELEASE_LTO=false cargo build --release
time ./target/release/ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --max-range 1000
```

## 28.5 Alternative Allocators

The engine supports two alternative memory allocators via Cargo features:

Listing 28.13: Allocator feature flags in `Cargo.toml`.

```
[features]
jemalloc = ["dep:tikv-jemallocator"]
mimalloc = ["dep:mimalloc"]

[target.'cfg(not(target_env = "msvc"))'.dependencies]
tikv-jemallocator = { version = "0.6", optional = true }
mimalloc = { version = "0.1", optional = true }
```

### 28.5.1 jemalloc

Facebook's `jemalloc` is optimized for multi-threaded workloads with frequent small allocations. It provides per-thread caches that reduce lock contention and improve allocation throughput. For Monte Carlo workloads where many threads allocate trajectory vectors simultaneously, jemalloc can reduce total runtime by 5–15%.

Listing 28.14: Building with jemalloc.

```
cargo build --release --features jemalloc

# Benchmark comparison
cargo bench --features jemalloc
```

### 28.5.2 mimalloc

Microsoft's `mimalloc` is a compact allocator designed for performance. It typically matches or slightly exceeds jemalloc for single-threaded workloads and has a smaller memory footprint. It works on all platforms including Windows (where jemalloc is not available on MSVC).

Listing 28.15: Building with mimalloc.

```
cargo build --release --features mimalloc

# Works on Windows too
cargo build --release --features mimalloc \
  --target x86_64-pc-windows-msvc
```

**Which Allocator?**

For most users, the default system allocator is fine. The alternative allocators matter most for:

- Monte Carlo with >5,000 iterations (jemalloc recommended)
- Embedded/mobile where memory footprint matters (mimalloc recommended)
- Windows where jemalloc is not available (mimalloc is the only option)

## 28.6 Profiling the Engine

### 28.6.1 CPU Profiling with `perf`

On Linux, `perf` provides hardware-level profiling with minimal overhead:

Listing 28.16: CPU profiling with perf.

```
# Build with debug symbols in release mode
CARGO_PROFILE_RELEASE_DEBUG=true cargo build --release

# Record a profile
perf record -g ./target/release/ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 --drag-model g7 \
  --auto-zero 100 --max-range 1000

# View the profile
perf report
```

A typical profile for a single trajectory computation shows:

| Time (%) | Function |
|---|---|
| 35–45 | `compute_derivatives` |
| 15–20 | `get_drag_coefficient` (or `_full`) |
| 10–15 | `get_local_atmosphere` |
| 8–12 | RK45 step logic (`integrate_trajectory`) |
| 5–8 | Vector arithmetic (nalgebra) |
| 5–10 | Miscellaneous (wind, BC interpolation, etc.) |

The derivative function dominates, as expected. Within it, drag coefficient lookup and atmosphere calculation are the two most expensive operations. Optimizing the drag lookup (e.g., replacing binary search with direct indexing for evenly-spaced Mach tables) or caching atmosphere calculations across steps at the same altitude would yield the largest speedups.

### 28.6.2 Flame Graphs

For visual analysis, generate a flame graph:

Listing 28.17: Generating a flame graph.

```
# Install the flame graph tools
cargo install flamegraph

# Generate a flame graph
cargo flamegraph -- trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 --drag-model g7 \
  --auto-zero 100 --max-range 1000

# Opens flamegraph.svg in your browser
```

### 28.6.3 Profiling on macOS with Instruments

On macOS, Apple's Instruments provides similar capabilities:

Listing 28.18: Profiling with Instruments on macOS.

```
# Build with debug info
CARGO_PROFILE_RELEASE_DEBUG=true cargo build --release

# Profile with Instruments (Time Profiler template)
xcrun xctrace record --template 'Time Profiler' \
  --launch ./target/release/ballistics -- trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 --drag-model g7 \
```

```
  --auto-zero 100 --max-range 1000
```

### 28.6.4 Cache Analysis

For understanding cache behavior, use `cachegrind` (part of Valgrind):

Listing 28.19: Cache analysis with cachegrind.

```
# Run under cachegrind
valgrind --tool=cachegrind ./target/release/ballistics \
  trajectory -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g7 --auto-zero 100 --max-range 1000

# View annotated source
cg_annotate cachegrind.out.* | head -50
```

Typical results show near-zero L1 data cache miss rates for the derivative function (confirming that the working set fits in L1) and low L2 miss rates for the drag table lookup. The most cache-unfriendly operation is usually the output vector push, which periodically triggers a reallocation when the pre-allocated capacity is exceeded.

## 28.7 WASM Performance Considerations

The WASM target introduces performance characteristics that differ from native compilation:

- **No SIMD by default.** WASM SIMD is a separate proposal that requires explicit opt-in. The auto-vectorized loops in the fast solver run as scalar operations in WASM, costing 2–4× versus native SIMD.
- **No thread parallelism.** WASM threads require `SharedArrayBuffer`, which is restricted by cross-origin isolation policies. Monte Carlo runs single-threaded in the browser by default.
- **Smaller memory budget.** The default WASM linear memory starts at 16 MB, which is sufficient for all trajectory operations but may need growth for very large Monte Carlo runs.
- **JIT warmup.** The first trajectory computation in WASM may be 2–3× slower than subsequent ones due to the browser's JIT compiler warming up.

Despite these limitations, WASM trajectory computation is still fast: 5–15 ms for a 1,000-yard trajectory on modern browsers, compared to 3–6 ms natively. The 2–3× overhead is largely attributable to the lack of SIMD.

Listing 28.20: Building with WASM SIMD (experimental).

```
# Enable WASM SIMD (requires recent browser support)
RUSTFLAGS="-C target-feature=+simd128" \
  wasm-pack build --target web --release \
  -- --no-default-features
```

**WASM SIMD Browser Support**

WASM SIMD is supported in Chrome 91+, Firefox 89+, and Safari 16.4+. Enabling it with `-C target-feature=+simd128` produces a WASM binary that will not run in older browsers. Test your target audience before enabling.

## 28.8 Performance Optimization Checklist

For users who want maximum throughput:

1. **Build in release mode**: `cargo build --release`. Debug builds are 10–50× slower.
2. **Enable LTO**: already the default in the release profile (`lto = true`).
3. **Use a single codegen unit**: already the default (`codegen-units = 1`).
4. **Pre-compute the zero angle**: call `calculate_zero_angle` once and reuse for all trajectories at the same zero distance.
5. **Disable unused physics effects**: each enabled effect adds computation to every derivative evaluation. If you do not need Coriolis at 300 yards, do not enable it.
6. **Use the fast solver for batch work**: Monte Carlo and BC estimation benefit from the simplified derivative.
7. **Consider jemalloc**: for Monte Carlo with >5,000 iterations, the allocation pattern benefits from jemalloc.
8. **Enable target CPU features**: compile with `RUSTFLAGS="-C target-cpu=native"` to unlock AVX2/NEON instructions.

Listing 28.21: Building with maximum performance settings.

```
# Maximum performance: native CPU features + jemalloc
RUSTFLAGS="-C target-cpu=native" \
  cargo build --release --features jemalloc

# Verify the optimizations are active
./target/release/ballistics trajectory \
```

```
-v 2700 -b 0.462 -m 168 -d 0.308 --drag-model g7 \
--auto-zero 100 --max-range 1000
```

**SAFETY: Performance vs. Accuracy**

The fast trajectory solver and the Euler integration method sacrifice accuracy for speed. At long range (beyond 800 yards) with significant wind, the fast solver's simplified physics may produce drop predictions that differ from the full solver by 1–3%. **Never** use simplified solvers for load development pressure estimation or for computing maximum effective range where errors could be dangerous. Always validate performance-optimized results against the full adaptive RK45 solver before use in the field.

## 28.9 Exercises

1. **Run the Criterion benchmarks.** Execute `cargo bench` and record the median time for the `trajectory_g7_1000yd` benchmark. How does it compare to the numbers in Section 28.1.2?

```
cargo bench -- trajectory_g7
```

2. **Measure the cost of physics effects.** Time a 6.5 Creedmoor trajectory (140 gr, G7 BC 0.315, 2710 fps) at 1,000 yards with no advanced effects, then with each effect added individually. Which effect has the highest cost?

```
# Baseline
time ballistics trajectory -v 2710 -b 0.315 -m 140 \
  -d 0.264 --drag-model g7 --max-range 1000

# With spin drift
time ballistics trajectory -v 2710 -b 0.315 -m 140 \
  -d 0.264 --drag-model g7 --max-range 1000 \
  --enable-spin-drift --twist-rate 8

# With Coriolis
time ballistics trajectory -v 2710 -b 0.315 -m 140 \
  -d 0.264 --drag-model g7 --max-range 1000 \
  --enable-coriolis --latitude 45

# With Magnus
time ballistics trajectory -v 2710 -b 0.315 -m 140 \
  -d 0.264 --drag-model g7 --max-range 1000 \
  --enable-magnus --twist-rate 8
```

3. **Compare allocators.** Build the engine with the default allocator, then with jemalloc, and run a 5,000-iteration Monte Carlo simulation. Measure the wall-clock time difference.

```
# Default allocator
cargo build --release
time ./target/release/ballistics monte-carlo \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 600 \
  --num-sims 5000 --velocity-std 15

# jemalloc
cargo build --release --features jemalloc
time ./target/release/ballistics monte-carlo \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 600 \
  --num-sims 5000 --velocity-std 15
```

4. **Profile with a flame graph.** Install `cargo flamegraph` and generate a flame graph for a 1,000-yard .308 Win trajectory with all physics effects enabled. What percentage of time is spent in `compute_derivatives`?

```
cargo install flamegraph
cargo flamegraph -- trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 --drag-model g7 \
  --auto-zero 100 --max-range 1000 \
  --enable-spin-drift --twist-rate 10 \
  --enable-coriolis --latitude 45 \
  --enable-magnus --enable-pitch-damping
```

5. **Measure the zeroing cost.** Time a trajectory *without* zeroing (using a fixed muzzle angle) versus the same trajectory with `--auto-zero`. What is the ratio?

```
# Without zeroing (fixed angle)
time ballistics trajectory -v 2700 -b 0.462 -m 168 \
  -d 0.308 --drag-model g7 --max-range 1000

# With auto-zeroing at 100 yards
time ballistics trajectory -v 2700 -b 0.462 -m 168 \
  -d 0.308 --drag-model g7 --max-range 1000 \
  --auto-zero 100
```

## What's Next

This chapter concludes Part IX, "Under the Hood." You now understand the engine's architecture (Chapter 26), its language bindings (Chapter 27), and its performance characteristics. Armed with this knowledge, you can make informed decisions about integration method selection, physics-effect tradeoffs, and deployment targets.

The appendices that follow provide reference material: a complete CLI reference (Chapter A), tables of physical constants (Chapter B), drag tables (Chapter C), and a glossary of ballistic terms (Chapter E).

# Part X

# Reticles & Online Solvers

# Chapter 29

# Reticles, Holds, and BDC Solvers

Every chapter so far has assumed you dial. You read a come-up, you turn a turret, you break the shot with the crosshair on the target. That is the precision-rifle way, and it is the right way when you have the time.

You often do not. A running hog at 240 yards, a competition stage with six targets and ninety seconds, a mountain shot where taking your hand off the rifle costs you the position—in all of those you *hold*, and what you hold on is a mark etched into a piece of glass. This chapter is about the marks: how to describe a reticle to the engine, how to place a firing solution inside one, and how to work the relationship backwards from marks to ranges, from a ladder to a magnification, or from a whole stage of targets to the one zero that makes them all holdable.

Everything here arrived in v0.31.0.

## 29.1 Describing a Reticle

No reticle concept existed anywhere in the engine before v0.31.0, so the first thing the release had to do was define one. A `ReticleDescription` is a name, a focal plane, a reference magnification, and a list of marks—each mark an angular offset down and right of center, with a kind and an optional label. That single schema is shared by the CLI, the saved-profile format, the browser terminal, and the solve-json protocol, which is what keeps them from disagreeing about what a reticle is.

You can write one by hand, but for the common layouts the `reticle generate` verb builds it for you:

Listing 29.1: A one-mil hash cross out to six mils

```
ballistics reticle generate mil-grid --spacing 1 --extent 6
```

Listing 29.2: The generated description

```
Reticle: mil-cross 1/6
Focal plane: FFP  Marks: 25

   # Kind    Down(mil)  Right(mil)  Label
---- ------- ---------- ----------- --------------------
   0 center       0.000       0.000  -
   1 hash         1.000       0.000  -
   2 hash        -1.000       0.000  -
   3 hash         0.000       1.000  -
   4 hash         0.000      -1.000  -
   5 hash         2.000       0.000  -
   6 hash        -2.000       0.000  -
```

Three generators are available:

- `mil-grid` — a plain mil-hash *cross*: marks along both stadia, at `--spacing` intervals out to `--extent`.
- `tree` — a widening holdover tree, where row $N$ carries windage dots at $\pm 1 \ldots N$ steps. Controlled by `--rows`, `--row-spacing`, and `--spread-step`.
- `bdc` — a bullet-drop-compensating ladder built from drops you have *already solved*.

Listing 29.3: A BDC ladder from a solved .308 come-up table

```
ballistics reticle generate bdc \
  --drop 300:0.788 --drop 400:1.710 \
  --drop 500:2.763 --drop 600:3.960 \
  --name "308 ladder" --focal-plane sfp --reference-mag 10
```

Listing 29.4: The ladder, labeled by range

```
Reticle: 308 ladder
Focal plane: SFP  Reference magnification: 10.00x  Marks: 5

   # Kind    Down(mil)  Right(mil)  Label
---- ------- ---------- ----------- --------------------
   0 center       0.000       0.000  -
   1 hash         0.788       0.000  300 yd
   2 hash         1.710       0.000  400 yd
   3 hash         2.763       0.000  500 yd
   4 hash         3.960       0.000  600 yd
```

Note what the `bdc` generator does *not* do: it runs no solve of its own. You produce the drops with `come-ups` or `trajectory` first and hand them over, so the ladder's provenance—which load, which atmosphere, which zero—stays yours and stays visible.

**`-o json` Round-Trips**

`reticle generate -o json` emits exactly what `--reticle-json` consumes, so a generated description can be saved, edited, version-controlled, and fed back in. It is also what `profile save --reticle-json` stores.

## 29.2 Placing a Solution in the Glass

Given a reticle and a firing solution, `reticle hold` places the solution among the marks and names the nearest one:

Listing 29.5: Which mark do I hold?

```
ballistics reticle hold \
  --reticle-json mil-grid.json \
  --drop-mil 3.4 --wind-mil 0.8
```

Listing 29.6: The nearest mark, and how far off it is

```
Reticle Hold Point
==================

Reticle:          mil-cross 1/6
Focal plane:      FFP
Magnification:    10.00x (FFP: subtensions are magnification-independent)

Hold down:     3.400 mil
Hold right:    0.800 mil

Nearest mark:     #9 hash (-)
  at (down 3.000, right 0.000) mil true
  distance from hold: 0.894 mil
```

That last line is the one that matters. A reticle is a discrete set of places you can put the crosshair, and the honest question is not "which mark is closest" but "how much am I giving away by using it". Nearly nine tenths of a mil at, say, 600 yards is about nineteen inches—enough to miss with.

Instead of supplying the solution directly you can let the engine solve it: `--range` runs the trajectory and uses the resulting hold. That form is native-CLI only; the browser terminal takes `--drop-mil` and `--wind-mil`.

### 29.2.1 First and Second Focal Plane

A first-focal-plane reticle is etched in front of the erector, so it grows and shrinks with the image. A mark that subtends one mil subtends one mil at every magnification, which is why FFP is the long-range default and why the hold in the example above needed no magnification at all.

A second-focal-plane reticle is etched behind the erector. It stays the same apparent size while the image changes, so its marks only mean what the label says at one magnification—the *reference magnification* the optic was calibrated at. At any other setting:

$$\text{true subtension} = \text{nominal} \times \frac{\text{reference magnification}}{\text{magnification}} \tag{29.1}$$

A 2-mil mark on a reticle calibrated at 10× covers 4 mil of target at 5×. The engine applies this by published optics-manual math, and the consequences are visible in the output:

Listing 29.7: The same BDC ladder, used at half its calibration magnification

```
ballistics reticle hold \
  --reticle-json bdc.json \
  --drop-mil 2.763 --mag 5
```

Listing 29.8: An SFP hold at 5x on a 10x-calibrated reticle

```
Reticle Hold Point
==================

Reticle:          bdc 4 marks
Focal plane:      SFP
Reference mag:    10.00x
Magnification:    5.00x
Subtension scale: 2.0000x (marks read 2.0000x their etched value)

Hold down:     2.763 mil
Hold right:    0.000 mil

Nearest mark:     #2 hash (400 yd)
  at (down 3.420, right 0.000) mil true
  distance from hold: 0.657 mil
```

Read that carefully, because it is the whole point of the feature. At 10× this hold would land exactly on the mark labeled "500 yd"—it is the 500-yard drop, after all. At 5× the marks have doubled their angular value, the ladder no longer matches the load at all, and the nearest mark is the one labeled *400*, two thirds of a mil away. A shooter who dialed the magnification down for a wider field of view and kept holding on the labels would be badly wrong and would have no indication of it.

**The Hold Stays True; Only the Marks Move**

The hold point is a property of the trajectory, so it is always expressed in true angular units. Only the *marks* are rescaled, and the nearest-mark distance is measured after that rescaling. Non-physical magnifications are rejected on both planes, with typed errors throughout.

**Off the Reticle**

An `off_reticle` flag fires when the hold runs outside the marks' bounding box grown by 20% of its span per axis. It is not an error—you may legitimately be holding into empty glass—but it is the engine telling you that this reticle does not cover this shot.

### 29.2.2 Reticles on Profiles and in Solutions

A reticle is a property of the rifle, so it can live on the profile:

Listing 29.9: Storing a reticle with a rifle

```
ballistics profile save "prs-65cm" \
  --velocity 2710 --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --sight-height 2.0 \
  --reticle-json my-reticle.json
```

`--clear-reticle` removes it, and an unrelated re-save carries it forward untouched—the same forward-compatibility pattern the BC segments and the DSF table already follow. The solve-json protocol gained a matching optional `reticle` request block, whose response adds a `reticle_hold` object and only then, so every request that does not use it produces a byte-identical response.

**Deliberate Omissions**

Three things are permanently excluded unless licensed, and it is worth knowing they are absent by decision rather than by oversight. There are no Horus/TREMOR-family grid layouts—`mil-grid` builds a plain *cross*, not a filled two-dimensional grid. There is no time-of-flight wind-dot calibration; wind enters only as an angular deflection the caller has already solved. And there is no vendor reticle catalog: you describe your optic, the engine does not recognise it.

## 29.3 Working Backwards: The Inverse Solvers

Three commands run the reticle relationship in reverse. All three are read-only over an existing load—no new physics, no feature flags—and all three share *one* drop-versus-range helper: a single solved, finely sampled angular-drop curve with a forward lookup and a monotone bisection inverse. That sharing is deliberate. It means none of them can disagree with each other, or with `reticle hold --range`, about what the drop at 500 yards is.

### 29.3.1 `mark-to-range`: What Is This Mark Good For?

You have a reticle full of hashes and a load. Which range does each hash correspond to?

Listing 29.10: Mapping marks to ranges

```
ballistics mark-to-range \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 200 --sight-height 1.5 \
  --mark 1 --mark 2 --mark 3 --mark 4
```

Listing 29.11: One row per mark

```
Mark-to-Range
=============

Reticle:      supplied marks
Focal plane:  FFP
Zero:         200 yd

Mark(mil)  True(mil)  Range(yd)  Time(s)  Vel(fps)  Energy(ft-lb)
---------  ---------  ----------  -------  ----------  ------------
     1.00      1.000         324    0.409        2097          1641
     2.00      2.000         429    0.565        1921          1376
     3.00      3.000         521    0.715        1774          1174
     4.00      4.000         603    0.859        1650          1015
```

This is the Nightforce / Nikon Spot On / Swarovski / TRACT calculation, and it converts nominal marks to true angular values for the focal plane and magnification in use before mapping anything. Marks that do not map to a range are *reported* rather than silently dropped: `not_a_holdover` for a mark at or above center, and `beyond_max_range` for one deeper than the load actually reaches, carrying the deepest hold it does reach. The table always has one row per mark.

Take `--mark` flags for individual subtensions, or `--reticle-json` for a whole description.

### 29.3.2 `bdc-match`: What Magnification Makes This Fit?

The Zeiss Rapid-Z idea inverts Equation (29.1): rather than accepting whatever the ladder means at the reference magnification, you *choose* a magnification that makes the ladder match your load. Supply the mark-and-range pairs you want to hit and let the engine solve for it:

Listing 29.12: Fitting an SFP BDC ladder to a load

```
ballistics bdc-match \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 200 --sight-height 1.5 \
  --reference-mag 12 \
  --mark-range 1:400 --mark-range 2:500 --mark-range 3:580
```

Listing 29.13: A fit, with its residuals

```
BDC Magnification Match
=======================

Reference magnification: 12.00x
Fitted magnification:    9.150x
Subtension scale at fit: 1.3115x
Zero:                    200 yd

Mark(mil)  Range(yd)  Apparent(mil)  Drop(mil)  Residual(mil)
---------  ----------  -------------  ---------  -------------
     1.00         400          1.312      1.710         -0.399
     2.00         500          2.623      2.763         -0.140
     3.00         580          3.935      3.708          0.226

Worst residual: 0.399 mil   RMS: 0.277 mil

WARNING: the worst residual exceeds 0.200 mil — this reticle does not fit this load
    well at ANY magnification. Treat the fitted value as the least-bad compromise,
    not a BDC calibration.
```

The fit itself is exact rather than iterative. Substituting $u =$ reference mag $\div$ mag makes the residual linear in $u$, so the answer is the closed-form least-squares slope through the origin—no starting guess, no convergence to worry about, the same answer every time.

The warning is as important as the number. A magnification exists that minimises the residual for *any* ladder and *any* load; whether that minimum is small enough to shoot is a separate question, and `--residual-warn` (default 0.2 mil) is where the command draws the line. A first-focal-plane reticle is

rejected outright with an explanation rather than a meaningless number: FFP subtensions do not depend on magnification, so there is nothing to solve.

### 29.3.3 `optimal-zero`: One Zero for a Whole Stage

The last of the three asks a question that only makes sense if you hold rather than dial. Given a list of targets you must engage without touching the turret, which single zero minimises the *largest* hold any of them requires?

Listing 29.14: Min-max zero for a three-target stage

```
ballistics optimal-zero \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --sight-height 1.5 \
  --target 300 --target 500 --target 700 --vital 8
```

Listing 29.15: The answer, and an honest verdict

```
Optimal Zero (min-max hold)
===========================

Optimal zero:  525.5 yd
Largest hold:  2.266 mil

Range(yd)  Target(in)  Hold(mil)  Center-hold miss(in)  Fits
---------  ----------  ---------  --------------------  ----
      300         8.0     -2.266                -24.47  NO
      500         8.0     -0.290                 -5.23  NO
      700         8.0      2.266                 57.10  NO

At least one target needs a hold: no single zero keeps them all centered.
Center-hold miss is signed: positive = the bullet lands LOW of the aim point.
```

The symmetry in the hold column is the min-max solution showing its work: the optimizer has pushed the zero until the worst hold above center exactly equals the worst hold below it, because any further movement would make one of them larger. Nothing else can do better.

It also tells you plainly that a dead-center hold misses every one of these targets, which is the useful output here. The search is golden-section over a bracketed zero range with hard-coded constants, so the answer is reproducible bit for bit, and targets are sorted internally so the order you type them in cannot change it. `--target` takes `RANGE[:HEIGHT]`, two to sixteen of them, with `--vital` supplying the default height.

## 29.4 Hold Corridors Across Wind Scenarios

Here is a situation the rest of this book handles badly. You are looking at a 600-yard target. The wind might be doing three things: it might be the lull you can see in the near flags, it might be the steady eight you have been holding for, or it might be the gust that just moved the grass at the target. You do not have a probability distribution over those. You have three concrete calls, and you have to pick one hold.

A nominal trajectory hides the ambiguity—it just picks one. Monte Carlo demands a probability model you do not have and would be making up. The `hold-corridor` command takes the third path: give it a bounded set of *named* scenarios and it solves each one exactly once, then reports what they span.

Listing 29.16: Three named wind calls

```
{"version":1,"nominal":"steady",
 "scenarios":[
   {"name":"lull","segments":["3:90:1000"]},
   {"name":"steady","segments":["8:90:1000"]},
   {"name":"gust","segments":["14:80:1000"]}
 ]}
```

The segments are the same `SPEED:ANGLE:UNTIL[:VERTICAL]` tokens `--wind-segment` takes (Section 10.6.3), so a scenario can be as structured as any segmented wind you would otherwise type.

Listing 29.17: Solving the corridor

```
ballistics hold-corridor \
  --scenarios scenarios.json \
  --ranges 400,600,800 --target rect:18x30 \
  --velocity 2700 --bc 0.462 --mass 168 --diameter 0.308 \
  --zero-distance 200 --sight-height 1.5
```

Listing 29.18: The corridor at two ranges

```
Robust Hold Corridor
====================

Scenarios (3): gust, lull, steady
Nominal:        steady
Target:         rectangle 18.0 x 30.0 in (per-axis / L-inf metric)

Holds are milliradians: elevation positive = hold UP, windage positive = hold RIGHT.
```

```
The corridor is the span of the scenarios you supplied. It is NOT a probability
    interval.

