

Numerical optimization and optimal control for dynamical systems [146442]

Numerical Methods

Enrico Bertolazzi



Copyright © 2025 Enrico Bertolazzi

University of Trento — Department of Industrial Engineering

Via Sommarive 9, 38123 Povo (Trento), Italy

Licensed under the Creative Commons Attribution-NonCommercial 4.0 License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at https://creativecommons.org/licenses/by-nc-sa/4.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "as is" basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Contents

I Constrained Minimization

1 Optimization Algorithms 7

1.1 Unconstrained Optimization 7

1.1.1 From optimality conditions to an iteration 8
1.1.2 Conditioning and scaling 8
1.1.3 Convergence rates 9
1.1.4 Gradient methods 9
1.1.5 Line-search globalization 10
1.1.6 Newton and modified-Newton methods 10
1.1.7 BFGS and limited-memory methods 11
1.1.8 Trust-region globalization 11
1.1.9 Stopping tests and diagnostic evidence 12
1.1.10 Choosing a method 13
1.1.11 Computational laboratory 13
1.1.12 Exercises 13

1.2 Constrained Optimization 14

1.2.1 The NLP viewpoint and algorithmic map 14
1.2.2 Unconstrained methods as the local computational model 15
1.2.3 Penalty methods and exact penalties 16
1.2.4 Equality-constrained minimization 17
1.2.5 Implementing SQP in practice 20
1.2.6 Inequality constraints 24
1.2.7 Augmented Lagrangian method 32

1.2.8 A worked example . . . 33
1.2.9 Exercises . . . 35
1.2.10 Local convergence and regularity . . . 37
1.2.11 Linear algebra for KKT systems . . . 37
1.2.12 Stopping criteria and diagnostics . . . 38
1.2.13 Choosing a method . . . 38
1.2.14 A practical hierarchy for smooth NLP . . . 39

II ODE

2 Numerical Methods for IVPs . . . 45
2.0.1 Explicit Methods . . . 46
2.1 A First Look at the Explicit Euler Method 46
2.1.1 Consistency, Local Error, and Convergence . . . 48
2.2 Euler Method in Vector Form 50
2.2.1 General Form of the Euler Method . . . 50
2.3 Stability: continuous problem versus numerical method 52
2.4 Heun and Collatz Methods 53
2.4.1 Heun's Method . . . 53
2.4.2 Collatz Method . . . 54
2.4.3 Comparison of Methods . . . 55
2.5 Numerical Verification of the Order 55

3 Taylor Methods for ODEs . . . 61
3.1 Taylor Methods of Low Order 61
3.1.1 From Taylor's Formula to Euler . . . 61
3.1.2 The Second-Order Taylor Method . . . 62
3.1.3 The Third-Order Taylor Method . . . 62
3.2 A Recursive Formulation 63
3.3 An Improved Recursion 65

4 Runge-Kutta Methods . . . 69
4.1 The General Runge–Kutta Method 70
4.2 Constructing Runge-Kutta Methods 71
4.2.1 The Structure of A . . . 71
4.2.2 Solving the Stage Equations . . . 73
4.2.3 Order Conditions: the Two-Stage Case . . . 74
4.2.4 Order Conditions: the General Case . . . 76
4.2.5 Embedded Runge–Kutta Pairs and Adaptive Step Size . . . 82

4.3 Examples of Runge–Kutta Methods **83**
4.3.1 Explicit methods . . . 83
4.3.2 Implicit methods . . . 86

5 Linear Multistep Methods . . . 91
5.1 Foundations **91**
5.1.1 Characteristic Polynomials . . . 91
5.1.2 Consistency and Order . . . 92
5.2 Adams methods **92**
5.2.1 Adams–Bashforth methods . . . 93
5.2.2 Adams–Moulton methods . . . 93
5.3 Backward Differentiation Formulas (BDF) **94**
5.4 Stability and Convergence **95**
5.4.1 Zero-Stability . . . 95
5.4.2 Convergence and the Dahlquist Equivalence Theorem . . . 96
5.4.3 Absolute Stability of Multistep Methods . . . 97
5.5 Practical Comparison **98**

6 Absolute Stability and Stability Regions . . . 99
6.1 Why study stability? Start with explicit Euler **99**
6.2 Why do complex numbers appear? Linear systems and diagonalization **100**
6.2.1 Explicit Euler applied to the system . . . 101
6.3 The complex scalar test equation and the stability region **101**
6.3.1 Explicit Euler: derivation of its disk . . . 102
6.4 Runge–Kutta methods: deriving the stability function step by step **103**
6.4.1 Explicit Euler (one stage) . . . 103
6.4.2 Explicit second-order Runge–Kutta: Heun's method . . . 104
6.4.3 Explicit midpoint RK2: the same stability polynomial . . . 104
6.4.4 Classical fourth-order Runge–Kutta (RK4) . . . 104
6.4.5 General RK stability function from the Butcher tableau . . . 105
6.4.6 Implicit Euler . . . 106
6.4.7 Trapezoidal Rule . . . 106
6.5 A-Stability and L-Stability **106**
6.6 Stability for Systems and Multistep Methods **107**
6.6.1 From the Scalar Test Equation to Systems . . . 107
6.6.2 Linear Multistep Methods . . . 107
6.6.3 Boundary-Locus Method . . . 108
6.6.4 BDF Methods . . . 108
6.7 Stability Diagrams and Examples **109**
6.7.1 Stability Plots . . . 109
6.7.2 A Stiff Two-Mode Example . . . 111

6.8 Practical Guidance **112**
6.8.1 Stability Versus Accuracy . . . 112
6.8.2 Oscillatory Problems and the Imaginary Axis . . . 112
6.8.3 Practical Interpretation of Stability Diagrams . . . 112
6.8.4 Comparison of Representative Methods . . . 113
6.9 Geometric Integration and Invariant Preservation **114**
6.9.1 Symplectic Euler and Störmer–Verlet . . . 114
6.9.2 Constraints and invariants . . . 114
6.9.3 A practical long-time test . . . 114

7 Checking the Order of a RK Method . . . 117
7.1 The Conditions in Vector Form **117**
7.2 Worked Verifications **118**
7.2.1 Explicit Euler . . . 118
7.2.2 Explicit Midpoint . . . 119
7.2.3 Ralston's Second-Order Method . . . 119
7.2.4 Third-Order Runge–Kutta (RK3) . . . 119
7.2.5 Fourth-Order Runge–Kutta: the 3/8 Rule . . . 120
7.2.6 Implicit Midpoint . . . 121
7.2.7 Crank–Nicolson (Trapezoidal Rule) . . . 121
7.2.8 Gauss–Legendre of Order Four . . . 121
7.2.9 Lobatto IIIB of Order Four . . . 122
7.3 Automated Verification **123**

III DAE

8 Numerical Methods for DAEs . . . 131
8.1 Consistent Initialization **131**
8.1.1 A practical procedure . . . 131
8.1.2 Projection to nearby consistent data . . . 132
8.1.3 Acceptance checklist . . . 132
8.2 Implicit Integration of Linear DAEs **132**
8.2.1 Consistent Initial Values . . . 133
8.2.2 Explicit Euler . . . 133
8.2.3 Implicit Euler . . . 133
8.2.4 Implicit Euler Results . . . 134
8.2.5 Baumgarte Stabilization . . . 135
8.2.6 Crank–Nicolson Results . . . 136
8.3 Constraint Drift **138**
8.3.1 Why drift accumulates . . . 138
8.3.2 What to monitor . . . 138
8.3.3 Remedies . . . 138

8.4 Baumgarte Stabilization 139
8.4.1 DAEs in Mechanics . . . 139
8.4.2 Numerical Scheme . . . 140
8.4.3 Redundant-Coordinate Pendulum . . . 141
8.4.4 Unstabilized Solution . . . 142
8.4.5 Baumgarte Method . . . 144
8.4.6 First-Order Pendulum DAE . . . 149
8.4.7 Unstabilized Integration . . . 152
8.4.8 Stabilized Integration . . . 153
8.5 Projection Methods 156
8.5.1 Projection Principles . . . 157
8.5.2 Formulation . . . 159
8.5.3 Projected Integration . . . 159
8.5.4 Constrained Minimization . . . 159
8.6 Projection Algorithms 162
8.6.1 Index-1 Projection . . . 162
8.6.2 Index-2 Projection . . . 164
8.6.3 Index-3 Projection . . . 167
8.7 Implicit Projection 170
8.7.1 Index-1 DAE . . . 170
8.7.2 Index-3 DAEs . . . 171
8.8 Coordinate Partitioning 173
8.8.1 Independent Coordinates . . . 173
8.8.2 Constraint Propagation . . . 173
8.8.3 Collatz . . . 174
8.8.4 Fourth-Order Runge–Kutta . . . 174
8.9 Order Reduction in DAE Integration 176
8.9.1 Main mechanisms . . . 176
8.9.2 How to detect it . . . 177
8.9.3 Mitigation . . . 177
8.10 Choosing a Numerical DAE Method 177
8.10.1 Decision table . . . 178
8.11 Examples 179

IV Optimal Control

9 Numerical Methods for Optimal Control . . . 187
9.1 Reference optimal control problem 188
9.2 Optimize then discretize or discretize then optimize 188
9.3 Indirect methods 189
9.3.1 Single shooting . . . 189

9.3.2 Multiple shooting . . . 190
9.4 Direct control parameterization 190
9.4.1 Forward sensitivities . . . 191
9.4.2 Adjoint gradients . . . 191
9.5 Direct transcription 191
9.5.1 Euler transcription . . . 192
9.5.2 Trapezoidal collocation . . . 192
9.5.3 Hermite–Simpson collocation . . . 192
9.5.4 General collocation and Runge–Kutta schemes . . . 192
9.6 A complete trapezoidal transcription 192
9.7 Time scaling and free terminal time 193
9.8 Control, endpoint, and path constraints 193
9.9 Sparse nonlinear programming structure 194
9.10 Scaling, initialization, and continuation 194
9.10.1 Scaling . . . 194
9.10.2 Initial guess . . . 194
9.10.3 Continuation . . . 195
9.11 Mesh refinement and error estimation 195
9.12 Diagnostics and verification 195
9.12.1 Primal feasibility . . . 195
9.12.2 Discrete optimality . . . 195
9.12.3 Dynamic verification . . . 196
9.12.4 Adjoint consistency . . . 196
9.12.5 Sensitivity and initialization . . . 196
9.13 Choosing a numerical formulation 196
9.14 Implementation checklist 197
9.15 Exercises 197
9.15.1 Fundamental . . . 197
9.15.2 Computational . . . 197
9.15.3 Advanced . . . 197

V Python Toolchain

10 Configuring Python with Miniconda . . . 203
10.1 Why use a dedicated environment? 203
10.2 Install Miniconda 203
10.2.1 Windows . . . 203
10.2.2 macOS . . . 203
10.2.3 Linux . . . 203

10.3 Create the NOOCDS environment **204**
10.4 Run JupyterLab **204**
10.5 Optional: Visual Studio Code **204**
10.6 Updating or rebuilding the environment **205**
10.7 Common problems **205**

11 Notebook Catalogue and Reproducibility 207
11.1 Download the complete collection **207**
11.2 Learning path **207**
11.3 Individual notebooks **208**
11.3.1 01 — Unconstrained minimization 208
11.3.2 02 — Constrained minimization 208
11.3.3 03 — Numerical ODE solvers 208
11.3.4 04 — Linear DAEs 208
11.3.5 05 — Nonlinear DAEs 209
11.3.6 06 — DAE index analysis and reduction 209
11.3.7 07 — Nine DAE case studies 209
11.3.8 08 — Numerical methods for DAEs 209
11.4 Running a notebook **209**
11.5 A disciplined experiment **209**

Appendices

Contents

This volume collects the numerical methods of the course *Numerical optimization and optimal control for dynamical systems*: the algorithms, their order and stability properties, and the Python laboratories that reproduce them.

The mathematical theory they rest on is in the companion volume, the course notes, whose parts and chapter names this book mirrors; exercises and worked examples are in Exercises and worked examples.

- **Constrained Minimization**
  - Optimization Algorithms
    - Unconstrained Optimization
    - Constrained Optimization
- **Ordinary Differential Equations**
  - Numerical Methods for IVPs
  - Taylor Methods for ODEs
  - Runge–Kutta Methods
  - Linear Multistep Methods
  - Absolute Stability and Stability Regions
  - Checking the Order of a RK Method
- **Differential-Algebraic Equations**
  - Numerical Methods for DAEs
- **Optimal Control**
  - Numerical Methods for Optimal Control
- **Python Toolchain**
  - Configuring Python with Miniconda
  - Notebook Catalogue and Reproducibility

I Constrained Minimization

1 Optimization Algorithms 7
1.1 Unconstrained Optimization
1.2 Constrained Optimization

Numerical algorithms for unconstrained and constrained minimization: line searches, Newton and quasi-Newton steps, active-set and interior-point methods, and the KKT systems they solve. The theory is in the part of the same name of the course notes.

1. Optimization Algorithms

This chapter turns the optimality conditions of the course notes into algorithms. The first section treats smooth unconstrained minimization and supplies the globalization and quasi-Newton tools reused in the second, which develops methods for constrained nonlinear programming.

1.1 Unconstrained Optimization

Consider the smooth optimization problem

$$\min_{x \in \mathbb{R}^n} f(x). \tag{1.1}$$

The optimality conditions developed earlier in this part describe a solution. This section addresses the complementary computational question: how should an iterate be moved toward a stationary point, and how can one decide whether the computed result is trustworthy?

The presentation separates three ingredients that every practical method must specify:

1. a **local model** of the objective;
2. a **step computation** based on that model;
3. a **globalization and stopping strategy**.

After this section, the downloadable unconstrained-minimization laboratory can be used to compare these choices on classical test problems.

Learning objectives

After studying this section, the reader should be able to

- distinguish local convergence from convergence from a remote initial point;
- explain how conditioning affects gradient and Newton iterations;
- apply Armijo and Wolfe line searches;

- construct modified-Newton and BFGS directions;
- explain the acceptance test of a trust-region method;
- interpret termination tests and common solver failure modes.

Throughout this section, the analytical gradient $\nabla f(\boldsymbol{x})$ remains a row vector, as established in the mathematical preliminaries. For numerical linear algebra we use the column vector

$$\boldsymbol{q}(\boldsymbol{x}) := \nabla f(\boldsymbol{x})^T, \qquad \boldsymbol{H}(\boldsymbol{x}) := \nabla^2 f(\boldsymbol{x}).$$

1.1.1 From optimality conditions to an iteration

If f is differentiable and $\boldsymbol{x}^\star$ is an unconstrained local minimizer, then

$$\boldsymbol{q}(\boldsymbol{x}^\star) = \boldsymbol{0}.$$

If f is twice differentiable, a necessary second-order condition is $\boldsymbol{H}(\boldsymbol{x}^\star) \succeq \boldsymbol{0}$. The sufficient condition $\boldsymbol{H}(\boldsymbol{x}^\star) \succ \boldsymbol{0}$ implies a strict local minimum and local quadratic growth.

These facts suggest solving the nonlinear system $\boldsymbol{q}(\boldsymbol{x}) = \boldsymbol{0}$. Optimization algorithms exploit more structure than a generic root solver: they also seek a reduction of f and can use negative curvature to escape points that are not minima.

Most smooth methods have the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \tag{1.2}$$

where $\boldsymbol{p}_k$ is a search direction and $\alpha_k > 0$ is a step length. A direction is a **descent direction** when

$$\boldsymbol{q}_k^T \boldsymbol{p}_k < 0. \tag{1.3}$$

Indeed, differentiability gives

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) = f(\boldsymbol{x}_k) + \alpha \boldsymbol{q}_k^T \boldsymbol{p}_k + o(\alpha),$$

so every sufficiently small positive step decreases the objective. This is a local statement; it does not say that a full step $\alpha = 1$ is safe.

1.1.2 Conditioning and scaling

The quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T \boldsymbol{x}, \qquad \boldsymbol{A} \succ \boldsymbol{0}, \tag{1.4}$$

is the basic model for understanding conditioning. Its minimizer solves $\boldsymbol{A}\boldsymbol{x}^\star = \boldsymbol{b}$. Let

$$0 < \lambda_{\min} \le \lambda_{\max}$$

be the extreme eigenvalues of $\boldsymbol{A}$. The spectral condition number is

$$\kappa_2(\boldsymbol{A}) = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

For steepest descent with an exact line search, the error in the $\boldsymbol{A}$-norm satisfies

$$\frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^\star\|_{\boldsymbol{A}}}{\|\boldsymbol{x}_k - \boldsymbol{x}^\star\|_{\boldsymbol{A}}} \le \frac{\kappa_2(\boldsymbol{A}) - 1}{\kappa_2(\boldsymbol{A}) + 1}. \tag{1.5}$$

When $\kappa_2(\boldsymbol{A})$ is large, the factor is close to one and the iterates zigzag across a narrow valley. Poor conditioning is therefore not merely a linear-algebra issue: it directly controls algorithmic progress.

Variable scaling replaces $\boldsymbol{x} = \boldsymbol{D}\boldsymbol{z}$ and minimizes $\widehat{f}(\boldsymbol{z}) = f(\boldsymbol{D}\boldsymbol{z})$. A good diagonal scaling makes typical components of $\boldsymbol{z}$ and the curvature along coordinate directions comparable. It also makes step and gradient tolerances more meaningful.

> **Scaling is part of the model**
>
> Multiplying the objective by a constant does not change its minimizer, but it does change raw function and gradient magnitudes. Similarly, changing the units of one variable changes the apparent conditioning. Solver tolerances must therefore be interpreted together with the scaling of the model.

1.1.3 Convergence rates

Let $e_k = \|\boldsymbol{x}_k - \boldsymbol{x}^\star\|$. The sequence converges with order $p \geq 1$ if, asymptotically,

$$e_{k+1} \leq C e_k^p \tag{1.6}$$

for some constant $C > 0$. The main cases are:

Rate	Characterization	Typical method near a regular minimizer
Linear	$e_{k+1} \leq \rho e_k$, $0 < \rho < 1$	gradient method
Superlinear	$e_{k+1}/e_k \to 0$	BFGS with suitable line search
Quadratic	$e_{k+1} \leq C e_k^2$	exact Newton method

These are **local** rates. They describe behavior after the iterates enter a neighborhood in which the model is accurate. They do not guarantee that a method started far from $\boldsymbol{x}^\star$ will reach that neighborhood.

A global convergence theorem normally states that every accumulation point is stationary, or that $\|\boldsymbol{q}_k\| \to 0$, under specified assumptions. It does not mean convergence to a global minimizer of a nonconvex function.

1.1.4 Gradient methods

The steepest-descent direction in the Euclidean norm is

$$\boldsymbol{p}_k = -\boldsymbol{q}_k.$$

If the gradient is L-Lipschitz continuous, the descent lemma gives

$$f(\boldsymbol{x}_k - \alpha \boldsymbol{q}_k) \leq f(\boldsymbol{x}_k) - \alpha \left(1 - \frac{L\alpha}{2}\right) \|\boldsymbol{q}_k\|^2.$$

Thus every fixed step $0 < \alpha < 2/L$ decreases the objective unless $\boldsymbol{q}_k = \boldsymbol{0}$. The estimate is useful theoretically, but L is rarely known accurately. A line search estimates an acceptable scale from function values.

Gradient descent is inexpensive and robust but uses no curvature information. It is useful for large problems and as a safeguard, although it can be very slow on badly conditioned objectives.

1.1.5 Line-search globalization

Given a descent direction $\boldsymbol{p}_k$, a line search considers

$$\phi_k(\alpha) = f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k).$$

Exact minimization of ϕ_k is normally unnecessary. Practical methods seek a step that is inexpensive to compute and sufficiently productive.

1.1.5.1 Armijo sufficient decrease

The Armijo condition is

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \le f(\boldsymbol{x}_k) + c_1 \alpha \boldsymbol{q}_k^T \boldsymbol{p}_k, \qquad 0 < c_1 < 1. \tag{1.7}$$

Since $\boldsymbol{q}_k^T \boldsymbol{p}_k < 0$, the right-hand side is smaller than $f(\boldsymbol{x}_k)$. Backtracking begins with $\alpha = 1$ and repeatedly replaces α by $\beta\alpha$, with $0 < \beta < 1$, until Equation 1.7 holds. Common choices are $c_1 = 10^{-4}$ and $\beta \in [0.1, 0.8]$ [@armijo1966minimization].

Armijo decrease prevents steps that are too long, but by itself it can accept steps that are unnecessarily short.

1.1.5.2 Wolfe curvature conditions

The Wolfe conditions add

$$\boldsymbol{q}(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)^T \boldsymbol{p}_k \ge c_2 \boldsymbol{q}_k^T \boldsymbol{p}_k, \qquad c_1 < c_2 < 1. \tag{1.8}$$

The strong Wolfe condition instead requires

$$\left|\boldsymbol{q}(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)^T \boldsymbol{p}_k\right| \le c_2 |\boldsymbol{q}_k^T \boldsymbol{p}_k|. \tag{1.9}$$

The curvature test ensures that the directional derivative has changed sufficiently. It is especially important for quasi-Newton methods because it helps preserve positive curvature in the BFGS update.

1.1.5.3 Backtracking line search

1. Choose a descent direction $\boldsymbol{p}_k$ and set $\alpha \leftarrow 1$.
2. While the Armijo condition fails, set $\alpha \leftarrow \beta\alpha$.
3. Accept $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \boldsymbol{p}_k$.

The full step should be tested first. Near a regular minimizer, Newton and BFGS methods need to recover $\alpha = 1$ in order to exhibit their fast local rates.

1.1.6 Newton and modified-Newton methods

At $\boldsymbol{x}_k$, the quadratic Taylor model is

$$m_k(\boldsymbol{p}) = f(\boldsymbol{x}_k) + \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{H}_k \boldsymbol{p}. \tag{1.10}$$

If $\boldsymbol{H}_k$ is nonsingular, stationarity of the model gives

$$\boldsymbol{H}_k \boldsymbol{p}_k = -\boldsymbol{q}_k. \tag{1.11}$$

Near a minimizer at which $\boldsymbol{H}(\boldsymbol{x}^\star) \succ \boldsymbol{0}$, the full Newton step is well defined and converges quadratically, provided the Hessian is sufficiently regular. Far from the solution, however, the Hessian can be indefinite and the Newton direction need not be a descent direction.

A modified-Newton method replaces $\boldsymbol{H}_k$ with a positive-definite matrix

$$\boldsymbol{B}_k = \boldsymbol{H}_k + \tau_k \boldsymbol{I}, \qquad \tau_k \geq 0, \tag{1.12}$$

and solves $\boldsymbol{B}_k \boldsymbol{p}_k = -\boldsymbol{q}_k$. The shift can be increased until a Cholesky factorization succeeds. Then

$$\boldsymbol{q}_k^T \boldsymbol{p}_k = -\boldsymbol{p}_k^T \boldsymbol{B}_k \boldsymbol{p}_k < 0$$

for every nonzero step. This is a modified-Newton or Levenberg–Marquardt-type regularization [@marquardt1963algorithm]. A line search supplies globalization; close to a well-conditioned minimizer the shift should return to zero.

1.1.7 BFGS and limited-memory methods

Quasi-Newton methods replace the exact Hessian by an approximation. Define

$$\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k, \qquad \boldsymbol{y}_k = \boldsymbol{q}_{k+1} - \boldsymbol{q}_k.$$

The secant equation

$$\boldsymbol{B}_{k+1} \boldsymbol{s}_k = \boldsymbol{y}_k \tag{1.13}$$

requires the new matrix to reproduce the observed change of gradient. The BFGS update is

$$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k - \frac{\boldsymbol{B}_k \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{B}_k}{\boldsymbol{s}_k^T \boldsymbol{B}_k \boldsymbol{s}_k} + \frac{\boldsymbol{y}_k \boldsymbol{y}_k^T}{\boldsymbol{y}_k^T \boldsymbol{s}_k}. \tag{1.14}$$

If $\boldsymbol{B}_k \succ \boldsymbol{0}$ and $\boldsymbol{y}_k^T \boldsymbol{s}_k > 0$, then $\boldsymbol{B}_{k+1} \succ \boldsymbol{0}$. A Wolfe line search normally enforces this curvature condition. The search direction is obtained from

$$\boldsymbol{B}_k \boldsymbol{p}_k = -\boldsymbol{q}_k.$$

BFGS avoids exact second derivatives and generally achieves superlinear local convergence. Limited-memory BFGS stores only a small number of recent pairs $(\boldsymbol{s}_k, \boldsymbol{y}_k)$, making it suitable for large problems. Its lower memory cost comes at the price of a less complete curvature model.

1.1.8 Trust-region globalization

A trust-region method computes the step by approximately solving

$$\min_{\|\boldsymbol{p}\| \leq \Delta_k} m_k(\boldsymbol{p}) = f_k + \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{B}_k \boldsymbol{p}. \tag{1.15}$$

The radius Δ_k defines the region in which the local model is trusted. After computing a trial step, the agreement between model and objective is measured by

$$\rho_k = \frac{f(\boldsymbol{x}_k) - f(\boldsymbol{x}_k + \boldsymbol{p}_k)}{m_k(\boldsymbol{0}) - m_k(\boldsymbol{p}_k)}. \tag{1.16}$$

- If ρ_k is small or negative, the model is poor: reject the step and reduce Δ_k.

- If ρ_k is moderate, accept the step and keep a comparable radius.
- If ρ_k is close to one and the step reaches the boundary, accept it and enlarge Δ_k.

Unlike a line search, a trust region does not commit to a direction before choosing the step size. It can exploit directions of negative curvature and is therefore particularly effective for indefinite Hessians. Dogleg methods are convenient for positive-definite models; truncated conjugate gradients are attractive for large systems and can stop when negative curvature is detected.

1.1.9 Stopping tests and diagnostic evidence

No single floating-point test proves that an iterate is a minimizer. A robust implementation records several diagnostics.

1.1.9.1 First-order stationarity

A scaled gradient test can be written as

$$\|\boldsymbol{q}_k\|_\infty \le \varepsilon_a + \varepsilon_r \|\boldsymbol{q}_0\|_\infty. \tag{1.17}$$

The scaling should be chosen consistently with the units of the variables. Comparing $\|\boldsymbol{q}_k\|$ directly with $1 + |f_k|$ is dimensionally questionable because the gradient scales as objective divided by variable.

1.1.9.2 Step and objective changes

Small values of

$$\frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\|}{1 + \|\boldsymbol{x}_k\|} \quad \text{and} \quad \frac{|f_{k+1} - f_k|}{1 + |f_k|}$$

are useful secondary tests. They are not sufficient by themselves: a line search can produce a tiny step because the direction is poor, and a flat or badly scaled objective can change very little far from stationarity.

1.1.9.3 Second-order evidence

At a reported solution, inspect the smallest eigenvalue of the Hessian when this is computationally feasible. A small gradient with a negative Hessian eigenvalue indicates a saddle point, not a local minimum. Near-zero eigenvalues indicate degeneracy and make the observed convergence rate sensitive to roundoff and scaling.

1.1.9.4 Derivative verification

Before diagnosing an algorithm, verify the derivatives. For a normalized direction $\boldsymbol{d}$, compare

$$\nabla f(\boldsymbol{x})\boldsymbol{d} \quad \text{with} \quad \frac{f(\boldsymbol{x} + h\boldsymbol{d}) - f(\boldsymbol{x} - h\boldsymbol{d})}{2h}.$$

The discrepancy should initially decrease like h^2 and then increase when roundoff dominates. Testing several random directions is more informative than checking coordinate directions alone.

> **!** A solver status is not a certificate
>
> A success flag, a small step, agreement between two solvers, or a small gradient taken separately is not conclusive. The interpretation should combine stationarity, curvature,

scaling, iteration history, and sensitivity to the initial point.

1.1.10 Choosing a method

Situation	Suitable first choice	Main diagnostic
Small smooth problem, accurate Hessian	damped or trust-region Newton	Hessian definiteness and full-step recovery
Smooth problem, Hessian expensive	BFGS	curvature pairs and gradient norm
Large smooth problem	limited-memory BFGS	projected/scaled gradient and line-search history
Hessian-vector products available	Newton–CG	residual and negative-curvature detection
Strong nonconvexity	trust-region Newton	ratio of actual to predicted reduction
Poorly scaled variables	rescale before selecting the method	componentwise steps and gradients

The method should match the information available. Supplying an inaccurate Hessian to Newton's method can be worse than using a carefully safeguarded quasi-Newton approximation.

1.1.11 Computational laboratory

The notebook **01 – Unconstrained minimization** implements gradient, Newton, modified-Newton, BFGS, and SciPy methods. It records iteration histories and examines Rosenbrock, Himmelblau, Beale, Powell singular, Brown badly scaled, and Freudenstein–Roth problems. Its purpose is to test the theory above rather than to replace it.

Download: unconstrained-minimization algorithms and diagnostic experiments

Suggested reading order:

1. predict the behavior from conditioning and curvature;
2. run the reference experiment unchanged;
3. inspect gradient norms, accepted step lengths, and Hessian information;
4. change one parameter or initial point at a time;
5. explain failures using the diagnostics rather than the status flag alone.

1.1.12 Exercises

The labels indicate the intended level: **F** fundamental, **C** computational, and **A** advanced. Each item includes a short starting hint.

1. **F.** For the quadratic problem Equation 1.4, derive the exact line-search step for the steepest-descent direction. *Hint:* minimize the one-variable quadratic $f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)$.
2. **F.** Show that a positive-definite modified-Newton matrix produces a descent direction whenever $\boldsymbol{q}_k \neq \mathbf{0}$. *Hint:* premultiply the Newton equation by the direction transpose.

3. **F.** Explain why Armijo backtracking alone does not guarantee $\boldsymbol{y}_k^T \boldsymbol{s}_k > 0$ for a BFGS update. *Hint:* compare Armijo with the curvature Wolfe condition.
4. **C.** Apply one BFGS update to a two-dimensional quadratic and verify the secant equation Equation 1.13. *Hint:* compute the matrix-vector product after the update, not only the updated entries.
5. **A.** For an indefinite quadratic model, compare the Newton step with the solution of Equation 1.15. *Hint:* inspect the most negative curvature direction.
6. **A.** Construct an example in which a relative objective-change test terminates far from a stationary point. *Hint:* rescale or offset the objective while leaving its gradient appreciable.
7. **C.** Use directional finite differences to determine an appropriate range of h for derivative checking in double precision. *Hint:* plot the error against h on logarithmic axes.
8. **C.** In the computational laboratory, compare the convergence history from two distinct attraction basins of the Himmelblau function. *Hint:* retain the complete iteration history and not only the final point.

1.2 Constrained Optimization

The optimality conditions developed in the preceding sections characterize candidate minimizers. This section turns the constrained conditions into numerical algorithms, building on the local models, line searches, quasi-Newton updates, and trust-region ideas developed in Section 1.1. The presentation follows the standard globalization framework of @nocedal2006numerical, @bertsekas2016nonlinear, and @fletcher2000practical.

Throughout this section, gradients used in linear algebra formulas are column vectors. Thus, if the analytical convention established in the mathematical preliminaries writes $\nabla f(\boldsymbol{x})$ as a row vector, here

$$\boldsymbol{q}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^T.$$

The symbol $\boldsymbol{g}$ is reserved for inequality constraints, whereas $\boldsymbol{q}$ always denotes the objective gradient. Throughout, n is the number of variables, m_h the number of equality constraints, and m_g the number of inequality constraints; $\boldsymbol{J}_h$ and $\boldsymbol{J}_g$ are the corresponding Jacobians.

1.2.1 The NLP viewpoint and algorithmic map

A smooth nonlinear programming problem (NLP) can be written in the form

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & f(\boldsymbol{x}), \\ \text{subject to} \quad & \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}, \\ & \boldsymbol{g}(\boldsymbol{x}) \geq \boldsymbol{0}, \\ & \boldsymbol{\ell} \leq \boldsymbol{x} \leq \boldsymbol{u}. \end{aligned} \tag{1.18}$$

The numerical methods in this section should not be viewed as unrelated algorithms. Most modern smooth NLP methods are different ways of solving, regularizing, or approximating the KKT equations. They differ mainly in four choices:

1. how feasibility is enforced or approached;
2. how complementarity for inequalities is handled;

3. how the Lagrangian Hessian is represented;
4. how the local Newton model is globalized away from the solution.

This viewpoint leads to the following map.

Family	Treatment of constraints	Local model	Typical globalization
Projected methods	explicit projection	gradient/Newton	line search
Penalty methods	penalize violation	unconstrained	penalty continuation
Augmented Lagrangian	multipliers + finite penalty	unconstrained/bound constrained	outer multiplier/penalty update
Active-set SQP	predict active inequalities	QP	merit/filter line search or trust region
Interior-point	perturbed complementarity	primal-dual KKT system	barrier continuation + filter/line search
Sequential convex methods	convex local approximation	LP/QP/convex program	trust region/filter

The common computational core is frequently a saddle-point system

$$\begin{bmatrix} \boldsymbol{B} & \boldsymbol{A}^T \\ \boldsymbol{A} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \end{bmatrix}, \tag{1.19}$$

possibly after elimination of slack variables. Understanding this structure is therefore more important than memorizing individual solver names.

1.2.1.1 Local models and globalization

A constrained Newton model is reliable only locally. A practical NLP method therefore has two layers:

- a **local model**, usually Newton, quasi-Newton, QP, or a primal-dual Newton system;
- a **globalization mechanism**, usually a merit-function line search, a filter, a trust region, or a continuation parameter.

Near a regular solution the globalization should become inactive so that the fast local convergence of the underlying Newton-type iteration is recovered.

1.2.2 Unconstrained methods as the local computational model

The unconstrained algorithms developed in Section 1.1 provide the local computational engine for many constrained methods below. Penalty and augmented-Lagrangian approaches repeatedly solve unconstrained subproblems, while SQP and interior-point methods reuse Newton, quasi-Newton, line-search, and trust-region principles on constrained local models.
Simple constraints: projection and bound constraints

When the feasible set Ω is simple enough that the Euclidean projection

$$P_\Omega(\boldsymbol{z}) = \operatorname*{argmin}_{\boldsymbol{x}\in\Omega} \frac{1}{2}\|\boldsymbol{x}-\boldsymbol{z}\|^2$$

is inexpensive, a projected-gradient iteration is

$$\boldsymbol{x}_{k+1} = P_\Omega(\boldsymbol{x}_k - \alpha_k \boldsymbol{q}_k)\,. \tag{1.20}$$

For box constraints, projection is componentwise,

$$[P_\Omega(\boldsymbol{z})]_i = \min\{u_i, \max\{\ell_i, z_i\}\}.$$

The natural first-order residual is the **gradient mapping**

$$\boldsymbol{G}_\alpha(\boldsymbol{x}) = \frac{1}{\alpha}\left[\boldsymbol{x} - P_\Omega(\boldsymbol{x} - \alpha \boldsymbol{q}(\boldsymbol{x}))\right].$$

Thus $\boldsymbol{G}_\alpha(\boldsymbol{x}) = \boldsymbol{0}$ characterizes first-order stationarity for a closed convex feasible set. Projected Newton and active-set bound-constrained methods replace the gradient step by a second-order step on the currently free variables.

1.2.3 Penalty methods and exact penalties

A quadratic exterior penalty for equality constraints is

$$P_\rho(\boldsymbol{x}) = f(\boldsymbol{x}) + \frac{\rho}{2}\|\boldsymbol{h}(\boldsymbol{x})\|_2^2. \tag{1.21}$$

For inequalities $\boldsymbol{g}(\boldsymbol{x}) \geq \boldsymbol{0}$, one may add

$$\frac{\rho}{2}\|\min\{\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{0}\}\|_2^2.$$

If the inner minimizations are sufficiently accurate and $\rho \to \infty$, limit points can approach constrained solutions under standard regularity assumptions. The drawback is visible from the Hessian: near feasibility it contains the term

$$\rho \boldsymbol{J}_h^T \boldsymbol{J}_h,$$

so increasing ρ can make the subproblem severely ill-conditioned.

An important alternative is the nonsmooth exact ℓ_1 penalty

$$P_\rho^{(1)}(\boldsymbol{x}) = f(\boldsymbol{x}) + \rho\|\boldsymbol{h}(\boldsymbol{x})\|_1 + \rho\|\min\{\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{0}\}\|_1. \tag{1.22}$$

Under suitable constraint qualifications, a finite sufficiently large ρ is enough for a local constrained minimizer to be a local minimizer of the penalty function. This exactness explains why ℓ_1 functions are also useful as merit functions in SQP methods.

1.2.3.1 The multiplier hidden in the quadratic penalty

The quadratic penalty is not merely a crude device: it already contains a multiplier estimate, and seeing it explains the augmented Lagrangian of Section 1.2.7. Stationarity of Equation 1.21 gives

$$\boldsymbol{q}(\boldsymbol{x}_\rho) + \rho \boldsymbol{J}_h(\boldsymbol{x}_\rho)^T \boldsymbol{h}(\boldsymbol{x}_\rho) = \boldsymbol{0}, \qquad \text{that is} \qquad \boldsymbol{q}(\boldsymbol{x}_\rho) = \boldsymbol{J}_h(\boldsymbol{x}_\rho)^T\big(-\rho \boldsymbol{h}(\boldsymbol{x}_\rho)\big).$$

Comparing with the stationarity condition $\boldsymbol{q} = \boldsymbol{J}_h^T \boldsymbol{\lambda}$ of the constrained problem shows that

$$\boxed{\boldsymbol{\lambda} \approx -\rho\, \boldsymbol{h}(\boldsymbol{x}_\rho).} \tag{1.23}$$

The multiplier is recovered only in the limit, because it is obtained by multiplying a quantity that tends to zero by one that tends to infinity: this product is exactly the source of the ill-conditioning. The augmented Lagrangian removes the difficulty by carrying $\boldsymbol{\lambda}$ as an explicit variable and using Equation 1.23 only as a *correction* to the current estimate, which keeps ρ finite.

1.2.4 Equality-constrained minimization

Consider

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x), \\ \text{subject to} \quad & h(x) = 0, \end{aligned}$$

with $h : \mathbb{R}^n \to \mathbb{R}^{m_h}$. Consistently with the preceding KKT chapter, define

$$\mathcal{L}(x, \lambda) = f(x) - \lambda^T h(x),$$

and let $J(x)$ be the Jacobian of h. The first-order system is

$$F(x, \lambda) = \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda)^T \\ h(x) \end{bmatrix} = 0. \tag{1.24}$$

1.2.4.1 Newton's method for the KKT system

The Jacobian of Equation 1.24, written out in full in Equation 1.26 below, has lower-left block J_k and upper-right block $-J_k^T$. For numerical factorization it is useful to multiply the feasibility equation by -1. This row scaling does not change the Newton step and gives the symmetric-indefinite system

$$\begin{bmatrix} W_k & -J_k^T \\ -J_k & 0 \end{bmatrix} \begin{bmatrix} p_k \\ \delta\lambda_k \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L}_k^T \\ h_k \end{bmatrix}, \tag{1.25}$$

where

$$W_k = \nabla_{xx}^2 \mathcal{L}(x_k, \lambda_k).$$

The second block row is still the linearized feasibility equation $J_k p_k = -h_k$. The coefficient matrix is symmetric but indefinite; a Cholesky factorization of the complete KKT matrix is therefore not appropriate. Without the row scaling, the raw Newton Jacobian Equation 1.26 is algebraically equivalent but nonsymmetric.

If LICQ holds and the reduced Hessian is positive definite, the KKT matrix is nonsingular near the solution. The full Newton step is then locally quadratically convergent.

1.2.4.2 SQP as Newton's method applied to the KKT equations

The relation between Newton's method and SQP is exact, not merely heuristic. For the equality-constrained problem define the primal–dual variable

$$z = \begin{bmatrix} x \\ \lambda \end{bmatrix}, \qquad F(z) = \begin{bmatrix} q(x) - J(x)^T \lambda \\ h(x) \end{bmatrix}.$$

A KKT point is a zero of F. Its Jacobian is

$$F'(z) = \begin{bmatrix} \nabla_{xx}^2 \mathcal{L}(x, \lambda) & -J(x)^T \\ J(x) & 0 \end{bmatrix}. \tag{1.26}$$

Newton's method for $F(z) = 0$ computes $\Delta z_k = (p_k, \delta\lambda_k)$ from

$$F'(z_k) \Delta z_k = -F(z_k).$$

Writing the first block row explicitly gives

$$W_k p_k - J_k^T \delta\lambda_k = -q_k + J_k^T \lambda_k,$$

or, after defining the new multiplier estimate

$$\widehat{\lambda}_{k+1} = \lambda_k + \delta\lambda_k,$$

$$\boldsymbol{W}_k \boldsymbol{p}_k - \boldsymbol{J}_k^T \widehat{\boldsymbol{\lambda}}_{k+1} = -\boldsymbol{q}_k. \tag{1.27}$$

The second block row is

$$\boldsymbol{J}_k \boldsymbol{p}_k = -\boldsymbol{h}_k. \tag{1.28}$$

Now consider the quadratic program

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & q_k(\boldsymbol{p}) = \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{W}_k \boldsymbol{p}, \\ \text{subject to} \quad & \boldsymbol{h}_k + \boldsymbol{J}_k \boldsymbol{p} = \boldsymbol{0}. \end{aligned} \tag{1.29}$$

Its Lagrangian is

$$\mathcal{L}_k^{\mathrm{QP}}(\boldsymbol{p}, \widehat{\boldsymbol{\lambda}}) = \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{W}_k \boldsymbol{p} - \widehat{\boldsymbol{\lambda}}^T (\boldsymbol{h}_k + \boldsymbol{J}_k \boldsymbol{p}).$$

The QP KKT equations are precisely Equation 1.27 and Equation 1.28. Multiplying the latter by -1 produces the symmetric system Equation 1.25. Consequently,

exact-Hessian SQP step ≡ Newton step for the nonlinear KKT equations.

The distinction is conceptual rather than algebraic. Newton views the problem as a nonlinear system in $(\boldsymbol{x}, \boldsymbol{\lambda})$; SQP views the same equations as the optimality conditions of a local quadratic optimization problem. The SQP viewpoint is particularly valuable because inequalities and bounds can be retained explicitly in the QP subproblem.

1.2.4.3 Why the Lagrangian Hessian appears

It is important not to replace $\boldsymbol{W}_k$ blindly by $\nabla^2 f(\boldsymbol{x}_k)$. Differentiating stationarity gives

$$\boldsymbol{W}_k = \nabla^2 f(\boldsymbol{x}_k) - \sum_{i=1}^{m_h} \lambda_{k,i} \nabla^2 h_i(\boldsymbol{x}_k).$$

Thus the curvature of the constraint manifold contributes to the Newton model. Omitting these terms generally destroys exact Newton equivalence and can reduce the local convergence rate.

1.2.4.4 Nonsingularity of the KKT Jacobian

Let $\boldsymbol{z}^\star = (\boldsymbol{x}^\star, \boldsymbol{\lambda}^\star)$ be a KKT point. Suppose $\boldsymbol{J}_\star$ has full row rank (LICQ), and let the columns of $\boldsymbol{Z}_\star$ form a basis for the tangent space

$$\mathcal{T}_\star = \ker(\boldsymbol{J}_\star).$$

Assume the second-order sufficient condition

$$\boldsymbol{d}^T \boldsymbol{W}_\star \boldsymbol{d} > 0 \qquad \text{for every } \boldsymbol{d} \neq \boldsymbol{0} \text{ with } \boldsymbol{J}_\star \boldsymbol{d} = \boldsymbol{0}, \tag{1.30}$$

or equivalently

$$\boldsymbol{Z}_\star^T \boldsymbol{W}_\star \boldsymbol{Z}_\star \succ \boldsymbol{0}.$$

Then the KKT Jacobian is nonsingular. Indeed, if

$$\begin{bmatrix} \boldsymbol{W}_\star & -\boldsymbol{J}_\star^T \\ \boldsymbol{J}_\star & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{v} \end{bmatrix} = \boldsymbol{0},$$

then $\boldsymbol{J}_\star \boldsymbol{d} = \boldsymbol{0}$. Multiplying the first row by $\boldsymbol{d}^T$ gives $\boldsymbol{d}^T \boldsymbol{W}_\star \boldsymbol{d} = 0$, because $\boldsymbol{d}^T \boldsymbol{J}_\star^T \boldsymbol{v} = 0$. By Equation 1.30, $\boldsymbol{d} = \boldsymbol{0}$. The first row then gives $\boldsymbol{J}_\star^T \boldsymbol{v} = \boldsymbol{0}$, and full row rank of $\boldsymbol{J}_\star$ implies $\boldsymbol{v} = \boldsymbol{0}$.

1.2.4.5 Local quadratic convergence of exact-Hessian SQP

The preceding equivalence permits the standard Newton convergence theorem to be applied directly.

Theorem 1.2.1 — Local quadratic convergence of SQP. Suppose that f and $\boldsymbol{h}$ are sufficiently smooth in a neighborhood of a KKT point $(\boldsymbol{x}^\star, \boldsymbol{\lambda}^\star)$, LICQ holds there, the second-order sufficient condition Equation 1.30 holds, and $\boldsymbol{F}'$ is locally Lipschitz continuous. Then there exists a neighborhood $\mathcal{N}$ of $(\boldsymbol{x}^\star, \boldsymbol{\lambda}^\star)$ such that, for every $(\boldsymbol{x}_0, \boldsymbol{\lambda}_0) \in \mathcal{N}$, exact-Hessian SQP with full steps is well defined and

$$\left\| \begin{bmatrix} \boldsymbol{x}_{k+1} - \boldsymbol{x}^\star \\ \boldsymbol{\lambda}_{k+1} - \boldsymbol{\lambda}^\star \end{bmatrix} \right\| \le C \left\| \begin{bmatrix} \boldsymbol{x}_k - \boldsymbol{x}^\star \\ \boldsymbol{\lambda}_k - \boldsymbol{\lambda}^\star \end{bmatrix} \right\|^2 \tag{1.31}$$

for every $k \ge 0$ and some constant $C > 0$.

To see the mechanism, write

$$\boldsymbol{e}_k = \boldsymbol{z}_k - \boldsymbol{z}^\star.$$

Newton's update is

$$\boldsymbol{e}_{k+1} = \boldsymbol{e}_k - \boldsymbol{F}'(\boldsymbol{z}_k)^{-1}\boldsymbol{F}(\boldsymbol{z}_k).$$

Since $\boldsymbol{F}(\boldsymbol{z}^\star) = \boldsymbol{0}$,

$$\boldsymbol{F}(\boldsymbol{z}_k) = \int_0^1 \boldsymbol{F}'(\boldsymbol{z}^\star + t\boldsymbol{e}_k)\boldsymbol{e}_k \, dt.$$

Hence

$$\boldsymbol{e}_{k+1} = \boldsymbol{F}'(\boldsymbol{z}_k)^{-1} \int_0^1 \left[\boldsymbol{F}'(\boldsymbol{z}_k) - \boldsymbol{F}'(\boldsymbol{z}^\star + t\boldsymbol{e}_k)\right] \boldsymbol{e}_k \, dt.$$

Local nonsingularity makes $\|\boldsymbol{F}'(\boldsymbol{z}_k)^{-1}\|$ uniformly bounded, and Lipschitz continuity gives

$$\left\|\boldsymbol{F}'(\boldsymbol{z}_k) - \boldsymbol{F}'(\boldsymbol{z}^\star + t\boldsymbol{e}_k)\right\| \le L(1-t)\|\boldsymbol{e}_k\|.$$

Integration therefore yields

$$\|\boldsymbol{e}_{k+1}\| \le \frac{ML}{2}\|\boldsymbol{e}_k\|^2,$$

which proves Equation 1.31.

For inequality-constrained NLP, once the correct active set has been identified, the locally active inequalities can be appended to $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$. Under LICQ, a suitable second-order sufficient condition, and nondegeneracy such as strict complementarity, the same Newton argument applies to this active KKT system. This is the source of the local quadratic rate of exact-Hessian active-set SQP.

1.2.4.6 Quasi-Newton SQP and superlinear convergence

In practice $\boldsymbol{W}_k$ is often replaced by $\boldsymbol{B}_k$. The SQP equations are then an inexact or quasi-Newton approximation to the KKT Newton equations. Quadratic convergence is generally lost, but superlinear convergence is possible if the Hessian approximation becomes sufficiently accurate along the computed steps. A constrained analogue of the Dennis–Moré condition is, schematically,

$$\frac{\|(\boldsymbol{B}_k - \boldsymbol{W}_\star)\boldsymbol{p}_k\|}{\|\boldsymbol{p}_k\|} \longrightarrow 0,$$

with the relevant condition often required only on the tangent space; in the unconstrained case this is exactly the characterization of @dennis1974characterization. BFGS updates of the Lagrangian Hessian are designed to achieve this behavior while avoiding second derivatives.

1.2.4.7 Sequential quadratic programming

Replacing $\boldsymbol{W}_k$ by a symmetric approximation $\boldsymbol{B}_k$ produces the equality-constrained SQP subproblem

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{B}_k \boldsymbol{p}, \\ \text{subject to} \quad & \boldsymbol{h}_k + \boldsymbol{J}_k \boldsymbol{p} = \boldsymbol{0}. \end{aligned} \tag{1.32}$$

Its KKT equations have the same block structure as Equation 1.25. Hence Newton on the nonlinear KKT equations and SQP with the exact Lagrangian Hessian produce the same local step.

A line search can be based on the exact ℓ_1 merit function

$$\phi_\rho(\boldsymbol{x}) = f(\boldsymbol{x}) + \rho \|\boldsymbol{h}(\boldsymbol{x})\|_1, \tag{1.33}$$

where ρ must be large enough to balance objective reduction and feasibility improvement.

Algorithm 1: Damped Newton–SQP for equality constraints

Input: $\boldsymbol{x}_0$, $\boldsymbol{\lambda}_0$, dual and primal tolerances ε_d, ε_p
Output: An approximate KKT pair $(\boldsymbol{x}, \boldsymbol{\lambda})$

$(\boldsymbol{x}, \boldsymbol{\lambda}) \leftarrow (\boldsymbol{x}_0, \boldsymbol{\lambda}_0)$
repeat
  evaluate $\boldsymbol{h}$, $\boldsymbol{J}$ and $\boldsymbol{r}_d = \nabla f(\boldsymbol{x})^T - \boldsymbol{J}^T \boldsymbol{\lambda}$
  if $\|\boldsymbol{r}_d\| \le \varepsilon_d$ *and* $\|\boldsymbol{h}\| \le \varepsilon_p$ **then**
    return $(\boldsymbol{x}, \boldsymbol{\lambda})$
  end
  form a stabilized or BFGS approximation $\boldsymbol{B}$ of $\nabla^2_{xx} \mathcal{L}$
  solve the KKT system for $(\boldsymbol{p}, \delta\boldsymbol{\lambda})$
  choose the merit parameter ρ
  choose $\alpha \in (0, 1]$ with a merit-function or filter line search
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{p}$
  $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} + \alpha \delta\boldsymbol{\lambda}$
until *the maximum number of iterations is reached*

For linear constraints and a quadratic objective, the KKT equations are linear and the solution is obtained in one exact Newton step.

1.2.5 Implementing SQP in practice

The mathematical description of SQP is compact, but a robust implementation requires several additional decisions. A useful implementation separates the iteration into derivative evaluation, QP construction, QP solution, globalization, Hessian update, and convergence diagnostics.

1.2.5.1 Data evaluated at each major iteration

At $\boldsymbol{x}_k$ evaluate, or provide through automatic differentiation,

$$f_k, \qquad \boldsymbol{q}_k = \nabla f(\boldsymbol{x}_k)^T, \qquad \boldsymbol{h}_k, \qquad \boldsymbol{J}_{h,k},$$

and, for inequalities,

$$\boldsymbol{g}_k = \boldsymbol{g}(\boldsymbol{x}_k), \qquad \boldsymbol{J}_{g,k}.$$

If exact second derivatives are used, form

$$\boldsymbol{W}_k = \nabla^2 f(\boldsymbol{x}_k) - \sum_i \lambda_{k,i} \nabla^2 h_i(\boldsymbol{x}_k) - \sum_j \mu_{k,j} \nabla^2 g_j(\boldsymbol{x}_k). \tag{1.34}$$

For large sparse problems it is often preferable not to assemble this matrix explicitly; a solver may require only products $\boldsymbol{W}_k \boldsymbol{v}$.

1.2.5.2 Building and solving the QP

The general QP step is

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{B}_k \boldsymbol{p}, \\ \text{s.t.} \quad & \boldsymbol{J}_{h,k} \boldsymbol{p} = -\boldsymbol{h}_k, \\ & \boldsymbol{J}_{g,k} \boldsymbol{p} \geq -\boldsymbol{g}(\boldsymbol{x}_k), \\ & \boldsymbol{\ell} - \boldsymbol{x}_k \leq \boldsymbol{p} \leq \boldsymbol{u} - \boldsymbol{x}_k. \end{aligned} \tag{1.35}$$

The QP solver should return both the primal step $\boldsymbol{p}_k$ and multiplier estimates. The latter are not auxiliary bookkeeping: they are needed to form the next Lagrangian Hessian and the dual KKT residual.

For an equality-constrained QP, a direct implementation can solve

$$\begin{bmatrix} \boldsymbol{B}_k & -\boldsymbol{J}_k^T \\ -\boldsymbol{J}_k & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_k \\ \widehat{\boldsymbol{\lambda}}_{k+1} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{q}_k \\ \boldsymbol{h}_k \end{bmatrix}. \tag{1.36}$$

Do not explicitly invert this matrix. Use a symmetric-indefinite LDL^T factorization, a null-space/range-space factorization, or a suitable preconditioned Krylov method. Sparse symbolic analysis and ordering should be reused when the sparsity pattern is unchanged.

1.2.5.3 Null-space implementation

When $m \ll n$, write a step satisfying linearized feasibility as

$$\boldsymbol{p} = \boldsymbol{p}_N + \boldsymbol{Z}\boldsymbol{v},$$

where

$$\boldsymbol{J}\boldsymbol{p}_N = -\boldsymbol{h}, \qquad \boldsymbol{J}\boldsymbol{Z} = \boldsymbol{0}.$$

Substitution into the QP gives the reduced system

$$(\boldsymbol{Z}^T \boldsymbol{B} \boldsymbol{Z}) \boldsymbol{v} = -\boldsymbol{Z}^T (\boldsymbol{q} + \boldsymbol{B}\boldsymbol{p}_N). \tag{1.37}$$

This formulation makes the second-order condition transparent: only the reduced Hessian must be positive definite. In numerical software, an explicit basis $\boldsymbol{Z}$ is not always formed; QR factorizations or orthogonal transformations can apply the null-space projection implicitly.

The null-space method returns the primal step but not the multipliers, which are nevertheless needed for the next Lagrangian Hessian and for the dual residual. They are recovered afterwards from the first block row of Equation 1.36, which after the step has been computed reads $J^T\widehat{\lambda} = q + Bp$. Since J has full row rank this overdetermined system is consistent in exact arithmetic and is solved in the least-squares sense,

$$\widehat{\lambda} = \left(JJ^T\right)^{-1} J\,(q + Bp), \tag{1.38}$$

the **least-squares multiplier estimate**. In practice one does not form JJ^T, whose condition number is the square of that of J: the same QR factorization $J^T = QR$ used to build Z yields $\widehat{\lambda}$ by a triangular solve.

1.2.5.4 Hessian approximations

For BFGS-SQP define

$$s_k = x_{k+1} - x_k,$$

and use the Lagrangian-gradient difference

$$y_k = \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}, \mu_{k+1})^T - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}, \mu_{k+1})^T.$$

When $s_k^T y_k > 0$, the BFGS update is

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k}. \tag{1.39}$$

Away from the solution the curvature condition may fail. Damped BFGS replaces y_k by a convex combination with $B_k s_k$ so that positive curvature is enforced. Limited-memory BFGS is useful when storing a dense B_k is impossible, although preserving constrained curvature can require more care than in unconstrained optimization.

1.2.5.5 Merit-function line search

A practical ℓ_1 merit line search uses

$$\phi_\rho(x) = f(x) + \rho\theta(x),$$

where

$$\theta(x) = \|h(x)\|_1 + \|\min\{g(x), 0\}\|_1.$$

The penalty parameter should dominate the current multiplier estimates; a common conceptual rule is

$$\rho_k > \max\{\|\lambda_{k+1}^{\mathrm{QP}}\|_\infty, \|\mu_{k+1}^{\mathrm{QP}}\|_\infty\}$$

with a safety margin. Starting with $\alpha = 1$, backtrack until sufficient merit decrease is obtained. Close to a regular solution the algorithm should accept $\alpha = 1$; otherwise quadratic convergence cannot be observed.

1.2.5.6 QP infeasibility and elastic variables

The original NLP may be feasible while its local linearization is not. A robust SQP implementation therefore needs a mechanism for an infeasible QP. One possibility is an elastic formulation, for example

$$h_k + J_{h,k} p = v^+ - v^-, \qquad v^+, v^- \geq 0,$$

and a large linear penalty

$$\rho_e \boldsymbol{e}^T(\boldsymbol{v}^+ + \boldsymbol{v}^-)$$

in the QP objective. Similar slacks can be introduced for inequalities; this is the mechanism used by SNOPT [@gill2005snopt]. An alternative is to enter a restoration phase devoted to reducing constraint violation.

1.2.5.7 Regularization and inertia

The QP model need not be convex. If the goal is a local minimizer, the reduced Hessian should have the appropriate positive inertia. Practical methods may replace

$$\boldsymbol{B}_k \leftarrow \boldsymbol{B}_k + \delta_k \boldsymbol{I}$$

or regularize both diagonal blocks of the KKT system. With a direct LDL^T factorization, the inertia of the factors provides useful information about whether the model has the desired curvature.

1.2.5.8 A more complete SQP algorithm

A production-oriented SQP iteration can be summarized as follows.

1. Scale variables, objective, and constraints if their characteristic magnitudes differ strongly.
2. Evaluate objective, constraints, gradients, and Jacobians.
3. Compute primal feasibility, dual stationarity, and complementarity residuals; stop only when all required residuals are small.
4. Form the exact or quasi-Newton Lagrangian Hessian.
5. Build the QP Equation 1.35, including bounds.
6. Solve the QP and obtain both $\boldsymbol{p}_k$ and multiplier estimates.
7. If the QP is infeasible or numerically singular, regularize it, use elastic variables, or invoke restoration.
8. Test the full step first. Use a merit-function, filter, or trust-region globalization only as needed.
9. Optionally compute a second-order correction if a nearly full SQP step is rejected because of nonlinear constraint curvature.
10. Update primal and dual variables.
11. Update the quasi-Newton matrix only after deciding which step has actually been accepted.
12. Recompute residuals and monitor conditioning, step size, QP status, active set changes, and globalization failures.

For debugging an implementation, derivative checks are essential. Directional finite differences should verify

$$\frac{f(\boldsymbol{x} + \varepsilon \boldsymbol{v}) - f(\boldsymbol{x})}{\varepsilon} \approx \boldsymbol{q}(\boldsymbol{x})^T \boldsymbol{v}$$

and

$$\frac{\boldsymbol{h}(\boldsymbol{x} + \varepsilon \boldsymbol{v}) - \boldsymbol{h}(\boldsymbol{x})}{\varepsilon} \approx \boldsymbol{J}_h(\boldsymbol{x})\boldsymbol{v}.$$

For second derivatives, check Hessian-vector products rather than individual matrix entries whenever possible.

1.2.6 Inequality constraints

Now consider

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}), \\ \text{subject to} \quad & \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}, \\ & \boldsymbol{g}(\boldsymbol{x}) \geq \boldsymbol{0}. \end{aligned}$$

Three important numerical strategies are:

1. **active-set methods**, which estimate the constraints satisfying $g_i(\boldsymbol{x}) = 0$ and solve an equality-constrained subproblem;
2. **interior-point methods**, which keep the iterates strictly inside $g_i(\boldsymbol{x}) > 0$ and drive perturbed complementarity to zero;
3. **augmented Lagrangian methods**, which combine multiplier updates with a controlled penalty on infeasibility.

Active-set methods are especially effective when the solution has few active constraints and a good active-set estimate is available. Interior-point methods are often preferable for large sparse problems with many inequalities.

1.2.6.1 Active sets and SQP for general inequalities

For the general NLP Equation 1.18, SQP forms the quadratic subproblem

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \boldsymbol{q}_k^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T \boldsymbol{B}_k \boldsymbol{p}, \\ \text{subject to} \quad & \boldsymbol{h}_k + \boldsymbol{J}_{h,k} \boldsymbol{p} = \boldsymbol{0}, \\ & \boldsymbol{g}_k + \boldsymbol{J}_{g,k} \boldsymbol{p} \geq \boldsymbol{0}, \end{aligned} \tag{1.40}$$

with corresponding linearized bounds on $\boldsymbol{x}_k + \boldsymbol{p}$. The QP multipliers provide new estimates of the NLP multipliers. If the correct active set is identified, the QP locally reduces to an equality-constrained problem and the SQP step becomes a Newton step for the active KKT equations.

Let

$$\mathcal{A}(\boldsymbol{x}^\star) = \{ i \,:\, g_i(\boldsymbol{x}^\star) = 0 \}$$

be the active set. Under LICQ, strict complementarity, and a second-order sufficient condition, an exact-Hessian SQP method with eventually accepted unit steps has quadratic local convergence. With a suitable BFGS update of the Lagrangian Hessian, superlinear convergence can be obtained.

Active-set QP methods repeatedly add constraints that block a trial step and drop active constraints whose multiplier has the wrong sign. They are often very efficient when only a small subset of inequalities is active or when a warm start gives a good estimate of $\mathcal{A}$.

