Numerical Methods in Python for N-Body Dynamics

1 Motivation and Scope

Many gravitational dynamics problems, such as orbits near Lagrange points or general N-body motion, are governed by ordinary differential equations (ODEs) that rarely have closed-form solutions. Instead, these systems are integrated numerically by stepping the solution forward in time. This handout focuses on the numerical tools required for:

- Simulating orbits in gravitational fields (e.g. near Lagrange points).
- Building an N-body simulator for point masses interacting via Newtonian gravity.

The emphasis is on initial value problems for ODEs, explicit time-stepping schemes (Euler and classical Runge–Kutta 4), and practical Python implementations using NumPy, SciPy, Matplotlib, and Astropy for units and physical constants.

2 Initial Value Problems for ODEs

A large class of dynamical systems can be written as an initial value problem (IVP)

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{1}$$

where:

- $\mathbf{y}(t)$ is the *state vector* (positions, velocities, etc.).
- $\mathbf{f}(t, \mathbf{y})$ gives the time derivative of the state.

For example, for a single body moving in a plane under a central gravitational field with parameter μ, one convenient state choice is

$$\mathbf{y} = (x,\, y,\, v_x,\, v_y),$$

and the acceleration is

$$\mathbf{a} = -\mu \frac{\mathbf{r}}{\|\mathbf{r}\|^3}, \qquad \mathbf{r} = (x, y).$$

Then the ODE system becomes

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ a_x \\ a_y \end{pmatrix}.$$

3 Time-Stepping Methods

3.1 Forward Euler Method

The simplest explicit method for numerically solving an IVP is the forward Euler method. Given a time step h and a current state $\mathbf{y}_n \approx \mathbf{y}(t_n)$, the update is

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\,\mathbf{f}(t_n, \mathbf{y}_n). \tag{2}$$

Intuitively:

- Compute the current slope $\mathbf{f}(t_n, \mathbf{y}_n)$.
- Pretend this slope stays constant for a short time h.
- Move along this straight line for time h to get $\mathbf{y}_{n+1}$.

This method is easy to code but only first-order accurate and often unstable for orbital problems unless h is very small.

3.2 Classical Runge–Kutta 4 (RK4)

A more accurate explicit scheme is the classical fourth-order Runge–Kutta method (RK4). For a step from t_n to $t_{n+1} = t_n + h$, the method computes

$$k_1 = \mathbf{f}(t_n, \mathbf{y}_n), \tag{3}$$

$$k_2 = \mathbf{f}\left(t_n + \frac{h}{2},\, \mathbf{y}_n + \frac{h}{2}k_1\right), \tag{4}$$

$$k_3 = \mathbf{f}\left(t_n + \frac{h}{2},\, \mathbf{y}_n + \frac{h}{2}k_2\right), \tag{5}$$

$$k_4 = \mathbf{f}(t_n + h,\, \mathbf{y}_n + hk_3), \tag{6}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{7}$$

Here each k_i is a slope (a derivative of $\mathbf{y}$) evaluated at a different point inside the time step:

- k_1 uses the slope at the beginning of the interval.
- k_2 and k_3 use slopes at the midpoint, using improved guesses for $\mathbf{y}$.
- k_4 uses the slope at the end of the interval, based on the best guess so far.

The final update is a weighted average of these slopes, with more weight on the midpoints (which are usually more representative of the whole step than just the endpoints).

3.3 Why RK4 Gives Better Approximations

The exact solution $\mathbf{y}(t)$ can be expanded in a Taylor series around t_n:

$$\mathbf{y}(t_n + h) = \mathbf{y}(t_n) + h\mathbf{y}'(t_n) + \frac{h^2}{2}\mathbf{y}''(t_n) + \frac{h^3}{6}\mathbf{y}'''(t_n) + \frac{h^4}{24}\mathbf{y}^{(4)}(t_n) + \mathcal{O}(h^5).$$

Euler's method only matches the first term (the slope), so its error per step is of order h^2. RK4 cleverly combines the four slopes $k_1, \ldots, k_4$ so that this average matches the Taylor series up to the h^4 term. As a result:

- The error made in a single RK4 step is of order h^5.

- The accumulated error over many steps is of order h^4.

This means that if the time step h is made 10 times smaller, the global error scales roughly like $10^4 = 10{,}000$ times smaller (as long as h is already reasonably small). In practice, this allows much larger h than Euler for the same accuracy, which is crucial for orbital and Lagrange-point simulations.

