Simulation Assignment Report

Bruno Páscoa, 107418
David Mendonça, 107360

May 13, 2025

Abstract

Report for the assignment for the Optimization section of the Optimization and Simulation course.

1 Introduction

2 Bus queue

2.1 Simulation Metrics

For this problem, we were tasked with simulating a bus maintenance station with 1 inspection queue and 2 maintenance queues, and were asked to simulate for 160 hours and then compute:

- average delay in each queue
- average length for each queue
- the utilization of the inspection station
- the utilization of each repair station

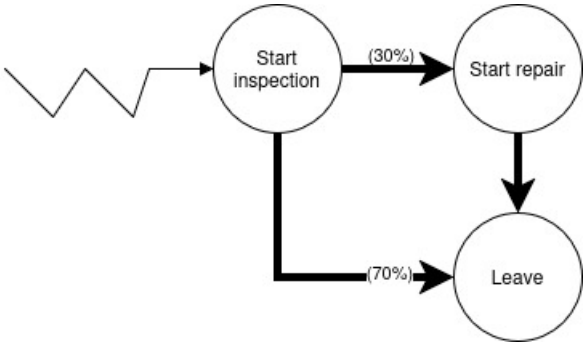

Figure 1: Event diagram

2.1.1 Strategy used

To solve this problem, I created a simpy program that can be split into 5 parts across 2 files:

- bus_sim.py:
  - A main function for gathering the input, calling the simulate function, gathering the results, and then plotting them
  - A Simulate() function to set up simpy
  - A bus_generator() function to generate new buses at the defined interarrival time

  - A bus() function to simulate the bus's lifecycle (arrive, go to inspection station, have a 30% chance to go to the inspection station and then leave)

- bus_statistics: a class that holds all the metrics as well as the methods to access and update them

(Additionally, an extra file (utils.py) was added containing functions to verify input and to help with plotting)

The final part was deciding when to record the queue size and utilization. For this, I decided to record the value before the change and the time of change (which, along with the time of the last change, gave us the time at that value); as such, I divided the lifecycle into 3 parts:

- Before the resource.request(), when the queue size increases
- At the yield, when the bus leaves a queue and enters the station (as the actual change happens "inside" the yield, I had to record the value pre-yield and the time after the yield itself)
- At the end of the "request context" (last line of the "with"), when the bus leaves the station

(This process was done separately for inspection and repair.)

2.1.2 Results

After preparing the simulation and running it with the provided (default) values for 1000 runs, these were the results:

- Average inspection utilization: 0.99
- Average repair queue size: 0.00
- Average repair utilization: 0.57
- Average inspection delay (minutes): 10.47
- Average repair delay (minutes): 6.00
- Average bus duration: 109.84569024003115
- Average bus exit rate: 0.009103679878699231
- Average arrival rate: 0.008333333333333333

We also generated some GIFs for the first 10 runs (as Overleaf does not allow embedding GIFs into PDFs, these can be found in Bus_Queue/results)

2.2 Minimum arrival rate that can still be handled

2.2.1 Problem description

For this one, we are tasked with taking the same system and finding the interarrival rate at which the system "can no longer handle the buses".

2.2.2 Strategy used

Since most of the code was already done, the only question remaining was "what criteria do I use to determine if the system is overloaded or not" and while I initially intended to use graph analysis (check if the delays just keep rising), this always comes with the problem that we cannot guarantee that a system will not become overloaded eventually. As such, I decided to use a much more objective criterion: a system is considered "overloaded" whenever the rate at which buses arrive is greater than the rate at which they exit.

It should be noted, however, that these criteria only guarantee that (unless the variables change) the system will eventually become overloaded (if given an infinite amount of time), not that it may happen within a "reasonable" amount of time (however, since the problem did not specify a time limit, I will assume my criteria is valid)

2.2.3 Results

For this criterion, the minimum interarrival time that the station can still handle is 112 minutes (although with 110 and 111 minutes, the results are close enough that they can vary, even with a substantial number of runs), with an average arrival rate of 0.00893 and an average exit rate of 0.00897.

2.3 Experimentation and Validation

For experimentation, I added ample logging to help with debugging, as well as a "–runs" argument for determining the number of runs (the results displayed will be the average of those runs). I then ran the program with 100 runs (10 runs for plotting, due to the GIF generation not being very efficient), to get the results displayed. For determining the minimum interarrival rate, I first started by decreasing by 10 minutes at a time to find the range, followed by testing each value in said range.

