Matrix Formulation of ODEs and Euler Method in Classical Dynamics

Marco Fronzi

1 General Form of a First-Order System

Suppose you have a system of N first-order ODEs:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$$

Where:

- $\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_N(t) \end{bmatrix}$ is an $N \times 1$ column vector of dependent variables,
- $\mathbf{f}(\mathbf{x}, t) = \begin{bmatrix} f_1(\mathbf{x}, t) \\ f_2(\mathbf{x}, t) \\ \vdots \\ f_N(\mathbf{x}, t) \end{bmatrix}$ is the vector-valued function describing the system.

2 Linear ODE System with Constant Coefficients

If the system is linear, it can be written as:

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x}(t) + \mathbf{b}(t)$$

Where:

- A is an $N \times N$ matrix of constants (or time-dependent coefficients),
- $\mathbf{b}(t)$ is a known $N \times 1$ vector function (forcing term or source),
- $\mathbf{x}(t)$ is the vector of unknowns.

Example

Consider a simple 2D system:

$$\begin{cases} \frac{dx_1}{dt} = -3x_1 + 4x_2 \\ \frac{dx_2}{dt} = -2x_1 + x_2 \end{cases}$$

This can be written in matrix form as:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -3 & 4 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with no forcing term ($\mathbf{b}(t) = \mathbf{0}$).

3 Writing Higher-Order ODEs in General Form

The same approach used for second-order systems can be extended to **any Nth-order ODE**. We transform it into a system of first-order equations by defining new state variables for each derivative.

General Nth-Order Linear ODE

Consider a scalar Nth-order linear ODE:

$$\frac{d^N y_1}{dt^N} + a_1 \frac{d^{N-1} y_1}{dt^{N-1}} + \cdots + a_N y_1 = 0$$

We define new variables for each derivative:

$$\begin{aligned} y_2 &= \frac{dy_1}{dt}, \\ y_3 &= \frac{dy_2}{dt} = \frac{d^2 y_1}{dt^2}, \\ &\vdots \\ y_N &= \frac{d^{N-1} y_1}{dt^{N-1}} \end{aligned}$$

Then the original Nth-order equation becomes a first-order system:

$$\begin{cases} \frac{dy_1}{dt} = y_2 \\ \frac{dy_2}{dt} = y_3 \\ \vdots \\ \frac{dy_{N-1}}{dt} = y_N \\ \frac{dy_N}{dt} = -\frac{1}{a_{N+1}}(a_1 y_N + a_2 y_{N-1} + \cdots + a_N y_1) \end{cases}$$

We can then define:

$$\mathbf{x} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, \quad \mathbf{f} = \frac{d\mathbf{x}}{dt} = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ -\frac{1}{a_{N+1}}(a_1 y_N + a_2 y_{N-1} + \cdots + a_N y_1) \end{bmatrix}$$

So the higher-order ODE becomes:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$$

4 Quick Recap: Euler Method

For a first-order ODE:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$$

The Euler method approximates:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot \mathbf{f}(\mathbf{x}_n, t_n)$$

5 Euler Method for Second-Order ODE in Classical Dynamics

For:

$$M\ddot{\mathbf{r}} + C\dot{\mathbf{r}} + K\mathbf{r} = \mathbf{f}(t)$$

with

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{r}(t) \\ \mathbf{v}(t) \end{bmatrix}, \quad \mathbf{v} = \dot{\mathbf{r}}$$

then:

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} \dot{\mathbf{r}} \\ M^{-1}(\mathbf{f}(t) - C\dot{\mathbf{r}} - K\mathbf{r}) \end{bmatrix}$$

6 Second-Order Systems

For second-order ODEs (like Newton's laws), we convert them to a first-order system. Example:

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{a}(\mathbf{r}, \mathbf{v}, t) \quad \Rightarrow \quad \begin{cases} \frac{d\mathbf{r}}{dt} = \mathbf{v} \\ \frac{d\mathbf{v}}{dt} = \mathbf{a} \end{cases}$$

Define:

$$\mathbf{x} = \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}, \quad \frac{d\mathbf{x}}{dt} = \begin{bmatrix} \mathbf{v} \\ \mathbf{a} \end{bmatrix}$$

Thus, we have a first-order system.