1.2.6.2 Merit functions, filters, and the Maratos effect

A globalization strategy must balance decrease of f against decrease of constraint violation. A common scalar measure is

$$\theta(\boldsymbol{x}) = \|\boldsymbol{h}(\boldsymbol{x})\|_1 + \|\min\{\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{0}\}\|_1,$$

leading to the merit function

$$\phi_\rho(\boldsymbol{x}) = f(\boldsymbol{x}) + \rho\theta(\boldsymbol{x}).$$

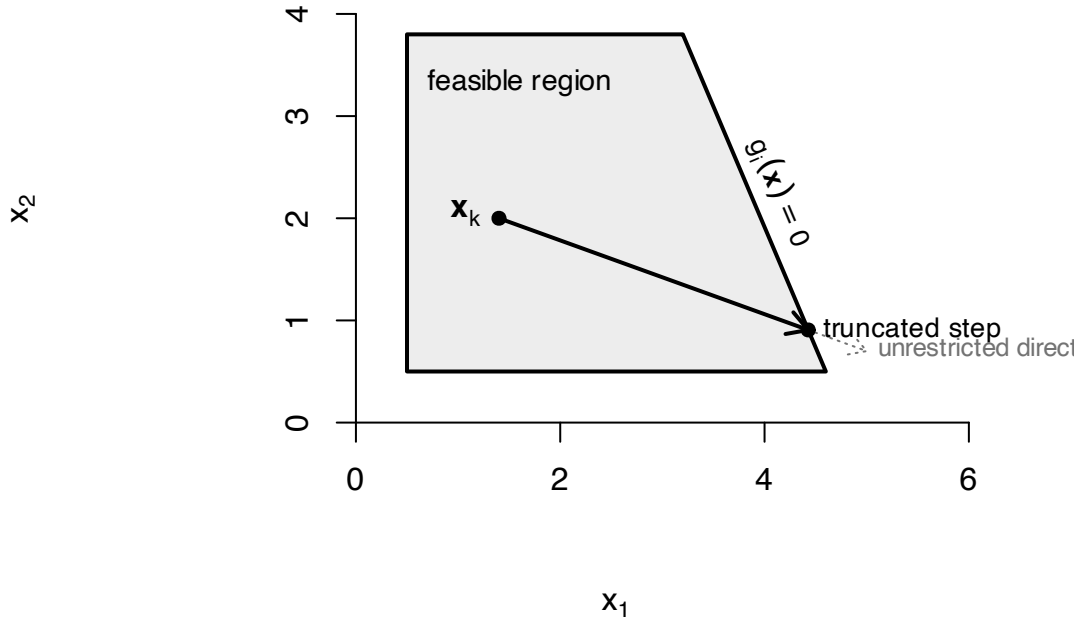


Figure 1.1: Geometric interpretation of an active-set step. The unrestricted QP direction would leave the feasible polygon; the step is truncated at the first blocking inequality, which then becomes a candidate for the active set.

A **filter method** avoids combining the two objectives with a fixed penalty parameter. It stores pairs (θ, f) and accepts a new point when it gives sufficient improvement in feasibility or objective value without being dominated by previously accepted pairs.

Even close to a solution, a first-order linearization of curved constraints can cause a merit-function line search to reject the full SQP step. This is the **Maratos effect** [@maratos1978exact], and it is worth seeing once on a concrete problem rather than taking it on trust.

▪ **Example 1.1 — The Maratos effect.** Consider

$$\min_{x\in\mathbb{R}^2} \; f(\boldsymbol{x}) = 2\big(x_1^2 + x_2^2 - 1\big) - x_1, \qquad h(\boldsymbol{x}) = x_1^2 + x_2^2 - 1 = 0,$$

whose solution is $\boldsymbol{x}^\star = (1,0)^T$: on the circle $f = -x_1$, which is minimal at $x_1 = 1$. From $\nabla f(\boldsymbol{x}^\star) = (3,0)$ and $\nabla h(\boldsymbol{x}^\star) = (2,0)$ the multiplier is $\lambda^\star = 3/2$, so

$$\boldsymbol{W}_\star = \nabla^2 f - \lambda^\star \nabla^2 h = 4\boldsymbol{I} - 3\boldsymbol{I} = \boldsymbol{I}.$$

Take a feasible iterate on the unit circle,

$$\boldsymbol{x}_k = (\cos\theta, \sin\theta)^T,$$

with θ small, so that $h(\boldsymbol{x}_k) = 0$ and $\boldsymbol{J}_k = (2\cos\theta,\, 2\sin\theta)$. The QP of Equation 1.29 minimizes $\boldsymbol{q}_k^T\boldsymbol{p} + \tfrac{1}{2}\|\boldsymbol{p}\|^2$ on the tangent line $\cos\theta\, p_1 + \sin\theta\, p_2 = 0$, and its solution is

$$\boldsymbol{p}_k = \big(\sin^2\theta,\; -\sin\theta\cos\theta\big)^T.$$

It is an excellent step. Since $\boldsymbol{x}_k + \boldsymbol{p}_k - \boldsymbol{x}^\star = (1-\cos\theta)(\cos\theta,\sin\theta)^T$ and $\|\boldsymbol{x}_k - \boldsymbol{x}^\star\|^2 = 2 - 2\cos\theta$,

$$\|\boldsymbol{x}_k + \boldsymbol{p}_k - \boldsymbol{x}^\star\| = \tfrac{1}{2}\|\boldsymbol{x}_k - \boldsymbol{x}^\star\|^2,$$

which is precisely the quadratic rate of Theorem 1.2.1. Yet the trial point is **rejected** by any ℓ_1 merit function, because

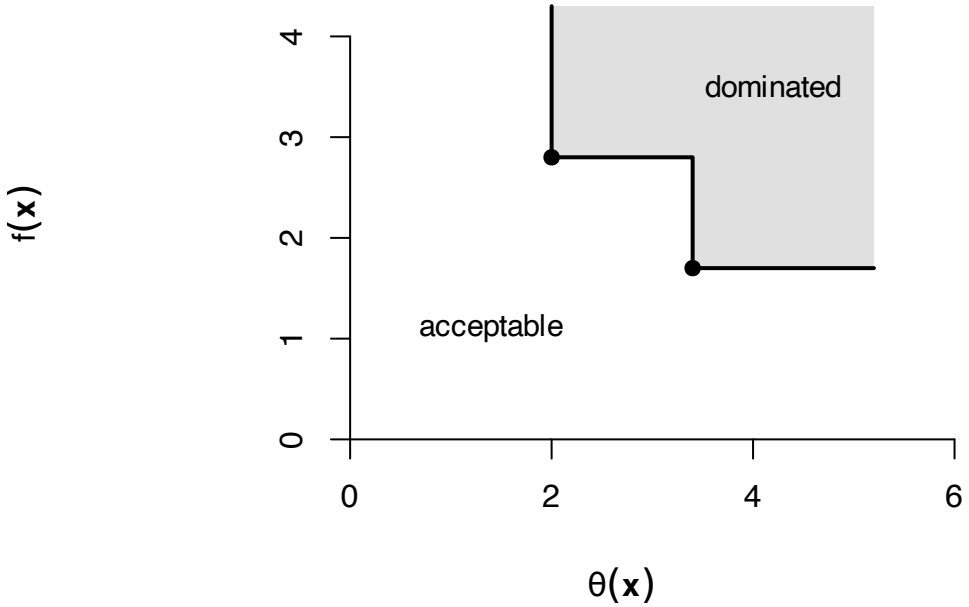


Figure 1.2: Schematic filter in the (θ, f) plane. Each accepted pair forbids the quadrant above and to the right of it; a trial point is acceptable only below the resulting staircase, that is, if it improves feasibility or objective value by a sufficient margin.

$$f(\boldsymbol{x}_k + \boldsymbol{p}_k) - f(\boldsymbol{x}_k) = \sin^2\theta > 0, \qquad |h(\boldsymbol{x}_k + \boldsymbol{p}_k)| = \sin^2\theta > |h(\boldsymbol{x}_k)| = 0.$$

Both terms of ϕ_ρ increase, for every ρ. The line search therefore cuts the step, and the quadratic rate is destroyed by the very mechanism meant to make the method robust.

▪

The cause is that the constraint is curved while the QP sees only its tangent. The standard remedy is a **second-order correction**: after the step $\boldsymbol{p}_k$ has been computed, solve a second system with the *same* matrix and the constraint residual re-evaluated at the trial point,

$$\begin{bmatrix} \boldsymbol{B}_k & -\boldsymbol{J}_k^T \\ -\boldsymbol{J}_k & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{p}} \\ \cdot \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{h}(\boldsymbol{x}_k + \boldsymbol{p}_k) \end{bmatrix}, \tag{1.41}$$

and test $\boldsymbol{x}_k + \boldsymbol{p}_k + \widehat{\boldsymbol{p}}$ before shortening the step. In Example 1.1 the correction is $\widehat{\boldsymbol{p}} = -\frac{1}{2}\sin^2\theta\,(\cos\theta, \sin\theta)^T + O(\theta^4)$: it is $O(\theta^2)$, so it does not spoil the quadratic rate, while it reduces the constraint violation from $O(\theta^2)$ to $O(\theta^4)$ and makes the objective decrease. The unit step is then accepted. Because the matrix is unchanged, the extra cost is one back-substitution. Nonmonotone strategies and filters [@fletcher2002nonlinear] are the other standard remedies, and are important for recovering fast local convergence.

1.2.6.3 Trust-region SQP

Instead of searching along a computed SQP direction, one may restrict the local model to

$$\|\boldsymbol{p}\| \le \Delta_k.$$

The radius is updated using the ratio between actual and predicted reduction [@conn2000trust].

For constrained problems the trust region creates a difficulty that does not arise in the unconstrained case: the linearized constraints $\boldsymbol{h}_k + \boldsymbol{J}_k\boldsymbol{p} = \boldsymbol{0}$ and the bound $\|\boldsymbol{p}\| \le \Delta_k$ may be

jointly infeasible, since a small radius need not contain any point of the linearized feasible set. Shrinking the radius then produces no step at all.

The Byrd–Omojokun decomposition [@omojokun1989trust] resolves this by giving up exact linearized feasibility. The step is split as $\boldsymbol{p} = \boldsymbol{p}_N + \boldsymbol{p}_T$ where the **normal step** $\boldsymbol{p}_N$ reduces infeasibility as much as the radius permits,

$$\min_{\boldsymbol{p}_N} \|\boldsymbol{h}_k + \boldsymbol{J}_k \boldsymbol{p}_N\|_2 \quad \text{subject to} \quad \|\boldsymbol{p}_N\| \le \zeta \Delta_k, \qquad \zeta \in (0,1),$$

and the **tangential step** $\boldsymbol{p}_T \in \ker(\boldsymbol{J}_k)$ then reduces the quadratic model within the remaining radius. The two subproblems are always feasible, $\boldsymbol{p}_N = \boldsymbol{0}$ and $\boldsymbol{p}_T = \boldsymbol{0}$ being admissible. This normal–tangential decomposition is the trust-region counterpart of the null-space method for the KKT equations.

1.2.6.4 Sequential convex methods

The last family of the map in Section 1.2.1 has so far only been named. The idea is the same as SQP, with the quadratic model replaced by any local model that is **convex** and cheap to solve globally. At $\boldsymbol{x}_k$ one replaces f and the constraints by convex approximations valid in a trust region, solves the resulting convex program to global optimality, and iterates.

Two classical instances are worth distinguishing.

- **Sequential linear programming** [@palacios1982nonlinear] uses a purely linear model, $\boldsymbol{q}_k^T \boldsymbol{p}$ with linearized constraints and a box trust region $\|\boldsymbol{p}\|_\infty \le \Delta_k$. The subproblem is an LP, which is attractive when a mature LP solver is available and the Hessian is unavailable. The price is that, without curvature, the method is at best linearly convergent unless the solution is a vertex of the linearized feasible set.
- **Convex–concave procedures and difference-of-convex methods** replace each nonconvex term by a convex majorizer, so that every subproblem bounds the true objective from above. Every iterate then improves the objective monotonically, without any line search.

The attraction is that the subproblem has no spurious local minima and the trust region alone provides globalization. The limitation is that the convexified model is generally a weaker local approximation than the exact Lagrangian Hessian, so superlinear convergence is lost unless the model is asymptotically exact. These methods are the natural choice when the problem is nearly convex, when a dedicated convex solver exploits structure that a general QP solver would miss, or when global solvability of each subproblem is itself a requirement.

1.2.6.5 Primal-dual interior-point Newton system

For inequalities, an interior-point method can first be understood through the logarithmic barrier. With slack variables $\boldsymbol{s} = \boldsymbol{g}(\boldsymbol{x}) > \boldsymbol{0}$, the barrier subproblem is

$$\min_{\boldsymbol{x}} \quad f(\boldsymbol{x}) - \tau \sum_{i=1}^{m_g} \log g_i(\boldsymbol{x}), \qquad \tau > 0. \tag{1.42}$$

Its stationary points trace the **central path** as $\tau \downarrow 0$. Introducing multipliers converts the barrier stationarity equations into perturbed complementarity $\mu_i s_i = \tau$, which leads directly to the primal-dual formulation below.

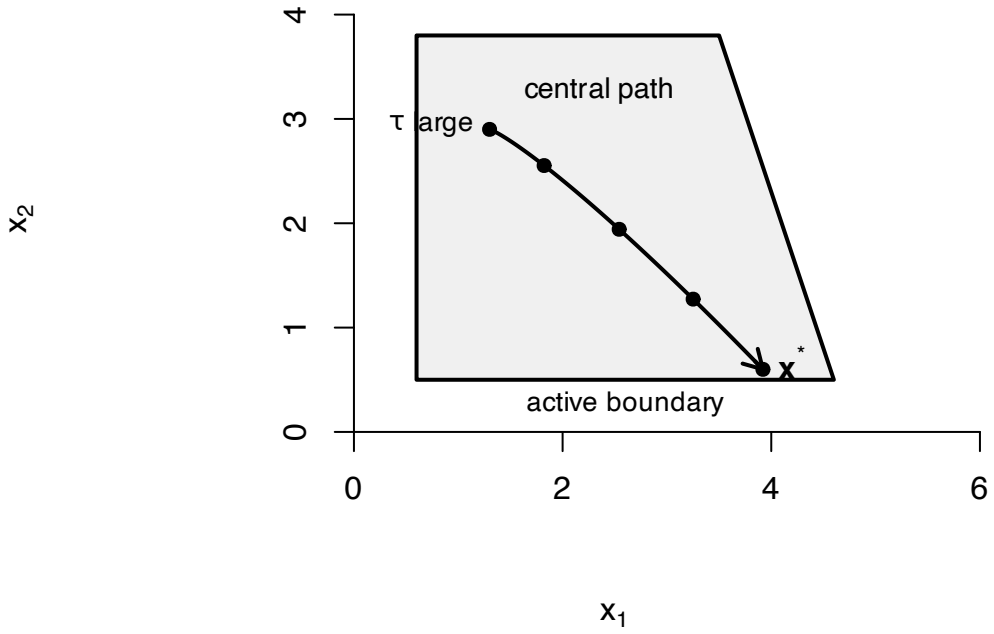


Figure 1.3: Schematic central path. Barrier minimizers stay strictly feasible for every $\tau > 0$ and approach a KKT point on the boundary as $\tau \downarrow 0$. The markers correspond to decreasing values of the barrier parameter.

> **Bound constraints**
>
> The general NLP Equation 1.18 also carries $\boldsymbol{\ell} \le \boldsymbol{x} \le \boldsymbol{u}$, and the SQP subproblem Equation 1.35 keeps those bounds explicitly. An interior-point method does not: bounds are inequalities like any other and receive their own slacks, their own multipliers and their own barrier terms. Production codes such as @wachter2006implementation treat them separately only for efficiency, because $\partial(\boldsymbol{x} - \boldsymbol{\ell})/\partial \boldsymbol{x} = \boldsymbol{I}$ makes the corresponding Jacobian blocks trivial and the resulting diagonal contribution can be added directly to $\boldsymbol{W}$. In the formulas below the bounds are therefore assumed to have been absorbed into $\boldsymbol{g}$.

Introduce positive slack variables $\boldsymbol{s}$ through

$$\boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{s} = \boldsymbol{0}, \qquad \boldsymbol{s} > \boldsymbol{0},$$

and positive inequality multipliers $\boldsymbol{\mu} > \boldsymbol{0}$. Instead of exact complementarity, impose

$$\boldsymbol{S}\boldsymbol{\mu} = \tau \boldsymbol{e},$$

where $\boldsymbol{S} = \operatorname{diag}(\boldsymbol{s})$ and $\tau > 0$ is the barrier parameter. With $\boldsymbol{M} = \operatorname{diag}(\boldsymbol{\mu})$, a primal-dual Newton step solves

$$\begin{bmatrix} \boldsymbol{W} & -\boldsymbol{J}_h^T & -\boldsymbol{J}_g^T & \boldsymbol{0} \\ \boldsymbol{J}_h & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{J}_g & \boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{I} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{S} & \boldsymbol{M} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{\lambda} \\ \Delta \boldsymbol{\mu} \\ \Delta \boldsymbol{s} \end{bmatrix} = - \begin{bmatrix} \boldsymbol{r}_d \\ \boldsymbol{r}_h \\ \boldsymbol{r}_g \\ \boldsymbol{r}_c \end{bmatrix}, \tag{1.43}$$

where

$$
\begin{aligned}
\boldsymbol{r}_d &= \nabla_x \mathcal{L}^T, \\
\boldsymbol{r}_h &= \boldsymbol{h}(\boldsymbol{x}), \\
\boldsymbol{r}_g &= \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{s}, \\
\boldsymbol{r}_c &= \boldsymbol{S}\boldsymbol{\mu} - \tau \boldsymbol{e}.
\end{aligned}
$$

Algorithm 2: Basic primal-dual interior-point iteration

Input: $\boldsymbol{x}_0$, $\boldsymbol{\lambda}_0$, $\boldsymbol{s}_0 > 0$, $\boldsymbol{\mu}_0 > 0$, tolerance ε
Output: An approximate primal-dual KKT point

$(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{s}, \boldsymbol{\mu}) \leftarrow (\boldsymbol{x}_0, \boldsymbol{\lambda}_0, \boldsymbol{s}_0, \boldsymbol{\mu}_0)$
repeat
  form $\boldsymbol{r}_d$, $\boldsymbol{r}_h$, $\boldsymbol{r}_g$ and set $\eta \leftarrow \boldsymbol{s}^T\boldsymbol{\mu}/m_g$
  if *the primal, dual and complementarity residuals are at most* ε **then**
    return $(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu})$
  end
  $\tau \leftarrow \sigma\eta$, with $\sigma \in (0, 1)$
  solve the primal-dual KKT system for $(\Delta\boldsymbol{x}, \Delta\boldsymbol{\lambda}, \Delta\boldsymbol{\mu}, \Delta\boldsymbol{s})$
  choose fraction-to-the-boundary steps α_p and α_d preserving $\boldsymbol{s} + \alpha_p\Delta\boldsymbol{s} > 0$ and $\boldsymbol{\mu} + \alpha_d\Delta\boldsymbol{\mu} > 0$
  $(\boldsymbol{x}, \boldsymbol{s}) \leftarrow (\boldsymbol{x}, \boldsymbol{s}) + \alpha_p(\Delta\boldsymbol{x}, \Delta\boldsymbol{s})$
  $(\boldsymbol{\lambda}, \boldsymbol{\mu}) \leftarrow (\boldsymbol{\lambda}, \boldsymbol{\mu}) + \alpha_d(\Delta\boldsymbol{\lambda}, \Delta\boldsymbol{\mu})$
until *the maximum number of iterations is reached*

The basic algorithm displays the central mechanism, but solving the complete four-by-four system directly is rarely the best implementation. The following reductions and predictor–corrector construction are used by practical sparse interior-point solvers.

1.2.6.6 Eliminating the slack and inequality-multiplier steps

The third block row of Equation 1.43 gives

$$
\Delta\boldsymbol{s} = \boldsymbol{J}_g\Delta\boldsymbol{x} + \boldsymbol{r}_g. \tag{1.44}
$$

Substituting this expression into the complementarity row gives

$$
\Delta\boldsymbol{\mu} = -\boldsymbol{S}^{-1}\left(\boldsymbol{r}_c + \boldsymbol{M}\boldsymbol{r}_g + \boldsymbol{M}\boldsymbol{J}_g\Delta\boldsymbol{x}\right). \tag{1.45}
$$

Eliminating $\Delta\boldsymbol{s}$ and $\Delta\boldsymbol{\mu}$ from the stationarity equation produces the augmented primal Hessian

$$
\boldsymbol{D} = \boldsymbol{W} + \boldsymbol{J}_g^T\boldsymbol{S}^{-1}\boldsymbol{M}\boldsymbol{J}_g. \tag{1.46}
$$

As in the equality-constrained case, multiply the equality-feasibility row by -1. The condensed symmetric-indefinite system, whose coefficient matrix is denoted by $\boldsymbol{K}$ below, is

$$
\begin{bmatrix} \boldsymbol{D} & -\boldsymbol{J}_h^T \\ -\boldsymbol{J}_h & \boldsymbol{0} \end{bmatrix}
\begin{bmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix}
=
\begin{bmatrix} -\boldsymbol{r}_d - \boldsymbol{J}_g^T\boldsymbol{S}^{-1}(\boldsymbol{r}_c + \boldsymbol{M}\boldsymbol{r}_g) \\ \boldsymbol{r}_h \end{bmatrix}. \tag{1.47}
$$

After solving Equation 1.47, recover $\Delta\boldsymbol{s}$ and $\Delta\boldsymbol{\mu}$ from Equation 1.44 and Equation 1.45. This reduction is particularly useful when the number of inequalities is large, because the

factorization involves only the primal variables and equality multipliers. In software, the products with $S^{-1}M$ should be formed by diagonal scaling; neither S nor any KKT matrix should be explicitly inverted.

The condensed and full formulations have different numerical tradeoffs. Condensation reduces dimension but can amplify poor scaling when some s_i/μ_i are very small. A full-space factorization preserves more of the original block structure and can be preferable when the condensed matrix becomes dense.

1.2.6.7 Fraction-to-the-boundary steps

A Newton direction need not preserve strict positivity. Define

$$\widehat{\alpha}_p = \min_{i\,:\,\Delta s_i<0} \left(-\frac{s_i}{\Delta s_i}\right),$$
$$\widehat{\alpha}_d = \min_{i\,:\,\Delta \mu_i<0} \left(-\frac{\mu_i}{\Delta \mu_i}\right),$$

where the minimum of an empty set is $+\infty$. A **fraction-to-the-boundary rule** chooses

$$\alpha_p = \min\{1, \xi\widehat{\alpha}_p\}, \qquad \alpha_d = \min\{1, \xi\widehat{\alpha}_d\}, \qquad 0 < \xi < 1. \tag{1.48}$$

Values of ξ close to one, such as 0.99 or 0.995, permit rapid progress while keeping $\boldsymbol{s} + \alpha_p \Delta \boldsymbol{s} > \boldsymbol{0}$ and $\boldsymbol{\mu} + \alpha_d \Delta \boldsymbol{\mu} > \boldsymbol{0}$. Separate primal and dual step lengths are common; a single step length may be used when a globalization strategy requires it.

1.2.6.8 Mehrotra predictor–corrector

Mehrotra's method [@mehrotra1992implementation] estimates how much complementarity a pure Newton step would remove and then chooses the centering parameter adaptively. Let

$$\eta = \frac{\boldsymbol{s}^T \boldsymbol{\mu}}{m_g}$$

be the current average complementarity.

1. The **affine-scaling predictor** sets $\tau = 0$ and solves the Newton system, producing $(\Delta \boldsymbol{x}^{\mathrm{aff}}, \Delta \boldsymbol{\lambda}^{\mathrm{aff}}, \Delta \boldsymbol{\mu}^{\mathrm{aff}}, \Delta \boldsymbol{s}^{\mathrm{aff}})$.
2. Compute maximum affine steps α_p^{aff} and α_d^{aff} that reach, but do not cross, the positivity boundary, and form
$$\eta_{\mathrm{aff}} = \frac{(\boldsymbol{s} + \alpha_p^{\mathrm{aff}} \Delta \boldsymbol{s}^{\mathrm{aff}})^T (\boldsymbol{\mu} + \alpha_d^{\mathrm{aff}} \Delta \boldsymbol{\mu}^{\mathrm{aff}})}{m_g}.$$
3. Select the centering parameter, commonly
$$\sigma = \left(\frac{\eta_{\mathrm{aff}}}{\eta}\right)^3, \qquad \tau = \sigma \eta. \tag{1.49}$$
4. The predictor misses the second-order product generated by complementarity. Therefore the corrector solve uses
$$\boldsymbol{r}_c^{\mathrm{pc}} = S\boldsymbol{\mu} + \Delta S^{\mathrm{aff}} \Delta \boldsymbol{\mu}^{\mathrm{aff}} - \sigma \eta \boldsymbol{e}, \tag{1.50}$$
where $\Delta S^{\mathrm{aff}} = \mathrm{diag}(\Delta \boldsymbol{s}^{\mathrm{aff}})$. The other residuals remain unchanged.

Both solves use the same KKT matrix, so a direct method can reuse the factorization. This is one reason the predictor–corrector method is effective: the second solve is relatively inexpensive.

Algorithm 3: Mehrotra primal-dual predictor–corrector

Input: A strictly positive primal-dual iterate, tolerance ε, fraction parameter ξ

Output: An approximate primal-dual KKT point

repeat
  form $\boldsymbol{r}_d$, $\boldsymbol{r}_h$, $\boldsymbol{r}_g$ and $\eta = \boldsymbol{s}^T\boldsymbol{\mu}/m_g$
  if *all scaled KKT residuals are at most* ε **then**
    return *the current iterate*
  end
  solve the affine Newton system with $\tau = 0$
  compute α_p^{aff}, α_d^{aff} and η_{aff}
  $\sigma \leftarrow (\eta_{\text{aff}}/\eta)^3$, $\tau \leftarrow \sigma\eta$
  add $\Delta\boldsymbol{S}^{\text{aff}}\Delta\boldsymbol{\mu}^{\text{aff}}$ to the complementarity residual
  solve the corrector system, reusing the KKT factorization
  compute α_p, α_d by the fraction-to-the-boundary rule
  update primal and dual variables and recompute the residuals
until *the iteration limit is reached*

1.2.6.9 Regularization, inertia, and sparse implementation

On the central path $\boldsymbol{S}^{-1}\boldsymbol{M}$ is positive diagonal. Nevertheless, $\boldsymbol{D}$ need not be positive definite because the Lagrangian Hessian can be indefinite. If $\boldsymbol{D}$ is positive definite on $\ker(\boldsymbol{J}_h)$ and LICQ holds, the symmetric matrix in Equation 1.47 has inertia

$$\operatorname{inertia}(\boldsymbol{K}) = (n, m_h, 0),$$

namely n positive, m_h negative, and no zero eigenvalues. A sparse LDL^T factorization reveals this inertia. An incorrect inertia signals negative curvature, rank deficiency, or insufficient regularization.

A common stabilized system replaces

$$\begin{bmatrix} \boldsymbol{D} & -\boldsymbol{J}_h^T \\ -\boldsymbol{J}_h & \boldsymbol{0} \end{bmatrix} \quad \text{by} \quad \begin{bmatrix} \boldsymbol{D} + \delta_x \boldsymbol{I} & -\boldsymbol{J}_h^T \\ -\boldsymbol{J}_h & -\delta_h \boldsymbol{I} \end{bmatrix}, \tag{1.51}$$

with small positive δ_x and δ_h, increased only when required by factorization stability or inertia. Scaling the rows and columns before factorization is essential, because complementarity can make the diagonal ratios μ_i/s_i span many orders of magnitude.

For a fixed sparsity pattern, the symbolic ordering of a sparse factorization should be computed once and reused. The affine and corrector solves have the same coefficient matrix, so the numerical factors can also be reused.

1.2.6.10 Central-path interpretation and convergence

The perturbed KKT mapping is

$$\boldsymbol{F}_\tau(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{s}) = \begin{bmatrix} \nabla_x \mathcal{L}^T \\ \boldsymbol{h}(\boldsymbol{x}) \\ \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{s} \\ \boldsymbol{S}\boldsymbol{\mu} - \tau \boldsymbol{e} \end{bmatrix}. \tag{1.52}$$

At an exact central-path point, $\eta = \boldsymbol{s}^T\boldsymbol{\mu}/m_g = \tau$. Under LICQ, a second-order sufficient condition, and strict complementarity at the limiting KKT point, the Jacobian of the appropriate primal-dual system is locally nonsingular and the central path approaches that KKT point as $\tau \downarrow 0$.

An implementation does not solve every barrier problem exactly. Instead it keeps the residuals in a neighborhood of the central path while reducing η. The affine predictor emphasizes progress toward the boundary; the centering corrector moves the iterate away from an unbalanced boundary approach. Globalization, regularization, and restoration are still required when the starting point is remote, the problem is nonconvex, or the Jacobian is nearly rank deficient; @wachter2006implementation describes a complete filter line-search interior-point method built on these ingredients.

1.2.7 Augmented Lagrangian method

Pure quadratic penalty methods solve

$$\min_{x} f(\boldsymbol{x}) + \frac{\rho}{2}\|\boldsymbol{h}(\boldsymbol{x})\|^2.$$

They require $\rho \to \infty$ to enforce feasibility accurately, which makes the Hessian increasingly ill-conditioned, as Equation 1.23 made explicit. The augmented Lagrangian [@hestenes1969multiplier; @powell1969method; @rockafellar1973dual] avoids this drawback by combining a finite penalty with multiplier estimates:

$$\mathcal{L}_\rho(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \boldsymbol{\lambda}^T\boldsymbol{h}(\boldsymbol{x}) + \frac{\rho}{2}\|\boldsymbol{h}(\boldsymbol{x})\|^2. \tag{1.53}$$

With the sign convention used here, the multiplier update is

$$\boldsymbol{\lambda}_{k+1} = \boldsymbol{\lambda}_k - \rho_k\boldsymbol{h}(\boldsymbol{x}_{k+1}). \tag{1.54}$$

The inner minimization is unconstrained and can be performed with damped Newton, BFGS, or a trust-region method. It need not be solved very accurately at early outer iterations.

Algorithm 4: Method of multipliers

Input: $\boldsymbol{x}_0$, $\boldsymbol{\lambda}_0$, penalty $\rho_0 > 0$, tolerance ε

Output: An approximate KKT pair $(\boldsymbol{x}, \boldsymbol{\lambda})$

$(\boldsymbol{x}, \boldsymbol{\lambda}, \rho) \leftarrow (\boldsymbol{x}_0, \boldsymbol{\lambda}_0, \rho_0)$
repeat
  approximately minimize $\mathcal{L}_\rho(\boldsymbol{x}, \boldsymbol{\lambda})$ with respect to $\boldsymbol{x}$, obtaining $\boldsymbol{x}^+$
  $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \rho\boldsymbol{h}(\boldsymbol{x}^+)$
  if *the KKT residual at* $(\boldsymbol{x}^+, \boldsymbol{\lambda})$ *is at most* ε **then**
    return $(\boldsymbol{x}^+, \boldsymbol{\lambda})$
  end
  if *feasibility has not decreased sufficiently* **then**
    $\rho \leftarrow \gamma\rho$, with $\gamma > 1$
  end
  tighten the inner-solve tolerance
  $\boldsymbol{x} \leftarrow \boldsymbol{x}^+$
until *the maximum number of outer iterations is reached*

For inequalities $\boldsymbol{g}(\boldsymbol{x}) \geq \boldsymbol{0}$, the Powell–Hestenes–Rockafellar term compatible with the present sign convention uses

$$\widehat{\boldsymbol{\mu}}(\boldsymbol{x}) = \max\{\boldsymbol{0}, \boldsymbol{\mu} - \rho \boldsymbol{g}(\boldsymbol{x})\},$$

where the maximum is componentwise. The augmented term can be written as

$$\frac{1}{2\rho}\left(\|\widehat{\boldsymbol{\mu}}(\boldsymbol{x})\|^2 - \|\boldsymbol{\mu}\|^2\right),$$

and the multiplier update is

$$\boldsymbol{\mu}_{k+1} = \max\{\boldsymbol{0}, \boldsymbol{\mu}_k - \rho_k \boldsymbol{g}(\boldsymbol{x}_{k+1})\}.$$

This projection preserves dual feasibility $\boldsymbol{\mu} \geq \boldsymbol{0}$.

1.2.7.1 Why a finite penalty suffices

The quadratic penalty needs $\rho \to \infty$; the augmented Lagrangian does not. The reason is that Equation 1.54 is a **fixed-point iteration** in the multipliers, and its contraction factor is $O(1/\rho)$.

Consider the equality-constrained case and suppose LICQ and the second-order sufficient condition hold at $(\boldsymbol{x}^\star, \boldsymbol{\lambda}^\star)$. Two facts follow.

First, there exists a finite threshold $\bar{\rho}$ such that for every $\rho > \bar{\rho}$ the point $\boldsymbol{x}^\star$ is a strict local minimizer of $\boldsymbol{x} \mapsto \mathcal{L}_\rho(\boldsymbol{x}, \boldsymbol{\lambda}^\star)$. Indeed

$$\nabla^2_{xx}\mathcal{L}_\rho(\boldsymbol{x}^\star, \boldsymbol{\lambda}^\star) = \boldsymbol{W}_\star + \rho\, \boldsymbol{J}_\star^T \boldsymbol{J}_\star, \tag{1.55}$$

and $\boldsymbol{W}_\star$ is positive definite on $\ker(\boldsymbol{J}_\star)$ while $\rho \boldsymbol{J}_\star^T \boldsymbol{J}_\star$ supplies positive curvature on the complementary subspace. For ρ large enough the sum is positive definite on all of $\mathbb{R}^n$. The exact multiplier therefore converts the constrained problem into an unconstrained one **without** any limit.

Second, if $\boldsymbol{x}(\boldsymbol{\lambda})$ denotes the local minimizer of $\mathcal{L}_\rho(\cdot, \boldsymbol{\lambda})$, the update $\boldsymbol{\lambda}^+ = \boldsymbol{\lambda} - \rho \boldsymbol{h}(\boldsymbol{x}(\boldsymbol{\lambda}))$ satisfies

$$\|\boldsymbol{\lambda}^+ - \boldsymbol{\lambda}^\star\| \leq \frac{C}{\rho}\, \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^\star\| \tag{1.56}$$

for a constant C independent of ρ. Convergence of the outer iteration is therefore **linear**, with a rate that can be made as small as desired by increasing ρ — but the required ρ is finite and fixed, so the conditioning of Equation 1.55 stays bounded. This is the whole point of the method, and it is why the penalty is increased in the algorithm only when feasibility fails to improve, never as a matter of course.

The price is the rate: linear, against the quadratic rate of SQP. Augmented Lagrangian methods are therefore preferred when the inner problem has exploitable structure, when feasibility is hard to attain, or when bound constraints can be handled directly in the inner solve [@conn1991globally], rather than when fast local convergence is the priority.

1.2.8 A worked example

Everything above is easier to trust after seeing it run. Consider the smallest nontrivial equality-constrained problem,

$$\min_{\boldsymbol{x} \in \mathbb{R}^2} \; f(\boldsymbol{x}) = -x_1 - x_2 \qquad \text{subject to} \qquad h(\boldsymbol{x}) = x_1^2 + x_2^2 - 1 = 0, \tag{1.57}$$

whose solution is immediate: on the unit circle $f = -x_1 - x_2$ is minimal at

$$\boldsymbol{x}^\star = \tfrac{1}{\sqrt{2}}(1,1)^T, \qquad \lambda^\star = -\tfrac{1}{\sqrt{2}} \approx -0.7071,$$

since $\boldsymbol{q} = (-1,-1)^T$ and $\boldsymbol{J}_\star = \sqrt{2}\,(1,1)$ give $\boldsymbol{q} = \boldsymbol{J}_\star^T \lambda^\star$. The Lagrangian Hessian is

$$\boldsymbol{W} = \nabla^2 f - \lambda \nabla^2 h = -2\lambda\, \boldsymbol{I},$$

so at the solution $\boldsymbol{W}_\star = \sqrt{2}\,\boldsymbol{I} \succ \boldsymbol{0}$ and the second-order sufficient condition Equation 1.30 holds.

> **!** The objective Hessian alone would fail here
>
> Note that $\nabla^2 f = \boldsymbol{0}$. Replacing $\boldsymbol{W}$ by $\nabla^2 f$, as warned against in the discussion preceding Equation 1.34, would make the KKT matrix
>
> $$\begin{bmatrix} \boldsymbol{0} & -\boldsymbol{J}^T \\ -\boldsymbol{J} & \boldsymbol{0} \end{bmatrix}$$
>
> singular: the method would not merely converge more slowly, it would not produce a step at all. The same happens if the iteration is started with $\lambda_0 = 0$. All the curvature of this problem lives in the constraint.

The following implements Equation 1.25 literally — assemble, solve, update — with no globalization, starting from an infeasible point $\boldsymbol{x}_0 = (0.6, 0.9)^T$ and $\lambda_0 = -1$.

```r
q  <- function(x) c(-1, -1)              # objective gradient (column)
Hf <- function(x) matrix(0, 2, 2)        # objective Hessian
h  <- function(x) x[1]^2 + x[2]^2 - 1    # constraint
Jh <- function(x) matrix(c(2 * x[1], 2 * x[2]), nrow = 1)
Hh <- function(x) 2 * diag(2)            # constraint Hessian

xstar <- c(1, 1) / sqrt(2)

x <- c(0.6, 0.9)
lam <- -1
rows <- list()

for (k in 0:5) {
  J  <- Jh(x)
  W  <- Hf(x) - lam * Hh(x)              # Lagrangian Hessian
  rd <- q(x) - as.vector(t(J)) * lam     # dual residual
  rows[[k + 1]] <- data.frame(
    k      = k,
    err    = sqrt(sum((x - xstar)^2)),
    kkt    = max(abs(rd), abs(h(x))),
    lambda = lam
  )
  K   <- rbind(cbind(W, -t(J)), cbind(-J, 0))   # symmetric indefinite
  d   <- solve(K, c(-rd, h(x)))                 # (p, delta lambda)
  x   <- x + d[1:2]
  lam <- lam + d[3]
}

tab <- do.call(rbind, rows)
```

```
knitr::kable(
  data.frame(
    k        = tab$k,
    error    = sprintf("%.3e", tab$err),
    residual = sprintf("%.3e", tab$kkt),
    lambda   = sprintf("%.8f", tab$lambda)
  ),
  col.names = c("$k$", "$\\lVert\\bm x_k-\\bm x^\\star\\rVert$",
                "$\\lVert\\bm F(\\bm z_k)\\rVert_\\infty$", "$\\lambda_k$"),
  align = "rrrr", booktabs = TRUE, escape = FALSE
)
```

Table 1.1: Exact-Hessian SQP on Equation 1.57. The number of correct digits doubles at each iteration, as predicted by Theorem 1.2.1.

k	$\|\boldsymbol{x}_k - \boldsymbol{x}^\star\|$	$\|\boldsymbol{F}(\boldsymbol{z}_k)\|_\infty$	λ_k
0	2.206e-01	8.000e-01	-1.00000000
1	6.169e-02	8.149e-02	-0.71367521
2	1.959e-03	3.689e-03	-0.70587593
3	2.236e-06	4.349e-06	-0.70710317
4	6.405e-12	1.568e-11	-0.70710678
5	1.110e-16	2.220e-16	-0.70710678

The error goes from $6 \cdot 10^{-2}$ to $2 \cdot 10^{-3}$ to $2 \cdot 10^{-6}$ to $6 \cdot 10^{-12}$: each iteration roughly squares the previous error, which is Equation 1.31 in action. Machine precision is reached in five iterations, and the multiplier converges at the same rate as the primal variables — it is a component of $\boldsymbol{z}$, not an afterthought.

Two remarks close the example. The iteration above has no line search, no merit function and no trust region: near the solution none is needed, and a correctly implemented globalization must become inactive exactly here, or the rate shown in Table 1.1 is lost — which is precisely what Example 1.1 illustrates. And the entire method is six lines of linear algebra; the difficulty of constrained optimization lies in making those six lines survive a bad starting point, not in the lines themselves.

1.2.9 Exercises

Exercises 1–3 are **fundamental**, 4–6 **computational**, and 7–10 **advanced**. Starting hints: use block elimination for 1–2; inspect the penalty Hessian spectrum for 3–4; retain second-order terms for 5; compare reduced and full KKT residuals for 6; preserve positivity during elimination for 7–8; and use regularization or a better multiplier estimate for 9–10.

1. **Row scaling.** Show that multiplying the second block row of Equation 1.26 by -1 changes neither the Newton step nor the solution set, and that the resulting matrix is symmetric. Why is a Cholesky factorization still inapplicable?
2. **Inertia.** Let $\boldsymbol{K}$ be the matrix of Equation 1.25 with $\boldsymbol{J}$ of full row rank m_h. Prove that if $\boldsymbol{Z}^T \boldsymbol{W} \boldsymbol{Z} \succ \boldsymbol{0}$ then $\operatorname{inertia}(\boldsymbol{K}) = (n, m_h, 0)$. Deduce a practical test for second-order sufficiency from a sparse LDL^T factorization.

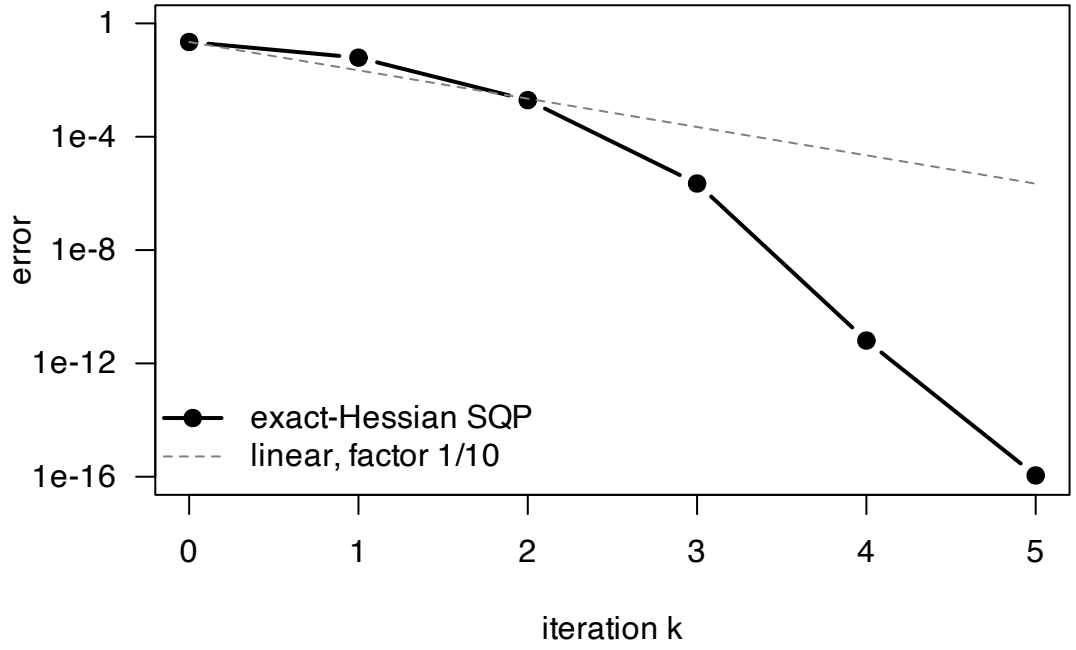


Figure 1.4: Quadratic convergence of exact-Hessian SQP on Equation 1.57. On a logarithmic scale a quadratic rate is a curve whose slope doubles, not a straight line; the dashed reference line is linear convergence with factor $1/10$.

3. **Penalty conditioning.** For Equation 1.57 write the Hessian of the quadratic penalty Equation 1.21 at the solution and compute its condition number as a function of ρ. Compare with Equation 1.55 for the augmented Lagrangian with the exact multiplier.
4. **The multiplier hidden in the penalty.** Minimize Equation 1.21 numerically for $\rho = 10, 10^2, \dots, 10^6$ and verify Equation 1.23 against $\lambda^\star = -1/\sqrt{2}$. At which ρ does the estimate stop improving, and why?
5. **Maratos.** Reproduce Example 1.1 numerically for $\theta = 10^{-1}, 10^{-2}, 10^{-3}$: compute $\boldsymbol{p}_k$, verify that both f and $|h|$ increase, then compute the second-order correction Equation 1.41 and check that the full step becomes acceptable.
6. **Null-space versus full space.** Solve the QP of Equation 1.57 by the reduced system Equation 1.37, recover the multiplier from Equation 1.38, and check that the result agrees with the full KKT solve of Table 1.1 to machine precision.
7. **Interior point by hand.** Replace the equality in Equation 1.57 by $1 - x_1^2 - x_2^2 \geq 0$. Write the system Equation 1.43, perform the elimination of Equation 1.44 and Equation 1.45 explicitly, and verify that the condensed matrix is the one in Equation 1.47. Trace the central path for $\tau = 10^{-1}, \dots, 10^{-6}$.
8. **Fraction to the boundary.** Show that Equation 1.48 with $\xi < 1$ guarantees $\boldsymbol{s} > \boldsymbol{0}$ and $\boldsymbol{\mu} > \boldsymbol{0}$ after the update, and explain why $\xi = 1$ would be unsafe even though the iterate would remain nonnegative.
9. **Quasi-Newton.** Replace $\boldsymbol{W}$ in the worked example by a BFGS matrix updated with Equation 1.39 and the Lagrangian-gradient difference. Measure the observed rate and check it against the Dennis–Moré condition.
10. **Degenerate multiplier.** Explain why starting the worked example with $\lambda_0 = 0$ makes the first KKT matrix singular, and propose two remedies consistent with the

regularization discussed above.

1.2.10 Local convergence and regularity

Fast local convergence requires more than smoothness. The standard assumptions that repeatedly appear in NLP analysis are:

- a constraint qualification, typically LICQ, so that the relevant constraint Jacobian has full row rank;
- a second-order sufficient condition, meaning that the Lagrangian Hessian is positive definite on the critical or tangent subspace;
- for the cleanest inequality-constrained local theory, strict complementarity or an appropriate weaker nondegeneracy assumption;
- sufficiently accurate derivative information and linear solves.

Under these assumptions, exact Newton applied to the equality-constrained KKT system is locally quadratically convergent. Exact-Hessian SQP inherits this rate once the correct active structure is identified and unit steps are accepted. Quasi-Newton SQP can be superlinear when the Hessian approximation satisfies an appropriate Dennis–Moré condition on the relevant subspace.

Global convergence has a different meaning: it normally asserts that, from a sufficiently broad set of starting points, accumulation points are stationary or satisfy first-order KKT conditions. It does **not** imply convergence to a global minimizer for a nonconvex NLP. Merit functions, filters, trust regions, restoration phases, and penalty/barrier updates are devices for obtaining this kind of globalization.

1.2.10.1 Infeasible starts and restoration

Requiring a feasible initial point can itself be as difficult as the original NLP. Modern SQP and interior-point solvers therefore commonly permit infeasible iterates [@gill2005snopt; @wachter2006implementation]. If the main step cannot make acceptable progress, a **restoration phase** approximately solves a feasibility problem such as

$$\min_{x} \quad \|\boldsymbol{h}(\boldsymbol{x})\|_1 + \|\min\{\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{0}\}\|_1,$$

possibly subject to simple bounds or a trust region. The objective of the restoration phase is not to optimize f, but to return to a region where the main NLP model is meaningful.

1.2.11 Linear algebra for KKT systems

The linear solve is usually the dominant cost of a constrained optimization iteration. Several formulations are useful.

- **Symmetric-indefinite factorization.** Sparse LDL^T factorization with pivoting is the standard robust choice for a moderate-size KKT matrix.
- **Null-space method.** If $\boldsymbol{Z}$ spans $\ker(\boldsymbol{J})$, compute the tangential step from the reduced Hessian $\boldsymbol{Z}^T\boldsymbol{W}\boldsymbol{Z}$. This is attractive when the number of constraints is small.
- **Range-space or Schur-complement method.** Eliminate the primal step and solve a system in the multipliers. This is effective when solves with $\boldsymbol{W}$ are cheap.
- **Iterative methods.** For very large sparse problems, MINRES or GMRES with block preconditioning avoids a complete factorization.

Scaling is essential: badly scaled variables or constraints can make an otherwise regular KKT matrix appear nearly singular. Regularization of both the primal and multiplier blocks is common in production solvers.

1.2.12 Stopping criteria and diagnostics

For equality constraints, a natural scaled KKT measure is

$$\mathcal{E}_{\text{eq}} = \max\left\{\frac{\|\nabla_x \mathcal{L}^T\|_\infty}{1+\|\boldsymbol{\lambda}\|_\infty}, \|\boldsymbol{h}(\boldsymbol{x})\|_\infty\right\}.$$

For inequalities, monitor four distinct quantities:

$$\begin{aligned} \text{stationarity:} &\quad \|\nabla_x \mathcal{L}^T\|_\infty, \\ \text{equality feasibility:} &\quad \|\boldsymbol{h}(\boldsymbol{x})\|_\infty, \\ \text{inequality violation:} &\quad \|\min(\boldsymbol{g}(\boldsymbol{x}), \boldsymbol{0})\|_\infty, \\ \text{complementarity:} &\quad \|\min\{\boldsymbol{\mu}, \boldsymbol{g}(\boldsymbol{x})\}\|_\infty. \end{aligned}$$

The last measure deserves a comment. The product $\|\boldsymbol{\mu} \odot \boldsymbol{g}\|_\infty$ is the obvious candidate, but it is misleading on infeasible iterates: if $g_i < 0$ and μ_i is small, the product is tiny although neither feasibility nor complementarity holds. The natural residual $\min\{\mu_i, g_i\}$ vanishes if and only if $\mu_i \geq 0$, $g_i \geq 0$ and $\mu_i g_i = 0$, so it measures dual feasibility, primal feasibility and complementarity at once. All residuals should be scaled, for example by $1 + \|\boldsymbol{\lambda}\|_\infty$ for stationarity and by a characteristic constraint magnitude for feasibility.

A small step is not, by itself, evidence of convergence: it may instead indicate a failed line search, a singular KKT matrix, poor scaling, or an incorrect active set. A reliable implementation reports all primal, dual, and complementarity residuals separately.

1.2.13 Choosing a method

Problem structure	Suitable first choice
Smooth unconstrained, accurate Hessian available	Damped Newton or trust-region Newton
Smooth unconstrained, expensive Hessian	BFGS or limited-memory BFGS
Equality constraints, moderate sparse system	Newton–SQP with sparse LDL^T
Few active inequalities	Active-set SQP
Many large sparse inequalities	Primal-dual interior-point method
Simple convex set or box constraints	Projected gradient/Newton or bound active set
Difficult initial feasibility or decomposable inner solves	Augmented Lagrangian method
Convex local approximations are natural	Sequential convex programming

No method removes the need for good scaling, accurate derivatives, and a globalization strategy. These three ingredients often matter more in practice than the distinction between closely related Newton-type variants.

1.2.14 A practical hierarchy for smooth NLP

For a smooth constrained problem, the following hierarchy is useful when selecting or designing a numerical method.

1. Exploit simple constraints directly: eliminate fixed variables and use projections for boxes or other cheaply projectable convex sets.
2. For general smooth constraints, regard the KKT equations as the local target and decide whether active-set/SQP or interior-point treatment of inequalities is more appropriate.
3. Use the exact Lagrangian Hessian when it is affordable and reliable; otherwise use BFGS, limited-memory approximations, or matrix-free Hessian-vector products.
4. Globalize the local step with a merit function, filter, or trust region.
5. Use augmented-Lagrangian or restoration mechanisms when feasibility is particularly difficult.
6. Measure convergence with primal feasibility, dual stationarity, and complementarity residuals rather than with the step norm alone.

This hierarchy emphasizes a central theme: modern constrained optimization is largely the numerical linear algebra of KKT systems combined with a mechanism for deciding **which KKT system to solve** and **how much of its Newton step to accept**.

II ODE

2 Numerical Methods for IVPs 45

2.1 A First Look at the Explicit Euler Method
2.2 Euler Method in Vector Form
2.3 Stability: continuous problem versus numerical method
2.4 Heun and Collatz Methods
2.5 Numerical Verification of the Order

3 Taylor Methods for ODEs 61

3.1 Taylor Methods of Low Order
3.2 A Recursive Formulation
3.3 An Improved Recursion

4 Runge-Kutta Methods 69

4.1 The General Runge–Kutta Method
4.2 Constructing Runge-Kutta Methods
4.3 Examples of Runge–Kutta Methods

5 Linear Multistep Methods 91

5.1 Foundations
5.2 Adams methods
5.3 Backward Differentiation Formulas (BDF)
5.4 Stability and Convergence
5.5 Practical Comparison

6 Absolute Stability and Stability Regions . 99

6.1 Why study stability? Start with explicit Euler
6.2 Why do complex numbers appear? Linear systems and diagonalization
6.3 The complex scalar test equation and the stability region
6.4 Runge–Kutta methods: deriving the stability function step by step
6.5 A-Stability and L-Stability
6.6 Stability for Systems and Multistep Methods
6.7 Stability Diagrams and Examples
6.8 Practical Guidance
6.9 Geometric Integration and Invariant Preservation

7 Checking the Order of a RK Method 117

7.1 The Conditions in Vector Form
7.2 Worked Verifications
7.3 Automated Verification

Numerical integration of initial-value problems: one-step and multistep methods, order conditions, and absolute stability. The theory is in the part of the same name of the course notes.

2. Numerical Methods for IVPs

In many cases, finding an analytical solution to initial value problems (IVPs) is either impossible or impractical due to the complexity of the integrals involved. Therefore, numerical methods are commonly used to approximate solutions. These methods can be applied not only to single IVPs but also extend naturally to systems of differential equations. Additionally, since higher-order differential equations can be reformulated as systems of first-order equations, we will focus on numerical methods for first-order systems.

Definition 2.0.1 — Time Discretization. On a time interval $I = [t_0, t_0 + T]$, introduce a mesh

$$t_0 < t_1 < \cdots < t_N = t_0 + T.$$

The k-th time step is the subinterval

$$I_k = [t_k, t_{k+1}], \qquad k = 0, 1, \ldots, N-1,$$

with step size

$$h_k = t_{k+1} - t_k > 0.$$

For a uniform mesh, $h_k = h = T/N$ for every k. Using N for the number of time steps avoids confusing it with the dimension of the ODE system.

Numerical methods for solving IVPs are divided into two primary categories: *one-step methods* and *multi-step methods*. In one-step methods, the next approximation z_{k+1} depends only on the current step z_k. In contrast, multi-step methods calculate z_{k+1} using information from several previous time levels. The number of previous levels (the number of steps) must not be confused with the **order of accuracy** of the method.

Definition 2.0.2 — One-Step Methods.

2.0.1 Explicit Methods

An *explicit method* (also called forward integration) calculates approximations sequentially. Starting with an initial condition z_0 at time t_0, it computes the sequence $z_1, z_2, \dots, z_n$ using the following update formula:

$$z_{k+1} = z_k + \Delta t_k F(t_k, z_k, \dots, z_{k-p}; \Delta t_k)$$

Here, F is called the *increment function*, which determines the change from one time step to the next. Here $p + 1$ indicates how many previously computed states enter the increment function; it is **not** the order of accuracy. A one-step method depends only on z_k.

Definition 2.0.3 — Implicit Methods. An *implicit method* (also called backward integration) is similar in structure to explicit methods but requires solving for the new value z_{k+1} implicitly. The update formula is given by:

$$z_{k+1} = z_k + \Delta t_k F(t_{k+1}, z_{k+1}, z_k, \dots, z_{k-p}; \Delta t_k)$$

In this case, the increment function F depends on z_{k+1}, meaning the equation must be solved iteratively or using numerical techniques to find z_{k+1}. Implicit methods often provide better stability, especially for stiff differential equations, but are computationally more intensive because of the need to solve for z_{k+1} at each step.

In summary, the distinction between explicit and implicit methods is primarily in how the next step is computed. Explicit methods are generally simpler but may require smaller time steps for stability, whereas implicit methods can handle larger time steps but at the cost of increased computational effort.

2.1 A First Look at the Explicit Euler Method

Before introducing the general convergence theory for one-step methods, it is useful to see the complete argument in the simplest possible setting. We therefore consider first the scalar initial value problem

$$\begin{cases} y'(t) = f(y(t), t), \\ y(a) = y_a, \end{cases}$$

on the interval $[a, b]$. Let

$$t_k = a + kh, \qquad h = \frac{b-a}{N}, \qquad k = 0, 1, \dots, N.$$

Derivation from Taylor's Formula

Assume for the moment that the exact solution $y(t)$ is sufficiently smooth. Taylor's formula gives

$$y(t+h) = y(t) + hy'(t) + \mathcal{O}(h^2).$$

Since the differential equation gives

$$y'(t) = f(y(t), t),$$

we obtain

$$y(t+h) = y(t) + hf(y(t), t) + \mathcal{O}(h^2).$$

If h is small, the quadratic remainder can be neglected and we obtain the approximation

$$y(t+h) \approx y(t) + hf(y(t), t).$$

Applying the same idea successively at the mesh points leads to the **explicit Euler method**

$$\boxed{\begin{aligned} y_0 &= y_a, \\ y_{k+1} &= y_k + hf(y_k, t_k), \qquad k = 0, 1, \dots, N-1. \end{aligned}}$$

Geometrically, at every point (t_k, y_k) the method follows for one time step the tangent line whose slope is $f(y_k, t_k)$. The new point is then used to compute the tangent for the following step.

A Direct Error Estimate for Euler

The preceding derivation suggests that the error made in one step is of order h^2. We now show directly how these one-step errors accumulate.

Let

$$e_k = y(t_k) - y_k$$

be the global error. Taylor's theorem with Lagrange remainder gives

$$y(t_{k+1}) = y(t_k) + hf(y(t_k), t_k) + \frac{h^2}{2} y''(\xi_k), \qquad \xi_k \in (t_k, t_{k+1}).$$

Euler's method gives

$$y_{k+1} = y_k + hf(y_k, t_k).$$

Subtracting the two relations yields

$$e_{k+1} = e_k + h\,[f(y(t_k), t_k) - f(y_k, t_k)] + \frac{h^2}{2} y''(\xi_k).$$

Assume that f is Lipschitz continuous with respect to y,

$$|f(y, t) - f(z, t)| \le L|y - z|,$$

and define

$$M_2 = \max_{t \in [a,b]} |y''(t)|.$$

Then

$$|e_{k+1}| \le (1 + hL)|e_k| + \frac{h^2}{2} M_2.$$

It is convenient to write

$$A = 1 + hL, \qquad B = \frac{h^2}{2} M_2.$$

Since the initial condition is imposed exactly, $e_0 = 0$. Hence

$$\begin{aligned} |e_1| &\le B, \\ |e_2| &\le AB + B = (1 + A)B, \\ |e_3| &\le A(1 + A)B + B = (1 + A + A^2)B, \end{aligned}$$

and, in general,

$$|e_k| \le \left(1 + A + \cdots + A^{k-1}\right) B.$$

Using the geometric-series formula,

$$|e_k| \le \frac{A^k - 1}{A - 1} B = \frac{(1 + hL)^k - 1}{hL} \, \frac{h^2}{2} M_2.$$

Since

$$1 + x \le e^x,$$

we have

$$(1 + hL)^k \le e^{khL} = e^{(t_k - a)L} \le e^{(b-a)L}.$$

Therefore, for $L > 0$,

$$\boxed{|e_k| \le \frac{e^{(b-a)L} - 1}{2L} M_2 \, h}$$

for every $k = 0, 1, \dots, N$. Consequently,

$$\boxed{\max_{0 \le k \le N} |y(t_k) - y_k| \le Ch,}$$

where C is independent of h. Thus the explicit Euler method converges with **global order one**.

This elementary calculation already contains the essential mechanism:

$$\boxed{\text{one-step defect } \mathcal{O}(h^2) \quad \Longrightarrow \quad \text{global error } \mathcal{O}(h).}$$

The next theorem abstracts exactly this argument and applies it to a general one-step method.

2.1.1 Consistency, Local Error, and Convergence

A large class of one-step methods can be written as

$$z_{k+1} = z_k + h \, \Phi(t_k, z_k; h),$$

where Φ is the increment function. To discuss accuracy, it is useful to distinguish the error made in **one exact step** from the error accumulated over many steps.

Definition 2.1.1 — Local Defect and Order. Let $z(t)$ be the exact solution. The local defect is

$$\boldsymbol{d}_{k+1} = z(t_{k+1}) - z(t_k) - h \, \Phi(t_k, z(t_k); h).$$

The method has **local order** $p + 1$, or equivalently **global order** p under the stability hypotheses below, if

$$\|\boldsymbol{d}_{k+1}\| \le Ch^{p+1}$$

uniformly on every fixed finite time interval. It is **consistent** if

$$\frac{\|\boldsymbol{d}_{k+1}\|}{h} \longrightarrow 0 \qquad (h \to 0).$$

The global error is

$$\boldsymbol{e}_k = z(t_k) - z_k.$$

The distinction is essential: a local defect $O(h^{p+1})$ is produced at each of approximately T/h steps, so one expects a global error $O(h^p)$ provided perturbations are not amplified too strongly.

Theorem 2.1.1 — Convergence Theorem for One-Step Methods. Assume that, for $0 < h \le h_0$, the increment function is uniformly Lipschitz in the state variable,

$$\|\Phi(t, \boldsymbol{x}; h) - \Phi(t, \boldsymbol{y}; h)\| \le L_\Phi \|\boldsymbol{x} - \boldsymbol{y}\|,$$

and that the local defect satisfies

$$\|\boldsymbol{d}_{k+1}\| \leq Ch^{p+1}.$$

Then, on a fixed interval $[t_0, t_0 + T]$, the global error satisfies

$$\boxed{\|\boldsymbol{e}_k\| \leq \frac{C}{L_\Phi}\left(e^{L_\Phi(t_k - t_0)} - 1\right) h^p}$$

when $L_\Phi > 0$, with the limiting bound CTh^p when $L_\Phi = 0$. In particular, the method converges with order p.

Proof

Subtracting the numerical step from the exact step gives

$$\boldsymbol{e}_{k+1} = \boldsymbol{e}_k + h\left[\Phi(t_k, z(t_k); h) - \Phi(t_k, z_k; h)\right] + \boldsymbol{d}_{k+1}.$$

Hence

$$\|\boldsymbol{e}_{k+1}\| \leq (1 + hL_\Phi)\|\boldsymbol{e}_k\| + Ch^{p+1}.$$

Since $\boldsymbol{e}_0 = 0$, iteration yields

$$\|\boldsymbol{e}_k\| \leq Ch^{p+1} \sum_{j=0}^{k-1} (1 + hL_\Phi)^j.$$

For $L_\Phi > 0$,

$$\begin{aligned}\|\boldsymbol{e}_k\| &\leq \frac{C}{L_\Phi}\left[(1 + hL_\Phi)^k - 1\right] h^p \\ &\leq \frac{C}{L_\Phi}\left(e^{L_\Phi(t_k - t_0)} - 1\right) h^p.\end{aligned}$$

This proves the result. □

This theorem is the basic finite-time mechanism behind the rule

$$\text{local error } O(h^{p+1}) \quad \Longrightarrow \quad \text{global error } O(h^p).$$

Note the structure of the bound: the local defects accumulate, and the accumulation is amplified by the factor $e^{L_\Phi(t_k - t_0)}$. This is exactly the amplification of the continuous-dependence estimate obtained from Grönwall's lemma — same structure, same reason. The constant degrades exponentially with the length of the integration interval, which is why a global error bound of this type is informative on a fixed interval and essentially useless for long-time integration.