3 Flying Projectile

For this problem, we were tasked with a prediction problem: a flying projectile. This is broken down into:

- $x(t)$: which represents the x position of a falling projectile
- $z(t)$: which represents the z position of a falling projectile
- u: which represents the air resistance

The differential equations that govern this model are:

$$m\frac{d^2x(t)}{dt^2} = -u\left(\frac{dx(t)}{dt}\right)^2 \cdot \text{sign}\left(\frac{dx(t)}{dt}\right)$$

$$m\frac{d^2z(t)}{dt^2} = -m\cdot g - u\left(\frac{dz(t)}{dt}\right)^2 \cdot \text{sign}\left(\frac{dz(t)}{dt}\right)$$

Two method solving solutions are discussed: the Forward Euler (1^{st} order) method; and the Runge Kutta (4^{th} order) method. Due to analytical solving limitations, it was used as the ideal trajectory the functionalities of (from scipy.integrate import solve_ivp) from Python, which should give a reasonable result comparable to the analytical results.

3.1 Simulation Metrics

Table 1: Summary of Simulation Metrics for Euler and RK4 Methods

Metric	**Euler Method**	**Runge-Kutta (RK4)**
Global Error (L2 norm)	$\|\vec{x} - \vec{x}_{\text{ref}}\|_2 + \|\vec{z} - \vec{z}_{\text{ref}}\|_2$	Same as Euler
Error Slope (Full range)	≈ 1	≈ 4
Error Slope (Initial region)	Computed via linear regression	Computed via linear regression
Absolute Error vs. Time	Plotted for x, z, v_x, v_z	Plotted for x, z, v_x, v_z
Error Statistics	Max, Mean, Final error per variable	Max, Mean, Final error per variable
Comparison Reference	`solve_ivp` (numerical solution)	`solve_ivp` (numerical solution)

3.2 Strategies used

For the resolution of these problems, the following code file structure was adopted:

1. `euler.py`
   File that deals with the resolution through the Forward Euler method.

- `initialize(x0, vx0, z0, vz0)`
  Initializes the global state variables with the projectile's initial position and velocity.
- `observe(t)`
  Appends the current state and time to their respective result arrays for later plotting or analysis.
- `update()`
  Computes the next position and velocity using the Forward Euler method, based on the current state.
- `main(x0=10, z0=5, vx0=70, vz0=30, u_val=3, m_val=1, g_val=9.8, delta=1e-3, t_final=10000, if_plot=True)`
  Runs the full Euler integration loop, stops when the projectile hits the ground, and optionally plots the trajectory and velocities.

2. `runge_kutta.py`
   File that deals with the resolution through the 4^{th} order Runge-Kutta method.
   - `initialize(x0, vx0, z0, vz0)`
     Initializes the global state variables similarly to `euler.py`.
   - `observe(t)`
     Stores the current position, velocity, and time for visualization or comparison purposes.
   - `update()`
     Applies the 4^{th}-order Runge-Kutta method to compute updated position and velocity using four intermediate stages for each.
   - `main(x0=10, z0=5, vx0=70, vz0=30, u_val=3, m_val=1, g_val=9.8, delta=1e-3, t_final=10000, if_plot=True)`
     Executes the RK4 integration loop, halts when the projectile reaches the ground, and can display plots of motion and speed.
3. `numerical.py`
   File that deals with the 'numerical' solution through Python's built in SciPy's adaptive integrator, which will act as the 'ideal' description of the projectile's trajectory.
   - `projectile_with_drag(t, y, u, m, g)`
     Defines the system of first-order ODEs governing projectile motion with drag, used as input to `solve_ivp`.
   - `hit_ground(t, y, *args)`
     Event function that detects when the projectile hits the ground (`z = 0`) and stops integration.
   - `main(x0=10, z0=5, vx0=70, vz0=30, u_val=3, m_val=1, g_val=9.8, delta=1e-3, t_final=10000, if_plot=True)`
     Calls SciPy's adaptive integrator `solve_ivp` with ground impact events, and optionally plots the results.
4. `main.py`
   File that deals with the user's requests. It calls upon the previous files capabilities, acting as a middle man. Furthermore, it gathers the results of their results, providing statistics, like error and precision of methods.
   - `compute_stats(label, true_vals, approx_vals)`
     Calculates the maximum, mean, and final absolute error between the reference (numerical) and approximate (Euler or RK4) values.
5. `precision.pyx0=10, z0=5, vx0=70, vz0=30, u_val=3, m_val=1, g_val=9.8, delta=np.logspace(-4, -1, num=1000)[::-1], t_final=10000, if_plot=True`
   File that computes the global error for the Euler and Runge-Kutta methods over a range of decreasing time steps Δt, comparing them to a numerical reference solution. It then estimates the slope of $\log(\text{error})$ versus $\log(\Delta t)$, thereby confirming the theoretical convergence orders for both methods.