--- 400 yd ---
  Scenario              Elev(mil)  Wind(mil)
  gust                      1.712     -1.288
  lull                      1.710     -0.280
  steady                    1.710     -0.746
  Corridor                  1.710     -1.288   (min)
                            1.712     -0.280   (max)
  Minimax hold              1.711     -0.784
  Worst case from it        0.504 mil (7.26 in), scenario 'gust'
  Holding the nominal       0.542 mil (7.80 in)
  Fits target           yes — one hold covers every scenario

--- 600 yd ---
  Scenario              Elev(mil)  Wind(mil)
  gust                      3.966     -2.102
  lull                      3.960     -0.457
  steady                    3.960     -1.217
  Corridor                  3.960     -2.102   (min)
                            3.966     -0.457   (max)
  Minimax hold              3.963     -1.279
  Worst case from it        0.823 mil (17.77 in), scenario 'lull'
  Holding the nominal       0.884 mil (19.10 in)
  Fits target           NO — no single hold covers every scenario here
```

At 400 yards the ambiguity is survivable: one hold keeps all three scenarios on an 18-by-30 silhouette, and the minimax hold beats the nominal by a small margin. At 600 it is not. No single hold covers all three, and the worst case is nearly eighteen inches. That is the number worth knowing before you break the shot—not the drift under your best guess, but the cost of your guess being wrong.

### 29.4.1 Two Metrics, Because the Shapes Differ

`--target` accepts two forms and they are solved differently, because a rectangle and a circle genuinely are different problems.

- `rect:WxH` (or no target at all) treats the axes independently—the $L_\infty$ metric—so the minimax hold is simply the per-axis midpoint of the extremes.
- `circle:D` minimises the largest *Euclidean* distance, which makes the hold the center of the minimum enclosing circle of the scenario holds. The engine computes it exhaustively over the candidate circles: exact and deterministic, where the usual randomized construction is neither.

Boundary contact counts as a fit.

### 29.4.2 What the Command Refuses to Do

Several design decisions here are worth stating because they are the reason to trust the output.

**The zero is solved once, in calm air, and reused by every scenario.** A rifle has one zero. Re-zeroing per scenario would collapse the very corridor the command exists to show.

**Reordering the scenarios cannot change the answer.** They are sorted by name internally before anything is solved.

**Caps are enforced before any solving happens**: eight scenarios and sixty-four ranges maximum, alongside malformed segments, unknown `nominal` names, duplicate names, duplicate ranges, and unsupported versions—all structured, typed errors raised before a single trajectory runs.

**No Probabilities, Anywhere**

This is an explicit non-goal and the output says so on its own. Nothing is weighted, nothing is interpolated between scenarios, and the finite set is never folded into a standard deviation. **A three-scenario corridor is not a confidence interval.** It is the span of three specific calls you chose to consider. Statistical dispersion remains `monte-carlo --wez`'s job (Section 6.9), and the two answer different questions: the WEZ tells you how often you hit given a distribution, the corridor tells you what it costs if your call is the wrong one of several.

`-o json` emits a versioned `RobustHoldReportV1`.

## 29.5 Exercises

1. **Build and read a reticle.** Generate a one-mil cross out to eight mils, save it as JSON, and place a 4.2-mil drop with 1.1 mil of wind in it. How far is the nearest mark from your hold, and how many inches is that at 700 yards?

```
ballistics reticle generate mil-grid --spacing 1 --extent 8 -o json > grid.json

ballistics reticle hold --reticle-json grid.json \
  --drop-mil 4.2 --wind-mil 1.1
```

2. **Watch an SFP reticle lie to you.** Build a BDC ladder at a reference magnification of 10, then take the same hold at 10×, 6×, and 4×. At what magnification does the nearest mark change identity?

3. **Map your own glass.** Run `mark-to-range` for your rifle against marks at 1 through 6 mils. Compare the answers to the ranges you have actually been holding those marks for.

```
ballistics mark-to-range \
  --velocity 2710 --bc 0.610 --mass 140 --diameter 0.264 \
  --drag-model g7 --zero-distance 100 --sight-height 2.0 \
  --mark 1 --mark 2 --mark 3 --mark 4 --mark 5 --mark 6
```

4. **Find the stage zero.** Take four targets at 200, 350, 500 and 650 yards with a 10-inch vital zone and solve the min-max zero. Does one zero keep them all inside? Now remove the 650 and try again.
5. **Price your wind call.** Write a three-scenario corridor for a range you actually shoot—your lull, your steady, and your gust—and find the distance at which one hold stops covering all three against a 10-inch plate.

## What's Next

The reticle is one of two places a firing solution can land; the other is a turret. Chapter 30 turns to a different kind of question entirely—not where to put the crosshair for a load you already have, but what load to build in the first place.

# Chapter 30

# The Online Reverse Solvers

Every command in this book so far answers the same shape of question. You have a load—a bullet, a charge, a barrel—and you want to know what it does. The inputs are the cartridge and the outputs are the trajectory.

Load development runs the other way. You know what you *want*: a particular muzzle velocity, a bullet that will stabilise, an overall length that will feed. What you do not know is the powder, the twist, or the seating depth that gets you there. That is a search over a body of data—powder burn rates, published load tables, cartridge dimensions—rather than a physics solve, and it is not something a self-contained binary can carry.

Version 0.32.0 added four commands that ask a hosted service instead. This chapter covers them, and the `login` and `logout` verbs that authenticate them.

**Everything Else Stays Offline**

These four subcommands—and only these four—call an online service. Every local trajectory and analysis command in this book is unchanged and needs no token and no network access. If you never run `recommend-*` or `calibrate-bc`, the engine behaves exactly as it did before v0.32.0.

## 30.1 Signing In

The reverse solvers require an account token. Create one at `https://ballisticsinsight.com/account` and hand it to `login`:

Listing 30.1: Saving a CLI access token

```
# Prompts for the token if --token is omitted
```

```
ballistics login

ballistics login --token "<your token>"
```

The token is written to `~/.ballistics/credentials.toml` with mode `0600`. `logout` removes it:

Listing 30.2: Clearing the saved token

```
ballistics logout
```

```
Removed saved CLI token.
```

**The Environment Variable Wins**

Token resolution has a documented precedence: the `BALLISTICS_API_TOKEN` environment variable takes priority over the on-disk credentials file. That is what you want in CI, in a container, or on a shared machine—set the variable for the process and leave nothing behind on disk.

**When the Service Says No**

Two failures are worth recognising. A `401` means the request carried no valid token, and the CLI answers with a hint rather than a status code:

```
Unauthorized - run `ballistics login` (create a token at https://
    ballisticsinsight.com/account).
```

A `402` means the token is valid but the endpoint needs an active subscription:

```
This endpoint needs an active Ballistics Insight subscription: https://
    ballisticsinsight.com/subscribe
```

Anything else—a genuine network failure, a server error—falls back to the underlying message, prefixed `Online request failed:`. Every one of these exits non-zero, so a script can tell.

Each solver targets `https://api.ballistics.7.62x51mm.sh` by default. `--api-url` overrides it, which is what you want when pointing at a staging deployment or a local instance.

## 30.2 `recommend-twist`: What Barrel Do I Need?

Stability is the first question a bullet asks of a barrel, and Chapter 14 covers what happens when the answer is wrong. Rather than solving Miller's formula backwards by hand and guessing at margins, hand the bullet's geometry over:

Listing 30.3: Recommending a twist rate for a 168gr SMK

```
ballistics recommend-twist \
  --caliber 0.308 --weight 168 \
  --overall-length 1.24 --nose-length 0.65
```

Caliber, overall length and nose length are inches; weight is grains. `--muzzle-velocity` is optional—supplying it lets the service account for the velocity dependence of the stability factor.

**Nose Length Is the Input People Skip**

Two bullets of the same weight and caliber can need meaningfully different twists if one has a long secant ogive and the other does not. Length forward of the bearing surface is destabilising in a way that length behind it is not. Measure it, or get it from the manufacturer's drawing—do not estimate.

## 30.3 `recommend-powder`: Getting to a Velocity

Given a cartridge, a bullet weight, and the velocity you are trying to reach, this returns powders and charges that will get you there:

Listing 30.4: Powders for a 2700 fps .308 load

```
ballistics recommend-powder \
  --cartridge ".308 Winchester" \
  --bullet-weight 168 \
  --desired-velocity 2700 \
  --barrel-length 24 \
  --velocity-tolerance 50
```

`--barrel-length` and `--velocity-tolerance` are optional. The tolerance is worth setting: a wide window returns more candidates, a narrow one returns the powders that actually land where you asked.

**SAFETY: A Recommendation Is Not Load Data**

Charge weights returned by this command are a starting point for consulting a reloading manual, not a substitute for one. **Work up from a reduced starting charge, incrementally, watching for pressure signs at every step.** Component substitution—a different primer, a different case, a different lot of the same powder—changes pressure, and no service can know what is in your chamber. Never exceed the published maximum for your components, and always verify computational predictions against real-world chronograph data before relying on them.

## 30.4 `recommend-col`: Seating Depth

Cartridge overall length is a compromise between the magazine, the throat, and the pressure curve. This returns a recommended COL for a cartridge, optionally narrowed by bullet weight and type:

Listing 30.5: A recommended overall length

```
ballistics recommend-col \
  --cartridge "6.5 Creedmoor" \
  --bullet-weight 140 \
  --bullet-type "match"
```

**Your Chamber Has the Final Say**

A recommended COL is a published or derived figure for the cartridge, not a measurement of your rifle. Throat length varies between chambers and moves with barrel wear. Measure your own distance to the lands, confirm the round feeds from your magazine, and treat the recommendation as the starting point for that measurement rather than a replacement for it.

## 30.5 `calibrate-bc`: A BC From Dimensions

The last of the four estimates a bullet's ballistic coefficient from its physical dimensions:

Listing 30.6: Estimating a BC from geometry

```
ballistics calibrate-bc \
  --diameter 0.308 --length 1.24 --weight 168 \
  --bullet-type "match"
```

This is the right tool for a bullet with no published BC at all—a custom turning, an obscure import, something out of production. It is emphatically *not* a substitute for the truing workflow of Chapter 7.

A geometric estimate tells you roughly what a bullet of that shape should do; a fit against your own dope tells you what *yours* actually does, in your barrel, at your velocity. Use the estimate to get on paper and the fit to stay there.

## 30.6 Importing a Third-Party Reticle

One more thing arrived in v0.32.0, and although it is not an online solver it belongs beside them: the engine can now read a reticle authored somewhere else. `reticle import` converts a Ventum reticle specification into the `ReticleDescription` schema of Section 29.1:

Listing 30.7: Converting a Ventum reticle

```
# Preview the marks
ballistics reticle import my-reticle.json

# Emit the interchange form for the rest of the toolchain
ballistics reticle import my-reticle.json -o json > my-reticle.engine.json

ballistics reticle hold \
  --reticle-json my-reticle.engine.json \
  --drop-mil 3.4 --wind-mil 0.8
```

The output of `-o json` is exactly what `--reticle-json` consumes on `reticle hold` and `profile save`, so an imported reticle is a first-class one from that point on.

The reader is deliberately permissive about the input and strict about the output. Cosmetic keys in the Ventum file—colours, stroke widths, element sizes, tube diameter, manufacturer notes—are ignored rather than rejected, and unknown element *types* are skipped, because a drawing format carries a great deal that a hold solver does not need. What it will not do is emit something unusable: the converted description is validated before it is written, so a decoration-only spec with no hold-bearing marks fails as `NoMarks` rather than producing an empty reticle that silently never matches a hold.

## 30.7 Exercises

1. **Check your token plumbing.** Run `ballistics logout`, then a `recommend-twist` call, and read the error. Now set `BALLISTICS_API_TOKEN` in your shell and run it again. Which one does the engine use if both a file and the variable are present?

2. **Twist for a bullet you own.** Measure the overall length and nose length of a bullet in your safe and ask for a twist recommendation. Compare the answer to your barrel, then run `ballistics stability` for the same bullet at your velocity and see whether the two agree about the margin.

```
ballistics recommend-twist \
  --caliber 0.264 --weight 140 \
  --overall-length 1.375 --nose-length 0.72
```

3. **Estimate, then true.** Run `calibrate-bc` for a bullet whose published BC you already know. How close is the geometric estimate? Then fit the same bullet against a dope card with `estimate-bc` (Section 7.2) and compare all three numbers.
4. **Point somewhere else.** Every solver takes `--api-url`. Run one against a deliberately wrong host and confirm that the failure is a clean non-zero exit with a readable message, not a hang.

## What's Next

That is the whole command surface. Chapter A is the flag-by-flag reference for all of it, Chapter B lists the constants the solver runs on, and Chapter E defines the vocabulary. Everything from here is lookup.

# Appendix A

# CLI Reference

This appendix provides a comprehensive reference for every command, flag, and option available in BALLISTICS-ENGINE. Where flags accept unit-dependent values, the imperial interpretation is listed first, followed by the metric equivalent in parentheses. Default values shown assume `--units imperial`.

The general invocation pattern is:

```
ballistics [GLOBAL OPTIONS] <COMMAND> [COMMAND OPTIONS]
```

## A.1 Global Options

Global options appear *before* the subcommand name and apply to all commands uniformly.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--units`, `-u` | `imperial` \| `metric` | `imperial` | Unit system for all input and output values. Imperial uses fps, grains, yards, inches, °F, inHg. Metric uses m/s, grams, meters, mm, °C, hPa. |
| `--help`, `-h` | — | — | Print help information for the tool or any subcommand. |
| `--version`, `-V` | — | — | Print the BALLISTICS-ENGINE version string and exit. |

**Metric shortcut**

When working in metric, place `-u metric` immediately after `ballistics` so that every subsequent value—velocity, mass, distance, atmospheric parameters—is interpreted in SI units.

## A.2 trajectory — Single Trajectory Calculation

The `trajectory` command is the primary workhorse of BALLISTICS-ENGINE. It computes a full ballistic trajectory from muzzle to the specified maximum range (or ground impact), accounting for drag, gravity, atmospheric density, and—if enabled—spin drift, Coriolis deflection, Magnus effect, precession/nutation, and powder temperature sensitivity.

Parameters may be supplied directly on the command line, loaded from a saved profile (`--saved-profile`), read from a CSV profile file (`--profile`), or any combination thereof. Command-line flags always override profile values.

### A.2.1 Core Ballistic Parameters

| Flag | Type | Default | Description |
|---|---|---|---|
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). Range: 0–6000. |
| `--velocity-adjustment` | float | 0.0 | Added to base velocity for truing from chronograph data. Accepts negative values. |
| `-a, --angle` | float | 0.0 | Launch angle in degrees. |
| `-b, --bc` | float | — | Ballistic coefficient. Range: 0.001–2.0. |
| `--bc-adjustment` | float | 1.0 | BC multiplier for truing (e.g., 0.85 = 85% of stated BC). |
| `-m, --mass` | float | — | Projectile mass (grains / grams). Range: 0.1–2000. |
| `-d, --diameter` | float | — | Projectile diameter (inches / mm). Range: 0.01–60.0. |
| `--drag-model` | `g1` \| `g7` | `g1` | Standard drag function. |
| `--max-range` | float | 1000 | Maximum range (yards / meters). |
| `--time-step` | float | 0.001 | Integration time step in seconds. Range: 0.00001–0.1. |

### A.2.2 Profile and Location Loading

| Flag | Type | Description |
|---|---|---|
| `--profile FILE` | path | Load parameters from a CSV profile file. |
| `--profile-row NAME` | string | Select a specific row from the profile CSV (matches the first column). |
| `--saved-profile NAME` | string | Load a saved profile by name (from `~/.ballistics/profiles/`). |
| `--location FILE` | path | Load environmental data from a location CSV file. |
| `--site NAME` | string | Select a site from the location CSV (matches the first column). |

### A.2.3 Wind and Atmosphere

| Flag | Type | Default | Description |
|---|---|---|---|
| `--wind-speed` | float | 0.0 | Wind speed (mph / m/s). |
| `--wind-direction` | float or clock | 0.0 | Wind direction, wind-FROM (0 = headwind). Accepts degrees or a marked clock position: `3oc`, `10h30`, `10:30`. Bare numbers are degrees. |
| `--wind-ref` | `shooter` \| `compass` | `shooter` | Frame for every wind direction the run consumes. `compass` treats them as absolute bearings and requires `--shot-direction`. |
| `--temperature` | float | 59.0 | Temperature (°F / °C). Range: −100 to 200. |
| `--pressure` | float | 29.92 | Barometric pressure (inHg / hPa). Range: 15–1200. |
| `--humidity` | float | 50.0 | Relative humidity, 0–100%. |
| `--altitude` | float | 0.0 | Altitude above sea level (feet / meters). |
| `--pressure-type` | `absolute` \| `qnh` | `absolute` | Whether `--pressure` is station pressure or a sea-level-corrected altimeter setting. `qnh` is reduced against `--altitude`. |
| `--density-altitude` | float | — | Carry conditions as one density altitude (feet / meters). Supersedes `--altitude` and `--pressure`; an explicit `--temperature` still wins. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--bc-reference` | `icao` \| `army-standard-metro` | `icao` | Which reference atmosphere `--bc` was published against. |

### A.2.4 Zeroing and Geometry

| Flag | Type | Default | Description |
|---|---|---|---|
| `--auto-zero` | float | — | Auto-zero to target distance (overrides `--angle`). Value is in yards (imperial) or meters (metric). |
| `--sight-height` | float | — | Sight height above bore centerline (inches / mm). |
| `--bore-height`, `--muzzle-height` | float | 60 | Bore height above ground (inches / mm). Default: 60″ / 1500 mm (= 5 ft / 1.5 m). **Feet/metres before v0.26.0.** |
| `--shooting-angle` | float | 0.0 | Uphill/downhill angle in degrees. Positive = uphill. |
| `--sight-offset` | float | 0.0 | Lateral sight offset (inches / mm); positive = sight RIGHT of bore. A zero solve applies the windage-zero convergence. |
| `--zero-poi-up` | float | 0.0 | Deliberate vertical POI offset at the zero range (inches / cm); positive = impacts HIGH. |
| `--zero-poi-right` | float | 0.0 | Deliberate horizontal POI offset at the zero range (inches / cm); positive = impacts RIGHT. |
| `--zero-set` `NAME` | string | — | Select a named zero set from a saved profile or a `--profile` CSV row. |
| `--drops-reference` | `los` \| `target` | `los` | Reference plane for the sampled drop column. `target` divides by cos(shooting angle) and relabels the column. |
| `--ignore-ground-impact` | bool | false | Disable ground impact detection; trajectory continues to max range. |

### A.2.5 Advanced Physics Flags

| Flag | Type | Default | Description |
|---|---|---|---|
| `--enable-magnus` | bool | false | Enable Magnus effect. Requires `--twist-rate`. |
| `--enable-coriolis` | bool | false | Enable Coriolis effect. Requires `--latitude`. |
| `--enable-spin-drift` | bool | false | Enhanced spin drift calculations. |
| `--enable-wind-shear` | bool | false | Altitude-dependent wind modeling. |
| `--enable-pitch-damping` | bool | false | Transonic stability analysis via pitch damping. |
| `--enable-precession` | bool | false | Precession and nutation angular motion modeling. |
| `--use-bc-segments` | bool | false | Velocity-based BC segmentation. |
| `--use-cluster-bc` | bool | false | Cluster-based BC degradation for improved accuracy. |
| `--sample-trajectory` | bool | false | Enable trajectory sampling at regular intervals. |
| `--sample-interval` | float | 10.0 | Sampling interval in meters. |
| `--use-euler` | bool | false | Use Euler integration instead of RK45 adaptive. |
| `--use-rk4-fixed` | bool | false | Use fixed-step RK4 instead of adaptive RK45. |

### A.2.6 Bullet and Barrel Parameters

| Flag | Type | Default | Description |
|---|---|---|---|
| `--bullet-length` | float | — | Bullet length in inches (used for BC table lookup; estimated from diameter if omitted). |
| `--twist-rate` | float | — | Barrel twist rate (inches per turn, e.g., 10 for 1:10). |
| `--twist-right` | bool | true | Right-hand twist. Set to `false` for left-hand twist barrels. |
| `--latitude` | float | — | Latitude in degrees (−90 to 90). Required for Coriolis. |
| `--longitude` | float | — | Longitude in degrees (−180 to 180). Used for weather zones. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--shot-direction` | float | — | Shot azimuth in degrees (0=North, 90=East). |

### A.2.7 Zero-Day Conditions

These flags describe the day the rifle was *sighted in*, as distinct from the day of the shot. They apply only to the `--auto-zero` solve; the trajectory itself still runs under the shot-day values. Each is independently optional, and any flag left out falls back to its shot-day counterpart—so supplying none of them reproduces single-condition behavior exactly.

| Flag | Type | Description |
|---|---|---|
| `--zero-velocity` | float | Zero-day muzzle velocity (fps / m/s). Overrides both powder models for the zero solve. |
| `--zero-temperature` | float | Zero-day air temperature (°F / °C). |
| `--zero-pressure` | float | Zero-day barometric pressure (inHg / hPa). |
| `--zero-humidity` | float | Zero-day relative humidity, 0–100%. |
| `--zero-altitude` | float | Zero-day altitude (feet / meters). |
| `--zero-pressure-type` | enum | Whether `--zero-pressure` is absolute or QNH. Defaults to the shot-day `--pressure-type`. |
| `--zero-density-altitude` | float | Zero-day density altitude (feet / meters). Supersedes `--zero-altitude` and `--zero-pressure` for the zero solve only. |
| `--zero-powder-temp` | float | Zero-day powder temperature (°F / °C) for interpolating `--powder-temp-curve` on the zero side. |

### A.2.8 Powder Temperature Sensitivity

Two models are available and they are mutually exclusive: a linear fps-per-degree sensitivity, and a measured temperature-to-velocity curve. Supplying `--powder-temp-curve` overrides `--powder-temp-sensitivity`.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--use-powder-sensitivity` | bool | false | Enable the linear powder temperature sensitivity correction. |
| `--powder-temp-sensitivity` | float | 1.0 | Velocity change per degree (fps/°F, m/s per °C). |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--powder-temp` | float | — | Powder temperature (°F / °C). With a curve, the temperature the curve is read at, defaulting to `--temperature`. With the linear model, the reference temperature the stated velocity was measured at, defaulting to 70 °F / 21 °C. |
| `--powder-temp-curve` | string | — | Measured curve as `"TEMP:VEL,...."`, e.g. `"40:2620,70:2700,100:2760"`. Clamped at the endpoints—no extrapolation. |

### A.2.9 BC Correction Tables

| Flag | Type | Description |
|---|---|---|
| `--bc-segment V1:V2:BC` | string | Manual velocity-keyed BC segment, repeatable. The given BC applies while the bullet's speed is in $[V_1, V_2)$; velocities follow `--units`. Implies `--use-bc-segments` and overrides `--bc-table`/`--bc-table-dir`. |
| `--print-bc-segments` | bool | Print the BC5D-generated ladder as ready-to-paste `--bc-segment` arguments. Requires `--bc-table-dir`. |
| `--bc-table FILE` | path | Path to a precomputed BC correction table file for offline ML-enhanced corrections. |
| `--bc-table-dir DIR` | path | Directory containing caliber-specific BC5D tables (e.g., `bc5d_308.bin`). |
| `--bc-table-auto` | bool | Auto-download BC5D tables when needed. *Requires* `online` *feature.* |
| `--bc-table-url` | string | Base URL for BC5D downloads. Default: `https://ballistics.tools/downloads/bc5d`. *Requires* `online` *feature.* |
| `--bc-table-refresh` | bool | Force re-download of BC5D tables even if cached. *Requires* `online` *feature.* |

### A.2.10 Output and PDF Options

| Flag | Type | Default | Description |
|---|---|---|---|
| `-o`, `--output` | enum | `table` | Output format: `table`, `json`, `csv`, or `pdf`. |
| `--full` | bool | false | Show all trajectory points instead of the standard summary. |
| `--output-file FILE` | path | — | Output file path (required for PDF format). |
| `--plot` | `braille` \| `ascii` | `braille` | Draw inline terminal charts of drop, drift, velocity and energy. Bare `--plot` uses braille dots. |
| `--adjustment-unit` | enum | `mil` | Angular unit for the PDF dope card's Drop/Wind/Lead columns and the `--target-speed` Ring column only: `mil`, `moa`, `smoa`, `iphy`, `clicks`. Warns when neither is in play. |
| `--drag-table FILE` | path | — | Custom drag deck: CSV `mach,cd` per line, or a vendor `.drg` file by extension. Replaces the G-model and BC. |
| `--cd-scale` | float | 1.0 | Whole-curve multiplier for a custom deck. Requires `--drag-table`. |
| `--with-drag-coefficient` | bool | false | Add the projectile's own $C_D$ to each point of `--full` JSON. This is the reference value scaled by SD/BC, not the G-model curve. |
| `--target-speed` | float | 0.0 | Moving-target speed (mph / m/s), bounded to 0–300. Adds the mover-ring column to the table, CSV and JSON, and drives the PDF dope card's Lead column. |
| `--powder` | string | — | Powder name for PDF metadata. |
| `--bullet-name` | string | — | Bullet name for PDF metadata. |
| `--location-name` | string | — | Location name for PDF header (overrides `--site` for display). |
| `--font-scale` | float | 1.0 | Font scale factor for PDF output. |
| `--font-preset` | `small` \| `medium` \| `large` | — | Font size preset. Overridden by `--font-scale` if both given. |
| `--bold-data` | bool | false | Use bold font for data cells in PDF output. |

### A.2.11 Examples

A basic .308 Win trajectory out to 1000 yards, zeroed at 100:

