Timofey Zabelin ITMI20SP

USING NUMERICAL METHODS AND PARALLEL PROGRAMMING FOR SIMULATING PHOTON TRAJECTORIES IN ASTROPHYSICAL ENVIRONMENTS

Bachelor's Thesis

Bachelor of Engineering

Information Technology

2024

South-Eastern Finland
University of Applied Sciences

Degree: Bachelor of Engineering
Author's Name: Timofey Zabelin
Thesis title Using numerical methods and parallel programming for simulating photon trajectories in astrophysical environments
Year: 2024
Total Number of Pages: 32
Supervisors: Vuohelainen Reijo

Abstract
The objective of this thesis was to evaluate the efficiency of various numerical methods in simulating photon trajectories in astrophysical environments, specifically near a Kerr black hole. This study employed numerical methods integrated with NVIDIA CUDA technology to compute the solutions to a system of ordinary differential equations (ODEs) that model these photon movements. Initially, previous analyses of damping systems suggested that higher-order Runge-Kutta methods could offer computational time performance comparable to the fourth order Runge-Kutta (RK4). Motivated by these findings, this research aimed to determine whether similar efficiency could be achieved in more complex astrophysical simulations.

Quantitative methods were utilized to implement and compare the second, third, fourth, and eighth Order Runge-Kutta methods, along with the Adams-Bashforth second-order method. The implementation was carried out on a parallel programming architecture to leverage the computational power of GPUs. Contrary to initial assumptions, the results indicated that higher-order methods such as the Eighth Order Runge-Kutta (RK8) incurred longer computational times compared to RK4. Surprisingly, the Adams-Bashforth method exhibited the longest computation time, even exceeding that of RK8.

The study concluded that while higher-order numerical methods increased accuracy, they did not improve computational efficiency within the context of this specific astrophysical simulation. The findings suggest that lower-order methods might be more suitable for large-scale simulations where both accuracy and computational speed are critical. This research contributes to the field by clarifying the computational trade-offs involved in using advanced numerical methods for simulating complex astrophysical phenomena.

Keywords: numerical methods, parallel programming, photon trajectories, Kerr black hole, CUDA

CONTENTS

1 INTRODUCTION

Studying numerical methods is a key part of computational science. It's crucial for solving complex problems in many scientific fields, from engineering to astrophysics. This thesis aims to explore and implement various numerical algorithms to solve differential equations. It will also evaluate their performance and accuracy in the context of astrophysics, specifically simulating photon trajectories in environments affected by strong gravitational fields, like those near black holes.

There are two main reasons for choosing astrophysical simulations as the application domain for these numerical methods:

1. The dynamics involved in astrophysical calculations, such as the bending of light near massive objects and the mapping of space curvature under general relativity, present challenging problems that require robust numerical solutions.
2. These scenarios provide a rigorous testing ground for various algorithms, where the precision and efficiency of numerical methods can be critically evaluated in extreme conditions.

By focusing on the detailed implementation and comparison of these methods, this thesis will offer a comprehensive guide for other researchers interested in numerical simulations of astrophysical or other similarly complex environments. Furthermore, the development of a CUDA-based application to simulate and visualize these trajectories serves as a practical demonstration of integrating computational physics with high-performance computing.

1.1 My rationale for choosing this topic

On a personal level, this topic is chosen out of a desire to deepen understanding of both computational techniques and astrophysical theories. The intersection of computational science and astrophysics offers a rich field of study that promises substantial academic and practical rewards. Developing a this application enhances practical skills in programming and high-performance computing, which are valuable in today's technology-driven research landscape.

1.2 Objectives

The main objectives of this research are:

1.2.1 Implementing various numerical methods

This thesis aims to integrate and execute several numerical algorithms, especially those that can handle the complexities involved in simulating trajectories under the influence of gravitational fields. The methods include Runge-Kutta and Adams-Bashforth methods.

1.2.2 Error analysis

In addition to computational efficiency, this work will assess the accuracy of each numerical method through error analysis. The relative error metrics will be carefully recorded to identify which algorithms provide the most reliable results for simulation.

1.2.3 Performance benchmarking

A key part of this thesis will involve benchmarking the implemented methods against each other to evaluate their computational efficiency using CUDA for parallel processing. This will help determine the most effective algorithms under different simulation conditions.

2 ASTROPHYSICAL SIMULATIONS IN MODERN WORLD

Astrophysical simulations play a pivotal role in the modern understanding of the universe. They are fundamental tools that allow scientists to study celestial phenomena that cannot be replicated in laboratories or directly observed due to their scale or distance. As we delve deeper into this field, it becomes clear that the development and refinement of numerical methods are not just academic exercises but are crucial to expanding our knowledge of the cosmos.