Where L_Φ comes from

For explicit Euler $\Phi = \boldsymbol{f}$, so $L_\Phi = L$, the Lipschitz constant of $\boldsymbol{f}$. For a multistage method Φ is a composition of evaluations of $\boldsymbol{f}$ and one obtains $L_\Phi = L\big(1 + \mathcal{O}(hL)\big)$. The Lipschitz property of $\boldsymbol{f}$ is therefore the **only** substantial hypothesis — the same one that guarantees existence and uniqueness of the solution, Picard–Lindelöf theorem. Without it there is neither a unique solution to approximate nor a convergence statement to prove.

What the theorem does not say

The estimate holds as $h \to 0$ on a **fixed** interval. It says nothing about the behaviour as $k \to \infty$ at fixed h: that is the subject of absolute stability, a property entirely independent of the order p. A method of order four can produce a numerical solution that grows

without bound on a stiff problem, and the convergence theorem is not violated — it simply does not apply to that regime.

2.2 Euler Method in Vector Form

The Euler method is the simplest one-step numerical method for solving initial value problems (IVPs). One intuitive way to derive this method is through the truncated Taylor series expansion.

Consider the following IVP:

$$\begin{cases} z'(t) = f(z, t), \\ z(t_0) = z_0, \end{cases}$$

The Taylor series expansion of the solution $z(t)$ around t_0 gives us:

$$z(t_0 + \Delta t) = z(t_0) + \Delta t\, z'(t_0) + \mathcal{O}(\Delta t^2),$$

where $z'(t_0) = f(z_0, t_0)$. Neglecting higher-order terms, we approximate $z(t_0 + \Delta t)$ as:

$$z(t_0 + \Delta t) \approx z(t_0) + \Delta t z'(t_0),$$

The difference between this approximation and the exact solution $z(t_0 + \Delta t)$ is proportional to Δt^2, meaning the error decreases as Δt becomes smaller.

2.2.1 General Form of the Euler Method

To generalize this for any time interval $I = [t_0, t_0 + T]$, we divide the interval into subintervals and apply the approximation at each step. For the k-th subinterval

$$I_k = [t_{k-1}, t_k]$$

the *explicit Euler method* for the k-th step is given by:

$$z_{k+1} = z_k + \Delta t f(z_k, t_k), \quad k = 0, 1, \dots, n-1,$$

where Δt is the step size.

On the other hand, the *implicit Euler method* is

$$\boxed{z_{k+1} = z_k + \Delta t\, f(z_{k+1}, t_{k+1})}, \qquad k = 0, 1, \dots, n-1.$$

The right-hand side depends on the unknown z_{k+1}, so a nonlinear algebraic system generally has to be solved at every step.

Solving the implicit Euler step

Define the nonlinear residual

$$G(\boldsymbol{w}) = \boldsymbol{w} - z_k - hf(\boldsymbol{w}, t_{k+1}).$$

The new state is the root $G(z_{k+1}) = \mathbf{0}$. If f is differentiable with respect to the state, Newton's method gives

$$\boxed{\boldsymbol{w}^{(m+1)} = \boldsymbol{w}^{(m)} - \left[I - hJ_f(\boldsymbol{w}^{(m)}, t_{k+1})\right]^{-1} G(\boldsymbol{w}^{(m)})}.$$

In implementation one solves the linear system

$$\left[I - hJ_f(\boldsymbol{w}^{(m)}, t_{k+1})\right] \boldsymbol{s}^{(m)} = -G(\boldsymbol{w}^{(m)})$$

and sets $\boldsymbol{w}^{(m+1)} = \boldsymbol{w}^{(m)} + \boldsymbol{s}^{(m)}$; the inverse is not formed explicitly. A natural initial guess is the explicit Euler predictor

$$\boldsymbol{w}^{(0)} = \boldsymbol{z}_k + h\boldsymbol{f}(\boldsymbol{z}_k, t_k).$$

The nonlinear iteration should be stopped when both the residual and/or the Newton correction are sufficiently small relative to the requested ODE accuracy. Solving the nonlinear system much more accurately than the time-discretization error is usually unnecessary.

Convergence and Stability

Both the explicit and implicit Euler methods are first-order accurate, meaning that their local truncation error is $\mathcal{O}(\Delta t^2)$, leading to a global error that scales as $\mathcal{O}(\Delta t)$. As the step size $\Delta t \to 0$, both methods converge to the exact solution, with the error at any time

$$t \in I = [t_0, t_0 + T],$$

bounded by $C\Delta t$, where C is a constant. However, in practice, this constant can grow exponentially with T, following

$$C \propto e^{LT},$$

where L is the **Lipschitz constant** of the function $\boldsymbol{f}(\boldsymbol{z}, t)$. This growth in error amplification is particularly problematic for large-time integration.

Local and global error

For a one-step method, the **local truncation error** is the defect produced by one step when the exact value is supplied at the beginning of the step. The **global error** is

$$\boldsymbol{e}_k = \boldsymbol{z}(t_k) - \boldsymbol{z}_k.$$

For explicit Euler, if $\boldsymbol{z} \in C^2$ and $\boldsymbol{f}$ is Lipschitz in $\boldsymbol{z}$ with constant L, Taylor's theorem gives a local defect bounded by $C_0 h^2$. The error recursion satisfies

$$\|\boldsymbol{e}_{k+1}\| \le (1 + hL)\|\boldsymbol{e}_k\| + C_0 h^2.$$

Iterating this inequality and using $(1 + hL)^k \le e^{L(t_k - t_0)}$ yields

$$\boxed{\|\boldsymbol{e}_k\| \le \frac{C_0}{L}\left(e^{L(t_k - t_0)} - 1\right)h}$$

for $L > 0$ (with the obvious limit $C_0(t_k - t_0)h$ when $L = 0$). This proves first-order convergence on every fixed finite interval.

Stability Considerations

One of the key differences between the explicit and implicit Euler methods lies in their stability properties. The explicit Euler method has a **stability constraint** dictated by the step size Δt, requiring Δt to be sufficiently small to ensure numerical stability. Specifically, for a linear test equation of the form

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \lambda z, \quad \lambda \in \mathbb{C}$$

the explicit Euler method is stable only if h satisfies the **stability condition**:

$$|1 + \Delta t\lambda| \le 1.$$

For *stiff problems*, where solutions evolve on vastly different time scales and exhibit rapid transient dynamics, this condition can impose **severely restrictive time-step limitations**, making explicit Euler impractical.

In contrast, the **implicit Euler method** has stability function $R(q) = 1/(1-q)$, $q = \Delta t\,\lambda$. Its absolute-stability region contains the entire left half-plane: it is therefore **A-stable**. Moreover $R(q) \to 0$ as $|q| \to \infty$ in the left half-plane, so implicit Euler is also **L-stable**. "Unconditionally stable" should only be understood in this precise test-equation sense, not as a statement that every numerical problem is stable for arbitrary step sizes. This property makes it particularly advantageous for stiff problems, where it allows for significantly larger time steps without numerical instability. However, this comes at the cost of solving a nonlinear equation at each step, increasing computational expense.

Thus, while both methods share the same convergence rate of $\mathcal{O}(\Delta t)$, their stability properties determine their applicability:

- **Explicit Euler**: Suitable for *non-stiff problems* but requires small Δt for stability.
- **Implicit Euler**: More robust for *stiff problems*, allowing larger Δt, but requires solving implicit equations at each step.

What is stiffness?

There is no single definition of stiffness that covers every nonlinear problem, but the computational phenomenon is clear: a problem is called **stiff** when stability, rather than accuracy, forces an explicit method to use a step size much smaller than the time scale on which the solution of interest changes.

For the linear system

$$z' = Az,$$

suppose the eigenvalues of A have negative real parts. Each eigenmode behaves approximately like $e^{\lambda_i t}$. If some modes decay extremely rapidly while others decay slowly, an explicit method must resolve the fastest stable mode even after that transient has become negligible.

When all relevant eigenvalues are real and negative, the heuristic **stiffness ratio**

$$\kappa_{\text{stiff}} = \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|}$$

measures the separation of decay time scales. A large value is an indication of stiffness, not a universal definition. For general non-normal or nonlinear systems, eigenvalues alone may not capture the whole phenomenon.

This explains why A-stable and especially L-stable implicit methods are useful: they can strongly damp rapidly decaying modes without forcing h to resolve their shortest time scale.

2.3 Stability: continuous problem versus numerical method

The word **stability** is used for several distinct concepts. Keeping them separate prevents a common source of confusion.

Stability of the continuous IVP

For two exact solutions of

$$z' = f(z, t),$$

with nearby initial data, continuous dependence gives, under a Lipschitz assumption,

$$\|z(t) - \widetilde{z}(t)\| \leq e^{L|t-t_0|}\|z_0 - \widetilde{z}_0\|.$$

This is a property of the differential equation, not of a discretization.

Zero-stability and propagation of perturbations

A numerical method is also affected by perturbations: roundoff, inexact nonlinear solves, perturbed starting values, and local truncation errors. For one-step methods, a Lipschitz bound on the increment function provides the required finite-time stability. For linear multistep methods the corresponding concept is **zero-stability**, treated in the dedicated chapter.

Absolute stability

Absolute stability addresses a different question: whether a numerical method reproduces the decay of the scalar test problem

$$z' = \lambda z, \qquad \operatorname{Re}\lambda < 0.$$

If a method gives

$$z_{n+1} = R(q)z_n, \qquad q = h\lambda,$$

its absolute-stability region is

$$\mathcal{S} = \{q \in \mathbb{C} : |R(q)| \leq 1\}.$$

This is a property of the numerical method applied to the test equation; it is not the same as convergence or stability of the exact IVP.

A-stability and L-stability

A method is **A-stable** if

$$\{q \in \mathbb{C} : \operatorname{Re} q \leq 0\} \subseteq \mathcal{S}.$$

Thus every exponentially decaying scalar mode is numerically stable for every step size. A method is **L-stable** if it is A-stable and additionally

$$R(q) \to 0 \qquad \text{as } |q| \to \infty, \quad \operatorname{Re} q < 0.$$

L-stability is especially useful for stiff problems because very fast stable modes are strongly damped rather than merely kept bounded.

These notions form a hierarchy of different questions:

1. **well-posedness / continuous dependence** concerns the exact ODE;
2. **consistency** measures the one-step defect of the discretization;
3. **zero-stability or one-step stability** controls perturbation growth;
4. **convergence** asks whether the numerical solution approaches the exact one as $h \to 0$;
5. **absolute, A-, and L-stability** describe behavior for decaying test modes at finite step size.

2.4 Heun and Collatz Methods

2.4.1 Heun's Method

Heun's method, also known as the improved Euler method, is a second-order numerical method for solving initial value problems (IVPs). It refines the accuracy of the explicit Euler

method by incorporating an additional correction step.

Consider the IVP:

$$\begin{cases} z'(t) = f(z, t), \\ z(t_0) = z_0, \end{cases}$$

Similar to the Euler method, Heun's method begins by estimating the next step using the explicit Euler approximation:

$$\tilde{z}_{k+1} = z_k + \Delta t f(z_k, t_k).$$

This intermediate value $\tilde{z}_{k+1}$ is then used to refine the estimate by averaging the slopes at t_k and t_{k+1}:

$$z_{k+1} = z_k + \frac{\Delta t}{2} \left[f(z_k, t_k) + f(\tilde{z}_{k+1}, t_{k+1}) \right].$$

This two-stage approach makes Heun's method **more accurate than explicit Euler**, achieving a global error of $\mathcal{O}(\Delta t^2)$ instead of $\mathcal{O}(\Delta t)$.

Convergence and Stability

Heun's method is second-order accurate, meaning that its local truncation error is $\mathcal{O}(\Delta t^3)$, leading to a global error of $\mathcal{O}(\Delta t^2)$. Compared to the explicit Euler method, it provides improved accuracy for the same step size.

In terms of absolute stability, Heun's method is **conditionally stable**. For the scalar test equation its stability polynomial is

$$R(q) = 1 + q + \frac{q^2}{2}.$$

Along the negative real axis the stability interval is again $[-2, 0]$, exactly as for explicit Euler; therefore second-order accuracy does **not** automatically imply a less restrictive step-size bound on purely dissipative scalar problems. The two-dimensional stability regions are different, but neither statement should be reduced to the vague claim that one is simply "more stable." Like every explicit Runge–Kutta method, Heun's method remains unsuitable for strongly stiff problems when stability forces very small steps.

2.4.2 Collatz Method

The Collatz method, is another improvement over the Euler method. It introduces multiple stages to achieve better accuracy by approximating the solution using additional intermediate steps.

Formulation of the Collatz Method

For a given step size Δt, the Collatz method approximates the solution as follows:

1. Compute an initial (half) Euler step:

$$\tilde{z} = z_k + \frac{\Delta t}{2} f\left(t_k, z_k\right).$$

2. Compute a midpoint correction:

$$z_{k+1} = z_k + \Delta t f\left(t_k + \frac{\Delta t}{2}, \tilde{z}\right).$$

This results in overall second-order accuracy. The improvement over explicit Euler is primarily one of **accuracy**; its absolute-stability properties must be analyzed separately.

Convergence and Stability

Like Heun's method, the Collatz method achieves a global error of $\mathcal{O}(\Delta t^2)$, making it more accurate than the first-order Euler method. For the scalar test equation, its stability polynomial is $R(q) = 1 + q + q^2/2$, the same as Heun's method. Thus the two methods have the **same absolute-stability region**; neither has a larger stability region than the other.

However, similar to Heun's method, the Collatz method is still **not unconditionally stable** and may require small step sizes for stiff problems.

2.4.3 Comparison of Methods

Method	Order of Accuracy	Stability	Advantages	Disadvantages
Explicit Euler	$\mathcal{O}(\Delta t)$	Conditionally stable	Simple and fast	Requires small Δt for stability
Heun's Method	$\mathcal{O}(\Delta t^2)$	Improved but not unconditionally stable	More accurate than Euler	Still requires small h for stiff problems
Collatz (midpoint)	$\mathcal{O}(\Delta t^2)$	Same test-equation stability polynomial as Heun	Two-stage, second order	Still unsuitable for strongly stiff problems

Both Heun's and Collatz methods improve upon the Euler method by incorporating corrections that enhance accuracy and stability, making them more suitable for practical numerical integration.

2.5 Numerical Verification of the Order

Whenever a method is implemented — or an existing implementation is used on a new problem — the observed order should be checked against the theoretical one. The test is cheap, it catches most coding mistakes, and it is the only practical way to confirm that the convergence theory applies to the problem at hand.

The Estimator

A method has order p when the maximum error over the mesh,

$$E_h = \max_{0 \le k \le N} \left| y_k - y(t_k) \right|,$$

behaves like

$$E_h \approx C h^p, \tag{2.1}$$

with C depending on the problem but not on h. Computing the solution with two step sizes h_1 and h_2 and taking the ratio,

$$\frac{E_{h_1}}{E_{h_2}} \approx \frac{C h_1^p}{C h_2^p} = \left(\frac{h_1}{h_2} \right)^p,$$

eliminates the unknown constant. Taking logarithms gives the estimator

$$\boxed{p \approx \frac{\log\left(E_{h_1}/E_{h_2}\right)}{\log\left(h_1/h_2\right)}} \tag{2.2}$$

which for a halved step, $h_2 = h_1/2$, reduces to $p \approx \log_2\left(E_{h_1}/E_{h_2}\right)$.

Equivalently, taking logarithms in Equation 2.1,

$$\log E_h \approx \log C + p \log h,$$

so that plotting $\log E_h$ against $\log h$ gives a straight line of slope p and intercept $\log C$. Good linearity over several refinements is itself part of the verification: it confirms that the method is in its asymptotic regime.

A Worked Example

Consider the linear problem

$$y' = -2y + t^2, \qquad y(0) = \frac{1}{4},$$

whose exact solution is

$$y(t) = \frac{t(t-1)}{2} + \frac{1}{4} + \left(y_0 - \frac{1}{4}\right) e^{-2t},$$

so that for $y_0 = \frac{1}{4}$ the exponential term disappears and the solution is the polynomial $t(t-1)/2 + 1/4$. The right-hand side still depends on y, so no method below reproduces it exactly — the test is a genuine one. We integrate on $[0, 1]$ with the explicit Euler, Heun and classical RK4 methods, halving the step size seven times.

```r
f     <- function(y, t) -2 * y + t^2
exact <- function(t, y0) t * (t - 1) / 2 + 1/4 + exp(-2 * t) * (y0 - 1/4)

# generic explicit one-step driver: y_{k+1} = y_k + h * Psi(t_k, y_k, h)
one_step <- function(Psi, t0, y0, h, t_end) {
  t <- seq(t0, t_end, by = h)
  y <- numeric(length(t)); y[1] <- y0
  for (i in seq_len(length(t) - 1)) y[i + 1] <- y[i] + h * Psi(t[i], y[i], h)
  data.frame(t = t, y = y)
}

psi_euler <- function(t, y, h) f(y, t)
psi_heun  <- function(t, y, h) {
  k1 <- f(y, t); k2 <- f(y + h * k1, t + h); (k1 + k2) / 2
}
psi_rk4   <- function(t, y, h) {
  k1 <- f(y, t);            k2 <- f(y + h/2 * k1, t + h/2)
  k3 <- f(y + h/2 * k2, t + h/2); k4 <- f(y + h * k3, t + h)
  (k1 + 2*k2 + 2*k3 + k4) / 6
}

t0 <- 0; y0 <- 0.25; t_end <- 1
hs <- 0.1 / 2^(0:6)

max_error <- function(Psi) sapply(hs, function(h) {
  s <- one_step(Psi, t0, y0, h, t_end)
```

```
  max(abs(s$y - exact(s$t, y0)))
})

E <- data.frame(h     = hs,
                Euler = max_error(psi_euler),
                Heun  = max_error(psi_heun),
                RK4   = max_error(psi_rk4))

tabE <- E
tabE[-1] <- lapply(tabE[-1], formatC, format = "e", digits = 2)
tabE$h   <- formatC(E$h, format = "e", digits = 2)
knitr::kable(tabE, align = "r",
             col.names = c("$h$", "Euler", "Heun", "RK4"),
             caption = "Maximum error $E_h$ on $[0,1]$.")
```

Table 2.1: Maximum error E_h on $[0, 1]$.

h	Euler	Heun	RK4
1.00e-01	2.23e-02	2.40e-03	3.98e-06
5.00e-02	1.10e-02	5.69e-04	2.37e-07
2.50e-02	5.45e-03	1.39e-04	1.44e-08
1.25e-02	2.71e-03	3.42e-05	8.91e-10
6.25e-03	1.35e-03	8.50e-06	5.53e-11
3.13e-03	6.76e-04	2.12e-06	3.45e-12
1.56e-03	3.38e-04	5.29e-07	2.15e-13

The observed orders between consecutive refinements follow from Equation 2.2:

```
obs <- function(e) c(NA, log(head(e, -1) / tail(e, -1)) /
                         log(head(hs, -1) / tail(hs, -1)))
O <- data.frame(h = formatC(hs, format = "e", digits = 2),
                Euler = obs(E$Euler), Heun = obs(E$Heun), RK4 = obs(E$RK4))
knitr::kable(O, align = "r", digits = 3,
             col.names = c("$h$", "Euler", "Heun", "RK4"),
             caption = "Observed order between consecutive step sizes.")
```

Table 2.2: Observed order between consecutive step sizes.

h	Euler	Heun	RK4
1.00e-01	NA	NA	NA
5.00e-02	1.023	2.075	4.070
2.50e-02	1.011	2.037	4.036
1.25e-02	1.006	2.018	4.018
6.25e-03	1.003	2.009	4.009
3.13e-03	1.001	2.005	4.004
1.56e-03	1.001	2.002	4.003

The three columns converge to 1, 2 and 4, matching the theoretical orders. A single global estimate can also be obtained by least squares on the log-log data, which is what the slopes of the following plot represent.

```
long <- reshape2::melt(E, id.vars = "h",
                       variable.name = "method", value.name = "error")

ggplot2::ggplot(long, ggplot2::aes(h, error, colour = method)) +
  ggplot2::geom_line() + ggplot2::geom_point() +
  ggplot2::scale_x_log10() + ggplot2::scale_y_log10() +
  ggplot2::labs(x = "h", y = expression(E[h]), colour = "method") +
  ggplot2::theme_minimal()
```

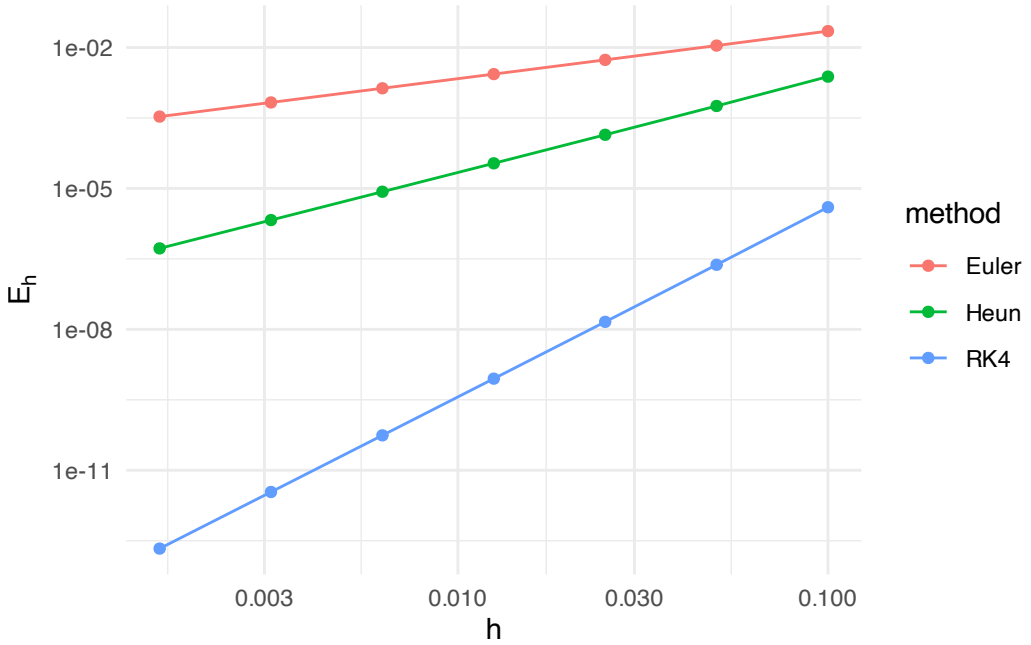


Figure 2.1: Maximum error versus step size in logarithmic scale. The slopes are the observed orders.

⚠ How to read an order verification

- **Look for a plateau, not a single number.** Three or four consecutive estimates settling on the same value are evidence; one value is not.
- **Stay in the useful window.** The estimate is meaningful only when h is small enough for the asymptotic regime Equation 2.1 to hold, and large enough that E_h is well above the rounding level $\sim \varepsilon_{\text{mach}}$. For a fourth-order method on a smooth problem this window can be only two or three refinements wide, which is why the verification should be run with a *loose* tolerance rather than at the limit of machine precision.
- **A wrong order is informative.** An observed order of 1 for a method designed to be of order 2 usually means a misplaced h in the increment function; an observed order one *higher* than expected usually means the test problem is too special — a polynomial solution, for instance, can be integrated exactly by a method of

sufficient order.
- **Without an exact solution**, replace $y(t_k)$ by a reference computed on a much finer mesh, or use the three-mesh estimator $p \approx \log_2\left|(y^{4h} - y^{2h})/(y^{2h} - y^{h})\right|$, which needs no exact solution at all.

Summary

- A one-step method is written $y_{k+1} = y_k + h\Psi(t_k, y_k; h)$; the whole theory relates the **local defect** $\mathcal{O}(h^{p+1})$ to the **global error** $\mathcal{O}(h^p)$.
- The convergence theorem Theorem 2.1.1 gives the explicit bound $\frac{C}{L_\Psi}\left(e^{L_\Psi(t_k - t_0)} - 1\right)h^p$: the Lipschitz property of f is the only substantial hypothesis.
- Explicit Euler has order 1; Heun and Collatz reach order 2 with two evaluations of f per step.
- Implicit Euler and Crank–Nicolson require solving an equation at every step but have far better stability properties.
- The order is **verified** numerically from the ratio of errors at halved step sizes, Equation 2.2 — a check that should accompany every implementation.

3. Taylor Methods for ODEs

The explicit Euler method was obtained in the previous chapter by truncating the Taylor expansion of the exact solution after the linear term. Nothing forces us to stop there: if the solution is smooth enough, keeping more terms produces a one-step method of arbitrarily high order. The resulting family, the **Taylor methods**, is the natural reference against which Runge–Kutta methods are judged, and it is the only elementary way to reach high order using the information available at a *single* point.

Throughout this chapter we consider the scalar initial value problem

$$\begin{cases} y'(t) = f(y(t), t), \\ y(a) = y_a, \end{cases} \tag{3.1}$$

on $[a, b]$, discretised on the uniform mesh

$$t_k = a + kh, \qquad h = \frac{b-a}{N}, \qquad k = 0, 1, \dots, N,$$

and we assume that f has as many continuous partial derivatives as the expansions below require. The extension to systems is purely notational: every derivative becomes a vector and every partial derivative a Jacobian or a higher-order tensor.

3.1 Taylor Methods of Low Order

The three lowest-order Taylor methods are built the same way: truncate the Taylor expansion of the solution and replace the derivatives by their expressions in terms of f. Order one gives Euler, and each further term costs one more total derivative.

3.1.1 From Taylor's Formula to Euler

Taylor's formula with Lagrange remainder gives

$$y(t_{k+1}) = y(t_k) + y'(t_k)h + \frac{y''(\zeta_k)}{2}h^2, \qquad \zeta_k \in (t_k, t_{k+1}).$$

The differential equation supplies the first derivative, $y'(t_k) = f(y(t_k), t_k)$, so

$$y(t_{k+1}) = y(t_k) + h f(y(t_k), t_k) + \frac{h^2}{2} y''(\zeta_k).$$

Neglecting the $\mathcal{O}(h^2)$ remainder yields the **explicit Euler method**

$$y_{k+1} = y_k + h f(y_k, t_k),$$

whose local truncation error is $\mathcal{O}(h^2)$ and whose global order is therefore one. The question addressed here is what happens when the remainder is pushed one or more orders further.

3.1.2 The Second-Order Taylor Method

Retaining one more term,

$$y(t_{k+1}) = y(t_k) + y'(t_k)h + \frac{y''(t_k)}{2}h^2 + \frac{y'''(\zeta_k)}{6}h^3, \qquad \zeta_k \in (t_k, t_{k+1}),$$

raises an immediate difficulty: $y''(t_k)$ is not directly available. It is, however, completely determined by the differential equation. Differentiating $y'(t) = f(y(t), t)$ with respect to t and using the chain rule,

$$y''(t) = \frac{\mathrm{d}}{\mathrm{d}t} f(y(t), t) = \frac{\partial f(y(t), t)}{\partial t} + \frac{\partial f(y(t), t)}{\partial y}\, y'(t),$$

and replacing $y'(t)$ by $f(y(t), t)$ gives the **computable** expression

$$y''(t) = \frac{\partial f(y(t), t)}{\partial t} + \frac{\partial f(y(t), t)}{\partial y}\, f(y(t), t).$$

Substituting into the expansion and discarding the remainder produces the **second-order Taylor method**

$$\boxed{y_{k+1} = y_k + h f(y_k, t_k) + \frac{h^2}{2}\left(\frac{\partial f(y_k, t_k)}{\partial t} + \frac{\partial f(y_k, t_k)}{\partial y} f(y_k, t_k)\right).} \tag{3.2}$$

> **i** Symbolic differentiation
>
> Building a Taylor method of order p requires the partial derivatives of f up to order $p-1$. Computing them by hand is laborious and error prone already for $p = 3$. In practice one relies on a computer algebra system (Mathematica, Maple, SymPy) or on **automatic differentiation**, which evaluates the derivatives to machine accuracy without producing their symbolic expressions. The recursive formulation of Section 3.2 is precisely what such tools implement.

3.1.3 The Third-Order Taylor Method

One further term requires y'''. Differentiating the expression just obtained for y'',

$$\begin{aligned} y'''(t) &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) \\ &= \frac{\partial^2 f}{\partial t^2} + 2\frac{\partial^2 f}{\partial t\,\partial y} f + \frac{\partial^2 f}{\partial y^2} f^2 + \left(\frac{\partial f}{\partial y}\right)^2 f + \frac{\partial f}{\partial t}\frac{\partial f}{\partial y}, \end{aligned}$$

all partial derivatives being evaluated at $(y(t), t)$, with subscripts indicating differentiation with respect to the named variable.

Compact Notation

Writing

$$f_k = f(y(t_k), t_k), \qquad f_{t,k} = \frac{\partial f(y(t_k), t_k)}{\partial t}, \qquad f_{y,k} = \frac{\partial f(y(t_k), t_k)}{\partial y},$$

$$f_{tt,k} = \frac{\partial^2 f(y(t_k), t_k)}{\partial t^2}, \quad f_{ty,k} = \frac{\partial^2 f(y(t_k), t_k)}{\partial t \partial y}, \quad f_{yy,k} = \frac{\partial^2 f(y(t_k), t_k)}{\partial y^2},$$

the expansion of the exact solution becomes

$$y(t_{k+1}) = y(t_k) + h f_k + \frac{h^2}{2}\left(f_{t,k} + f_{y,k} f_k\right) + \frac{h^3}{6}\left(f_{tt,k} + 2 f_{ty,k} f_k + f_{yy,k} f_k^2 + f_{y,k}^2 f_k + f_{t,k} f_{y,k}\right) + \mathcal{O}(h^4).$$

Dropping the remainder gives the **third-order Taylor method**

$$\boxed{\begin{aligned} y_{k+1} = y_k + h f_k + \frac{h^2}{2}\left(f_{t,k} + f_{y,k} f_k\right) \\ + \frac{h^3}{6}\left(f_{tt,k} + 2 f_{ty,k} f_k + f_{yy,k} f_k^2 + f_{y,k}^2 f_k + f_{t,k} f_{y,k}\right), \end{aligned}} \tag{3.3}$$

with all quantities now evaluated at the *numerical* point (y_k, t_k).

> **!** The cost of order
>
> Comparing Equation 3.2 and Equation 3.3 shows the structural weakness of the family: the number of elementary differentials grows combinatorially with the order (1, 2, 5, 15, … terms for orders 1, 2, 3, 4, …), exactly as the number of rooted trees in the Runge–Kutta order conditions. Each of them involves a different partial derivative of f. Runge–Kutta methods were invented to buy the same order **without any derivative of** f, paying instead with a few extra evaluations of f per step. This is the comparison that motivates the next chapter.

3.2 A Recursive Formulation

The derivations above are instances of a single recursion that can be carried out to any order. Let y solve $y' = f(y, t)$ and consider the Taylor expansion

$$y(t+h) = y(t) + h y'(t) + \frac{h^2}{2} y''(t) + \cdots + \frac{h^p}{p!} y^{(p)}(t) + \mathcal{O}(h^{p+1}). \tag{3.4}$$

We look for functions $D_k(y, t)$ — independent of the particular solution — such that

$$y^{(k)}(t) = D_k\big(y(t), t\big), \qquad k = 1, 2, \ldots$$

Definition of the Functions D_k

The recursion starts from

$$D_1(y, t) = f(y, t),$$

since $y'(t) = f(y(t), t)$. Assuming $y^{(k)}(t) = D_k(y(t), t)$ and differentiating,

$$\begin{aligned} y^{(k+1)}(t) &= \frac{\mathrm{d}}{\mathrm{d}t} D_k\big(y(t), t\big) \\ &= \frac{\partial D_k(y(t),t)}{\partial t} + \frac{\partial D_k(y(t),t)}{\partial y}\, y'(t) \\ &= \frac{\partial D_k(y(t),t)}{\partial t} + \frac{\partial D_k(y(t),t)}{\partial y}\, f(y(t),t), \end{aligned}$$

which yields the recurrence

$$\boxed{\begin{cases} D_1(y,t) = f(y,t), \\ D_{k+1}(y,t) = \dfrac{\partial D_k(y,t)}{\partial t} + \dfrac{\partial D_k(y,t)}{\partial y} f(y,t), \qquad k = 1, 2, \dots \end{cases}} \tag{3.5}$$

Substituting into Equation 3.4 gives the **Taylor method of order** p

$$\boxed{\begin{cases} y_{k+1} = y_k + h\, y_k' + \dfrac{h^2}{2!} y_k'' + \cdots + \dfrac{h^p}{p!} y_k^{(p)}, \\ y_k^{(j)} = D_j(y_k, t_k), \qquad j = 1, \dots, p. \end{cases}} \tag{3.6}$$

Its local truncation error is $\mathcal{O}(h^{p+1})$ and, under the usual Lipschitz hypothesis on the increment function, the global order is p.

Example: $f(y,t) = t + y + ty$

For the problem

$$\begin{cases} y' = t + y + ty, \\ y(a) = y_a, \end{cases}$$

the recurrence Equation 3.5 gives

$$D_1 = t + y + ty,$$

$$\begin{aligned} D_2 &= \frac{\partial D_1}{\partial t} + \frac{\partial D_1}{\partial y} D_1 = (y+1) + (t+1)(t + y + ty) \\ &= (y+1)t^2 + (2y+1)t + 2y + 1, \end{aligned}$$

$$D_3 = (y+1)t^3 + (3y+2)t^2 + (6y+4)t + 4y + 1,$$

$$D_4 = (y+1)t^4 + (4y+3)t^3 + (12y+9)t^2 + (16y+8)t + 10y + 4,$$

$$D_5 = (y+1)t^5 + (5y+4)t^4 + (20y+16)t^3 + (40y+25)t^2 + (50y+28)t + 26y + 8,$$

$$\begin{aligned} D_6 = {} & (y+1)t^6 + (6y+5)t^5 + (30y+25)t^4 + (80y+56)t^3 \\ & + (150y+98)t^2 + (156y+76)t + 76y + 28. \end{aligned}$$

Using these six functions in Equation 3.6 produces a sixth-order method for this particular equation. Note how quickly the expressions grow even for a right-hand side as simple as a bilinear polynomial: this growth is the price of eliminating y' at every stage of the recursion.

3.3 An Improved Recursion

The explosion observed above is largely an artefact of insisting that D_k depend on (y, t) only. Each substitution $y' \mapsto f(y,t)$ is immediately followed by a differentiation, which reintroduces the same subexpressions in a more complicated form.

If instead the derivatives already computed are kept as independent arguments, the recursion stays simple. Consider

$$y'(t) = f(y(t),t) = D_1(y(t),t),$$

$$y''(t) = \frac{\mathrm{d}}{\mathrm{d}t} D_1(y(t),t) = \frac{\partial D_1}{\partial t} + \frac{\partial D_1}{\partial y} y'(t) = D_2\big(y(t), y'(t), t\big),$$

$$\begin{aligned} y'''(t) &= \frac{\mathrm{d}}{\mathrm{d}t} D_2\big(y(t), y'(t), t\big) \\ &= \frac{\partial D_2}{\partial t} + \frac{\partial D_2}{\partial y} y'(t) + \frac{\partial D_2}{\partial y'} y''(t) = D_3\big(y(t), y'(t), y''(t), t\big), \end{aligned}$$

and so on. This suggests functions

$$D_k\big(y, y', \dots, y^{(k-1)}, t\big)$$

such that, along the exact solution,

$$y^{(k)}(t) = D_k\big(y(t), y'(t), \dots, y^{(k-1)}(t), t\big),$$

defined by the recurrence

$$\boxed{\begin{cases} D_1(y,t) = f(y,t), \\ \qquad D_{k+1} = \dfrac{\partial D_k}{\partial t} + \displaystyle\sum_{j=0}^{k-1} \frac{\partial D_k}{\partial y^{(j)}}\, y^{(j+1)}, \qquad k = 1, 2, \dots \end{cases}} \tag{3.7}$$

with the convention $y^{(0)} = y$. The derivatives $y'_k, y''_k, \dots$ are evaluated in sequence at each mesh point, so the right-hand side of the recurrence is always available.

Example: $f(y,t) = t + y + ty$ Revisited

For the same equation, and omitting the arguments,

$$D_1 = t + y + ty,$$

$$D_2 = \frac{\partial D_1}{\partial t} + \frac{\partial D_1}{\partial y} y' = (t+1)y' + y + 1,$$

$$D_3 = \frac{\partial D_2}{\partial t} + \frac{\partial D_2}{\partial y} y' + \frac{\partial D_2}{\partial y'} y'' = (t+1)y'' + 2y',$$

$$D_4 = (t+1)y''' + 3y'',$$

$$D_5 = (t+1)y^{(4)} + 4y''',$$

$$D_6 = (t+1)y^{(5)} + 5y^{(4)}.$$

The pattern $D_{k+1} = (t+1)y^{(k)} + k\, y^{(k-1)}$ for $k \geq 2$ is evident, and the sixth-order method becomes

$$
\begin{cases}
y_{k+1} = y_k + h y_k' + \dfrac{h^2}{2!} y_k'' + \dfrac{h^3}{3!} y_k''' + \dfrac{h^4}{4!} y_k^{(4)} + \dfrac{h^5}{5!} y_k^{(5)} + \dfrac{h^6}{6!} y_k^{(6)}, \\
\quad y_k' = (t_k + 1) y_k + t_k, \\
\quad y_k'' = (t_k + 1) y_k' + y_k + 1, \\
\quad y_k''' = (t_k + 1) y_k'' + 2 y_k', \\
\quad y_k^{(4)} = (t_k + 1) y_k''' + 3 y_k'', \\
\quad y_k^{(5)} = (t_k + 1) y_k^{(4)} + 4 y_k''', \\
\quad y_k^{(6)} = (t_k + 1) y_k^{(5)} + 5 y_k^{(4)}.
\end{cases}
$$

Each derivative now costs a couple of arithmetic operations instead of the evaluation of a polynomial of growing degree: the same sixth-order accuracy is obtained at a small fraction of the cost of Equation 3.6.

Numerical Check of the Order

The following experiment integrates $y' = t + y + ty$ with $y(0) = 1$ on $[0, 1]$ using the improved recursion truncated at orders $p = 2, 4, 6$, and compares the error at $t = 1$ against the reference solution produced by a stringent adaptive solver.

```
taylor_step <- function(t, y, h, p) {
  d <- numeric(p)              # d[j] = y^(j)
  d[1] <- (t + 1) * y + t
  if (p >= 2) d[2] <- (t + 1) * d[1] + y + 1
  if (p >= 3) for (k in 3:p) d[k] <- (t + 1) * d[k-1] + (k-1) * d[k-2]
  y + sum(h^(1:p) / factorial(1:p) * d)
}

taylor_solve <- function(N, p, t0 = 0, y0 = 1, T = 1) {
  h <- (T - t0) / N; t <- t0; y <- y0
  for (k in 1:N) { y <- taylor_step(t, y, h, p); t <- t + h }
  y
}

# reference solution: y' = (1+t) y + t, solved to near machine accuracy
ref <- deSolve::ode(y = 1, times = c(0, 1),
                    func = function(t, y, parms) list((1 + t) * y + t),
                    parms = NULL, rtol = 1e-14, atol = 1e-14)[2, 2]

Ns  <- 2^(2:7)
err <- expand.grid(N = Ns, p = c(2, 4, 6))
err$error <- mapply(function(N, p) abs(taylor_solve(N, p) - ref), err$N, err$p)
err$h <- 1 / err$N

tab <- reshape2::dcast(err, N ~ p, value.var = "error")
tab[-1] <- lapply(tab[-1], formatC, format = "e", digits = 2)
knitr::kable(
  tab,
  col.names = c("$N$", "$p=2$", "$p=4$", "$p=6$"),
```

```
  align = "r",
  caption = "Absolute error at $t=1$ as a function of the number of steps."
)
```

Table 3.1: Absolute error at $t = 1$ as a function of the number of steps.

N	$p = 2$	$p = 4$	$p = 6$
4	3.44e-01	8.51e-03	1.34e-04
8	1.05e-01	6.77e-04	2.77e-06
16	2.90e-02	4.78e-05	4.98e-08
32	7.62e-03	3.17e-06	8.35e-10
64	1.95e-03	2.04e-07	1.44e-11
128	4.94e-04	1.30e-08	1.16e-12

```
# observed orders from consecutive refinements
ord <- do.call(rbind, lapply(c(2, 4, 6), function(p) {
  e <- err$error[err$p == p]
  data.frame(p = p, observed = c(NA, log2(head(e, -1) / tail(e, -1))))
}))
ord$N <- rep(Ns, times = 3)
```

```
ggplot2::ggplot(err, ggplot2::aes(h, error, colour = factor(p))) +
  ggplot2::geom_line() + ggplot2::geom_point() +
  ggplot2::scale_x_log10() + ggplot2::scale_y_log10() +
  ggplot2::labs(x = "h", y = "error at t = 1", colour = "order p") +
  ggplot2::theme_minimal()
```

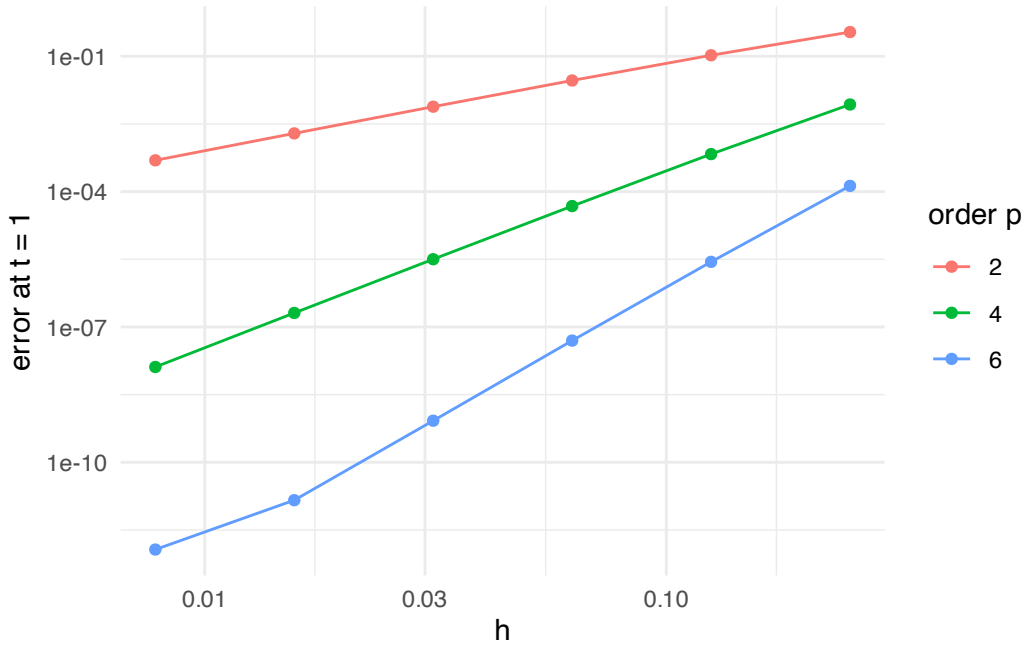


Figure 3.1: Error at $t = 1$ for the Taylor methods of order 2, 4 and 6.

```
knitr::kable(
  reshape2::dcast(ord, N ~ p, value.var = "observed"),
  col.names = c("$N$", "$p=2$", "$p=4$", "$p=6$"),
  digits = 3, align = "r",
  caption = "Observed order $\\log_2(e_{N}/e_{2N})$ between consecutive refinements."
)
```

Table 3.2: Observed order $\log_2(e_N/e_{2N})$ between consecutive refinements.

N	$p = 2$	$p = 4$	$p = 6$
4	NA	NA	NA
8	1.709	3.652	5.595
16	1.856	3.826	5.799
32	1.929	3.913	5.898
64	1.965	3.957	5.853
128	1.983	3.978	3.633

The observed slopes approach the nominal orders 2, 4 and 6. The last entry of the sixth-order column degrades not because the method fails, but because the error has reached the accuracy of the reference solution itself: with $p = 6$ and $N = 128$ the approximation is already as good as the yardstick used to measure it. This is the usual warning that an order verification is meaningful only in the window between the pre-asymptotic transient and the accuracy floor of the comparison.

Summary

- Repeated differentiation of the ODE expresses y'', y''', … in terms of f and its partial derivatives; truncating the Taylor series at order p gives a one-step method of order p.
- The method reaches any order using information at a **single point**, but requires the derivatives of f: it is practical only with symbolic or automatic differentiation.
- The recursion Equation 3.5 is the straightforward formulation; the improved recursion Equation 3.7 keeps the previously computed derivatives as arguments and avoids the combinatorial growth of the expressions.
- The comparison is what motivates Runge–Kutta methods: the same order, **no derivatives of** f, at the price of several evaluations of f per step.

4. Runge-Kutta Methods

The Euler method uses a single evaluation of f per step and is limited to first order; the Taylor methods of the previous chapter reach any order, but at the price of the derivatives of f. Runge–Kutta methods occupy the middle ground: they obtain high order using **only evaluations of f**, placed at several intermediate points inside the step (see @butcher2008numerical, @hairer1993solving).

The presentation here goes from the general to the particular. The general method is defined first, with no restriction on its coefficients; the familiar explicit schemes then appear as the special case in which the coefficient matrix is strictly lower triangular, a structure that reduces the computation of a step to a sequence of function evaluations. Taking the opposite route — explicit first, implicit as an afterthought — hides the fact that the order conditions, the Butcher tableau and the whole theory are the same in both cases, and that only the *cost* of one step changes.

Before introducing Runge–Kutta schemes, recall the standard initial-value problem for an explicit ODE,

$$\begin{cases} \boldsymbol{x}'(t) = \boldsymbol{f}(\boldsymbol{x}(t), t), \\ \boldsymbol{x}(0) = \boldsymbol{x}_0. \end{cases}$$

More generally, an ODE may be given in **fully implicit form**,

$$\begin{cases} \boldsymbol{F}(\boldsymbol{x}(t), \boldsymbol{x}'(t), t) = \boldsymbol{0}, \\ \boldsymbol{x}(0) = \boldsymbol{x}_0, \end{cases}$$

where the equation need not be solved explicitly for $\boldsymbol{x}'$. The explicit form is the special case $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{v}, t) = \boldsymbol{v} - \boldsymbol{f}(\boldsymbol{x}, t)$. To describe an ODE rather than a DAE, $\partial \boldsymbol{F}/\partial \boldsymbol{v}$ must be locally nonsingular along the solution, so that the implicit-function theorem locally determines $\boldsymbol{v} = \boldsymbol{x}'$. Runge–Kutta methods for this general form will be written below using the same stage increments, with the stage derivatives entering $\boldsymbol{F}$ directly.

4.1 The General Runge–Kutta Method

An s-stage Runge–Kutta method advances the solution of $\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}, t)$ from t_k to $t_{k+1} = t_k + \Delta t$ through s **stages** $\boldsymbol{K}_1, \dots, \boldsymbol{K}_s$:

$$
\begin{aligned}
\boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + \sum_{i=1}^{s} b_i \boldsymbol{K}_i \\
&\begin{cases}
\boldsymbol{K}_1 = \Delta t\, \boldsymbol{f}\left(\boldsymbol{z}_k + \displaystyle\sum_{j=1}^{s} a_{1j} \boldsymbol{K}_j,\ t_k + \Delta t\, c_1 \right) \\
\boldsymbol{K}_2 = \Delta t\, \boldsymbol{f}\left(\boldsymbol{z}_k + \displaystyle\sum_{j=1}^{s} a_{2j} \boldsymbol{K}_j,\ t_k + \Delta t\, c_2 \right) \\
\quad \vdots \\
\boldsymbol{K}_s = \Delta t\, \boldsymbol{f}\left(\boldsymbol{z}_k + \displaystyle\sum_{j=1}^{s} a_{sj} \boldsymbol{K}_j,\ t_k + \Delta t\, c_s \right)
\end{cases}
\end{aligned}
\tag{4.1}
$$

The method is completely described by the $s^2 + 2s$ coefficients a_{ij}, b_i, c_i, collected in the **Butcher tableau**

$$
\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \dots & a_{1s} \\
c_2 & a_{21} & a_{22} & \dots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\
\hline
 & b_1 & b_2 & \dots & b_s
\end{array}
\qquad \text{or} \qquad
\begin{array}{c|c}
\boldsymbol{c} & \boldsymbol{A} \\
\hline
 & \boldsymbol{b}^T
\end{array}
$$

The quantities $\boldsymbol{K}_i/\Delta t$ are approximations of $\boldsymbol{f}$ at the internal times $t_k + c_i \Delta t$, and the final update is a weighted average of them. The abscissae are not free: they must satisfy the **stage-consistency** relation

$$
c_i = \sum_{j=1}^{s} a_{ij}, \qquad i = 1, \dots, s, \qquad \text{that is} \qquad \boldsymbol{A}\mathbf{1} = \boldsymbol{c},
$$

so that the i-th stage really is an approximation at time $t_k + c_i \Delta t$; when $0 \le c_i \le 1$ the stage times lie inside the current step. This relation is assumed throughout and is revisited in 2.

> **!** In general, a step is a nonlinear system
>
> In Equation 4.1 the stages appear on **both** sides: each $\boldsymbol{K}_i$ depends on all the others. For a problem of dimension d, one step therefore requires the solution of a nonlinear system of $s \cdot d$ equations, not the evaluation of a formula. Whether that system is expensive, cheap, or absent altogether depends entirely on the shape of $\boldsymbol{A}$ — which is the subject of the next section.

Runge–Kutta stages for a fully implicit ODE

For the fully implicit problem, the same tableau and stage increments give

$$\begin{aligned} z_{k+1} &= z_k + \sum_{i=1}^{s} b_i K_i, \\ F\left(z_k + \sum_{j=1}^{s} a_{ij} K_j, \frac{K_i}{\Delta t}, t_k + c_i \Delta t \right) &= 0, \qquad i = 1, \dots, s. \end{aligned} \tag{4.2}$$

Here $K_i/\Delta t$ approximates the derivative at stage i. Unlike the explicit-ODE formula, even a **strictly lower-triangular** tableau does not in general give an explicit algorithm: once the earlier stages are known, the stage equation still has to be solved for K_i because the derivative is an argument of F. A lower-triangular (DIRK) tableau makes these nonlinear stage solves sequential; a full tableau couples all stages and requires a coupled nonlinear solve. Thus fully implicit ODEs generally call for a diagonally implicit or fully implicit RK procedure, rather than an explicit RK procedure in the usual computational sense.

4.2 Constructing Runge-Kutta Methods

Everything needed to build a Runge–Kutta method and to use it in practice: the shape of the coefficient matrix, how a step is actually solved, the conditions that fix the order, and the embedded pairs that drive step-size selection.

4.2.1 The Structure of A

Three shapes of the coefficient matrix are used in practice, and they correspond to three very different costs per step.

Fully implicit methods

A is a full matrix: all stages are coupled and the $sd \times sd$ system must be solved as a whole. This is the most expensive option, and it buys the most: an s-stage method can reach order $2s$ (the Gauss–Legendre family), and these methods have the best stability properties available — several of them are A-stable or L-stable for every s. They are the methods of choice when the problem is very stiff, when high accuracy is required at large step sizes, or when a structural property such as symplecticity must be preserved.

Diagonally implicit methods (DIRK)

A is **lower triangular**, $a_{ij} = 0$ for $j > i$:

$$\begin{array}{c|cccc} c_1 & a_{11} & 0 & \dots & 0 \\ c_2 & a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

The i-th stage then involves only $K_1, \dots, K_i$, so the stages are computed **one at a time**:

$$K_i = \Delta t f\Big(z_k + \underbrace{\sum_{j<i} a_{ij} K_j}_{\text{already known}} + a_{ii} K_i, t_k + \Delta t c_i \Big), \qquad i = 1, \dots, s.$$

Each stage is still implicit — $\boldsymbol{K}_i$ appears on both sides through the diagonal entry a_{ii} — but the system to be solved has dimension d, not sd. Since the cost of a direct solve grows like the cube of the dimension, replacing one $sd \times sd$ factorization by s factorizations of size $d \times d$ saves a factor of about s^2.

Two refinements matter in practice.

- **SDIRK** (*singly diagonally implicit*): all the diagonal entries are equal, $a_{11} = \cdots = a_{ss} = \gamma$. Then every stage requires the same Newton matrix $\boldsymbol{I} - \Delta t \gamma \boldsymbol{J}_f$, which can be factorized **once per step** — or even reused across several steps — instead of once per stage.
- **ESDIRK** (*explicit first stage*): $a_{11} = 0$, so the first stage is explicit and free, while the remaining ones are singly diagonally implicit. This also makes the method FSAL (Section 4.2.5), saving one further evaluation.

DIRK and SDIRK methods are the standard compromise in production stiff solvers: they keep A-stability and L-stability, which an explicit method can never have, while paying only a few $d \times d$ solves per step. Their price is the order barrier — a DIRK method of order p generally needs more stages than a fully implicit one — and a slightly larger error constant.

Explicit methods

$\boldsymbol{A}$ is **strictly** lower triangular, $a_{ij} = 0$ for $j \geq i$ — the diagonal vanishes as well:

$$\begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ c_2 & a_{21} & 0 & 0 & 0 & 0 \\ c_3 & a_{31} & a_{32} & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 & 0 \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}$$

Now the right-hand side of the i-th stage involves only stages already computed, and Equation 4.1 collapses from a nonlinear system to an explicit recipe:

$$\begin{aligned} \boldsymbol{z}_{k+1} &= \boldsymbol{z}_k + \sum_{i=1}^{s} b_i \boldsymbol{K}_i, \\ &\begin{cases} \boldsymbol{K}_1 = \Delta t \boldsymbol{f}\Big(\boldsymbol{z}_k, t_k\Big), \\ \boldsymbol{K}_2 = \Delta t \boldsymbol{f}\Big(\boldsymbol{z}_k + a_{21}\boldsymbol{K}_1, t_k + \Delta t c_2\Big), \\ \boldsymbol{K}_3 = \Delta t \boldsymbol{f}\Big(\boldsymbol{z}_k + a_{31}\boldsymbol{K}_1 + a_{32}\boldsymbol{K}_2, t_k + \Delta t c_3\Big), \\ \quad \vdots \\ \boldsymbol{K}_s = \Delta t \boldsymbol{f}\Big(\boldsymbol{z}_k + \sum_{j=1}^{s-1} a_{sj}\boldsymbol{K}_j, t_k + \Delta t c_s\Big). \end{cases} \end{aligned} \tag{4.3}$$

One step costs exactly s evaluations of $\boldsymbol{f}$, with no linear algebra, no Jacobian and no iteration. This is why explicit methods dominate non-stiff computation. The simplification is not free: as shown in Section 4.2.4.1, the strict triangularity makes $\boldsymbol{A}$ nilpotent and caps the attainable order at s, and the stability function becomes a polynomial, so the absolute-stability region is necessarily bounded.

Summary of the three structures

Table 4.1: Cost and capability of the three coefficient-matrix structures.

structure	A	cost of one step	max order	A-stable?
explicit	strictly lower triangular	s evaluations of f	s	never
DIRK / SDIRK	lower triangular	s systems $d \times d$	below $2s$	possible
fully implicit	full	one system $sd \times sd$	$2s$	possible

Everything that follows — order conditions, embedded pairs, stability functions — applies verbatim to all three. Only the work needed to evaluate one step changes.

4.2.2 Solving the Stage Equations

For a fully implicit or diagonally implicit method, Equation 4.1 must actually be solved at every step. The two candidates are the methods of the chapter on nonlinear equations: fixed-point iteration and Newton's method.

Fixed-point iteration, and why it is not enough

The simplest option is to read Equation 4.1 directly as a fixed-point iteration,

$$K_i^{(m+1)} = \Delta t f\Big(z_k + \sum_{j=1}^{s} a_{ij} K_j^{(m)},\ t_k + c_i \Delta t\Big), \qquad m = 0, 1, 2, \dots$$

started for instance from $K_i^{(0)} = \Delta t f(z_k, t_k)$. If f is Lipschitz in y with constant L, the map is a contraction as soon as

$$\boxed{\Delta t\, L \max_i \sum_{j=1}^{s} |a_{ij}| < 1,} \tag{4.4}$$

and the iteration then converges linearly with a rate equal to the left-hand side of Equation 4.4.

> ⚠ Why fixed-point iteration defeats the purpose
>
> Condition Equation 4.4 imposes $\Delta t \lesssim 1/L$. But implicit methods exist precisely for **stiff** problems, where L is huge: the contraction condition would reintroduce exactly the step-size restriction that implicitness was meant to remove. On a stiff problem a fixed-point solver cancels the benefit of the implicit formula, and Newton's method becomes mandatory.

Solving the coupled stage equations with Newton's method

For an implicit Runge–Kutta method the stages are coupled. It is convenient to work with stage states

$$Y_i = z_n + \Delta t \sum_{j=1}^{s} a_{ij} f(Y_j, t_n + c_j \Delta t), \qquad i = 1, \dots, s.$$

If each state has dimension d, stack all stages into the vector

$$Y = \begin{pmatrix} Y_1 \\ \vdots \\ Y_s \end{pmatrix} \in \mathbb{R}^{sd}.$$

Define the nonlinear residual blocks

$$G_i(Y) = Y_i - z_n - \Delta t \sum_{j=1}^{s} a_{ij} f(Y_j, t_n + c_j \Delta t).$$

The stage problem is the nonlinear system

$$\boxed{G(Y) = 0},$$

of dimension sd. Newton's method computes $S^{(m)}$ from

$$J_G(Y^{(m)}) S^{(m)} = -G(Y^{(m)}),$$

and sets

$$Y^{(m+1)} = Y^{(m)} + S^{(m)}.$$

The (i, j) block of the Newton matrix is

$$\boxed{[J_G]_{ij} = \delta_{ij} I - \Delta t a_{ij} J_f(Y_j, t_n + c_j \Delta t)}$$

with $d \times d$ blocks. Hence a fully implicit s-stage RK method generally requires the solution of a coupled linear system of dimension sd at every Newton iteration.

For large systems, forming and factorizing this full matrix can dominate the cost. Practical solvers may therefore use simplified Newton iterations, reuse Jacobians and factorizations, exploit Kronecker-product structure, or use iterative linear solvers with preconditioning. The structural alternative is to give up the full A altogether and use a DIRK method (Section 4.2.1), whose Newton matrix is block lower triangular.

Unlike the fixed-point iteration, Newton's method converges **quadratically** and requires no condition of the form Equation 4.4: the step size Δt stays free, which is the whole reason for choosing an implicit method.

Practical devices

- **Simplified Newton.** Freeze J_f at (t_k, z_k) and reuse the same factorization across all iterations, and often across several consecutive steps. Quadratic convergence is lost, but so is the dominant cost, the $\mathcal{O}\big((sd)^3\big)$ factorization.
- **DIRK and SDIRK methods.** With A lower triangular the Newton matrix is block lower triangular and the stages are solved **one at a time**: s systems of size $d \times d$ instead of one of size $sd \times sd$ (Section 4.2.1). If in addition all the diagonal entries are equal, the same factorization serves every stage.
- **Predictor.** A good starting guess $K^{(0)}$, extrapolated from the previous step, usually reduces the iteration count to two or three.

4.2.3 Order Conditions: the Two-Stage Case

The conditions listed in the next section are usually presented as a table to be looked up. They are, however, nothing more than the result of matching two Taylor expansions, and carrying out the match once — for the simplest nontrivial case — explains both their form and their number.

Two-Variable Taylor Expansion with Remainder

Everything rests on expanding f around (y, t) in **both** arguments. It is worth stating the expansion with a quantitative remainder, because the order conditions and the error constants both come from there.

Proposition 4.2.1 — Taylor expansion with remainder. Let $f \in C^1$ in a neighbourhood of (y, t). Then

$$f(y+\beta, t+\alpha) = f(y,t) + \beta f_y(y,t) + \alpha f_t(y,t) + r_1(\alpha,\beta),$$

and if the gradient (f_y, f_t) is Lipschitz with constant M_1 in that neighbourhood,

$$\boxed{|r_1(\alpha,\beta)| \le \frac{M_1}{2}\big(\alpha^2+\beta^2\big).} \tag{4.5}$$

If $f \in C^2$ the second-order expansion holds,

$$f(y+\beta, t+\alpha) = f + \beta f_y + \alpha f_t + \frac{1}{2}\big(\beta^2 f_{yy} + 2\alpha\beta f_{ty} + \alpha^2 f_{tt}\big) + r_2(\alpha,\beta),$$

with $|r_2| = \mathcal{O}\big((\alpha^2+\beta^2)^{3/2}\big)$, and $|r_2| \le \frac{M_2}{6}(\alpha^2+\beta^2)^{3/2}$ if the Hessian is Lipschitz with constant M_2.

In a Runge–Kutta method the expansion is always applied with $\alpha = c_i\Delta t$ and $\beta = \mathcal{O}(\Delta t)$: every increment is proportional to Δt, so the remainder Equation 4.5 is $\mathcal{O}(\Delta t^2)$ and, at second order, $\mathcal{O}(\Delta t^3)$. This is what allows the expansions to be truncated at the desired order and the terms to be collected in powers of Δt.

> **i** How much regularity is really needed
>
> A method of order p requires $f \in C^p$: the order conditions involve the partial derivatives of f up to order $p-1$, and the exact solution must be C^{p+1}. If f is merely Lipschitz the method still **converges** — the convergence theorem Theorem 2.1.1 uses only the Lipschitz property — but with order 1. High order is paid for in regularity, not only in stages.