3.3 Results

Table 2: Comparison Statistics between Euler, RK4, and Numerical (reference)

Method - Variable	Max Error	Mean Error	Final Error
Euler x	3.7357×10^{-2}	3.5678×10^{-2}	3.7357×10^{-2}
RK4 x	1.2379×10^{-5}	1.0301×10^{-5}	1.0265×10^{-5}
Euler z	1.7560×10^{-2}	1.6647×10^{-2}	1.7560×10^{-2}
RK4 z	3.4758×10^{-7}	2.2432×10^{-7}	2.2054×10^{-7}
Euler v_x	3.2020	1.9971×10^{-2}	1.6600×10^{-4}
RK4 v_x	3.5648×10^{-4}	1.1639×10^{-6}	2.1525×10^{-9}
Euler v_z	5.3289×10^{-1}	9.1498×10^{-3}	0.0000×10^{-00}
RK4 v_z	8.2800×10^{-6}	6.1857×10^{-8}	0.0000×10^{-00}

Average Mean Error: Euler: 2.0361×10^{-2}, RK4: 2.9377×10^{-6}
Best Method Overall: RK4

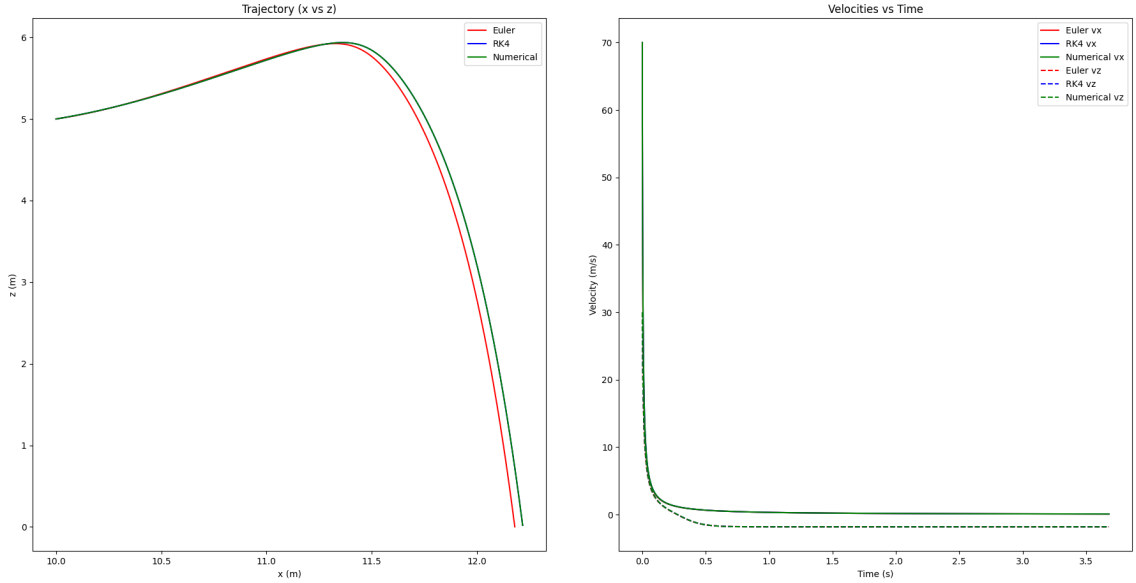

Figure 2: Trajectory and Velocities graph over Euler, Runge-Kutta and Reference methods