```
ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g1 --auto-zero 100 --sight-height 1.5 \
  --wind-speed 10 --wind-direction 90
```

A 6.5 Creedmoor trajectory using a saved profile with G7 drag, full output, and atmospheric conditions for a high-altitude range:

```
ballistics trajectory \
  --saved-profile "R700_65CM" \
  --altitude 5280 --temperature 45 --pressure 24.89 \
  --full -o table
```

Generating a PDF dope card with metadata:

```
ballistics trajectory \
  -v 2710 -b 0.610 -m 140 -d 0.264 --drag-model g7 \
  --auto-zero 100 --sight-height 1.5 \
  --bullet-name "140gr ELD-M" --powder "H4350" \
  --location-name "Quantico" \
  -o pdf --output-file dope_card.pdf --font-preset medium
```

## A.3 zero — Calculate Zero Angle

The `zero` command calculates the precise launch angle required to place the projectile at a specified height (typically zero drop) at the target distance. This is the angle your scope's mechanical zero effectively applies to the bore.

| Flag | Type | Default | Description |
|---|---|---|---|
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). Range: 0–6000. |
| `-b, --bc` | float | — | Ballistic coefficient. Range: 0.001–2.0. |
| `-m, --mass` | float | — | Projectile mass (grains / grams). Range: 0.1–2000. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `-d, --diameter` | float | — | Projectile diameter (inches / mm). Range: 0.01–60.0. |
| `--target-distance` | float | — | Zero distance (yards / meters). Required unless `--from-angle` is given. |
| `--from-angle` | float | — | Work backwards: recover the zero range(s) a stored bore angle implies, in degrees. Both the near and far crossing are reported. |
| `--target-height` | float | 0.0 | Target height offset (yards / meters). |
| `--sight-height` | float | — | Sight height above bore (inches / mm). |
| `--drag-model` | enum | g1 | Standard drag function. Added in v0.30.1; before that the zero always ran G1. |
| `--temperature` | float | 59.0 | Temperature (°F / °C). |
| `--pressure` | float | 29.92 | Barometric pressure (inHg / hPa). |
| `--pressure-type` | `absolute \| qnh` | `absolute` | How to read `--pressure`. |
| `--density-altitude` | float | — | Carry zeroing conditions as one density altitude. |
| `--humidity` | float | 50.0 | Relative humidity, 0–100%. |
| `--altitude` | float | 0.0 | Altitude (feet / meters). |
| `-o, --output` | enum | `table` | Output format: `table`, `json`, `csv`. With `--from-angle`, JSON gains `primary_crossing` (`"far"` or `"near"`). |

### A.3.1 Example

Calculate the zero angle for a .308 Win 168 gr SMK (BC `0.462`) at 100 yards with a 1.5-inch sight height:

```
ballistics zero \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --target-distance 100 --sight-height 1.5
```

## A.4 monte-carlo — Monte Carlo Simulation

The `monte-carlo` command runs stochastic trajectory simulations by varying velocity, launch angle, BC, and wind conditions according to user-specified standard deviations. It produces statistical distributions of impact points at a given target distance, enabling probability-of-hit analysis and sensitivity studies.

**Input units (0.22.0)**

`monte-carlo` follows `--units` like every other command: fps, grains, inches and yards in imperial, m/s, grams, mm and meters in metric. Earlier help text claimed the command took SI regardless of `--units`; that text was wrong and was corrected in v0.22.0.

| Flag | Type | Default | Description |
|---|---|---|---|
| `-v, --velocity` | float | — | Base velocity (fps / m/s). Required. |
| `-a, --angle` | float | 0.0 | Launch angle in degrees. |
| `-b, --bc` | float | — | Ballistic coefficient. Required. |
| `-m, --mass` | float | — | Projectile mass (grains / grams). Required. |
| `-d, --diameter` | float | — | Projectile diameter (inches / mm). Required. |
| `-n, --num-sims` | int | 1000 | Number of simulations to run. |
| `--velocity-std` | float | 1.0 | Velocity standard deviation (fps / m/s). |
| `--angle-std` | float | 0.1 | Angle standard deviation (degrees). |
| `--bc-std` | float | 0.01 | BC standard deviation. |
| `--wind-std` | float | 1.0 | Wind speed standard deviation (mph / m/s). |
| `--wind-direction-std` | float | 0.0 | Wind direction standard deviation (degrees). |
| `--wind-speed` | float | 0.0 | Base wind speed (mph / m/s). |
| `--wind-direction` | float | 0.0 | Base wind direction in degrees, wind-FROM (0 = headwind). |
| `--target-distance` | float | — | Target distance (yards / meters). |
| `--target-radius` | float | 0.3 m | Radius of the hit zone around the point of aim, in the target plane. A simulation landing inside it counts as a hit. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--wez` | bool | false | Run a Weapon Employment Zone sweep instead of a single-distance simulation. Requires `--target-size`. |
| `--target-size` | string | — | WEZ target: `WIDTHxHEIGHT` (inches / cm) for a rectangle centered on the point of aim, or a single number for a circular radius. |
| `--wind-call-error` | float | 0.0 | The shooter's error in *estimating* the wind (mph / m/s), as distinct from `--wind-std`'s physical gustiness. Composed in quadrature. WEZ only. |
| `--wez-start` | float | — | First range of the sweep. |
| `--wez-end` | float | — | Last range of the sweep. |
| `--wez-step` | float | — | Range increment of the sweep. |
| `-o, --output` | enum | `summary` | Output: `summary`, `full`, `statistics`. |

### A.4.1 Example

Run 5000 Monte Carlo simulations for a .308 Win 168 gr SMK at 800 meters:

```
# Metric: 2700 fps = 822.96 m/s,
# 168 gr = 10.89 g, 0.308 in = 7.823 mm
ballistics --units metric monte-carlo \
  -v 822.96 -b 0.462 -m 10.89 -d 7.823 \
  -n 5000 --target-distance 800 \
  --velocity-std 5.0 --bc-std 0.005 \
  --wind-speed 3.0 --wind-std 1.5 \
  -o statistics
```

## A.5 BC Estimation and Modeling Commands

### A.5.1 estimate-bc — Estimate BC from Trajectory Data

The `estimate-bc` command back-calculates a ballistic coefficient from observed drop and/or downrange velocity data. This is valuable when manufacturer-published BC data is unavailable or when you want to derive a field-validated BC from actual range data. It reports one row per (drag model × data basis) with a fit RMS and a reliability flag.

| Flag | Type | Default | Description |
|---|---|---|---|
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). |
| `-m, --mass` | float | — | Projectile mass (grains / grams). |
| `-d, --diameter` | float | — | Projectile diameter (inches / mm). |
| `--data` | string | — | Drop series `"dist,drop;...."` (yd/m, inches/m). |
| `--velocity-data` | string | — | Velocity-retention series `"dist,vel;...."` (yd/m, fps/mps). |
| `--drag-model` | string | `both` | `g1`, `g7`, `both`, or any single family (`g2/g5/g6/g8/gi/gs/ra4`). |
| `--zero-range` | float | — | Zero range the drop data was measured against (yd/m). Given, drop is fitted as drop below the line of sight; omitted, drop is bore-referenced. |
| `--sight-height` | float | 2″/50 mm | Sight height for the zeroed drop fit. |
| `--temperature` | float | 59/15 | Air temperature the data was measured at. |
| `--pressure` | float | 29.92/1013.25 | Barometric pressure the data was measured at. |
| `--humidity` | float | 50.0 | Relative humidity, 0–100%. |
| `--altitude` | float | 0.0 | Altitude the data was measured at (ft/m). |
| `--distance1` | float | — | First measurement distance (yd/m). Legacy 2-point input. |
| `--drop1` | float | — | Observed drop at distance 1 (inches / m). Legacy. |
| `--distance2` | float | — | Second measurement distance (yd/m). Legacy. |
| `--drop2` | float | — | Observed drop at distance 2 (inches / m). Legacy. |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. JSON and CSV carry a `reliable` field. |

### Example

Fit a five-point dope card shot from a 200 yd zero:

```
ballistics estimate-bc \
  -v 2700 -m 168 -d 0.308 \
  --data "300,8.5;400,24.6;500,49.7;600,85.5;700,134.1" \
```

```
  --zero-range 200 --sight-height 1.5
```

### A.5.2 generate-bc-segments — Velocity-Dependent BC Segments

The `generate-bc-segments` command produces a table of BC values across velocity bands, modeling the fact that a projectile's drag coefficient changes as it decelerates through different Mach regimes. The output can be fed to the `--use-bc-segments` flag in the `trajectory` command.

| Flag | Type | Default | Description |
|---|---|---|---|
| `-b, --bc` | float | — | Base ballistic coefficient. Range: 0.001–2.0. |
| `-m, --mass` | float | — | Projectile mass in kg. |
| `-d, --diameter` | float | — | Projectile diameter in meters. |
| `--model` | string | `""` | Bullet model name (e.g., "SMK", "ELD-M", "VLD"). |
| `--drag-model` | `G1` \| `G7` | `G1` | Standard drag function. |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

**Example**

```
ballistics --units metric generate-bc-segments \
  -b 0.462 -m 10.89 -d 7.823 \
  --model "SMK" --drag-model g1 -o table
```

### A.5.3 true-velocity — Effective Muzzle Velocity from Observed Drop

The `true-velocity` command solves for the effective muzzle velocity that produces a given observed drop (in MILs) at a known range. This is the definitive method for truing your ballistic solution against field data: rather than trusting a chronograph reading taken at the muzzle, you derive the velocity that actually explains the bullet's behavior downrange.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--measured-drop` | float | — | Measured drop in MILs at the target range. Required. |
| `--range` | float | — | Range at which drop was measured (yards / meters). Required. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `-b, --bc` | float | — | Ballistic coefficient. Range: 0.001–2.0. |
| `--drag-model` | `g1 \| g7` | `g1` | Drag function. |
| `-m, --mass` | float | — | Bullet weight (grains / grams). |
| `-d, --diameter` | float | — | Bullet diameter (inches / mm). |
| `--chrono-velocity` | float | — | Chronograph velocity for comparison (fps / m/s). Optional. |
| `--zero-distance` | float | 100.0 | Zero distance (yards / meters). |
| `--sight-height` | float | 2.0 | Sight height above bore (inches / mm). |
| `--temperature` | float | 59.0 | Temperature (°F / °C). |
| `--pressure` | float | 29.92 | Barometric pressure (inHg / hPa). |
| `--humidity` | float | 50.0 | Humidity, 0–100%. |
| `--altitude` | float | 0.0 | Altitude (feet / meters). |
| `--bullet-length` | float | — | Bullet length in inches for BC table lookup. |
| `--bc-table-dir DIR` | path | — | Directory for BC5D tables. |
| `--offline` | bool | false | Force offline mode (skip API). |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

**Example**

True your .308 Win solution using observed 4.2 MIL drop at 600 yards:

```
ballistics true-velocity \
  --measured-drop 4.2 --range 600 \
  -b 0.462 -m 168 -d 0.308 \
  --chrono-velocity 2700 --zero-distance 100 \
  --sight-height 1.5
```

## A.6 Utility Commands

### A.6.1 mpbr — Maximum Point-Blank Range

The `mpbr` command calculates the maximum range at which a projectile stays within a specified vital zone diameter without hold-over or hold-under adjustments. It determines the optimal zero distance that maximizes this point-blank envelope—critical information for hunting applications where rapid target engagement precludes dialing turrets.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--profile NAME` | string | — | Load parameters from a saved profile. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). |
| `-b, --bc` | float | — | Ballistic coefficient. |
| `-m, --mass` | float | — | Mass (grains / grams). |
| `-d, --diameter` | float | — | Diameter (inches / mm). |
| `--drag-model` | `g1 \| g7` | `g1` | Drag function. |
| `--vital-zone` | float | 8.0 | Vital zone diameter (inches / cm). |
| `--sight-height` | float | — | Sight height above bore (inches / mm). |
| `--temperature` | float | 59.0 | Temperature (°F / °C). |
| `--pressure` | float | 29.92 | Pressure (inHg / hPa). |
| `--humidity` | float | 50.0 | Humidity, 0–100%. |
| `--altitude` | float | 0.0 | Altitude (feet / meters). |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

**Example**

Determine MPBR for a .308 Win 168 gr SMK on a deer-sized vital zone:

```
ballistics mpbr \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --vital-zone 8 --sight-height 1.5
```

### A.6.2 come-ups — Elevation Adjustment Table

The `come-ups` command generates a table of elevation adjustments (in MIL or MOA) for a range of distances relative to your zero. This is the table a precision shooter dials onto their turret or holds in their reticle at each range increment.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--profile NAME` | string | — | Load a saved profile. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). |
| `-b, --bc` | float | — | Ballistic coefficient. |
| `-m, --mass` | float | — | Mass (grains / grams). |
| `-d, --diameter` | float | — | Diameter (inches / mm). |
| `--drag-model` | `g1 \| g7` | `g1` | Drag function. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--zero-distance` | float | — | Zero distance (yards / meters). Required. |
| `--start` | float | 100 | Start range (yards / meters). |
| `--end` | float | 1200 | End range (yards / meters). |
| `--step` | float | 50 | Range step (yards / meters). |
| `--adjustment-unit` | enum | `mil` | Angular unit: `mil`, `moa`, `smoa`, `iphy`, or `clicks`. `clicks` requires `--elevation-click-value` (or a profile's `elevation_click`) and rounds to whole turret clicks. |
| `--windage-unit` | enum | — | Independent windage unit; falls back to `--adjustment-unit`. |
| `--elevation-click-value` | string | — | Turret graduation with a unit suffix, e.g. `0.25moa` or `0.1mil`. A bare number is an error. |
| `--windage-click-value` | string | — | Windage turret graduation, same form. |
| `--sight-height` | float | — | Sight height (inches / mm). |
| `--temperature` | float | 59.0 | Temperature. |
| `--pressure` | float | 29.92 | Pressure. |
| `--humidity` | float | 50.0 | Humidity. |
| `--altitude` | float | 0.0 | Altitude. |
| `--wind-speed` | float | 0.0 | Wind speed (mph / m/s). |
| `--wind-direction` | float | 0.0 | Wind direction (degrees). |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

### Example

Generate a MIL come-up table for a 6.5 Creedmoor from 100 to 1200 yards:

```
ballistics come-ups \
  -v 2710 -b 0.610 -m 140 -d 0.264 --drag-model g7 \
  --zero-distance 100 --start 100 --end 1200 --step 50 \
  --adjustment-unit mil --sight-height 1.5 \
  --wind-speed 10 --wind-direction 90
```

### A.6.3 wind-card — Wind Drift Card

The `wind-card` command generates a matrix of wind deflection values across multiple wind speeds and range increments, expressed in MIL or MOA. This is the table a shooter references to apply wind holds in the field.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--profile NAME` | string | — | Load a saved profile. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). |
| `-b, --bc` | float | — | Ballistic coefficient. |
| `-m, --mass` | float | — | Mass (grains / grams). |
| `-d, --diameter` | float | — | Diameter (inches / mm). |
| `--drag-model` | `g1 \| g7` | `g1` | Drag function. |
| `--zero-distance` | float | — | Zero distance (yards / meters). Required. |
| `--wind-speeds` | string | `"5,10,15,20"` | Comma-separated wind speeds (mph / m/s). |
| `--start` | float | 100 | Start range (yards / meters). |
| `--end` | float | 1000 | End range (yards / meters). |
| `--step` | float | 100 | Range step (yards / meters). |
| `--adjustment-unit` | enum | `mil` | Angular unit: `mil`, `moa`, `smoa`, `iphy`, or `clicks`. |
| `--windage-unit` | enum | — | Independent windage unit; falls back to `--adjustment-unit`. |
| `--sight-height` | float | — | Sight height (inches / mm). |
| `--temperature` | float | 59.0 | Temperature. |
| `--pressure` | float | 29.92 | Pressure. |
| `--humidity` | float | 50.0 | Humidity. |
| `--altitude` | float | 0.0 | Altitude. |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

#### Example

Generate a wind card for a .308 Win at 5, 10, and 15 mph crosswinds:

```
ballistics wind-card \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --zero-distance 100 --wind-speeds "5,10,15" \
  --start 100 --end 1000 --step 100 \
  --adjustment-unit mil --sight-height 1.5
```

### A.6.4 range-table — Comprehensive Range Table

The `range-table` command produces a unified table combining drop, wind deflection, remaining velocity, kinetic energy, and time of flight at each range increment. It represents the most complete single-command output available in BALLISTICS-ENGINE.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--profile NAME` | string | — | Load a saved profile. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). |
| `-b, --bc` | float | — | Ballistic coefficient. |
| `-m, --mass` | float | — | Mass (grains / grams). |
| `-d, --diameter` | float | — | Diameter (inches / mm). |
| `--drag-model` | `g1` \| `g7` | `g1` | Drag function. |
| `--zero-distance` | float | — | Zero distance (yards / meters). Required. |
| `--start` | float | 100 | Start range. |
| `--end` | float | 1200 | End range. |
| `--step` | float | 50 | Range step. |
| `--wind-speed` | float | 10.0 | Wind speed (mph / m/s). |
| `--wind-direction` | float | 90.0 | Wind direction (degrees, 90 = full crosswind). |
| `--adjustment-unit` | enum | `mil` | Angular unit: `mil`, `moa`, `smoa`, `iphy`, or `clicks`. |
| `--windage-unit` | enum | — | Independent windage unit; falls back to `--adjustment-unit`. |
| `--sight-height` | float | — | Sight height (inches / mm). |
| `--temperature` | float | 59.0 | Temperature. |
| `--pressure` | float | 29.92 | Pressure. |
| `--humidity` | float | 50.0 | Humidity. |
| `--altitude` | float | 0.0 | Altitude. |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

**Example**

Full range table for a 6.5 Creedmoor 140 gr ELD-M with 10 mph crosswind:

```
ballistics range-table \
  -v 2710 -b 0.610 -m 140 -d 0.264 --drag-model g7 \
  --zero-distance 100 --start 100 --end 1200 --step 50 \
  --wind-speed 10 --wind-direction 90 \
  --adjustment-unit mil --sight-height 1.5
```

### A.6.5 stability — Gyroscopic Stability Analysis

The `stability` command computes the Miller stability factor ($S_g$) for a given bullet and twist rate combination. A stability factor above 1.0 indicates the bullet is gyroscopically stable; values between 1.3 and 2.0 are generally considered optimal, while values above 3.0 suggest an excessively fast twist that can amplify spin-related dispersion.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--profile NAME` | string | — | Load a saved profile. |
| `-m, --mass` | float | — | Bullet mass (grains / grams). |
| `-d, --diameter` | float | — | Bullet diameter (inches / mm). |
| `-l, --length` | float | — | Bullet length (inches / mm). Required. |
| `-t, --twist-rate` | float | — | Barrel twist rate (inches/turn or mm/turn). Required. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). Default: 2700 fps / 823 m/s. |
| `--temperature` | float | 59.0 | Temperature. |
| `--pressure` | float | 29.92 | Pressure. |
| `--altitude` | float | 0.0 | Altitude. |
| `-o, --output` | enum | `table` | Output: `table`, `json`, `csv`. |

**Example**

Check stability of a 6.5 Creedmoor 140 gr ELD-M (1.376 in length) in a 1:8 twist barrel:

```
ballistics stability \
  -m 140 -d 0.264 -l 1.376 -t 8 -v 2710
```

## A.7 Comparison and Calculator Commands

### A.7.1 compare — Multi-Load Comparison

Runs 2–8 loads side by side at the same conditions, each zeroed independently at the shared `--zero-distance`. See Section 25.2.3.

| Flag | Type | Description |
|---|---|---|
| `--load SPEC` | string | Inline load, repeatable: `NAME:DRAG:BC:MASS:VELOCITY[:DIAMETER]`. |
| `--profile NAME` | string | Saved profile as a load, repeatable and mixable with `--load`. Carries its BC segments and custom drag curve into both the zeroing and the trajectory runs. |
| `--zero-distance` | float | Shared zero distance (yd/m). |
| `--start`, `--end`, `--step` | float | Range band and increment for the table. |
| `--wind-speed`, `--wind-direction` | float | Shared wind applied to every load. |
| `--sight-height` | float | Sight height above bore (inches / mm). |
| `-o`, `--output` | enum | `table`, `json`, `csv`. JSON adds energy, time of flight, per-load zero angles, and deltas vs load #1. |

```
ballistics compare \
  --load "155 Palma:g7:0.235:155:2950:0.308" \
  --load "175 SMK:g7:0.253:175:2600:0.308" \
  --zero-distance 100 --sight-height 1.5 \
  --start 600 --end 1000 --step 200
```

### A.7.2 powder — Powder-Temperature Velocity

Resolves the powder-temperature-adjusted muzzle velocity without running a trajectory. See Section 25.6.3.

| Flag | Type | Description |
|---|---|---|
| `-v`, `--velocity` | float | Nominal muzzle velocity (fps / m/s). |
| `--temperature` | float | Shot temperature (°F / °C). |
| `--powder-temp-sensitivity` | float | Linear model, velocity change per degree. |
| `--powder-temp` | float | Reference temperature for the linear model, or the temperature a curve is read at. |
| `--powder-temp-curve` | string | Measured `TEMP:VEL,...` curve. Overrides the linear model. |
| `--sweep S:E:ST` | string | Print a velocity ladder across a temperature range instead of one answer. |
| `-m`, `--mass` | float | Bullet mass; adds the muzzle-energy column. |
| `-o`, `--output` | enum | `table`, `json`, `csv`. |

**Model asymmetry**

`powder` always applies the linear model, while `trajectory` and `lead` gate theirs behind `--use-powder-sensitivity`. A measured curve applies unconditionally everywhere.

### A.7.3 tall-target — Scope Tracking Correction Factor

Pure arithmetic—no trajectory solve. See Section 24.6.

| Flag | Type | Description |
|---|---|---|
| `--dialed` | float | Dialed travel, in `--unit`. |
| `--unit` | enum | Angular unit the turret was dialed in: `mil`, `moa`, `smoa`, `iphy`. |
| `--measured` | float | Actual measured group travel on the target (inches / cm). |
| `--range` | float | Distance to the tall target (yards / meters). |

### A.7.4 true-wind — Wind-Call Truing

Back-solves the effective crosswind from an observed horizontal miss. Fully offline. See Section 10.8.

| Flag | Type | Description |
|---|---|---|
| `--miss R:RIGHT[:S]` | string | Observed miss, repeatable. RANGE follows `--units`; the miss and its sigma are **inches in both unit systems**. Positive = impact RIGHT of aim. |
| `--twist-rate` | float | **Required.** Spin drift is always modelled and subtracted. |
| `--twist-right` | bool | Twist direction; a right-hand twist drifts right. |
| `--latitude`, `--shot-direction` | float | Optional, but must be supplied *together*. Adds Coriolis. |
| `--called-wind` | float | The wind you called; adds a correction factor (solved ÷ called). |
| `--zero-distance`, `--sight-height` | float | Sight geometry for the forward model. |
| `-o`, `--output` | enum | `table`, `json`, `csv`. |

### A.7.5 reticle — Reticle Descriptions and Hold Points

Two verbs. `reticle generate {mil-grid|tree|bdc}` builds a description; `reticle hold` places a solution in one. See Chapter 29.

| Flag | Verb | Description |
|---|---|---|
| `--spacing`, `--extent` | `generate mil-grid` | Hash spacing and half-extent, in mil. |
| `--rows`, `--row-spacing`, `--spread-step` | `generate tree` | Tree geometry. |
| `--drop RANGE:MIL` | `generate bdc` | An already-solved drop, repeatable. The gener runs no solve of its own. |
| `--focal-plane` | `generate` | `ffp` or `sfp`. |
| `--reference-mag` | `generate` | Magnification an SFP reticle is calibrated at. |
| `--name` | `generate` | Description name. |
| `<FILE>` | `import` | A Ventum reticle JSON file to convert. Positic Cosmetic keys are ignored; a decoration-only s fails as `NoMarks`. |
| `--reticle-json FILE` | `hold` | Description to place the hold in. Inline JSON in the browser build. |
| `--drop-mil`, `--wind-mil` | `hold` | Supply the solution directly. |
| `--range` | `hold` | Solve the trajectory instead. Native CLI only. |
| `--mag` | `hold` | Magnification in use. Matters only for SFP. |

### A.7.6 mark-to-range, bdc-match, optimal-zero

The three reticle/BDC inverse solvers, all read-only over an existing load. See Section 29.3.

| Flag | Command | Description |
|---|---|---|
| `--mark MIL` | `mark-to-range` | A subtension to map, repeatable. Or use `--reticle-json`. |
| `--mark-range MIL:R` | `bdc-match` | A mark-and-range pair to fit, repeatable. |
| `--reference-mag` | `bdc-match` | The reticle's calibration magnification. |
| `--residual-warn` | `bdc-match` | Warn above this worst residual. Default 0.2 mil. |
| `--target R[:HEIGHT]` | `optimal-zero` | A target, repeatable, 2–16 of them. |
| `--vital` | `optimal-zero` | Default target height (inches / cm). |

### A.7.7 hold-corridor — Robust Holds Across Wind Scenarios

See Section 29.4.

| Flag | Type | Description |
|---|---|---|
| `--scenarios FILE` | path | Named scenario set JSON. Segments use the `--wind-segment` token grammar. Max 8 scenarios. |
| `--ranges R1,R2,...` | list | Ranges to report. Max 64. |
| `--target SPEC` | string | `rect:WxH` (per-axis metric) or `circle:D` (minimum enclosing circle). Omit for no fit verdict. |
| `-o, --output` | enum | `table`, `json` (a versioned `RobustHoldReportV1`), `csv`. |

### A.7.8 drag-curve — Print a Reference Drag Curve

Prints a built-in reference drag function as $(\mathrm{Mach}, C_D)$ data. Points come out verbatim from the table—no resampling—so a consumer charts exactly what the solver interpolates. Note the tables do not share a Mach domain: most run to Mach 5, while GS and RA4 stop at Mach 4.

| Flag | Type | Description |
|---|---|---|
| `--drag-model` | enum | Which family to print: `g1`, `g2`, `g5`, `g6`, `g7`, `g8`, `gi`, `gs`, `ra4`. |
| `-o, --output` | enum | `table` (default), `csv`, `json`. |