The Local Truncation Error of a Two-Stage Method

Consider the scalar problem $y' = f(y,t)$ and the two-stage increment

$$\Phi(t, y, \Delta t) = y + A\Delta t f(y,t) + B\Delta t f\big(y + D\Delta t f(y,t),\ t + C\Delta t\big),$$

with four parameters A, B, C, D to be determined. The local truncation error is the difference between the exact solution and one step of the method started on it,

$$\tau(\Delta t) = y(t+\Delta t) - \Phi\big(t, y(t), \Delta t\big).$$

For the exact solution the differential equation supplies the first two derivatives,

$$y(t+\Delta t) = y(t) + \underbrace{f}_{y'}\Delta t + \underbrace{\big(f_t + f_y f\big)}_{y''}\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3),$$

all quantities being evaluated at $(y(t), t)$. For the method, Proposition 4.2.1 with $\alpha = C\Delta t$ and $\beta = D\Delta t f$ gives

$$f\big(y + D\Delta t f, t + C\Delta t\big) = f + C\Delta t f_t + D\Delta t f_y f + \mathcal{O}(\Delta t^2).$$

Subtracting and collecting powers of Δt,

$$\boxed{\begin{aligned}\tau(\Delta t) &= \Delta t\, f\,(1 - A - B)\\ &\quad + \frac{\Delta t^2}{2}\, f_t\,(1 - 2BC) + \frac{\Delta t^2}{2}\, f_y f\,(1 - 2BD) + \mathcal{O}(\Delta t^3).\end{aligned}} \tag{4.6}$$

Reading the Conditions Off the Expansion

The three quantities f, f_t and $f_y f$ are **independent**: no relation holds between them for a general right-hand side, so each of their coefficients must vanish separately. This is the mechanism that generates the order conditions, and the reason their number grows with the order — at order p one coefficient appears for each *elementary differential* of order p, that is, for each rooted tree with p nodes.

Requiring $\tau(\Delta t) = \mathcal{O}(\Delta t^3)$, hence a globally second-order method, gives

$$\begin{cases} 1 - A - B = 0, \\ 1 - 2BC = 0, \\ 1 - 2BD = 0, \end{cases} \qquad \Longrightarrow \qquad A = 1 - B, \quad C = D = \frac{1}{2B}.$$

Three conditions and four unknowns leave a **one-parameter family** of second-order two-stage methods,

$$y_{k+1} = y_k + (1 - B)\Delta t f(y_k, t_k) + B\Delta t f\left(y_k + \frac{\Delta t}{2B} f(y_k, t_k),\ t_k + \frac{\Delta t}{2B}\right),$$

which in the standard notation $b_1 = 1 - B$, $b_2 = B$, $c_1 = 0$, $c_2 = a_{21} = \frac{1}{2B}$ reads

$$\begin{aligned} y_{k+1} &= y_k + b_1 \boldsymbol{K}_1 + b_2 \boldsymbol{K}_2, \\ \boldsymbol{K}_1 &= \Delta t f(y_k, t_k), \\ \boldsymbol{K}_2 &= \Delta t f\big(y_k + a_{21}\boldsymbol{K}_1,\ t_k + c_2 \Delta t\big). \end{aligned}$$

The familiar members of the family correspond to particular choices of the free parameter:

B	c_2	method
$1/2$	1	Heun (explicit trapezoidal)
1	$1/2$	explicit midpoint
$3/4$	$2/3$	Ralston (minimal error constant)

The remaining freedom is not accuracy but the size of the leading error term: Ralston's choice minimises the coefficient of Δt^3 in τ, which is why it appears as the second-order member of embedded pairs.

Beyond Two Stages

Repeating the construction with s stages produces s^2+2s coefficients a_{ij}, b_i, c_i and, at each order, one algebraic equation per elementary differential. The bookkeeping becomes unmanageable by hand beyond order four, and the systematic tool is Butcher's rooted-tree formalism. The resulting conditions in vector form are collected next; Section 7.1 applies them to concrete tableaux.

4.2.4 Order Conditions: the General Case

The derivation of 1 produced three conditions for a two-stage method of order two. The same procedure, carried out with s stages and pushed to order p, produces a finite list of algebraic identities among the coefficients a_{ij}, b_i, c_i. This section collects that list.

Two facts organise the whole section.

- The conditions are **cumulative**. A method has order p when every condition listed for orders $1, \dots, p$ holds; it has order *exactly* p when in addition at least one condition of order $p+1$ fails. Nothing can be skipped: all the conditions of a level must hold before the level is attained.
- Each condition corresponds to one **elementary differential** of the right-hand side — one way of combining f and its derivatives — and therefore to one rooted tree. This is why the number of conditions grows the way it does, and it is the content of Section 4.2.4.1.

Preliminary: reduction to autonomous form

The conditions take their cleanest form for an **autonomous** system $\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y})$. This is no loss of generality: the non-autonomous problem $\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}, t)$ becomes autonomous by appending the time variable,

$$\widetilde{\boldsymbol{y}} = \begin{pmatrix} t \\ \boldsymbol{y} \end{pmatrix}, \qquad \widetilde{\boldsymbol{f}}(\widetilde{\boldsymbol{y}}) = \begin{pmatrix} 1 \\ \boldsymbol{f}(\boldsymbol{y}, t) \end{pmatrix}.$$

A Runge–Kutta method is **invariant** under this transformation — it produces the same numerical solution for the original and the augmented system — precisely when the stage-consistency relation below holds. For such methods the autonomous and non-autonomous order conditions coincide, and one may work with whichever form is more convenient.

Stage consistency

Before any order condition comes the relation defining the stage abscissae,

$$\boxed{c_i = \sum_{j=1}^{s} a_{ij}, \qquad i = 1, \dots, s, \qquad \text{that is} \qquad \boldsymbol{A}\mathbf{1} = \boldsymbol{c},} \tag{4.7}$$

with $\mathbf{1} = (1, \dots, 1)^T$. It states that the i-th stage is a consistent approximation of the solution at time $t_k + c_i \Delta t$. It is assumed throughout, it is what makes the method invariant under the autonomisation above, and a tableau that violates it has almost certainly been mistranscribed.

Order 1

In addition to Equation 4.7, one condition:

$$\sum_{i=1}^{s} b_i = 1.$$

The increment must reproduce the first term $\Delta t f$ of the Taylor expansion. Every usable method satisfies it; a tableau that does not is inconsistent and does not converge at all.

Order 2

One further condition:

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}.$$

This is the weight $\frac{1}{2}\Delta t^2 y''$ of the expansion, and it is where the abscissae first enter.

Order 3

Two further conditions:

$$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3}, \qquad \sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{6}.$$

The two correspond to the two distinct elementary differentials of order three, $f_{yy}(f,f)$ and $f_y f_y f$: the first is seen by the abscissae alone, the second only through the internal coupling $\boldsymbol{A}$. An explicit one-stage method can never satisfy the second, and this is the structural reason why more stages are needed as the order grows.

Order 4

Four further conditions:

$$\sum_{i=1}^{s} b_i c_i^3 = \frac{1}{4}, \qquad \sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8},$$

$$\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12}, \qquad \sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24}.$$

Note the last one: it involves a *double* application of $\boldsymbol{A}$, so an explicit method needs at least three nonzero sub-diagonal couplings, hence at least four stages, to satisfy it.

Vector form

Writing the sums as inner products makes the conditions far easier to check by hand. With $\circ$ the componentwise (Hadamard) product, set

$$\boxed{\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c}, \qquad \boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b}, \qquad \boldsymbol{c}^{\circ 2} = \boldsymbol{c} \circ \boldsymbol{c}, \qquad \boldsymbol{c}^{\circ 3} = \boldsymbol{c} \circ \boldsymbol{c} \circ \boldsymbol{c}.} \tag{4.8}$$

Then the eight conditions through order four read

$$\left.\begin{array}{l} \overbrace{\underbrace{\boldsymbol{A}\mathbf{1} = \boldsymbol{c}}_{\text{stage consistency}}, \quad \boldsymbol{b}\cdot\mathbf{1} = 1, \quad \boldsymbol{b}\cdot\boldsymbol{c} = \frac{1}{2}, \quad \boldsymbol{b}\cdot\boldsymbol{c}^{\circ 2} = \frac{1}{3}, \quad \boldsymbol{b}\cdot\boldsymbol{d} = \frac{1}{6}}^{\text{order 3}} \\ \boldsymbol{b}\cdot\boldsymbol{c}^{\circ 3} = \frac{1}{4}, \quad (\boldsymbol{b}\circ\boldsymbol{c})\cdot\boldsymbol{d} = \frac{1}{8}, \quad \boldsymbol{e}\cdot\boldsymbol{c}^{\circ 2} = \frac{1}{12}, \quad \boldsymbol{e}\cdot\boldsymbol{d} = \frac{1}{24} \end{array}\right\} \text{order 4} \tag{4.9}$$

The colours mark the levels and are used consistently in this section: stage consistency, order 1, order 2, order 3, order 4, order 5.

The two auxiliary vectors do all the work: $\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c}$ carries the internal coupling seen from the stages, and $\boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b}$ the same coupling seen from the weights. Both are computed once and reused. Section 7.1 applies this table to a dozen tableaux, and Section 7.3 turns it into a few lines of code.

Order 5, for reference

Nine further conditions, listed here so that the fourth-order claims of the tableaux below can actually be sharpened to *exactly* four:

$$\begin{gathered}
\sum_i b_i c_i^4 = \frac{1}{5}, \qquad \sum_i b_i c_i^2 d_i = \frac{1}{10}, \qquad \sum_i b_i d_i^2 = \frac{1}{20}, \\
\sum_i b_i c_i \big(A c^{\circ 2}\big)_i = \frac{1}{15}, \quad \sum_i b_i \big(A c^{\circ 3}\big)_i = \frac{1}{20}, \quad \sum_i b_i c_i \big(A d\big)_i = \frac{1}{30}, \\
\sum_i b_i \big(A(c \circ d)\big)_i = \frac{1}{40}, \quad \sum_i b_i \big(A^2 c^{\circ 2}\big)_i = \frac{1}{60}, \quad \sum_i b_i \big(A^2 d\big)_i = \frac{1}{120}.
\end{gathered} \tag{4.10}$$

For the classical RK4 tableau the very first of these already fails, $b \cdot c^{\circ 4} = \frac{5}{24} \neq \frac{1}{5}$, so its order is exactly four. All nine hold for the three-stage Radau IIA method (order five) and for the three-stage Gauss–Legendre method (order six).

4.2.4.1 Higher orders: the rooted-tree rule

Writing conditions one by one stops being practical at order five. Butcher's observation is that the terms of the Taylor expansion of the exact solution and of the numerical one are both indexed by **rooted trees**, and that matching them term by term gives one condition per tree.

In bracket notation a tree is built from the single node $\bullet$: $[\tau_1 \cdots \tau_m]$ denotes the tree whose root has the subtrees $\tau_1, \dots, \tau_m$. To each tree τ with $\rho(\tau)$ nodes one associates

- the **elementary differential** $F(\tau)$, a combination of f and its derivatives, which appears in the expansion of the exact solution;
- the **elementary weight** $\Phi(\tau)$, a polynomial in a_{ij}, b_i, c_i, which appears in the expansion of the numerical solution;
- the **density** $\gamma(\tau)$, an integer computed from the tree alone.

Theorem 4.2.2 — Butcher's order conditions. A Runge–Kutta method has order p if and only if

$$\boxed{\Phi(\tau) = \frac{1}{\gamma(\tau)} \quad \text{for every rooted tree } \tau \text{ with } \rho(\tau) \le p.}$$

The first eight trees reproduce exactly the conditions already listed:

ρ	τ	$F(\tau)$	$\Phi(\tau)$	$\gamma(\tau)$
1	$\bullet$	f	$b \cdot 1$	1
2	$[\bullet]$	$f_y f$	$b \cdot c$	2
3	$[\bullet^2]$	$f_{yy}(f,f)$	$b \cdot c^{\circ 2}$	3
3	$[[\bullet]]$	$f_y f_y f$	$b \cdot d$	6
4	$[\bullet^3]$	$f_{yyy}(f,f,f)$	$b \cdot c^{\circ 3}$	4
4	$[\bullet[\bullet]]$	$f_{yy}(f, f_y f)$	$(b \circ c) \cdot d$	8
4	$[[\bullet^2]]$	$f_y f_{yy}(f,f)$	$e \cdot c^{\circ 2}$	12
4	$[[[\bullet]]]$	$f_y f_y f_y f$	$e \cdot d$	24

Two structural facts follow at once and are worth keeping in mind.

- The right-hand side $1/\gamma(\tau)$ depends only on the tree, never on the method: the conditions are the same for explicit and implicit schemes, and only the *solvability* of the system differs.

- A tree of height m — a chain of m nested brackets — produces a condition involving $\boldsymbol{A}^{m-1}$. For an explicit method $\boldsymbol{A}$ is strictly lower triangular and nilpotent, so $\boldsymbol{A}^{m-1} = \boldsymbol{0}$ once $m - 1 \geq s$: the corresponding condition becomes $0 = 1/\gamma(\tau)$ and is unsatisfiable. This is the elementary reason why an explicit s-stage method cannot exceed order s.

How many conditions?

The number of rooted trees with p nodes grows quickly, and so does the cumulative number of conditions that must hold for order p:

Table 4.2: Number of order conditions as a function of the order.

p	new conditions	total through order p
1	1	1
2	1	2
3	2	4
4	4	8
5	9	17
6	20	37
7	48	85
8	115	200
9	286	486
10	719	1205

Constructing a method of order eight by solving two hundred polynomial equations by hand is out of the question; this is why high-order methods are built using the structural shortcuts of Section 4.2.4.1 rather than by brute force.

Attainable order: the Butcher barriers

For **explicit** methods the number of stages limits the order:

Table 4.3: Minimum number of stages of an explicit Runge–Kutta method of order p.

order p	1	2	3	4	5	6	7	8
minimum stages s	1	2	3	4	6	7	9	11

Up to order four an extra stage buys an extra order; from order five on it does not, and the gap widens. This is the practical reason why RK4 occupies the position it does, and why high-order explicit codes (Dormand–Prince 8(7), for instance) use a dozen stages or more.

For **implicit** methods the bound is $p \leq 2s$, attained by the Gauss–Legendre family; Radau methods reach $2s - 1$ and Lobatto methods $2s - 2$, trading one or two orders for the stiff behaviour that the position of the abscissae gives them.

Simplifying assumptions

High-order implicit methods are not obtained by solving the order conditions directly, but through three families of structural identities due to Butcher. For integers p, η, ξ:

$$
\begin{aligned}
B(p)&: \quad \sum_{i=1}^{s} b_i c_i^{k-1} = \frac{1}{k}, && k = 1, \dots, p;\\
C(\eta)&: \quad \sum_{j=1}^{s} a_{ij} c_j^{k-1} = \frac{c_i^k}{k}, && i = 1, \dots, s, \quad k = 1, \dots, \eta;\\
D(\xi)&: \quad \sum_{i=1}^{s} b_i c_i^{k-1} a_{ij} = \frac{b_j}{k}\left(1 - c_j^k\right), && j = 1, \dots, s, \quad k = 1, \dots, \xi.
\end{aligned}
\tag{4.11}
$$

Their meaning is concrete: $B(p)$ says that the weights b_i integrate polynomials of degree $< p$ exactly on $[0, 1]$ — it is a **quadrature** condition; $C(\eta)$ says that each stage integrates polynomials of degree $< \eta$ exactly on $[0, c_i]$; $D(\xi)$ is the adjoint statement.

Theorem 4.2.3 — Butcher's theorem. If a Runge–Kutta method satisfies $B(p)$, $C(\eta)$ and $D(\xi)$ with

$$p \le \eta + \xi + 1 \qquad \text{and} \qquad p \le 2\eta + 2,$$

then it has order p.

This reduces the construction of a high-order implicit method to a quadrature problem. Choosing the c_i as the nodes of a Gauss, Radau or Lobatto quadrature rule on $[0, 1]$ fixes $B(p)$; the linear systems $C(\eta)$ and $D(\xi)$ then determine $\boldsymbol{A}$ and $\boldsymbol{b}$. The classical families are exactly the three choices of nodes:

Table 4.4: Simplifying assumptions satisfied by the classical implicit families.

family	abscissae	B	C	D	order
Gauss–Legendre	interior	$2s$	s	$D(s)$	$2s$
Radau IA	$c_1 = 0$	$2s-1$	$s-1$	$D(s)$	$2s-1$
Radau IIA	$c_s = 1$	$2s-1$	s	$D(s-1)$	$2s-1$
Lobatto IIIA	$c_1 = 0,\ c_s = 1$	$2s-2$	s	$D(s-2)$	$2s-2$
Lobatto IIIB	$c_1 = 0,\ c_s = 1$	$2s-2$	$s-2$	$D(s)$	$2s-2$
Lobatto IIIC	$c_1 = 0,\ c_s = 1$	$2s-2$	$s-1$	$D(s-1)$	$2s-2$

The three-stage members of each family, whose tableaux appear below, illustrate the pattern: Gauss–Legendre reaches order six with three stages, Radau IIA order five, the three Lobatto variants order four. The Lobatto variants differ from one another not in order but in stability: IIIA and IIIB share the same stability function, with $R(\infty) = 1$, so they are A-stable but not L-stable, whereas IIIC has $R(\infty) = 0$ and damps the stiff modes completely. Stability is the subject of the next chapter.

How to use this section

- To **check** a given tableau, use the vector form Equation 4.9 and work level by level; Section 7.1 does this for a dozen methods and Section 7.3 automates it.
- To **construct** an explicit method of low order, write the conditions as a polynomial system in the free coefficients and solve it; the two-stage case is done in full in 1.

- To **construct** an implicit method of high order, never touch the order conditions: choose the quadrature nodes and use Theorem 4.2.3.

4.2.5 Embedded Runge–Kutta Pairs and Adaptive Step Size

In practical ODE solvers the step size is rarely kept fixed. A standard strategy is to use an **embedded Runge–Kutta pair**: the same stages $\boldsymbol{K}_i$ are combined with two different sets of weights, $\boldsymbol{b}$ and $\widehat{\boldsymbol{b}}$, producing two approximations of different orders,

$$z_{k+1} = z_k + \sum_{i=1}^{s} b_i \boldsymbol{K}_i, \qquad \widehat{z}_{k+1} = z_k + \sum_{i=1}^{s} \widehat{b}_i \boldsymbol{K}_i.$$

Their difference

$$\boxed{\boldsymbol{e}_{k+1}^{\text{est}} = z_{k+1} - \widehat{z}_{k+1} = \sum_{i=1}^{s} (b_i - \widehat{b}_i)\boldsymbol{K}_i}$$

is an inexpensive estimate of the local error because no additional evaluations of $\boldsymbol{f}$ are required.

For a vector problem, the error is normally scaled componentwise using absolute and relative tolerances. One common norm is

$$\text{err} = \left[\frac{1}{d} \sum_{j=1}^{d} \left(\frac{e_j^{\text{est}}}{\text{atol}_j + \text{rtol}_j \max(|z_{k,j}|, |z_{k+1,j}|)} \right)^2 \right]^{1/2}.$$

A step is accepted when $\text{err} \le 1$. If the estimator behaves like $C\Delta t^q$, a basic step-size controller is

$$\boxed{\Delta t_{\text{new}} = \eta \Delta t \, \text{err}^{-1/q}}$$

with a safety factor $0 < \eta < 1$ and practical upper/lower bounds on the change factor. For a common embedded pair of orders p and $p-1$, the difference is typically $O(\Delta t^p)$, so $q = p$.

Adaptive step-size control is one of the main reasons modern explicit RK methods such as Bogacki–Shampine and Dormand–Prince are substantially more efficient than fixed-step RK4 in general-purpose software.

Example: Bogacki–Shampine 3(2) pair

A concrete embedded pair is the Bogacki–Shampine method, used in MATLAB's `ode23`. Its four stages are determined by

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 \\ 1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ \hline & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\ & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8} \end{array}$$

The first weight vector gives an order-three approximation,

$$z_{n+1}^{[3]} = z_n + \Delta t \sum_{i=1}^{4} b_i \boldsymbol{k}_i,$$

whereas the second gives an order-two approximation,

$$z_{n+1}^{[2]} = z_n + \Delta t \sum_{i=1}^{4} \widehat{b}_i \boldsymbol{k}_i.$$

The difference

$$\boldsymbol{e}_{n+1}^{\text{est}} = \boldsymbol{z}_{n+1}^{[3]} - \boldsymbol{z}_{n+1}^{[2]}$$

is $O(\Delta t^3)$ and is used to control the local error. An important implementation feature is **FSAL** (*first same as last*): after an accepted step, the last stage

$$\boldsymbol{k}_4 = \boldsymbol{f}(\boldsymbol{z}_{n+1}^{[3]}, t_{n+1})$$

can be reused as the first stage of the next step. Thus, after the first accepted step, only three new right-hand-side evaluations are normally required.

A common controller has the form

$$\Delta t_{\text{new}} = \eta \Delta t \, \text{err}^{-1/3},$$

because the embedded difference is of order Δt^3. In robust implementations this factor is clipped to a prescribed interval to prevent abrupt changes of step size.

The stability function $R(q)$ of a Runge–Kutta method, and the absolute-stability region it defines, are derived from the Butcher tableau in Section 6.4.5.

4.3 Examples of Runge–Kutta Methods

The tableaux below are collected here for reference. Each one can be checked against the conditions of Equation 4.9; the verification is carried out explicitly for a dozen of them in Section 7.1, and automated in Section 7.3.

4.3.1 Explicit methods

Explicit Euler.

The Explicit Euler Method Represented Using a Butcher Tableau:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{4.12}$$

Midpoint method.

The Midpoint Method, also known as the Collatz Method, is a variant of the explicit Euler Method. This approach is classified as a second-order method and consists of two stages, allowing for improved accuracy in numerical integration.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array} \tag{4.13}$$

Heun's method.

Heun's Method, often referred to as the explicit trapezoid rule, enhances the standard explicit Euler Method. This technique is a second-order method that employs two stages to achieve greater accuracy in numerical integration.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array} \tag{4.14}$$

Ralston's method.

Ralston's method is a second-order technique with two stages, specifically designed to achieve a minimal local error bound.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 2/3 & 2/3 & 0 \\ \hline & 1/4 & 3/4 \end{array} \tag{4.15}$$

Generic second-order method.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \alpha & \alpha & 0 \\ \hline & 1-\frac{1}{2\alpha} & \frac{1}{2\alpha} \end{array} \qquad \alpha \neq 0. \tag{4.16}$$

If one additionally wants the internal stage time to lie in the current step, one restricts $0 < \alpha \leq 1$. The endpoint $\alpha = 1$ gives Heun's method and must therefore not be excluded.

Runge–Kutta's third-order method (RK3).

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array} \tag{4.17}$$

Heun's third-order method.

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 0 \\ 2/3 & 0 & 2/3 & 0 \\ \hline & 1/4 & 0 & 3/4 \end{array} \tag{4.18}$$

Ralston's third-order method.

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 3/4 & 0 & 3/4 & 0 \\ \hline & 2/9 & 1/3 & 4/9 \end{array} \tag{4.19}$$

Strong stability preserving Runge–Kutta method (SSPRK3).

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \tag{4.20}$$

Generic third-order method.

A one-parameter family of explicit three-stage, third-order methods with $c_2 = \alpha$ and $c_3 = 1$ is

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \alpha & \alpha & 0 & 0 \\ 1 & \dfrac{3\alpha^2 - 3\alpha + 1}{\alpha(3\alpha - 2)} & \dfrac{\alpha - 1}{\alpha(3\alpha - 2)} & 0 \\ \hline & \dfrac{3\alpha - 1}{6\alpha} & \dfrac{1}{6\alpha(1 - \alpha)} & \dfrac{2 - 3\alpha}{6(1 - \alpha)} \end{array} \tag{4.21}$$

with

$$\alpha \neq 0, \qquad \alpha \neq \frac{2}{3}, \qquad \alpha \neq 1.$$

Substitution into the third-order conditions gives

$$\boldsymbol{b}^T \mathbf{1} = 1, \qquad \boldsymbol{b}^T \boldsymbol{c} = \frac{1}{2}, \qquad \boldsymbol{b}^T \boldsymbol{c}^{\circ 2} = \frac{1}{3}, \qquad \boldsymbol{b}^T \boldsymbol{A} \boldsymbol{c} = \frac{1}{6}.$$

These identities provide a direct check that the family has order three.

Classical Runge–Kutta method (RK4).

The classical fourth-order Runge-Kutta:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array} \tag{4.22}$$

Ralston's fourth-order method.

It is a four-stage, fourth-order Runge–Kutta method whose coefficients were chosen to reduce the leading truncation-error terms.

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 2/5 & 2/5 & 0 & 0 & 0 \\ c_3 & a_{3,1} & a_{3,2} & 0 & 0 \\ 1 & a_{4,1} & a_{4,2} & a_{4,3} & 0 \\ \hline & b_1 & b_2 & b_3 & b_4 \end{array} \tag{4.23}$$

where

$$\begin{aligned} b_1 &= \frac{263}{1812} + \frac{2}{151}\sqrt{5} \\ b_2 &= \frac{125}{3828} - \frac{250}{957}\sqrt{5} \\ b_3 &= \frac{3426304}{5924787} + \frac{553984}{1974929}\sqrt{5} \\ b_4 &= \frac{10}{41} - \frac{4}{123}\sqrt{5} \\ c_3 &= \frac{7}{8} - \frac{3}{16}\sqrt{5} \\ a_{3,1} &= \frac{357}{256}\sqrt{5} - \frac{2889}{1024} \\ a_{4,1} &= \frac{1047}{3020}\sqrt{5} - \frac{673}{1208} \\ a_{3,2} &= \frac{3785}{1024} - \frac{405}{256}\sqrt{5} \\ a_{4,2} &= -\frac{975}{2552} - \frac{1523}{1276}\sqrt{5} \\ a_{4,3} &= \frac{93408}{48169} + \frac{203968}{240845}\sqrt{5} \end{aligned}$$

Fourth-order 3/8-rule method.

The 3/8 rule is a four-stage, fourth-order explicit Runge–Kutta method.

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 0 & 0 \\ 2/3 & -1/3 & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & 1/8 & 3/8 & 3/8 & 1/8 \end{array}$$

4.3.2 Implicit methods

The one- and two-stage schemes below are the implicit counterparts of the explicit methods above; the Gauss–Legendre, Lobatto and Radau families that follow are the high-order methods built from quadrature nodes through the simplifying assumptions of Section 4.2.4.1.

4.3.2.1 Basic implicit methods

Implicit Euler method.

The Implicit Euler Method Represented Using a Butcher Tableau:

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{4.24}$$

Implicit midpoint method.

The implicit midpoint rule is a **one-stage, second-order** implicit Runge–Kutta method. It should not be confused with the explicit midpoint (Collatz) method, which has two stages.

$$\begin{array}{c|c} 1/2 & 1/2 \\ \hline & 1 \end{array} \tag{4.25}$$

Crank–Nicolson method.

The Crank-Nicolson method corresponds to the implicit trapezoidal rule.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \tag{4.26}$$

4.3.2.2 Methods from quadrature nodes

Gauss–Legendre methods.

These methods are based on the points of Gauss-Legendre quadrature.

Two-stage Gauss–Legendre method (order 4).

$$\begin{array}{c|cc} \frac{1}{2}-\frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4}-\frac{\sqrt{3}}{6} \\ \frac{1}{2}+\frac{\sqrt{3}}{6} & \frac{1}{4}+\frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \tag{4.27}$$

Gauss–Legendre sixth-order method.

$$\begin{array}{c|ccc}
\frac{1}{2}-\frac{\sqrt{15}}{10} & \frac{5}{36} & \frac{2}{9}-\frac{\sqrt{15}}{15} & \frac{5}{36}-\frac{\sqrt{15}}{30} \\
\frac{1}{2} & \frac{5}{36}+\frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36}-\frac{\sqrt{15}}{24} \\
\frac{1}{2}+\frac{\sqrt{15}}{10} & \frac{5}{36}+\frac{\sqrt{15}}{30} & \frac{2}{9}+\frac{\sqrt{15}}{15} & \frac{5}{36} \\
\hline
 & \frac{5}{18} & \frac{4}{9} & \frac{5}{18}
\end{array} \tag{4.28}$$

Lobatto methods.

Lobatto IIIA fourth-order method.

$$\begin{array}{c|ccc}
0 & 0 & 0 & 0 \\
1/2 & 5/24 & 1/3 & -1/24 \\
1 & 1/6 & 2/3 & 1/6 \\
\hline
 & 1/6 & 2/3 & 1/6
\end{array} \tag{4.29}$$

Lobatto IIIB fourth-order method.

$$\begin{array}{c|ccc}
0 & 1/6 & -1/6 & 0 \\
1/2 & 1/6 & 1/3 & 0 \\
1 & 1/6 & 5/6 & 0 \\
\hline
 & 1/6 & 2/3 & 1/6
\end{array} \tag{4.30}$$

Lobatto IIIC fourth-order method.

$$\begin{array}{c|ccc}
0 & 1/6 & -1/3 & 1/6 \\
1/2 & 1/6 & 5/12 & -1/12 \\
1 & 1/6 & 2/3 & 1/6 \\
\hline
 & 1/6 & 2/3 & 1/6
\end{array} \tag{4.31}$$

Lobatto IIIC* fourth-order method.

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ 1 & 0 & 1 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array} \tag{4.32}$$

Radau methods.

Radau IA fifth-order method.

$$\begin{array}{c|ccc} 0 & \frac{1}{9} & \frac{-1-\sqrt{6}}{18} & \frac{-1+\sqrt{6}}{18} \\ \frac{3}{5}-\frac{\sqrt{6}}{10} & \frac{1}{9} & \frac{11}{45}+\frac{7\sqrt{6}}{360} & \frac{11}{45}-\frac{43\sqrt{6}}{360} \\ \frac{3}{5}+\frac{\sqrt{6}}{10} & \frac{1}{9} & \frac{11}{45}+\frac{43\sqrt{6}}{360} & \frac{11}{45}-\frac{7\sqrt{6}}{360} \\ \hline & \frac{1}{9} & \frac{4}{9}+\frac{\sqrt{6}}{36} & \frac{4}{9}-\frac{\sqrt{6}}{36} \end{array} \tag{4.33}$$

Radau IIA fifth-order method.

$$\begin{array}{c|ccc} \frac{2}{5}-\frac{\sqrt{6}}{10} & \frac{11}{45}-\frac{7\sqrt{6}}{360} & \frac{37}{225}-\frac{169\sqrt{6}}{1800} & -\frac{2}{225}+\frac{\sqrt{6}}{75} \\ \frac{2}{5}+\frac{\sqrt{6}}{10} & \frac{37}{225}+\frac{169\sqrt{6}}{1800} & \frac{11}{45}+\frac{7\sqrt{6}}{360} & -\frac{2}{225}-\frac{\sqrt{6}}{75} \\ 1 & \frac{4}{9}-\frac{\sqrt{6}}{36} & \frac{4}{9}+\frac{\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{4}{9}-\frac{\sqrt{6}}{36} & \frac{4}{9}+\frac{\sqrt{6}}{36} & \frac{1}{9} \end{array} \tag{4.34}$$

Summary

- A Runge–Kutta method replaces the derivatives of $\boldsymbol{f}$ required by a Taylor method with several evaluations of $\boldsymbol{f}$ at intermediate points. The Butcher tableau $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ contains all the information about the scheme, and the stage-consistency relation $\boldsymbol{A}\mathbf{1} = \boldsymbol{c}$ is presupposed everywhere.
- In general a step is a nonlinear system of dimension sd. The **shape of $\boldsymbol{A}$** decides the cost: full means one $sd \times sd$ system and up to order $2s$; lower triangular (DIRK, SDIRK)

means s systems of size d, with a single Newton matrix per step in the SDIRK case; strictly lower triangular means no system at all, s evaluations of $\boldsymbol{f}$, but order at most s and a bounded stability region.

- Fixed-point iteration on the stages is unusable on stiff problems because its contraction condition $\Delta t L \max_i \sum_j |a_{ij}| < 1$ reinstates exactly the step restriction that implicitness was meant to remove; Newton's method, in simplified or DIRK form, is what makes implicitness worthwhile.
- The order conditions follow from matching the Taylor expansion of the exact solution with that of the increment function. The two-stage case
  (1) shows the mechanism in full; the general case
  (2) gives one condition per rooted tree, $\Phi(\tau) = 1/\gamma(\tau)$, eight of them through order four and seventeen through order five.
- Explicit s-stage methods have order at most s, and from order five on they need strictly more stages than the order; implicit s-stage methods reach at most $2s$. High-order implicit methods are not built by solving the order conditions but by choosing quadrature nodes and applying the simplifying assumptions $B(p)$, $C(\eta)$, $D(\xi)$.
- Embedded pairs provide a local error estimate at no extra cost and are the basis of adaptive step-size control; the FSAL property makes the extra stage almost free.

5. Linear Multistep Methods

Linear multistep methods (LMMs) use information from several previous time levels to advance the numerical solution of an initial value problem

$$z'(t) = f(z(t), t), \qquad z(t_0) = z_0.$$

On a uniform grid $t_n = t_0 + nh$, a general k-step linear multistep method has the form

$$\boxed{\sum_{j=0}^{k} \alpha_j z_{n+j} = h \sum_{j=0}^{k} \beta_j f_{n+j},}$$

where

$$f_{n+j} = f(z_{n+j}, t_{n+j}),$$

and the coefficients α_j, β_j are fixed scalars. We assume $\alpha_k \neq 0$ and normalize the method so that $\alpha_k = 1$ whenever convenient.

The method is **explicit** if $\beta_k = 0$ and **implicit** if $\beta_k \neq 0$.

Unlike one-step Runge–Kutta methods, a k-step method requires k starting values. These may be generated by a suitable one-step method of sufficiently high order.

5.1 Foundations

5.1.1 Characteristic Polynomials

Associated with an LMM are the two polynomials

$$\boxed{\rho(\xi) = \sum_{j=0}^{k} \alpha_j \xi^j, \qquad \sigma(\xi) = \sum_{j=0}^{k} \beta_j \xi^j.}$$

They compactly encode consistency, zero-stability, and absolute stability.

For the scalar test equation

$$z' = \lambda z, \qquad q = h\lambda,$$

the recurrence becomes

$$\sum_{j=0}^{k}(\alpha_j - q\beta_j)z_{n+j} = 0,$$

whose stability polynomial is

$$\boxed{\pi(\xi; q) = \rho(\xi) - q\sigma(\xi).}$$

For a fixed q, numerical stability requires the roots of $\pi(\xi; q)$ to satisfy the root condition: every root must lie in $|\xi| \le 1$, and roots on the unit circle must be simple.

5.1.2 Consistency and Order

Insert the exact smooth solution into the multistep formula and define the local defect

$$\boldsymbol{d}_{n+k} = \sum_{j=0}^{k} \alpha_j z(t_{n+j}) - h \sum_{j=0}^{k} \beta_j z'(t_{n+j}).$$

Expanding about t_n gives

$$z(t_n + jh) = \sum_{m=0}^{\infty} \frac{j^m h^m}{m!} z^{(m)}(t_n).$$

Therefore the coefficient of $h^m z^{(m)}(t_n)$ is

$$\frac{1}{m!}\left(\sum_{j=0}^{k} \alpha_j j^m - m \sum_{j=0}^{k} \beta_j j^{m-1}\right), \qquad m \ge 1,$$

while for $m = 0$ it is $\sum_j \alpha_j$.

Hence an LMM has order at least p if and only if

$$\boxed{\sum_{j=0}^{k} \alpha_j = 0,}$$

and

$$\boxed{\sum_{j=0}^{k} \alpha_j j^m = m \sum_{j=0}^{k} \beta_j j^{m-1}, \qquad m = 1, \dots, p.}$$

In terms of the characteristic polynomials, **consistency** (order at least one) is equivalent to

$$\boxed{\rho(1) = 0, \qquad \rho'(1) = \sigma(1).}$$

The first identity states that constant exact solutions are reproduced; the second states that linear exact solutions are reproduced to first order.

5.2 Adams methods

Adams methods are obtained by integrating

$$z(t_{n+1}) = z(t_n) + \int_{t_n}^{t_{n+1}} f(z(t), t)\,\mathrm{d}t$$

and replacing the integrand by an interpolation polynomial through previously computed values of $\boldsymbol{f}$.

5.2.1 Adams–Bashforth methods

Adams–Bashforth methods use only already known values

$$f_n, f_{n-1}, \dots,$$

and are therefore explicit.

Using a constant interpolation polynomial gives explicit Euler:

$$\boxed{z_{n+1} = z_n + h f_n} \qquad \text{(AB1, order 1).}$$

Linear interpolation through f_n and f_{n-1} gives

$$\boxed{z_{n+1} = z_n + h\left(\frac{3}{2}f_n - \frac{1}{2}f_{n-1}\right)}$$

(AB2, order 2).

The next two members are

$$\boxed{z_{n+1} = z_n + h\left(\frac{23}{12}f_n - \frac{16}{12}f_{n-1} + \frac{5}{12}f_{n-2}\right)}$$

(AB3, order 3), and

$$\boxed{z_{n+1} = z_n + h\left(\frac{55}{24}f_n - \frac{59}{24}f_{n-1} + \frac{37}{24}f_{n-2} - \frac{9}{24}f_{n-3}\right)}$$

(AB4, order 4).

Derivation of AB2

Set

$$t = t_n + \theta h, \qquad 0 \le \theta \le 1.$$

The linear interpolation polynomial through $\theta = 0$ and $\theta = -1$ is

$$p(\theta) = (1+\theta)f_n - \theta f_{n-1}.$$

Therefore

$$\int_{t_n}^{t_{n+1}} f(z(t), t)\,\mathrm{d}t \approx h\int_0^1 p(\theta)\,\mathrm{d}\theta = h\left(\frac{3}{2}f_n - \frac{1}{2}f_{n-1}\right),$$

which is AB2.

5.2.2 Adams–Moulton methods

Adams–Moulton methods include the unknown endpoint value f_{n+1} in the interpolation formula and are therefore implicit.

The first members are

$$\boxed{z_{n+1} = z_n + h f_{n+1}}$$

(AM1, implicit Euler, order 1),

$$\boxed{z_{n+1} = z_n + \frac{h}{2}(f_{n+1} + f_n)}$$

(AM2, trapezoidal rule, order 2),

$$\boxed{z_{n+1} = z_n + h\left(\frac{5}{12}f_{n+1} + \frac{2}{3}f_n - \frac{1}{12}f_{n-1}\right)}$$

(AM3, order 3), and

$$\boxed{z_{n+1} = z_n + h\left(\frac{3}{8}f_{n+1} + \frac{19}{24}f_n - \frac{5}{24}f_{n-1} + \frac{1}{24}f_{n-2}\right)}$$

(AM4, order 4).

Because $f_{n+1} = f(z_{n+1}, t_{n+1})$, an Adams–Moulton step generally requires a nonlinear solve.

Predictor–corrector implementation

A classical inexpensive implementation combines an Adams–Bashforth predictor with an Adams–Moulton corrector. For example, AB2 can predict

$$z_{n+1}^{(0)} = z_n + h\left(\frac{3}{2}f_n - \frac{1}{2}f_{n-1}\right),$$

then the trapezoidal rule corrects it via

$$z_{n+1}^{(m+1)} = z_n + \frac{h}{2}\left[f_n + f(z_{n+1}^{(m)}, t_{n+1})\right].$$

Repeated correction is a fixed-point iteration. For stiff problems, Newton-type solves are generally preferable.

5.3 Backward Differentiation Formulas (BDF)

BDF methods interpolate the solution values and differentiate the interpolation polynomial at the newest time. Consequently, only the newest right-hand-side value appears:

$$\boxed{\sum_{j=0}^{k} \alpha_j z_{n+j} = h\beta_k f_{n+k}.}$$

After normalization, the first six BDF formulas may be written as follows.

BDF1

$$\boxed{z_{n+1} - z_n = hf_{n+1}}$$

(order 1; implicit Euler).

BDF2

$$\boxed{\frac{3}{2}z_{n+1} - 2z_n + \frac{1}{2}z_{n-1} = hf_{n+1}}$$

(order 2).

BDF3

$$\boxed{\frac{11}{6}z_{n+1} - 3z_n + \frac{3}{2}z_{n-1} - \frac{1}{3}z_{n-2} = hf_{n+1}}$$

(order 3).

BDF4

$$\boxed{\frac{25}{12}z_{n+1} - 4z_n + 3z_{n-1} - \frac{4}{3}z_{n-2} + \frac{1}{4}z_{n-3} = hf_{n+1}}$$

(order 4).

BDF5

$$\boxed{\frac{137}{60}z_{n+1} - 5z_n + 5z_{n-1} - \frac{10}{3}z_{n-2} + \frac{5}{4}z_{n-3} - \frac{1}{5}z_{n-4} = hf_{n+1}}$$

(order 5).

BDF6

$$\boxed{\frac{49}{20}z_{n+1} - 6z_n + \frac{15}{2}z_{n-1} - \frac{20}{3}z_{n-2} + \frac{15}{4}z_{n-3} - \frac{6}{5}z_{n-4} + \frac{1}{6}z_{n-5} = hf_{n+1}}$$

(order 6).

BDF methods are especially important for stiff problems. BDF1 and BDF2 are A-stable. BDF methods of orders 3–6 are not A-stable, but they retain large sectors of the left half-plane in their stability regions and are widely used in variable-order stiff integrators.

A crucial structural fact is that the classical BDF family is zero-stable only for orders 1 through 6; the formulas with $k \geq 7$ violate the root condition.

5.4 Stability and Convergence

5.4.1 Zero-Stability

Consistency controls the defect made in a single idealized step. It does **not** by itself guarantee convergence: perturbations introduced at successive steps must not be amplified without bound as $h \to 0$.

To isolate this mechanism, set $h = 0$ in the LMM. The homogeneous recurrence is

$$\sum_{j=0}^{k} \alpha_j \boldsymbol{v}_{n+j} = \mathbf{0}.$$

Its behavior is determined by the first characteristic polynomial ρ.

Definition 5.4.1 — Definition: Zero-Stability. A linear multistep method is **zero-stable** if solutions of its homogeneous recurrence remain uniformly bounded on every fixed finite time interval when the starting values are uniformly bounded, independently of sufficiently small h.

The algebraic criterion is the fundamental **root condition**.

Theorem 5.4.1 — Root Condition. A linear multistep method is zero-stable if and only if every root ξ_i of

$$\rho(\xi) = 0$$

satisfies

$$|\xi_i| \leq 1,$$

and every root with $|\xi_i| = 1$ is simple.

Why the Root Condition Is Necessary

If ρ has a root $|\xi| > 1$, the homogeneous recurrence contains a mode proportional to ξ^n, which grows geometrically. Since on a fixed interval $n \sim T/h \to \infty$ as $h \to 0$, arbitrarily small perturbations can be amplified without bound.

If a root on the unit circle has multiplicity $m > 1$, the recurrence contains generalized modes of the form

$$n^r \xi^n, \qquad r = 1, \dots, m-1,$$

which also become unbounded as $n \to \infty$. Thus all unit-modulus roots must be simple.

Conversely, if all roots satisfy the stated condition, the companion recurrence matrix has powers that remain uniformly bounded; hence perturbations of the starting data remain bounded on finite intervals. This proves sufficiency.

Adams Methods Are Zero-Stable

All Adams–Bashforth and Adams–Moulton methods have

$$\rho(\xi) = \xi^k - \xi^{k-1} = \xi^{k-1}(\xi - 1).$$

Thus the roots are 0 and 1, with 1 simple, so every Adams method is zero-stable.

BDF and Zero-Stability

For BDF methods, the root condition holds for $k \le 6$ and fails for $k \ge 7$. This is why the classical BDF family stops at order six in practical ODE solvers.

5.4.2 Convergence and the Dahlquist Equivalence Theorem

For an LMM, define the global error

$$\boldsymbol{e}_n = \boldsymbol{z}(t_n) - \boldsymbol{z}_n.$$

A method is **convergent** if, for every sufficiently regular well-posed IVP and consistent starting values,

$$\max_{0 \le n \le N} \|\boldsymbol{e}_n\| \to 0 \qquad \text{as } h \to 0,$$

while $Nh = T$ remains fixed.

The fundamental theorem of linear multistep analysis is the following.

Theorem 5.4.2 — Dahlquist Equivalence Theorem. For a linear multistep method, the following are equivalent:

1. the method is convergent;
2. the method is consistent and zero-stable.

Equivalently,

$$\boxed{\text{consistency} + \text{zero-stability} \iff \text{convergence.}}$$

Proof Idea: Sufficiency

Insert the exact solution into the method. The global error satisfies a perturbed recurrence of the form

$$\sum_{j=0}^{k} \alpha_j \boldsymbol{e}_{n+j} = h \sum_{j=0}^{k} \beta_j \left[\boldsymbol{f}(\boldsymbol{z}(t_{n+j}), t_{n+j}) - \boldsymbol{f}(\boldsymbol{z}_{n+j}, t_{n+j}) \right] + \boldsymbol{d}_{n+k},$$

where $\boldsymbol{d}_{n+k}$ is the local defect. If the method has order p, then

$$\|\boldsymbol{d}_{n+k}\| \le C h^{p+1}.$$

Zero-stability gives a uniform bound on the discrete resolvent of the homogeneous recurrence. If $\boldsymbol{f}$ is Lipschitz in its state argument, the nonlinear difference satisfies

$$\|\boldsymbol{f}(\boldsymbol{u}, t) - \boldsymbol{f}(\boldsymbol{v}, t)\| \leq L\|\boldsymbol{u} - \boldsymbol{v}\|.$$

Applying the zero-stability bound and then a discrete Grönwall inequality yields

$$\max_{0\leq n\leq N} \|\boldsymbol{e}_n\| \leq C_T \left(\max_{0\leq j<k} \|\boldsymbol{e}_j\| + h^p \right).$$

Therefore consistent starting values and consistency of order p imply convergence of order p on every fixed finite interval.

Why Both Hypotheses Are Necessary

If a method converges for every sufficiently smooth IVP, it must reproduce constant and linear solutions asymptotically; this forces

$$\rho(1) = 0, \qquad \rho'(1) = \sigma(1),$$

so the method is consistent.

If it were not zero-stable, arbitrarily small perturbations in the starting values could be amplified without bound even for the trivial equation

$$\boldsymbol{z}' = \boldsymbol{0}.$$

That would contradict convergence. Hence zero-stability is necessary as well.

5.4.3 Absolute Stability of Multistep Methods

For the test equation $z' = \lambda z$, a multistep method leads to

$$\pi(\xi; q) = \rho(\xi) - q\sigma(\xi), \qquad q = h\lambda.$$

The **absolute-stability region** is the set of all $q \in \mathbb{C}$ such that every root $\xi_i(q)$ of $\pi(\xi; q) = 0$ satisfies

$$|\xi_i(q)| \leq 1,$$

with all roots on the unit circle simple.

The boundary of the region can often be parametrized by setting $\xi = e^{i\theta}$:

$$\boxed{q(\theta) = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})}, \qquad 0 \leq \theta \leq 2\pi,}$$

where $\sigma(e^{i\theta}) \neq 0$.

This formula is the multistep analogue of the Runge–Kutta stability function.

Dahlquist Barriers

Two classical impossibility results explain important design limitations of linear multistep methods.

1. **First Dahlquist barrier.** A zero-stable explicit k-step linear multistep method has order at most k. More generally, a zero-stable k-step LMM satisfies $p \leq k + 1$ when k is odd and $p \leq k + 2$ when k is even.
2. **Second Dahlquist barrier.** An A-stable linear multistep method has order at most two.

Consequently, no high-order classical LMM can be both A-stable and of arbitrary order. The trapezoidal rule and BDF2 attain order two while remaining A-stable; higher-order BDF methods sacrifice full A-stability for sectorial stability.

5.5 Practical Comparison

The three main families have complementary roles.

Family	Explicit/implicit	Typical strength	Main limitation
Adams–Bashforth	explicit	low cost per step for non-stiff problems	restricted stability region
Adams–Moulton	implicit	higher accuracy using endpoint information	nonlinear solve may be required
BDF	implicit	strong performance on stiff dissipative problems	classical family zero-stable only through order 6

Linear multistep methods reuse previously computed information and can therefore require fewer new evaluations of f per step than Runge–Kutta methods. Their main complications are startup, step-size changes, storage of history, and the additional zero-stability analysis absent from one-step methods.

Modern variable-step, variable-order stiff solvers are often based on BDF or closely related formulas, while modern non-stiff solvers frequently use embedded Runge–Kutta methods because adaptive step-size control and dense output are easier to organize in a one-step framework.

6. Absolute Stability and Stability Regions

6.1 Why study stability? Start with explicit Euler

Consider the simple initial-value problem

$$y'(t) = -y(t), \qquad y(0) = 1.$$

Its exact solution is $y(t) = e^{-t}$: it is positive, decreases monotonically, and tends to zero. With a constant step $h > 0$, explicit Euler gives

$$y_{n+1} = y_n + h(-y_n) = (1-h)y_n, \qquad y_n = (1-h)^n.$$

The numerical behavior changes dramatically as h varies:

Step size	Amplification $1-h$	Numerical behavior
$0 < h < 1$	$0 < 1-h < 1$	positive monotone decay
$h = 1$	0	reaches zero in one step
$1 < h < 2$	$-1 < 1-h < 0$	alternating signs, decreasing amplitude
$h = 2$	-1	undamped sign alternation
$h > 2$	$1-h < -1$	alternating signs, growing amplitude

The differential equation is perfectly stable, but the numerical solution can diverge. **Stability of the ODE and stability of its discretization are different properties.** Notice also that decay of the numerical solution does not imply preservation of positivity or monotonicity.

```
hs <- c(0.5, 1, 1.5, 2, 2.3)
par(mfrow = c(2, 3))
for (h in hs) {
  n  <- 0:10
  tn <- n*h
  yn <- (1-h)^n
  tt <- seq(0, max(tn), length.out = 400)
```

```
  # la curva esatta va disegnata per prima, cosi' i punti di Euler le stanno sopra
  plot(tn, yn, type = "n",
       xlab = "t", ylab = expression(y[n]),
       main = paste("h =", h))
  abline(h = 0, lty = 3)
  lines(tt, exp(-tt), col = "grey55", lty = 2, lwd = 1.5)
  lines(tn, yn, type = "b", pch = 16)
}
# sesto riquadro: la sola soluzione esatta, sullo stesso intervallo del caso h = 0.5
t <- seq(0, 5, length.out = 400)
plot(t, exp(-t), type = "n",
     xlab = "t", ylab = expression(y(t)), ylim = c(0, 1),
     main = expression(paste("exact: ", e^-t)))
abline(h = 0, lty = 3)
lines(t, exp(-t), col = "grey55", lty = 2, lwd = 1.5)
legend("topright", bty = "n", cex = 0.9,
       legend = c("exact", "Euler"),
       col = c("grey55", "black"), lty = c(2, 1),
       pch = c(NA, 16), lwd = c(1.5, 1))
par(mfrow = c(1, 1))
```

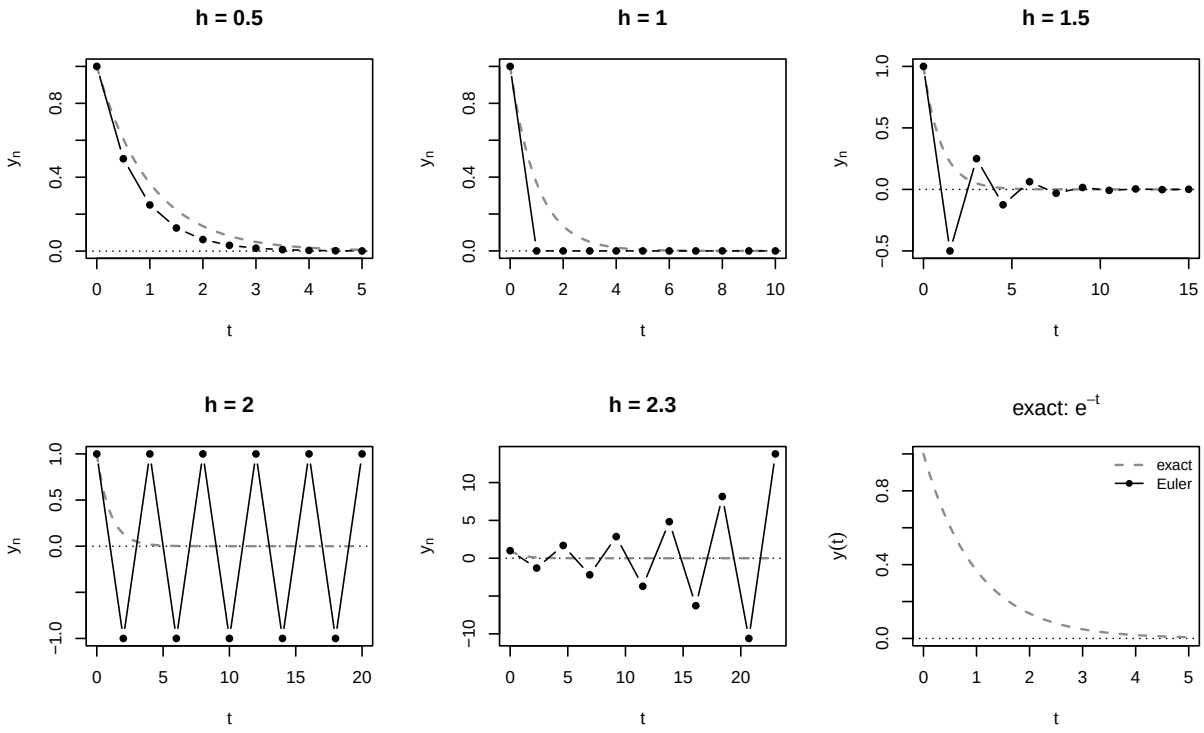


Figure 6.1: Explicit Euler for y'=-y: decay, alternating decay, neutral oscillations and instability. The exact solution is overlaid in grey.

6.2 Why do complex numbers appear? Linear systems and diagonalization

The scalar example is not an artificial choice. Consider a linear autonomous system

$$\boldsymbol{y}'(t) = \boldsymbol{A}\boldsymbol{y}(t), \qquad \boldsymbol{y}(0) = \boldsymbol{y}_0, \qquad \boldsymbol{A} \in \mathbb{R}^{m\times m}.$$

Assume initially that $\boldsymbol{A}$ is diagonalizable over $\mathbb{C}$:

$$A = V\Lambda V^{-1}, \qquad \Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_m).$$

Introduce the modal coordinates $z = V^{-1}y$. Because V is constant,

$$z' = V^{-1}y' = V^{-1}AVz = \Lambda z, \qquad z_j' = \lambda_j z_j.$$

Diagonalization is useful because it separates a coupled system into independent scalar modes. Each mode is described by the same test equation, with a different eigenvalue. A real matrix can have nonreal eigenvalues, so restricting the test equation to real λ would miss oscillatory dynamics.

For example, take

$$A = \begin{pmatrix} -1 & -4 \\ 4 & -1 \end{pmatrix}, \qquad \lambda_{1,2} = -1 \pm 4i.$$

The real solution is a decaying rotation:

$$y(t) = e^{-t} \begin{pmatrix} \cos(4t) & -\sin(4t) \\ \sin(4t) & \cos(4t) \end{pmatrix} y_0.$$

The real part -1 determines exponential damping; the imaginary parts ± 4 determine oscillation frequency. In general, writing $\lambda = a + ib$ gives

$$e^{\lambda t} = e^{at}(\cos(bt) + i\sin(bt)), \qquad |e^{\lambda t}| = e^{at}.$$

Thus each continuous mode decays when $\operatorname{Re}(\lambda) < 0$. For a diagonalizable system, all modes decay if all eigenvalues have negative real part. Non-diagonalizable systems require Jordan blocks, but the scalar equation still identifies the basic amplification mechanism; for non-normal matrices, transient growth may occur even when all modes are asymptotically stable.

6.2.1 Explicit Euler applied to the system

Explicit Euler applied directly to the system yields

$$y_{n+1} = (I + hA)y_n.$$

In modal coordinates,

$$z_{j,n+1} = (1 + h\lambda_j)z_{j,n}.$$

The continuous mode decays if $\operatorname{Re}(\lambda_j) < 0$, whereas the discrete mode decays strictly if

$$\boxed{|1 + h\lambda_j| < 1.}$$

For the example $\lambda = -1 + 4i$,

$$|1 + h\lambda|^2 = (1 - h)^2 + 16h^2 = 1 - 2h + 17h^2.$$

Hence explicit Euler damps both oscillatory modes only if $0 < h < 2/17$. The restriction is much more severe than $h < 2$, obtained from the scalar equation $y' = -y$. This is precisely why **stability must be studied in the complex plane**.

6.3 The complex scalar test equation and the stability region

The standard Dahlquist test equation is

$$z'(t) = \lambda z(t), \qquad \lambda \in \mathbb{C}.$$

Over one step its exact amplification is

$$z(t_{n+1}) = e^q z(t_n), \qquad q = h\lambda.$$

For a one-step method, define its *stability function* $R(q)$ through

$$z_{n+1} = R(q)z_n, \qquad z_n = R(q)^n z_0.$$

For a fixed step, $|R(q)| < 1$ gives asymptotic decay; $|R(q)| = 1$ gives bounded constant-amplitude propagation for this scalar recurrence; $|R(q)| > 1$ gives growth. We distinguish the **closed non-amplification region**

$$\boxed{S = \{q \in \mathbb{C} : |R(q)| \leq 1\}}$$

from its strict interior $|R(q)| < 1$, which gives decay. Boundary points need care: boundedness does not mean decay. For consistent methods $R(q) = 1 + q + \mathcal{O}(q^2)$ near $q = 0$.

> **Important.** Absolute stability of the test equation is distinct from existence and uniqueness of ODE solutions, consistency, zero-stability, convergence and accuracy.

6.3.1 Explicit Euler: derivation of its disk

On $z' = \lambda z$, explicit Euler reads

$$z_{n+1} = z_n + h\lambda z_n = (1 + q)z_n, \qquad \boxed{R_{\mathrm{EE}}(q) = 1 + q}.$$

Write $q = x + iy$. Then

$$|R_{\mathrm{EE}}(q)| \leq 1 \iff |1 + x + iy|^2 \leq 1 \iff \boxed{(x + 1)^2 + y^2 \leq 1}.$$

This is the **closed disk centered at** $(-1, 0)$ **with radius** 1. For strictly decaying modes one needs the open disk. Along the negative real axis the closed interval is $[-2, 0]$, and for $\lambda < 0$ the strict-decay condition is $0 < h < 2/|\lambda|$. At $h = 2/|\lambda|$ the numerical mode alternates without decay.

```
th <- seq(0, 2*pi, length.out = 500)
plot(-1 + cos(th), sin(th), type = "l", asp = 1,
     xlim = c(-2.5, 0.6), ylim = c(-1.4, 1.4),
     xlab = expression(Re(q)), ylab = expression(Im(q)))
polygon(-1 + cos(th), sin(th), col = adjustcolor("skyblue", alpha.f = 0.25), border = NA)
lines(-1 + cos(th), sin(th), lwd = 2)
abline(h = 0, v = 0, lty = 3)
points(-1, 0, pch = 16)
```

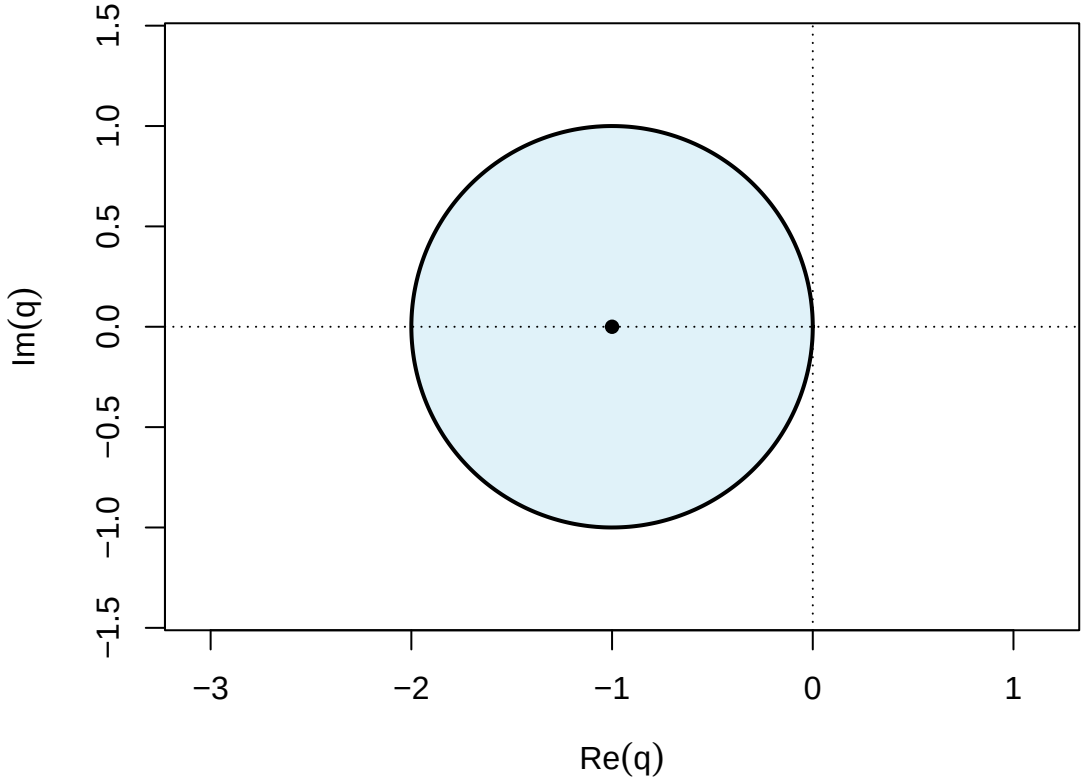


Figure 6.2: Explicit Euler stability disk in the complex q-plane, including its boundary.

For a diagonalizable system, each scaled eigenvalue $q_j = h\lambda_j$ must lie in this disk to prevent growth of its mode. This geometric condition explains the meaning of a *stability region*: varying h moves every point $h\lambda_j$ along a ray from the origin.

6.4 Runge–Kutta methods: deriving the stability function step by step

We now derive, rather than merely state, the amplification factor of Runge–Kutta methods. We use the scalar test equation

$$z' = \lambda z, \qquad q = h\lambda, \qquad \lambda \in \mathbb{C}.$$

For an s-stage RK method, with Butcher coefficients $A = (a_{ij})$, $b = (b_i)$ and $c = (c_i)$, define the stage **increments** K_i by

$$K_i = hf\left(t_n + c_i h, z_n + \sum_{j=1}^{s} a_{ij} K_j\right), \qquad z_{n+1} = z_n + \sum_{i=1}^{s} b_i K_i.$$

On the test equation, $f(z,t) = \lambda z$, so each stage becomes

$$\boxed{K_i = q\left(z_n + \sum_{j=1}^{s} a_{ij} K_j\right).}$$

This is the key substitution: **each K_i is a polynomial in q times z_n for explicit RK methods**. The final update then has the form $z_{n+1} = R(q)z_n$.