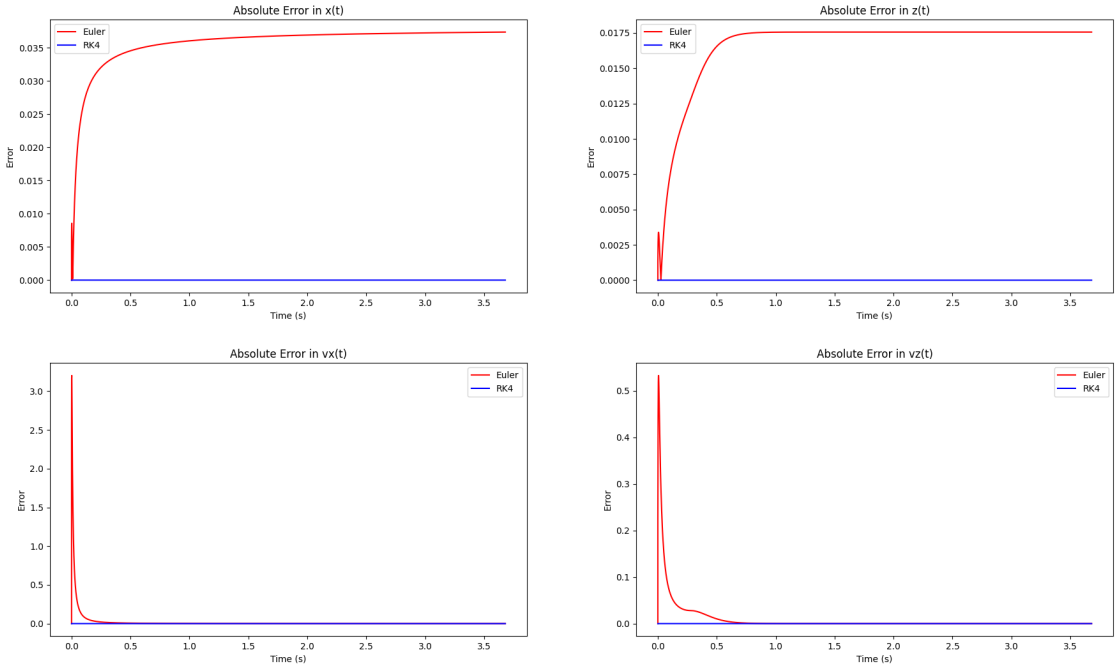

Figure 3: Error evolution for Euler and Runge-Kutta methods in reference to Reference

Method and Interval	**Estimated Slope**
Euler (full interval)	1.19
RK4 (full interval)	4.96
Euler (last 100 points)	1.11
RK4 (last 100 points)	4.05

Table 3: Estimated slope of log(error) vs log(Δt) for Euler and RK4 methods for 1000 points.