7 General Second-Order System

Suppose we have:

$$M\ddot{\mathbf{r}}(t) + C\dot{\mathbf{r}}(t) + K\mathbf{r}(t) = \mathbf{f}(t)$$

Where:

- $\mathbf{r}(t) \in \mathbb{R}^N$ is the vector of positions (generalised coordinates),
- M is the mass matrix, C is the damping matrix, and K is the stiffness matrix,
- $\mathbf{f}(t)$ is the external force vector.

Convert to First-Order System

Define:

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{r}(t) \\ \dot{\mathbf{r}}(t) \end{bmatrix} \in \mathbb{R}^{2N}$$

Then:

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} \dot{\mathbf{r}}(t) \\ M^{-1}\left(\mathbf{f}(t) - C\dot{\mathbf{r}}(t) - K\mathbf{r}(t)\right) \end{bmatrix}$$

Or compactly:

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x}(t) + B\mathbf{f}(t)$$

where

$$A = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ M^{-1} \end{bmatrix}$$

Simple Example: Mass-Spring-Damper

Consider:

$$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = f(t)$$

Define:

$$\mathbf{x}(t) = \begin{bmatrix} r \\ v \end{bmatrix}$$

Then:

$$\frac{d}{dt}\begin{bmatrix} r \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}\begin{bmatrix} r \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} f(t)$$

8 Euler Integration Step Using Matrices

At each time step:

- Compute acceleration:

$$\mathbf{a}_n = M^{-1}(\mathbf{f}_n - C\mathbf{v}_n - K\mathbf{r}_n)$$

- Update:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \Delta t \cdot \mathbf{v}_n$$
$$\mathbf{v}_{n+1} = \mathbf{v}_n + \Delta t \cdot \mathbf{a}_n$$

- Pack:

$$\mathbf{x}_{n+1} = \begin{bmatrix} \mathbf{r}_{n+1} \\ \mathbf{v}_{n+1} \end{bmatrix}$$

9 Example: Matrix Form for Euler Method

Consider:

$$m\ddot{x} + c\dot{x} + kx = f(t)$$

Define:

$$\mathbf{x}_n = \begin{bmatrix} x_n \\ v_n \end{bmatrix}$$

with

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}$$

Then:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \cdot (A\mathbf{x}_n + Bf_n)$$

10 Runge-Kutta Method

Motivation and Derivation

Starting from the Taylor series expansion of $y(t)$ around t_n:

$$y(t_{n+1}) = y(t_n) + \Delta t \frac{dy}{dt}\bigg|_{t_n} + \frac{(\Delta t)^2}{2} \frac{d^2y}{dt^2}\bigg|_{t_n} + \mathcal{O}(\Delta t^3)$$

Since computing higher-order derivatives of $f(y, t)$ directly is complicated, we seek an approximate method that captures higher-order accuracy by sampling f at multiple points.

The Runge-Kutta method achieves this by calculating intermediate slopes and combining them smartly.

Fourth-Order Runge-Kutta (RK4)

The RK4 method is one of the most popular and balances accuracy and computational efficiency. It involves four evaluations of the function $f(y, t)$ per time step:

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f\left(y_n + \frac{\Delta t}{2} k_1, t_n + \frac{\Delta t}{2}\right) \\
k_3 &= f\left(y_n + \frac{\Delta t}{2} k_2, t_n + \frac{\Delta t}{2}\right) \\
k_4 &= f\left(y_n + \Delta t k_3, t_n + \Delta t\right)
\end{aligned}
$$

Then update:

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

Key Idea

The Runge-Kutta method mimics the Taylor expansion up to fourth-order terms without needing explicit derivatives of $f(y, t)$. It blends the information at the beginning, midpoint, and end of the interval to achieve high accuracy.

11 Derivation of the Runge-Kutta Method from Taylor Expansion

Let us start simple.

Suppose we have the ODE:

$$\frac{dy}{dt} = f(y, t) \quad \text{with initial condition} \quad y(t_0) = y_0$$

We want to find $y(t_{n+1})$ from $y(t_n)$.