6.4.1 Explicit Euler (one stage)

Its tableau is

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}.$$

There is just one stage, $K_1 = qz_n$, hence

$$z_{n+1} = z_n + K_1 = (1+q)z_n, \qquad \boxed{R_{\mathrm{EE}}(q) = 1 + q}.$$

6.4.2 Explicit second-order Runge–Kutta: Heun's method

Heun's tableau and update are

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}, \qquad z_{n+1} = z_n + \tfrac{1}{2}(K_1 + K_2).$$

Substitute $z' = \lambda z$ **one stage at a time**:

$$K_1 = qz_n,$$
$$K_2 = q(z_n + K_1) = q(z_n + qz_n) = (q + q^2)z_n.$$

Therefore

$$\begin{aligned} z_{n+1} &= z_n + \tfrac{1}{2}\left[qz_n + (q+q^2)z_n\right] \\ &= \left(1 + q + \tfrac{1}{2}q^2\right) z_n, \end{aligned}$$

and thus

$$\boxed{R_{\mathrm{Heun}}(q) = 1 + q + \frac{q^2}{2}.}$$

6.4.3 Explicit midpoint RK2: the same stability polynomial

The explicit midpoint tableau is

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}.$$

Its stages are

$$K_1 = qz_n, \qquad K_2 = q\left(z_n + \tfrac{1}{2}K_1\right) = \left(q + \tfrac{1}{2}q^2\right) z_n.$$

Since $z_{n+1} = z_n + K_2$,

$$\boxed{R_{\mathrm{midpoint}}(q) = 1 + q + \frac{q^2}{2}.}$$

More generally, **all explicit two-stage RK methods of order two** (including Ralston's method) have this same stability polynomial, despite having different stage coefficients. On the negative real axis, their closed non-amplification interval is $[-2, 0]$: order two does not, by itself, increase the stable step for a real negative eigenvalue relative to explicit Euler.

6.4.4 Classical fourth-order Runge–Kutta (RK4)

The classical RK4 tableau is

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}.$$

On $z' = \lambda z$, calculate the four stage increments successively:

$$\begin{aligned} K_1 &= q z_n, \\ K_2 &= q\left(z_n + \tfrac{1}{2}K_1\right) = \left(q + \tfrac{1}{2}q^2\right) z_n, \\ K_3 &= q\left(z_n + \tfrac{1}{2}K_2\right) = \left(q + \tfrac{1}{2}q^2 + \tfrac{1}{4}q^3\right) z_n, \\ K_4 &= q(z_n + K_3) = \left(q + q^2 + \tfrac{1}{2}q^3 + \tfrac{1}{4}q^4\right) z_n. \end{aligned}$$

Now insert these expressions in the RK4 weighted update:

$$\begin{aligned} z_{n+1} &= z_n + \tfrac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ &= z_n + \frac{z_n}{6}\left[q + 2\left(q + \tfrac{1}{2}q^2\right) + 2\left(q + \tfrac{1}{2}q^2 + \tfrac{1}{4}q^3\right) + \left(q + q^2 + \tfrac{1}{2}q^3 + \tfrac{1}{4}q^4\right)\right] \\ &= \left(1 + q + \frac{q^2}{2} + \frac{q^3}{6} + \frac{q^4}{24}\right) z_n. \end{aligned}$$

Consequently,

$$\boxed{R_{\mathrm{RK4}}(q) = 1 + q + \frac{q^2}{2} + \frac{q^3}{6} + \frac{q^4}{24}.}$$

This is the degree-four Taylor polynomial of e^q. The closed stability interval along the negative real axis is approximately $[-2.785293563, 0]$. The region is nevertheless bounded, as for every explicit RK method.

6.4.5 General RK stability function from the Butcher tableau

The previous calculations can be collected in matrix notation. Set

$$K = (K_1, \dots, K_s)^T, \qquad \mathbf{e} = (1, \dots, 1)^T.$$

The stage equations become

$$K = q\mathbf{e}\, z_n + qAK, \qquad (I - qA)K = q\mathbf{e}\, z_n.$$

Whenever $I - qA$ is invertible,

$$K = q(I - qA)^{-1}\mathbf{e}\, z_n.$$

Substituting into $z_{n+1} = z_n + b^T K$ gives the general formula

$$\boxed{R(q) = 1 + q\, b^T (I - qA)^{-1}\mathbf{e}.}$$

For an **explicit** s-stage method, A is strictly lower triangular and hence nilpotent: $A^s = 0$. Thus

$$(I - qA)^{-1} = I + qA + q^2A^2 + \cdots + q^{s-1}A^{s-1},$$

and

$$\boxed{R(q) = 1 + \sum_{k=0}^{s-1} q^{k+1} b^T A^k \mathbf{e}.}$$

So R is a polynomial of degree **at most** s (not necessarily exactly s). In particular, if the method has order p, then

$$R(q) = \sum_{j=0}^{p} \frac{q^j}{j!} + \mathcal{O}(q^{p+1}) \quad (q \to 0).$$

For implicit RK methods, A need not be nilpotent and $R(q)$ is generally **rational**, with poles at zeros of $\det(I - qA)$. Its stability region is the set of points where the function is

defined and $|R(q)| \le 1$. This formula also provides a systematic route to stability-region plots for arbitrary Butcher tableaux.

6.4.6 Implicit Euler

Implicit Euler gives

$$z_{n+1} = z_n + qz_{n+1},$$

and hence

$$\boxed{R_{\mathrm{IE}}(q) = \frac{1}{1-q}.}$$

The stability condition is

$$|1 - q| \ge 1.$$

The entire closed left half-plane belongs to S, so implicit Euler is **A-stable**. Moreover,

$$R_{\mathrm{IE}}(q) \longrightarrow 0 \qquad \text{as } |q| \to \infty, \quad \mathrm{Re}(q) < 0,$$

so it is also **L-stable**. Very fast stable modes are therefore strongly damped.

6.4.7 Trapezoidal Rule

The implicit trapezoidal rule gives

$$z_{n+1} = z_n + \frac{q}{2}(z_n + z_{n+1}),$$

so

$$\boxed{R_{\mathrm{TR}}(q) = \frac{1 + q/2}{1 - q/2}.}$$

It is A-stable because $|R_{\mathrm{TR}}(q)| \le 1$ for $\mathrm{Re}(q) \le 0$. However,

$$R_{\mathrm{TR}}(q) \longrightarrow -1 \qquad \text{as } q \to -\infty.$$

Thus the trapezoidal rule is **not L-stable**. Extremely stiff decaying modes are not damped to zero; instead they can survive numerically as alternating oscillations. This is a fundamental practical difference between trapezoidal and implicit Euler.

6.5 A-Stability and L-Stability

Definition 6.5.1 — Definition: A-Stability. A one-step method is **A-stable** if

$$\boxed{\{q \in \mathbb{C} : \mathrm{Re}(q) \le 0\} \subseteq S.}$$

Thus every stable mode of the scalar test equation remains numerically stable for every positive step size.

Definition 6.5.2 — Definition: L-Stability. An A-stable one-step method is **L-stable** if additionally

$$\boxed{R(q) \to 0 \quad \text{as } |q| \to \infty \text{ in the left half-plane.}}$$

A-stability prevents growth of arbitrarily stiff stable modes; L-stability additionally suppresses them. This distinction is important for dissipative stiff systems.

6.6 Stability for Systems and Multistep Methods

6.6.1 From the Scalar Test Equation to Systems

Consider the linear system

$$\boldsymbol{y}' = \boldsymbol{A}\boldsymbol{y}.$$

For a Runge–Kutta method the scalar stability function extends formally to the matrix argument

$$\boldsymbol{y}_{n+1} = R(h\boldsymbol{A})\boldsymbol{y}_n.$$

If $\boldsymbol{A}$ is diagonalizable,

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{-1},$$

then

$$R(h\boldsymbol{A}) = \boldsymbol{V}R(h\boldsymbol{\Lambda})\boldsymbol{V}^{-1}.$$

The eigenvalues $h\lambda_j$ therefore indicate which scalar stability modes are present. A common practical requirement is

$$h\lambda_j \in \mathcal{S} \qquad \text{for all relevant eigenvalues } \lambda_j.$$

For normal matrices this spectral picture is especially informative. For strongly non-normal matrices, eigenvalues alone can be insufficient because transient growth may be governed by the conditioning of $\boldsymbol{V}$, the pseudospectrum, or related non-normal effects. Absolute-stability plots should therefore not be interpreted as a complete stability theory for every nonlinear or non-normal problem.

6.6.2 Linear Multistep Methods

For a k-step linear multistep method

$$\sum_{j=0}^{k} \alpha_j \boldsymbol{z}_{n+j} = h \sum_{j=0}^{k} \beta_j \boldsymbol{f}_{n+j},$$

application to $z' = \lambda z$ gives

$$\sum_{j=0}^{k} (\alpha_j - q\beta_j) z_{n+j} = 0.$$

With

$$\rho(\xi) = \sum_{j=0}^{k} \alpha_j \xi^j, \qquad \sigma(\xi) = \sum_{j=0}^{k} \beta_j \xi^j,$$

the stability polynomial is

$$\boxed{\pi(\xi; q) = \rho(\xi) - q\sigma(\xi).}$$

Unlike a one-step method, a multistep method generally has several amplification roots $\xi_j(q)$. Therefore stability cannot in general be described by a single function $R(q)$.

Definition 6.6.1 — Definition: Absolute Stability for an LMM. A value q belongs to the absolute-stability region if every root of

$$\pi(\xi; q) = 0$$

satisfies

$$|\xi_j(q)| \le 1,$$

and every root on $|\xi| = 1$ is simple.

The simplicity condition is essential: a repeated unit-modulus root produces polynomial growth in the recurrence even though its modulus is one.

6.6.3 Boundary-Locus Method

At a regular point of the stability boundary, at least one characteristic root lies on the unit circle. Set

$$\xi = e^{i\theta}, \qquad 0 \le \theta \le 2\pi.$$

Then

$$\rho(e^{i\theta}) - q\sigma(e^{i\theta}) = 0,$$

and hence

$$\boxed{q(\theta) = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})}, \qquad \sigma(e^{i\theta}) \neq 0.}$$

This is the **boundary locus**. The curve alone does not determine which side is stable; one must test a point in each connected component or directly inspect the roots of $\pi(\xi; q)$.

6.6.4 BDF Methods

6.6.4.1 BDF2

BDF2 is

$$\frac{3z_{n+2} - 4z_{n+1} + z_n}{2h} = f_{n+2}.$$

Thus

$$\rho(\xi) = \frac{1}{2} - 2\xi + \frac{3}{2}\xi^2, \qquad \sigma(\xi) = \xi^2,$$

and

$$\pi(\xi; q) = \left(\frac{3}{2} - q\right)\xi^2 - 2\xi + \frac{1}{2}.$$

Its boundary locus is

$$\boxed{q(\theta) = \frac{\frac{1}{2} - 2e^{i\theta} + \frac{3}{2}e^{2i\theta}}{e^{2i\theta}}.}$$

BDF2 is A-stable. It is the highest-order BDF method with full A-stability.

6.6.4.2 BDF3 and Sectorial Stability

BDF3 is

$$\frac{11z_{n+3} - 18z_{n+2} + 9z_{n+1} - 2z_n}{6h} = f_{n+3}.$$

Therefore

$$\rho(\xi) = -\frac{1}{3} + \frac{3}{2}\xi - 3\xi^2 + \frac{11}{6}\xi^3, \qquad \sigma(\xi) = \xi^3.$$

BDF3 is not A-stable: its region does not contain the entire left half-plane. Nevertheless it is **A(α)-stable**: a sector

$$\{q : |\arg(-q)| \le \alpha\}$$

about the negative real axis lies inside the stability region. For BDF3, $\alpha \approx 86.03^\circ$. This explains why higher-order BDF methods remain useful for many stiff dissipative problems despite the second Dahlquist barrier.

6.7 Stability Diagrams and Examples

6.7.1 Stability Plots

The following code generates the stability regions used for comparison. The plots are computed from the defining stability conditions rather than from hard-coded boundary curves.

```
# L'altezza conta: con asp = 1 R impone la stessa scala sui due assi, quindi
# se il riquadro e' piu' alto di quanto serve e' l'asse y a dilatarsi per
# riempirlo.  A 7x6, con tre riquadri affiancati, y arrivava a +/-10.24
# invece dei +/-4 della griglia.  A 7x3 viene +/-4.32.

R_EE  <- function(q) 1 + q
R_RK2 <- function(q) 1 + q + q^2/2
R_RK4 <- function(q) 1 + q + q^2/2 + q^3/6 + q^4/24

x <- seq(-4, 2, length.out = 600)
y <- seq(-4, 4, length.out = 600)
q <- outer(x, y, function(a, b) a + 1i*b)

plot_region <- function(R, main, xlim = range(x), ylim = range(y)) {
  z <- Mod(R(q))
  contour(x, y, z, levels = 1, drawlabels = FALSE,
          xlab = expression(Re(q)), ylab = expression(Im(q)),
          main = main, xlim = xlim, ylim = ylim, asp = 1)
  abline(h = 0, v = 0, lty = 3)
}

par(mfrow = c(1, 3))
plot_region(R_EE,  "Explicit Euler")
plot_region(R_RK2, "RK2")
plot_region(R_RK4, "Classical RK4")
par(mfrow = c(1, 1))
```

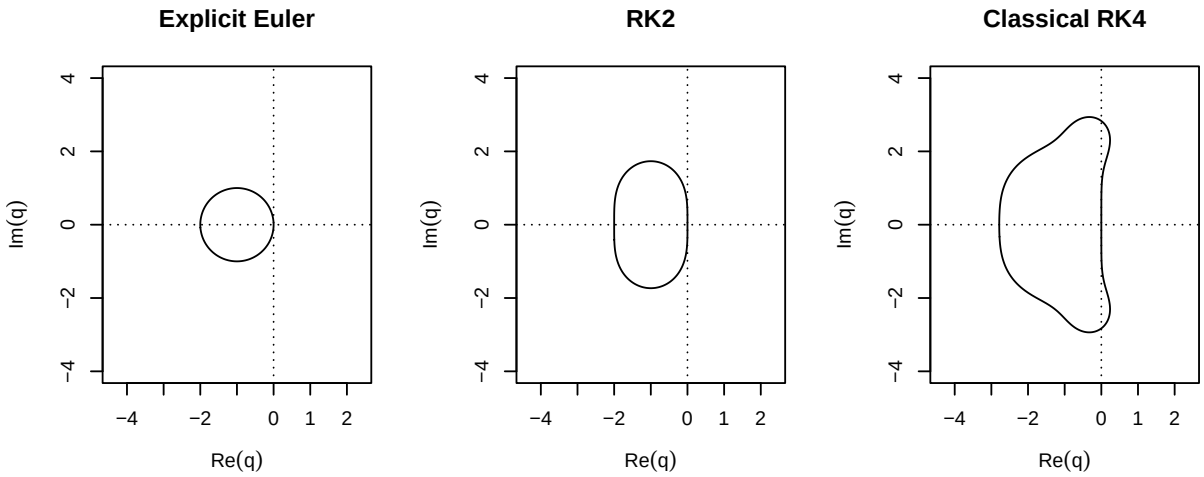


Figure 6.3: Absolute-stability regions of explicit Euler, RK2, and classical RK4.

The bounded regions make the limitation of explicit methods immediately visible. Increasing the order changes the shape and extent of the stability region, but does not produce A-stability.

```r
R_IE <- function(q) 1/(1-q)
R_TR <- function(q) (1+q/2)/(1-q/2)

x2 <- seq(-8, 4, length.out = 700)
y2 <- seq(-6, 6, length.out = 700)
q2 <- outer(x2, y2, function(a, b) a + 1i*b)

par(mfrow = c(1, 2))
contour(x2, y2, Mod(R_IE(q2)), levels = 1, drawlabels = FALSE,
        xlab = expression(Re(q)), ylab = expression(Im(q)),
        main = "Implicit Euler", asp = 1)
abline(h = 0, v = 0, lty = 3)
contour(x2, y2, Mod(R_TR(q2)), levels = 1, drawlabels = FALSE,
        xlab = expression(Re(q)), ylab = expression(Im(q)),
        main = "Trapezoidal rule", asp = 1)
abline(h = 0, v = 0, lty = 3)
par(mfrow = c(1, 1))
```

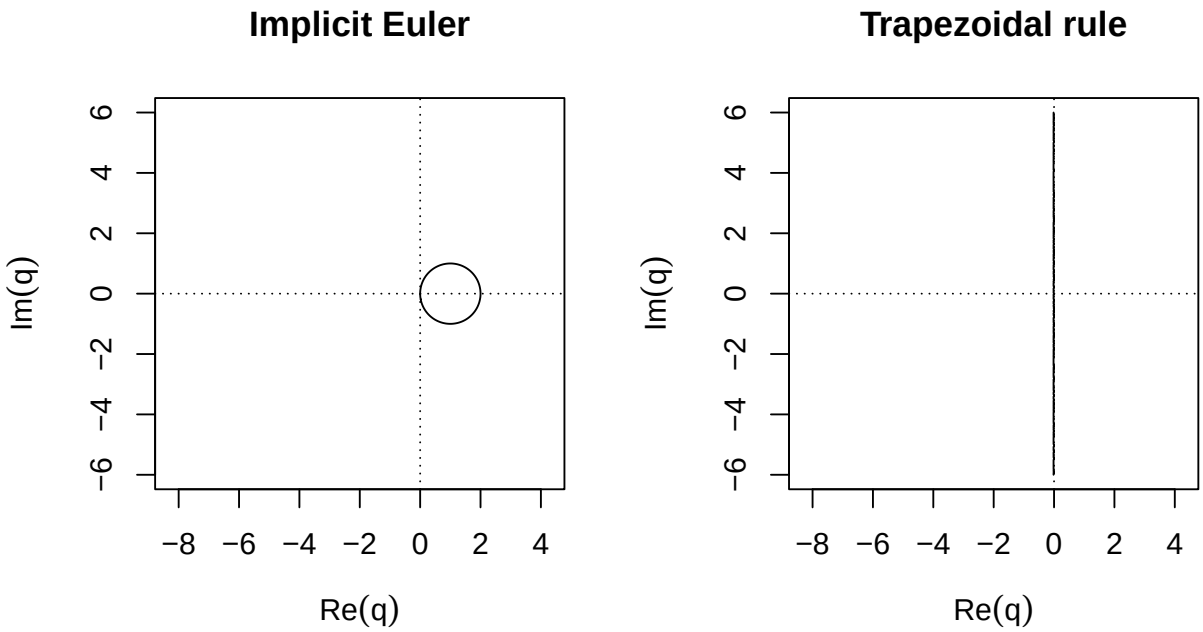


Figure 6.4: Absolute-stability boundaries of implicit Euler and the trapezoidal rule. Both methods contain the left half-plane; only implicit Euler is L-stable.

For implicit Euler the boundary is the circle $|1 - q| = 1$, but the **stable side is the exterior** of that circle. For the trapezoidal rule the boundary is the imaginary axis and the stable side is the entire left half-plane.

6.7.1.1 BDF Boundary Loci

```r
theta <- seq(0, 2*pi, length.out = 2000)
xi <- exp(1i*theta)

rho2 <- 1/2 - 2*xi + (3/2)*xi^2
sig2 <- xi^2
qbdf2 <- rho2/sig2
```

```
rho3 <- -1/3 + (3/2)*xi - 3*xi^2 + (11/6)*xi^3
sig3 <- xi^3
qbdf3 <- rho3/sig3

plot(Re(qbdf2), Im(qbdf2), type = "l", asp = 1,
     xlab = expression(Re(q)), ylab = expression(Im(q)),
     xlim = c(-2, 8), ylim = c(-5, 5),
     main = "BDF boundary loci")
lines(Re(qbdf3), Im(qbdf3), lty = 2)
abline(h = 0, v = 0, lty = 3)
legend("topright", legend = c("BDF2", "BDF3"), lty = c(1, 2), bty = "n")
```

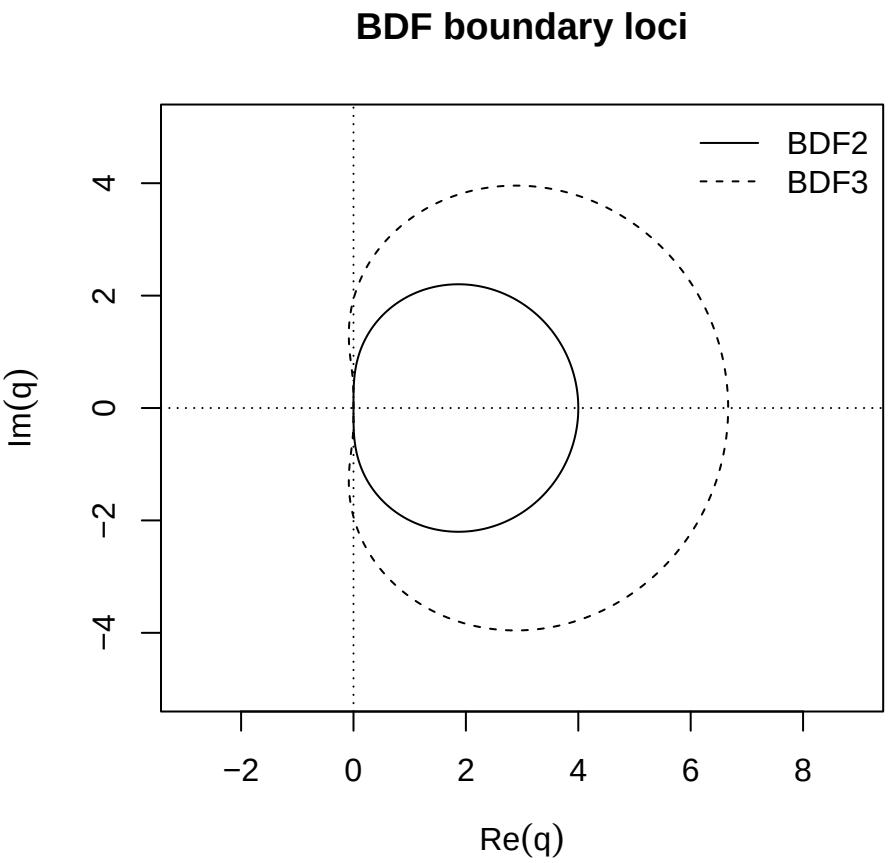


Figure 6.5: Boundary loci for BDF2 and BDF3. The stable component is identified using the root condition.

A boundary-locus plot should always be read together with the root condition. In particular, the stability region of BDF methods is the unbounded component on the left side of the relevant boundary, not the bounded area enclosed visually by a curve.

6.7.2 A Stiff Two-Mode Example

Consider

$$y' = Ay, \qquad A = \begin{bmatrix} -1 & 0 \\ 0 & -1000 \end{bmatrix}.$$

The exact solution contains a slow mode e^{-t} and a fast mode e^{-1000t}. For explicit Euler, stability of the fast mode requires

$$|-1{,}000h + 1| \le 1,$$

hence

$$\boxed{h \le 0.002.}$$

After the initial transient has decayed, the physically relevant solution may vary on a time scale of order one, yet explicit Euler must continue to take at least 500 steps per unit time solely because of stability.

Implicit Euler has no such stability restriction because every $q < 0$ lies in its stability region. Moreover, its L-stability rapidly removes the -1000 mode. The trapezoidal rule is A-stable but not L-stable; with a very large negative q its amplification factor approaches -1, so the fast mode can appear as a weakly damped numerical oscillation.

This example captures the computational meaning of stiffness: **the step size of an explicit method is dictated by stability rather than by the accuracy needed for the solution component of interest**.

6.8 Practical Guidance

6.8.1 Stability Versus Accuracy

A value q lying inside S only guarantees bounded numerical propagation for the test equation. It does not imply that the approximation is accurate. For example, implicit Euler is stable for arbitrarily large negative q, but

$$R_{\mathrm{IE}}(q) = \frac{1}{1-q}$$

need not accurately approximate e^q at a coarse step size.

A practical step size must therefore satisfy two distinct requirements:

1. **stability:** the relevant scaled eigenvalues $h\lambda_j$ must lie in the method's stability region;
2. **accuracy:** the local/global error must satisfy the requested tolerance.

For non-stiff problems accuracy usually determines h. For stiff problems an explicit method can instead be stability-limited.

6.8.2 Oscillatory Problems and the Imaginary Axis

For nearly conservative problems, eigenvalues may lie close to the imaginary axis. Then the behavior of $R(i\omega h)$ matters. Two errors should be distinguished:

- **dissipation error:** $|R(i\omega h)| \neq 1$;
- **dispersion or phase error:** $\arg R(i\omega h) \neq \omega h$.

A method can therefore be absolutely stable and still reproduce oscillations poorly. For long-time Hamiltonian or wave-like dynamics, geometric properties such as symplecticity may be more informative than absolute stability alone.

6.8.3 Practical Interpretation of Stability Diagrams

When selecting a method or a step size, the following procedure is useful.

1. Linearize the ODE locally when appropriate:

$$\boldsymbol{y}' \approx \boldsymbol{J}_f(t, \boldsymbol{y})\boldsymbol{y} + \cdots.$$

2. Estimate the relevant eigenvalues λ_j of $\boldsymbol{J}_f$.
3. Scale them by the candidate step size: $q_j = h\lambda_j$.
4. Check whether the q_j lie in the stability region.
5. Independently enforce the accuracy tolerance.

For strongly non-normal problems this eigenvalue test is only a first diagnostic, not a complete guarantee.

6.8.4 Comparison of Representative Methods

Method	Order	Type	$R(q)$	Stability on negative real axis	A-stable	L-stable
Explicit Euler	1	explicit RK	$1 + q$	$[-2, 0]$	no	no
Explicit RK2 (Heun, midpoint, Ralston)	2	explicit RK	$1 + q + \frac{q^2}{2}$	$[-2, 0]$	no	no
Classical RK4	4	explicit RK	$\sum_{k=0}^{4} \frac{q^k}{k!}$	approximately $[-2.7853, 0]$	no	no
Implicit Euler	1	implicit RK	$\frac{1}{1-q}$	$(-\infty, 0]$	yes	yes
Trapezoidal rule (Crank–Nicolson)	2	implicit RK	$\frac{1+q/2}{1-q/2}$	$(-\infty, 0]$	yes	no
Implicit midpoint	2	implicit RK	$\frac{1+q/2}{1-q/2}$	$(-\infty, 0]$	yes	no
Gauss–Legendre, s stages	$2s$	implicit RK	Padé (s, s)	$(-\infty, 0]$	yes	no
Radau IIA, s stages	$2s-1$	implicit RK	Padé $(s-1, s)$	$(-\infty, 0]$	yes	yes
BDF2	2	implicit LMM	–	$(-\infty, 0]$	yes	–
BDF3	3	implicit LMM	–	$(-\infty, 0]$ and a left sector	no	–

For multistep methods, the one-step definition of L-stability through a scalar function $R(q)$ does not apply directly; their stiff-decay behavior is instead analyzed through the roots of $\pi(\xi; q)$ and related notions.

6.9 Geometric Integration and Invariant Preservation

Absolute stability and local error do not completely describe long-time dynamics. A **geometric integrator** is designed to preserve a structural property of the differential equation—typically symplectic form, momentum, reversibility, positivity, or a first integral—exactly or with a controlled long-time error.

For a canonical Hamiltonian system

$$\dot{\boldsymbol{q}} = \nabla_{\boldsymbol{p}} H(\boldsymbol{q}, \boldsymbol{p}), \qquad \dot{\boldsymbol{p}} = -\nabla_{\boldsymbol{q}} H(\boldsymbol{q}, \boldsymbol{p}),$$

the exact flow is symplectic. A generic high-order method can give a small short-time error while producing systematic energy drift over a long interval. A symplectic method instead preserves a nearby modified Hamiltonian; its energy error is commonly bounded and oscillatory over long times.

6.9.1 Symplectic Euler and Störmer–Verlet

For a separable Hamiltonian $H(\boldsymbol{q}, \boldsymbol{p}) = T(\boldsymbol{p}) + V(\boldsymbol{q})$, one version of symplectic Euler is

$$\begin{aligned} \boldsymbol{p}_{n+1} &= \boldsymbol{p}_n - h\nabla V(\boldsymbol{q}_n), \\ \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + h\nabla T(\boldsymbol{p}_{n+1}). \end{aligned}$$

It is first order but symplectic. The symmetric second-order Störmer–Verlet method is

$$\begin{aligned} \boldsymbol{p}_{n+1/2} &= \boldsymbol{p}_n - \frac{h}{2}\nabla V(\boldsymbol{q}_n), \\ \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + h\nabla T(\boldsymbol{p}_{n+1/2}), \\ \boldsymbol{p}_{n+1} &= \boldsymbol{p}_{n+1/2} - \frac{h}{2}\nabla V(\boldsymbol{q}_{n+1}). \end{aligned}$$

Its symmetry makes the method time reversible and is central to its favorable long-time behavior.

6.9.2 Constraints and invariants

For holonomic constraints $\Phi(\boldsymbol{q}) = \boldsymbol{0}$, SHAKE and RATTLE augment a Verlet-type step with Lagrange multipliers so that position constraints, and for RATTLE also velocity constraints, hold at the accepted state. This links geometric ODE integration to the projection and coordinate methods used for DAEs in Section 8.10.

Not every invariant should be enforced by an arbitrary projection. Projection can destroy symplecticity, and exact energy conservation can distort phase. The property to preserve must follow from the qualitative behavior that matters for the model.

6.9.3 A practical long-time test

When comparing methods on a conservative problem, report

1. state error over a fixed short interval;
2. energy and momentum errors over a long interval;
3. constraint residuals when constraints are present;
4. phase or period error;
5. the same quantities under step refinement.

This separates high local accuracy from faithful qualitative dynamics.

Summary

Absolute stability explains why methods of the same order can behave very differently on stiff problems. The principal points are:

- explicit Runge–Kutta stability regions are bounded;
- A-stable methods contain the entire left half-plane;
- L-stability adds strong damping of very stiff stable modes;
- the trapezoidal rule is A-stable but not L-stable;
- BDF2 is A-stable, whereas BDF3–BDF6 retain only sectorial stability;
- for multistep methods stability is governed by **all** characteristic roots;
- stability and accuracy are independent requirements;
- eigenvalue-based stability diagrams are powerful diagnostics but are not a complete theory for nonlinear or strongly non-normal systems;
- geometric structure and invariants can dominate long-time behavior even when classical stability requirements are satisfied.

These ideas connect the qualitative theory of ODEs, the Runge–Kutta framework, and the zero-stability/convergence theory of linear multistep methods developed in the preceding chapters.

7. Checking the Order of a RK Method

The order conditions collected in 2 are usually quoted and rarely used. This chapter does the opposite: it applies them, one tableau at a time, to the methods introduced in the previous chapters. The point of the exercise is twofold — to make the conditions concrete, and to show that determining the order of a method is a finite, purely algebraic computation that needs no Taylor expansion and no numerical experiment.

7.1 The Conditions in Vector Form

The conditions themselves, their derivation and the rooted-tree rule behind them are in 2. This section only recalls the notation and then puts it to work.

Let the method have s stages, with Butcher tableau

$$\begin{array}{c|c} \boldsymbol{c} & \boldsymbol{A} \\ \hline & \boldsymbol{b}^T \end{array}$$

and let $\mathbf{1} = (1, \dots, 1)^T \in \mathbb{R}^s$. With $\circ$ the componentwise (Hadamard) product, write $\boldsymbol{c}^{\circ 2} = \boldsymbol{c} \circ \boldsymbol{c}$, $\boldsymbol{c}^{\circ 3} = \boldsymbol{c} \circ \boldsymbol{c} \circ \boldsymbol{c}$, and introduce the two auxiliary vectors of Equation 4.8,

$$\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c}, \qquad \boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b}.$$

The eight conditions through order four are then those of Equation 4.9, preceded by the stage-consistency relation $\boldsymbol{A}\mathbf{1} = \boldsymbol{c}$ of Equation 4.7:

order	new conditions
0	$\boldsymbol{A}\mathbf{1} = \boldsymbol{c}$ (stage consistency)
1	$\boldsymbol{b}\cdot\mathbf{1} = 1$
2	$\boldsymbol{b}\cdot\boldsymbol{c} = \frac{1}{2}$
3	$\boldsymbol{b}\cdot\boldsymbol{c}^{\circ 2} = \frac{1}{3}$, $\quad \boldsymbol{b}\cdot\boldsymbol{d} = \frac{1}{6}$
4	$\boldsymbol{b}\cdot\boldsymbol{c}^{\circ 3} = \frac{1}{4}$, $\quad (\boldsymbol{b}\circ\boldsymbol{c})\cdot\boldsymbol{d} = \frac{1}{8}$, $\quad \boldsymbol{e}\cdot\boldsymbol{c}^{\circ 2} = \frac{1}{12}$, $\quad \boldsymbol{e}\cdot\boldsymbol{d} = \frac{1}{24}$

The conditions are **cumulative**: a method has order p when all the conditions listed for orders $1, \dots, p$ hold and at least one condition of order $p+1$ fails. The row labelled 0 is not an order condition by itself, but every formula below presupposes it and it is worth verifying first because a tableau that violates it has been transcribed incorrectly.

> **How to organise the computation by hand**
>
> Build one table per method, adding rows as needed:
>
> 1. write $\boldsymbol{b}$ and check that its entries sum to 1;
> 2. write $\boldsymbol{c}$ (the row sums of $\boldsymbol{A}$), then $\boldsymbol{b}\circ\boldsymbol{c}$ and sum;
> 3. form $\boldsymbol{c}^{\circ 2}$, $\boldsymbol{c}^{\circ 3}$ by squaring and cubing entries;
> 4. form $\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c}$ and $\boldsymbol{e} = \boldsymbol{A}^{T}\boldsymbol{b}$ once, and reuse them.
>
> Stop as soon as a condition fails: the order is then one less than the level of the failing condition. Everything is rational arithmetic, so the checks are exact.

7.2 Worked Verifications

The nine tableaux below are checked one by one with the conditions in vector form. Each verification is self-contained: read the one you need.

7.2.1 Explicit Euler

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \boldsymbol{A} = (0), \quad \boldsymbol{b} = (1), \quad \boldsymbol{c} = (0).$$

Stage consistency: $\boldsymbol{A}\mathbf{1} = (0) = \boldsymbol{c}$. Then

condition	value	
$\boldsymbol{b}\cdot\mathbf{1} = 1$	1	OK
$\boldsymbol{b}\cdot\boldsymbol{c} = \frac{1}{2}$	0	fails

Order 1. The failure is immediate: a single stage evaluated at the left endpoint cannot see the curvature of the solution.

7.2.2 Explicit Midpoint

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array} \qquad \boldsymbol{c} = \begin{pmatrix} 0 \\ 1/2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

$\boldsymbol{b}$	0	1	1	OK
$\boldsymbol{c}$	0	1/2		
$\boldsymbol{b} \circ \boldsymbol{c}$	0	1/2	1/2	OK
$\boldsymbol{c}^{\circ 2}$	0	1/4		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 2}$	0	1/4	1/4	fails ($\neq 1/3$)

Order 2. Once a third-order condition fails there is no need to test $\boldsymbol{b} \cdot \boldsymbol{d} = \frac{1}{6}$: the order is already determined.

7.2.3 Ralston's Second-Order Method

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 2/3 & 2/3 & 0 \\ \hline & 1/4 & 3/4 \end{array} \qquad \boldsymbol{c} = \begin{pmatrix} 0 \\ 2/3 \end{pmatrix}, \quad \boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

$\boldsymbol{b}$	1/4	3/4	1	OK
$\boldsymbol{c}$	0	2/3		
$\boldsymbol{b} \circ \boldsymbol{c}$	0	1/2	1/2	OK
$\boldsymbol{c}^{\circ 2}$	0	4/9		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 2}$	0	1/3	1/3	OK
$\boldsymbol{d}$	0	0		
$\boldsymbol{b} \circ \boldsymbol{d}$	0	0	0	fails ($\neq 1/6$)

Order 2. This example is instructive: Ralston's method satisfies $\boldsymbol{b} \cdot \boldsymbol{c}^{\circ 2} = \frac{1}{3}$, one of the two third-order conditions, and fails the other. Satisfying *some* conditions of a given level buys nothing; the choice $c_2 = 2/3$ minimises the leading error constant among second-order two-stage methods, but it does not raise the order.

7.2.4 Third-Order Runge–Kutta (RK3)

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1 & -1 & 2 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array} \qquad \boldsymbol{c} = \begin{pmatrix} 0 \\ 1/2 \\ 1 \end{pmatrix}, \quad \boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

$\boldsymbol{b}$	1/6	2/3	1/6	1	OK
$\boldsymbol{c}$	0	1/2	1		
$\boldsymbol{b} \circ \boldsymbol{c}$	0	1/3	1/6	1/2	OK
$\boldsymbol{c}^{\circ 2}$	0	1/4	1		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 2}$	0	1/6	1/6	1/3	OK
$\boldsymbol{d}$	0	0	1		
$\boldsymbol{b} \circ \boldsymbol{d}$	0	0	1/6	1/6	OK
$\boldsymbol{c}^{\circ 3}$	0	1/8	1		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 3}$	0	1/12	1/6	1/4	OK
$(\boldsymbol{b} \circ \boldsymbol{c}) \circ \boldsymbol{d}$	0	0	1/6	1/6	fails ($\neq 1/8$)

Order 3. Here the method passes the first fourth-order condition $\boldsymbol{b} \cdot \boldsymbol{c}^{\circ 3} = \frac{1}{4}$ and fails the second. Testing the conditions in the order listed in Equation 4.9 is therefore not an early-exit strategy: all conditions of a level must be checked before the level can be declared satisfied.

7.2.5 Fourth-Order Runge–Kutta: the 3/8 Rule

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/3 & 0 & 0 & 0 \\ 2/3 & -1/3 & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & 1/8 & 3/8 & 3/8 & 1/8 \end{array}$$

Stage consistency gives $\boldsymbol{c} = (0, \frac{1}{3}, \frac{2}{3}, 1)^T$, and

$$\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} 0 \\ 0 \\ 1/3 \\ 1/3 \end{pmatrix}, \qquad \boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b} = \begin{pmatrix} 1/8 \\ 1/4 \\ 1/8 \\ 0 \end{pmatrix}.$$

$\boldsymbol{b}$	1/8	3/8	3/8	1/8	1	OK
$\boldsymbol{c}$	0	1/3	2/3	1		
$\boldsymbol{b} \circ \boldsymbol{c}$	0	1/8	1/4	1/8	1/2	OK
$\boldsymbol{c}^{\circ 2}$	0	1/9	4/9	1		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 2}$	0	1/24	1/6	1/8	1/3	OK
$\boldsymbol{d}$	0	0	1/3	1/3		
$\boldsymbol{b} \circ \boldsymbol{d}$	0	0	1/8	1/24	1/6	OK
$\boldsymbol{c}^{\circ 3}$	0	1/27	8/27	1		
$\boldsymbol{b} \circ \boldsymbol{c}^{\circ 3}$	0	1/72	1/9	1/8	1/4	OK
$(\boldsymbol{b} \circ \boldsymbol{c}) \circ \boldsymbol{d}$	0	0	1/12	1/24	1/8	OK
$\boldsymbol{e}$	1/8	1/4	1/8	0		
$\boldsymbol{e} \circ \boldsymbol{c}^{\circ 2}$	0	1/36	1/18	0	1/12	OK
$\boldsymbol{e} \circ \boldsymbol{d}$	0	0	1/24	0	1/24	OK

Order at least 4. All eight conditions hold. Establishing that the order is exactly four would require the nine fifth-order conditions Equation 4.10; for a four-stage explicit method this is unnecessary, since the attainable order of an explicit s-stage method is at most s (Table 4.3).

7.2.6 Implicit Midpoint

$$\begin{array}{c|c} 1/2 & 1/2 \\ \hline & 1 \end{array} \qquad \boldsymbol{A} = (1/2), \quad \boldsymbol{b} = (1), \quad \boldsymbol{c} = (1/2), \quad \boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = (1/4).$$

condition	value	
$\boldsymbol{b}\cdot\mathbf{1} = 1$	1	OK
$\boldsymbol{b}\cdot\boldsymbol{c} = \frac{1}{2}$	$1/2$	OK
$\boldsymbol{b}\cdot\boldsymbol{c}^{\circ 2} = \frac{1}{3}$	$1/4$	fails
$\boldsymbol{b}\cdot\boldsymbol{d} = \frac{1}{6}$	$1/4$	fails

Order 2. A single implicit stage already attains the maximum order $2s = 2$ of a one-stage method; no one-stage method of order three exists.

7.2.7 Crank–Nicolson (Trapezoidal Rule)

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \qquad \boldsymbol{c} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} 0 \\ 1/2 \end{pmatrix}.$$

$\boldsymbol{b}$	$1/2$	$1/2$	1	OK
$\boldsymbol{c}$	0	1		
$\boldsymbol{b}\circ\boldsymbol{c}$	0	$1/2$	$1/2$	OK
$\boldsymbol{c}^{\circ 2}$	0	1		
$\boldsymbol{b}\circ\boldsymbol{c}^{\circ 2}$	0	$1/2$	$1/2$	fails ($\neq 1/3$)

Order 2. The trapezoidal rule is only a diagonally implicit two-stage method; its interest lies entirely in its stability (A-stable, but not L-stable), not in its order.

7.2.8 Gauss–Legendre of Order Four

$$\begin{array}{c|cc} \frac{1}{2}-\frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4}-\frac{\sqrt{3}}{6} \\ \frac{1}{2}+\frac{\sqrt{3}}{6} & \frac{1}{4}+\frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Writing $\gamma = \sqrt{3}/6$ so that $\boldsymbol{c} = (\frac{1}{2}-\gamma,\ \frac{1}{2}+\gamma)^T$,

$$\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} \frac{1}{6}-\frac{\sqrt{3}}{12} \\ \frac{1}{6}+\frac{\sqrt{3}}{12} \end{pmatrix}, \qquad \boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b} = \begin{pmatrix} \frac{1}{4}+\frac{\sqrt{3}}{12} \\ \frac{1}{4}-\frac{\sqrt{3}}{12} \end{pmatrix}.$$

The checks use repeatedly $c_1 + c_2 = 1$ and $c_1 c_2 = \frac{1}{4} - \gamma^2 = \frac{1}{6}$:

condition	value	
$\boldsymbol{b}\cdot\mathbf{1}=1$	$\frac{1}{2}+\frac{1}{2}=1$	OK
$\boldsymbol{b}\cdot\boldsymbol{c}=\frac{1}{2}$	$\frac{1}{2}(c_1+c_2)=\frac{1}{2}$	OK
$\boldsymbol{b}\cdot\boldsymbol{c}^{\circ 2}=\frac{1}{3}$	$\frac{1}{2}\big((c_1+c_2)^2-2c_1c_2\big)=\frac{1}{2}\big(1-\frac{1}{3}\big)=\frac{1}{3}$	OK
$\boldsymbol{b}\cdot\boldsymbol{d}=\frac{1}{6}$	$\frac{1}{2}\cdot\frac{1}{3}=\frac{1}{6}$	OK
$\boldsymbol{b}\cdot\boldsymbol{c}^{\circ 3}=\frac{1}{4}$	$\frac{1}{4}$	OK
$(\boldsymbol{b}\circ\boldsymbol{c})\cdot\boldsymbol{d}=\frac{1}{8}$	$\frac{1}{8}$	OK
$\boldsymbol{e}\cdot\boldsymbol{c}^{\circ 2}=\frac{1}{12}$	$\frac{1}{12}$	OK
$\boldsymbol{e}\cdot\boldsymbol{d}=\frac{1}{24}$	$\frac{1}{24}$	OK

Order at least 4 — and in fact exactly $4 = 2s$, the maximum attainable by an s-stage implicit method, which is the defining property of the Gauss–Legendre family. The surds cancel in every condition, as they must: the conditions are symmetric functions of c_1, c_2, whose elementary symmetric polynomials $c_1 + c_2 = 1$ and $c_1c_2 = \frac{1}{6}$ are rational.

7.2.9 Lobatto IIIB of Order Four

$$\begin{array}{c|ccc} 0 & 1/6 & -1/6 & 0 \\ 1/2 & 1/6 & 1/3 & 0 \\ 1 & 1/6 & 5/6 & 0 \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

The row sums are $0, \frac{1}{2}, 1$, so stage consistency holds, and

$$\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c} = \begin{pmatrix} -1/12 \\ 1/6 \\ 5/12 \end{pmatrix}, \qquad \boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b} = \begin{pmatrix} 1/6 \\ 1/3 \\ 0 \end{pmatrix}.$$

$\boldsymbol{b}$	1/6	2/3	1/6	1	OK
$\boldsymbol{c}$	0	1/2	1		
$\boldsymbol{b}\circ\boldsymbol{c}$	0	1/3	1/6	1/2	OK
$\boldsymbol{c}^{\circ 2}$	0	1/4	1		
$\boldsymbol{b}\circ\boldsymbol{c}^{\circ 2}$	0	1/6	1/6	1/3	OK
$\boldsymbol{d}$	−1/12	1/6	5/12		
$\boldsymbol{b}\circ\boldsymbol{d}$	−1/72	1/9	5/72	1/6	OK
$\boldsymbol{c}^{\circ 3}$	0	1/8	1		
$\boldsymbol{b}\circ\boldsymbol{c}^{\circ 3}$	0	1/12	1/6	1/4	OK
$(\boldsymbol{b}\circ\boldsymbol{c})\circ\boldsymbol{d}$	0	1/18	5/72	1/8	OK
$\boldsymbol{e}$	1/6	1/3	0		
$\boldsymbol{e}\circ\boldsymbol{c}^{\circ 2}$	0	1/12	0	1/12	OK
$\boldsymbol{e}\circ\boldsymbol{d}$	−1/72	1/18	0	1/24	OK

Order at least 4. Two features are worth noticing. The last column of $\boldsymbol{A}$ vanishes, so the third stage does not feed back into the others: the method is not stiffly accurate in the

IIIA sense. And $\boldsymbol{d}$ has a negative entry, a reminder that the order conditions are algebraic identities with no sign structure to exploit.

7.3 Automated Verification

Checking a tableau by hand is useful once; after that it should be automated. The conditions of Equation 4.9 are a handful of inner products, so a few lines suffice to determine the order of any tableau — including the ones just examined, which serve as a regression test of the computations above.

```
rk_order <- function(A, b, tol = 1e-12) {
  A <- as.matrix(A); b <- as.numeric(b)
  s <- nrow(A); one <- rep(1, s)
  cc <- as.numeric(A %*% one)          # stage consistency defines c
  d  <- as.numeric(A %*% cc)
  e  <- as.numeric(t(A) %*% b)
  ok <- function(x, target) abs(x - target) < tol
  lev <- list(
    `1` = c(sum(b) - 1),
    `2` = c(sum(b * cc) - 1/2),
    `3` = c(sum(b * cc^2) - 1/3, sum(b * d) - 1/6),
    `4` = c(sum(b * cc^3) - 1/4, sum(b * cc * d) - 1/8,
            sum(e * cc^2) - 1/12, sum(e * d) - 1/24)
  )
  p <- 0
  for (k in names(lev)) {
    if (all(abs(lev[[k]]) < tol)) p <- as.integer(k) else break
  }
  p
}

g <- sqrt(3) / 6
tableaux <- list(
  "Explicit Euler"     = list(A = matrix(0, 1, 1), b = 1),
  "Explicit midpoint"  = list(A = rbind(c(0,0), c(1/2,0)), b = c(0,1)),
  "Ralston 2"          = list(A = rbind(c(0,0), c(2/3,0)), b = c(1/4,3/4)),
  "RK3"                = list(A = rbind(c(0,0,0), c(1/2,0,0), c(-1,2,0)),
                              b = c(1/6,2/3,1/6)),
  "RK4 (classical)"    = list(A = rbind(c(0,0,0,0), c(1/2,0,0,0),
                                        c(0,1/2,0,0), c(0,0,1,0)),
                              b = c(1/6,1/3,1/3,1/6)),
  "RK4 (3/8 rule)"     = list(A = rbind(c(0,0,0,0), c(1/3,0,0,0),
                                        c(-1/3,1,0,0), c(1,-1,1,0)),
                              b = c(1/8,3/8,3/8,1/8)),
  "Implicit midpoint"  = list(A = matrix(1/2, 1, 1), b = 1),
  "Crank--Nicolson"    = list(A = rbind(c(0,0), c(1/2,1/2)), b = c(1/2,1/2)),
  "Gauss--Legendre 4"  = list(A = rbind(c(1/4, 1/4 - g), c(1/4 + g, 1/4)),
                              b = c(1/2,1/2)),
  "Lobatto IIIA 4"     = list(A = rbind(c(0,0,0), c(5/24,1/3,-1/24),
                                        c(1/6,2/3,1/6)),
                              b = c(1/6,2/3,1/6)),
```

```
  "Lobatto IIIB 4"     = list(A = rbind(c(1/6,-1/6,0), c(1/6,1/3,0),
                                        c(1/6,5/6,0)),
                              b = c(1/6,2/3,1/6))
)

res <- data.frame(
  Method = names(tableaux),
  Stages = vapply(tableaux, function(m) nrow(as.matrix(m$A)), integer(1)),
  Order  = vapply(tableaux, function(m) rk_order(m$A, m$b), integer(1)),
  row.names = NULL
)
knitr::kable(res, align = "lrr",
             caption = "Order determined from the conditions of @eq-order-conditions-table (capped at 4).")
```

Table 7.1: Order determined from the conditions of Equation 4.9 (capped at 4).

Method	Stages	Order
Explicit Euler	1	1
Explicit midpoint	2	2
Ralston 2	2	2
RK3	3	3
RK4 (classical)	4	4
RK4 (3/8 rule)	4	4
Implicit midpoint	1	2
Crank–Nicolson	2	2
Gauss–Legendre 4	2	4
Lobatto IIIA 4	3	4
Lobatto IIIB 4	3	4

The column is capped at four because only the conditions up to that level have been implemented; for the two Lobatto methods and for Gauss–Legendre the true order is four, while a fifth-order test would be needed to confirm it rather than assume it.

What this check does and does not prove

The conditions verify the **order of consistency** of the tableau, that is, the rate at which the local truncation error vanishes. They say nothing about:

- **stability** — a high-order tableau may have a tiny absolute stability region and be useless on a stiff problem (compare the 3/8 rule with Gauss–Legendre);
- **the error constant** — two methods of the same order can differ by an order of magnitude in actual error, which is exactly what distinguishes Ralston's method from the explicit midpoint method;
- **implementation cost** — an implicit tableau of order four requires solving a nonlinear system of dimension sd at every step.

Order is a necessary property, never a sufficient criterion for choosing a method.

Exercises

1. Verify that the classical RK4 tableau satisfies all eight conditions of Equation 4.9, and identify which ones fail at order five.
2. Show that **no** explicit two-stage method can have order three, by writing the general tableau with $c_2 = \alpha$ and deriving a contradiction between $\boldsymbol{b} \cdot \boldsymbol{c}^{\circ 2} = \frac{1}{3}$ and $\boldsymbol{b} \cdot \boldsymbol{d} = \frac{1}{6}$.
3. Heun's third-order method has $\boldsymbol{c} = (0, \frac{1}{3}, \frac{2}{3})^T$ and $\boldsymbol{b} = (\frac{1}{4}, 0, \frac{3}{4})^T$. Determine $\boldsymbol{A}$ from the order conditions and confirm the order with `rk_order`.
4. For the Lobatto IIIA method of order four, verify that the last row of $\boldsymbol{A}$ equals $\boldsymbol{b}^T$ (the method is *stiffly accurate*) and explain why this implies $R(\infty) = 0$.
5. Modify `rk_order` to include the nine fifth-order conditions Equation 4.10 and recompute the table. Which of the methods listed actually attain order five?

Summary

- The order of a Runge–Kutta method follows from a finite set of algebraic identities among $\boldsymbol{A}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ — no Taylor expansion and no numerical experiment is required.
- Two auxiliary vectors, $\boldsymbol{d} = \boldsymbol{A}\boldsymbol{c}$ and $\boldsymbol{e} = \boldsymbol{A}^T\boldsymbol{b}$, express all the conditions through order four.
- The conditions are cumulative and must be checked level by level: satisfying some conditions of a level, as Ralston's method and RK3 do, does not raise the order.
- Explicit s-stage methods have order at most s, and from order five on they need strictly more stages than the order (Table 4.3); implicit s-stage methods reach at most $2s$, attained by the Gauss–Legendre family.

III DAE

8 Numerical Methods for DAEs 131

8.1 Consistent Initialization
8.2 Implicit Integration of Linear DAEs
8.3 Constraint Drift
8.4 Baumgarte Stabilization
8.5 Projection Methods
8.6 Projection Algorithms
8.7 Implicit Projection
8.8 Coordinate Partitioning
8.9 Order Reduction in DAE Integration
8.10 Choosing a Numerical DAE Method
8.11 Examples

Numerical integration of differential-algebraic equations: consistent initialization, constraint drift, stabilization, projection and coordinate partitioning. The theory is in the part of the same name of the course notes.

8. Numerical Methods for DAEs

This unit covers consistent initialization, direct and implicit integration, constraint drift, Baumgarte stabilization, projection, coordinate partitioning, order reduction, and method selection. Each topic is a section of this chapter.

8.1 Consistent Initialization

A DAE cannot in general be started from arbitrary values. For the implicit system

$$F(y, \dot{y}, t) = 0,$$

the pair $(y_0, \dot{y}_0)$ is **consistent** when it satisfies the original equations and every hidden constraint required by the index. An inconsistent start commonly produces a failed Newton iteration, a large initial transient, or constraint drift that is incorrectly attributed to the time integrator.

> Learning objectives
>
> After this chapter, the reader should be able to identify differential and algebraic unknowns, construct the consistency residual, compute a nearby consistent state, and verify the result before starting time integration.

8.1.1 A practical procedure

For a semi-explicit index-1 problem

$$\dot{x} = f(x, z, t), \qquad 0 = g(t, x, z),$$

the following workflow is robust.

1. **Choose the fixed data.** Decide which components of x_0 are prescribed by the physical experiment and which may be corrected.

2. **Solve the algebraic equations.** With the prescribed components fixed, solve $\boldsymbol{g}(t_0, \boldsymbol{x}_0, \boldsymbol{z}_0) = \boldsymbol{0}$ for the remaining state and algebraic variables.
3. **Recover derivatives.** Set $\dot{\boldsymbol{x}}_0 = \boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{z}_0, t_0)$.
4. **Add hidden constraints.** For index $q > 1$, differentiate the algebraic constraints only as far as required and append those relations to the nonlinear system. Alternatively, first apply an index-reduction method.
5. **Scale and solve.** Scale every residual by a characteristic magnitude and use a damped Newton or nonlinear least-squares method.
6. **Verify independently.** Report the norm of the original DAE residual, every hidden-constraint residual, and the size of the correction to the user-supplied data.

This order matters: merely satisfying the visible algebraic constraint does not guarantee that the derivative lies in its tangent space.

8.1.2 Projection to nearby consistent data

Suppose an approximate state $\widetilde{\boldsymbol{y}}_0$ is available and the assembled consistency conditions are $\boldsymbol{c}(\boldsymbol{y}_0) = \boldsymbol{0}$. A nearby consistent state can be defined by

$$\min_{\boldsymbol{y}_0} \frac{1}{2}\|\boldsymbol{W}(\boldsymbol{y}_0 - \widetilde{\boldsymbol{y}}_0)\|_2^2 \quad \text{subject to} \quad \boldsymbol{c}(\boldsymbol{y}_0) = \boldsymbol{0}, \tag{8.1}$$

where $\boldsymbol{W}$ protects accurately known components and accounts for units. One Newton–KKT correction solves

$$\begin{bmatrix} \boldsymbol{W}^T\boldsymbol{W} & \boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{y} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \boldsymbol{W}^T\boldsymbol{W}(\boldsymbol{y} - \widetilde{\boldsymbol{y}}_0) + \boldsymbol{C}^T\boldsymbol{\lambda} \\ \boldsymbol{c}(\boldsymbol{y}) \end{bmatrix},$$

with $\boldsymbol{C} = \partial\boldsymbol{c}/\partial\boldsymbol{y}$. Rank deficiency of $\boldsymbol{C}$ is a model diagnostic: redundant constraints should be removed or treated with a rank-revealing factorization.

8.1.3 Acceptance checklist

A computed initialization is acceptable only if

- scaled DAE and hidden-constraint residuals meet the requested tolerance;
- the constraint Jacobian has the expected numerical rank;
- the correction in Equation 8.1 is physically reasonable;
- one trial implicit step converges without an anomalous reduction of the step size.

The case studies in the companion volume (Case Studies) show how these checks change for mechanical, circuit, and high-index models.

8.2 Implicit Integration of Linear DAEs

Consider the DAE

$$x(t) = \sin(t)$$
$$x'(t) + y(t) = 0$$

with exact solution

$$x(t) = \sin(t)$$
$$y(t) = -\cos(t)$$

Is a linear DAE $\boldsymbol{F}\boldsymbol{x}'(t) + \boldsymbol{G}\boldsymbol{x}(t) = \boldsymbol{h}(t)$ where

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \sin(t) \\ 0 \end{pmatrix}$$

Using Kronecker/Weierstrass normal form with

$$\boldsymbol{U} = \boldsymbol{V} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{N} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

the DAE is written as

$$\boldsymbol{N} \begin{pmatrix} y' \\ x' \end{pmatrix} + \boldsymbol{I} \begin{pmatrix} y \\ x \end{pmatrix} = \begin{pmatrix} 0 \\ \sin(t) \end{pmatrix}$$

because $\boldsymbol{N} \neq \boldsymbol{0}$ and $\boldsymbol{N}^2 = \boldsymbol{0}$ the DAE is of index 2. No initial conditions are required because block $\boldsymbol{C}$ is absent.

8.2.1 Consistent Initial Values

Reconsider

$$x(t) = \sin(t)$$
$$x'(t) + y(t) = 0$$

I initial values x_0 and y_0 are consistent if

$$x(t) = \sin(t)$$

the hidden constraint is obtained by deriving the algebraic part

$$-y(t) = x'(t) = \cos(t)$$

and thus consistent initial condition must satisfy:

$$x_0 = \sin(0)$$
$$y_0 = -\cos(0)$$

8.2.2 Explicit Euler

Consider the ODE $\boldsymbol{x}' = \boldsymbol{f}(\boldsymbol{x}, t)$, explicit Euler take the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}(\boldsymbol{x}_k, t_k)$$

applied to the DAE become ($t_k = kh$)

$$0 = h(\sin(t_k) - x_k)$$
$$x_{k+1} = x_k - hy_k$$

Thus, the method cant be applied (y_{k+1} do not appear in the equations).

8.2.3 Implicit Euler

For $\boldsymbol{x}' = \boldsymbol{f}(\boldsymbol{x}, t)$, implicit Euler take the form

$$\boldsymbol{x}_{k+1} - h\boldsymbol{f}(\boldsymbol{x}_{k+1}, t_{k+1}) = \boldsymbol{x}_k$$

Applied to the DAE result in ($t_k = k\,h$)

$$h(\sin(t_{k+1}) - x_{k+1}) = 0$$
$$x_{k+1} + hy_{k+1} = x_k$$

so that

$$\begin{aligned} x_{k+1} &= \sin(t_{k+1}) \\ y_{k+1} &= (x_k - \sin(t_{k+1})/h \\ &= (\sin(t_k) - \sin(t_{k+1}))/h = -\cos(t_{k+1}) + \mathcal{O}(h) \end{aligned}$$

8.2.4 Implicit Euler Results

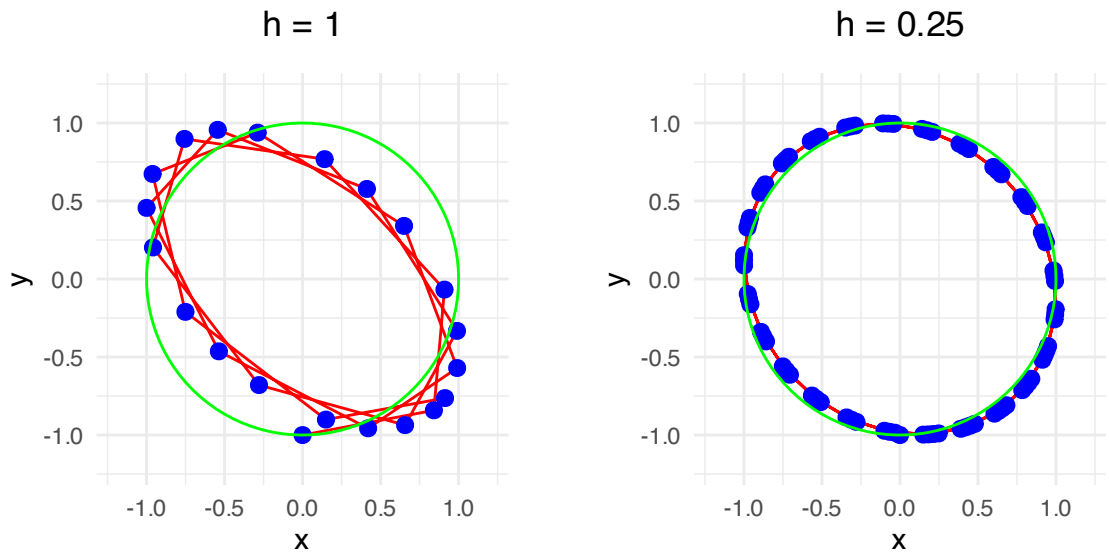


Figure 8.1: Implicit Euler. Consistent initial data.