```
ballistics drag-curve --drag-model g7
```

### A.7.9 recoil — Free Recoil

Free recoil energy, velocity and impulse from a SAAMI momentum balance, including the propellant-gas ejecta term. See Section 25.4.

| Flag | Type | Description |
|---|---|---|
| `--bullet-weight` | float | Bullet weight (grains / grams). Required. |
| `--charge-weight` | float | Propellant charge weight (grains / grams). Required. |
| `--velocity` | float | Muzzle velocity (fps / m/s). Required. |
| `--firearm-weight` | float | Firearm weight (lb / kg). Required. |
| `-o, --output` | enum | `table`, `json`, `csv`. |

### A.7.10 power-factor — Competition Power Factor

Power factor with per-organization USPSA, IDPA and SASS pass/fail. Scored values are truncated, not rounded. See Section 24.7.

| Flag | Type | Description |
|---|---|---|
| `--weight` | float | Bullet weight (grains / grams). Required. |
| `--velocity` | float | Muzzle velocity (fps / m/s). Required. |
| `-o, --output` | enum | `table`, `json`, `csv`. |

### A.7.11 plan-truing — Truing Experiment Design

Chooses the set of target ranges that will identify muzzle velocity and BC jointly, given the candidate ranges available and the reading precision achievable. See Section 7.4.7.

| Flag | Type | Description |
|---|---|---|
| `--range-grid S:E:ST` | string | Candidate ranges as a grid. |
| `--candidate-ranges R1,R2,...` | list | Explicit candidate list, as an alternative to the grid. |
| `--observation-count` | int | Number of stations to select. |
| `--minimum-separation` | float | Minimum spacing between selected stations (yd/m). |
| `--measurement-resolution` | float | Independent $1\sigma$ reading error, in `--drop-unit`. Required—the plan is sensitive to it. |
| `--drop-unit` | `mil` \| `moa` \| `in` | Unit the resolution and drops are expressed in. |
| `--zero-distance`, `--sight-height` | float | Sight geometry for the forward model. |
| `--profile NAME` | string | Take the load from a saved profile. |
| `-o, --output` | enum | `table`, `json`, `csv`. |

### A.7.12 true-velocity: uncertainty-aware flags

The flags below turn the joint MV + BC fit into a weighted MAP estimate with parameter intervals. See Section 7.4.8.

| Flag | Type | Description |
|---|---|---|
| `--observed R:D[:S]` | string | Additional observation, repeatable. The optional third field is that reading's own $1\sigma$. |
| `--observation-sigma` | float | Default known $1\sigma$ for every reading, in `--drop-unit`. |

| Flag | Type | Description |
|---|---|---|
| `--mv-prior MEAN:SIGMA` | string | Explicit independent prior on muzzle velocity. |
| `--bc-prior MEAN:SIGMA` | string | Explicit independent prior on BC. |
| `--predict-range` | float | Propagate the fitted covariance into a 95% drop band at this range. |
| `--prediction-sigma` | float | Additional per-observation noise for the future-observation band. |
| `--drop-unit` | `mil` \| `moa` \| `in` | Unit for `--measured-drop` and every `--observed` drop in multi-observation mode. |

## A.8 profile — Manage Saved Profiles

The `profile` command manages saved ballistic profiles stored in `~/.ballistics/profiles/`. Profiles persist complete ballistic configurations—projectile data, zero distance, atmospheric defaults, and barrel parameters—so that any other command can recall them by name with a single `--profile NAME` or `--saved-profile NAME` flag.

### A.8.1 profile save

Save a new profile with the specified parameters. All subsequent commands can reference this profile by name.

| Flag | Type | Default | Description |
|---|---|---|---|
| `NAME` (positional) | string | — | Profile name (used for recall). Required. |
| `-v, --velocity` | float | — | Muzzle velocity (fps / m/s). Required. |
| `-b, --bc` | float | — | Ballistic coefficient. Required. |
| `-m, --mass` | float | — | Mass (grains / grams). Required. |
| `-d, --diameter` | float | — | Diameter (inches / mm). Required. |
| `--drag-model` | `g1` \| `g7` | `g1` | Drag function. |
| `--twist-rate` | float | — | Barrel twist rate (inches/turn). |
| `--sight-height` | float | — | Sight height (inches / mm). |
| `--zero-distance` | float | — | Default zero distance. |
| `--temperature` | float | 59.0 | Default temperature. |
| `--pressure` | float | 29.92 | Default pressure. |
| `--humidity` | float | 50.0 | Default humidity. |

| Flag | Type | Default | Description |
|---|---|---|---|
| `--altitude` | float | 0.0 | Default altitude. |
| `--bullet-name` | string | — | Bullet name/description. |
| `--wind-speed` | float | — | Default wind speed. |
| `--wind-direction` | float | — | Default wind direction. |
| `--shooting-angle` | float | — | Default shooting angle (degrees). |
| `--auto-zero` | float | — | Default auto-zero distance. |
| `--twist-right` | bool | — | Right-hand twist. |
| `--use-bc-segments` | bool | — | Enable BC segmentation by default. |
| `--bullet-length` | float | — | Bullet length in inches. |

**Example**

```
ballistics profile save "R700_308" \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --drag-model g1 --twist-rate 10 \
  --sight-height 1.5 --zero-distance 100 \
  --bullet-name "168gr SMK"
```

### A.8.2 profile list

List all saved profiles with their names and summary information.

```
ballistics profile list
```

### A.8.3 profile show

Display the full contents of a saved profile.

```
ballistics profile show "R700_308"
```

### A.8.4 profile import

Imports a profile from a third-party file. Only the ArcherBC2 `.a7p` format is supported. A full mapping report is printed either way; see Section 8.5.5.

| Flag | Type | Description |
|---|---|---|
| `<FILE>` | path | Path to the `.a7p` file. Positional, required. |
| `--name` | string | Profile name to save under. Defaults to the file's own profile name, sanitized. |
| `--dry-run` | bool | Parse and print the mapping report without saving anything. |
| `--strict` | bool | Treat a checksum mismatch as fatal instead of a warning. |
| `--zero-click` | string | The source device's click graduation, e.g. `0.1mil` or `0.25moa`. Without it the file's `zero_x`/`zero_y` click counts cannot be converted and are reported as unmapped. |

```
ballistics profile import my-load.a7p --dry-run
```

### A.8.5 profile delete

Delete a saved profile permanently.

```
ballistics profile delete "R700_308"
```

## A.9 Online Reverse Solvers

These six subcommands are the only ones in BALLISTICS-ENGINE that call a network service, and they are gated behind the default-on `online` feature. Every local trajectory and analysis command is unaffected and needs no token. Each solver targets `https://api.ballistics.7.62x51mm.sh` by default; `--api-url` overrides it. See Chapter 30.

### A.9.1 login / logout

| Command | Flag | Description |
|---|---|---|
| `login` | `--token` | The access token from `https://ballisticsinsight.com/account`. Omit it and you are prompted to paste it. |
| `logout` | — | Removes the saved token. |

The token is stored at `~/.ballistics/credentials.toml`, written `0600`. The `BALLISTICS_API_TOKEN` environment variable takes precedence over the file.

### A.9.2 recommend-powder

| Flag | Type | Description |
|---|---|---|
| `--cartridge` | string | Cartridge name. Required. |
| `--bullet-weight` | float | Bullet weight in grains. Required. |
| `--desired-velocity` | float | Target muzzle velocity in fps. Required. |
| `--barrel-length` | float | Barrel length in inches. Optional. |
| `--velocity-tolerance` | float | Acceptable velocity window in fps. Optional. |
| `--api-url` | string | Override the service endpoint. |

### A.9.3 recommend-twist

| Flag | Type | Description |
|---|---|---|
| `--caliber` | float | Bullet caliber in inches. Required. |
| `--weight` | float | Bullet weight in grains. Required. |
| `--overall-length` | float | Total bullet length in inches. Required. |
| `--nose-length` | float | Nose length in inches. Required. |
| `--muzzle-velocity` | float | Muzzle velocity in fps. Optional. |
| `--api-url` | string | Override the service endpoint. |

### A.9.4 recommend-col

| Flag | Type | Description |
|---|---|---|
| `--cartridge` | string | Cartridge name. Required. |
| `--bullet-weight` | float | Bullet weight in grains. Optional. |
| `--bullet-type` | string | Bullet type, e.g. `match`. Optional. |
| `--api-url` | string | Override the service endpoint. |

### A.9.5 calibrate-bc

| Flag | Type | Description |
|---|---|---|
| `--diameter` | float | Bullet diameter in inches. Required. |
| `--length` | float | Bullet length in inches. Required. |
| `--weight` | float | Bullet weight in grains. Required. |
| `--bullet-type` | string | Bullet type. Optional. |
| `--api-url` | string | Override the service endpoint. |

**Exit codes and hints**

A `401` prints `Unauthorized - run `ballistics login`...` and a `402` points at the subscription page; anything else falls back to `Online request failed: ...`. All of them exit non-zero.

## A.10 Protocol Commands

### A.10.1 solve-json — One Solve, One JSON Document

Reads one `solve-json` v1 request from standard input and writes one compact JSON envelope to standard output. Every dimensional field is explicit SI regardless of `--units`; the command reads no profiles, files, or network, and input is capped at 1 MiB. Fully documented in Section 27.4.1.

```
ballistics solve-json < request.json > response.json
```

| Exit status | Meaning |
|---|---|
| `0` | Success envelope. |
| `1` | Standard-I/O or internal failure. |
| `2` | Malformed JSON, schema, shape, or semantic validation failure. |
| `3` | Resource limit or engine solve failure. |

### A.10.2 mcp — Model Context Protocol Server

Runs a zero-dependency MCP server over stdio, exposing a `solve` tool on the `solve-json` v1 schema and an `engine_info` tool.

```
ballistics mcp
```

## A.11 completions — Shell Completion Generation

The `completions` command generates shell completion scripts for tab-completion of commands, flags, and argument values. Supported shells include Bash, Zsh, Fish, PowerShell, and Elvish.

```
# Generate and install Bash completions
ballistics completions bash > ~/.local/share/bash-completion/completions/ballistics

# Generate and install Zsh completions
ballistics completions zsh > ~/.zfunc/_ballistics

# Generate Fish completions
ballistics completions fish > ~/.config/fish/completions/ballistics.fish
```

After generating the completion script, restart your shell or source the file for immediate effect.

## A.12 Output Formats

Most commands support multiple output formats via the `--output` (or `-o`) flag.

| Format | Description |
|---|---|
| `table` | Human-readable tabular output with aligned columns and headers, suitable for terminal display. This is the default for all commands. |
| `json` | Structured JSON output for programmatic consumption, scripting, and integration with external tools. All numeric values are emitted at full floating-point precision. |
| `csv` | Comma-separated values, directly importable into spreadsheets and data analysis tools. Column headers are included in the first row. |
| `pdf` | PDF dope card output (available only for the `trajectory` command). Generates a formatted range card suitable for printing and laminating for field use. Requires the `--output-file` flag to specify the destination path. |

**Output format availability**

The `pdf` format is exclusive to the `trajectory` command and requires the binary to be compiled with the `pdf` feature enabled. The `monte-carlo` command uses a distinct set of output modes: `summary`, `full`, and `statistics`.

### A.12.1 Piping and Composition

The structured output formats (`json` and `csv`) are designed for composition with standard Unix tools and scripting languages:

```
# Extract 500-yard drop from JSON output
ballistics trajectory \
  -v 2700 -b 0.462 -m 168 -d 0.308 \
  --auto-zero 100 --sight-height 1.5 \
  -o json | jq '.trajectory[] | select(.range == 500)'

# Export range table to spreadsheet
ballistics range-table \
  -v 2710 -b 0.610 -m 140 -d 0.264 --drag-model g7 \
  --zero-distance 100 \
  -o csv > range_table_65cm.csv
```

## A.13 Online Mode Flags

The following flags are available only when BALLISTICS-ENGINE is compiled with the `online` feature (`cargo build --features online`). They enable cloud-based ML-enhanced trajectory calculations, automatic BC$_5$D table downloads, and advanced weather modeling.

**Terms of Service**

The first invocation of any `--online` flag triggers an interactive Terms of Service prompt. The online feature sends ballistic data to a proprietary cloud service; acceptance is recorded locally and re-prompted only when the terms change.

### A.13.1 API Connection Flags

These flags are available on the `trajectory` and `true-velocity` commands.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--online` | bool | false | Route calculations through th Flask API for ML-enhanced traje tory computation. |
| `--api-url` | string | `https://api. ballistics. 7.62x51mm.sh` | API endpoint URL. |
| `--offline-fallback` | bool | false | Fall back to local calculation if th API is unreachable. |
| `--compare` | bool | false | Run both local and API calcul tions and display results side b side. |
| `--api-timeout` | int | 10 | API request timeout in seconds. |

### A.13.2 Weather Zone Flags

Available on the `trajectory` command when the `online` feature is enabled.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--enable-weather-zones` | bool | false | Enable automatic weather zone generation along the trajectory path. Requires `--latitude`, `--longitude`, and `--shot-direction`. |
| `--enable-3d-weather` | bool | false | Enable 3D weather corrections with altitude-dependent atmospheric changes. |
| `--wind-shear-model` | string | `logarithmic` | Wind shear model: `none`, `logarithmic`, `power_law`, `ekman_spiral`. |
| `--weather-zone-interpolation` | string | `linear` | Interpolation between weather zones: `linear`, `cubic`, `step`. |

### A.13.3 BC5D Auto-Download Flags

Available on the `trajectory` and `true-velocity` commands.

| Flag | Type | Default | Description |
|---|---|---|---|
| `--bc-table-auto` | bool | false | Auto-download BC5D correction tables when a matching table is not found locally. |
| `--bc-table-url` | string | `https://ballistics.tools/downloads/bc5d` | Base URL for BC5D table downloads. |
| `--bc-table-refresh` | bool | false | Force re-download of BC5D tables even if a cached version exists. |

### A.13.4 Example: Online Trajectory with Weather Zones