Step 1: Taylor series expansion

Expand $y(t)$ around t_n:

$$y(t_{n+1}) = y(t_n + \Delta t) = y(t_n) + \Delta t \frac{dy}{dt}\Big|_{t_n} + \frac{(\Delta t)^2}{2} \frac{d^2 y}{dt^2}\Big|_{t_n} + \mathcal{O}(\Delta t^3)$$

or equivalently:

$$y_{n+1} = y_n + \Delta t f(y_n, t_n) + \frac{(\Delta t)^2}{2} \frac{d}{dt} f(y_n, t_n) + \mathcal{O}(\Delta t^3)$$

Notice: we need $\frac{d}{dt} f(y, t)$, but thats not usually easy to compute!

Step 2: Expand $\frac{d}{dt}f(y,t)$

Using the chain rule:

$$\frac{d}{dt}f(y,t) = \frac{\partial f}{\partial y}\frac{dy}{dt} + \frac{\partial f}{\partial t} = \frac{\partial f}{\partial y}f(y,t) + \frac{\partial f}{\partial t}$$

Substituting back:

$$y_{n+1} = y_n + \Delta t f(y_n, t_n) + \frac{(\Delta t)^2}{2}\left(\frac{\partial f}{\partial y}f(y_n, t_n) + \frac{\partial f}{\partial t}\right) + \mathcal{O}(\Delta t^3)$$

However, computing $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial t}$ at every step is often cumbersome.

Idea: Find an approximate method that captures the second-order behaviour without needing explicit derivatives.

Step 3: Invent a smarter method: average slopes

Define two slopes:

- $k_1 = f(y_n, t_n)$ (slope at the beginning),
- $k_2 = f\left(y_n + \frac{\Delta t}{2}k_1, t_n + \frac{\Delta t}{2}\right)$ (predicted slope at the midpoint).

Then use k_2 to update y:

$$y_{n+1} = y_n + \Delta t k_2$$

This is known as the **second-order Runge-Kutta method** (or **midpoint method**).

Why does this work? Because the Taylor series at the midpoint automatically includes higher-order corrections, without explicitly needing $\partial f/\partial y$ or $\partial f/\partial t$.

Step 4: General form of Runge-Kutta methods

The general idea of Runge-Kutta methods is to take a weighted combination of slopes:

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i k_i$$

where each k_i is computed as:

$$k_i = f\left(y_n + \Delta t \sum_{j=1}^{i-1} a_{ij}k_j,\ t_n + c_i \Delta t\right)$$

The coefficients a_{ij}, b_i, c_i are carefully chosen to match the Taylor expansion up to the desired order.

Step 5: Example Runge-Kutta 4 (RK4)

The most famous and widely used case is the fourth-order Runge-Kutta (RK4) method:

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + \frac{\Delta t}{2}k_1, t_n + \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(y_n + \frac{\Delta t}{2}k_2, t_n + \frac{\Delta t}{2}\right)$$

$$k_4 = f\left(y_n + \Delta t k_3, t_n + \Delta t\right)$$

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

This method captures up to the fourth-order terms of the Taylor series, balancing accuracy and computational cost very efficiently.

Quick Summary

- Euler method uses only k_1 (first-order accurate),
- Midpoint method (RK2) uses k_1 and k_2 (second-order),
- RK4 uses k_1, k_2, k_3, k_4 (fourth-order).

Each method is derived by mimicking the Taylor expansion without explicitly computing derivatives of f.

12 Local and Global Errors of the Runge-Kutta 4 Method

Local Truncation Error (LTE)

The local truncation error is the error made in a **single step** of the method, assuming perfect knowledge of the solution at the previous time step.

Starting from the Taylor expansion of the exact solution around t_n:

$$y(t_n + \Delta t) = y(t_n) + \Delta t y_n' + \frac{(\Delta t)^2}{2}y_n'' + \frac{(\Delta t)^3}{6}y_n''' + \frac{(\Delta t)^4}{24}y_n^{(4)} + \mathcal{O}(\Delta t^5)$$

where derivatives are evaluated at t_n.