4 Python Tools for Numerical Integration

For the project, the main Python ecosystem pieces are:

- **NumPy** for arrays and vectorized operations on state vectors.
- **Matplotlib** for plotting trajectories, phase-space plots, and potential contours.
- **SciPy** (`scipy.integrate`) for robust ODE solvers like `solve_ivp` that implement adaptive Runge–Kutta methods.
- **Astropy** (`astropy.units`, `astropy.constants`) for handling physical units and gravitational parameters safely.

In the first example below, the RK4 algorithm is coded "by hand" using NumPy. In the second example, `solve_ivp` from SciPy is used as a higher-level integrator, and Astropy is used briefly to get realistic gravitational parameters.

5 Example 1: 1D Harmonic Oscillator with RK4 (From Scratch)

Mathematical Setup

The equation of motion for a harmonic oscillator with angular frequency ω is

$$\frac{d^2x}{dt^2} = -\omega^2 x. \tag{8}$$

Introduce the velocity $v = \frac{dx}{dt}$ and define the state $\mathbf{y} = (x, v)$. Then

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -\omega^2 x \end{pmatrix}. \tag{9}$$

This has exactly the $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ form needed for RK4.

Code and Explanation

Listing 1: RK4 integrator for a 1D harmonic oscillator

```
import numpy as np
import matplotlib.pyplot as plt

def f_osc(t, y, omega):
    """
    Right-hand side for 1D harmonic oscillator.
    Input:
        t : current time (float)
        y : current state [x, v] (NumPy array of length 2)
        omega : angular frequency (float)
    Output:
```

```
        dy_dt : derivative [dx/dt, dv/dt]
    """
    x, v = y # unpack the state vector
    dxdt = v # definition of velocity
    dvdt = -omega**2 * x # from the equation of motion
    return np.array([dxdt, dvdt])

def rk4_step(f, t, y, h, *args):
    """
    One step of classical RK4 for y' = f(t, y).
    Inputs:
        f : function f(t, y, *args) that returns dy/dt
        t : current time
        y : current state (NumPy array)
        h : time step
        args : extra parameters passed to f (here omega)
    Output:
        y_next : approximate state at time t + h
    """
    k1 = f(t, y, *args)
    k2 = f(t + 0.5*h, y + 0.5*h*k1, *args)
    k3 = f(t + 0.5*h, y + 0.5*h*k2, *args)
    k4 = f(t + h, y + h*k3, *args)

    # Weighted average of slopes
    return y + (h/6.0)*(k1 + 2*k2 + 2*k3 + k4)

# ----- Set up the problem -----
omega = 1.0 # angular frequency
t0, t_end = 0.0, 20.0 # start and end times
h = 0.01 # time step size

# Number of steps (integer division)
n_steps = int((t_end - t0) / h)

# Arrays to store the solution
t_vals = np.zeros(n_steps + 1) # times
y_vals = np.zeros((n_steps + 1, 2)) # [x, v] at each time

# Initial condition: x(0) = 1, v(0) = 0
t_vals[0] = t0
y_vals[0] = np.array([1.0, 0.0])

# ----- Time integration loop -----
for n in range(n_steps):
    t = t_vals[n] # current time
    y = y_vals[n] # current state [x, v]
    y_next = rk4_step(f_osc, t, y, h, omega)

    t_vals[n+1] = t + h # store next time
    y_vals[n+1] = y_next # store next state

# ----- Plot displacement over time -----
plt.figure()
plt.plot(t_vals, y_vals[:, 0]) # y_vals[:, 0] is x
plt.xlabel("t")
plt.ylabel("x(t)")
plt.title("1D Harmonic Oscillator (RK4)")
```

```
plt.grid(True)
plt.show()
```

Key points:

- `y` is always a NumPy array containing the whole state; here it has two entries.
- `rk4_step` is written to be reusable: it does not know any physics, it only calls `f`.
- The integration loop is always: take current time and state, do one RK4 step, store the result, repeat.

The same pattern will be used for higher-dimensional systems like orbits and N-body motion; only the definition of `f` changes.

6 Example 2: Simple Planar Two-Body Orbit with `solve_ivp`

The second example is closer to the project. It:

- Integrates the motion of a test particle around a central mass (e.g. a satellite around Earth) in a plane.
- Uses SciPy's `solve_ivp`, which internally uses an adaptive Runge–Kutta method similar in spirit to RK4.
- Demonstrates how Astropy can provide physical constants in consistent units (but the core idea works even if the Astropy lines are temporarily ignored).