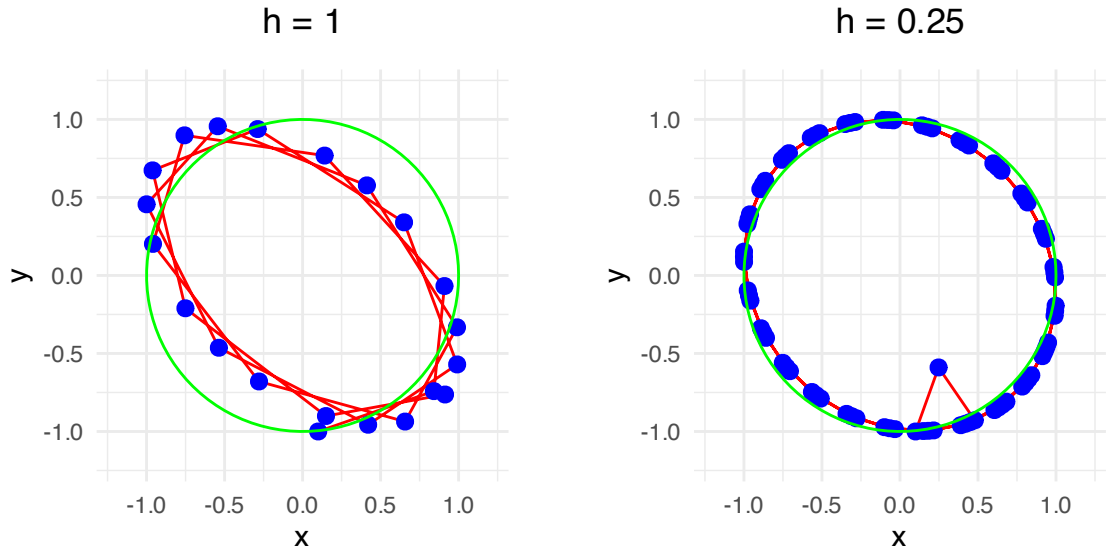


Figure 8.2: Implicit Euler. Inconsistent initial data.

8.2.4.1 Crank-Nicolson

$$\frac{x_{k+1} - x_k}{h} = \frac{f(x_{k+1}, t_{k+1}) + f(x_k, t_k)}{2}$$

Applied to the DAE result in

$$\sin(t_{k+1}) - x_{k+1} = x_k - \sin(t_k)$$

$$x_{k+1} - \frac{h}{2} y_{k+1} = x_k + \frac{h}{2} y_k$$

so that

$$x_{k+1} = \sin(t_{k+1}) + \sin(t_k) - x_k$$
$$y_{k+1} = \frac{2}{h}(x_k - x_{k+1}) - y_k$$

solving for x_k

$$x_k = \sin(t_k) + (-1)^k x_0$$

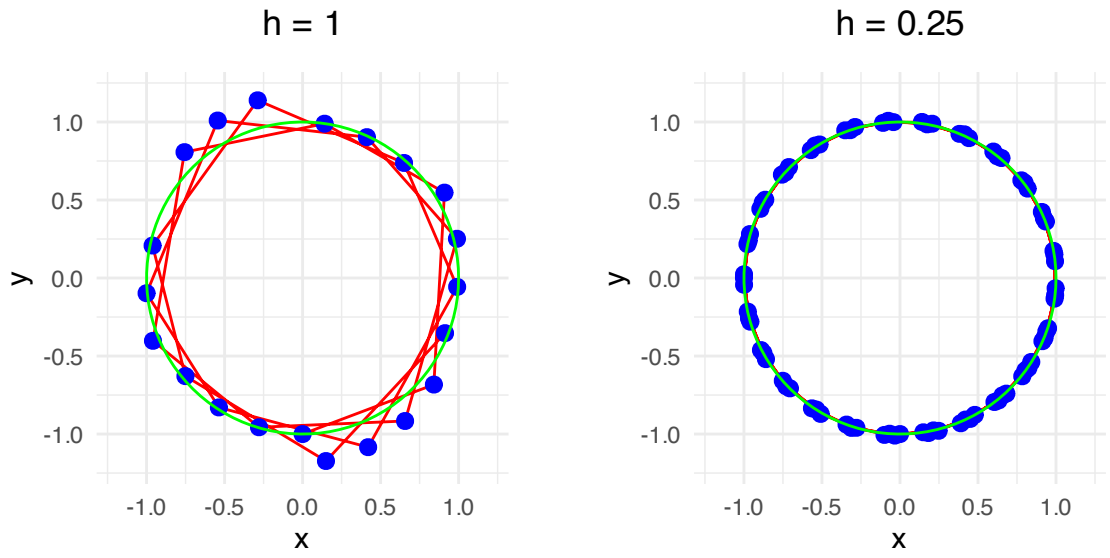


Figure 8.3: Consistent initial data $x(0) = -1$, $y(0) = 0$

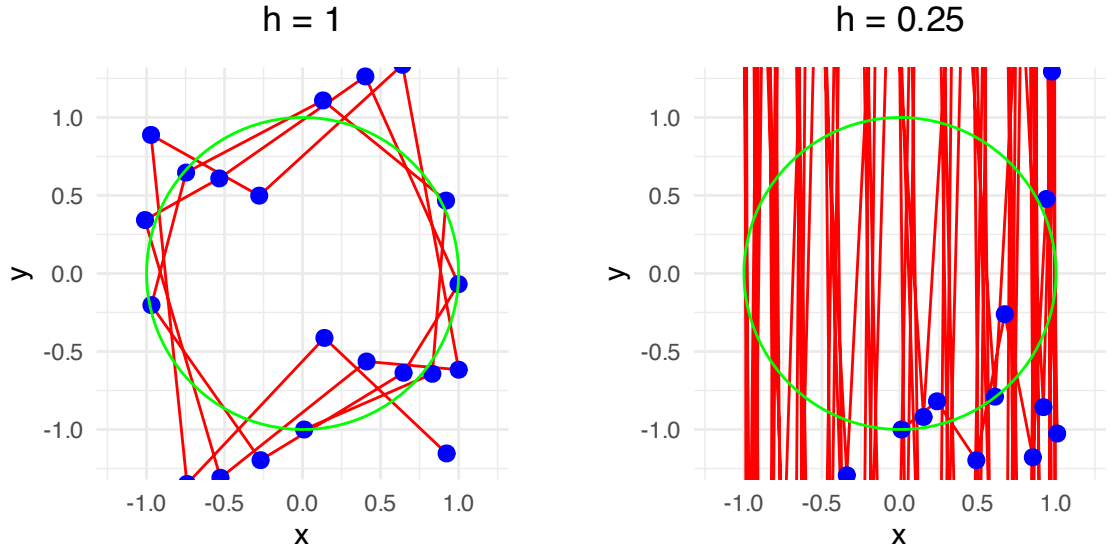


Figure 8.4: Inconsistent initial data $x(0) = -1$, $y(0) = 0.01$

8.2.5 Baumgarte Stabilization

Re-consider the DAE

$$x(t) = \sin(t)$$
$$x'(t) + y(t) = 0$$

stabilize the constraint $x(t) - \sin(t) = 0$ using Baumgarte

$$\tau(x(t) - \sin(t))' + (x(t) - \sin(t)) = 0$$
$$x'(t) + y(t) = 0$$

Observation

The ODE $\tau z'(t) + z(t) = 0$ has solution

$$z(t) = z(0)\exp\left(-\frac{t}{\tau}\right)$$

and $\lim_{t\to\infty} z(t) = 0$

Using Crank-Nicolson

$$\frac{\tau}{h}(x_{k+1} - x_k) - \frac{\tau}{h}(\sin(t_{k+1}) - \sin(t_k)) +$$
$$\frac{1}{2}(x_{k+1} + x_k) - \frac{1}{2}(\sin(t_{k+1}) + \sin(t_k)) = 0$$
$$\frac{x_{k+1} - x_k}{h} + \frac{y_{k+1} + y_k}{2} = 0$$

solving for x_{k+1} and y_{k+1}

$$x_{k+1} = \frac{2\,\tau - h}{2\,\tau + h}\left(x_k - \sin(t_k)\right) + \sin(t_{k+1})$$
$$y_{k+1} = \frac{\sin(t_k)(4\tau - 2h) - \sin(t_{k+1})(4\tau + 2h) + 4hx_k}{h(2\tau + h)} - y_k$$

8.2.6 Crank–Nicolson Results

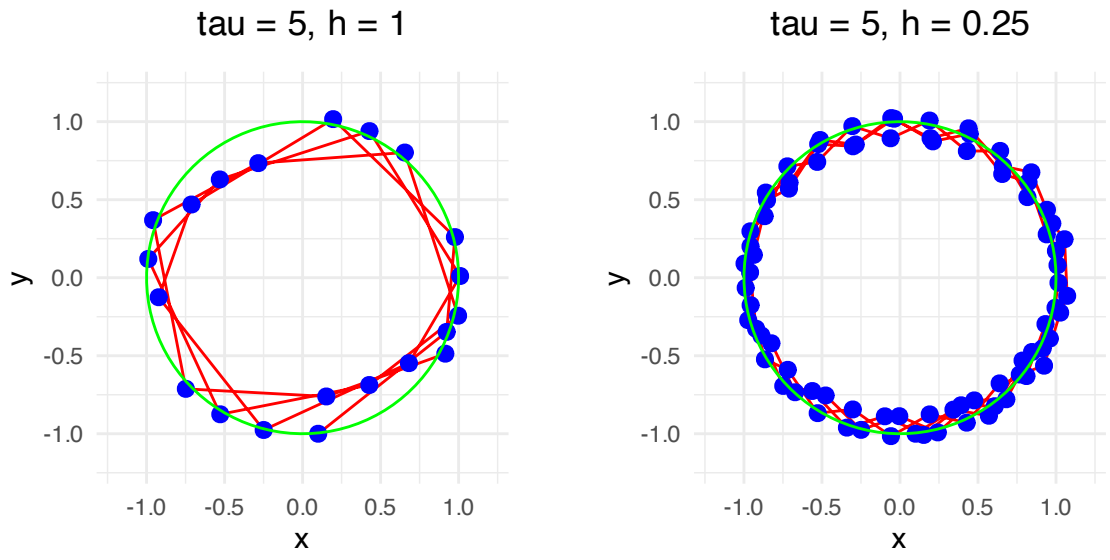


Figure 8.5: Inconsistent initial data $x(0) = -1$, $y(0) = 0.1$, $\tau = 5$

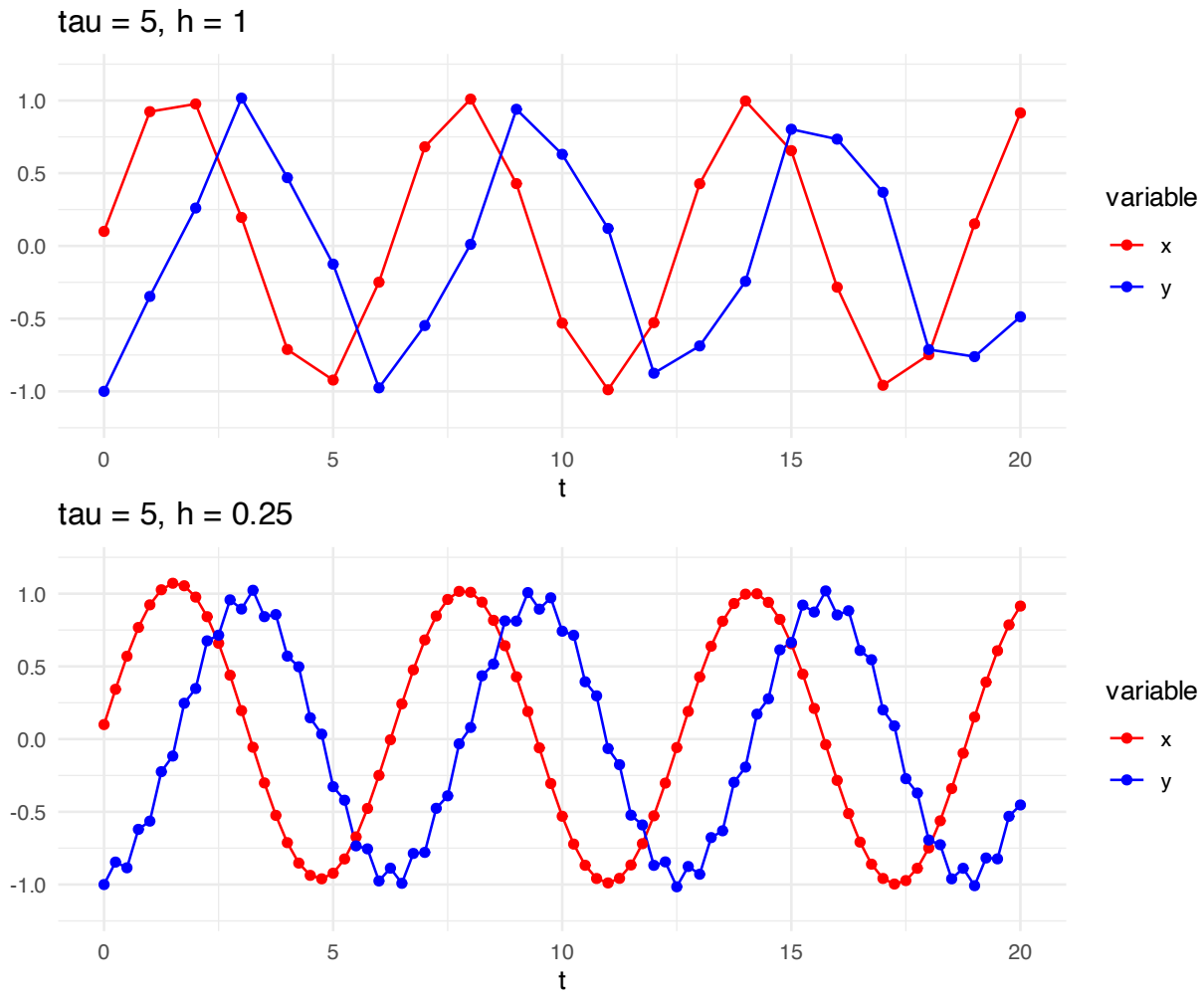


Figure 8.6: Inconsistent initial data $x(0) = -1$, $y(0) = 0.1$, $\tau = 5$

- Implicit Eulero seems to work also with inconsistent data.
- Crank-Nicolson do not work with inconsistent data
- Baumgarte stabilization work well but there is no general recipt to determine stabilization parameters (τ in this case).

8.3 Constraint Drift

Index reduction replaces some algebraic constraints by their time derivatives. The continuous solution still satisfies the original constraints, but a numerical trajectory generally does not. The resulting departure from the constraint manifold is called **constraint drift**.

For a holonomic mechanical constraint $\Phi(\boldsymbol{q}, t) = \mathbf{0}$, define

$$\boldsymbol{r}_p = \Phi(\boldsymbol{q}, t), \qquad \boldsymbol{r}_v = \Phi_{\boldsymbol{q}}\dot{\boldsymbol{q}} + \Phi_t.$$

An acceleration-level formulation enforces $\dot{\boldsymbol{r}}_v = \mathbf{0}$, not necessarily $\boldsymbol{r}_p = \mathbf{0}$ or $\boldsymbol{r}_v = \mathbf{0}$. Local truncation and nonlinear solver errors therefore act as forcing terms in the residual dynamics.

8.3.1 Why drift accumulates

If the numerical acceleration satisfies

$$\dot{\boldsymbol{r}}_v = \boldsymbol{\delta}(t),$$

then

$$\boldsymbol{r}_v(t) = \boldsymbol{r}_v(t_0) + \int_{t_0}^{t} \boldsymbol{\delta}(s)\,\mathrm{d}s,$$

and the position residual contains a second integration of the defect. Even small, nearly unbiased errors can therefore generate visible long-time drift. Reducing the step size may delay the effect but does not change this neutral residual dynamics.

8.3.2 What to monitor

Report at least four histories:

Quantity	Diagnostic role
$\|\boldsymbol{r}_p\|$	position-level feasibility
$\|\boldsymbol{r}_v\|$	tangency of the velocity
DAE residual	accuracy of the discrete nonlinear solve
energy or another invariant	distortion of the physical dynamics

These quantities distinguish drift from a loose Newton tolerance or an unstable time step.

8.3.3 Remedies

- **Baumgarte stabilization** assigns asymptotically stable dynamics to the residual; see Section 8.4.
- **Projection** returns the numerical state to the constraint manifold after a step; see Section 8.5.
- **Coordinate partitioning** removes redundant coordinates locally; see
  3.

- **Constraint-preserving integrators** enforce the geometry inside the step and are especially attractive for long-time mechanical simulation.

The correct choice depends on index, stiffness, time horizon, and whether exact constraint satisfaction or faithful invariant behavior is the main priority.

8.4 Baumgarte Stabilization

8.4.1 DAEs in Mechanics

This document offers an in-depth exploration of how Differential-Algebraic Equations (DAEs) are formulated and numerically solved, particularly within the context of mechanical systems. It emphasizes systems that involve constraints, which commonly appear in areas like multibody dynamics, robotics, and structural mechanics.

8.4.1.1 Notation

Consider a mechanical system with the following quantities:

- **Generalized (position) coordinates**: $\boldsymbol{p} \in \mathbb{R}^n$, representing the configuration of the system.
- **Kinetic energy**: $T(\boldsymbol{p}, \boldsymbol{p}', t)$, where $\boldsymbol{p}'$ is the time derivative of $\boldsymbol{p}$.
- **Potential energy**: $V(\boldsymbol{p}, t)$.
- **External generalized forces**: $\boldsymbol{Q}_{ex}(\boldsymbol{p}, t) \in \mathbb{R}^n$.
- **Constraint equations**: $\Phi(\boldsymbol{p}, t) \in \mathbb{R}^m$, which enforce m constraints on the system.
- **Lagrangian**:

$$L(\boldsymbol{p}, \boldsymbol{p}', t) = T(\boldsymbol{p}, \boldsymbol{p}', t) - V(\boldsymbol{p}, t).$$

The **equations of motion** are derived from Lagrange's equations with constraints:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \boldsymbol{p}'}\right) - \frac{\partial L}{\partial \boldsymbol{p}} + \left(\frac{\partial \Phi(\boldsymbol{p}, t)}{\partial \boldsymbol{p}}\right)^T \boldsymbol{\lambda} = \boldsymbol{Q}_{ex}(\boldsymbol{p}, \boldsymbol{p}', t),$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ are the **Lagrange multipliers** enforcing the constraints.

8.4.1.2 DAE Formulation

Kinetic Energy and Mass Matrix

The kinetic energy is often expressed in terms of the mass matrix $\boldsymbol{M}(\boldsymbol{p}, \boldsymbol{p}', t)$:

$$T(\boldsymbol{p}, \boldsymbol{p}', t) = \frac{1}{2}(\boldsymbol{p}')^T \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{p}', t)\boldsymbol{p}'.$$

Substituting this into the equations of motion yields:

$$\boldsymbol{M}(\boldsymbol{p}, \boldsymbol{p}', t)\boldsymbol{p}'' + \left(\frac{\partial \Phi}{\partial \boldsymbol{p}}\right)^T \boldsymbol{\lambda} = \boldsymbol{Q}(\boldsymbol{p}, \boldsymbol{p}', t),$$

where $\boldsymbol{Q}(\boldsymbol{p}, \boldsymbol{p}', t)$ includes external forces, potential energy gradients, and inertial terms.

Constraint Equations

The constraints $\Phi(\boldsymbol{p}, t) = 0$ must be satisfied at all times. Combining these with the equations of motion, we obtain the **full DAE system**:

$$\begin{cases} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{p}'t)\boldsymbol{p}'' + \Phi_p(\boldsymbol{p}, t)^T\boldsymbol{\lambda} = \boldsymbol{Q}(\boldsymbol{p}, \boldsymbol{p}', t), \\ \Phi(\boldsymbol{p}, t) = \boldsymbol{0}. \end{cases}$$

This system has $n + m$ equations and $n + m$ unknowns ($\boldsymbol{p}''$ and $\boldsymbol{\lambda}$), making it well-posed.

8.4.2 Numerical Scheme

Solving DAEs requires specialized numerical methods due to the presence of algebraic constraints. Below, we outline a common approach based on time discretization and constraint differentiation.

8.4.2.1 Time Discretization

A simple numerical scheme can be derived using a **Taylor expansion** in time:

$$\boldsymbol{p}'(t+\Delta t) = \boldsymbol{p}'(t) + \boldsymbol{p}''(t)\Delta t + \mathcal{O}(\Delta t^2),$$

$$\boldsymbol{p}(t+\Delta t) = \boldsymbol{p}(t) + \boldsymbol{p}'(t)\Delta t + \frac{1}{2}\boldsymbol{p}''(t)\Delta t^2 + \mathcal{O}(\Delta t^3).$$

Neglecting higher-order terms, the updates for position and velocity become:

$$\boldsymbol{p}'_{k+1} = \boldsymbol{p}'_k + \boldsymbol{p}''_k \Delta t,$$

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k + \boldsymbol{p}'_k \Delta t + \frac{1}{2}\boldsymbol{p}''_k \Delta t^2.$$

This requires computing $\boldsymbol{p}''_k$, which involves solving for $\boldsymbol{\lambda}$.

8.4.2.2 Accelerations and Multipliers

If the mass matrix $\boldsymbol{M}(\boldsymbol{p}, \boldsymbol{q}', t)$ is **nonsingular**, we can express $\boldsymbol{p}''$ as:

$$\boldsymbol{p}'' = \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{q}', t)^{-1}\left[\boldsymbol{Q}(\boldsymbol{p}, \boldsymbol{p}', t) - \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \boldsymbol{\lambda}\right].$$

However, $\boldsymbol{\lambda}$ is still unknown. To resolve this, we differentiate the constraint equations to incorporate $\boldsymbol{p}''$.

Differentiating Constraints

Differentiating the constraints $\boldsymbol{\Phi}(\boldsymbol{p}, t) = \boldsymbol{0}$ once with respect to time yields:

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{\Phi}(\boldsymbol{p}, t) = \boldsymbol{\Phi}_t + \boldsymbol{\Phi}_{\boldsymbol{p}}\boldsymbol{p}' = \boldsymbol{0}.$$

Differentiating a second time gives:

$$\boldsymbol{\Phi}_{tt} + 2\boldsymbol{\Phi}_{t\boldsymbol{p}}\boldsymbol{p}' + \boldsymbol{\Phi}_{\boldsymbol{pp}}(\boldsymbol{p}', \boldsymbol{p}') + \boldsymbol{\Phi}_{\boldsymbol{p}}\boldsymbol{p}'' = \boldsymbol{0}.$$

Rewriting in matrix form:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{q}', t) & \boldsymbol{\Phi}_{\boldsymbol{p}}^T \\ \boldsymbol{\Phi}_{\boldsymbol{p}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p}'' \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{Q}(\boldsymbol{p}, \boldsymbol{p}', t) \\ \boldsymbol{c}(\boldsymbol{p}, \boldsymbol{p}', t) \end{pmatrix},$$

where:

$$\boldsymbol{c}(\boldsymbol{p}, \boldsymbol{p}', t) = -\boldsymbol{\Phi}_{tt} - 2\boldsymbol{\Phi}_{t\boldsymbol{p}}\boldsymbol{p}' - (\boldsymbol{p}')^T \boldsymbol{\Phi}_{\boldsymbol{pp}}\boldsymbol{p}'.$$

This system can be solved for $\boldsymbol{p}''$ and $\boldsymbol{\lambda}$.

8.4.2.3 Integration Algorithm

Initialization

Given initial values $t_0 = 0$, $\boldsymbol{p}_0$, and $\boldsymbol{p}'_0$.

Time Stepping

For each time step $k = 0, 1, \ldots$:

1. **Solve for $\boldsymbol{p}''_k$ and $\boldsymbol{\lambda}_k$:**

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}_k, \boldsymbol{p}'_k, t_k) & \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}_k, t_k)^T \\ \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}_k, t_k) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{p}''_k \\ \boldsymbol{\lambda}_k \end{pmatrix} = \begin{pmatrix} \boldsymbol{Q}(\boldsymbol{p}_k, \boldsymbol{p}'_k, t_k) \\ \boldsymbol{c}(\boldsymbol{p}_k, \boldsymbol{p}'_k, t_k) \end{pmatrix}.$$

2. **Update position and velocity**:

$$\boldsymbol{p}'_{k+1} = \boldsymbol{p}'_k + \Delta t\, \boldsymbol{p}''_k$$

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k + \Delta t\, \boldsymbol{p}'_k + \frac{1}{2}\Delta t^2 \boldsymbol{p}''_k.$$

8.4.2.4 Constraint Drift

Over time, numerical errors can cause the system to drift away from the constraints $\Phi(\boldsymbol{p}, t) = \boldsymbol{0}$. To mitigate this, we redefine the constraints as:

$$\widetilde{\Phi}(\boldsymbol{p}, t) = \Phi(\boldsymbol{p}, t) + \boldsymbol{a}_0 + \boldsymbol{a}_1 t,$$

where $\boldsymbol{a}_0$ and $\boldsymbol{a}_1$ are chosen to satisfy the initial conditions:

$$\widetilde{\Phi}(\boldsymbol{p}(0), 0) = \boldsymbol{0}, \quad \frac{\mathrm{d}}{\mathrm{d}t}\widetilde{\Phi}(\boldsymbol{p}(0), 0) = \boldsymbol{0}.$$

If $\boldsymbol{a}_0 = \boldsymbol{a}_1 = \boldsymbol{0}$, the constraints are perfectly satisfied initially. However, small errors can accumulate, leading to drift over long simulations.

8.4.3 Redundant-Coordinate Pendulum

This section demonstrates the application of Differential-Algebraic Equations (DAEs) to model a pendulum using redundant coordinates. The pendulum is a classic example of a mechanical system with constraints, making it an ideal case study for DAE formulation and numerical solution.

8.4.3.1 Model

Kinetic and Potential Energy

The pendulum consists of a mass m attached to a rigid, massless rod of length 1. The system is described using Cartesian coordinates (x, y), where x and y represent the horizontal and vertical positions of the mass, respectively.

- **Kinetic energy**:

$$T(t) = \frac{1}{2} m \left(x'(t)^2 + y'(t)^2 \right).$$

- **Potential energy**:

$$V(t) = mgy(t),$$

where g is the gravitational acceleration.

Constraint Equation

The pendulum is constrained to move along a circular path of radius 1. This constraint is expressed as:

$$\Phi(x, y) = x^2 + y^2 - 1 = 0.$$

Schematic

8.4.3.2 Equations of Motion

Using Lagrange's equations with constraints, the equations of motion for the pendulum are:

$$\begin{cases} mx''(t) + 2x(t)\lambda(t) = 0, \\ my''(t) + 2y(t)\lambda(t) = -mg, \\ x(t)^2 + y(t)^2 = 1, \end{cases} \tag{8.2}$$

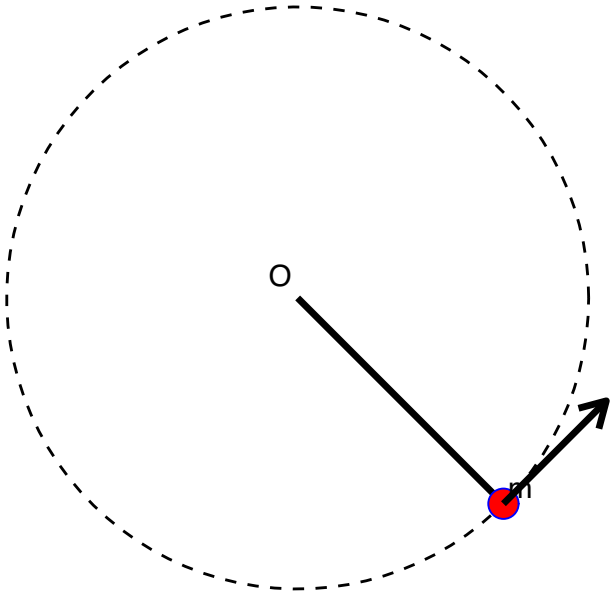


Figure 8.7: Geometry, position constraint, and forces for the planar pendulum.

where $\lambda(t)$ is the Lagrange multiplier enforcing the constraint.

8.4.4 Unstabilized Solution

To solve the system numerically Equation 8.2, we differentiate the constraint equation twice to incorporate the acceleration terms $x''(t)$ and $y''(t)$.

First Constraint Derivative

Differentiating $\Phi(x, y) = 0$ once with respect to time yields:

$$2x(t)x'(t) + 2y(t)y'(t) = 0.$$

Second Constraint Derivative

Differentiating again gives:

$$2x(t)x''(t) + 2y(t)y''(t) + 2x'(t)^2 + 2y'(t)^2 = 0.$$

This equation introduces the acceleration terms $x''(t)$ and $y''(t)$, allowing us to solve for $\lambda(t)$.

The system of equations can be written in matrix form as:

$$\begin{pmatrix} m & 0 & 2x(t) \\ 0 & m & 2y(t) \\ 2x(t) & 2y(t) & 0 \end{pmatrix} \begin{pmatrix} x''(t) \\ y''(t) \\ \lambda(t) \end{pmatrix} = \begin{pmatrix} 0 \\ -mg \\ -2x'(t)^2 - 2y'(t)^2 \end{pmatrix}.$$

8.4.4.1 Time Discretization

Given initial conditions x_0, y_0, x_0', and y_0', we solve the system iteratively using the following steps:

1. Solve for x_k'', y_k'', and λ_k:

$$\begin{pmatrix} m & 0 & 2x_k \\ 0 & m & 2y_k \\ 2x_k & 2y_k & 0 \end{pmatrix} \begin{pmatrix} x_k'' \\ y_k'' \\ \lambda_k \end{pmatrix} = \begin{pmatrix} 0 \\ -mg \\ -2(x_k')^2 - 2(y_k')^2 \end{pmatrix}.$$

2. Update position and velocity:

$$
\begin{aligned}
x_{k+1} &= x_k + \Delta t\, x'_k + \frac{\Delta t^2}{2} x''_k, \\
y_{k+1} &= y_k + \Delta t\, y'_k + \frac{\Delta t^2}{2} y''_k, \\
x'_{k+1} &= x'_k + \Delta t\, x''_k, \\
y'_{k+1} &= y'_k + \Delta t\, y''_k.
\end{aligned}
$$

Numerical solution shows that the numerical method do not work well

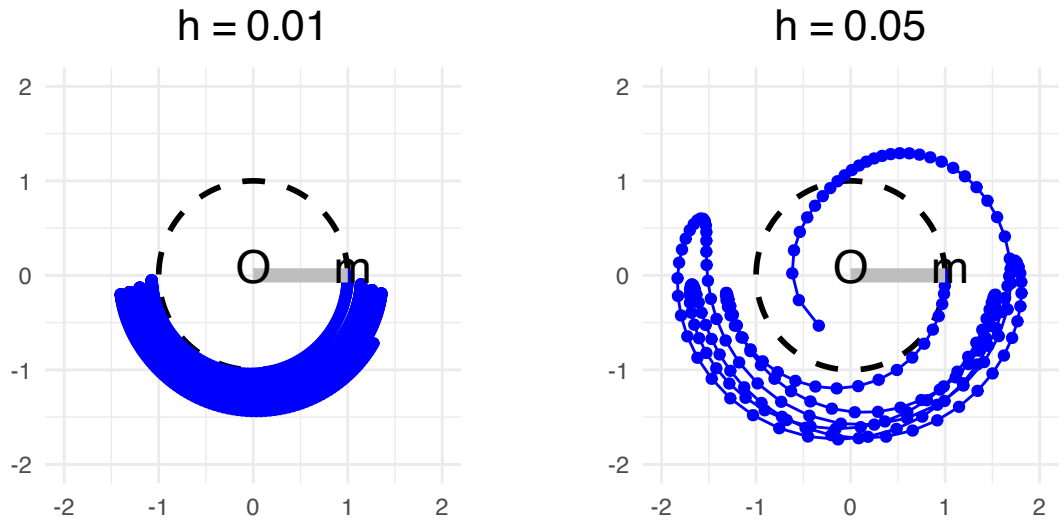


Figure 8.8: **Consistent Initial Conditions**: $x(0) = 1$, $y(0) = u(0) = v(0) = 0$. The system remains stable for short times but exhibits drift over longer simulations.

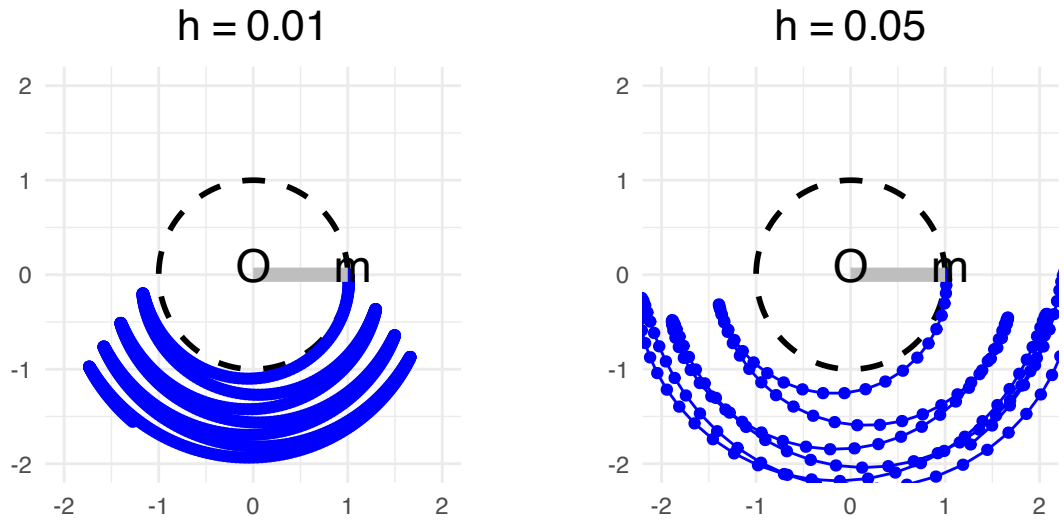


Figure 8.9: **Inconsistent Initial Conditions**: $x(0) = 1$, $y(0) = v(0) = 0$, $u(0) = 0.1$:. The system quickly diverges from the constraint.

8.4.5 Baumgarte Method

A very popular way to stabilize the constraint $\boldsymbol{\phi}(\boldsymbol{p}, t) = 0$ and thus avoid the drift effect is using Baumgarte's stabilization, see @baumgarte1972stabilization. This idea is well presented with other techniques in the survey paper @ascher1995stabilization. The key idea is to substitute the constraint with a *differential equation* that *asymptotically* satisfies $\boldsymbol{\phi}(\boldsymbol{p}, t) = 0$. Obviously this technique is effective only if you are interested in the *long term* behaviour, since during transients the constraint may be violated. Since constrained mechanical systems require to differentiations to eliminate the Lagrange multipliers, we consider a linear second order stable differential equation of the form,

$$\frac{\mathrm{d}^2 z(t)}{\mathrm{d}t^2} + 2\xi\omega_\mathrm{n}\frac{\mathrm{d}z(t)}{\mathrm{d}t} + \omega_\mathrm{n}^2 z(t) = 0, \tag{8.3}$$

where $z(t) \in \mathbb{R}$. In this form we recognize the well known equation for second order linear mechanical or electrical systems, such as the *RLC circuit* or the *mass spring damper*. In these framework $\xi \in \mathbb{R}_{>0}$ is the *damping ratio* and $\omega_\mathrm{n} \in \mathbb{R}_{>0}$ is the *natural frequency*. Depending on the value of ξ different cases can arise:

- $0 < \xi < 1$: solution z is *underdamped*,
- $\xi = 1$: solution z is *critically damped*,
- $\xi > 1$: solution z is *overdamped*.

The general form for the solution is the following

$$z(t) = k_1 e^{\left(-\xi + \sqrt{\xi^2-1}\right)\omega_\mathrm{n} t} + k_2 e^{\left(-\xi - \sqrt{\xi^2-1}\right)\omega_\mathrm{n} t},$$

where $k_1, k_2 \in \mathbb{R}$ are constants determined by the initial conditions $z_0 = z(t = 0)$ and $\dot{z}_0 = \dot{z}(t = 0)$. Solutions for the different cases can be arranged in different ways, here we show some of them. For a deeper treatment see for example @rao1995mechanical. Let us start from the underdamped case; solution takes the form

$$z(t) = c_1 e^{-\xi\omega_\mathrm{n} t} \cos\left(\sqrt{1-\xi^2}\omega_\mathrm{n} t - c_2\right), \tag{8.4}$$

where constants $c_1, c_2 \in \mathbb{R}$ are given by,

$$c_1 = \frac{\sqrt{z_0^2\omega_\mathrm{n}^2 + \dot{z}_0^2 + 2z_0\dot{z}_0\xi\omega_\mathrm{n}}}{\sqrt{1-\xi^2}\omega_\mathrm{n}}$$

$$c_2 = \arctan\left(\frac{\dot{z}_0 + \xi\omega_\mathrm{n} z_0}{x_0\omega_\mathrm{n}\sqrt{1-\xi^2}}\right).$$

As we can observe in Equation 8.4 solution is a cosine function modulated by a decreasing exponential function. For the critically damped case a linear term appears and the solution takes the form,

$$z(t) = (c_1 + c_2 t)e^{-\omega_\mathrm{n} t}, \tag{8.5}$$

where $c_1, c_2 \in \mathbb{R}$ are given by $c_1 = z_0$ and $c_2 = \dot{z}_0 + \omega_\mathrm{n} z_0$. Finally for the overdamped case the solution takes the form

$$z(t) = c_1 e^{\left(-\xi + \sqrt{\xi^2-1}\right)\omega_\mathrm{n} t} + c_2 e^{\left(-\xi - \sqrt{\xi^2-1}\right)\omega_\mathrm{n} t}, \tag{8.6}$$

with constants,

$$c_1 = \frac{x_0\omega_n\left(\xi + \sqrt{\xi^2-1}\right) + \dot{x}_0}{2\omega_n\sqrt{\xi^2-1}},$$
$$c_2 = \frac{-x_0\omega_n\left(\xi - \sqrt{\xi^2-1}\right) - \dot{x}_0}{2\omega_n\sqrt{\xi^2-1}}.$$

However, regardless of the specific form of the solution, all the three cases satisfy a *global exponential stability* property, i.e., $\forall(z_0, \dot{z}_0) \in \mathbb{R}\times\mathbb{R}$ the solution $z(t, z_0, \dot{z}_0)$ satisfies

$$\lim_{t\to\infty} z(t, z_0, \dot{z}_0) = 0.$$

The notation $z(t, z_0, \dot{z}_0)$ is just to stress that the solution depends on time but also from initial conditions $(z_0, \dot{z}_0)$. This consideration suggests to substitute the constraint $\boldsymbol{\phi}(\boldsymbol{p}, t)$ with an ODE of the form Equation 8.3. In this way, despite the initial conditions, the solution will asymptotically satisfies $\boldsymbol{\phi}(\boldsymbol{p}, t) = 0$. Thus the explicit *Baumgarte's stabilization* for DAEs of index three takes the form,

$$\frac{\mathrm{d}^2\boldsymbol{\phi}}{\mathrm{d}t^2} + 2\xi\omega_n\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t} + \omega_n^2\boldsymbol{\phi} = 0.$$

Computing explicitly the terms for mechanical systems the stabilization results,

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\boldsymbol{\phi}}{\partial\boldsymbol{p}}\dot{\boldsymbol{p}} + \frac{\partial\boldsymbol{\phi}}{\partial\boldsymbol{p}}\ddot{\boldsymbol{p}} + \frac{\partial^2\boldsymbol{\phi}}{\partial t^2} + 2\xi\omega_n\frac{\partial\phi}{\partial\boldsymbol{p}}\dot{\boldsymbol{p}} + 2\xi\omega_n\frac{\partial\boldsymbol{\phi}}{\partial t} + \omega_n^2\boldsymbol{\phi} = 0.$$

which can be re-arranged in the following form

$$\begin{aligned} -\frac{\partial\boldsymbol{\phi}}{\partial\boldsymbol{p}}\dot{\boldsymbol{v}} &= \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial\boldsymbol{\phi}}{\partial\boldsymbol{p}}\boldsymbol{v} + \frac{\partial^2\boldsymbol{\phi}}{\partial t^2} + 2\xi\omega_n\frac{\partial\phi}{\partial\boldsymbol{p}}\boldsymbol{v} + 2\xi\omega_n\frac{\partial\boldsymbol{\phi}}{\partial t} + \omega_n^2\boldsymbol{\phi} \\ &= \boldsymbol{m}(\boldsymbol{p}, \boldsymbol{v}, t). \end{aligned} \tag{8.7}$$

Now Equation 8.7 can be easy implemented without any modification in the scheme of DAEs from Lagrangian Mechanics; indeed it is sufficient to redefine $\boldsymbol{m}(\boldsymbol{p}, \boldsymbol{v}, t)$ as in Equation 8.7. However a question is still open, how can we tune the parameters ω_n and ξ?. Unfortunately there is not general rule, but it is useful to notice that the velocity of convergence to zero of the solution depends on bothξ and ω_n. Therefore it is a good idea to select them in order to obtain an high decay rate compatibly with the *numerical stability*. This means that the dynamics associated to the constraint $\boldsymbol{\phi}(\boldsymbol{p}, t)$ cannot be too fast compared to other ones, otherwise the system could be very hard (stiff) to simulate numerically. Indeed systems where two or more very different time scales are involved are usually called *stiff problems* or *stiff systems**. These may cause some numerical problems, since stiff equations require an integration step extremely small and are numerically unstable. In general it is hard to provide a unique definition of stiffness, but the main idea is that the differential equations includes some terms that can lead to rapid variation in the solution.

▪ **Example 8.1** Continuing Example 8.1} we want use the Baumgarte's echnique to stabilize the pendulum. Therefore the idea is to substitute the constraint

$$2\boldsymbol{v}^\top\boldsymbol{v} + 2\boldsymbol{p}\dot{\boldsymbol{v}} = 0$$

with

$$-2(\boldsymbol{p}\cdot\dot{\boldsymbol{v}}) = 2(\boldsymbol{v}\cdot\boldsymbol{v}) + 4\omega_n\xi(\boldsymbol{p}\cdot\boldsymbol{v}) + \omega_n^2(\boldsymbol{p}\cdot\boldsymbol{p} - \ell^2).$$

Thus final equations appears as,

$$\begin{bmatrix} \boldsymbol{I} & 0 & 0 \\ 0 & m\boldsymbol{I} & -2\boldsymbol{p} \\ 0 & -2\boldsymbol{p}^T & 0 \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{v}} \\ \lambda \end{bmatrix} = \begin{bmatrix} \boldsymbol{v} \\ -mg\boldsymbol{e}_2 \\ 2(\boldsymbol{v}\cdot\boldsymbol{v}) + 4\omega_n\xi(\boldsymbol{p}\cdot\boldsymbol{v}) + \omega_n^2(\boldsymbol{p}\cdot\boldsymbol{p} - \ell^2) \end{bmatrix}. \tag{8.8}$$

Again to appreciate the effects of Baumgarte's stabilization we can simulate Equation 8.8 with same parameters as Example 8.1 and stabilization parameters $\xi = 0.8$, $\omega_n = 2$. The result is showed in Figure 8.10.

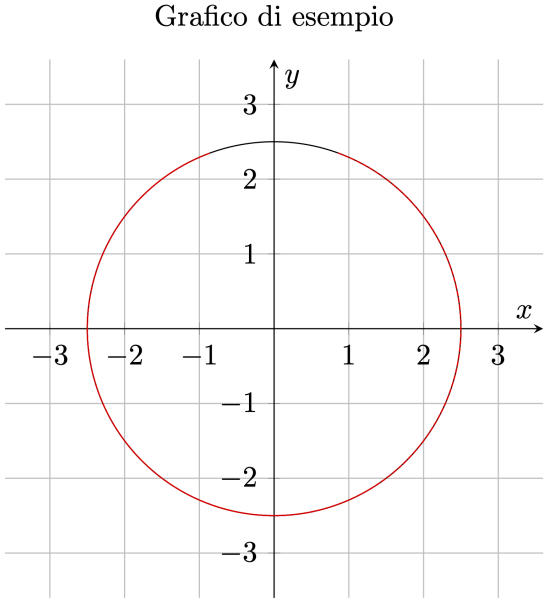


Figure 8.10: Pendulum simulation with Baumgarte's stabilization

▪

8.4.5.1 Method Details

The **Baumgarte method** stabilizes the constraints by introducing damping into the system. Consider the following ODE:

$$z''(t) + 2\zeta\omega z'(t) + \omega^2 z(t) = 0,$$

which has the solution:

$$z(t) = A(t)e^{-\zeta\omega t},$$

where $A(t)$ depends on the initial conditions. For $0 \leq \zeta \leq 1$, the solution decays exponentially without oscillation.

Over time, numerical errors can cause the system to drift away from the constraint $\Phi(x, y)$ of Equation 8.2 where

$$\Phi(x, y) = x^2 + y^2 - 1 = 0 \tag{8.9}$$

To mitigate this, we use the **Baumgarte stabilization** method, which modifies the constraint equation to include damping terms.

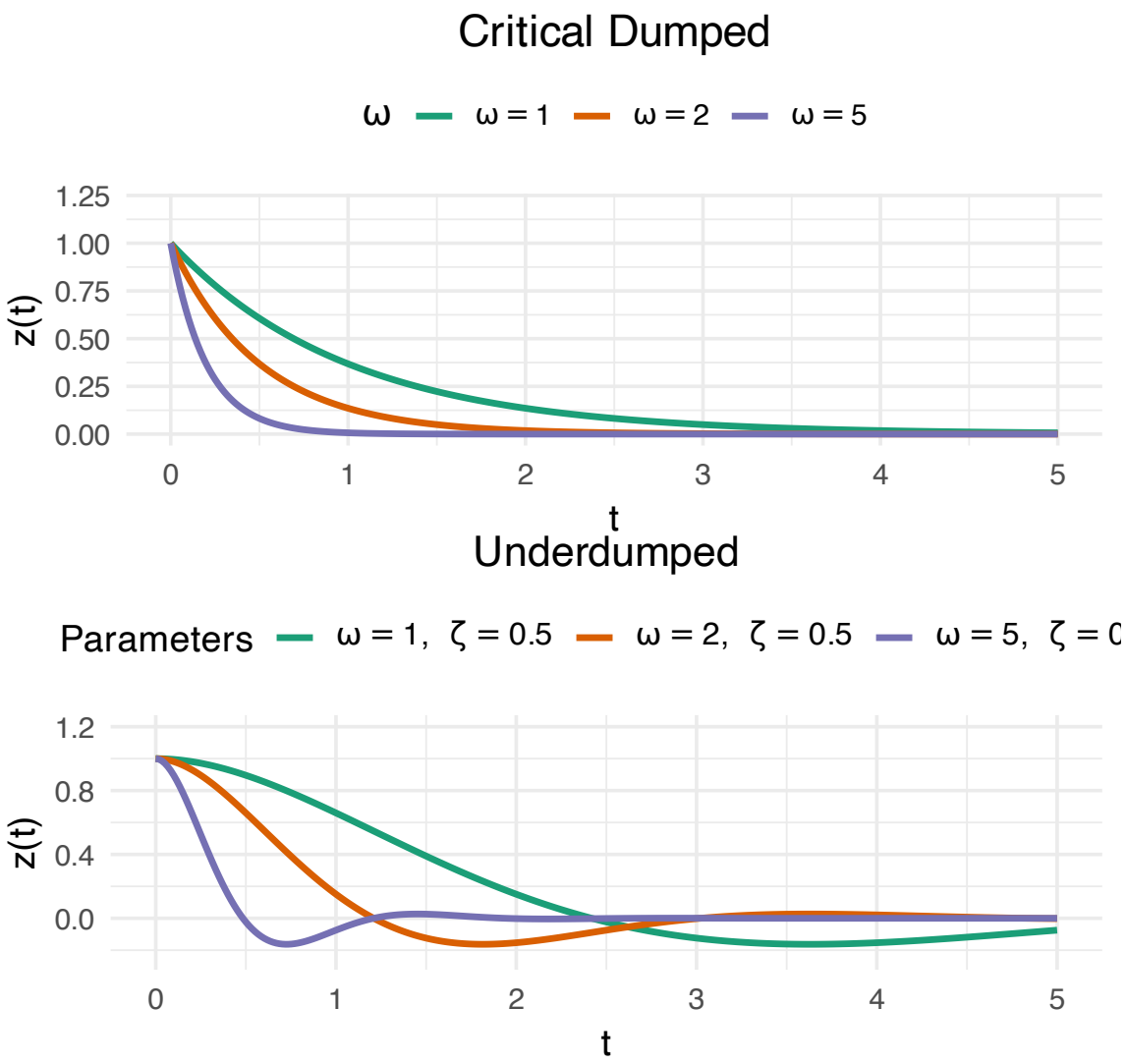


Figure 8.11: Solution example for some values of ω and ζ.

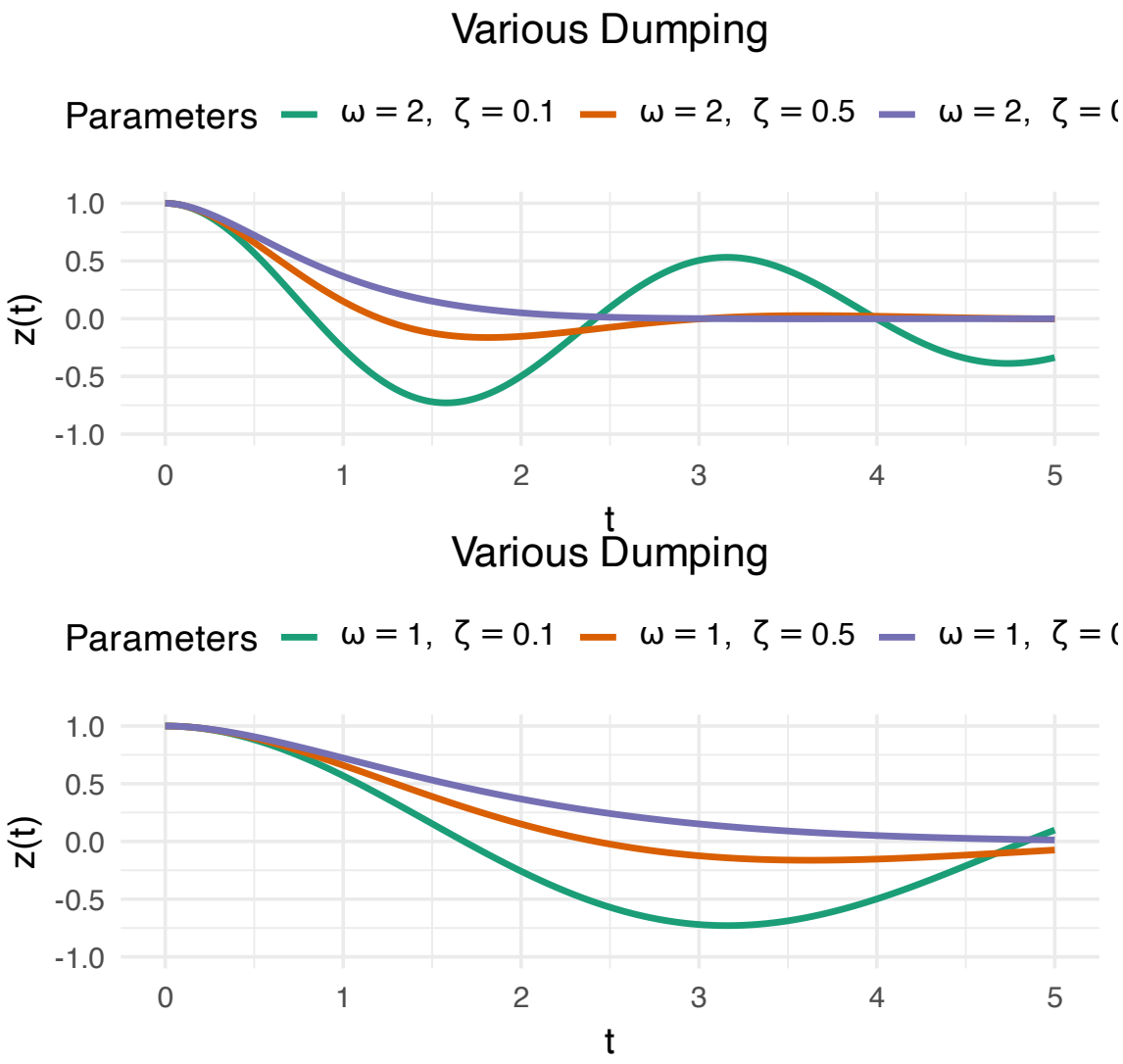


Figure 8.12: Solution example for some values of ω and ζ.

The stabilized constraint equation is:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\Phi(x,y) + 2\zeta\omega\frac{\mathrm{d}}{\mathrm{d}t}\Phi(x,y) + \omega^2\Phi(x,y) = 0,$$

where ζ and ω are damping parameters. For Equation 8.9 this becomes:

$$2xx'' + 2yy'' + 2x'^2 + 2y'^2 + 4\zeta\omega(xx' + yy') + \omega^2(x^2 + y^2 - 1) = 0.$$

8.4.5.2 Modified Linear System

The stabilized system is solved using the same time-stepping approach, but with the right-hand side of the linear system modified to include the stabilization terms:

$$r_k = 2\left[(x_k')^2 + (y_k')^2\right] + 4\zeta\omega(x_k x_k' + y_k y_k') + \omega^2(x_k^2 + y_k^2 - 1).$$

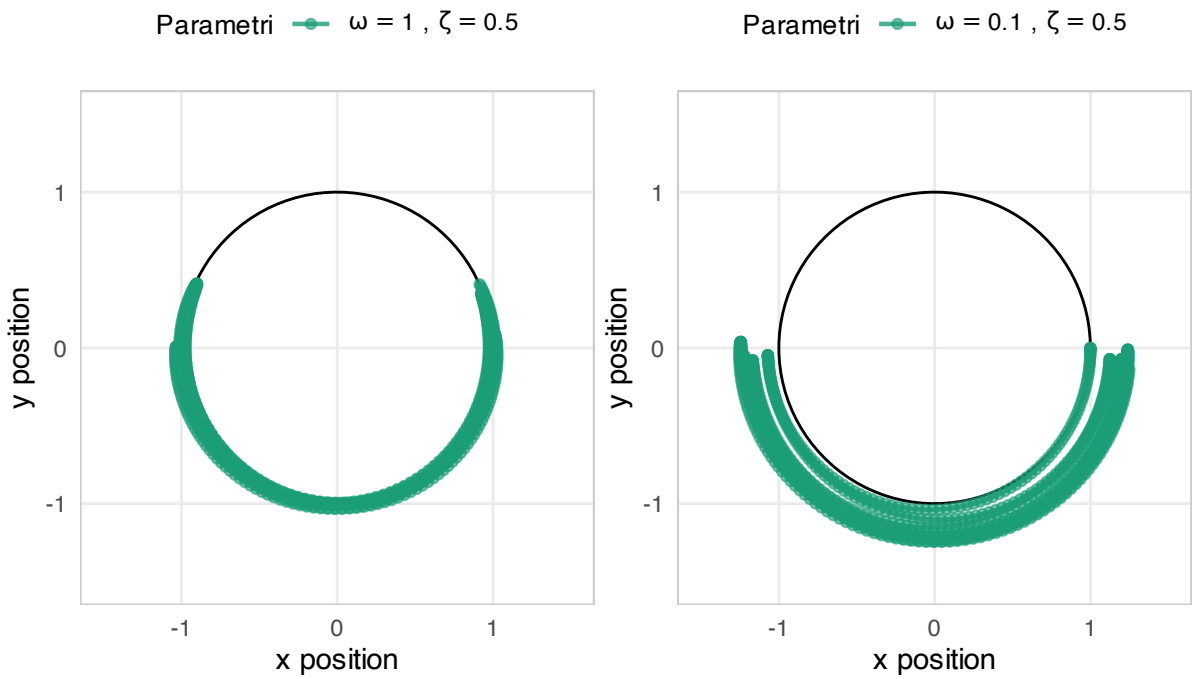


Figure 8.13: **Consistent Initial Conditions**: The system remains stable even for long simulation times. Energy is not conserved. Initial condition $x(0) = 1$, $y(0) = u(0) = v(0) = 0$, time step $\Delta t = 0.01$ final time $t_{\max} = 10$. Stabilization parameters are $\zeta = 0.5$ and $\omega = 1$ on the left and $\omega = 0.1$ on the right.

Inconsistent Initial Conditions: The system converges back to the constraint due to the damping terms. Initial condition $x(0) = 1$, $y(0) = v(0) = 0$, $u(0) = 0.1$ time step $\Delta t = 0.01$ final time $t_{\max} = 10$. Stabilization parameters are $\zeta = 0.5$ and $\omega = 1$ on the left and $\omega = 0.1$ on the right.

The pendulum example illustrates the challenges of solving DAEs numerically, particularly the issue of constraint drift. The Baumgarte stabilization method effectively mitigates this drift, ensuring long-term stability of the solution. Proper choice of damping parameters ζ and ω is critical for achieving optimal performance.