Recall that the RK4 update rule is:

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

with:

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + \frac{\Delta t}{2}k_1, t_n + \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(y_n + \frac{\Delta t}{2}k_2, t_n + \frac{\Delta t}{2}\right)$$

$$k_4 = f\left(y_n + \Delta t k_3, t_n + \Delta t\right)$$

Expanding each k_i around (y_n, t_n) using Taylor series and substituting into the RK4 formula shows that:

- The terms up to Δt^4 match exactly the Taylor expansion of the true solution. - The first unmatched term appears at order $\mathcal{O}(\Delta t^5)$.

Therefore, the **local truncation error per step** for RK4 satisfies:

$$\text{LTE} = \mathcal{O}(\Delta t^5)$$

Global Truncation Error (GTE)

The global truncation error is the error accumulated over the entire integration interval $[t_0, T]$.

Suppose we perform N steps, where:

$$N = \frac{T - t_0}{\Delta t}$$

Each step introduces an error of order Δt^5. The cumulative global error after N steps is approximately:

$$\text{GTE} = N \times \text{LTE} \sim \frac{1}{\Delta t} \times \mathcal{O}(\Delta t^5) = \mathcal{O}(\Delta t^4)$$

Thus, the **global truncation error** of RK4 satisfies:

$$\text{GTE} = \mathcal{O}(\Delta t^4)$$

Summary

- **Local truncation error:** $\mathcal{O}(\Delta t^5)$
- **Global truncation error:** $\mathcal{O}(\Delta t^4)$

This confirms that the Runge-Kutta 4 method is a **fourth-order accurate** numerical integration scheme.

13 Solving the Pendulum Problem Using RK4

1. Governing Equation of the Pendulum

The equation for a simple pendulum of length L under gravity g is:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0$$

where $\theta(t)$ is the angular displacement.

2. Non-Dimensionalisation

Define a characteristic time scale:

$$\omega_0 = \sqrt{\frac{g}{L}}$$

Introduce a non-dimensional time:

$$\tau = \omega_0 t$$

Then:

$$\frac{d}{dt} = \omega_0 \frac{d}{d\tau}, \quad \frac{d^2}{dt^2} = \omega_0^2 \frac{d^2}{d\tau^2}$$

Substituting into the original equation:

$$\omega_0^2 \frac{d^2\theta}{d\tau^2} + \omega_0^2 \sin(\theta) = 0$$

Simplifying:

$$\frac{d^2\theta}{d\tau^2} + \sin(\theta) = 0$$

Thus, the non-dimensionalised equation becomes:

$$\frac{d^2\theta}{d\tau^2} + \sin(\theta) = 0$$

3. Writing as a System of First-Order ODEs

Introduce:

$$\theta_1 = \theta, \quad \theta_2 = \frac{d\theta}{d\tau}$$

Then:

$$\begin{cases} \frac{d\theta_1}{d\tau} = \theta_2 \\ \\ \frac{d\theta_2}{d\tau} = -\sin(\theta_1) \end{cases}$$

We can write this compactly as:

$$\frac{d\mathbf{x}}{d\tau} = \mathbf{f}(\mathbf{x}, \tau) \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} \theta_1 \\ \\ \theta_2 \end{bmatrix} \quad \text{and} \quad \mathbf{f}(\mathbf{x}, \tau) = \begin{bmatrix} \theta_2 \\ \\ -\sin(\theta_1) \end{bmatrix}$$

4. Applying the RK4 Method

We apply the fourth-order Runge-Kutta method to the system:

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{x}_n, \tau_n)$$

$$\mathbf{k}_2 = \mathbf{f}\left(\mathbf{x}_n + \frac{\Delta\tau}{2}\mathbf{k}_1, \tau_n + \frac{\Delta\tau}{2}\right)$$

$$\mathbf{k}_3 = \mathbf{f}\left(\mathbf{x}_n + \frac{\Delta\tau}{2}\mathbf{k}_2, \tau_n + \frac{\Delta\tau}{2}\right)$$

$$\mathbf{k}_4 = \mathbf{f}\left(\mathbf{x}_n + \Delta\tau\mathbf{k}_3, \tau_n + \Delta\tau\right)$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{\Delta\tau}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$$

Each $\mathbf{k}_i$ is a vector corresponding to (θ_1, θ_2).