```
ballistics trajectory \
  -v 2710 -b 0.610 -m 140 -d 0.264 --drag-model g7 \
  --auto-zero 100 --sight-height 1.5 \
  --latitude 38.52 --longitude -77.31 \
  --shot-direction 270 \
  --online --offline-fallback \
  --enable-weather-zones --enable-3d-weather \
  --wind-shear-model logarithmic
```

# Appendix B

# Physics Constants

This appendix catalogues every physical constant, conversion factor, and numerical threshold compiled into BALLISTICS-ENGINE. All values are defined at compile time in `src/constants.rs` and `src/atmosphere.rs`; they are reproduced here as a quick reference for anyone wishing to verify calculations, audit the solver against independent data, or extend the engine with new physics modules.

Where applicable, authoritative sources are cited. Unless noted otherwise, SI base units are used internally and imperial quantities appear only in user-facing input/output paths.

## B.1 Gravitational and Atmospheric Constants

The constants in Table B.1 underpin every trajectory integration step. Gravitational acceleration uses the conventional standard value adopted by the 3rd CGPM (1901). Air density and the speed of sound correspond to ICAO Standard Atmosphere sea-level conditions: $15\,^{\circ}\mathrm{C}$, $1013.25\,\mathrm{hPa}$, dry air.

Table B.1: Gravitational and atmospheric constants (`src/constants.rs`, `src/atmosphere.rs`).

| Constant | Value | Unit | Description |
|---|---|---|---|
| `G_ACCEL_MPS2` | $9.80665$ | $\mathrm{m/s^2}$ | Standard gravitational acceleration |
| `STANDARD_AIR_DENSITY` | $1.225$ | $\mathrm{kg/m^3}$ | Sea-level air density (ICAO) |
| `AIR_DENSITY_SEA_LEVEL` | $1.225$ | $\mathrm{kg/m^3}$ | Alias of above |
| `SPEED_OF_SOUND_MPS` | $340.29$ | $\mathrm{m/s}$ | Speed of sound at sea level |
| `R_AIR` | $287.0531$ | $\mathrm{J/(kg \cdot K)}$ | Specific gas constant, dry air |
| `GAMMA` | $1.4$ | — | Heat capacity ratio ($c_p/c_v$) for air |
| `CD_TO_RETARD` | $2.08551 \times 10^{-4}$ | — | Drag-to-retardation conversion, ICAO-referenced |

The drag-to-retardation constant `CD_TO_RETARD` is the imperial retardation coefficient for a density normalised to ICAO sea-level air ($1.225\,\mathrm{kg/m^3}$, or `ICAO_DENSITY_LB_FT3` $= 0.076474\,\mathrm{lb/ft^3}$). It enters the equation of motion as

$$a_{\text{drag}} = \texttt{CD_TO_RETARD} \cdot C_d \cdot v^2 \cdot \frac{\rho/\rho_0}{\mathrm{BC}} \tag{B.1}$$

where $C_d$ is the drag coefficient, $v$ the projectile velocity in fps, and $\rho/\rho_0$ the ratio of local air density to the ICAO sea-level value.

**This Constant Changed in 0.22.0**

Through v0.21.x the constant was $0.000683 \times 0.30 = 2.049 \times 10^{-4}$. That is the *Army Standard Metro* retardation coefficient, and it is only consistent with a reference density of `ASM_DENSITY_LB_FT3` $= 0.075126\,\mathrm{lb/ft^3}$—about 1.8% denser than ICAO. Since virtually every published BC the engine consumes, and the engine's own drag math, are ICAO-referenced, drag came out uniformly about 1.75% low. Version 0.22.0 replaced it with the ICAO figure above. The ASM density is still in `src/constants.rs`, but only so that `ASM_TO_ICAO_BC` can convert an ASM-referenced published BC into the ICAO-referenced number the solver expects (see Chapter 12).

## B.2 Unit Conversion Factors

BALLISTICS-ENGINE accepts input in both metric and imperial systems and converts to SI internally. Table B.2 lists every conversion factor hard-coded in the engine.

Table B.2: Unit conversion factors used throughout BALLISTICS-ENGINE.

| Conversion | Multiply by | Source constant |
|---|---|---|
| $\mathrm{m/s} \to \mathrm{ft/s}$ | $3.28084$ | `MPS_TO_FPS` |
| $\mathrm{ft/s} \to \mathrm{m/s}$ | $0.3048$ | `FPS_TO_MPS` |
| $\mathrm{grains} \to \mathrm{kg}$ | $6.479891 \times 10^{-5}$ | `GRAINS_TO_KG` |
| $\mathrm{grains} \to \mathrm{g}$ | $0.06479891$ | (derived) |
| $\mathrm{inches} \to \mathrm{m}$ | $0.0254$ | (inline) |
| $\mathrm{yards} \to \mathrm{m}$ | $0.9144$ | (inline) |
| $^\circ\mathrm{F} \to {}^\circ\mathrm{C}$ | $(F - 32) \times 5/9$ | (inline) |
| $\mathrm{inHg} \to \mathrm{hPa}$ | $33.8639$ | (inline) |
| $\mathrm{ft \cdot lbf} \to \mathrm{J}$ | $1.35582$ | (inline) |

Constants marked "(inline)" are not named constants in the source but appear as literal values in conversion code paths. The grain-to-kilogram factor $6.479891 \times 10^{-5}$ is the exact SI definition of the grain (1 grain $= \frac{1}{7000}$ lb avoirdupois).

## B.3 Standard Projectile Reference Values

When the user does not supply a ballistic coefficient, BALLISTICS-ENGINE falls back to statistically derived reference values. These constants were computed from a database of over 2,000 measured projectile BCs and represent the 25th percentile of each category—a deliberately conservative choice that avoids over-predicting ballistic performance.

### B.3.1 Overall Fallback

The general-purpose fallback is `0.31` (`BC_FALLBACK_CONSERVATIVE`), applicable when neither weight nor caliber information is available.

### B.3.2 By Projectile Weight

Table B.3: BC fallback values by projectile weight category (`src/constants.rs`).

| Constant | Weight Range | BC | Typical Cartridges |
|---|---|---|---|
| `BC_FALLBACK_ULTRA_LIGHT` | 0–50 gr | `0.172` | .17 cal varmint, .22 target |
| `BC_FALLBACK_LIGHT` | 50–100 gr | `0.242` | .223 Rem, .243 Win |
| `BC_FALLBACK_MEDIUM` | 100–150 gr | `0.310` | .270 Win, .30-06 |
| `BC_FALLBACK_HEAVY` | 150–200 gr | `0.393` | .308 Win match, .300 WM |
| `BC_FALLBACK_VERY_HEAVY` | 200+ gr | `0.441` | .338 Lapua, .50 BMG |

### B.3.3 By Caliber

Table B.4: BC fallback values by caliber category (`src/constants.rs`).

| Constant | Caliber Range | BC | Examples |
|---|---|---|---|
| `BC_FALLBACK_SMALL_CALIBER` | .224″ and smaller | `0.215` | .17 Rem, .223 Rem |
| `BC_FALLBACK_MEDIUM_CALIBER` | .243″ | `0.300` | .243 Win, 6mm Creedmoor |
| `BC_FALLBACK_LARGE_CALIBER` | .264″—.284″ | `0.404` | .270 Win, 7mm Rem Mag |
| `BC_FALLBACK_XLARGE_CALIBER` | .308″ and larger | `0.291` | .308 Win, .30-06, .300 WM |

**Why is the extra-large caliber BC lower than large caliber?**

The .308″-and-larger category includes a large population of older, flat-base, and round-nose bullet designs whose low BCs pull the 25th percentile below that of the .264″—.284″ range, which is dominated by modern, high-BC match projectiles.

## B.4 ICAO Standard Atmosphere

BALLISTICS-ENGINE implements the full ICAO Standard Atmosphere (ISO 2533) from sea level to 84 km. The model divides the atmosphere into seven layers, each characterised by a constant temperature lapse rate. Table B.5 lists the layer parameters as encoded in `src/atmosphere.rs`.

Table B.5: ICAO Standard Atmosphere layers (`src/atmosphere.rs`).

| Layer | Name | Base Alt (m) | Base Temp (K) | Base Press (Pa) | Lapse Rate |
|---|---|---|---|---|---|
| 0 | Troposphere | 0 | 288.15 | 101 325.0 | −6.5 K/km |
| 1 | Tropopause | 11 000 | 216.65 | 22 632.1 | 0.0 (isothermal) |
| 2 | Stratosphere 1 | 20 000 | 216.65 | 5474.89 | +1.0 K/km |
| 3 | Stratosphere 2 | 32 000 | 228.65 | 868.02 | +2.8 K/km |
| 4 | Stratopause | 47 000 | 270.65 | 110.91 | 0.0 (isothermal) |
| 5 | Mesosphere 1 | 51 000 | 270.65 | 66.94 | −2.8 K/km |
| 6 | Mesosphere 2 | 71 000 | 214.65 | 3.96 | −2.0 K/km |

Within each layer, temperature varies linearly with altitude:

$$T(h) = T_b + L_b\,(h - h_b) \tag{B.2}$$

where $T_b$ is the base temperature, $L_b$ the lapse rate, and $h_b$ the base altitude of the layer. Pressure is then obtained from the barometric formula. For layers with a non-zero lapse rate ($L_b \neq 0$):

$$P(h) = P_b \left( \frac{T(h)}{T_b} \right)^{-\,g_0 / (L_b\, R^*)} \tag{B.3}$$

and for isothermal layers ($L_b = 0$):

$$P(h) = P_b \exp\!\left( \frac{-\,g_0\,(h - h_b)}{R^*\,T_b} \right) \tag{B.4}$$

Here $g_0 = 9.806\,65\ \mathrm{m/s^2}$ and $R^* = 287.0531\ \mathrm{J/(kg \cdot K)}$ (specific gas constant for dry air).

The speed of sound in dry air follows from

$$c = \sqrt{\gamma\, R^{*}\, T} \tag{B.5}$$

with $\gamma = 1.4$. At sea level this yields $c = 340.29\,\mathrm{m/s}$.

## B.5 Air Density Calculation Constants

When humidity is nonzero, BALLISTICS-ENGINE uses the CIPM[1] humid-air-density formula, which requires gas constants, molar masses, vapor-pressure models, and compressibility-factor coefficients. All values below are defined in `src/atmosphere.rs`.

### B.5.1 Gas Constants and Molar Masses

Table B.6: Fundamental gas constants and molar masses.

| Constant | Value | Description |
|---|---|---|
| `R` | $8.314\,472\,\mathrm{J/(mol\cdot K)}$ | Universal gas constant |
| `R_DRY` | $287.05\,\mathrm{J/(kg\cdot K)}$ | Gas constant for dry air |
| `R_VAPOR` | $461.495\,\mathrm{J/(kg\cdot K)}$ | Gas constant for water vapor |
| `M_A` | $28.965\,46\times10^{-3}\,\mathrm{kg/mol}$ | Molar mass of dry air |
| `M_V` | $18.015\,28\times10^{-3}\,\mathrm{kg/mol}$ | Molar mass of water vapor |

### B.5.2 Saturation Vapor Pressure

Two models are used depending on the code path. The simplified **Arden Buck equations** are applied in the standard atmosphere routine:

$$e_s = 6.1121\,\exp\!\left(\frac{(18.678 - T/234.5)\,T}{257.14 + T}\right) \qquad \text{(over water, } T \geq 0\,^{\circ}\mathrm{C}\text{)} \tag{B.6}$$

$$e_s = 6.1115\,\exp\!\left(\frac{(23.036 - T/333.7)\,T}{279.82 + T}\right) \qquad \text{(over ice, } T < 0\,^{\circ}\mathrm{C}\text{)} \tag{B.7}$$

where $e_s$ is in hPa and $T$ is in $^{\circ}\mathrm{C}$.

For the high-precision CIPM density calculation, the **IAPWS-IF97** formulation is used:

[1] Comité International des Poids et Mesures.

$$\ln\!\left(\frac{p_{sv}}{p_c}\right) = \frac{T_c}{T}\sum_{i=1}^{6} a_i\,\tau^{\,n_i}, \qquad \tau = 1 - \frac{T}{T_c} \tag{B.8}$$

with $T_c = 647.096\,\mathrm{K}$ (critical temperature of water), $p_c = 220\,640\,\mathrm{hPa}$ (22.064 MPa, critical pressure), and the coefficients listed in Table B.7.

Table B.7: IAPWS-IF97 saturation vapor pressure coefficients.

| **Index** | **Coefficient** $a_i$ | **Exponent** $n_i$ |
|---|---|---|
| 1 | $-7.85951783$ | $1.0$ |
| 2 | $+1.84408259$ | $1.5$ |
| 3 | $-11.7866497$ | $3.0$ |
| 4 | $+22.6807411$ | $3.5$ |
| 5 | $-15.9618719$ | $4.0$ |
| 6 | $+1.80122502$ | $7.5$ |

### B.5.3 Enhancement Factor

The enhancement factor $f$ accounts for the non-ideal behaviour of moist air at elevated pressures:

$$f = \alpha + \beta\,p + \gamma\,T^2 + \delta\,p\,T \tag{B.9}$$

where $p$ is pressure in hPa and $T$ is temperature in °C. The coefficients are given in Table B.8.

Table B.8: Enhancement factor constants.

| **Coefficient** | **Value** |
|---|---|
| $\alpha$ (`ALPHA`) | $1.00062$ |
| $\beta$ (`BETA`) | $3.14 \times 10^{-8}$ |
| $\gamma$ (`GAMMA_EF`) | $5.6 \times 10^{-7}$ |
| $\delta$ (`DELTA`) | $1.2 \times 10^{-10}$ |

### B.5.4 Compressibility Factor (Virial Coefficients)

The compressibility factor $Z$ corrects for real-gas behaviour and is computed from a virial expansion:

$$Z = 1 \;-\; \frac{p}{T}\left[A_0 + A_1 t + A_2 t^2 + (B_0 + B_1 t)\,x_v + (C_0 + C_1 t)\,x_v^2\right] \\ +\; \frac{p^2}{T^2}\left[D + E\,x_v^2\right] \;+\; \frac{p^3}{T^3}\left[F_0 + F_1\,x_v^3\right] \tag{B.10}$$

where $p$ is pressure (hPa), $T$ is absolute temperature (K), $t = T - 273.15$ (°C), and $x_v$ is the mole fraction of water vapor. The virial coefficients are listed in Table B.9.

Table B.9: Virial coefficients for the compressibility factor (`src/atmosphere.rs`).

| Coeff. | Value | Coeff. | Value |
|---|---|---|---|
| $A_0$ | $1.58123 \times 10^{-6}$ | $C_0$ | $1.9898 \times 10^{-4}$ |
| $A_1$ | $-2.9331 \times 10^{-8}$ | $C_1$ | $-2.376 \times 10^{-6}$ |
| $A_2$ | $1.1043 \times 10^{-10}$ | $D$ | $1.83 \times 10^{-11}$ |
| $B_0$ | $5.707 \times 10^{-6}$ | $E$ | $-0.765 \times 10^{-8}$ |
| $B_1$ | $-2.051 \times 10^{-8}$ | $F_0$ | $2.1 \times 10^{-12}$ |
| | | $F_1$ | $-1.1 \times 10^{-14}$ |

The CIPM density is then:

$$\rho = \frac{p\,M_a}{Z\,R\,T}\left(1 - x_v\left(1 - \frac{M_v}{M_a}\right)\right) \tag{B.11}$$

## B.6 Numerical Stability Constants

Floating-point arithmetic introduces unavoidable rounding, and the trajectory solver must guard against division by zero and convergence failures. Table B.10 lists the thresholds and tolerances defined in `src/constants.rs`.

Table B.10: Numerical stability constants (`src/constants.rs`).

| Constant | Value | Purpose |
|---|---|---|
| `NUMERICAL_TOLERANCE` | $10^{-9}$ | General floating-point comparison |
| `MIN_VELOCITY_THRESHOLD` | $10^{-6}$ | Minimum velocity magnitude (m/s) |
| `MIN_DIVISION_THRESHOLD` | $10^{-12}$ | Guard against division by zero |
| `ROOT_FINDING_TOLERANCE` | $10^{-6}$ | Convergence criterion for root-finding |
| `MIN_MACH_THRESHOLD` | $10^{-3}$ | Clamp for Mach number near unity |

### Interpreting the thresholds

`MIN_VELOCITY_THRESHOLD` prevents the solver from dividing by a near-zero velocity vector when computing drag direction. If the velocity magnitude falls below $10^{-6}\ \mathrm{m/s}$, the projectile is considered stopped and the integration terminates. `MIN_MACH_THRESHOLD` avoids singularities in drag-coefficient lookup tables near Mach 1 by clamping the Mach number away from exactly $1.0$.

# Appendix C

# Drag Tables

Every trajectory computation in BALLISTICS-ENGINE begins with a drag table: a mapping from Mach number to drag coefficient ($C_D$) for a specific standard reference projectile. When a user specifies a ballistic coefficient and a drag model (G1, G7, etc.), the engine looks up the drag coefficient at the projectile's current Mach number, then scales it by the ratio of the standard projectile's sectional density to the actual projectile's sectional density. The result is the instantaneous drag deceleration applied at each integration step.

The tables in this appendix are the exact values compiled into `src/drag_tables.rs`. At runtime, the engine loads higher-resolution data from NumPy binary files or CSV files when available; the compiled tables serve as fallback data. For the G1 and G7 models, the fallback tables are themselves high-resolution—79 and 84 data points, respectively—derived from measured drag data. The remaining models (G2, G5, G6, G8, GI, GS) currently carry placeholder fallback data in `src/drag_tables.rs`, though production-quality tables for G6 and G8 are available in `src/drag.rs` and are loaded at runtime when the drag table data files are present.

Between tabulated points, the engine uses Catmull–Rom cubic spline interpolation when four surrounding points are available, and falls back to linear interpolation at the edges of the table. This approach preserves the smooth, continuous drag curve that the trajectory integrator requires (see Section C.4).

**Reading These Tables**

The drag coefficient $C_D$ listed here is the drag coefficient of the *standard reference projectile* for each model, not the drag coefficient of any particular bullet. A bullet's actual drag is obtained by dividing the standard $C_D$ by the bullet's ballistic coefficient. See Chapter 11 for a full treatment of this relationship.

## C.1 G1 Standard Drag Model

The G1 drag model is the oldest and most widely used standard in commercial ballistics. Its reference projectile is a flat-base design with a 2-caliber tangent ogive nose—a blunt, utilitarian shape that broadly approximates many hunting and military bullets manufactured over the past century.

Because the G1 reference projectile has a flat base and relatively blunt nose, its drag coefficient is higher than that of boat-tail designs across all velocity regimes. The G1 curve rises steeply through the transonic zone (Mach 0.8–1.2), peaks near Mach 1.4 at $C_D \approx 0.6625$, then declines gradually through the supersonic regime before asymptoting near $C_D \approx 0.50$ at high Mach numbers.

The G1 model remains the industry default for published ballistic coefficients. Most bullet manufacturers list G1 BCs, and most commercial ballistic calculators assume G1 unless told otherwise. For flat-base or short boat-tail bullets fired at moderate ranges, G1 produces adequate predictions. For long, sleek boat-tail match bullets at extended range, the G7 model (Section C.2) is a better choice.

Table C.1 lists all 79 data points of the G1 drag table as compiled in `src/drag_tables.rs`.

Table C.1: G1 Standard Drag Table — 79 data points.

| Mach | $C_D$ | Mach | $C_D$ | Mach | $C_D$ |
|---|---|---|---|---|---|
| 0.000 | 0.2629 | 0.925 | 0.3734 | 1.700 | 0.6347 |
| 0.050 | 0.2558 | 0.950 | 0.4084 | 1.750 | 0.6280 |
| 0.100 | 0.2487 | 0.975 | 0.4448 | 1.800 | 0.6210 |
| 0.150 | 0.2413 | 1.000 | 0.4805 | 1.850 | 0.6141 |
| 0.200 | 0.2344 | 1.025 | 0.5136 | 1.900 | 0.6072 |
| 0.250 | 0.2278 | 1.050 | 0.5427 | 1.950 | 0.6003 |
| 0.300 | 0.2214 | 1.075 | 0.5677 | 2.000 | 0.5934 |
| 0.350 | 0.2155 | 1.100 | 0.5883 | 2.050 | 0.5867 |
| 0.400 | 0.2104 | 1.125 | 0.6053 | 2.100 | 0.5804 |
| 0.450 | 0.2061 | 1.150 | 0.6191 | 2.150 | 0.5743 |
| 0.500 | 0.2032 | 1.200 | 0.6393 | 2.200 | 0.5685 |
| 0.550 | 0.2020 | 1.250 | 0.6518 | 2.250 | 0.5630 |
| 0.600 | 0.2034 | 1.300 | 0.6589 | 2.300 | 0.5577 |
| 0.700 | 0.2165 | 1.350 | 0.6621 | 2.350 | 0.5527 |
| 0.725 | 0.2230 | 1.400 | 0.6625 | 2.400 | 0.5481 |
| 0.750 | 0.2313 | 1.450 | 0.6607 | 2.450 | 0.5438 |
| 0.775 | 0.2417 | 1.500 | 0.6573 | 2.500 | 0.5397 |
| 0.800 | 0.2546 | 1.550 | 0.6528 | 2.600 | 0.5325 |
| 0.825 | 0.2706 | 1.600 | 0.6474 | 2.700 | 0.5264 |

*Continued on next page*

Table C.1: G1 Standard Drag Table (continued)

| Mach | $C_D$ | Mach | $C_D$ | Mach | $C_D$ |
|---|---|---|---|---|---|
| 0.850 | 0.2901 | 1.650 | 0.6413 | 2.800 | 0.5211 |
| 0.875 | 0.3136 | | | 2.900 | 0.5168 |
| 0.900 | 0.3415 | | | 3.000 | 0.5133 |
| | | | | 3.100 | 0.5105 |
| | | | | 3.200 | 0.5084 |
| | | | | 3.300 | 0.5067 |
| | | | | 3.400 | 0.5054 |
| | | | | 3.500 | 0.5040 |
| | | | | 3.600 | 0.5030 |
| | | | | 3.700 | 0.5022 |
| | | | | 3.800 | 0.5016 |
| | | | | 3.900 | 0.5010 |
| | | | | 4.000 | 0.5006 |
| | | | | 4.200 | 0.4998 |
| | | | | 4.400 | 0.4995 |
| | | | | 4.600 | 0.4992 |
| | | | | 4.800 | 0.4990 |
| | | | | 5.000 | 0.4988 |

Several features of the G1 curve are worth noting:

- The subsonic drag minimum occurs near Mach 0.55, where $C_D = 0.2020$.
- The transonic rise begins at approximately Mach 0.6 and accelerates sharply above Mach 0.8 as shock waves form on the projectile.
- The drag coefficient peaks at Mach 1.400 with $C_D = 0.6625$—more than three times the subsonic minimum.
- Above Mach 1.4, the curve declines monotonically as the bow shock stabilizes and wave drag diminishes.
- At very high Mach numbers (above 4.0), the curve flattens to an asymptotic value near $C_D \approx 0.499$.

## C.2 G7 Standard Drag Model

The G7 drag model represents a very low drag (VLD) boat-tail projectile with a long, 10-caliber secant ogive nose. This shape closely matches modern long-range match bullets such as the Sierra MatchKing, Berger Hybrid, and similar designs that dominate precision rifle competition.

The G7 curve differs from G1 in several fundamental ways. Subsonic drag is substantially lower ($C_D \approx 0.119$ versus $C_D \approx 0.203$ for G1), reflecting the streamlined form and boat-tail base. The transonic rise is steeper and more abrupt—the G7 reference projectile exhibits a near-vertical drag increase between Mach 0.925 and Mach 1.025, where $C_D$ more than doubles from 0.1660 to 0.4015. The supersonic peak occurs near Mach 1.050 ($C_D = 0.4043$), considerably lower than the G1 peak, and the curve then declines steadily through the supersonic regime.

For modern boat-tail match bullets, the G7 model produces more accurate trajectory predictions than G1, particularly at extended range where the bullet decelerates through the transonic zone. When a bullet manufacturer publishes a G7 ballistic coefficient, it typically requires fewer velocity-band corrections than an equivalent G1 BC, because the G7 reference shape more closely matches the actual projectile.

Table C.2 lists all 84 data points of the G7 drag table as compiled in `src/drag_tables.rs`.

Table C.2: G7 Standard Drag Table — 84 data points.

| Mach | $C_D$ | Mach | $C_D$ | Mach | $C_D$ |
|---|---|---|---|---|---|
| 0.000 | 0.1198 | 1.000 | 0.3803 | 2.350 | 0.2779 |
| 0.050 | 0.1197 | 1.025 | 0.4015 | 2.400 | 0.2752 |
| 0.100 | 0.1196 | 1.050 | 0.4043 | 2.450 | 0.2725 |
| 0.150 | 0.1194 | 1.075 | 0.4034 | 2.500 | 0.2697 |
| 0.200 | 0.1193 | 1.100 | 0.4014 | 2.550 | 0.2670 |
| 0.250 | 0.1194 | 1.125 | 0.3987 | 2.600 | 0.2643 |
| 0.300 | 0.1194 | 1.150 | 0.3955 | 2.650 | 0.2615 |
| 0.350 | 0.1194 | 1.200 | 0.3884 | 2.700 | 0.2588 |
| 0.400 | 0.1193 | 1.250 | 0.3810 | 2.750 | 0.2561 |
| 0.450 | 0.1193 | 1.300 | 0.3732 | 2.800 | 0.2533 |
| 0.500 | 0.1194 | 1.350 | 0.3657 | 2.850 | 0.2506 |
| 0.550 | 0.1193 | 1.400 | 0.3580 | 2.900 | 0.2479 |
| 0.600 | 0.1194 | 1.500 | 0.3440 | 2.950 | 0.2451 |
| 0.650 | 0.1197 | 1.550 | 0.3376 | 3.000 | 0.2424 |
| 0.700 | 0.1202 | 1.600 | 0.3315 | 3.100 | 0.2368 |
| 0.725 | 0.1207 | 1.650 | 0.3260 | 3.200 | 0.2313 |
| 0.750 | 0.1215 | 1.700 | 0.3209 | 3.300 | 0.2258 |
| 0.775 | 0.1226 | 1.750 | 0.3160 | 3.400 | 0.2205 |
| 0.800 | 0.1242 | 1.800 | 0.3117 | 3.500 | 0.2154 |
| 0.825 | 0.1266 | 1.850 | 0.3078 | 3.600 | 0.2106 |
| 0.850 | 0.1306 | 1.900 | 0.3042 | 3.700 | 0.2060 |

*Continued on next page*

Table C.2: G7 Standard Drag Table (continued)

| Mach | $C_D$ | Mach | $C_D$ | Mach | $C_D$ |
|---|---|---|---|---|---|
| 0.875 | 0.1368 | 1.950 | 0.3010 | 3.800 | 0.2017 |
| 0.900 | 0.1464 | 2.000 | 0.2980 | 3.900 | 0.1975 |
| 0.925 | 0.1660 | 2.050 | 0.2951 | 4.000 | 0.1935 |
| 0.950 | 0.2054 | 2.100 | 0.2922 | 4.200 | 0.1861 |
| 0.975 | 0.2993 | 2.150 | 0.2892 | 4.400 | 0.1793 |
| | | 2.200 | 0.2864 | 4.600 | 0.1730 |
| | | 2.250 | 0.2835 | 4.800 | 0.1672 |
| | | 2.300 | 0.2807 | 5.000 | 0.1618 |

Key characteristics of the G7 curve:

- Subsonic drag is remarkably flat, hovering between $C_D = 0.1193$ and $C_D = 0.1198$ from Mach 0.0 through Mach 0.5. The boat-tail and long ogive produce minimal pressure drag at these velocities.
- The transonic rise is extremely steep. Between Mach 0.925 ($C_D = 0.1660$) and Mach 0.975 ($C_D = 0.2993$), the drag coefficient nearly doubles in a span of just Mach 0.05.
- Peak drag occurs at Mach 1.050 ($C_D = 0.4043$), which is 39% lower than the G1 peak.
- The supersonic decline is steady and nearly linear above Mach 1.5, reflecting the well-organized shock structure of a streamlined projectile.
- At Mach 5.0, the G7 coefficient ($C_D = 0.1618$) is roughly one-third of the G1 value at the same speed, underscoring the aerodynamic advantage of the boat-tail form at high velocity.

**G1 vs. G7 at Extended Range**

Using a G1 ballistic coefficient for a VLD boat-tail bullet will overestimate velocity retention at long range, because the G1 reference shape does not match the actual bullet's drag profile in the transonic regime. If a manufacturer provides a G7 BC, always prefer it for trajectory predictions beyond 600 yards (549 m).

## C.3 Other Standard Drag Models

In addition to G1 and G7, BALLISTICS-ENGINE recognizes six other standard drag model designators. These models cover a range of projectile geometries from pointed military bullets to spherical round balls. Table C.3 summarizes each model, its reference projectile, and the current implementation status.

All eight drag models can be selected via the `--drag-model` flag:

Table C.3: Additional drag models supported by BALLISTICS-ENGINE.

| Model | Origin | Reference Projectile | Status |
|---|---|---|---|
| G2 | Aberdeen | Pointed conical nose (Aberdeen J projectile). Used for certain military flechette and conical-nose designs. | Placeholder |
| G5 | BRL | Short 7.5° total-angle ogive with boat-tail. Represents a low-drag military profile with moderate boat-tail angle. | Placeholder |
| G6 | BRL | Flat-base, 6-caliber secant ogive. A compromise between the blunt G1 shape and more streamlined designs; common among military FMJ bullets. | Placeholder* |
| G8 | BRL | Flat-base, capped projectile similar to the .30 caliber M2 Ball. The reference shape has a 10-caliber secant ogive with a flat base. | Placeholder* |
| GI | Ingalls | Historical drag function predating the Gavre Commission standards. Based on the Mayevski/Ingalls tables from the 1880s. Retained for historical comparison. | Placeholder |
| GS | Spherical | Spherical (round ball). Used for shotgun slugs, muzzleloader round balls, and other spherical projectiles where form factor is unity. | Placeholder |

*Production-quality G6 and G8 tables are available in `src/drag.rs` and are loaded at runtime when the drag table data files are present. The placeholder entries in `src/drag_tables.rs` serve only as a compiled fallback.

Listing C.1: Selecting a drag model

```
ballistics trajectory \
  --diameter 0.308 --mass 168 --bc 0.462 \
  --velocity 2700 --auto-zero 200 --max-range 1000 \
  --drag-model g7
```

The drag model name is case-insensitive; `g7`, `G7`, and `G7` are all equivalent, as handled by the `from_str` method in `src/drag_model.rs`.

**Placeholder Data**

The placeholder fallback tables for G2, G5, G6 (fallback only), G8 (fallback only), GI, and GS each contain only six coarse data points. These are sufficient for the engine to run without crashing, but they do not represent physically accurate drag curves. When using these models, ensure that the runtime drag table data files are present, or supply a custom drag table (see Section C.5).

## C.4 Interpolation Method

The drag tables listed above are discrete: they provide $C_D$ values at specific Mach numbers. During trajectory integration, the projectile's Mach number will rarely coincide exactly with a tabulated value, so the engine must interpolate between entries.

The interpolation logic resides in the `DragTable::interpolate` method in `src/drag.rs` and proceeds as follows:

1. **Exact boundary check.** If the query Mach number falls at or below the first tabulated value, or at or above the last, the engine performs linear extrapolation from the nearest two points. The result is clamped to a minimum of $0.01$ to prevent non-physical negative drag coefficients.
2. **Segment search.** The engine performs a linear scan to find the table segment $[M_i,\ M_{i+1}]$ containing the query Mach number.
3. **Cubic interpolation (interior points).** When both a preceding point ($M_{i-1}$) and a following point ($M_{i+2}$) are available, the engine applies Catmull–Rom spline interpolation using the four surrounding data points. Given the normalized parameter

$$t = \frac{M - M_i}{M_{i+1} - M_i} \tag{C.1}$$

the interpolated value is computed as:

$$C_D(M) = a_0\, t^3 + a_1\, t^2 + a_2\, t + a_3 \tag{C.2}$$

where the coefficients are:

$$\begin{aligned} a_0 &= -\tfrac{1}{2}\,y_{i-1} + \tfrac{3}{2}\,y_i - \tfrac{3}{2}\,y_{i+1} + \tfrac{1}{2}\,y_{i+2} \\ a_1 &= y_{i-1} - \tfrac{5}{2}\,y_i + 2\,y_{i+1} - \tfrac{1}{2}\,y_{i+2} \\ a_2 &= -\tfrac{1}{2}\,y_{i-1} + \tfrac{1}{2}\,y_{i+1} \\ a_3 &= y_i \end{aligned} \tag{C.3}$$

and $y_k = C_D(M_k)$ denotes the tabulated drag coefficient at point $k$.

4. **Linear interpolation (edge segments).** For the first and last segments of the table, where four surrounding points are not available, the engine falls back to linear interpolation:

$$C_D(M) = y_i + \frac{y_{i+1} - y_i}{M_{i+1} - M_i}\,(M - M_i) \tag{C.4}$$

The Catmull–Rom scheme preserves $C^1$ continuity across segment boundaries (the first derivative is continuous), which prevents artificial "kinks" in the drag curve that could introduce numerical noise into the trajectory integrator. Linear interpolation at the edges is acceptable because the drag curve is nearly flat in those regions (low subsonic and high supersonic).

**Why Not Pure Linear Interpolation?**

Linear interpolation between table entries would produce a piecewise-linear drag curve with slope discontinuities at every tabulated Mach number. In the transonic region, where the drag coefficient changes rapidly, these discontinuities can cause step-size oscillations in the RK4 integrator. The Catmull–Rom spline eliminates this issue at negligible computational cost.

## C.5 Custom Drag Tables

While the built-in G1 and G7 tables cover the vast majority of use cases, advanced users may wish to supply their own drag data. This is particularly valuable when:

- Doppler-derived drag measurements are available for the specific bullet being modeled, providing a drag-versus-Mach profile that matches the actual projectile rather than a standard reference shape.
- The projectile has an unusual geometry (saboted, fin-stabilized, sub-caliber) that does not conform to any standard drag model.
- Research or development work requires testing hypothetical drag curves.

BALLISTICS-ENGINE supports custom drag tables through the `custom_drag_table` field of the `BallisticInputs` struct defined in `src/cli_api.rs`. This field accepts an `Option<DragTable>`—the same `DragTable` structure used internally for the standard models. When a custom drag table is provided, it overrides

the standard model lookup entirely; the engine interpolates directly from the user-supplied Mach-versus-$C_D$ data using the same Catmull–Rom / linear interpolation logic described in Section C.4.

A custom drag table is constructed by providing two parallel vectors: one of Mach numbers (in ascending order) and one of corresponding $C_D$ values. There is no minimum or maximum number of data points, though tables with fewer than four entries will use linear interpolation exclusively (since Catmull–Rom requires four surrounding points).

Listing C.2: Constructing a custom drag table