Mathematical Setup

We again use the state

$$\mathbf{y} = (x, y, v_x, v_y),$$

with (x, y) in meters and (v_x, v_y) in meters per second. The gravitational parameter is

$$\mu = GM,$$

where G is the gravitational constant and M is the mass of the central body. The acceleration is

$$\mathbf{a} = -\mu \frac{\mathbf{r}}{\|\mathbf{r}\|^3}, \qquad \mathbf{r} = (x, y).$$

The right-hand side function then returns (v_x, v_y, a_x, a_y).

Code and Explanation

Listing 2: Planar two-body orbit using solve$_i$*vpandastropy*

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import solve_ivp
from astropy import units as u
from astropy import constants as const

def two_body_rhs(t, y, mu):
    """
    Right-hand side for planar two-body problem.
```

```
    Input:
        t : time (float)
        y : state [x, y, vx, vy] in SI units (m, m/s)
        mu : gravitational parameter G*M (float, in m^3/s^2)
    Output:
        derivative [dx/dt, dy/dt, dvx/dt, dvy/dt]
    """
    x, y_pos, vx, vy = y # unpack the state
    r_vec = np.array([x, y_pos]) # position vector
    r = np.linalg.norm(r_vec) # distance from origin

    # Gravitational acceleration vector
    ax, ay = -mu * r_vec / r**3

    return [vx, vy, ax, ay]

# ----- Define central body (e.g. Earth) -----
# mu_earth has units of m^3 / s^2
mu_earth = (const.G * const.M_earth).to(u.m**3 / u.s**2).value

# ----- Choose initial position and velocity -----
# Start at 7000 km from Earth's center on the x-axis
r0 = 7000.0 * 1e3 * u.m # 7000 km in meters

# Circular-orbit speed at radius r0 (approximately)
v_circ = np.sqrt(mu_earth / r0.to_value(u.m)) * u.m / u.s

# State vector y0 = [x0, y0, vx0, vy0]
y0 = np.array([
    r0.to_value(u.m), 0.0, # initial position (x0, y0)
    0.0, v_circ.to_value(u.m/u.s) # initial velocity (vx0, vy0)
])

# Integrate for a certain amount of time (in seconds)
t_span = (0.0, 10_000.0)
t_eval = np.linspace(t_span[0], t_span[1], 2000)

# ----- Call solve_ivp -----
sol = solve_ivp(
    fun=lambda t, y: two_body_rhs(t, y, mu_earth),
    t_span=t_span,
    y0=y0,
    t_eval=t_eval,
    rtol=1e-9, # relative error tolerance
    atol=1e-9 # absolute error tolerance
)

# Extract solution arrays
x = sol.y[0] # x(t)
y_pos = sol.y[1] # y(t)

# ----- Plot the orbit in the xy-plane -----
plt.figure()
plt.plot(x / 1e3, y_pos / 1e3) # convert to km for plotting
plt.gca().set_aspect("equal", "box")
plt.xlabel("x [km]")
plt.ylabel("y [km]")
plt.title("Planar Two-Body Orbit (solve_ivp)")
```

```
plt.grid(True)
plt.show()
```

The main ideas are:

- `two_body_rhs` is the only function that knows about gravity; `solve_ivp` only knows that it must integrate $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$.
- `solve_ivp` returns an object `sol` where `sol.t` is the time grid and `sol.y` is a 2D array containing all components of the state over time.
- The orbital and N-body problems in the project will reuse exactly this structure, with more components in `y` and more complicated accelerations.

7 Connection to Lagrange-Point Orbits and N-Body Simulation

The patterns in these two examples are the building blocks for the project deliverables:

- **Orbits around Lagrange points:** Start from the CR3BP equations in a rotating frame, write them as $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, and then integrate using either a custom RK4 (like in Example 1) or `solve_ivp` (like in Example 2).
- **General N-body simulator:** Let $\mathbf{y}$ contain positions and velocities of all bodies, and let $\mathbf{f}$ compute the sum of all pairwise gravitational accelerations using NumPy arrays. The integration loop (or `solve_ivp` call) then follows the same pattern as above.

By keeping the numerical integration part (RK4 or `solve_ivp`) separate from the physics part (definition of $\mathbf{f}$), the code stays modular and easier to debug, extend, and reuse for different gravitational setups.