5. Python Code for RK4 Pendulum Solver

```
# Define the right-hand side
def pendulum_rhs(x, tau):
    theta1, theta2 = x
    dtheta1_dtau = theta2
    dtheta2_dtau = -np.sin(theta1)
    return np.array([dtheta1_dtau, dtheta2_dtau])
```

```
# Runge-Kutta 4 solver
def rk4_step(f, x, tau, dtau):
    k1 = f(x, tau)
    k2 = f(x + 0.5 * dtau * k1, tau + 0.5 * dtau)
    k3 = f(x + 0.5 * dtau * k2, tau + 0.5 * dtau)
    k4 = f(x + dtau * k3, tau + dtau)
    return x + (dtau / 6.0) * (k1 + 2*k2 + 2*k3 + k4)

# Initial conditions
theta0 = np.pi / 4  # 45 degrees
omega0 = 0.0        # initial angular velocity
x0 = np.array([theta0, omega0])

# Time parameters
tau0 = 0.0
tau_max = 20.0
dtau = 0.01
N_steps = int((tau_max - tau0) / dtau)

# Storage
tau_values = np.zeros(N_steps + 1)
x_values = np.zeros((N_steps + 1, 2))

# Initial assignment
tau_values[0] = tau0
x_values[0, :] = x0

# Time integration
for n in range(N_steps):
    x_values[n+1, :] = rk4_step(pendulum_rhs, x_values[n, :], tau_values[n], dtau)
    tau_values[n+1] = tau_values[n] + dtau
```

6. Notes

- This solver assumes no damping and small numerical time steps.
- For small angles θ, the system approximates a simple harmonic oscillator.
- Non-dimensionalisation helps reduce dependence on physical parameters, improving generality and scaling.

14 Example: Matrix Form for RK4 Method

Consider:

$$m\ddot{x} + c\dot{x} + kx = f(t)$$

Define:

$$\mathbf{x}_n = \begin{bmatrix} x_n \\ v_n \end{bmatrix}$$

with:

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}$$

The system can be written as:

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x}(t) + Bf(t)$$

The RK4 update rule becomes:

$$\mathbf{k}_1 = A\mathbf{x}_n + Bf_n$$

$$\mathbf{k}_2 = A\left(\mathbf{x}_n + \frac{\Delta t}{2}\mathbf{k}_1\right) + Bf\left(t_n + \frac{\Delta t}{2}\right)$$

$$\mathbf{k}_3 = A\left(\mathbf{x}_n + \frac{\Delta t}{2}\mathbf{k}_2\right) + Bf\left(t_n + \frac{\Delta t}{2}\right)$$

$$\mathbf{k}_4 = A\left(\mathbf{x}_n + \Delta t\mathbf{k}_3\right) + Bf\left(t_n + \Delta t\right)$$

Then the update for the trajectory is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{\Delta t}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)$$

15 Explicit and Implicit Euler Methods

1. Explicit Euler Method

The **Explicit Euler** method is the simplest numerical integrator for ordinary differential equations (ODEs).

Given an ODE:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$$

the Explicit Euler update rule is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t\, \mathbf{f}(\mathbf{x}_n, t_n)$$

Key Characteristics:

- Easy to implement: only requires evaluating $\mathbf{f}$ at the current state.
- **First-order accurate**: local truncation error $\mathcal{O}(\Delta t^2)$, global error $\mathcal{O}(\Delta t)$.

- **Conditionally stable**: requires small Δt for stiff problems or oscillatory systems.

2. Implicit Euler Method

The **Implicit Euler** method (also called the backward Euler method) improves stability, especially for stiff problems, by using the function evaluated at the *future* state.

The update rule is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t\, \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1})$$

Notice that $\mathbf{x}_{n+1}$ appears on both sides. Thus, to advance one step, we must **solve a system of equations** for $\mathbf{x}_{n+1}$.

Key Characteristics:

- More complex to implement: may require iterative solvers (e.g., Newton-Raphson method).
- **First-order accurate**: same order of accuracy as Explicit Euler.
- **Unconditionally stable**: suitable for stiff problems.