8.4.6 First-Order Pendulum DAE

Start from the motion equations

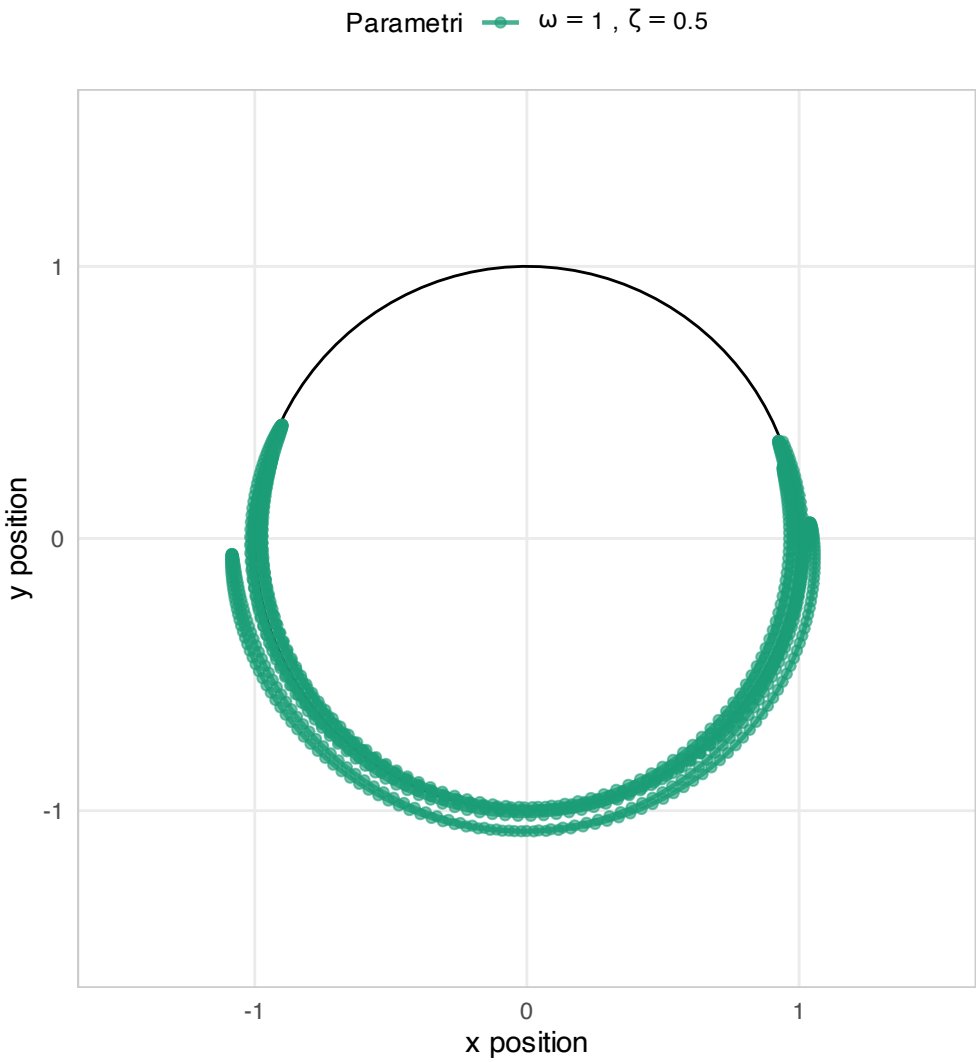


Figure 8.14: Recovery from inconsistent initial data under Baumgarte stabilization.

$$\begin{cases} mx'' + 2x\lambda = 0 \\ my'' + 2y\lambda + mg = 0 \\ x^2 + y^2 = 1 \end{cases}$$

divide by m and set $\mu = -2\lambda/m$

$$\begin{cases} x'' = \mu x \\ y'' = \mu y - g \\ x^2 + y^2 = r^2 \end{cases}$$

set $u = x'$ and $v = y'$ and transform to first order

$$\begin{cases} u' = \mu x \\ v' = \mu y - g \\ x' = u \\ y' = v \\ x^2 + y^2 = r^2 \end{cases}$$

Derive one time the constraint

$$2xx' + 2yy' = 0 \quad \Longrightarrow \quad xu + yv = 0$$

Derive another time the constraint

$$\begin{aligned} x\dot{u} + y\dot{v} + \dot{x}u + \dot{y}v = 0 \quad &\Longrightarrow \\ \mu(x^2 + y^2) - gy + (u^2 + v^2) = 0 \quad &\Longrightarrow \quad r^2\mu - gy + (u^2 + v^2) = 0 \end{aligned}$$

Derive a third time the constraint obtaining a differential equation for μ:

$$\begin{aligned} r^2\dot{\mu} - gv + 2(u\dot{u} + v\dot{v}) = 0 \quad &\Longrightarrow \\ r^2\dot{\mu} - gv + 2\mu(ux + vy) - 2gv = 0 \quad &\Longrightarrow \\ r^2\dot{\mu} - 3gv = 0 \end{aligned}$$

The DAE has index 3

i Summary

Original DAE

$$\begin{cases} u' = \mu x \\ v' = \mu y - g \\ x' = u \\ y' = v \\ x^2 + y^2 = 1 \end{cases}$$

hidden constraints

$$\begin{aligned} xu + yv = 0 \\ \mu - gy + (u^2 + v^2) = 0 \end{aligned}$$

constraint differential equation

$$\mu' - 3gv = 0$$

8.4.7 Unstabilized Integration

Consider the ODE without stabilization

$$\begin{cases} u' = \mu x \\ v' = \mu y - g \\ x' = u \\ y' = v \\ \mu' = 3gv/r^2 \end{cases}$$

Integration with implicit Euler:

$$\begin{aligned} \frac{u_{k+1} - u_k}{h} &= \mu_{k+1} x_{k+1} \\ \frac{v_{k+1} - v_k}{h} &= \mu_{k+1} y_{k+1} - g \\ \frac{x_{k+1} - x_k}{h} &= u_{k+1} \\ \frac{y_{k+1} - y_k}{h} &= v_{k+1} \\ \frac{\mu_{k+1} - \mu_k}{h} &= \frac{3g v_{k+1}}{r^2} \end{aligned}$$

Solving the first four equations for x_{k+1}, y_{k+1}, u_{k+1} and v_{k+1}

$$\begin{aligned} c_{k+1} &= 1 - h^2 \mu_{k+1} \\ x_{k+1} &= (x_k + h u_k)/c_{k+1} \\ y_{k+1} &= (y_k + h v_k - g h^2)/c_{k+1} \\ u_{k+1} &= (u_k + h \mu_{k+1} x_k)/c_{k+1} \\ v_{k+1} &= (v_k + h \mu_{k+1} y_k - g h)/c_{k+1} \end{aligned}$$

substituting into the multiplier equation and set $\delta = \mu_{k+1} - \mu_k$ a quadratic in δ is obtained:

$$r^2 h^2 \delta^2 + (3gh^2 y_k + r^2(h^2 \mu_k - 1))\delta + 3gh(v_k + y_k \mu_k h - gh) = 0$$

Choose for the solution the δ closer to the esplicit step, i.e. $\delta \approx 3hgv_k/r^2$

Implicit Eulero implicito without stabilization do not work. The constraint $x^2 + y^2 = 1$ is violated in few steps:

onsider the DAE without stabilization

$$\begin{cases} u' = \mu x \\ v' = \mu y - g \\ x' = u \\ y' = v \\ x^2 + y^2 = r^2 \end{cases}$$

Using implicit Euler:

$$\begin{aligned}
\frac{u_{k+1}-u_k}{h} &= \mu_{k+1}x_{k+1}\\
\frac{v_{k+1}-v_k}{h} &= \mu_{k+1}y_{k+1}-g\\
\frac{x_{k+1}-x_k}{h} &= u_{k+1}\\
\frac{y_{k+1}-y_k}{h} &= v_{k+1}\\
x_{k+1}^2+y_{k+1}^2 &= 1
\end{aligned}$$

Solving the first four equation for x_{k+1}, y_{k+1}, u_{k+1} and v_{k+1}

$$\begin{aligned}
c_{k+1} &= 1-h^2\mu_{k+1}\\
x_{k+1} &= (x_k+hu_k)/c_{k+1}\\
y_{k+1} &= (y_k+hv_k-gh^2)/c_{k+1}\\
u_{k+1} &= (u_k+h\mu_{k+1}x_k)/c_{k+1}\\
v_{k+1} &= (v_k+h\mu_{k+1}y_k-gh)/c_{k+1}
\end{aligned}$$

substituting into constraint a quadratic equation in μ_{k+1} is obtained:

$$x_k^2+y_k^2-r^2+2h(x_ku_k+y_0v_k)+h^2(u_k^2+v_k^2)+g^2h^4-2gh^2(y_k+hv_k)+2h^2\mu_{k+1}-h^4\mu_{k+1}^2=0$$

$$\underbrace{[x_k^2+y_k^2-r^2]}_{A}+2h\underbrace{(x_ku_k+y_0v_k)}_{B}+h^2(u_k^2+v_k^2)$$
$$+g^2h^4-2gh^2(y_k+hv_k)+2h^2\mu_{k+1}-h^4\mu_{k+1}^2=0$$

On the solution block (A) is zero so it is removed. Block (B) is the constraint derivative not 0 in the discretization. Thus,

$$2(x_ku_k+y_0v_k)+h(u_k^2+v_k^2)+g^2h^3-2gh(y_k+hv_k)+2h\mu_{k+1}-h^3\mu_{k+1}^2=0$$

Solving the quadratic

$$\mu_{k+1}=\frac{r\pm\sqrt{r^2+A}}{h^2r}$$

$$A=g^2h^4+2h(x_ku_k+y_kv_k)-2gh^2(y_k-hv_k)+h^2(u_k^2+v_k^2)$$

μ represent centripetal force maintaining mass on the constraint. Thus μ must be negative and solving trying to avoid cancellation error:

$$A=g^2h^4+2h(x_ku_k+y_kv_k)-2gh^2(y_k-hv_k)+h^2(u_k^2+v_k^2)$$

$$\mu_{k+1}=\frac{-A}{h^2(r^2+r\sqrt{r^2+A})}=\frac{-A/r^2}{h^2(1+\sqrt{1+A/r^2})}$$

8.4.7.1 Numerical Results

8.4.8 Stabilized Integration

Inplicit Euler work but it is too dissipative. Cranck-Nicolson needs stabilization. DAE transformed to ODE need a third order ODE to stabilize as follows:

$$\frac{\mathrm{d}^3V(x,y)}{\mathrm{d}t^3}+a\frac{\mathrm{d}^2V(x,y)}{\mathrm{d}t^2}+b\frac{\mathrm{d}V(x,y)}{\mathrm{d}t}+cV(x,y)=0$$

where $V(x,y)=x^2+y^2-1$. And a, b, c are chosen in such a way

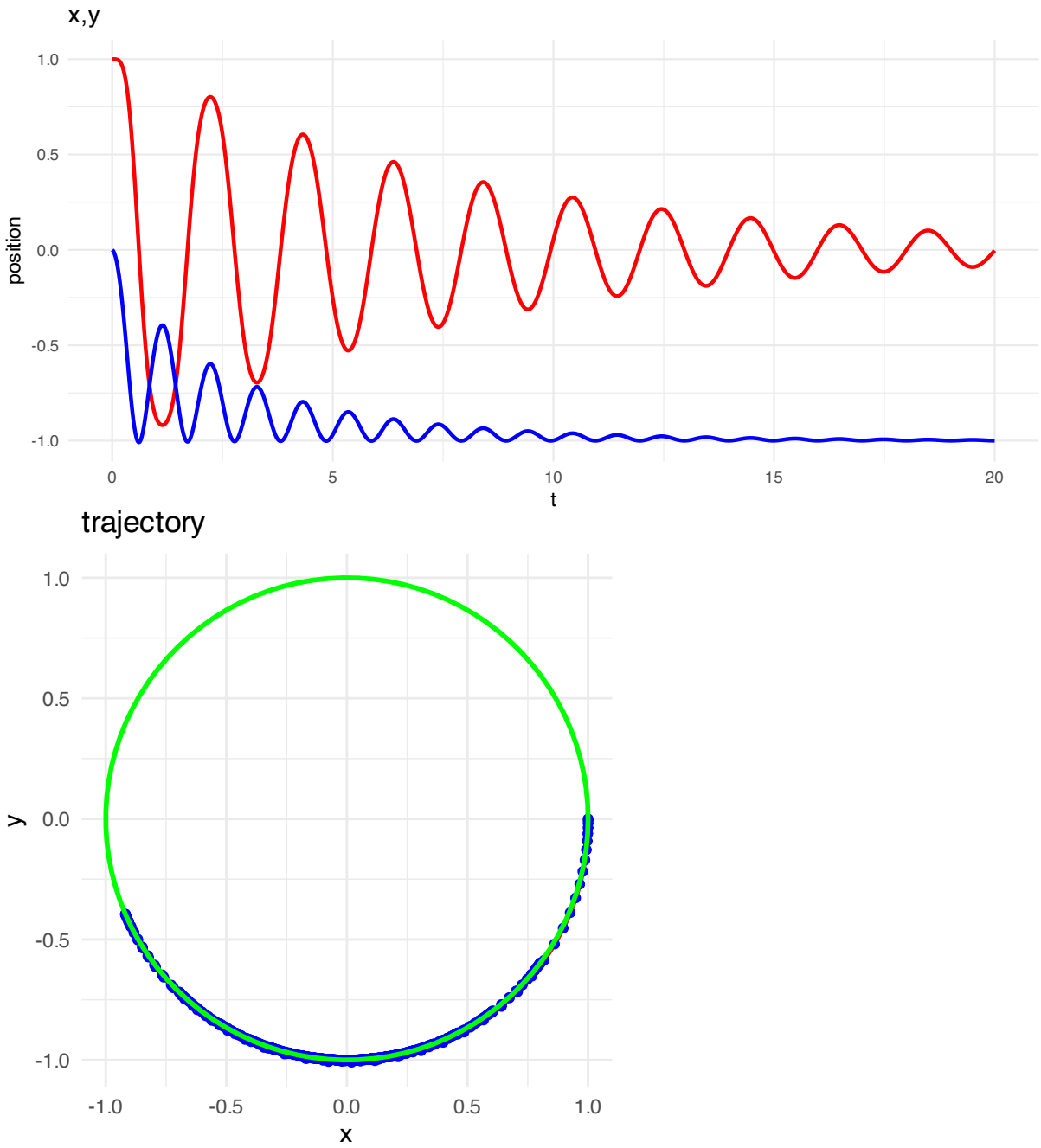


Figure 8.15: **Consistent** initial data $x_0 = 1$, $y_0 = u_0 = v_0 = 0$. Implicit Euler, no stabilization $\Delta t = 0.025$, $t_{\max} = 20$.

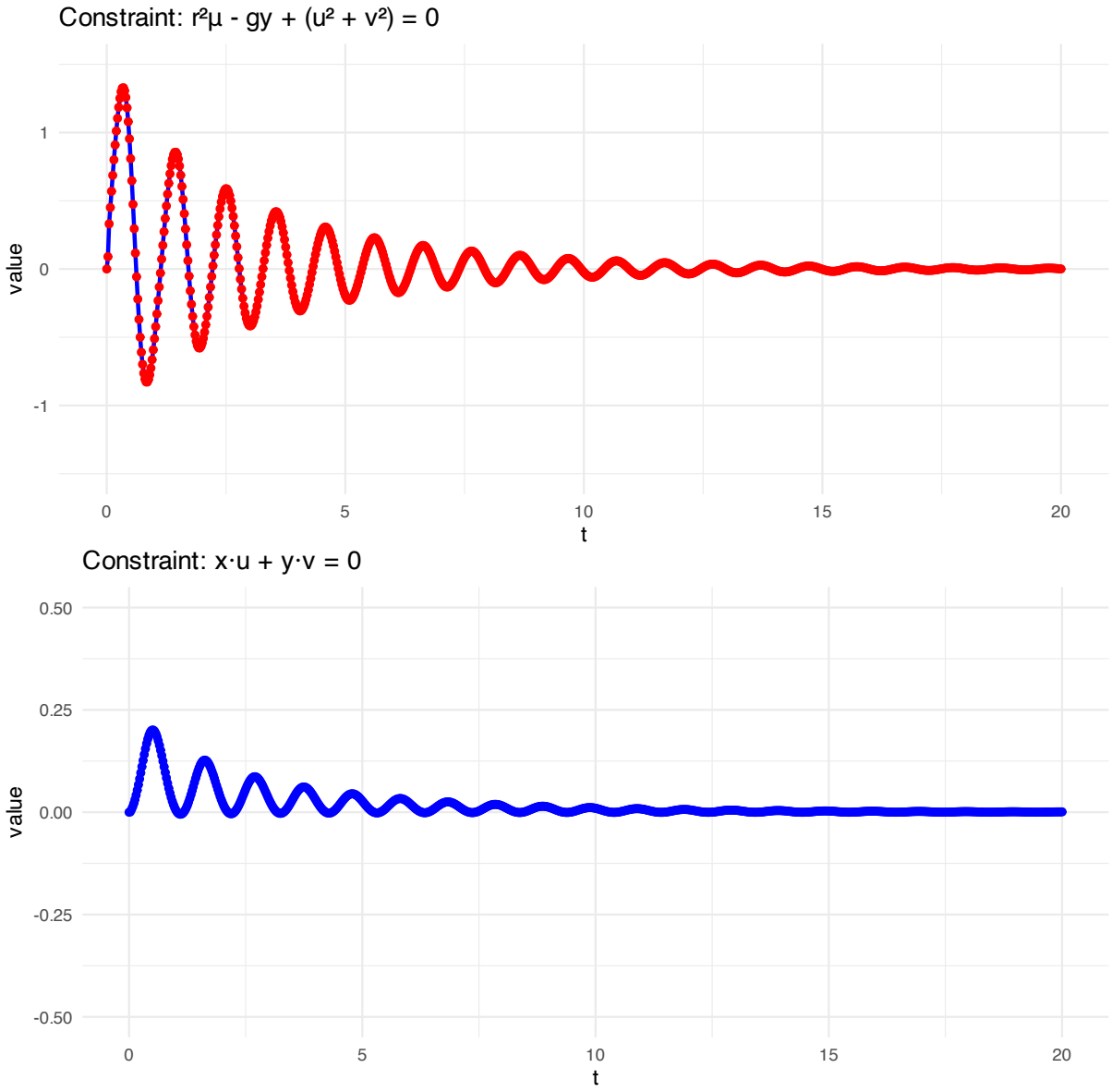


Figure 8.16: **Consistent** initial data $x_0 = 1$, $y_0 = u_0 = v_0 = 0$. Implicit Euler, no stabilization $\Delta t = 0.025$, $t_{\max} = 20$.

$$z'''(t) + az''(t) + bz'(t) + cz(t) = 0$$

satisfy $\lim_{t\to\infty} z(t) = 0$.

8.4.8.1 Crank–Nicolson Results

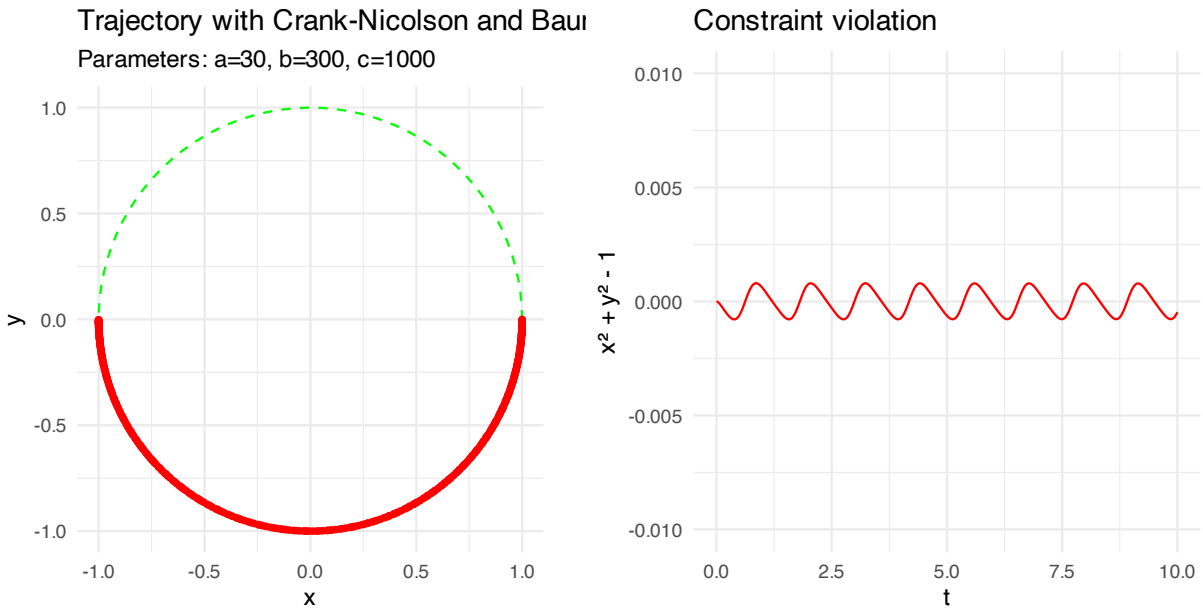


Figure 8.17: Crank–Nicolson integration with third-order Baumgarte stabilization.

Initial **consistent** data and stabilizing values $a = 30$, $b = 300$, $c = 1000$.

8.5 Projection Methods

Another way to stabilize the constraint

$$\boldsymbol{\phi}(\boldsymbol{p}, t) = 0$$

and avoid the drift effect is using the **projection method**. The idea is to firstly reduce the index of the DAE, thus obtaining an ODE and a list of **hidden constraints**. Then the resulting ODE can be numerically integrated, leading to a generic intermediate solution

$$\widetilde{\boldsymbol{p}}(t_n).$$

This temporary solution is projected to the **hidden constraints** in order to minimize the error between the numerical integration solution and the actual system constraints, which is a **nonlinear constrained least squares problem** because of the norm chosen.

It should be noticed that unlike the Baumgarte's stabilization method this technique is useful and particularly effective if you are interested in the **short term** behavior, since even during transients the constraint is always satisfied. Different information from the original and reduced system can be used for various projection methods. For example, we can decide to get an advantage by first projecting the ODE solution

$$\widetilde{\boldsymbol{p}}(t_n)$$

to the **position** constraints and than to the **velocity** constraints.

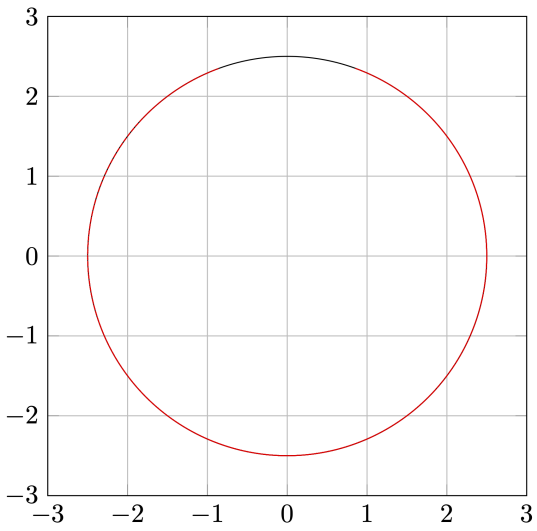


Figure 8.18: Pendulum simulation with Projection method.

▪ **Example 8.2**

Continuing Example 8.1 we want use the Projection technique to stabilize the pendulum. The numerical scheme for integrating the reduced DAE is that in the scheme of DAEs from Lagrangian Mechanics. We want now to project the results of the generic ODE numerical integrator to the following *hidden constraints*,

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = 2(\boldsymbol{p}\cdot\boldsymbol{v}) = 0.$$

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}t^2} = 2(\dot{\boldsymbol{p}}\cdot\boldsymbol{v}) + 2(\boldsymbol{p}\cdot\dot{\boldsymbol{v}}) = 2(\boldsymbol{v}\cdot\boldsymbol{v}) + 2(\boldsymbol{p}\cdot\dot{\boldsymbol{v}}) = \boldsymbol{0}.$$

At each time step, the resulting orthogonal projection to the constraints is equal to

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\|\widetilde{\boldsymbol{p}} - \boldsymbol{p}\|^2 \\ \text{subject to:} \quad & \boldsymbol{\phi}(\boldsymbol{p}, t) = \boldsymbol{p}\cdot\boldsymbol{p} - \ell^2 = 0 \end{aligned}$$

for the position coordinates projection, and

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\|\widetilde{\boldsymbol{v}} - \boldsymbol{v}\|^2 \\ \text{subject to:} \quad & \frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t}(\boldsymbol{p}, t) = 2(\boldsymbol{v}\cdot\boldsymbol{p}) = 0 \end{aligned}$$

for the velocity coordinates projection.

▪

8.5.1 Projection Principles

Projection methods aim to leverage well-established numerical techniques to efficiently solve DAEs while ensuring consistency with their inherent constraints. The core idea is to exploit the robustness and accuracy of existing solvers for ordinary differential equations (ODEs) by transforming the problem into an ODE system with constraints and subsequently applying a projection step to maintain consistency.

8.5.1.1 Key Principles

1. **Utilization of existing ODE Solvers**:
   - Many sophisticated and highly accurate solvers are available for ODEs.
   - By reformulating a DAE as an ODE, one can take advantage of these mature numerical techniques.
2. **Constraint Satisfaction via Projection**:
   - Directly solving a transformed ODE may lead to constraint violation over time due to numerical errors, a phenomenon known as *drift*.
   - To mitigate drift, a projection step is introduced to enforce consistency with the hidden constraints at each step or whenever the drift becomes non-negligible.

Projection Method Workflow

The projection method follows a straightforward two-step paradigm:

1. **ODE Step**: Advance the solution using any standard numerical solver for ODEs.
2. **Projection Step**: Adjust the computed solution by projecting it onto the manifold defined by the constraints, thereby ensuring consistency.

This approach provides a balance between computational efficiency and constraint preservation, making it an attractive method for solving DAEs.

8.5.1.2 Requirements

The projection method relies on the following essential components:

- **Index Reduction**:
  - DAEs often have a higher differential index, which makes them difficult to solve directly.
  - Index reduction techniques are applied to convert the DAE into an ODE system with explicit algebraic constraints.
- **Collection of Constraints and Jacobians**:
  - To perform projection, the constraints and their Jacobians must be determined.
  - This involves symbolic computations to extract hidden constraints and their derivatives.

Thus, while the projection method simplifies numerical integration, it does not eliminate the need for symbolic preprocessing:

- Index reduction typically requires symbolic manipulations proportional to the index of the DAE.
- The projection step involves a smaller symbolic computation effort, primarily related to evaluating constraint Jacobians, whose complexity depends on the chosen projection algorithm.

8.5.2 Formulation

Consider a general DAE system:

$$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{x}', t) = \boldsymbol{0}$$

where $\boldsymbol{x}$ represents the state variables, and $\boldsymbol{F}$ defines the differential-algebraic structure.

To facilitate numerical integration, the system is transformed into an ODE system with explicit constraints using index reduction:

$$\begin{aligned} \boldsymbol{x}' &= \boldsymbol{G}(\boldsymbol{x}, t), && \text{[ODE]} \\ \boldsymbol{0} &= \boldsymbol{H}(\boldsymbol{x}, t), && \text{[Invariants or hidden constraints]} \end{aligned}$$

where:

- $\boldsymbol{G}(\boldsymbol{x}, t)$ defines the transformed r.h.s. of the ODE system.
- $\boldsymbol{H}(\boldsymbol{x}, t)$ represents the constraints that must be satisfied throughout the integration process.

8.5.3 Projected Integration

Step 1: ODE Integration

Using an explicit numerical scheme, such as the explicit Euler method, we compute a preliminary update:

$$\widetilde{\boldsymbol{x}}_{k+1} = \boldsymbol{x}_k + \Delta t\, \boldsymbol{G}(\boldsymbol{x}_k, t_k)$$

where $\widetilde{\boldsymbol{x}}_{k+1}$ is the intermediate solution, which may not satisfy the constraints due to numerical errors.

Step 2: Projection onto the Constraint Manifold

To restore consistency, the next step is to compute the corrected state $\boldsymbol{x}_{k+1}$ by solving the constrained optimization problem:

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{z}) = \frac{1}{2}\|\boldsymbol{z} - \widetilde{\boldsymbol{x}}_{k+1}\|^2 \\ \text{subject to the constraint:} \quad & \boldsymbol{H}(\boldsymbol{z}, t_{k+1}) = \boldsymbol{0}. \end{aligned}$$

then set $\boldsymbol{x}_{k+1} = \boldsymbol{z}$.

This optimization step ensures that $\boldsymbol{x}_{k+1}$ remains on the constraint manifold, effectively eliminating the drift introduced by numerical integration.

8.5.4 Constrained Minimization

In many numerical applications, we encounter the problem of minimizing a function while satisfying a set of constraints. The general form of such a constrained optimization problem is:

$$\text{minimize} \quad f(\boldsymbol{z}) \quad \text{subject to} \quad \boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{0}$$

where:

- $f(\boldsymbol{z})$ is the objective function to be minimized.
- $\boldsymbol{h}(\boldsymbol{z})$ represents the set of constraints that must be satisfied.

Here is shown a sketch in matlab for the solution of constrained minimization problem using function `fmincon`:

```
% Dimensione di z
n = ...;

% objective function f(z)
objective = @(z) ...;

% constraint h(z)
nonlcon = @(z) ...;

% Initial point
z0 = xtilde;

% Solve using fmincon
options = optimoptions(  'fmincon', ...
  'Display',             'iter',  ...
  'Algorithm',           'sqp',   ...
  'MaxIterations',       100,     ...
  'OptimalityTolerance', 1e-8     ...
);

z_sol = fmincon( objective, z0, ...
                 [], [], [], [], [], [], ...
                 nonlcon, options);
```

Solution Using Lagrange Multipliers

To find the solution, we introduce the **Lagrangian function**:

$$\mathcal{L}(z, \lambda) = f(z) - \lambda \cdot h(z)$$

where:

- λ is the vector of **Lagrange multipliers**, which enforces the constraints.

The necessary conditions for a solution (also called the **Karush-Kuhn-Tucker conditions**) correspond to the stationary points of the Lagrangian:

$$\begin{cases} \dfrac{\partial \mathcal{L}(z, \lambda)}{\partial z} = \mathbf{0}, \\ \dfrac{\partial \mathcal{L}(z, \lambda)}{\partial \lambda} = \mathbf{0}. \end{cases}$$

Expanding these derivatives explicitly:

$$\begin{cases} \nabla f(z) - h_z(z)^T \lambda = \mathbf{0}, \\ h(z) = \mathbf{0}. \end{cases}$$

where h_z is the Jacobian matrix of the constraints:

$$h_z = \frac{\partial h(z)}{\partial z}.$$

This system of nonlinear equations must be solved to find the optimal solution (z, λ).

Linearization of the Minimization Problem

Since the nonlinear system above is often difficult to solve directly, we employ a **linearization technique** around a current estimate z. This leads to an iterative solution approach.

By performing a first-order Taylor expansion of the necessary conditions with respect to z, we obtain:

$$\begin{cases} \nabla f(z) + \nabla^2 f(z)\delta z - \boldsymbol{h}_z(z)^T \boldsymbol{\lambda} = \mathcal{O}(\|\delta z\|^2), \\ \boldsymbol{h}(z) + \boldsymbol{h}_z(z)\delta z = \mathcal{O}(\|\delta \boldsymbol{\lambda}\|^2). \end{cases}$$

Ignoring the higher-order terms $\mathcal{O}(\|\delta z\|^2)$, we derive an iterative **Newton-like method** for solving the optimization problem.

Iterative Algorithm for the Constrained Minimization Problem

1. Initialize with an initial guess $z_0 = \widetilde{\boldsymbol{x}}_{k+1}$.

2. Update the solution iteratively:

$$z_{\ell+1} = z_\ell + \delta z, \quad \ell = 0, 1, \ldots \quad \text{(until convergence)}.$$

3. Solve for the update δz by solving the following **linear system**:

$$\begin{pmatrix} \nabla^2 f(z_\ell) & -\boldsymbol{h}_z(z_\ell)^T \\ -\boldsymbol{h}_z(z_\ell) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta z \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -\nabla f(z_\ell) \\ \boldsymbol{h}(z_\ell) \end{pmatrix}$$

This system provides the correction δz for the next iteration step.

Special Case: Quadratic Objective Function

A particularly important case is when the objective function is **quadratic**, given by:

$$f(z) = \frac{1}{2}\|z - \widetilde{\boldsymbol{x}}_{k+1}\|^2.$$

For this choice of $f(z)$, the gradient and Hessian simplify to:

$$\nabla f(z) = z - \widetilde{\boldsymbol{x}}_{k+1}, \qquad \nabla^2 f(z) = \boldsymbol{I}.$$

Thus, the linear system simplifies to:

$$\begin{pmatrix} \boldsymbol{I} & -\boldsymbol{h}_z(z_\ell)^T \\ -\boldsymbol{h}_z(z_\ell) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta z \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{x}}_{k+1} - z_\ell \\ \boldsymbol{h}(z_\ell) \end{pmatrix}$$

This case is particularly useful in applications like **projection methods** for constrained dynamics, where $\widetilde{\boldsymbol{x}}_{k+1}$ represents an unconstrained update that needs to be corrected to satisfy the constraints $\boldsymbol{h}(z) = \boldsymbol{0}$.

A sketch in matlab for the projection solution is shown here:

```
n      = ...; % dimension of the problem
m      = ...; % number of constraints
xt     = ...; % point to be projected
H_fun  = @(z) ...; % The constraints m x 1 matrix
Hz_fun = @(z) ...; % Jacobian of the constraints m x n matrix
z      = xt; % initial point
for k=1:max_iter
  Hz  = Hz_fun( z );
  H   = H_fun( z );
  M   = [ eye(n), Hz.' ; Hz, zeros(n,n) ];
  b   = [ xt - z; H ];
  sol = M \ b;
  dz  = sol(1:n);
  if norm(dz,inf) < tolerance
    break;
  end
```

```
  z = z+dz;
end
```

8.6 Projection Algorithms

8.6.1 Index-1 Projection

Consider an index-1 Differential-Algebraic Equation (DAE) system of the form:

$$\begin{cases} x' = f(x, y, t) \\ 0 = g(x, y, t) \end{cases} \tag{8.10}$$

where:

- x: differential variables
- y: algebraic variables
- $f(x, y, t)$: differential dynamics
- $g(x, y, t) = 0$: algebraic constraints

Reformulate with Dummy Variables by introducing a dummy derivative $y_{\text{dot}}(t)$:

$$\begin{cases} x'(t) = f(x(t), y(t), t) \\ y'(t) = y_{\text{dot}}(t) \\ \quad 0 = h(x, y, y_{\text{dot}}, t) \\ \quad 0 = g(x, y, t) \end{cases}$$

Define:

$$\begin{aligned} h(x, y, y_{\text{dot}}, t) &= \frac{d}{dt} g(x, y, t) \\ &= \frac{\partial g}{\partial x} x' + \frac{\partial g}{\partial y} y' + \frac{\partial g}{\partial t} \\ &= g_x(x, y, t)\, f(x, y, t) + g_y(x, y, t)\, y_{\text{dot}} + g_t(x, y, t) \end{aligned}$$

Using:

$$g_x = \frac{\partial g}{\partial x}, \quad g_y = \frac{\partial g}{\partial y}, \quad g_t = \frac{\partial g}{\partial t}$$

we solve for y_{dot}:

$$y_{\text{dot}} = -g_y(x, y, t)^{-1} \left[g_x(x, y, t)(x, y, t)\, f + g_t(x, y, t) \right]$$

Define:

$$e(x, y, t) = -g_y(x, y, t)^{-1} \left[g_x(x, y, t)\, f(x, y, t) + g_t(x, y, t) \right]$$

and obtain the ODE + constraint system:

$$\begin{cases} x' = f(x, y, t) \\ y' = e(x, y, t) \\ 0 = g(x, y, t) \end{cases}$$

The projection method advances the ODE using any numerical integration scheme (Euler, RK, etc.) and then **projects** onto the constraints (**red part**).

8.6.1.1 Projection Phase

The system is advanced with a time integrator (e.g., Euler, Runge–Kutta), then projected onto the constraint manifold.

E.g. using Explicit Euler step:

$$\begin{cases} \widetilde{\boldsymbol{x}}_{k+1} = \boldsymbol{x}_k + \Delta t\, \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{y}_k, t_k) \\ \widetilde{\boldsymbol{y}}_{k+1} = \boldsymbol{y}_k + \Delta t\, \boldsymbol{e}(\boldsymbol{x}_k, \boldsymbol{y}_k, t_k) \end{cases}$$

The true solution will be

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \widetilde{\boldsymbol{x}}_{k+1} + \delta \boldsymbol{x} \\ \boldsymbol{y}_{k+1} &= \widetilde{\boldsymbol{y}}_{k+1} + \delta \boldsymbol{y} \end{aligned}$$

Index-1 Projection as Minimization

Project the point $(\widetilde{\boldsymbol{x}}_{k+1}, \widetilde{\boldsymbol{y}}_{k+1})$ onto the constraint manifold:

$$\min_{\boldsymbol{x},\boldsymbol{y}} \quad \frac{1}{2}\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1}\|^2 + \frac{1}{2}\|\boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1}\|^2$$

$$\text{subject to} \quad \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) = \boldsymbol{0}$$

Using **Lagrange multipliers**, the candidate solutions are the stationary points of the Lagrangian:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1}\|^2 + \frac{1}{2}\|\boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1}\|^2 - \boldsymbol{\lambda} \cdot \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})$$

which leads to the stationery conditions:

$$\begin{cases} \boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{g}_{\boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})^T \boldsymbol{\lambda} = \boldsymbol{0} \\ \boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{g}_{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})^T \boldsymbol{\lambda} = \boldsymbol{0} \\ \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) = \boldsymbol{0} \end{cases}$$

Newton's Method for Projection

Use Newton (like) iterations to solve the nonlinear system. Set

$$\boldsymbol{x}_{k+1} = \widetilde{\boldsymbol{x}}_{k+1} + \delta \boldsymbol{x} \quad \text{and} \quad \boldsymbol{y}_{k+1} = \widetilde{\boldsymbol{y}}_{k+1} + \delta \boldsymbol{y}$$

after linearization neglecting high order term the Newton like iterative method become:

1. Initialization $\ell \leftarrow 0$

$$\begin{aligned} \boldsymbol{x}^{(\ell)} &= \widetilde{\boldsymbol{x}}_{k+1} \\ \boldsymbol{y}^{(\ell)} &= \widetilde{\boldsymbol{y}}_{k+1} \end{aligned}$$

2. Solve:

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & -\boldsymbol{g}_{\boldsymbol{x}}^T \\ \boldsymbol{0} & \boldsymbol{I} & -\boldsymbol{g}_{\boldsymbol{y}}^T \\ -\boldsymbol{g}_{\boldsymbol{x}} & -\boldsymbol{g}_{\boldsymbol{y}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta \boldsymbol{x} \\ \delta \boldsymbol{y} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{x}^{(\ell)} \\ \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{y}^{(\ell)} \\ \boldsymbol{g}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}, t_{k+1}) \end{pmatrix}$$

3. Update:

$$\begin{aligned} \boldsymbol{x}^{(\ell+1)} &= \boldsymbol{x}^{(\ell)} + \delta \boldsymbol{x}, \\ \boldsymbol{y}^{(\ell+1)} &= \boldsymbol{y}^{(\ell)} + \delta \boldsymbol{y} \end{aligned}$$

4. $\ell \leftarrow \ell + 1$ and repeat step 2 and 3 until convergence.

Index-1 Remarks

For Equation 8.10 **it is not convenient to use projection**, instead solve the ODE

$$x' = F(x, t)$$

where

$$F(x, t) = f(x, y(x, t), t)$$

and $y(x, t)$ is the solution of

$$g(x, y, t) = 0 \qquad (\star)$$

how to solve $(\star)$?
For example using explicit Euler

$$x_{k+1} = x_k + \Delta t f(x_k, y_k, t_k)$$

y_{k+1} is the solution of $c(z) = 0$ where $c(z) = g(x_{k+1}, z, t_{k+1})$ that can be solved using Newton-Raphson method

$$z^{(0)} = y_k, \qquad z^{(\ell+1)} = z^{(\ell)} - \left(\frac{\partial c(z^{(\ell)})}{\partial z}\right)^{-1} c(z^{(\ell)})$$

or better, start with $y_{k+1}^{(0)} = y_k$ and then

$$y_{k+1}^{(\ell+1)} = y_{k+1}^{(\ell)} - g_y(x_{k+1}, y_{k+1}^{(\ell)}, t_{k+1})^{-1} g(x_{k+1}, y_{k+1}^{(\ell)}, t_{k+1})$$

set $y_{k+1} = y_{k+1}^{(L)}$ with L accepted value.

8.6.2 Index-2 Projection

An index-2 differential-algebraic equation (DAE) can be written as:

$$\begin{cases} x' = f(x, y, t) \\ 0 = g(x, t) \end{cases}$$

where:

- x and y are the differential and algebraic variables;
- $f(x, y, t)$ is the right-hand side of the differential equations;
- $g(x, t)$ encodes the algebraic constraints.

8.6.2.1 Constraint Differentiation

To regularize the system, differentiate the constraint and introduce a dummy derivative for y:

$$\begin{cases} x'(t) = f(x(t), y(t), t) \\ y'(t) = y_{\text{dot}}(t) \\ 0 = a(x, y, t) + g_x(x, t) f_y(x, y, t) y_{\text{dot}} \\ 0 = h(x, y, t) \\ 0 = g(x, t) \end{cases}$$

where

- First hidden constraint

$$\begin{aligned}\mathbf{0} &= \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{g}(\boldsymbol{x}, t)\\ &= \boldsymbol{g}_{\boldsymbol{x}}(\boldsymbol{x}, t)\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t) + \boldsymbol{g}_t(\boldsymbol{x}, t)\\ &= \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t)\end{aligned}$$

- Second hidden constraint

$$\begin{aligned}\mathbf{0} &= \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t)\\ &= \underbrace{\boldsymbol{h}_{\boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{y}, t)\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t) + \boldsymbol{h}_t(\boldsymbol{x}, \boldsymbol{y}, t)}_{\boldsymbol{a}(\boldsymbol{x},\boldsymbol{y},t)} + \underbrace{\boldsymbol{h}_{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, t)}_{\boldsymbol{g}_{\boldsymbol{x}}(\boldsymbol{x},t)\boldsymbol{f}_{\boldsymbol{y}}(\boldsymbol{x},\boldsymbol{y},t)}\ \boldsymbol{y}_{\mathrm{dot}}\end{aligned}$$

  Solving for $\boldsymbol{y}_{\mathrm{dot}}$ gives the regularized system:

$$\begin{cases}\boldsymbol{x}'(t) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t)\\ \boldsymbol{y}'(t) = -\left[\boldsymbol{g}_{\boldsymbol{x}}(\boldsymbol{x}, t)\boldsymbol{f}_{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, t)\right]^{-1}\boldsymbol{a}_{\boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{y}, t)\\ \mathbf{0} = \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t)\\ \mathbf{0} = \boldsymbol{g}(\boldsymbol{x}, t)\end{cases}$$

  an ODE with hidden constraints.

8.6.2.2 Numerical Solution

The system is solved using a numerical method (e.g., Euler or Runge-Kutta) followed by a projection step to enforce the constraints.

Explicit Euler Step Example

1. **Advance solution**

$$\begin{cases}\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta t\, \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{y}_k, t_k)\\ \boldsymbol{y}_{k+1} = \boldsymbol{y}_k - \Delta t \left[\boldsymbol{g}_{\boldsymbol{x}}(\boldsymbol{x}_k, t_k)\boldsymbol{f}_{\boldsymbol{y}}(\boldsymbol{x}_k, \boldsymbol{y}_k, t_k)\right]^{-1}\boldsymbol{a}_{\boldsymbol{x}}(\boldsymbol{x}_k, \boldsymbol{y}_k, t_k)\end{cases}$$

2. **Project onto Constraints:**
   Solve the constrained minimization problem:

$$\text{minimize} \qquad \frac{1}{2}\left\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1}\right\|^2 + \frac{1}{2}\left\|\boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1}\right\|^2$$

$$\text{subject to} \qquad \begin{cases}\boldsymbol{g}(\boldsymbol{x}, t_{k+1}) = \mathbf{0}\\ \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) = \mathbf{0}\end{cases}$$

   This is solved using Lagrange multipliers and linearization.

Projection algorithm

Using Lagrange multiplier:

$$\begin{aligned}\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \frac{1}{2}\left\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1}\right\|^2 + \frac{1}{2}\left\|\boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1}\right\|^2\\ - \boldsymbol{\lambda} \cdot \boldsymbol{g}(\boldsymbol{x}, t_{k+1}) - \boldsymbol{\mu} \cdot \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})\end{aligned}$$

or the solution of the non linear system:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} = \boldsymbol{0}.$$

that written explicitly is

$$\begin{cases} \boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{g}_x(\boldsymbol{x}, t_{k+1})^T \boldsymbol{\lambda} - \boldsymbol{h}_x(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})^T \boldsymbol{\mu} = \boldsymbol{0} \\ \boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{h}_y(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})^T \boldsymbol{\mu} = \boldsymbol{0} \\ \boldsymbol{g}(\boldsymbol{x}, t_{k+1}) = \boldsymbol{0} \\ \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) = \boldsymbol{0} \end{cases}$$

after linearization do until convergence ($\ell = 0, 1, \ldots$) the following iterations

1. Initialization $\ell \leftarrow 0$

$$\begin{aligned} \boldsymbol{x}^{(\ell)} &= \widetilde{\boldsymbol{x}}_{k+1}, \\ \boldsymbol{y}^{(\ell)} &= \widetilde{\boldsymbol{y}}_{k+1} \end{aligned}$$

2. Solve

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & -\boldsymbol{g}_x^T & -\boldsymbol{h}_x^T \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & -\boldsymbol{h}_y^T \\ -\boldsymbol{g}_x & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{h}_x & -\boldsymbol{h}_y & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta \boldsymbol{x} \\ \delta \boldsymbol{y} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{x}^{(\ell)} \\ \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{y}^{(\ell)} \\ \boldsymbol{g}(\boldsymbol{x}^{(\ell)}, t_{k+1}) \\ \boldsymbol{h}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}, t_{k+1}) \end{pmatrix}$$

3. Advance:

$$\begin{aligned} \boldsymbol{x}^{(\ell+1)} &= \boldsymbol{x}^{(\ell)} + \delta \boldsymbol{x}, \\ \boldsymbol{y}^{(\ell+1)} &= \boldsymbol{y}^{(\ell)} + \delta \boldsymbol{y}, \end{aligned}$$

4. $\ell \leftarrow \ell + 1$ and repeat step 2 and 3 until convergence.

Approximate Projection algorithm

To speed up computation we can project separately position and velocity solving two consecutive project:

- **x projection:**

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \left\| \boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1} \right\|^2 \\ \text{subject to} \quad & \boldsymbol{g}(\boldsymbol{x}, t_{k+1}) = \boldsymbol{0} \end{aligned}$$

- **y projection:**

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \left\| \boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1} \right\|^2 \\ \text{subject to} \quad & \boldsymbol{h}(\boldsymbol{x}_{k+1}, \boldsymbol{y}, t_{k+1}) = \boldsymbol{0} \end{aligned}$$

Using Lagrange multiplier we obtain the two nonlinear system to be solved

- **x problem:**

$$\begin{cases} \boldsymbol{x} - \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{g}_x(\boldsymbol{x}, t_{k+1})^T \boldsymbol{\lambda} = \boldsymbol{0} \\ \boldsymbol{g}(\boldsymbol{x}, t_{k+1}) = \boldsymbol{0} \end{cases}$$

1. Initialization $\ell \leftarrow 0$

$$\boldsymbol{x}^{(\ell)} = \widetilde{\boldsymbol{x}}_{k+1}$$

2. Solve
$$\begin{pmatrix} \boldsymbol{I} & -\boldsymbol{g}_{\boldsymbol{x}}^T \\ -\boldsymbol{g}_{\boldsymbol{x}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta \boldsymbol{x} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{x}}_{k+1} - \boldsymbol{x}^{(\ell)} \\ \boldsymbol{g}(\boldsymbol{x}^{(\ell)}, t_{k+1}) \end{pmatrix}$$
3. Advance:
$$\boldsymbol{x}^{(\ell+1)} = \boldsymbol{x}^{(\ell)} + \delta \boldsymbol{x}$$
4. $\ell \leftarrow \ell + 1$ and repeat step 2 and 3 until convergence.

- **y problem:**
$$\begin{cases} \boldsymbol{y} - \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{h}_{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1})^T \boldsymbol{\mu} = \boldsymbol{0} \\ \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) = \boldsymbol{0} \end{cases}$$
1. Initialization $\ell \leftarrow 0$
$$\boldsymbol{y}^{(\ell)} = \widetilde{\boldsymbol{y}}_{k+1}$$
2. Solve
$$\begin{pmatrix} \boldsymbol{I} & -\boldsymbol{h}_{\boldsymbol{y}}^T \\ -\boldsymbol{h}_{\boldsymbol{y}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta \boldsymbol{y} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{y}}_{k+1} - \boldsymbol{y}^{(\ell)} \\ \boldsymbol{h}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}, t_{k+1}) \end{pmatrix}$$
3. Advance:
$$\boldsymbol{y}^{(\ell+1)} = \boldsymbol{y}^{(\ell)} + \delta \boldsymbol{y},$$
4. $\ell \leftarrow \ell + 1$ and repeat step 2 and 3 until convergence.

8.6.3 Index-3 Projection

An **Index-3** DAE typically arises in mechanical systems with constraints, such as:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t)\boldsymbol{v}' + \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \boldsymbol{\lambda} = \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ \Phi(\boldsymbol{p}, t) = \boldsymbol{0} \end{cases}$$

where:

- $\boldsymbol{p}$ and $\boldsymbol{v}$ are the position and velocity variables,
- $\boldsymbol{M}$ is the mass matrix,
- Φ represents the constraints,
- $\boldsymbol{\lambda}$ are the Lagrange multipliers.

To handle the higher-index constraints, we introduce dummy variables and differentiate the constraints

$$\begin{cases} \boldsymbol{p}'(t) = \boldsymbol{v}(t), \\ \boldsymbol{v}'(t) = \boldsymbol{v}_{\text{dot}} \\ \quad \text{Index one part (solve for } \boldsymbol{v}_{\text{dot}} \text{ and } \boldsymbol{\lambda}) \\ \boldsymbol{0} = \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t)\boldsymbol{v}_{\text{dot}} + \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \boldsymbol{\lambda} - \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ \boldsymbol{0} = \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)\boldsymbol{v}_{\text{dot}} + \boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) \\ \quad \text{Hidden constraints (projection for } \boldsymbol{p} \text{ and } \boldsymbol{v}) \\ \boldsymbol{0} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t), \\ \boldsymbol{0} = \Phi(\boldsymbol{p}, t) \end{cases}$$

where

$$\begin{aligned} \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t) &= \frac{\mathrm{d}}{\mathrm{d}t}\Phi(\boldsymbol{p}, t) \\ &= \Phi_{\boldsymbol{p}}(\boldsymbol{x}, t)\boldsymbol{v} + \Phi_t(\boldsymbol{p}, t) \end{aligned}$$

$$\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) = \boldsymbol{h}_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, t)\boldsymbol{v} + \boldsymbol{h}_t(\boldsymbol{p}, \boldsymbol{v}, t)$$

By solving for $\boldsymbol{v}_{\text{dot}}$ and $\boldsymbol{\lambda}$ we obtain the reduced ODE system:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{v} \\ \boldsymbol{v}' = \boldsymbol{v}_{\text{dot}}(\boldsymbol{p}, \boldsymbol{v}, t) \end{cases}$$

where $\boldsymbol{v}_{\text{dot}}$ is obtained from $\boldsymbol{p}$ and $\boldsymbol{v}$ by solving the linear system:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ -\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) \end{pmatrix}$$

The invariants

$$\begin{cases} \boldsymbol{0} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t) \\ \boldsymbol{0} = \Phi(\boldsymbol{p}, t) \end{cases}$$

are used in the projection step.

Index-3 Advancement

Projection method advance the ODE using any numerical method (Euler, RK, …) and next project onto the constraints (red and blue part).

E.g. using Explicit Euler step:

$$\begin{cases} \boldsymbol{p}_{k+1} = \boldsymbol{p}_k + \Delta t\, \boldsymbol{v}_k \\ \boldsymbol{v}_{k+1} = \boldsymbol{v}_k + \Delta t\, (\boldsymbol{v}_{\text{dot}})_k \end{cases}$$

where $\boldsymbol{v}_{\text{dot}}$ is obtained from $\boldsymbol{p}_k$ and $\boldsymbol{v}_k$ by solving the linear system:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}_k, \boldsymbol{v}_k, t_k) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}_k, t_k)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}_k, t_k) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{p}_k, \boldsymbol{v}_k, t_k) \\ -\boldsymbol{a}(\boldsymbol{p}_k, \boldsymbol{v}_k, t_k) \end{pmatrix}$$

Index-3 Projection

The projection phase solve the problem

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\left\|\boldsymbol{p} - \widetilde{\boldsymbol{p}}_{k+1}\right\|^2 + \frac{1}{2}\left\|\boldsymbol{v} - \widetilde{\boldsymbol{v}}_{k+1}\right\|^2 \\ &\text{subject to} && \begin{cases} \Phi(\boldsymbol{p}, t_{k+1}) = \boldsymbol{0} \\ \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t_{k+1}) = \boldsymbol{0} \end{cases} \end{aligned}$$

using Lagrange multiplier:

$$\begin{aligned} \mathcal{L}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = &\frac{1}{2}\|\boldsymbol{p} - \widetilde{\boldsymbol{p}}_{k+1}\|^2 + \frac{1}{2}\|\boldsymbol{v} - \widetilde{\boldsymbol{v}}_{k+1}\|^2 \\ &- \boldsymbol{\lambda} \cdot \Phi(\boldsymbol{x}, t_{k+1}) - \boldsymbol{\mu} \cdot \boldsymbol{h}(\boldsymbol{x}, \boldsymbol{y}, t_{k+1}) \end{aligned}$$

or the solution of the non linear system:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{p}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} = \boldsymbol{0}.$$

Use Newton (like) iterations to solve the nonlinear system. Set

$$\boldsymbol{p}_{k+1} = \widetilde{\boldsymbol{p}}_{k+1} + \delta\boldsymbol{p} \quad \text{and} \quad \boldsymbol{v}_{k+1} = \widetilde{\boldsymbol{v}}_{k+1} + \delta\boldsymbol{v}$$

after linearization neglecting high order term the Newton like iterative method become:

1. Initialization $\ell \leftarrow 1$

$$\boldsymbol{p}^{(\ell)} = \widetilde{\boldsymbol{p}}_{k+1} \quad \text{and} \quad \boldsymbol{v}^{(\ell)} = \widetilde{\boldsymbol{v}}_{k+1}$$

2. Solve:

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & -\boldsymbol{\Phi}_{\boldsymbol{p}}^T & -\boldsymbol{h}_{\boldsymbol{p}}^T \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & -\boldsymbol{h}_{\boldsymbol{v}}^T \\ -\boldsymbol{\Phi}_{\boldsymbol{p}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{h}_{\boldsymbol{p}} & -\boldsymbol{h}_{\boldsymbol{v}} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta\boldsymbol{p} \\ \delta\boldsymbol{v} \\ \boldsymbol{\lambda} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{p}}_{k+1} - \boldsymbol{p}^{(\ell)} \\ \widetilde{\boldsymbol{v}}_{k+1} - \boldsymbol{v}^{(\ell)} \\ \boldsymbol{\Phi}(\boldsymbol{p}^{(\ell)}, t_{k+1}) \\ \boldsymbol{h}(\boldsymbol{p}^{(\ell)}, \boldsymbol{v}^{(\ell)}, t_{k+1}) \end{pmatrix}$$

3. Update:

$$\begin{aligned} \boldsymbol{p}^{(\ell+1)} &= \boldsymbol{p}^{(\ell)} + \delta\boldsymbol{p}, \\ \boldsymbol{v}^{(\ell+1)} &= \boldsymbol{v}^{(\ell)} + \delta\boldsymbol{v} \end{aligned}$$

4. Repeat step 2 and 3 until convergence.

Approximate Projection

Position Projection

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\left\|\boldsymbol{p} - \widetilde{\boldsymbol{p}}_{k+1}\right\|^2 \\ &\text{subject to} && \boldsymbol{\Phi}(\boldsymbol{p}, t_{k+1}) = \boldsymbol{0} \end{aligned}$$

using Lagrange multiplier:

$$\mathcal{L}(\boldsymbol{p}, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{p} - \widetilde{\boldsymbol{p}}_{k+1}\|^2 - \boldsymbol{\lambda} \cdot \boldsymbol{\Phi}(\boldsymbol{p}, t_{k+1})$$

or find the solution of the non linear system:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{p}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} = \boldsymbol{0}.$$

Velocity Projection

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\left\|\boldsymbol{v} - \widetilde{\boldsymbol{v}}_{k+1}\right\|^2 \\ &\text{subject to} && \boldsymbol{h}(\boldsymbol{p}_{k+1}, \boldsymbol{v}, t_{k+1}) = \boldsymbol{0} \end{aligned}$$

using Lagrange multiplier:

$$\mathcal{L}(\boldsymbol{v}, \boldsymbol{\mu}) = \frac{1}{2}|\boldsymbol{v} - \widetilde{\boldsymbol{v}}_{k+1}\|^2 - \boldsymbol{\mu} \cdot \boldsymbol{h}(\boldsymbol{p}_{k+1}, \boldsymbol{v}, t_{k+1})$$

or the solution of the non linear system:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{v}} = \boldsymbol{0}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} = \boldsymbol{0}.$$

$$\begin{aligned} \boldsymbol{p}^{(0)} &= \widetilde{\boldsymbol{p}}_{k+1}, & \boldsymbol{p}^{(\ell+1)} &= \boldsymbol{p}^{(\ell)} + \delta\boldsymbol{p}, \\ \boldsymbol{v}^{(0)} &= \widetilde{\boldsymbol{v}}_{k+1}, & \boldsymbol{v}^{(\ell+1)} &= \boldsymbol{v}^{(\ell)} + \delta\boldsymbol{v}, \end{aligned}$$

$$\begin{pmatrix} \boldsymbol{I} & -\boldsymbol{\Phi}_{\boldsymbol{p}}^T \\ -\boldsymbol{\Phi}_{\boldsymbol{p}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta\boldsymbol{p} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{p}}_{k+1} - \boldsymbol{p}^{(\ell)} \\ \boldsymbol{\Phi}(\boldsymbol{p}^{(\ell)}, t_{k+1}) \end{pmatrix}$$

$$\begin{pmatrix} \boldsymbol{I} & -\boldsymbol{h}_{\boldsymbol{v}}^T \\ -\boldsymbol{h}_{\boldsymbol{v}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \delta\boldsymbol{v} \\ \boldsymbol{\mu} \end{pmatrix} = \begin{pmatrix} \widetilde{\boldsymbol{v}}_{k+1} - \boldsymbol{v}^{(\ell)} \\ \boldsymbol{h}(\boldsymbol{p}_{k+1}, \boldsymbol{v}^{(\ell)}, t_{k+1}) \end{pmatrix}$$

where $\boldsymbol{h}_{\boldsymbol{v}} = \boldsymbol{h}_{\boldsymbol{v}}(\boldsymbol{p}_{k+1}, \boldsymbol{v}^{(\ell)}, t_{k+1})$ and $\boldsymbol{\Phi}_{\boldsymbol{p}} = \boldsymbol{\Phi}(\boldsymbol{p}^{(\ell)}, t_{k+1})$.

8.7 Implicit Projection

8.7.1 Index-1 DAE

Consider a semi-explicit DAE (with index > 1):

$$\begin{cases} z' = f(z, w, t) \\ 0 = g(z, w, t) \end{cases}$$

This system can be transformed into an index-1 DAE by index reduction, along with invariants:

$$\text{Index-1 DAE: } \begin{cases} x' = f(x, y, t) \\ 0 = a(x, y, t) \end{cases} \qquad \underbrace{b(x, y, t) = 0}_{\text{invariants}}$$

The index-1 system is then solved as an implicitly defined ODE:

$$x' = F(x, t)$$

where $F(x, t) = f(x, y(x, t), t)$ and $y(x, t)$ is the solution to the equation $b(x, y, t) = 0$.

8.7.1.1 Explicit Advancement

Explicit methods, such as the explicit Euler method, do not require complex handling. The update step is given by:

$$x_{k+1} = x_k + \Delta t F(x_k, t_k)$$

The evaluation of $F(x_k, t_k)$ proceeds as follows:

1. Solve $h(y) = a(x_k, y, t_k) = 0$ to compute y_k.
2. Compute $F(x_k, t_k) = f(x_k, y_k, t_k)$.

8.7.1.2 Implicit Advancement

For implicit methods, such as implicit Euler, the update step is:

$$x_{k+1} = x_k + \Delta t F(x_{k+1}, t_{k+1})$$

This requires solving the implicit equation:

$$G(z) = z - \Delta t F(z, t_{k+1}) - x_k = 0$$

The solution of $G(z) = 0$ requires an iterative method such as Newton-Raphson:

$$z^{(\ell+1)} = z^{(\ell)} - \left(\frac{\partial G(z^{(\ell)})}{\partial z} \right)^{-1} G(z^{(\ell)})$$

This method requires the Jacobian of $F(x, t)$:

$$\frac{\partial G(z)}{\partial z} = I - \Delta t \frac{\partial F(z, t_{k+1})}{\partial z}$$

8.7.1.3 RHS Jacobian

Given the system

$$F(x, t) = f(x, y(x, t), t), \qquad a(x, y(x, t), t) = 0,$$

we can derive the following expressions for the Jacobian:

$$\begin{cases} \dfrac{\partial F(x, t)}{\partial x} = \dfrac{\partial f(x, y, t)}{\partial x} + \dfrac{\partial f(x, y, t)}{\partial y} \dfrac{\partial y(x, t)}{\partial x} \\ 0 = \dfrac{\partial a(x, y, t)}{\partial x} + \dfrac{\partial a(x, y, t)}{\partial y} \dfrac{\partial y(x, t)}{\partial x} \end{cases}$$

From this, we can express the derivative of $\boldsymbol{y}(\boldsymbol{x}, t)$ with respect to $\boldsymbol{x}$ as:

$$\frac{\partial \boldsymbol{y}(\boldsymbol{x}, t)}{\partial \boldsymbol{x}} = -\boldsymbol{a}_y(\boldsymbol{x}, \boldsymbol{y}, t)^{-1} \boldsymbol{a}_x(\boldsymbol{x}, \boldsymbol{y}, t),$$

and the Jacobian of $\boldsymbol{F}(\boldsymbol{x}, t)$ becomes:

$$\boxed{\frac{\partial \boldsymbol{F}(\boldsymbol{x}, t)}{\partial \boldsymbol{x}} = \boldsymbol{f}_x(\boldsymbol{x}, \boldsymbol{y}, t) - \boldsymbol{f}_y(\boldsymbol{x}, \boldsymbol{y}, t) \boldsymbol{a}_y(\boldsymbol{x}, \boldsymbol{y}, t)^{-1} \boldsymbol{a}_x(\boldsymbol{x}, \boldsymbol{y}, t)}$$

8.7.1.4 Jacobian Computation

In practice, the Jacobian should be computed as follows:

1. Start with the system:

$$\boldsymbol{F}(\boldsymbol{x}, t) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, t), t), \qquad \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}, t), t) = \boldsymbol{0}.$$

2. Solve for $\boldsymbol{Z}$ by solving the linear system:

$$\boldsymbol{a}_y(\boldsymbol{x}, \boldsymbol{y}, t) \boldsymbol{Z} = -\boldsymbol{a}_x(\boldsymbol{x}, \boldsymbol{y}, t)$$

3. Compute the Jacobian:

$$\frac{\partial \boldsymbol{F}(\boldsymbol{x}, t)}{\partial \boldsymbol{x}} = \boldsymbol{f}_x(\boldsymbol{x}, \boldsymbol{y}, t) + \boldsymbol{f}_y(\boldsymbol{x}, \boldsymbol{y}, t) \boldsymbol{Z}$$

8.7.2 Index-3 DAEs

We begin with the following system of differential-algebraic equations (DAEs):

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) \boldsymbol{v}' + \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \boldsymbol{\lambda} = \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ \boldsymbol{\Phi}(\boldsymbol{p}, t) = \boldsymbol{0} \end{cases}$$

ODE Form with Invariants

We aim to transform this system into a set of ordinary differential equations (ODEs) with implicit right-hand sides (RHS) and invariants:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{v} \\ \boldsymbol{v}' = \boldsymbol{v}_{\text{dot}}(\boldsymbol{p}, \boldsymbol{v}, t) \end{cases} \qquad \begin{cases} \boldsymbol{0} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t) = \dfrac{\mathrm{d}}{\mathrm{d}t} \boldsymbol{\Phi}(\boldsymbol{p}, t) \\ \boldsymbol{0} = \boldsymbol{\Phi}(\boldsymbol{p}, t) \end{cases}$$

Here, $\boldsymbol{v}_{\text{dot}}$ is derived from the system by solving the following linear system:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ -\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) \end{pmatrix}$$

Additionally, we define:

$$\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) = \boldsymbol{h}_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, t) \boldsymbol{v} + \boldsymbol{h}_t(\boldsymbol{p}, \boldsymbol{v}, t)$$

Defining Functions for RHS and Invariants

Let us introduce the following functions to represent the system:

- **Function F:**

$$\boldsymbol{F}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) = \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) \boldsymbol{v}_{\text{dot}} + \boldsymbol{\Phi}_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \boldsymbol{\lambda} - \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t)$$

- **Function G:**

$$G(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) = \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)\boldsymbol{v}_{\text{dot}} + \boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t)$$

Thus, the system can be expressed as:

$$F(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}(\boldsymbol{p}, \boldsymbol{v}, t), \boldsymbol{\lambda}(\boldsymbol{p}, \boldsymbol{v}, t), t) = \boldsymbol{0}$$

$$G(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}(\boldsymbol{p}, \boldsymbol{v}, t), t) = \boldsymbol{0}$$

Jacobian Derivatives of Functions F and G

Taking derivatives of F and G with respect to $\boldsymbol{p}$, we obtain the following system of equations:

$$F_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) + F_{\boldsymbol{v}_{\text{dot}}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t)\frac{\partial \boldsymbol{v}_{\text{dot}}}{\partial \boldsymbol{p}} + F_{\boldsymbol{\lambda}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t)\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{p}} = \boldsymbol{0}$$

$$G_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) + G_{\boldsymbol{v}_{\text{dot}}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t)\frac{\partial \boldsymbol{v}_{\text{dot}}}{\partial \boldsymbol{p}} = \boldsymbol{0}$$

Expressions for the Jacobians of Functions F and G

From the definition of the functions, we have the following expressions for the Jacobian derivatives:

$$F_{\boldsymbol{v}_{\text{dot}}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) = \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t)$$

$$F_{\boldsymbol{\lambda}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) = \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T$$

$$G_{\boldsymbol{v}_{\text{dot}}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) = \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)$$

Final System of Equations for the Jacobians

The system of Jacobian equations can now be written as:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \frac{\partial}{\partial \boldsymbol{p}} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -F_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) \\ -G_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) \end{pmatrix}$$

Similarly, for derivatives with respect to $\boldsymbol{v}$:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \frac{\partial}{\partial \boldsymbol{v}} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -F_{\boldsymbol{v}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) \\ -G_{\boldsymbol{v}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) \end{pmatrix}$$

Final Transformed Index-3 DAE

Thus, the transformed system of index-3 DAEs is:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{v} \\ \boldsymbol{v}' = \boldsymbol{v}_{\text{dot}}(\boldsymbol{p}, \boldsymbol{v}, t) \end{cases} \qquad \begin{cases} \boldsymbol{0} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{v}, t) = \dfrac{\mathrm{d}}{\mathrm{d}t}\Phi(\boldsymbol{p}, t) \\ \boldsymbol{0} = \Phi(\boldsymbol{p}, t) \end{cases}$$

The RHS and its Jacobians are obtained from:

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{p}, \boldsymbol{v}, t) \\ -\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{v}, t) \end{pmatrix}$$

$$\begin{pmatrix} \boldsymbol{M}(\boldsymbol{p}, \boldsymbol{v}, t) & \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t)^T \\ \Phi_{\boldsymbol{p}}(\boldsymbol{p}, t) & \boldsymbol{0} \end{pmatrix} \frac{\partial}{\partial \boldsymbol{p}} \begin{pmatrix} \boldsymbol{v}_{\text{dot}} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} -F_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, \boldsymbol{\lambda}, t) \\ -G_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{v}_{\text{dot}}, t) \end{pmatrix}$$

8.8 Coordinate Partitioning

8.8.1 Independent Coordinates

Consider redundant coordinate pendulum DAE

$$\begin{aligned} u' &= \mu x \\ v' &= \mu y - g \\ x' &= u \\ y' &= v \\ x^2 + y^2 &= r^2 \end{aligned}$$

Solve y as a function of x (use negative branch)

$$y(x) = -\sqrt{r^2 - x^2}$$

constraint can be eliminated from the DAE by introducing $y \equiv y(x)$.