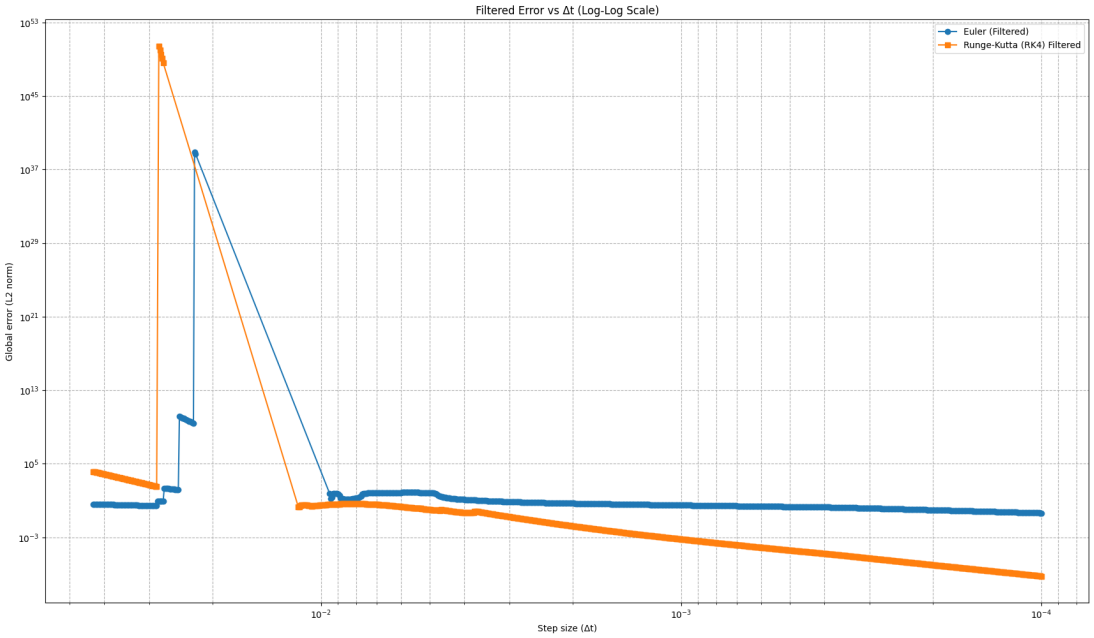

Figure 4: Error through dt (Logarithmic)

This comes to show the following conclusions and behaviors:

- Trajectory Runge-Kutta presents itself to have closer values to Reference when compared to the Euler method.
- Error As it can be deduced from the previous point, the Runge-Kutta methods shows to be far bettr than Euler's, being effectively null.
- Precision The experiments realized confirm the theoretical results, proving that Euler is a 1^{th} order error method, while Runge-Kutta 4^{th} order, as the name states, is a 4^{th} order error method. As it is expected from the previous points, the global error raises much slower in the former method when compared to the latter. It is noteworthy that both show a brusque change of error though time interval slope when the time interval goes beyond 10^{-2}, showing the limitations of both methods for greater time intervals, though Runge-Kutta remains far better despite all compared to Euler.

A Euler and Runge-Kuta

The Flying Projectiles continuous differential equations can be turned into discrete ones by establishing:

$$\frac{dx}{dt} = v_x, \quad \frac{dz}{dt} = v_z$$

Alas discrete equations using the Forward Euler method are:

$$x(t + \Delta t) = x(t) + v_x(t) \cdot \Delta t$$

$$z(t + \Delta t) = z(t) + v_z(t) \cdot \Delta t$$

For velocity updates:

$$v_x(t + \Delta t) = v_x(t) + \left(-\frac{u}{m} v_x^2(t) \cdot \text{sign}(v_x(t))\right) \cdot \Delta t$$

$$v_z(t+\Delta t) = v_z(t) + \left(-g - \frac{u}{m} v_z^2(t) \cdot \text{sign}(v_z(t))\right) \cdot \Delta t$$

As of the 4^{th} order Runge-Kutta:

$$\begin{aligned} K_1^x &= \Delta t \cdot v_x(t) \\ K_1^{v_x} &= \Delta t \cdot \left(-\frac{u}{m} v_x(t)^2 \cdot \text{sign}(v_x(t))\right) \end{aligned} \qquad \begin{aligned} K_1^z &= \Delta t \cdot v_z(t) \\ K_1^{v_z} &= \Delta t \cdot \left(-g - \frac{u}{m} v_z(t)^2 \cdot \text{sign}(v_z(t))\right) \end{aligned}$$

$$\begin{aligned} K_2^x &= \Delta t \cdot \left(v_x(t) + \frac{1}{2} K_1^{v_x}\right) \\ K_2^{v_x} &= \Delta t \cdot \left(-\frac{u}{m}\left(v_x(t) + \frac{1}{2} K_1^{v_x}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_x(t) + \frac{1}{2} K_1^{v_x}\right)\right) \end{aligned} \qquad \begin{aligned} K_2^z &= \Delta t \cdot \left(v_z(t) + \frac{1}{2} K_1^{v_z}\right) \\ K_2^{v_z} &= \Delta t \cdot \left(-g - \frac{u}{m}\left(v_z(t) + \frac{1}{2} K_1^{v_z}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_z(t) + \frac{1}{2} K_1^{v_z}\right)\right) \end{aligned}$$

$$\begin{aligned} K_3^x &= \Delta t \cdot \left(v_x(t) + \frac{1}{2} K_2^{v_x}\right) \\ K_3^{v_x} &= \Delta t \cdot \left(-\frac{u}{m}\left(v_x(t) + \frac{1}{2} K_2^{v_x}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_x(t) + \frac{1}{2} K_2^{v_x}\right)\right) \end{aligned} \qquad \begin{aligned} K_3^z &= \Delta t \cdot \left(v_z(t) + \frac{1}{2} K_2^{v_z}\right) \\ K_3^{v_z} &= \Delta t \cdot \left(-g - \frac{u}{m}\left(v_z(t) + \frac{1}{2} K_2^{v_z}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_z(t) + \frac{1}{2} K_2^{v_z}\right)\right) \end{aligned}$$