```
use ballistics_engine::drag::DragTable;

let custom_table = DragTable::new(
    vec![0.0, 0.5, 0.8, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 3.0],
    vec![0.12, 0.12, 0.13, 0.18, 0.38, 0.40, 0.39, 0.34, 0.30, 0.24],
);
```

**Custom Table Quality**

The accuracy of any trajectory prediction is bounded by the accuracy of the drag data. Custom drag tables derived from doppler radar measurements should include sufficient data density in the transonic region (Mach 0.8–1.2), where drag changes most rapidly. A table with coarse spacing in this region will produce inaccurate transonic predictions regardless of the integrator's precision.

The engine also supports loading drag tables from external NumPy binary files (`.npy`) or CSV files at runtime via the `load_drag_table` function in `src/drag.rs`. This mechanism is used internally to load the high-resolution production drag tables and can be leveraged for custom data as well. The expected format for CSV files is two columns: Mach number and $C_D$, one pair per row, with an optional header line.

# Appendix D

# BC5D Table Format

This appendix provides a complete specification of the BC5D binary table format used by BALLISTICS-ENGINE for offline, caliber-specific ballistic coefficient corrections. All byte offsets, data types, and algorithmic details are drawn directly from the implementation in `src/bc_table_5d.rs`, `src/bc_table.rs`, and `src/bc_table_download.rs`. Implementors writing parsers, validators, or generation tools for BC5D files should treat this appendix as the authoritative reference.

## D.1 Overview

Published ballistic coefficients are single-point values that assume a specific set of flight conditions. In practice, the effective BC of a projectile varies with velocity, weight class, muzzle velocity, and drag model. The BC5D table system bridges this gap by storing precomputed correction factors derived from machine-learning model predictions across a dense, five-dimensional parameter space.

Each BC5D file is caliber-specific and encodes a five-dimensional grid of correction factors indexed by:

1. **Drag type** — discrete: G1 or G7.
2. **Bullet weight** — continuous: weight in grains.
3. **Base BC** — continuous: the published BC value.
4. **Muzzle velocity** — continuous: initial launch velocity in feet per second.
5. **Current velocity** — continuous: in-flight velocity in feet per second, with dense sampling through the transonic region.

At runtime, BALLISTICS-ENGINE performs a 4D linear interpolation (the drag type dimension is discrete and not interpolated) to retrieve a correction factor for the exact flight conditions. The effective BC is then:

$$\mathrm{BC}_{\mathrm{eff}} = \mathrm{BC}_{\mathrm{base}} \times f_{\mathrm{correction}} \tag{D.1}$$

where $f_{\mathrm{correction}}$ is the interpolated value from the table, clamped to the range $[0.5, 1.5]$.

This approach provides the accuracy of a full ML model prediction at a fraction of the computational cost, requiring no network access and no floating-point model inference at runtime.

## D.2 BC5D Binary Header Format

A BC5D file begins with a fixed 80-byte header, followed by variable-length bin definitions and correction data. All multi-byte fields use little-endian byte order. The current format version is 2.

Table D.1: BC5D header layout (80 bytes, `src/bc_table_5d.rs`).

| Offset | Size | Type | Field | Description |
|---|---|---|---|---|
| 0 | 4 | bytes | Magic | File signature: `BC5D` (`0x42 0x43 0x35 0x44`) |
| 4 | 4 | uint32 LE | Version | Format version (currently 2) |
| 8 | 4 | float32 LE | Caliber | Caliber in inches (e.g., 0.308) |
| 12 | 4 | uint32 LE | Flags | Reserved flags field |
| 16 | 4 | uint32 LE | Padding | Reserved padding |
| 20 | 4 | uint32 LE | dim_weight | Number of weight bins |
| 24 | 4 | uint32 LE | dim_bc | Number of BC bins |
| 28 | 4 | uint32 LE | dim_muzzle_vel | Number of muzzle velocity bins |
| 32 | 4 | uint32 LE | dim_current_vel | Number of current velocity bins |
| 36 | 4 | uint32 LE | dim_drag_types | Number of drag types (typically 2) |
| 40 | 8 | uint64 LE | timestamp | Unix timestamp of table generation |
| 48 | 4 | uint32 LE | checksum | CRC32 of data section |
| 52 | 16 | string | api_version | Null-padded API version string |
| 68 | 12 | bytes | reserved | Reserved for future use |

The magic bytes uniquely identify the file format and allow quick rejection of non-BC5D files during loading. The version field enables forward-compatible evolution of the format; the loader in BALLISTICS-ENGINE rejects any version other than 2. The flags and padding fields are reserved for future use and must be set to zero in current files.

A hex dump of a valid header might appear as follows:

Listing D.1: Example BC5D header (first 20 bytes in hex).

```
42 43 35 44   02 00 00 00   9A 99 9D 3E
00 00 00 00   00 00 00 00
```

In this example, the magic bytes spell `BC5D`, the version is 2 (`0x00000002`), and the caliber field encodes the IEEE 754 float32 representation of 0.308 (`0x3E9D999A` in big-endian, stored little-endian as `9A 99 9D 3E`).

## D.3 Five-Dimensional Bin Layout

Immediately following the 80-byte header, four arrays of float32 values define the bin edges for each continuous dimension. Each array consists of monotonically increasing values stored as little-endian IEEE 754 single-precision floats (4 bytes each).

### D.3.1 Bin Definitions

The bins are stored in the following order with no padding between arrays:

Table D.2: Bin definition arrays following the header.

| Array | Count Field | Size (bytes) | Unit / Range |
|---|---|---|---|
| Weight bins | dim_weight | `dim_weight` $\times$ 4 | Grains; caliber-specific |
| BC bins | dim_bc | `dim_bc` $\times$ 4 | Dimensionless; 0.05–1.2 |
| Muzzle vel. bins | dim_muzzle_vel | `dim_muzzle_vel` $\times$ 4 | ft/s; 2,000–4,000 |
| Current vel. bins | dim_current_vel | `dim_current_vel` $\times$ 4 | ft/s; 500–4,000 |

The current velocity bins are sampled more densely in the transonic region (approximately 900–1,200 ft/s) where drag coefficients change most rapidly and BC corrections are most significant.

### D.3.2 Dimension Summary

The five dimensions of the table are summarised in Table D.3. The drag type dimension is discrete and is not accompanied by a bin array; its index is determined directly from the drag model string (0 for G1, 1 for G7).

Table D.3: The five dimensions of a BC5D table.

| Dim. | Name | Type | Typical Range |
|---|---|---|---|
| 0 | Drag type | Discrete | 0 = G1, 1 = G7 |
| 1 | Weight | Continuous | Caliber-specific (e.g., 100–230 gr) |
| 2 | Base BC | Continuous | 0.05–1.2 |
| 3 | Muzzle velocity | Continuous | 2,000–4,000 ft/s |
| 4 | Current velocity | Continuous | 500–4,000 ft/s |

### D.3.3 Data Section and Memory Layout

The correction factor data follows the bin arrays. The total number of cells is the product of all five dimension sizes:

$$N_{\text{cells}} = n_{\text{drag}} \times n_{\text{weight}} \times n_{\text{bc}} \times n_{\text{muzzle}} \times n_{\text{current}} \tag{D.2}$$

Each cell is a single float32 (4 bytes, little-endian) representing a correction factor—the ratio of predicted effective BC to the base published BC. Valid correction factors lie in the range $[0.5, 1.5]$; values outside this range are clamped during table generation.

The data is stored in row-major (C) order. The flat index for a given set of dimension indices is computed as:

$$\begin{aligned} \text{index} = {} & i_{\text{drag}} \cdot (n_w \cdot n_b \cdot n_m \cdot n_c) \\ & + i_{\text{weight}} \cdot (n_b \cdot n_m \cdot n_c) \\ & + i_{\text{bc}} \cdot (n_m \cdot n_c) \\ & + i_{\text{muzzle}} \cdot n_c \\ & + i_{\text{current}} \end{aligned} \tag{D.3}$$

where $n_w, n_b, n_m, n_c$ are the sizes of the weight, BC, muzzle velocity, and current velocity dimensions, respectively.

**Row-major ordering**

The innermost (fastest-varying) dimension is current velocity. This means that all correction factors for a given drag type, weight, BC, and muzzle velocity are stored contiguously, which is cache-friendly for the most common access pattern: sweeping through velocities along a single trajectory.

### D.3.4 Total File Size

The total file size in bytes can be computed from the header as:

$$\begin{aligned} S = {} & 80 \\ & + 4 \times (n_{\text{weight}} + n_{\text{bc}} + n_{\text{muzzle}} + n_{\text{current}}) \\ & + 4 \times N_{\text{cells}} \end{aligned} \tag{D.4}$$

where the first term is the fixed header, the second term accounts for the bin definition arrays, and the third term is the correction data.

## D.4 4D Linear Interpolation

Because the drag type dimension is discrete, interpolation is performed only over the four continuous dimensions: weight, BC, muzzle velocity, and current velocity. The algorithm constructs a four-dimensional hypercube around the query point and computes a weighted average of the $2^4 = 16$ corner values.

### D.4.1 Finding Bracketing Indices

For each continuous dimension, the lookup procedure performs a binary search on the sorted bin array to find the interval $[b_k, b_{k+1}]$ that brackets the query value $x$. Three cases arise:

1. **In range**: $b_0 < x < b_{N-1}$. Binary search yields the lower index $k$ and the interpolation weight $w = (x - b_k)/(b_{k+1} - b_k)$.
2. **Below range**: $x \leq b_0$. The index is clamped to $k = 0$ with weight $w = 0$.
3. **Above range**: $x \geq b_{N-1}$. The index is clamped to $k = N - 2$ with weight $w = 1$.

This clamping-to-edge strategy ensures that extrapolation requests gracefully degrade to the nearest available data rather than producing errors.

### D.4.2 Hypercube Interpolation

Let $(k_w, w_w), (k_b, w_b), (k_m, w_m), (k_c, w_c)$ denote the index and interpolation weight for the weight, BC, muzzle velocity, and current velocity dimensions. The interpolated correction factor for drag type index $d$ is:

$$f = \sum_{\delta_w=0}^{1} \sum_{\delta_b=0}^{1} \sum_{\delta_m=0}^{1} \sum_{\delta_c=0}^{1} W(\delta_w, \delta_b, \delta_m, \delta_c) \cdot T\big[d,\ k_w{+}\delta_w,\ k_b{+}\delta_b,\ k_m{+}\delta_m,\ k_c{+}\delta_c\big] \quad \text{(D.5)}$$

where $T[\cdots]$ denotes the table value at the given indices (with index clamping to the last valid bin) and the corner weight is the product of per-dimension weights:

$$W(\delta_w, \delta_b, \delta_m, \delta_c) = \hat{w}(\delta_w, w_w) \cdot \hat{w}(\delta_b, w_b) \cdot \hat{w}(\delta_m, w_m) \cdot \hat{w}(\delta_c, w_c) \quad \text{(D.6)}$$

with the per-axis weight function:

$$\hat{w}(\delta, w) = \begin{cases} 1 - w & \text{if } \delta = 0, \\ w & \text{if } \delta = 1. \end{cases} \quad \text{(D.7)}$$

The sum of all 16 corner weights is guaranteed to equal 1 (a partition of unity), so the interpolated result is a convex combination of the corner values. The final result is clamped to $[0.5, 1.5]$.

**Interpolation complexity**

Each lookup requires exactly 16 table reads and 16 multiply-accumulate operations—a trivial cost even at very high trajectory step rates. The binary search for each dimension adds $O(\log N)$ overhead per dimension, but since bin counts are typically in the tens to low hundreds, this is negligible.

## D.5 Checksum Verification (CRC32)

Each BC5D file contains a CRC32 checksum in the header that protects the integrity of all data following the header. The checksum uses the IEEE 802.3 polynomial in reflected (LSB-first) form.

### D.5.1 Algorithm

The CRC32 implementation uses the standard reflected algorithm:

1. Initialise the CRC register to `0xFFFFFFFF`.
2. For each byte of the input data, XOR it with the low byte of the CRC, use the result as an index into a 256-entry lookup table, and XOR the table entry with the CRC shifted right by 8 bits.
3. Invert (bitwise NOT) the final CRC register.

The lookup table is generated from the IEEE polynomial `0xEDB88320` (the bit-reversed form of `0x04C11DB7`).

### D.5.2 Data Covered by the Checksum

The checksum is computed over the concatenation of all bin arrays and the entire correction data section, serialised as little-endian float32 bytes:

$$\mathrm{CRC32}\big(\, weight_bins \parallel bc_bins \parallel muzzle_vel_bins \parallel current_vel_bins \parallel correction_data \,\big) \tag{D.8}$$

The header itself is *not* included in the checksum, as the checksum field resides within the header and would create a circular dependency.

### D.5.3 Verification Procedure

When loading a BC5D file, the loader performs the following steps:

1. Read and validate the header (magic, version, dimensions).
2. Read all bin arrays and correction data into memory.
3. Serialise the bin arrays and data back to little-endian bytes.
4. Compute CRC32 over the serialised bytes.
5. Compare the computed checksum against the stored `checksum` field in the header.
6. If the checksums do not match, reject the file with a `ChecksumMismatch` error.

**Known test vector**

The implementation can be verified against the standard CRC32 test vector: `CRC32("123456789") = 0xCBF43926`. This test is included in the unit test suite in `src/bc_table_5d.rs`.

## D.6 File Naming and Discovery

BC5D tables are caliber-specific. Each file covers a single caliber and is named according to a deterministic convention that allows the loader to locate the correct file without an external index.

### D.6.1 Naming Convention

The filename pattern is:

```
bc5d_{caliber_thousandths}.bin
```

where `caliber_thousandths` is the caliber in inches multiplied by 1,000 and rounded to the nearest integer. Table D.4 lists representative examples.

Table D.4: BC5D filename examples.

| Caliber (in) | Key | Filename |
|---|---|---|
| .224 | 224 | `bc5d_224.bin` |
| .264 (6.5 mm) | 264 | `bc5d_264.bin` |
| .308 | 308 | `bc5d_308.bin` |
| .338 | 338 | `bc5d_338.bin` |

### D.6.2 Caliber Key Calculation

The caliber key used for both filenames and in-memory table indexing is computed as:

$$\text{key} = \text{round}(\text{caliber_inches} \times 1000) \tag{D.9}$$

This integer key avoids floating-point comparison issues when matching caliber values to table files. For example, a user-supplied caliber of `0.308` yields a key of 308, matching the file `bc5d_308.bin`.

### D.6.3 Discovery Algorithm

Given a table directory and a caliber, the loader attempts to find a matching file in the following order:

1. Compute the caliber key and construct the primary filename `bc5d_{key}.bin`.
2. If found, return the path.
3. Try zero-padded variations: `bc5d_{key:03}.bin` and `bc5d_0{key}.bin`.
4. If no file matches, return a `TableNotFound` error.

The `available_calibers()` method enumerates all `.bin` files in the table directory whose names match the `bc5d_*` pattern, parses the integer suffix, and returns a sorted list of calibers.

## D.7 Table Download and Caching

When built with the `online` feature flag, BALLISTICS-ENGINE can automatically download BC5D tables from a remote server and cache them locally. The download system is implemented in `src/bc_table_download.rs`.

### D.7.1 Manifest Format

The server hosts a `manifest.json` file at the base URL that describes all available tables. The manifest follows this structure:

Listing D.2: BC5D manifest.json schema.

```
{
  "version": "2.1",
  "generated": "2025-01-15T12:00:00Z",
  "tables": {
    "308": {
      "file": "bc5d_308.bin",
      "size": 1048576,
      "crc32": "a1b2c3d4"
    },
    "224": {
      "file": "bc5d_224.bin",
      "size": 524288,
      "crc32": "e5f6a7b8"
    }
  }
```