8.8.2 Constraint Propagation

$$\begin{aligned} u' &= \mu x \\ v' &= \mu y(x) - g \\ x' &= u \\ y' &= v \\ y(x) &= -\sqrt{r^2 - x^2} \end{aligned}$$

Deriving constraint $x^2 + y^2 - r^2 = 0$ derivative of $y'(x)$ is deduced

$$x\dot{x} + y\dot{y} = 0 \quad \Longrightarrow \quad \dot{y} = -x\dot{x}/y \quad \Longrightarrow \quad v = -xu/y$$

so that

$$\dot{y} = v(x, u) = -\frac{xu}{y(x)}$$

Step 2b

Substitute $\dot{y}$ with $v(x, u)$

$$\begin{aligned} u' &= \mu x \\ v' &= \mu y(x) - g \\ x' &= u \\ v(x, u) &= -\frac{xu}{y(x)} \\ y(x) &= -\sqrt{r^2 - x^2} \end{aligned}$$

Step 2c

Deriving two time the constraint $x^2 + y^2 - r^2 = 0$ derivative $\ddot{y} = \dot{v}$ is deduced:

$$xu' + y\dot{v} + x'u + y'v = 0 \quad \Longrightarrow$$

$$\mu(x^2 + y^2) - gy + (u^2 + v^2) = 0$$

so that μ can be written as a function of x and u

$$\mu(x, u) = \frac{gy(x) - (u^2 + v(x, u)^2)}{r^2}$$

thus $\mu = \mu(x, u)$ and

$$v'(x, u) = \mu(x, u)y(x) - g$$

can be eliminated as redundant algebraic equation.

Step 2d

Substitute $\mu = \mu(x, u)$

$$\begin{aligned} u' &= \mu(x, u)x \\ x' &= u \\ \mu(x, u) &= \frac{g y(x) - (u^2 + v(x, u)^2)}{r^2} \\ v(x, u) &= -\frac{x u}{y(x)} \\ y(x) &= -\sqrt{r^2 - x^2} \end{aligned}$$

in practice an ODE with 2 equations and $\mu(x, u)$ computed recursively.

8.8.3 Collatz

Using Collatz explicit integrator

$$\begin{aligned} u_{k+1/2} &= u_k + (h/2)\mu(x_k, u_k)x_k \\ x_{k+1/2} &= x_k + (h/2)u_k \\ u_{k+1} &= u_k + h\,\mu(x_{k+1/2}, u_{k+1/2})x_{k+1/2} \\ x_{k+1} &= x_k + h\,u_{k+1/2} \end{aligned}$$

where $\mu(x, u)$

$$\begin{aligned} y_k &= -\sqrt{r^2 - x_k^2} \\ v_k &= -\frac{x_k u_k}{y_k} \\ \mu(x_k, u_k) &= \frac{g y_k - (u_k^2 + {v_k}^2)}{r^2} \end{aligned}$$

$T_{\max} = 10$, $x(0) = 0.9$, $u(0) = 0$. The remaining value $y(0) = \sqrt{r^2 - x(0)^2}$ and $v(0) = -x(0)u(0)/y(0)$.

8.8.4 Fourth-Order Runge–Kutta

$$\begin{aligned} K_1 &= h u_k & L_1 &= h\mu\,(x_k, u_k)\,x_k \\ K_2 &= h\left(u_k + \frac{L_1}{2}\right) & L_2 &= h\mu\left(x_k + \frac{K_1}{2}, u_k + \frac{L_1}{2}\right)\left(x_k + \frac{K_1}{2}\right) \\ K_3 &= h\left(u_k + \frac{L_2}{2}\right) & L_3 &= h\mu\left(x_k + \frac{K_2}{2}, u_k + \frac{L_2}{2}\right)\left(x_k + \frac{K_2}{2}\right) \\ K_4 &= h\,(u_k + L_3) & L_4 &= h\mu\,(x_k + K_3, u_k + L_3)\,(x_k + K_3) \end{aligned}$$

$$\begin{aligned} x_{k+1} &= x_k + \frac{K_1 + 2K_2 + 2K_3 + K_4}{6} \\ u_{k+1} &= u_k + \frac{L_1 + 2L_2 + 2L_3 + L_4}{6} \end{aligned}$$

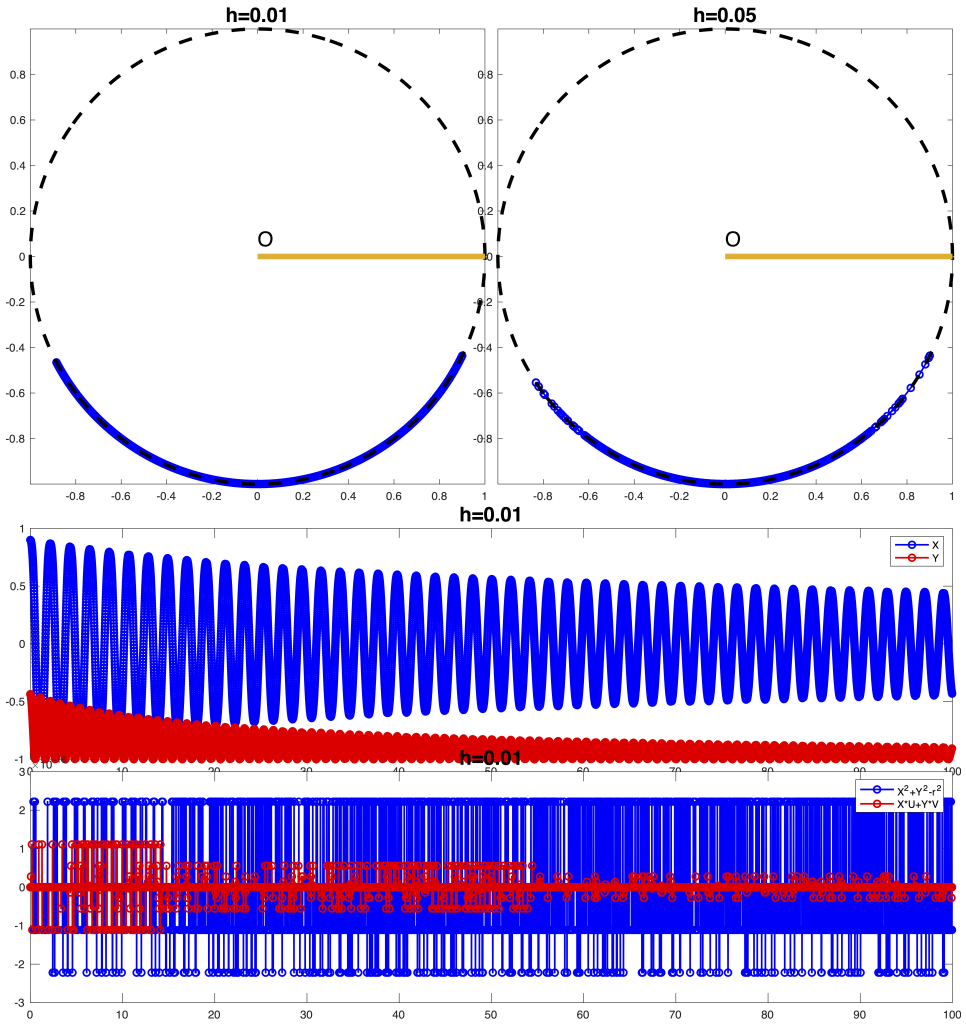


Figure 8.19: Trajectory and constraint violation:

$T_{\max} = 10$, $x(0) = 0.9$, $u(0) = 0$. The remaining value $y(0) = \sqrt{r^2 - x(0)^2}$ and $v(0) = -x(0)u(0)/y(0)$.

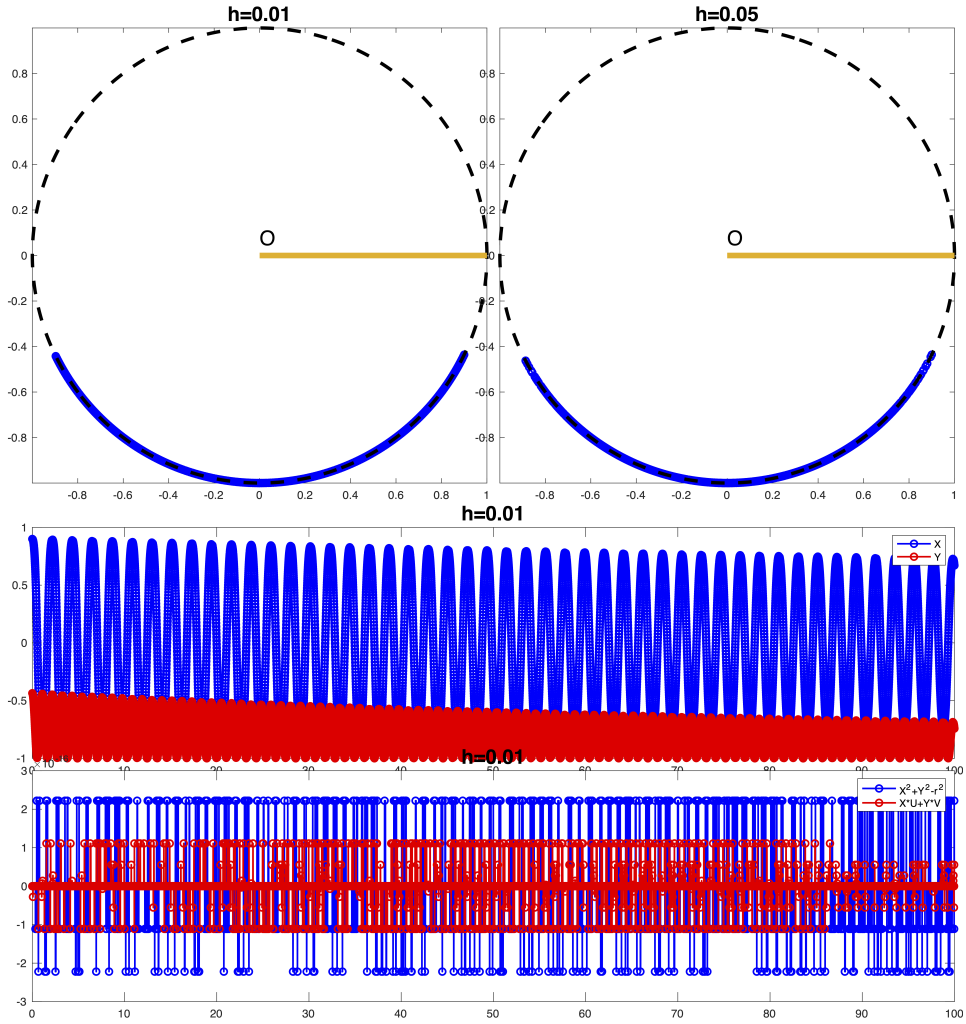


Figure 8.20: Trajectory and constraint violation:

8.9 Order Reduction in DAE Integration

A time integrator of classical order p on ODEs can converge with a smaller order on a DAE. This **order reduction** is distinct from index reduction: index reduction changes the equations, whereas order reduction describes the observed convergence rate of a discretization.

8.9.1 Main mechanisms

Order reduction is most likely when

- algebraic or multiplier variables have a lower stage order than the differential variables;
- the DAE has index greater than one;
- the initial data satisfy the visible constraints but not all compatibility conditions;
- boundary data or external forcing are insufficiently smooth;
- a stiff transient makes the asymptotic error expansion nonuniform;
- projection or index reduction introduces an error larger than the local truncation error of the base method.

For Runge–Kutta methods, classical order alone is therefore not enough. Stage order and the method's DAE-specific order conditions control how well the internal stages satisfy the constraints.

8.9.2 How to detect it

Run the same problem with h, $h/2$, and $h/4$ and estimate

$$p_{\text{obs}} = \log_2 \frac{\|\boldsymbol{e}_h\|}{\|\boldsymbol{e}_{h/2}\|}.$$

Measure errors separately for differential variables, algebraic variables, multipliers, and constraint residuals. A single combined norm can conceal that only one component has lost order. Repeat the experiment after tightening the nonlinear-solver tolerance so that iteration error is not mistaken for discretization error.

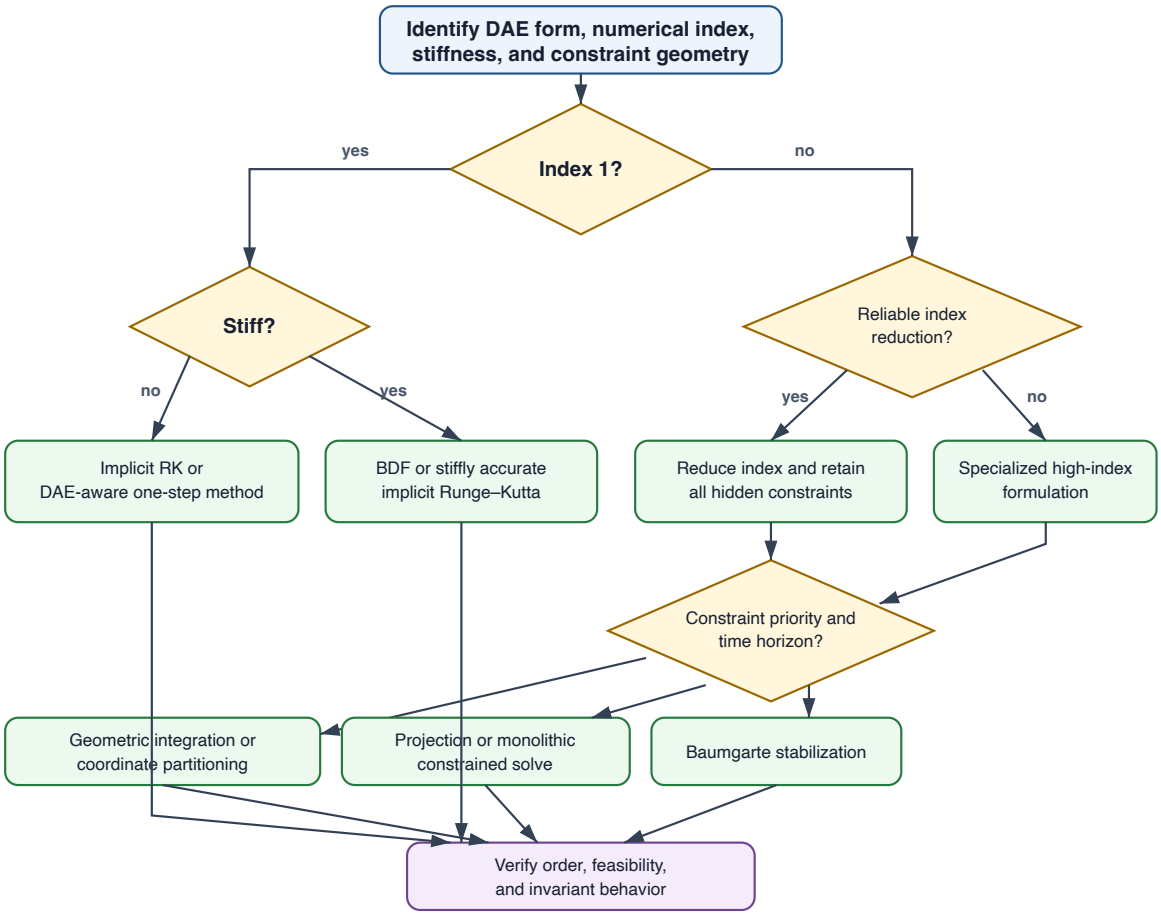
8.9.3 Mitigation

1. Compute fully consistent initial values, including hidden constraints.
2. Prefer methods designed for the DAE class, such as stiffly accurate implicit Runge–Kutta formulas or appropriate BDF methods.
3. Reduce the index while retaining and monitoring all invariants.
4. Match projection accuracy to the local error target of the time integrator.
5. Use a start-up procedure of sufficient order after events or discontinuous inputs.
6. Confirm the claimed order by a mesh-refinement table rather than by the solver's status flag.

> ⚠ What an adaptive controller sees
>
> An ODE error estimator can control its own embedded difference while missing the dominant algebraic or constraint error. For a DAE, acceptance tests should include both the temporal error estimate and scaled constraint residuals.

8.10 Choosing a Numerical DAE Method

Method selection begins with structure, not with a solver name. The diagram summarizes a practical first decision; every branch still requires consistent initialization and residual monitoring.



{#fig-dae-method-selection width="100%",bg="white"}

8.10.1 Decision table

Structure	Good first choice	Main check
index-1, non-stiff	implicit RK or Rosenbrock-type DAE method	local error and algebraic residual
index-1, stiff	BDF or L-stable implicit RK	nonlinear convergence and stiff decay
reducible high index	index reduction plus projection	hidden constraints and observed order
constrained mechanics, long time	SHAKE/RATTLE-type or partitioned geometric method	constraints, energy, momentum
moderate drift acceptable	Baumgarte stabilization	parameter sensitivity and artificial damping
exact pointwise feasibility	projection or monolithic constrained solve	projection conditioning
changing constraint rank	event handling and rank-revealing linear algebra	rank and consistency after events

No branch removes the need for a convergence study. Stiffness, index, and constraint geometry can change along one trajectory, so production codes should expose diagnostics rather than silently switching interpretations.

8.11 Examples

An index-two Van der Pol oscillator

The Van der Pol equation

$$x'' - \mu(1 - x^2)x' + x = 0$$

represents a classic example of a nonlinear oscillator with nonlinear damping, widely used in the study of dynamical systems and electrical circuits.

8.11.0.1 Stiff DAE Form

To transform the equation into an index-2 DAE (Differential-Algebraic Equation) system @Mazzia2003, we introduce:

1. The variable $y = x'$ to obtain a first-order system
2. An additional state z

The resulting DAE is:

$$\begin{cases} x' = y \\ y' = \mu(1 - x^2)y - x \\ 0 = x^2 - z^2 + 5 \end{cases}$$

Semi-Explicit Form

For the previous DAE system introduce dummy variables y_{dot}, z_{dot} and the system decomposes into differential (blue) and algebraic (red) components:

$$\begin{cases} x' = y \\ y' = y_{\text{dot}} \\ z' = z_{\text{dot}} \end{cases} \oplus \begin{cases} 0 = y_{\text{dot}} - \mu(1 - x^2)y + x & \text{(Constraint 1)} \\ 0 = x^2 - z^2 + 5 & \text{(Constraint 2)} \end{cases}$$

For this DAE we have:

- **Differential variables**: x, y, z.
- **Algebraic variables**: y_{dot}, z_{dot}.

Reduction Procedure

We attempt to solve for y_{dot}, z_{dot}, but the system is not yet solvable explicitly. We proceed by **differentiating**.

Step 1: Differentiate algebraic constraints We now differentiate the constraints to reduce the index. Here we want to differentiate **only** the algebraic equation that can reduce the index and do not introduce new differential equation. Here we want to avoid to introduce derivative of the algebraic variable unless is strictly necessary.

We construct a dependency table to identify which constraints to differentiate:

	y_{dot}	z_{dot}
(Constraint 1)	1	0
(Constraint 2)	0	0

a 0 means the the function do not depend on the variable on the column. A 1 means dependence. Last constraint is candidate for differentiation as it doesn't introduce new algebraic variables. Differentiate thus the last constraints and substitute known variable in differential form using the ODE part

$$(\text{Constraint 2})' = 2xx' - 2zz' = 2xy - 2zz_{\text{dot}}$$

so that the DAE become:

$$\begin{cases} x' = y \\ y' = y_{\text{dot}} \\ z' = z_{\text{dot}} \end{cases} \bigoplus \begin{cases} 0 = y_{\text{dot}} - \mu(1-x^2)y + x & \text{(Constraint 1)} \\ 0 = 2xy - 2zz_{\text{dot}} & \text{(Constraint 2)}' \\ 0 = x^2 - z^2 + 5 & \text{(Constraint 2)} \end{cases}$$

The red algebraic equations are the new algebraic equation after the derivation. The green algebraic equation is the hidden constraints that is removed from the DAE system but that is satisfied by choosing the right initial conditions.

The red part cannot determine the algebraic variable and must be differentiated again.

Step 2: Differentiate (again) the algebraic constraints To find the right candidate(s) here we write a table of the dependencies

	y_{dot}	z_{dot}
(Constraint 1)	1	0
(Constraint 2)$'$	0	1

The table shows that the nonlinear system is **structurally** non singular and algebraic constraints can be solved. From (Constraint 2)$'$ solve for z_{dot} and from (Constraint 1) solve for y_{dot}

$$z_{\text{dot}} = \frac{xy}{z}, \qquad y_{\text{dot}} = \mu(1-x^2)y - x$$

ODE Form

Thus DAE now can be reduced to an ODE and can be written as:

$$\begin{cases} x' = y \\ y' = y_{\text{dot}} = \mu(1-x^2)y - x \\ z' = z_{\text{dot}} = \dfrac{xy}{z} \end{cases} \bigoplus \quad x^2 - z^2 + 5 = 0, \quad \text{(Constraint 1)}$$

8.11.0.2 Baumgarte Stabilization

Baumgarte stabilization is a classical technique used to prevent this drift. Instead of enforcing a constraint like:

$$z(t) = 0$$

we instead modify it to a differential equation:

$$\frac{dz}{dt} + \frac{1}{\tau} z = 0$$

This form ensures that any initial deviation from the constraint decays exponentially with a rate controlled by the parameter τ. The smaller τ is, the stronger the correction applied.

The exact solution is:

$$z(t) = z_0 \, e^{-t/\tau}$$

In the following example, we simulate the stabilized equation for different values of τ and visualize how quickly the constraint error decays.

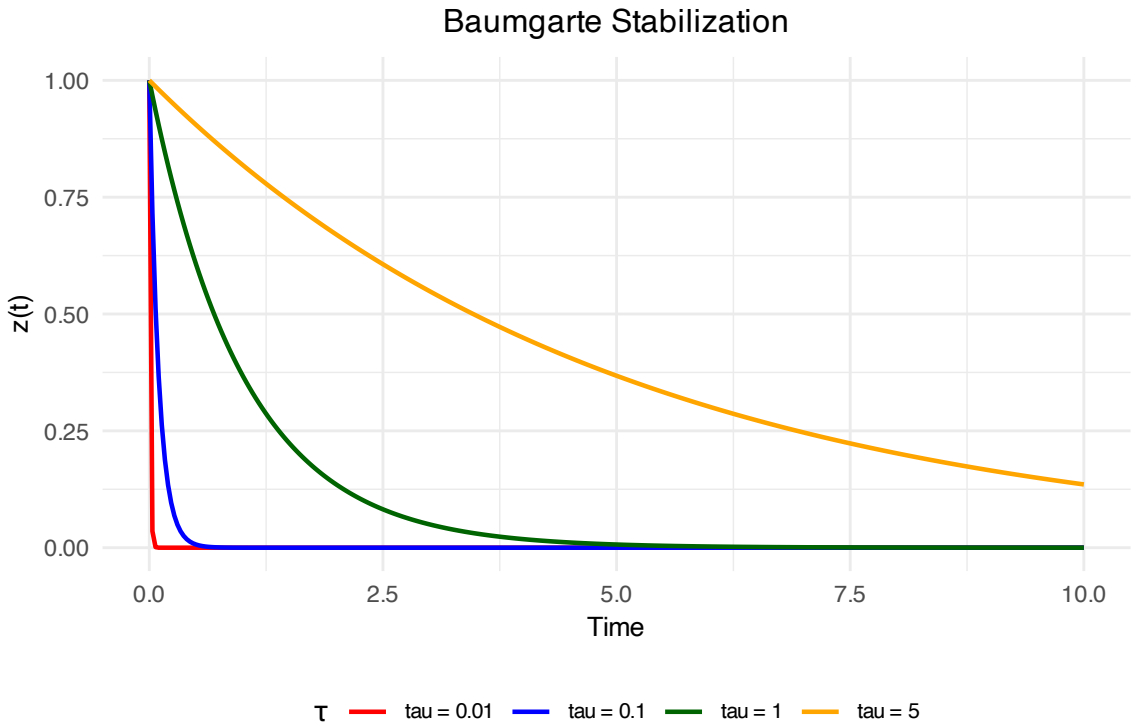


Figure 8.21: Decay of the constraint error for different stabilization parameters.

Impact of the stabilization parameter τ.

The plot clearly shows the impact of the stabilization parameter τ:

- **Small τ (e.g., 0.01)**: The solution quickly decays to zero, meaning constraint violations are rapidly corrected. However, this can introduce numerical stiffness.
- **Large τ (e.g., 5)**: The decay is slow, allowing more deviation from the constraint over time.

This trade-off is at the heart of Baumgarte stabilization: small τ improves accuracy but increases stiffness and numerical cost, while large τ is easier to integrate but less effective at controlling constraint drift.

In real DAE systems, choosing τ often involves tuning based on system stiffness and desired constraint enforcement.

For this DAE the hidden constraints can be used to stabilize the algebraic constraint (Constraint 1)

$$(\text{Constraint 1})_{\text{stab}} = (\text{Constraint 1})' + \frac{1}{\tau}(\text{Constraint 1}) = 0$$

and substituting (diving also by 2)

$$(\text{Constraint 1})_{\text{stab}} = x\,y - z\,z_{\text{dot}} + \frac{1}{\tau}\frac{x^2 - z^2 + 5}{2}$$

and using the stabilized constraints

$$\begin{cases} x' = y \\ y' = \mu(1 - x^2)y - x \\ z' = \dfrac{xy}{z} + \dfrac{1}{\tau}\dfrac{x^2 - z^2 + 5}{2z} \end{cases} \quad \bigoplus \quad x^2 - z^2 + 5 = 0, \quad \text{(Constraint 1)}$$

8.11.0.3 Numerical Examples

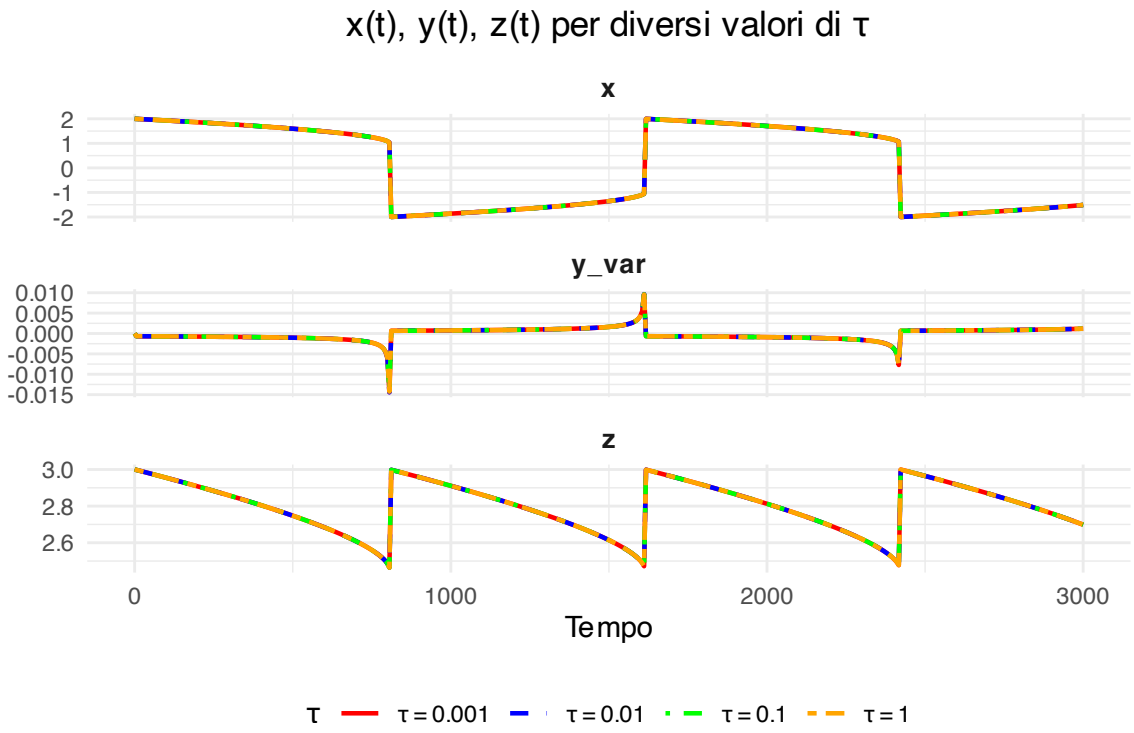


Figure 8.22: Numerical solution and residuals for the stabilized DAE.

IV Optimal Control

9 Numerical Methods for Optimal Control 187

9.1 Reference optimal control problem
9.2 Optimize then discretize or discretize then optimize
9.3 Indirect methods
9.4 Direct control parameterization
9.5 Direct transcription
9.6 A complete trapezoidal transcription
9.7 Time scaling and free terminal time
9.8 Control, endpoint, and path constraints
9.9 Sparse nonlinear programming structure
9.10 Scaling, initialization, and continuation
9.11 Mesh refinement and error estimation
9.12 Diagnostics and verification
9.13 Choosing a numerical formulation
9.14 Implementation checklist
9.15 Exercises

Numerical solution of optimal control problems: indirect and direct methods, transcription, and the nonlinear programs they produce. The theory is in the part of the same name of the course notes.

9. Numerical Methods for Optimal Control

The necessary conditions developed in the preceding chapters transform an optimal control problem into differential equations, algebraic conditions, and endpoint constraints. A numerical method must still choose a finite representation of the trajectories and solve the resulting nonlinear system or optimization problem.

This chapter develops the two principal approaches:

- **indirect methods**, which discretize the optimality system derived from Pontryagin's principle;
- **direct methods**, which discretize the state and control first and solve a finite-dimensional nonlinear programming problem.

Dynamic programming provides a third viewpoint through the Hamilton–Jacobi–Bellman equation. It is conceptually fundamental, but a state-space grid becomes prohibitively expensive as the dimension grows. The discussion below therefore concentrates on shooting, transcription, and collocation methods used for trajectory optimization [@betts2010practical].

Learning objectives

After studying this chapter, the reader should be able to

- distinguish indirect shooting from direct transcription;
- formulate single- and multiple-shooting residuals;
- derive trapezoidal and Hermite–Simpson defect constraints;
- explain how sensitivities and adjoints provide derivatives;
- transcribe endpoint, control, path, and free-time constraints;
- recognize the sparse structure of the resulting NLP;
- assess mesh accuracy, feasibility, and first-order optimality.

9.1 Reference optimal control problem

Consider the Bolza problem

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{u},\boldsymbol{p},T}{\text{minimize}} \quad & J = \Phi\big(\boldsymbol{x}(0), \boldsymbol{x}(T), \boldsymbol{p}, T\big) + \int_0^T L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t)\,\mathrm{d}t, \\ \text{subject to} \quad & \boldsymbol{x}'(t) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t), \\ & \boldsymbol{b}\big(\boldsymbol{x}(0), \boldsymbol{x}(T), \boldsymbol{p}, T\big) = \boldsymbol{0}, \\ & \boldsymbol{c}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) \le \boldsymbol{0}, \\ & \boldsymbol{u}_{\min} \le \boldsymbol{u}(t) \le \boldsymbol{u}_{\max}. \end{aligned} \tag{9.1}$$

The vector $\boldsymbol{p}$ contains constant decision parameters and T may be fixed or free. Equality path constraints can be included by replacing selected components of $\boldsymbol{c}$ with equalities.

A numerical approximation should be judged on four distinct properties:

1. **dynamic consistency**: the state approximately satisfies the ODE;
2. **feasibility**: endpoint and path constraints are satisfied;
3. **optimality**: the discrete KKT residual is small;
4. **mesh accuracy**: the result is stable under refinement.

A small NLP residual alone does not guarantee that the continuous-time OCP has been accurately approximated.

9.2 Optimize then discretize or discretize then optimize

An indirect method first derives the continuous necessary conditions and then discretizes them. This is often described as **optimize then discretize**. A direct method first creates a finite-dimensional approximation and then derives the KKT conditions of that approximation: **discretize then optimize**.

Approach	Unknowns	Equations solved	Main strength	Main difficulty
Indirect shooting	missing state/costate data, switching times, parameters	boundary residuals of the PMP system	high local accuracy and physical interpretation	sensitive initialization and switching structure
Direct single shooting	control parameters and static parameters	NLP with states eliminated by integration	few optimization variables	dense, ill-conditioned sensitivities
Multiple shooting	controls, parameters, and states at subinterval boundaries	sparse NLP with continuity constraints	improved robustness	more variables and constraints

Approach	Unknowns	Equations solved	Main strength	Main difficulty
Direct collocation	state and control values at mesh/collocation points	sparse NLP with defect constraints	robust handling of constraints	mesh design and transcription error

For a consistent discretization, the discrete adjoints of a direct method normally approach the continuous costates as the mesh is refined. On a coarse mesh, however, the two optimality systems need not coincide exactly.

9.3 Indirect methods

With the Hamiltonian convention

$$H = L + \boldsymbol{\lambda}^{\mathsf{T}} \boldsymbol{f},$$

an interior normal extremal satisfies

$$\begin{aligned} \boldsymbol{x}' &= H_{\boldsymbol{\lambda}} = \boldsymbol{f}, \\ \boldsymbol{\lambda}' &= -H_{\boldsymbol{x}}, \\ \boldsymbol{0} &= H_{\boldsymbol{u}}, \end{aligned} \tag{9.2}$$

together with endpoint and transversality conditions. Control bounds replace $H_{\boldsymbol{u}} = \boldsymbol{0}$ by pointwise minimization of the Hamiltonian.

After eliminating the control when possible, the state and costate equations form a two-point boundary-value problem. Shooting solves this BVP by converting the missing boundary data into finite-dimensional unknowns.

9.3.1 Single shooting

Assume for simplicity that $\boldsymbol{x}(0) = \boldsymbol{x}_0$ is fixed and that the unknown initial costate is $\boldsymbol{\nu} = \boldsymbol{\lambda}(0)$. For every trial $\boldsymbol{\nu}$, integrate the canonical system from 0 to T and evaluate the terminal residual

$$\boldsymbol{R}(\boldsymbol{\nu}) = \boldsymbol{r}\big(\boldsymbol{x}(T;\boldsymbol{\nu}), \boldsymbol{\lambda}(T;\boldsymbol{\nu})\big). \tag{9.3}$$

The shooting problem is

$$\boldsymbol{R}(\boldsymbol{\nu}) = \boldsymbol{0}.$$

Newton's method requires

$$\boldsymbol{R}_{\boldsymbol{\nu}}(\boldsymbol{\nu}_k)\,\delta\boldsymbol{\nu}_k = -\boldsymbol{R}(\boldsymbol{\nu}_k), \qquad \boldsymbol{\nu}_{k+1} = \boldsymbol{\nu}_k + \delta\boldsymbol{\nu}_k.$$

The Jacobian may be computed by finite differences, automatic differentiation, or variational equations. If $\boldsymbol{z} = (\boldsymbol{x}^{\mathsf{T}}, \boldsymbol{\lambda}^{\mathsf{T}})^{\mathsf{T}}$ satisfies

$$\boldsymbol{z}' = \boldsymbol{F}(\boldsymbol{z}, t),$$

then the sensitivity matrix

$$\boldsymbol{S}(t) = \frac{\partial \boldsymbol{z}(t)}{\partial \boldsymbol{\nu}}$$

satisfies

$$S' = F_z(z,t)S, \qquad S(0) = \begin{bmatrix} 0 \\ I \end{bmatrix}. \tag{9.4}$$

The terminal value $S(T)$ supplies the shooting Jacobian by the chain rule.

Single shooting is compact and can be very accurate near a known extremal. It becomes fragile when unstable dynamics amplify small errors in the guessed costate, when the horizon is long, or when the switching structure is unknown.

> **A converged shooting root is not automatically an optimum**
>
> Shooting enforces first-order necessary conditions. The computed extremal may be a maximum, a saddle, an abnormal solution, or a member of a nonunique family. Curvature, neighboring extremals, and the original objective must still be examined.

9.3.2 Multiple shooting

Partition the time interval,

$$0 = t_0 < t_1 < \cdots < t_N = T,$$

and introduce an independent initial value z_i for each subinterval. Let

$$\varphi_i(z_i)$$

denote the numerical flow from t_i to t_{i+1}. Continuity is imposed by

$$z_{i+1} - \varphi_i(z_i) = 0, \qquad i = 0, \dots, N-1. \tag{9.5}$$

Perturbations are prevented from growing across the entire horizon before a correction is made. The price is a larger nonlinear system, but its Jacobian has a sparse block-banded structure. Multiple shooting is therefore often more robust than single shooting for unstable or long-horizon problems.

Unknown switching times can be added as decision variables. Each smooth arc is integrated separately and matching, switching, and transversality conditions are imposed at the interfaces.

9.4 Direct control parameterization

The simplest direct method represents the control in a finite basis,

$$u(t;a) = \sum_{j=1}^{M} a_j \psi_j(t), \tag{9.6}$$

and integrates the state equation for every parameter vector a. The state is thereby eliminated, producing a nonlinear program in a and any static parameters:

$$\min_{a,p,T} \widehat{J}(a,p,T) \quad \text{subject to} \quad \widehat{b} = 0, \qquad \widehat{c} \le 0.$$

Piecewise-constant and piecewise-linear controls are common. The approach is also called **direct single shooting**.

Advantages include a small decision vector and exact dynamic feasibility up to the ODE integration tolerance. Disadvantages include dense sensitivities, poor behavior for unstable systems, and difficulty enforcing path constraints between sampled times.

9.4.1 Forward sensitivities

For a parameter a_j, define

$$S_j(t) = \frac{\partial \boldsymbol{x}(t;\boldsymbol{a})}{\partial a_j}.$$

Differentiating the state equation gives

$$S_j' = \boldsymbol{f}_{\boldsymbol{x}} S_j + \boldsymbol{f}_{\boldsymbol{u}} \frac{\partial \boldsymbol{u}}{\partial a_j}, \qquad S_j(0) = \frac{\partial \boldsymbol{x}_0}{\partial a_j}. \tag{9.7}$$

Forward sensitivities are efficient when the number of decision parameters is small. Their cost grows roughly in proportion to that number.

9.4.2 Adjoint gradients

When the number of control parameters is large but the objective is scalar, an adjoint calculation can compute the complete objective gradient at a cost largely independent of the number of parameters. The continuous adjoint resembles the costate equation, while a **discrete adjoint** differentiates the actual time-stepping scheme.

For implementation and derivative verification, it is essential to state which gradient is being used:

- the continuous adjoint followed by discretization;
- the exact derivative of the discrete integrator;
- automatic differentiation through the integrator.

The last two are consistent with the discrete NLP. A continuous adjoint may introduce a mismatch unless the state and adjoint discretizations are chosen compatibly.

9.5 Direct transcription

Direct transcription retains discrete states as optimization variables. Choose mesh points

$$0 = t_0 < t_1 < \cdots < t_N = T, \qquad h_k = t_{k+1} - t_k,$$

and introduce

$$\boldsymbol{X}_k \approx \boldsymbol{x}(t_k), \qquad \boldsymbol{U}_k \approx \boldsymbol{u}(t_k).$$

The integral is replaced by quadrature, and the ODE is replaced by algebraic **defect constraints**. The resulting NLP has the schematic form

$$\begin{aligned} \min_{\boldsymbol{X},\boldsymbol{U},\boldsymbol{p},T} \quad & J_h(\boldsymbol{X},\boldsymbol{U},\boldsymbol{p},T), \\ \text{subject to} \quad & \boldsymbol{D}_k(\boldsymbol{X}_k,\boldsymbol{X}_{k+1},\boldsymbol{U}_k,\boldsymbol{U}_{k+1},\boldsymbol{p},T) = \boldsymbol{0}, \\ & \boldsymbol{b}_h(\boldsymbol{X}_0,\boldsymbol{X}_N,\boldsymbol{p},T) = \boldsymbol{0}, \\ & \boldsymbol{c}_h(\boldsymbol{X},\boldsymbol{U},\boldsymbol{p},T) \le \boldsymbol{0}. \end{aligned} \tag{9.8}$$

Each defect involves only neighboring mesh variables. Consequently, the Jacobian is sparse even though the number of variables may be large.

9.5.1 Euler transcription

The forward-Euler defect is

$$\boldsymbol{D}_k = \boldsymbol{X}_{k+1} - \boldsymbol{X}_k - h_k \boldsymbol{f}(\boldsymbol{X}_k, \boldsymbol{U}_k, \boldsymbol{p}, t_k). \tag{9.9}$$

This first-order formula is useful pedagogically but usually too inaccurate for serious trajectory optimization. Its discretization error can bias the computed control and objective.

9.5.2 Trapezoidal collocation

The trapezoidal defect is

$$\boldsymbol{D}_k = \boldsymbol{X}_{k+1} - \boldsymbol{X}_k - \frac{h_k}{2}\left[\boldsymbol{f}(\boldsymbol{X}_k, \boldsymbol{U}_k, t_k) + \boldsymbol{f}(\boldsymbol{X}_{k+1}, \boldsymbol{U}_{k+1}, t_{k+1})\right]. \tag{9.10}$$

The running cost is approximated consistently by

$$\int_{t_k}^{t_{k+1}} L \, \mathrm{d}t \approx \frac{h_k}{2}\left[L(\boldsymbol{X}_k, \boldsymbol{U}_k, t_k) + L(\boldsymbol{X}_{k+1}, \boldsymbol{U}_{k+1}, t_{k+1})\right].$$

The method is second-order accurate for sufficiently smooth solutions.

9.5.3 Hermite–Simpson collocation

Let $t_{k+\frac{1}{2}} = (t_k + t_{k+1})/2$ and define

$$\begin{aligned} \boldsymbol{f}_k &= \boldsymbol{f}(\boldsymbol{X}_k, \boldsymbol{U}_k, t_k), \\ \boldsymbol{f}_{k+1} &= \boldsymbol{f}(\boldsymbol{X}_{k+1}, \boldsymbol{U}_{k+1}, t_{k+1}), \\ \boldsymbol{X}_{k+\frac{1}{2}} &= \frac{1}{2}(\boldsymbol{X}_k + \boldsymbol{X}_{k+1}) + \frac{h_k}{8}(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}). \end{aligned} \tag{9.11}$$

With a midpoint control $\boldsymbol{U}_{k+\frac{1}{2}}$, the Simpson defect is

$$\boldsymbol{D}_k = \boldsymbol{X}_{k+1} - \boldsymbol{X}_k - \frac{h_k}{6}\left(\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1}\right). \tag{9.12}$$

For smooth solutions this gives substantially higher accuracy than Euler or trapezoidal transcription at comparable mesh sizes. Discontinuities and switches should be placed at mesh boundaries; otherwise high-order polynomial interpolation can smear the switching structure.

9.5.4 General collocation and Runge–Kutta schemes

Implicit Runge–Kutta collocation introduces stage states and enforces the dynamics at collocation points. Gauss schemes provide high order, Radau schemes include one endpoint, and Lobatto schemes include both endpoints. Pseudospectral methods use global polynomial approximations and are highly accurate for smooth trajectories, but local mesh methods are usually easier to refine near switches and active path constraints.

9.6 A complete trapezoidal transcription

Consider the minimum-energy transfer

$$\begin{aligned} \min_{x,v,u} \quad & \frac{1}{2}\int_0^1 u(t)^2\,\mathrm{d}t, \\ \text{subject to} \quad & x' = v, \qquad v' = u, \\ & x(0) = 0, \quad v(0) = 0, \quad x(1) = 1, \quad v(1) = 0. \end{aligned} \tag{9.13}$$

On a uniform mesh with $h = 1/N$, the decision variables are

$$\boldsymbol{w} = \{X_k, V_k, U_k\}_{k=0}^{N}.$$

The trapezoidal objective is

$$J_h(\boldsymbol{w}) = \frac{h}{4}\sum_{k=0}^{N-1}\left(U_k^2 + U_{k+1}^2\right), \tag{9.14}$$

and the defects are

$$\begin{aligned} X_{k+1} - X_k - \frac{h}{2}(V_k + V_{k+1}) = 0, \\ V_{k+1} - V_k - \frac{h}{2}(U_k + U_{k+1}) = 0. \end{aligned} \tag{9.15}$$

Together with the four endpoint conditions, this is a sparse convex quadratic program. The continuous solution is

$$x^\star(t) = 3t^2 - 2t^3, \qquad v^\star(t) = 6t - 6t^2, \qquad u^\star(t) = 6 - 12t. \tag{9.16}$$

This example exposes the main ingredients of direct transcription without nonlinear complications. It also provides an exact reference for checking mesh convergence, objective error, and the signs of discrete multipliers.

9.7 Time scaling and free terminal time

For a free terminal time, introduce $\tau \in [0,1]$ and set $t = T\tau$. Then

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\tau} = T\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, T\tau), \tag{9.17}$$

and

$$\int_0^T L\,\mathrm{d}t = T\int_0^1 L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, T\tau)\,\mathrm{d}\tau.$$

The normalized mesh is fixed while T becomes an NLP variable. Bounds $T_{\min} \le T \le T_{\max}$ are often useful safeguards. The derivatives with respect to T must include its effect on the dynamics, running cost, endpoint terms, and explicit time dependence.

9.8 Control, endpoint, and path constraints

Control bounds become ordinary variable bounds at control nodes. Endpoint constraints are imposed directly on $\boldsymbol{X}_0$ and $\boldsymbol{X}_N$. Path constraints are commonly imposed at every collocation point:

$$\boldsymbol{c}(\boldsymbol{X}_{k,j}, \boldsymbol{U}_{k,j}, \boldsymbol{p}, t_{k,j}) \le \boldsymbol{0}.$$

Checking constraints only at mesh nodes can miss violations between nodes. After solving, reconstruct the polynomial trajectory and evaluate the path constraints on a denser verification grid.

An active path constraint may create entry, boundary, and exit arcs. Mesh points should be added near the junctions. Complementarity information from the NLP multipliers helps identify the active intervals, but isolated multiplier spikes on a coarse mesh should not be overinterpreted.

Integral constraints can be handled either by quadrature in the NLP or by the auxiliary-state transformation developed in the preceding chapter. The auxiliary-state form is often convenient because it converts the requirement into a terminal equality and exposes an associated multiplier.

9.9 Sparse nonlinear programming structure

A direct transcription may contain thousands of variables, but each defect depends only on a small local subset. Sparse SQP and primal-dual interior-point methods exploit this structure [@wachter2006implementation].

The NLP Jacobian has a block-banded dynamics portion plus rows for endpoint and global integral constraints. The Lagrangian Hessian is also sparse for a local transcription. Eliminating all states before optimization destroys much of this sparsity, which explains why a larger multiple-shooting or collocation formulation can be faster than a smaller dense single-shooting problem.

Derivative options include:

- exact analytical Jacobians;
- automatic differentiation;
- sparse finite differences using a known sparsity pattern;
- matrix-free products for very large systems.

Finite differences are useful for verification but are often too noisy and expensive as the primary derivative source.

9.10 Scaling, initialization, and continuation

9.10.1 Scaling

States, controls, time, objective terms, and constraints should have comparable scaled magnitudes. A useful nondimensionalization is

$$\widehat{x}_i = \frac{x_i - x_{i,\mathrm{ref}}}{s_{x_i}}, \qquad \widehat{u}_j = \frac{u_j - u_{j,\mathrm{ref}}}{s_{u_j}},$$

with characteristic scales s_{x_i} and s_{u_j}. Poor scaling can make a well-posed OCP appear numerically singular.

9.10.2 Initial guess

A direct method does not require a feasible initial trajectory, but a dynamically plausible guess is valuable. Common strategies are:

- interpolate endpoint states and use a neutral control;
- simulate a feedback controller;

- solve a relaxed or shorter-horizon problem;
- warm-start from a neighboring parameter value;
- solve on a coarse mesh and interpolate to a finer mesh.

9.10.3 Continuation

Introduce difficult features gradually. For a parameter γ, solve a sequence

$$\gamma_0, \gamma_1, \dots, \gamma_M,$$

using each solution to initialize the next. Continuation can increase weights, tighten path constraints, reduce regularization, extend the horizon, or move the endpoints to their desired values.

9.11 Mesh refinement and error estimation

A converged NLP on a fixed mesh is only the solution of the discrete problem. The continuous residual is

$$\boldsymbol{r}(t) = \widetilde{\boldsymbol{x}}'(t) - \boldsymbol{f}(\widetilde{\boldsymbol{x}}(t), \widetilde{\boldsymbol{u}}(t), t), \tag{9.18}$$

where tildes denote reconstructed trajectories. Evaluate $\boldsymbol{r}$ at points not used to impose the defects.

Two main refinement strategies are:

- h-**refinement**: subdivide intervals with large residuals;
- p-**refinement**: increase the polynomial degree on smooth intervals.

An hp strategy uses high order on smooth arcs and small intervals near switches, corners, or active-set changes.

A practical mesh-convergence study records

$$J_h, \qquad \max_t \|\boldsymbol{r}(t)\|, \qquad \max_t \|[\boldsymbol{c}(t)]_+\|, \qquad \|\boldsymbol{b}\|,$$

and compares the state, control, switching times, and multipliers after refinement. Reported digits that change materially under refinement are not numerically justified.

9.12 Diagnostics and verification

The final report should include more than the solver's success flag.

9.12.1 Primal feasibility

Check endpoint constraints, defect norms, control bounds, path constraints on a dense grid, and any integral constraints.

9.12.2 Discrete optimality

Inspect the scaled KKT residual of the NLP. For a direct method, the relevant stationarity equations are those of the discretized problem.

9.12.3 Dynamic verification

Reintegrate the original ODE with the computed control using tighter tolerances. Compare the resulting terminal state with the transcribed state. This test is especially important for coarse collocation meshes.

9.12.4 Adjoint consistency

Compare discrete NLP multipliers with the expected costate structure. After accounting for sign convention and scaling, the discrepancy should decrease under mesh refinement.

9.12.5 Sensitivity and initialization

Resolve the problem from several initial guesses or through different continuation paths. Different local solutions are expected for a nonconvex OCP; agreement is useful evidence but not a proof of global optimality.

> **!** Three independent tolerances
>
> ODE integration error, mesh transcription error, and NLP termination error are different quantities. Tightening the NLP tolerance cannot repair a coarse mesh, and refining the mesh cannot compensate for inaccurate derivatives.

9.13 Choosing a numerical formulation

Problem characteristic	Recommended starting point
Smooth solution and a reliable costate guess	indirect single shooting
Unstable or long-horizon canonical dynamics	multiple shooting
Few control parameters and inexpensive simulation	direct single shooting
Path constraints or control bounds	direct collocation
Large sparse trajectory problem	sparse collocation with SQP or interior point
Bang–bang control with known switches	multiphase shooting or collocation with switch times
Unknown switching structure	coarse direct transcription followed by structure detection
High accuracy on smooth arcs	high-order Runge–Kutta or pseudospectral collocation

A productive hybrid workflow is often:

1. solve a coarse direct transcription from a simple initial guess;
2. inspect active constraints and switching structure;
3. refine the direct mesh;
4. optionally use the reconstructed costate to initialize an indirect method for high-accuracy local refinement.

9.14 Implementation checklist

1. State the sign convention and continuous OCP precisely.
2. Decide whether the solution structure is known well enough for shooting.
3. Choose control parameterization or collocation order.
4. Scale variables, objective terms, and constraints.
5. Supply sparse derivatives and verify them directionally.
6. Use a feasible or dynamically plausible initial guess when possible.
7. Solve on a coarse mesh before refinement.
8. Verify constraints between nodes.
9. Reintegrate the original dynamics independently.
10. Report mesh, feasibility, KKT, and sensitivity evidence.

9.15 Exercises

Hints: for the fundamental group, begin from the integral state equation and differentiate the shooting map; for the computational group, keep mesh and solver tolerances fixed while changing one discretization choice; for the advanced group, inspect sparsity before eliminating variables and base mesh refinement on a continuous residual.

9.15.1 Fundamental

1. Derive the single-shooting residual for the minimum-energy transfer Equation 9.13 when the initial costates are unknown.
2. Show that the Jacobian of the multiple-shooting matching conditions Equation 9.5 is block banded.
3. Derive Equation 9.10 from the integral form of the state equation.

9.15.2 Computational

4. Transcribe Equation 9.13 using forward Euler and determine the order of the terminal-state error.
5. Add the bound $|u| \leq u_{\max}$ to the double-integrator problem and predict how the optimal control structure changes.
6. Formulate the free-time version of the double-integrator transcription using Equation 9.17.
7. Construct a path constraint that is satisfied at every mesh node but violated between two nodes.
8. Compare a forward-sensitivity gradient with a centered finite-difference directional derivative.

9.15.3 Advanced

9. Explain why eliminating all state variables makes the reduced shooting Hessian dense.
10. Design an h-refinement indicator based on Equation 9.18 and describe when p-refinement would be preferable.

V Python Toolchain

10 Configuring Python with Miniconda 203

10.1 Why use a dedicated environment?
10.2 Install Miniconda
10.3 Create the NOOCDS environment
10.4 Run JupyterLab
10.5 Optional: Visual Studio Code
10.6 Updating or rebuilding the environment
10.7 Common problems

11 Notebook Catalogue and Reproducibility 207

11.1 Download the complete collection
11.2 Learning path
11.3 Individual notebooks
11.4 Running a notebook
11.5 A disciplined experiment

This part collects the reproducible computational material that accompanies the methods developed in the preceding chapters. It first describes the Python environment and the reproducibility workflow, then presents the notebooks in the same conceptual order as the book: optimization, ODEs, linear and nonlinear DAEs, index reduction, numerical DAE methods, and case studies.

Prerequisites, objectives, and expected outcomes

Prerequisites: basic Python syntax and the mathematical topic associated with each laboratory.
Objectives: reproduce the book's numerical experiments, vary parameters systematically, and interpret solver diagnostics rather than relying only on success flags.
Expected outcomes: the reader can create the environment, run and modify a notebook, preserve reproducibility, and report convergence, residual, and stability evidence.

Each notebook is self-contained: it can be read as a worked example, modified interactively, and rerun without hidden dependencies. The notebooks are the place where methods developed throughout the theoretical chapters are turned into working code and reproducible numerical experiments.

Contents

- Configuring Python with Miniconda
- Notebook Catalogue and Reproducibility
- 01 — Unconstrained Minimization
- 02 — Constrained Minimization
- 03 — Numerical ODE Solvers
- 04 — Linear DAEs
- 05 — Nonlinear DAEs
- 06 — DAE Index Analysis and Reduction
- 07 — Nine DAE Case Studies
- 08 — Numerical Methods for DAEs

10. Configuring Python with Miniconda

10.1 Why use a dedicated environment?

The examples in this book use a small, fixed collection of scientific Python packages. Installing them in a dedicated Conda environment keeps the course software isolated from the system Python and makes the same notebooks reproducible on Windows, macOS, and Linux.

The recommended setup consists of:

- **Miniconda**, which supplies Python and the `conda` environment manager;
- the provided `environment.yml`, which records the packages and compatible version ranges;
- **JupyterLab** or **Visual Studio Code** with the official Python and Jupyter extensions.

10.2 Install Miniconda

Download Miniconda from the official installer page and select the installer for your operating system and processor.

10.2.1 Windows

Run the graphical installer. The default per-user installation is adequate. After installation, open **Miniconda Prompt** from the Start menu and use it for the commands in this chapter.

10.2.2 macOS

Choose the installer matching the processor reported by **About This Mac**: Apple silicon (`arm64`) or Intel (`x86_64`). After installation, close and reopen Terminal.

10.2.3 Linux

Download the shell installer, run it from a terminal, and accept Conda shell initialization when prompted. Close and reopen the terminal afterwards.

Verify that Conda is available:

```
conda --version
```

If the command is not found, open a new terminal. On Windows, use Miniconda Prompt; on macOS or Linux, initialize the current shell once with `conda init`, then restart it.

10.3 Create the NOOCDS environment

Download the environment description:

Download `environment.yml`

Open a terminal in the directory containing the downloaded file and run:

```
conda env create -f environment.yml
conda activate NOOCDS
```

The first command creates the environment; the second makes its Python interpreter active in the current terminal. The prompt should now begin with `(NOOCDS)`.

Check the installation:

```
python --version
python -c "import numpy, scipy, matplotlib, sympy; print('environment ready')"
jupyter --version
```

The environment uses the `conda-forge` channel only. This avoids silently mixing binary packages from incompatible channels.

10.4 Run JupyterLab

Activate the environment and start JupyterLab from the directory in which the notebooks were downloaded:

```
conda activate NOOCDS
jupyter lab
```

Jupyter opens in the default web browser. Open a notebook and verify that the kernel displayed in the upper-right corner is **Python 3 (ipykernel)** from the `NOOCDS` environment. Use **Kernel > Restart Kernel and Run All Cells** to reproduce the complete computation from a clean state.

10.5 Optional: Visual Studio Code

Install Visual Studio Code and the official Python and Jupyter extensions. Open the folder containing the examples, then:

1. open the Command Palette;
2. choose **Python: Select Interpreter**;
3. select the interpreter named `NOOCDS`;
4. open the notebook and select the same environment as its kernel.

The interpreter selected by the editor and the kernel selected by the notebook should refer to the same environment.

10.6 Updating or rebuilding the environment

When `environment.yml` changes, update an existing installation with

```
conda activate NOOCDS
conda env update -f environment.yml --prune
```

The `--prune` option removes packages that are no longer part of the declared environment. For a completely clean rebuild:

```
conda deactivate
conda env remove -n NOOCDS
conda env create -f environment.yml
```

10.7 Common problems

The notebook imports packages from the wrong location. Display the active interpreter from a notebook cell:

```
import sys
print(sys.executable)
```

Its path must contain the `NOOCDS` environment. If it does not, change the notebook kernel rather than installing the package again.

The environment name already exists. Use the update command above, or remove the old environment before recreating it.

Plots do not appear. Restart the kernel and run all cells in order. The provided notebooks configure Matplotlib explicitly and do not require a separate graphical backend.

11. Notebook Catalogue and Reproducibility

11.1 Download the complete collection

The archive contains the eight course notebooks listed below and the Conda environment specification needed to run them.

Download: all Python laboratories and the environment file

11.2 Learning path

The numerical prefix records the recommended study order. Each notebook is a self-contained computational lecture, but its *Theory* link identifies the part of the book that should normally be read first.

Notebook	Main learning objective	Theory
01 — Unconstrained minimization	Compare gradient, Newton, quasi-Newton, and SciPy solvers; interpret diagnostics and failure modes	Unconstrained minimization
02 — Constrained minimization	Work through equality- and inequality-constrained problems, KKT candidates, and numerical feasibility checks	Lagrange multipliers and KKT conditions
03 — Numerical ODE solvers	Implement one-step methods; measure order, stability, stiffness, and invariant drift	ODE numerical methods

Notebook	Main learning objective	Theory
04 — Linear DAEs	Study regular pencils, consistent data, index, and implicit integration	Linear DAEs
05 — Nonlinear DAEs	Solve nonlinear constraints and examine drift, stabilization, and projection	DAEs from Lagrangian mechanics
06 — DAE index analysis and reduction	Detect hidden constraints and perform linear and nonlinear index reduction	Index reduction
07 — Nine DAE case studies	Connect the theory to constrained mechanical and high-index models	DAE examples
08 — Numerical methods for DAEs	Compare implicit schemes, Baumgarte stabilization, projection, and coordinate partitioning	DAE numerical methods

11.3 Individual notebooks

11.3.1 01 — Unconstrained minimization

Algorithms and diagnostics on Rosenbrock, Himmelblau, Beale, Powell singular, Brown badly scaled, and Freudenstein–Roth test problems. The examples show why solver agreement, a small gradient, or a success flag are not sufficient on their own.

Download: unconstrained-minimization algorithms and test problems

11.3.2 02 — Constrained minimization

Worked equality- and inequality-constrained problems illustrate Lagrange multipliers, KKT conditions, candidate filtering, and the numerical checks needed to distinguish stationarity from feasibility and local optimality.

Download: constrained-minimization examples

11.3.3 03 — Numerical ODE solvers

Explicit and implicit one-step methods, convergence experiments, absolute stability, stiffness, adaptive integration, and applications including SIR, Robertson kinetics, and gravitational dynamics.

Download: ODE algorithms, stability experiments, and dynamical-system examples

11.3.4 04 — Linear DAEs

Matrix pencils, regularity, consistent initial conditions, index-1 through index-3 examples, Weierstrass form, backward Euler, and implicit midpoint.

Download: linear-DAE theory and numerical experiments

11.3.5 05 — Nonlinear DAEs

Nonlinear algebraic constraints, fully implicit backward Euler, consistent initialization, the constrained pendulum, drift, and post-step projection.

Download: nonlinear-DAE solvers, constraints, and projection

11.3.6 06 — DAE index analysis and reduction

Differential index, regular pencils, nilpotent chains, Wong sequences, hidden constraints, and reduction of semi-explicit nonlinear systems.

Download: DAE index-analysis and reduction experiments

11.3.7 07 — Nine DAE case studies

Nine models drawn from the DAE examples in the book. The notebook clearly distinguishes a direct DAE computation from an index-reduced ODE or a constraint-projection experiment.

Download: nine DAE models with complete derivations and numerical experiments

11.3.8 08 — Numerical methods for DAEs

Implicit time stepping, experimental convergence, high-index mechanics, constraint drift, Baumgarte stabilization, projection, and coordinate partitioning.

Download: implicit DAE integration and constraint-control methods

11.4 Running a notebook

After configuring the environment described in the previous chapter:

```
conda activate NOOCDS
cd path/to/the/downloaded/examples
jupyter lab
```

Open a notebook and select **Restart Kernel and Run All Cells**. A clean run should finish without manual input. Before changing an algorithm, first check that the reference run reproduces its tables, plots, convergence orders, and constraint residuals.

11.5 A disciplined experiment

Change one item at a time: the initial condition, step size, tolerance, stabilization parameter, or solver. Record both the solution error and the diagnostic that matters for the model (for example a gradient norm, local error estimate, algebraic residual, or invariant drift). This makes the notebooks experiments rather than demonstrations.

References