$$\begin{aligned} K_4^x &= \Delta t \cdot \left(v_x(t) + K_3^{v_x}\right) \\ K_4^{v_x} &= \Delta t \cdot \left(-\frac{u}{m}\left(v_x(t) + K_3^{v_x}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_x(t) + K_3^{v_x}\right)\right) \end{aligned} \qquad \begin{aligned} K_4^z &= \Delta t \cdot \left(v_z(t) + K_3^{v_z}\right) \\ K_4^{v_z} &= \Delta t \cdot \left(-g - \frac{u}{m}\left(v_z(t) + K_3^{v_z}\right)^2 \cdot \right. \\ &\qquad \left. \text{sign}\left(v_z(t) + K_3^{v_z}\right)\right) \end{aligned}$$

$$x(t+\Delta t) = x(t) + \frac{1}{6}(K_1^x + 2K_2^x + 2K_3^x + K_4^x) \qquad z(t+\Delta t) = z(t) + \frac{1}{6}(K_1^z + 2K_2^z + 2K_3^z + K_4^z)$$

$$v_x(t+\Delta t) = v_x(t) + \frac{1}{6}(K_1^{v_x} + 2K_2^{v_x} + 2K_3^{v_x} + K_4^{v_x}) \qquad v_z(t+\Delta t) = v_z(t) + \frac{1}{6}(K_1^{v_z} + 2K_2^{v_z} + 2K_3^{v_z} + K_4^{v_z})$$

Theoretical Precision Deduction

To understand the precision of the applied numerical methods, consider a general first-order ordinary differential equation (ODE) of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

A numerical method is said to have order p if the global error satisfies:

$$E(\Delta t) = \mathcal{O}(\Delta t^p),$$

where Δt is the integration step size.

Forward Euler Method. The explicit Euler method approximates the solution by:

$$y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n).$$

Expanding the exact solution using Taylor series about t_n:

$$y(t_n + \Delta t) = y(t_n) + \Delta t \cdot f(t_n, y(t_n)) + \frac{\Delta t^2}{2} f_t(t_n, y_n) + \mathcal{O}(\Delta t^3),$$

where $f_t(t_n, y_n) = \frac{d}{dt} f(t, y(t))$ evaluated at t_n. Comparing both expressions:

$$\tau_{\text{Euler}} = y(t_{n+1}) - y_{n+1} = \frac{\Delta t^2}{2} f_t(t_n, y_n) + \mathcal{O}(\Delta t^3),$$

implying that the **local truncation error** is $\mathcal{O}(\Delta t^2)$, and hence the **global error** is:

$$E_{\text{Euler}}(\Delta t) = \mathcal{O}(\Delta t).$$

This proves that the classical Forward Euler method is of first order accuracy.

Fourth-Order Runge-Kutta Method. The classical RK4 method applies four weighted evaluations of the function f:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_1\right), \\
k_3 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_2\right), \\
k_4 &= f\left(t_n + \Delta t, y_n + \Delta t \cdot k_3\right), \\
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned}$$

To determine the order of this method, consider the Taylor expansion of the exact solution around t_n:

$$\begin{aligned}
y(t_n + \Delta t) = y(t_n) + \Delta t \cdot y'(t_n) + \frac{\Delta t^2}{2!} y''(t_n) + \frac{\Delta t^3}{3!} y^{(3)}(t_n) \\
+ \frac{\Delta t^4}{4!} y^{(4)}(t_n) + \frac{\Delta t^5}{5!} y^{(5)}(t_n) + \mathcal{O}(\Delta t^6).
\end{aligned}$$

Each of the intermediate stages k_1, k_2, k_3, k_4 can be expanded as follows, assuming $f(t, y) = y'(t)$:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n, y_n) + \frac{\Delta t}{2}(f_t + f_y f) + \mathcal{O}(\Delta t^2), \\
k_3 &= f(t_n, y_n) + \frac{\Delta t}{2}(f_t + f_y f) + \mathcal{O}(\Delta t^2), \\
k_4 &= f(t_n, y_n) + \Delta t (f_t + f_y f) + \mathcal{O}(\Delta t^2),
\end{aligned}$$

where $f_t = \partial f / \partial t$ and $f_y = \partial f / \partial y$ evaluated at (t_n, y_n).