```
}
```

Each entry in the `tables` object is keyed by caliber (as a string of thousandths of an inch) and contains the filename, file size in bytes, and a CRC32 checksum of the entire file as a hexadecimal string.

### D.7.2 Cache Directories

Downloaded tables are stored in a platform-specific cache directory. Table D.5 lists the default paths.

Table D.5: Default cache directories by platform.

| Platform | Cache Path |
|---|---|
| macOS | `~/Library/Caches/ballistics-engine/bc5d/` |
| Linux | `~/.cache/ballistics-engine/bc5d/` |
| Windows | `%LOCALAPPDATA%\ballistics-engine\cache\bc5d\` |

The implementation uses the `dirs` crate's `cache_dir()` function as the primary resolution mechanism. The platform-specific paths listed above serve as fallbacks when the crate cannot determine the cache directory.

### D.7.3 Download Flow

The download procedure for a given caliber proceeds as follows:

1. **Fetch manifest.** Download `manifest.json` from the base URL. The default base URL is `https://ballistics.tools/downloads/bc5d`. All HTTP requests use a 60-second timeout.
2. **Locate caliber entry.** Look up the caliber key (e.g., `"308"`) in the manifest's `tables` map. If the caliber is not present, return a `CaliberNotAvailable` error listing all available calibers.
3. **Check local cache.** If the expected file exists in the cache directory and the `--bc-table-refresh` flag is not set, compute the CRC32 of the cached file and compare it to the manifest's expected checksum. If they match, return the cached file path.
4. **Download.** Fetch the table file from the server. Read the entire response body into memory.
5. **Verify integrity.** Compute CRC32 of the downloaded data and compare against the manifest checksum. If they do not match, return a `ChecksumMismatch` error without writing the file.
6. **Write to cache.** Save the verified data to the cache directory.

### D.7.4 CLI Flags

Table D.6 lists the command-line flags that control BC5D table loading and downloading.

Table D.6: CLI flags for BC5D table management.

| Flag | Online? | Description |
|---|---|---|
| `--bc-table-dir` *DIR* | No | Use a local directory of pre-downloaded BC5D table files. |
| `--bc-table-auto` | Yes | Automatically download the required table when it is not found locally. |
| `--bc-table-url` *URL* | Yes | Override the default base URL for table downloads. |
| `--bc-table-refresh` | Yes | Force re-download of all tables, ignoring the local cache. |

Flags marked "Yes" in the Online column require `--features online` at build time. When `--bc-table-dir` is specified, the downloader is not used; tables are loaded directly from the given directory.

**Offline environments**

In air-gapped or offline environments, use `--bc-table-dir` to point to a directory of pre-provisioned BC5D files. The `--bc-table-auto` flag will fail with a network error if no internet access is available.

## D.8 The Bc5dTableManager

The `Bc5dTableManager` struct provides the primary runtime interface for loading, caching, and querying BC5D tables. It is defined in `src/bc_table_5d.rs`.

### D.8.1 In-Memory Caching

The manager maintains a `HashMap<i32, Bc5dTable>` that maps caliber keys to loaded tables. The first call to `get_table(caliber)` for a given caliber reads the file from disk, parses the header, validates the checksum, and stores the table in the hash map. Subsequent calls return a reference to the cached table with no I/O overhead.

### D.8.2 Key Methods

**`get_table(caliber: f64)`**
: Load or retrieve the cached table for the given caliber. Returns an error if no table file exists or if the file fails validation.

**`lookup(caliber, weight, bc, muzzle_vel, current_vel, drag_type)`**
Perform a full five-dimensional lookup and return the correction factor. Internally calls `get_table` followed by `Bc5dTable::lookup`.

**`get_effective_bc(...)`**
Convenience method that returns $\mathrm{BC}_{\mathrm{base}} \times f_{\mathrm{correction}}$—the corrected BC ready for use in trajectory computation.

**`available_calibers()`**
Scan the table directory and return a sorted list of calibers for which `.bin` files exist.

## D.9 Legacy BCCR Format

Prior to the introduction of BC5D, BALLISTICS-ENGINE used a single-file correction table with a different structure, identified by the magic bytes `BCCR`. The legacy format is implemented in `src/bc_table.rs` and remains loadable for backward compatibility.

Table D.7: BCCR header layout (64 bytes, `src/bc_table.rs`).

| Offset | Size | Type | Field | Description |
|---|---|---|---|---|
| 0 | 4 | bytes | Magic | `BCCR` (`0x42 0x43 0x43 0x52`) |
| 4 | 4 | uint32 LE | Version | Format version (1) |
| 8 | 4 | uint32 LE | Flags | Reserved |
| 12 | 4 | uint32 LE | num_bc | Number of BC bins |
| 16 | 4 | uint32 LE | num_mass | Number of mass bins |
| 20 | 4 | uint32 LE | num_length | Number of length bins |
| 24 | 4 | uint32 LE | num_velocity | Number of velocity bins |
| 28 | 4 | uint32 LE | num_bc_types | Number of BC types |
| 32 | 8 | uint64 LE | timestamp | Generation timestamp |
| 40 | 4 | uint32 LE | checksum | CRC32 of data section |
| 44 | 16 | bytes | reserved | Reserved |

The key differences between BCCR and BC5D are:

- **Dimensions.** BCCR uses *bc_type* × *bc* × *mass* × *length* × *velocity* (bullet length replaces muzzle velocity).
- **Caliber specificity.** BCCR uses a single file for all calibers, whereas BC5D uses per-caliber files.
- **Header size.** BCCR has a 64-byte header (no caliber field, no API version string).
- **Version.** BCCR is version 1; BC5D is version 2.

**Migration path**

BC5D supersedes BCCR for all new deployments. The legacy format is retained solely for backward compatibility with existing table files. No new BCCR tables are generated.

## D.10 Error Handling

The BC5D subsystem defines two error enumerations: `Bc5dError` for file-level operations and `Bc5dDownloadError` for network operations. Table D.8 and Table D.9 catalogue all variants.

Table D.8: Table-level errors (`Bc5dError` in `src/bc_table_5d.rs`).

| Variant | Meaning |
|---|---|
| `IoError` | Underlying I/O error (file not found, permission denied, etc.). |
| `InvalidMagic` | First 4 bytes are not `BC5D`. The file is not a valid BC5D table. |
| `UnsupportedVersion` | The version field does not equal 2. A newer or incompatible format was encountered. |
| `ChecksumMismatch` | The CRC32 computed over the bin and data sections does not match the value stored in the header. The file may be corrupt or truncated. Reports both expected and actual checksums. |
| `InvalidDimensions` | One or more dimension sizes in the header are zero, making the table unusable. |
| `TableNotFound` | No `.bin` file matching the requested caliber was found in the table directory. |
| `NoTableDirectory` | No table directory has been configured via `--bc-table-dir` or the download system. |

Table D.9: Download-level errors (`Bc5dDownloadError` in `src/bc_table_download.rs`).

| Variant | Meaning |
|---|---|
| `NetworkError` | A network-level failure occurred during the HTTP request (DNS resolution failure, connection refused, TLS error, etc.). |
| `Timeout` | The download did not complete within the 60-second timeout. |
| `IoError` | File system error when reading or writing the cache. |
| `ChecksumMismatch` | The CRC32 of the downloaded file does not match the expected value from the manifest. Reports both expected and actual checksums as hexadecimal strings. |
| `CaliberNotAvailable` | The requested caliber is not listed in the server manifest. The error message includes a list of all available calibers. |
| `ManifestParseError` | The manifest JSON could not be parsed, or a required field is missing. |
| `CacheDirectoryError` | The cache directory could not be created (permissions, disk full, etc.). |

# Appendix E

# Glossary

This glossary defines the key terms, abbreviations, and concepts used throughout this handbook and in the field of computational exterior ballistics. Entries are organised alphabetically in two groups and include cross-references where a term is closely related to another. For the physical constants behind many of these definitions, see Chapter B; for drag-model specifics, see Appendix C.

## E.1 Terms A–M

**Adiabatic Lapse Rate**
: The rate at which air temperature decreases with altitude in the absence of heat exchange with the surroundings. In the troposphere, the standard value is approximately $-6.5\,\mathrm{K/km}$. The lapse rate governs how temperature, pressure, and density change with elevation in the ICAO Standard Atmosphere model. See *ICAO Standard Atmosphere*, *Lapse Rate*.

**Air Density**
: Mass of air per unit volume, denoted $\rho$ and expressed in $\mathrm{kg/m^3}$. Varies with temperature, pressure, humidity, and altitude. The standard sea-level value is $1.225\,\mathrm{kg/m^3}$ (ICAO Standard Atmosphere). Air density is the single most important atmospheric variable in drag calculations: higher density increases aerodynamic drag, lower density decreases it. See *CIPM Formula*, *Density Altitude*.

**Angle of Departure (AoD)**
: The angle between the bore axis and the horizontal plane at the moment the projectile exits the muzzle. Not to be confused with the *shooting angle* (the angle at which the rifle is aimed), because muzzle rise during recoil causes the bore to move upward before the bullet clears the crown. In most computational models, including BALLISTICS-ENGINE, the angle of departure is assumed equal to the bore angle at the start of integration.

**Azimuth**

The horizontal direction of fire, measured clockwise from true north in degrees. Required for Coriolis and spin-drift calculations, where the deflection magnitude and direction depend on the compass heading of the shot. See *Coriolis Effect*, *Spin Drift*.

**Ballistic Coefficient (BC)**

A measure of a projectile's ability to overcome air resistance in flight. Defined as the ratio of sectional density to form factor:

$$\mathrm{BC} = \frac{SD}{i}$$

A higher BC indicates less drag and a flatter trajectory. A BC value is meaningful only when referenced to a specific drag model (G1, G7, etc.), because the form factor $i$ is computed relative to the standard reference projectile of that model. See *Sectional Density*, *Form Factor*, *Drag Model*, *Effective BC*.

**Ballistic Table**

A tabulated set of trajectory data—drop, windage, velocity, energy, and time of flight—at discrete range intervals for a specific load and atmospheric condition. The primary output format of BALLISTICS-ENGINE's `trajectory` command. See *Range Table*, *Dope Card*.

**BC5D Table**

A five-dimensional binary correction table used by BALLISTICS-ENGINE to apply ML-derived corrections to published ballistic coefficient values. The five dimensions are: drag model type, projectile weight, base BC, muzzle velocity, and current (in-flight) velocity. The table is stored in a compact binary format with CRC32 integrity verification. See *Ballistic Coefficient*, *Effective BC*, *CRC32*.

**Boat Tail**

A tapered base design on a bullet that reduces base drag by streamlining airflow separation at the projectile's rear. Boat-tail bullets typically have higher ballistic coefficients than their flat-base counterparts of similar weight and caliber. The G7 drag model is specifically designed for boat-tail projectile shapes. See *Flat-Base*, *G7 Drag Model*.

**Bore Height**

The vertical distance from the centre of the bore to the ground. Used by BALLISTICS-ENGINE for ground-impact detection during trajectory integration: when the projectile's height above ground falls to zero, the simulation terminates. Not to be confused with *sight height*.

**Bore Sight**

The optical axis of the bore, parallel to the barrel's centreline. Boresighting aligns the

riflescope's optical axis with the bore axis as a starting point before final zeroing at the range.

**CIPM Formula**
: The Comité International des Poids et Mesures formula for calculating air density from temperature, pressure, and humidity with high precision. Accounts for real-gas behaviour through a compressibility factor $Z$, the enhancement factor $f$, and accurate vapor-pressure models. Used by BALLISTICS-ENGINE for all atmospheric density calculations when humidity data is available. See *Air Density*.

**Click Value**
: The angular change per adjustment click on a riflescope turret. Typical values are $0.1$ MIL ($1$ cm at $100$ m) or $0.25$ MOA ($\approx 0.25$ in at 100 yards) per click. The `trajectory` command output includes come-up values in both MIL and MOA to match any turret system. See *MIL*, *MOA*.

**Coefficient of Drag ($C_D$)**
: A dimensionless number representing the drag force on a projectile as a function of its shape and Mach number. The relationship between drag force and $C_D$ is:

$$F_D = \tfrac{1}{2}\,\rho\, v^2\, C_D\, A$$

where $\rho$ is air density, $v$ is velocity, and $A$ is the reference (cross-sectional) area. In practice, $C_D$ is obtained by interpolating a standard drag table at the projectile's current Mach number and scaling by the ballistic coefficient. See *Drag Model*, *Form Factor*.

**Come-Up**
: The scope elevation adjustment, expressed in MIL or MOA, needed to hit a target at a given range relative to the zero range. Positive come-up means dialling "up" (increasing elevation); negative values indicate the bullet impacts above the line of sight and the scope must be dialled down. See *Elevation Adjustment*, *Zero*.

**Coriolis Effect**
: The apparent deflection of a projectile's path caused by Earth's rotation during the time of flight. The magnitude depends on latitude, azimuth (direction of fire), and time of flight. The horizontal (Eötvös) component deflects shots east or west; the vertical component raises or lowers the impact point. At typical rifle ranges below 500 yards, the Coriolis deflection is negligible; it becomes significant at extreme long range ($>$1000 yards) and is enabled in BALLISTICS-ENGINE with the `--coriolis` flag. See *Eötvös Effect*.

**CRC32**
: Cyclic Redundancy Check using the 32-bit polynomial specified by IEEE 802.3. Used by BALLISTICS-ENGINE to verify the integrity of BC5D table binary files: a CRC32 checksum is stored in the file header and validated on load. See *BC5D Table*.

**Crosswind**

The component of wind perpendicular to the line of fire, causing horizontal deflection (windage). A crosswind from the shooter's right is conventionally positive. See *Wind Deflection*, *Windage*.

**Density Altitude**

The altitude in the ICAO Standard Atmosphere that corresponds to the current actual air density. High density altitude (caused by high temperature, high humidity, or high elevation) means thinner air and less drag on the projectile, resulting in less drop at a given range. See *Air Density*, *ICAO Standard Atmosphere*.

**Doppler Radar**

A method of measuring projectile velocity throughout its flight using the Doppler shift of reflected radar signals. Provides the most accurate drag data for specific bullet designs, yielding a continuous $C_D$-vs-Mach curve rather than a single-point BC. Doppler-derived drag data underpins the most accurate trajectory predictions available. See *Coefficient of Drag*.

**Drag Coefficient**

See *Coefficient of Drag ($C_D$)*.

**Drag Model**

A standardised reference drag curve—$C_D$ as a function of Mach number—against which a specific bullet's performance is compared via the ballistic coefficient. Common models include G1 (flat-base reference), G7 (boat-tail/VLD reference), and several others (G2, G5, G6, G8, GI, GS). The choice of drag model significantly affects the accuracy of trajectory predictions, especially in the transonic region. See *G1 Drag Model*, *G7 Drag Model*.

**Dope Card**

A printed or digital reference card showing elevation and windage adjustments at various ranges for a specific rifle, load, and atmospheric condition. Typically generated from a ballistic table and carried in the field for quick reference. See *Ballistic Table*, *Come-Up*.

**Drop** The vertical distance a bullet falls below the bore line at a given range, measured in inches, centimetres, MIL, or MOA. Drop increases nonlinearly with range because gravity acts on the projectile for a longer time as velocity decreases. See *Trajectory*, *Come-Up*.

**Effective BC**

The ballistic coefficient after applying corrections for the current velocity regime, atmospheric conditions, or BC5D table adjustments. May differ substantially from the published (box) BC, particularly in the transonic region where drag behaviour diverges from the reference model. See *Ballistic Coefficient*, *BC5D Table*.

**Elevation Adjustment**
: The vertical angular adjustment applied to a riflescope turret to compensate for bullet drop at a given range. Expressed in MIL or MOA clicks. See *Come-Up*, *Click Value*.

**Eötvös Effect**
: The vertical component of the Coriolis effect. A bullet fired eastward gains apparent lift because it moves faster relative to Earth's surface and experiences increased centrifugal force; a bullet fired westward loses altitude for the opposite reason. Named after the Hungarian physicist Loránd Eötvös. See *Coriolis Effect*.

**Euler Method**
: A first-order numerical integration method that advances the solution by a single derivative evaluation per step: $y_{n+1} = y_n + h\,f(t_n, y_n)$. The simplest ODE solver available in BALLISTICS-ENGINE; less accurate than RK4 or RK45 for the same step size but faster per step. Useful primarily as a pedagogical reference or for coarse estimates. See *RK4*, *RK45*.

**Flat-Base**
: A bullet with a square, non-tapered base perpendicular to the bullet axis. Flat-base designs have higher base drag than boat-tail bullets because airflow separates abruptly at the base, creating a large low-pressure wake. The G1 drag model was designed around a flat-base projectile shape. See *Boat Tail*, *G1 Drag Model*.

**Form Factor ($i$)**
: The ratio of a bullet's drag coefficient to the drag coefficient of the standard reference projectile at the same Mach number:

$$i = \frac{C_D}{C_{D,\text{ref}}}$$

A form factor of 1.0 means the bullet matches the reference projectile exactly. Values below 1.0 indicate lower drag (sleeker shape); values above 1.0 indicate higher drag. Related to the ballistic coefficient by BC $= SD/i$. See *Ballistic Coefficient*, *Sectional Density*.

**G1 Drag Model**
: The most widely used standard drag model, based on a flat-base projectile with a 2-caliber secant ogive nose (the "Ingalls" standard projectile). Best suited for flat-base hunting and varmint bullets. G1 BCs are the most commonly published by manufacturers but can produce significant errors for long-range boat-tail bullets, especially in the transonic region. See *Drag Model*, *G7 Drag Model*.

**G7 Drag Model**
: A drag model based on a VLD (Very Low Drag) boat-tail projectile with a long secant ogive. Preferred for modern match-grade and long-range bullets because its reference shape

closely matches typical boat-tail designs, yielding a more constant BC across the velocity envelope. See *Drag Model*, *G1 Drag Model*, *VLD*.

**G2, G5, G6, G8, GI, GS Drag Models**
Additional standard drag models for various projectile shapes. G2 is designed for a cone-nosed projectile, G5 for a low-drag boat-tail with short nose, G6 for a flat-base with secant ogive, G8 for a flat-base with conic-curve nose, GI for a blunt-nosed projectile, and GS for a sphere. All are supported by BALLISTICS-ENGINE. See Appendix C for detailed drag tables.

**Greenhill Formula**
An empirical formula for estimating the barrel twist rate needed to gyroscopically stabilise a given bullet, based on bullet length and diameter. The classical form is:

$$T = \frac{C\,d^2}{L}$$

where $T$ is the twist rate (inches per turn), $d$ is the bullet diameter, $L$ is the bullet length, and $C$ is a constant (150 for velocities up to $\sim$2800 fps, 180 for higher velocities). Largely superseded by the more accurate Miller stability formula. See *Miller Stability Factor*, *Twist Rate*.

**Gyroscopic Stability**
The tendency of a spinning projectile to maintain its nose-forward orientation due to angular momentum imparted by the rifling. Quantified by the gyroscopic stability factor $S_g$: values above 1.0 indicate stability, above 1.4 indicate robust stability. A bullet that is not gyroscopically stable will tumble in flight. See *Stability Factor*, *Twist Rate*, *Miller Stability Factor*.

**Headwind**
Wind blowing toward the shooter from the direction of the target (a $0°$ or $360°$ wind angle). Headwind increases the relative airspeed of the projectile, thereby increasing drag and reducing downrange velocity and maximum range. The effect on drop is typically small compared to the effect of crosswind on windage. See *Tailwind*, *Crosswind*.

**ICAO Standard Atmosphere**
The International Civil Aviation Organization's model of atmospheric conditions as a function of altitude (ISO 2533). Defines standard temperature, pressure, and density profiles in seven layers from sea level to 84 km. The sea-level reference conditions are 15 °C, 1013.25 hPa, and $1.225\ \mathrm{kg/m^3}$. BALLISTICS-ENGINE implements the full seven-layer model. See *Adiabatic Lapse Rate*, *Density Altitude*.

**Jacket** The outer metallic layer of a bullet, typically made of copper or gilding metal (a copper-zinc alloy), enclosing a lead or lead-alloy core. Jacket material and thickness affect bore

friction and therefore muzzle velocity consistency, but do not directly enter exterior ballistic calculations.

**Kinetic Energy**
The energy of a projectile in motion, given by:

$$E_k = \frac{1}{2}\, m\, v^2$$

Typically expressed in foot-pounds (ft-lbs) in imperial units or Joules (J) in metric. Kinetic energy is a key metric for terminal performance and is reported at each range increment in BALLISTICS-ENGINE trajectory tables.

**Lag Time**
See *Time of Flight*.

**Lapse Rate**
The rate at which atmospheric temperature changes with altitude. In the standard atmosphere, the tropospheric lapse rate is $-6.5$ K/km (temperature decreases with altitude). Other atmospheric layers have different lapse rates, including isothermal (zero) and positive (temperature increasing with altitude) regions. See *Adiabatic Lapse Rate*, *ICAO Standard Atmosphere*.

**Lead** The horizontal distance ahead of a moving target at which the shooter must aim to compensate for the target's lateral motion during the projectile's time of flight. Expressed as an angular offset (MIL or MOA) or a linear distance at the target. See *Time of Flight*.

**Line of Sight (LOS)**
The straight line from the shooter's eye through the centre of the riflescope to the target. The trajectory of a bullet is always below the line of sight (except between the muzzle and the near zero crossing, and between the two zero crossings if a mid-range zero is used). See *Sight Height*, *Zero*.

**Mach Number**
The ratio of projectile velocity to the local speed of sound:

$$M = \frac{v}{a}$$

The Mach number is the fundamental parameter for determining the drag regime: subsonic ($M < 0.8$), transonic ($0.8 \leq M \leq 1.2$), or supersonic ($M > 1.2$). The local speed of sound $a$ depends on air temperature. See *Speed of Sound*, *Transonic Region*.

**Magnus Effect**
A lateral force on a spinning projectile caused by differential air pressure arising from the interaction between the bullet's spin and crosswind or angle of attack. The Magnus force

is generally much smaller than aerodynamic drag and is most significant for bullets at high angles of attack or in strong crosswinds. See *Spin Drift*, *Gyroscopic Stability*.

**Maximum Point-Blank Range (MPBR)**
: The farthest distance at which a bullet remains within a defined vertical zone (e.g., $\pm 4$ inches for big-game hunting) above and below the line of sight, requiring no holdover or scope adjustment. MPBR is computed by finding the zero range that maximises the distance before the trajectory exits the vital zone. See *Zero*, *Drop*.

**MIL (Milliradian)**
: An angular measurement equal to $1/1000$ of a radian ($\approx 0.0573°$). At $100\,\mathrm{m}$, 1 MIL subtends exactly $10\,\mathrm{cm}$; at 100 yards, 1 MIL subtends approximately 3.6 inches. One of the two standard angular systems used for riflescope adjustments and ballistic tables, the other being MOA. See *MOA*, *Click Value*.

**Miller Stability Factor**
: A formula developed by Don Miller for calculating the gyroscopic stability factor $S_g$ from bullet dimensions (length, diameter), weight, twist rate, and atmospheric conditions. An $S_g > 1.0$ indicates gyroscopic stability; $S_g > 1.4$ is considered well-stabilised. Bullets with $S_g$ between 1.0 and 1.3 are marginally stable and may lose stability in the transonic region. See *Stability Factor*, *Gyroscopic Stability*, *Greenhill Formula*.

**MOA (Minute of Angle)**
: An angular measurement equal to $1/60$ of a degree ($\approx 0.2909$ mrad). At 100 yards, 1 MOA subtends approximately 1.047 inches. Widely used in the US shooting community for scope adjustments, group-size measurement, and ballistic tables. See *MIL*, *Click Value*.

**Monte Carlo Simulation**
: A statistical method using repeated random sampling to model the effect of uncertainty on a system's output. In BALLISTICS-ENGINE, Monte Carlo mode propagates shot-to-shot variations in muzzle velocity, ballistic coefficient, shooting angle, wind speed, and wind direction through the trajectory solver to produce confidence ellipses and hit-probability estimates. See *Trajectory*.

**Muzzle Velocity (MV)**
: The speed of the projectile as it exits the barrel, typically measured in feet per second (fps) or metres per second (m/s) using a chronograph. Muzzle velocity is one of the most important inputs to any trajectory calculation; small changes in MV produce significant changes in drop at long range. See *True Velocity*, *Kinetic Energy*.

## E.2 Terms N–Z

**Nutation**
: A small, rapid oscillation of the projectile's spin axis about its mean (precessing) direction. In a gyroscopically stable bullet, nutation decays quickly due to aerodynamic damping, typically within the first few hundred metres of flight. See *Precession*, *Gyroscopic Stability*.

**Ogive**
: The curved nose section of a bullet, forward of the cylindrical bearing surface. The ogive shape strongly influences the drag coefficient. Common types include the *tangent ogive* (the curve is tangent to the cylindrical body at their junction) and the *secant ogive* (the curve intersects the body at an angle, producing a sleeker profile). VLD bullets use long secant ogives for minimum drag. See *VLD*, *Form Factor*.

**Pitch Damping**
: The aerodynamic moment that resists changes in a projectile's angle of attack, tending to restore the bullet to its equilibrium orientation. Adequate pitch damping is essential for dynamic stability, particularly in the transonic region where aerodynamic forces change rapidly. See *Nutation*, *Transonic Region*, *Stability Factor*.

**Point-Blank Range**
: See *Maximum Point-Blank Range (MPBR)*.

**Powder Temperature Sensitivity**
: The change in muzzle velocity per degree of temperature change in the propellant charge. Typically expressed in fps/°F or m/s/°C. Modern temperature-insensitive powders exhibit sensitivities of 0.5–1.0 fps/°F, while older or less-refined propellants may reach 2–3 fps/°F. Affects long-range accuracy by shifting the muzzle velocity used in trajectory calculations. See *Muzzle Velocity*.

**Precession**
: The slow gyroscopic rotation of a projectile's spin axis around the velocity vector, caused by the aerodynamic overturning moment acting on a spinning body. In right-hand twist barrels, the nose traces a clockwise circle (when viewed from behind) around the flight path. Precession is the physical mechanism behind spin drift. See *Spin Drift*, *Nutation*, *Gyroscopic Stability*.

**Profile**
: In BALLISTICS-ENGINE, a saved set of ballistic parameters—muzzle velocity, ballistic coefficient, projectile mass, diameter, drag model, sight height, twist rate, and zero range—that can be recalled by name for repeated use. Profiles eliminate the need to re-enter lengthy command-line arguments for frequently used rifle and load combinations.

**Range Table**
: A comprehensive tabulation of trajectory data at regular distance intervals, including

drop, windage, velocity, energy, time of flight, Mach number, and elevation/windage adjustments in MIL and MOA. Functionally synonymous with *Ballistic Table* in the context of BALLISTICS-ENGINE.