3. Comparison Between Explicit and Implicit Euler

- **Explicit Euler** is simple but unstable for large time steps.
- **Implicit Euler** is stable for any time step but requires solving equations at every step.
- Both methods are only **first-order accurate**.
- Neither is symplectic for conservative Hamiltonian systems.

4. Example: Matrix Form for Explicit and Implicit Euler

Consider the linear system:

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x}(t) + Bf(t)$$

where A is a constant matrix and B a constant vector.

Explicit Euler

The update becomes:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t\,(A\mathbf{x}_n + Bf_n)$$

Implicit Euler

The update requires solving:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \left(A\mathbf{x}_{n+1} + Bf_{n+1} \right)$$

which can be rearranged as:

$$\left(I - \Delta t A \right) \mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t B f_{n+1}$$

where I is the identity matrix.
Thus, at each step, we must solve a linear system for $\mathbf{x}_{n+1}$.

5. Summary

- **Explicit Euler:**

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \, \mathbf{f}(\mathbf{x}_n, t_n)$$

- **Implicit Euler:**

$$\left(I - \Delta t A \right) \mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t B f_{n+1}$$

  or in nonlinear cases:

$$\mathbf{x}_{n+1} \quad \text{solves} \quad \mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \, \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1})$$

16 Symplectic Euler as a Combination of Explicit and Implicit Euler Methods

1. Motivation

The Symplectic Euler method can be interpreted as a clever combination of the **Explicit Euler** and **Implicit Euler** methods:

- One variable (typically momentum or velocity) is updated **explicitly**.
- The other variable (typically position) is updated **implicitly**.

This asymmetric treatment is the key to preserving the symplectic structure.

2. Symplectic Euler Scheme

Consider the simple mechanical system:

$$\begin{cases} \frac{dx}{dt} = v \\ \\ \frac{dv}{dt} = a(x, v, t) \end{cases}$$

The Symplectic Euler updates:

$$v_{n+1} = v_n + \Delta t\, a(x_n, v_n, t_n) \quad \text{(Explicit update)}$$

$$x_{n+1} = x_n + \Delta t\, v_{n+1} \quad \text{(Implicit update)}$$

Notice that v_{n+1} is used immediately to update x_{n+1}.

3. Explicit and Implicit Euler Interpretation

We can think of Symplectic Euler as:

- **Explicit Euler on** v:

$$v_{n+1} = v_n + \Delta t\, a(x_n, v_n, t_n)$$

- **Implicit Euler on** x, but using v_{n+1}:

$$x_{n+1} = x_n + \Delta t\, v_{n+1}$$

Thus, even though we do not solve a nonlinear system, the position update uses the future value of velocity introducing an *implicit character* in the scheme.

4. Matrix Formulation

Define:

$$\mathbf{x}_n = \begin{bmatrix} x_n \\ \\ v_n \end{bmatrix}$$

The updates can be written as:

$$v_{n+1} = v_n + \Delta t \left(-\frac{k}{m} x_n - \frac{c}{m} v_n + \frac{1}{m} f(t_n) \right)$$

$$x_{n+1} = x_n + \Delta t v_{n+1}$$

For a linear system of the form:

$$\frac{d}{dt} \begin{bmatrix} x \\ \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix} \begin{bmatrix} x \\ \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \\ \frac{1}{m} \end{bmatrix} f(t)$$

the Symplectic Euler method updates sequentially, first velocity, then position.

5. Error Analysis

- **Local truncation error:** $\mathcal{O}(\Delta t^2)$
- **Global truncation error:** $\mathcal{O}(\Delta t)$

Thus, Symplectic Euler is a **first-order accurate** method.

6. Why Symplectic?

Despite being only first-order accurate, Symplectic Euler exactly preserves the symplectic two-form:

$$J^\top \Omega J = \Omega$$

where Ω is the canonical symplectic matrix:

$$\Omega = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

This preservation implies:

- Conservation of phase space volume (Liouville's theorem),
- No artificial secular growth or decay of energy over long times,
- Accurate qualitative behaviour of Hamiltonian systems even over very long integrations.

7. Summary

- Symplectic Euler combines an explicit step for momentum/velocity with an implicit step for position.
- It is symplectic, thus ideal for conservative systems.
- It is only first-order accurate in Δt, but exhibits excellent long-term behaviour.