Substituting into the RK4 formula gives:

$$\begin{aligned}
y_{n+1} &= y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
&= y_n + \Delta t \cdot f + \frac{\Delta t^2}{2}(f_t + f_y f) + \frac{\Delta t^3}{6}\left(f_{tt} + 2f_{ty} f + f_y f_t + f_y^2 f + f_{yy} f^2\right) + \mathcal{O}(\Delta t^4).
\end{aligned}$$

This expansion matches the Taylor expansion of the exact solution up to $\mathcal{O}(\Delta t^4)$. Therefore, the **local truncation error** is:

$$\tau_{\text{RK4}} = y(t_n + \Delta t) - y_{n+1} = C \Delta t^5 + \mathcal{O}(\Delta t^6),$$

where C is a bounded constant depending on higher-order derivatives of y.

Consequently, the **global error** accumulated over $N = T/\Delta t$ steps satisfies:

$$E_{\text{RK4}}(\Delta t) = \mathcal{O}(\Delta t^4).$$

This proves that the classical Runge-Kutta method is of fourth-order accuracy.

B Additional Figures of Flying Projectiles

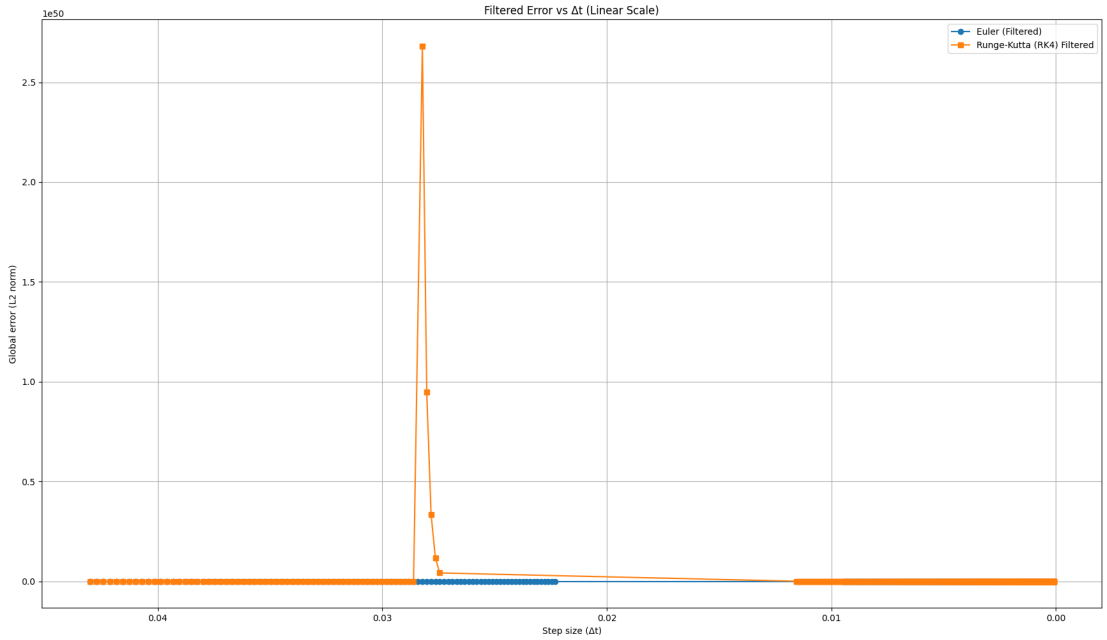

Figure 5: Error through dt (Linear)