**Retardation**

The deceleration of a projectile due to aerodynamic drag, expressed in $\mathrm{m/s^2}$ or $\mathrm{ft/s^2}$. Retardation depends on the drag coefficient, air density, projectile velocity, and the projectile's ballistic coefficient. See *Coefficient of Drag*, *Ballistic Coefficient*.

**RK4 (Runge-Kutta 4th Order)**

A fixed-step numerical integration method that evaluates the derivative four times per step to achieve 4th-order accuracy. Significantly more accurate than the Euler method for the same step size, with a local truncation error proportional to $h^5$. Available in BALLISTICS-ENGINE as an alternative to the default RK45 integrator. See *Euler Method*, *RK45*.

**RK45 (Runge-Kutta-Fehlberg)**

An adaptive-step numerical integration method that embeds a 4th-order and a 5th-order Runge-Kutta solution within the same set of derivative evaluations. The difference between the two solutions provides an error estimate, allowing the solver to automatically adjust the step size to maintain a specified accuracy tolerance. The default integrator in BALLISTICS-ENGINE, offering the best balance of accuracy and computational efficiency. See *RK4*, *Euler Method*.

**Sectional Density (SD)**

The ratio of a bullet's mass to the square of its diameter:

$$\mathrm{SD} = \frac{m}{d^2}$$

where $m$ is mass (in pounds, for the imperial convention) and $d$ is diameter (in inches). Higher sectional density generally correlates with better penetration and a higher ballistic coefficient for a given form factor. See *Ballistic Coefficient*, *Form Factor*.

**Shooting Angle**

The angle above or below horizontal at which the rifle is aimed. Positive values indicate uphill shots; negative values indicate downhill. At steep angles, the effective gravitational component along the bullet's path is reduced, leading to less drop than a horizontal shot at the same slant range—the basis of the "rifleman's rule" (multiply the slant range by the cosine of the angle to obtain the equivalent horizontal range). See *Drop*, *Trajectory*.

**Sight Height**

The vertical distance from the centre of the bore to the centre of the optical axis of the riflescope. Typically 1.5–2.0 inches for bolt-action rifles with conventional scope mounts. Sight height affects the geometry of the zero crossing and near-range trajectory. See *Line of Sight*, *Zero*.

**Speed of Sound**

The velocity at which sound waves propagate through air. At standard conditions (15 °C, sea level, dry air): $a = 340.29\,\mathrm{m/s}$ (1116.8 fps). Temperature-dependent according to:

$$a = 331.3\sqrt{\frac{T}{273.15}}$$

where $T$ is the absolute temperature in Kelvin. The speed of sound defines the Mach number and thus the drag regime. See *Mach Number*.

**Spin Drift**

The gradual lateral deflection of a spinning projectile from its initial plane of fire, caused by gyroscopic precession. The direction of drift depends on the twist direction: a right-hand twist barrel produces rightward drift; a left-hand twist produces leftward drift. Spin drift increases with time of flight and is most significant at extreme long range. Enabled in BALLISTICS-ENGINE with the `--spin-drift` flag. See *Precession*, *Twist Rate*.

**Stability Factor ($S_g$)**

A dimensionless number indicating the degree of gyroscopic stability of a spinning projectile. $S_g > 1.0$ is required for stability; $S_g$ between 1.0 and 1.3 indicates marginal stability with risk of destabilisation in the transonic region; $S_g > 1.4$ indicates robust stability. See *Miller Stability Factor*, *Gyroscopic Stability*.

**Standard Atmosphere**

See *ICAO Standard Atmosphere*.

**Subsonic**

Velocities below approximately Mach 0.8 ($\lesssim 272\,\mathrm{m/s}$ at sea level). In the subsonic regime, drag is dominated by form drag and skin friction; wave drag is absent. Subsonic flight is relevant for suppressed firearms, certain pistol cartridges, and the terminal portion of long-range rifle trajectories. See *Mach Number*, *Transonic Region*, *Supersonic*.

**Supersonic**

Velocities above Mach 1.0 ($\gtrsim 340\,\mathrm{m/s}$ at sea level). In the supersonic regime, drag is dominated by wave drag caused by shock waves attached to the projectile's nose and base. Most rifle bullets are launched well into the supersonic regime. See *Mach Number*, *Transonic Region*, *Subsonic*.

**Tailwind**

Wind blowing from behind the shooter toward the target (a $180°$ wind angle). A tailwind slightly reduces the projectile's airspeed relative to the surrounding air mass, thereby reducing drag and marginally extending range. The effect is typically small compared to crosswind-induced windage. See *Headwind*, *Crosswind*.

**Time of Flight (ToF)**

The elapsed time from muzzle exit to arrival at a given downrange position, expressed in seconds. Time of flight increases nonlinearly with range as the projectile decelerates due to drag. ToF is a key input for Coriolis, spin drift, and moving-target lead calculations. See *Coriolis Effect*, *Spin Drift*, *Lead*.

**Trajectory**

The path of a projectile through the atmosphere from muzzle to impact, influenced by gravity, aerodynamic drag, wind, Coriolis effect, spin drift, and other forces. In BALLISTICS-ENGINE, the trajectory is computed by numerically integrating the equations of motion using the selected ODE solver. See *Drop*, *RK45*.

**Transonic Region**

The velocity range between approximately Mach 0.8 and Mach 1.2, where mixed subsonic and supersonic flow exists simultaneously around the projectile. Drag behaviour in the transonic region is complex and nonlinear; the drag coefficient changes rapidly with small velocity changes. Marginally stable bullets ($S_g < 1.3$) may lose stability in this region. Accurate modelling of transonic drag requires high-fidelity drag data, such as doppler-derived measurements. See *Subsonic*, *Supersonic*, *Stability Factor*.

**True Velocity**

The effective muzzle velocity calculated by matching observed field data (typically drop at a known range) to the ballistic model, as opposed to the chronograph-measured muzzle velocity. True velocity accounts for systematic errors in chronograph placement, barrel harmonics, and environmental conditions at the time of measurement. See *Muzzle Velocity*.

**Twist Rate**

The distance the rifling grooves travel to complete one full revolution of the bullet, expressed as "1:$N$" where $N$ is the distance in inches or millimetres (e.g., 1:10″ means one full turn in 10 inches). Faster twist rates (smaller $N$) are required to stabilise longer, heavier bullets. See *Gyroscopic Stability*, *Greenhill Formula*, *Miller Stability Factor*.

**Vapor Pressure**

The partial pressure exerted by water vapor in the atmosphere at a given temperature. Affects air density because water vapor ($M_v \approx 18.015\,\mathrm{g/mol}$) is lighter than dry air ($M_a \approx 28.965\,\mathrm{g/mol}$): at the same temperature and total pressure, moist air is less dense than dry air, resulting in slightly less drag. See *Air Density*, *CIPM Formula*.

**VLD (Very Low Drag)**

A bullet design optimised for minimum aerodynamic drag, typically featuring a long secant ogive and a boat-tail base. VLD bullets achieve high ballistic coefficients and are favoured for long-range competition. The G7 reference projectile is a VLD shape. See *Boat Tail*, *Ogive*, *G7 Drag Model*.

**Wind Card**
: A reference table showing wind deflection in MIL or MOA at various ranges for multiple wind speeds. Generated by the BALLISTICS-ENGINE `wind-card` command for a specific load and atmospheric condition. Carried in the field as a quick-reference companion to the dope card. See *Dope Card*, *Wind Deflection*.

**Wind Deflection**
: The horizontal displacement of a projectile from its no-wind trajectory, caused by a cross-wind component. Wind deflection increases nonlinearly with range because the wind acts on the bullet for a longer time and the bullet's decreasing velocity makes it more susceptible to lateral displacement. See *Crosswind*, *Windage*.

**Wind Shear**
: A change in wind speed or direction with altitude above ground level. BALLISTICS-ENGINE supports three wind-shear models: logarithmic (log-law), power-law, and the Ekman spiral (which also models directional changes with altitude). Wind shear is most significant for long-range, high-angle shots where the projectile traverses a large altitude range during flight. See *Crosswind*, *Wind Deflection*.

**Windage**
: The horizontal angular adjustment on a riflescope turret to compensate for wind deflection, expressed in MIL or MOA clicks. Also used colloquially to refer to any lateral displacement of the point of impact from the point of aim. See *Wind Deflection*, *Click Value*.

**Zero**
: The range at which the bullet's trajectory crosses the line of sight on the descending leg of its arc. A rifle "zeroed at 100 yards" has its scope adjusted so that the bullet impact coincides with the point of aim at that distance. Most trajectory calculations use the zero range as the baseline from which all come-up and windage adjustments are referenced. See *Line of Sight*, *Zero Angle*, *Come-Up*.

**Zero Angle**
: The bore elevation angle required to achieve a specific zero distance, accounting for sight height, gravity, and aerodynamic drag. Determined internally by BALLISTICS-ENGINE using a root-finding algorithm that iterates until the trajectory crosses the line of sight at the specified zero range. See *Zero*, *Sight Height*.

# Index


.270 Winchester
  hunting, 458
.30-06 Springfield
  hunting, 458
.300 Winchester Magnum
  hunting, 459
.308 Winchester, 24
.375 H&H Magnum
  hunting, 460
3-DOF model, *see* point-mass model
3D weather, 421
4D interpolation, 271
4DOF effects, 324
5D interpolation, 269
6.5 Creedmoor
  hunting, 457

a7p format, 162
Aberdeen Proving Ground, 11
accuracy, 363
accuracy threshold, 447
adaptive step size, 359
adiabatic lapse rate, 673
adjustment-unit flag, 80
aerodynamic drag, 44
aerodynamic jump, 317–327
air density, 182–183, 673
  CIPM calculation, 427
  humidity correction, 426
  temperature dependence, 177
altimeter setting, 179
altitude, 30
  effect on trajectory, 91
  hunting corrections, 452
altitude effects, 190
  practical comparison, 191
angle of departure, 673
angular precision, 465
apex, 375
API
  endpoint, 404
  error handling, 406
  Flask API, 401–417
  request structure, 408
  response structure, 409
Applied Ballistics, 11
ArcherBC2, 162
architecture, 511
  core solver, 512
  drag modules, 512
  flat layout, 529
  module organization, 511
  physics modules, 513
  pipeline, 516
Arden Buck equation, 182, 426
Army Standard Metro, 264
atmosphere, 173–196
  density, 53
  engine implementation, 186–189
  humidity, 54
  ICAO model, 52
  layers, 52

overriding, 54
pressure, 53
atmospheric conditions, 91
atmospheric layers, 174
atmospheric modeling, 419–440
atmospheric profiles, 421
atmospheric sensitivity, 194
audience, 9
auto-zero, 36
azimuth, 674

ballistic coefficient, 48, 674
advanced modeling, 245–262
as input, 71
bounds, 254
bullet selection, 489
common mistakes, 273
definition, 48
fallback, 534
from dimensions, 604
G1 vs. G7, 49, 228
interpolation, 356
practical use, 263–277
published values, 263–264
trajectory effects, 50
truing, 267
uncertainty, 119
velocity dependence, 51, 245
vs. weight, 489
ballistic table, 674
`BallisticInputs`, 517, 524
ballistics-engine
CLI, 4
design goals, 8
library, 4
overview, 3
subcommands, 5
BALLISTICS_API_TOKEN, 601
barometric formula, 176, 430
barometric pressure, 179
corrections, 429
barrel length
hunting trade-offs, 489
velocity prediction, 488
Bashforth, Francis, 10
BC, 48
confidence intervals, 146
BC degradation, 245–262
BC estimation, 133
from ladder test, 496
from velocity measurements, 138
BC interpolation, 242
BC measurement methods, 265
BC model comparison, 259
BC segments, 242, 250–253
BC truing, 133, 267–269
motivation, 133
bc-reference flag, 264
bc-segment flag, 252
BC5D correction table, 269–272
BC5D format, 270
BC5D table, 674
BC5D tables
download, 414
BCSegmentData struct, 250
BCSegmentEstimator struct, 250
bdc-match command, 595
benchmarks, 567
binary search, 370
bisection method, 384
boat tail, 674
book organization, 12
bore height, 31, 674
bore sight, 674
bracket technique, 217
Brent's method, 103, 386
BRL (Ballistic Research Laboratory), 11
bullet diameter, 72
bullet expansion
minimum velocity, 448

bullet mass, 72
bullet selection, 489
    trajectory comparison, 490

calculate_atmosphere(), 186
`Calculator` (WASM), 548
caliber, *see* bullet diameter
calibrate-bc command, 604
Cargo, 20
Cargo features, 530
    online, 403
cartridge overall length, 604
cartridge selection
    competition, 467
Catmull–Rom spline, 236, 655
cd-scale flag, 234
CD_TO_RETARD, 350
CDM, *see* custom drag table
CEP, 122
    definition, 122
CEP (Circular Error Probable), 471
charge weight
    velocity prediction, 502
chrono-distance flag, 139
chronograph
    BC measurement, 266
    downrange, 138
    downrange offset, 139
    sample size, 500
CIPM formula, 186, 427, 675
    implementation, 186
Circular Error Probable, 122
click value, 675
clicks, 80
clock position, 213
cluster BC degradation, 246–250
ClusterBCDegradation struct, 246
coefficient of drag, 675
cold conditions, 190
come-up, 675
come-up table, 37
come-ups, 455
compare command, 491
competition
    dope book, 468
    workflow, 481
compiler optimization, 530
compressibility factor, 187
computational ballistics
    motivation, 6
    theory vs. practice, 6
    workflows, 7
`compute_derivatives`, 531
confidence ellipse, 122, 471
    computation, 123
constants, 533
`constants.rs`, 533
convergence
    criteria, 390
`convert_inputs`, 559
coordinate system, 55
    ballistics, 202
    shooter terminology, 57
Coriolis
    acceleration, 351
Coriolis effect, 95, 297–310, 675
    CLI usage, 303
    competition, 476
    heading dependence, 302
    horizontal, 298
    latitude dependence, 302
    magnitude, 477
    practical significance, 305
crate types, 529
CRC32, 675
    IEEE polynomial, 664
Criterion, 567
critical Mach number, 255
cross-platform, 560
cross-surface consistency, 560

crossover range, 494
crosswind, 676
crosswind deflection, 201–203
  zero in vacuum, 201
CSV profiles, 167
cubic interpolation, 236, 655
custom drag table, 656

Dalton's law, 181, 426
data flow, 522
  ownership, 523
data structures, 524
density altitude, 183–185, 454, 676
  definition, 184
  zeroing, 107
density ratio, 182
density-altitude flag, 184
dependencies, 535
derivatives, 531
  computation, 354
design decisions, 527
dew point, 421
dope book, 468
dope card, 34, 93, 454, 676
Doppler radar, 676
doppler radar
  BC measurement, 265
doppler-derived data, 232, 265, 656
Dormand–Prince method, 359, 519
Dormand-Prince method, 83
downhill shooting, 88
drag, 44
  acceleration, 350
  retardation formula, 60
drag coefficient, 45, 223–244, 675, *see* Coefficient of Drag
  lookup, 355
  tables, 649–657
drag equation, 44
drag evaluation pipeline, 240–242
drag function
  G1, 46
  G6, 46
  G7, 46
  G8, 46
drag model, 676
  custom, 73
  definition, 223
  for truing, 148
  G1, 73, 650
  G1 vs. G7 selection, 493
  G2, 653
  G5, 653
  G6, 73, 653
  G7, 35, 73, 651
  G8, 73, 653
  GI, 653
  GS, 653
drag models, 223–244
drag table
  custom, 232–235, 656
  loading strategy, 241
drag tables, 649–657
DragTable struct, 235
`.drg` file, 234
drift, 29
drop, 29, 676
  gravitational, 43
  line of sight, 374
drop sensitivity, 499
drop-scale factor, 142
drops-reference flag, 89
DSF, 142
dsf command, 142

Earth's rotation, 297
effective BC, 676
Ekman spiral, 206, 434
elevation adjustment, 676
embedding, 558

energy
  minimum for game, 446
  thresholds for game, 446
enhancement factor, 187
ENIAC, 11
environmental conditions, 30
Eötvös effect, 297–310
  vertical deflection, 300
Eötvös effect, 677
epicyclic motion, 311–327
  definition, 315
equations of motion, 349
equivalent horizontal range, 88
error estimation, 359
estimate-bc command, 134
  data, 135
  drag-model, 135
  velocity-data, 136
  zero-range, 137
EstimateBC command, 269
Euler method, 677
event detection, 366
expected value
  stage decisions, 474
exterior ballistics
  history, 10
  overview, 41
  primer, 41
extreme spread, 118, 498
  definition, 124
  Monte Carlo, 122
extreme spread vs. SD, 500

F-Class, 465, 466
far-zero, 104
fast trajectory solver, 365, 573
`fast_trajectory.rs`, 573
FastSolution, 365
feature flags, 21
feature gates, 530
FFI, 539
  building, 543
  C example, 544
  C interface, 539
  data structures, 540
  drag model codes, 541
  exported functions, 542
  Swift example, 546
`ffi.rs`, 539
`FFIBallisticInputs`, 540
`FFITrajectoryResult`, 541
firing tables, 10
first focal plane, 592
first-round hit probability, 127, 473
  computation, 473
flags
  –api-timeout, 403
  –api-url, 403
  –bc-table-auto, 414
  –bc-table-refresh, 414
  –bc-table-url, 414
  –compare, 403, 411
  –latitude, 406
  –longitude, 406
  –offline-fallback, 403
  –online, 402
  –shot-direction, 406
flat-base, 677
forces on a bullet, 41
form factor, 238–239, 677
free recoil, 497

G1
  truing considerations, 148
G1 drag model, 225–227, 677
  drag table, 650
G2 drag model, 229, 653, 678
G5 drag model, 229, 653, 678
G6 drag model, 229, 653, 678
  military FMJ, 230

G7
    truing considerations, 148
G7 drag model, 35, 227–229, 677
    drag table, 651
G8 drag model, 229, 653, 678
    long ogive, 231
get_local_atmosphere, 431
get_local_atmosphere(), 188
get_wind_at_position(), 207
GI drag model, 229, 653, 678
gravitational acceleration, 42
gravity, 42, 351
Greenhill formula, 678
GS drag model, 229, 653, 678
    round ball, 231
gyroscopic stability, 281–295, 678

handloading, 487
headwind, 678
    effect on drop, 203
history of ballistics computation, 10
hit probability, 124
hold-corridor command, 597
holdover, 589
hot conditions, 189
humidity, 54, 91
    correction, 186
    counterintuitive effect, 180
    effect on air density, 177
    effects on air density, 426
    from dew point, 421
hunting applications, 443
hunting cartridges, 457

IAPWS-IF97, 186
ICAO Standard Atmosphere, 52, 173, 678
    BC reference, 264
    definition, 173
    lapse rates, 423
    lookup, 186
IDPA, 480
inclination, 88, 682
Ingalls tables, 231
initial conditions, 353
`InitialConditions`, 525
installation, 17
    building from source, 20
    FreeBSD, 20
    Linux, 18
    macOS, 18
    NetBSD, 20
    OpenBSD, 20
    pre-built binaries, 17
    verification, 23
    Windows, 19
integration
    Euler, 83
    RK4, 83
    RK45, 83
International Standard Atmosphere, *see* ICAO Standard Atmosphere
interpolation
    drag table, 236–237, 655
    linear, 370
introduction, 3
inverse solvers, 594
IPHY, 80
ISA, *see* ICAO Standard Atmosphere
iteration limit, 391

jacket, 678
jemalloc, 578
JSON
    profile format, 156

Kestrel, 216
kinetic energy, 679

ladder test, 494
lag angle, 201
lag time, *see* Time of Flight
lapse rate, 423–426, 679

launch angle, 44
LazyLock, 241
lead (aiming), 679
line of sight, 374, 679
linear interpolation, 655
link-time optimization, 577
Litz
    spin drift formula, 284
Litz, Bryan, 11
load development, 163, 487
    safe workflow, 501
local atmosphere, 188
location
    GPS coordinates, 406
login command, 601
logout command, 601

Mach number, 45, 679
    transonic, 255
Mach transition, 375
Magnus effect, 95, 289–295, 679
    acceleration, 351
    coefficient, 290, 533
mark-to-range command, 594
maximum effective range, 446
maximum ordinate, 105
maximum point-blank range, 105, 680
MCP server, 555
memory layout, 576
METAR, 419
metric units, 34
MIL, 680
    competition use, 465
MIL (milliradian), 29, 57
    definition, 57
Miller stability factor, 680
Miller stability formula, 332–336
    advanced corrections, 334
    corrections, 333
milliradian, 680
mimalloc, 578
min-max, 596
minute of angle, 680
mirage
    wind reading, 216
ML corrections, 409
MOA, 680
    accuracy limit, 447
    competition use, 465
MOA (Minute of Angle), 29, 57
    definition, 57
Monte Carlo
    parallelism, 570
    wind analysis, 470
    wind uncertainty, 217
Monte Carlo simulation, 115, 680
    BC confidence, 146
    BC variation, 119
    convergence, 121
    input distributions, 118
    motivation, 115
    number of iterations, 121
    practical example, 127
    wind variation, 120
monte-carlo command, 116
mountain hunting, 452
mover ring, 82
MPBR, 105, 443, 680
    flat-shooting myth, 445
mpbr command, 106
multi-segment wind, 209
muzzle, 41
muzzle velocity, 70, 680
    barrel length effect, 488
    extreme spread, 118
    standard deviation, 118

nalgebra, 528
near-zero, 104
Newton's method, 385

Newton, Isaac, 10
non-standard conditions, 189
NRL, 465
numerical integration, 349
numerical stability, 388
nutation, 311–327, 681
  CLI usage, 322
  damping, 314
  definition, 313
  frequency, 313

observation-sigma flag, 145
OCW (Optimal Charge Weight), 494
ogive, 681
one-seventh power law, 206
online mode, 401–417
  –online flag, 402
  compare mode, 411
  data flow, 408
  overview, 401
  request lifecycle, 415
  reverse solvers, 601
  when to use, 415
optimal-zero command, 596
output
  CSV, 33
  formats, 32
  JSON, 32
  PDF, 34
  specific ranges, 377
  table, 32
  understanding, 29
output format
  CSV, 378
  JSON, 378
  table, 378
output formats
  CSV, 470
  JSON, 470
over-stabilisation, 292

PDF dope card, 454
  competition, 484
  generation, 455
performance, 567
  allocation, 577
  allocators, 578
  benchmarks, 567
  cache efficiency, 576
  drag lookup, 242
  drag table lookup, 576
  fast solver, 573
  integration method comparison, 569
  LTO, 577
  Monte Carlo, 570
  raw arrays, 575
  single trajectory, 568
  step size, 363
  WebAssembly, 581
physics-bounded BC, 254
pitch damping, 319–327, 681
  coefficient, 320
  transonic sign reversal, 338
plan-truing command, 144
plot flag, 80
point-blank range, 105, 443, *see* Maximum Point-Blank Range
point-mass model, 60
powder
  recommendation, 603
powder command, 504
powder temperature
  measured curve, 111
powder temperature sensitivity, 395, 503, 681
powder-temp-curve flag, 111
power factor, 480
power-factor command, 480
pre-match preparation, 481
precession, 311–327, 681
  CLI usage, 322
  frequency, 312

gyroscopic, 311
precision budget, 466
precision rifle competition, 465
pressure
altitude correction, 429
barometric, 91, 177
pressure-type flag, 179
probability of hit, 470
profile, 681
saving, 482
profile command
delete, 159
list, 159
save, 154
show, 159
profile import, 162
profile zero-set, 160
profiles, 153
creating, 154
CSV format, 167
JSON format, 156
loading, 157
managing multiple, 159
motivation, 153
online mode, 416
profiling, 579
cache misses, 581
flame graphs, 580
Instruments, 580
perf, 579
projectile shape
classification, 256
ProjectileShape enum, 256
PRS, 465
vs. F-Class, 466
PyO3, 552
Python bindings, 539, 552
installation, 553
Monte Carlo, 554
usage, 553

QNH, 179
quartering wind, 204

RA4 drag model, 229
range day
weather workflow, 437
range table, 37, 681
Rayon, 570
parallel iterators, 571
recoil, 497
recoil command, 497
recommend-col command, 604
recommend-powder command, 603
recommend-twist command, 603
relative velocity, 198
retardation, 60, 682
retardation constant, 350
reticle, 589–600
description schema, 589
reticle command, 589
reticle hold, 591
reticle import, 605
reverse solvers, 601–606
Reynolds number, 340
rifleman's rule, 450
limitations, 452
rifling, 281
RK4, 357, 682
RK45, 682
root finding, 103
round ball drag, 231
Runge–Kutta
RK4, 357
RK45, 359, 519
Runge-Kutta, 682
Runge-Kutta-Fehlberg, 682
Rust
advantages, 527

sampling
distances, 372

pipeline, 371
problem, 369
saturation vapor pressure, 182, 426
saved-profile flag, 157
scope offset, 86
second focal plane, 592
sectional density, 492, 682
BC adjustment, 253
shock waves, 255
shoot-to range, 88
shooting angle, 88, 450, 682
magnitude of effect, 451
physics, 450
steep, 392
Sierra Infinity, 11
Sierra MatchKing, 25
sight height, 31, 86, 682
sight-offset flag, 90
SMOA, 80
solve-json, 555
solver pipeline, 516
atmosphere, 518
input parsing, 517
integration, 519
output, 521
zeroing, 519
source code
directory layout, 511
speed of sound, 683
humidity effect, 429
moist air correction, 186
temperature dependence, 177
spin decay, 287–295
exponential model, 287
spin drift, 95, 283–295, 683
advanced model, 285
altitude effects, 294
bullet type coefficients, 286
CLI usage, 284
competition, 476
enhanced model, 290
practical significance, 292
spin effects, 281–295
spin rate
calculation, 282
stability, 329–346
CLI subcommand, 291
downrange changes, 330
dynamic, 331
factor $S_d$, 331
factor $S_g$, 329
gyroscopic, 329
subcommand usage, 335
stability factor, 683
stage planning, 473
sequence, 473
standard atmosphere, 52, *see* ICAO Standard Atmosphere
sea level, 52
standard deviation
velocity, 118, 498
standard projectile
G1, 225
G7, 227
state vector, 352, 522
station pressure, 179
step size, 83, 361
stratosphere, 175
subsonic, 683
subtension, 592
supersonic, 683
surface roughness length, 205

tailwind, 683
effect on drop, 203
tall-target command, 478
tall-target test, 478
target detection, 358, 520
target height, 392
target-radius flag, 117

target-speed flag, 82
Tartaglia, Niccolò, 10
temperature, 30
  effect on trajectory, 91, 177
  powder sensitivity, 503
terminal chart, 80
terminal performance, 448
testing, 22
thread safety, 561
time of flight, 684
tolerance, 390
tracking correction factor, 478
trajectory, 684
  complete example, 60
  first example, 24
trajectory command, 69
  advanced physics, 95
  auto-zero, 74
  output columns, 75
  output formats, 78
  overview, 69
  required inputs, 70
trajectory flags, 375
trajectory sampling, 369
trajectory solver, 349
TrajectoryParams, 354
`TrajectoryParams`, 526
  cache implications, 577
`TrajectoryPoint`, 526
`TrajectoryResult`, 526
transonic
  competition, 475
transonic drag correction, 256–259
transonic regime, 255–262
transonic region, 684
transonic zone, 329–346
  destabilisation, 338
  drag rise, 337
  engine modeling, 340
  physics, 336
  practical guidance, 342
  shape estimation, 341
tropopause, 52, 175
troposphere, 52, 175
true altitude, 183
true velocity, 684
true-velocity command, 138
  observed, 141
  online, 413
true-wind command, 214
truing
  experiment design, 144
  for competition, 469
  joint MV and BC, 141
  to observed drop, 138
  uncertainty-aware, 145
twist rate, 684
  and stability, 291
  definition, 281
  recommendation, 603
typographic conventions, 13

unit conversion
  API boundary, 410
  MIL to inches, 58
  MOA to inches, 58
unit system, 34
units, 57
uphill shooting, 88
uphill/downhill shooting, 450
USPSA, 480

vapor pressure, 684
velocity SD, 498
velocity truing
  online, 413
Ventum reticle, 605
vital zone, 105, 443
  sizing, 445
VLD, 684
VLD bullet, 651

WASM, 22
wasm.rs, 547
WasmBallistics, 547
wave drag, 257, 338
Weapon Employment Zone, 125
weather integration, 419–440
weather stations, 419
weather zones, 436–437
    interpolation, 436
WebAssembly, 22, 539, 547
    building, 550
    integration, 551
    performance, 581
    random numbers, 562
WEZ, 125
wez flag, 125
wind, 197–220
    call strategy, 471
    competition, 470
    direction, 85
    effect on trajectory, 197
    hunting considerations, 449
    shear, 86
    speed, 85
    uncertainty modeling, 470
    variability, 120
    variable, *see* wind segments
    vector decomposition, 202
wind bearing, 213
wind bracket, 217
wind call
    truing, 214
wind card, 685
wind clock, 198–200
wind deflection, 685
wind direction, 56
wind indicators, 217
wind meter, 216
wind reading, 216–218
wind scenarios, 597
wind segments, 209–212
    CLI usage, 211
wind shear, 205–209, 432–435, 685
    CLI flags, 206
    custom layers, 206, 435
    definition, 205
    Ekman spiral, 206, 434
    logarithmic profile, 205, 432
    models, 205, 432
    power law, 206
    power law profile, 433
    practical significance, 208
    step size, 362
    trajectory integration, 207
wind-induced yaw, 197
wind-ref flag, 213
windage, 85, 685
WindSegmentError, 210
WindShearWindSock, 437
WindSock, 209
workflow, 153
    complete, 163

yaw of repose, 283

Zeiss Rapid-Z, 595
zero, 685
zero angle, 44, 383, 685
zero command, 100
    from-angle, 102
zero crossing, 375
    detection, 375
zero sets, 160
zero-day conditions, 110
zero-poi-up flag, 90
zero-velocity flag, 110
zeroing, 36, 99
    algorithm, 103
    altitude change, 453
    at altitude, 107
    auto-zero, 74

Brent's method, 386
concept, 99
edge cases, 392
long range, 393
non-standard conditions, 107
pipeline, 388
problem, 383
subsonic, 394
zeroing algorithm, 383