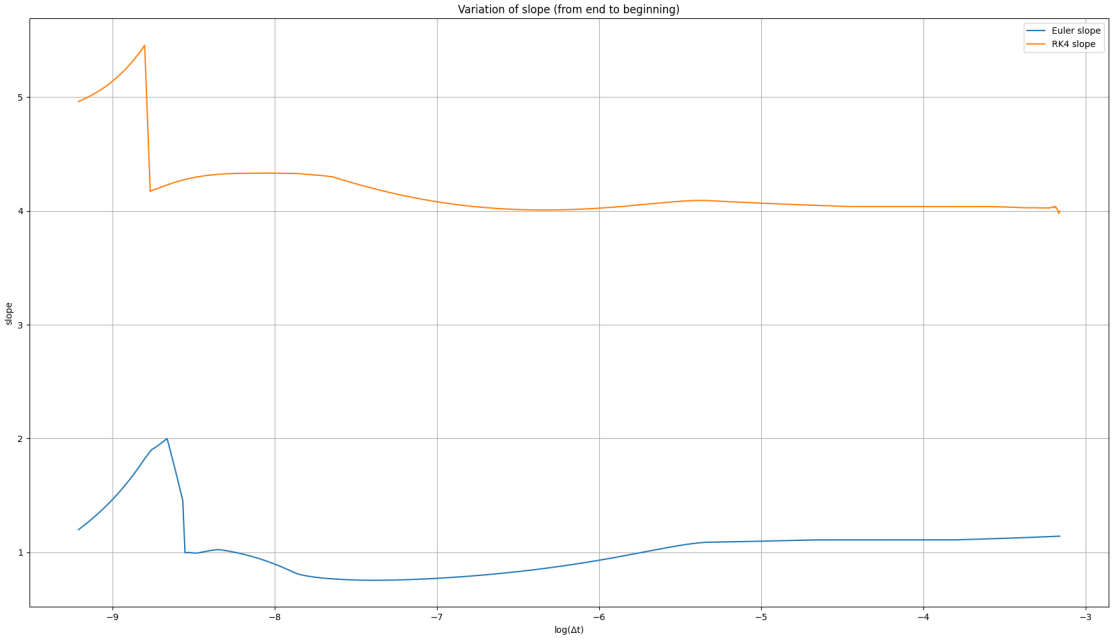

Figure 6: Slope Variation Counting from End Point to Beginning

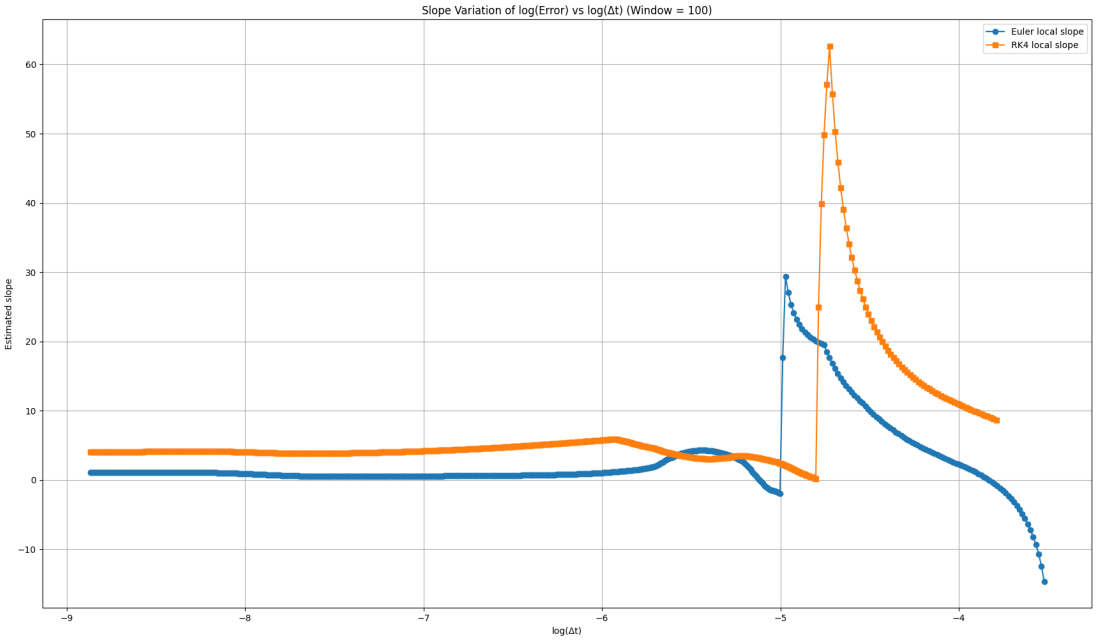

Figure 7: Slope Variation of Logarithmic Error through dt (Visualization window of 100 points)