2.1 Evolution of astrophysical simulations

The field of astrophysical simulations has evolved dramatically over the past few decades. Initially, these simulations were limited to simplistic models due to the computational constraints of the time. However, with the advent of more powerful computers and sophisticated algorithms, simulations have become increasingly complex and realistic. Today, they integrate a variety of physical processes, from hydrodynamics and magnetohydrodynamics to general relativity and nuclear physics, enabling more detailed and accurate models of stars, galaxies, and large-scale structures in the universe.

2.2 The Role of high-performance computing

The exponential growth of high-performance computing (HPC) has been a major catalyst in this evolution. HPC facilities around the world now routinely

run simulations that require processing vast amounts of data at high speeds. The use of GPUs, in particular, has transformed simulation capabilities, enabling the processing of complex calculations at speeds previously unattainable. This high computational power allows for the simulation of highly dynamic and chaotic systems such as supernovae explosions, black hole mergers, and galaxy formation.

2.3 Challenges and innovations

Despite significant advancements, several challenges remain. The accuracy of simulations is continually tested against the precision of observational data. As telescopes and detectors become more capable, simulations must also improve to match the new levels of observational accuracy. This creates a continuous cycle of improvement that drives both observational and theoretical astrophysics forward.

Moreover, as the complexity of simulations increases, so does the need for more sophisticated numerical methods and algorithms. Innovations in numerical analysis, such as adaptive mesh refinement, higher-order integration techniques, and machine learning models for predicting physical interactions, are vital for advancing simulation technology. These methods help in managing the trade-offs between accuracy, computational load, and real-time processing needs.

2.4 Interdisciplinary impact

The impact of astrophysical simulations extends beyond astrophysics, influencing other scientific fields such as climate science, oceanography, and even biology. Techniques developed for astrophysics are often applicable in these other domains, where similar complex systems are studied. Furthermore, the technology developed for astrophysical simulations, particularly in terms of software and visualization tools, has broader applications in data analysis and graphical processing.

3 INTRODUCTION TO NUMERICAL METHODS

In the world of mathematics and computing, numerical methods play a vital role in solving complex problems that cannot be easily tackled using traditional analytical approaches. These powerful tools are designed to approximate solutions to differential equations, fundamental mathematical constructs used to model various phenomena in fields such as physics, engineering, finance, and even cutting-edge areas like astrophysics and cosmology (Gilat & Subramaniam 2000).

3.1 Runge-Kutta methods

When it comes to numerical techniques, the Runge-Kutta methods stand out as some of the most widely used and effective tools for solving ordinary differential equations (ODEs). An ODE is essentially an equation that involves a function of one independent variable and its derivatives. In its general form, an ODE can be expressed as:

$$\frac{dy}{dt} = f(t, y(t)) \tag{1}$$

where $y(t)$ represents the unknown function (also known as the dependent variable), t is the independent variable, and f is a given function that defines the relationship between the variables.

3.1.1 Overview

The popularity of Runge-Kutta methods can be attributed to their ability to maintain a perfect balance between complexity and accuracy. This makes them the go-to choice for many practical applications. The core idea behind these methods is to generate a series of predictions for the dependent variable, gradually refining each prediction by incorporating an increment based on the weighted average of derivatives calculated at various points within the interval (Butcher 2016).

3.1.2 Formulation of Runge-Kutta methods

A typical Runge-Kutta method step from y_n to y_{n+1} is formulated as:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \tag{2}$$

where h is the step size, s the number of stages, b_i the weights, and k_i the increments defined by:

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + c_2 h, y_n + a_{21} h k_1), \\
&\vdots \\
k_s &= f\left(t_n + c_s h, y_n + \sum_{j=1}^{s-1} a_{sj} h k_j\right).
\end{aligned}$$

The coefficients c_i and a_{ij} specify the particular Runge-Kutta method being used (Hairer & Wanner 1993).

3.2 Adams-Bashforth methods

In addition to the Runge-Kutta methods, the Adams-Bashforth methods are another class of numerical integrators used extensively for solving ordinary differential equations (ODEs). Unlike the Runge-Kutta methods, which are explicit single-step methods, the Adams-Bashforth methods belong to the category of multistep methods. This means they use information from multiple previous points to predict the future value of the dependent variable.

The main advantage of Adams-Bashforth methods is their efficiency in solving problems where high precision over large intervals is required. They are particularly well-suited for smooth problems where using historical data can significantly reduce the computation needed per step. The general form of an Adams-Bashforth method for step n+1 is given by:

$$y_{n+1} = y_n + h \sum_{i=0}^{m-1} \beta_i f(t_{n-i}, y_{n-i}) \tag{3}$$

where m is the number of steps (or the order of the method), beta are the constant coefficients predetermined for the method, and h is the step size. The coefficients are derived from the interpolation of the derivative function f at previous time steps, thereby projecting the future value of y (Gilat & Subramaniam 2000). This reliance on past values makes Adams-Bashforth methods particularly effective in scenarios where the differential equation does not change drastically over successive intervals.

4 ASTROPHYSICS SIMULATIONS

Astrophysical simulations serve as a fundamental tool for astronomers and cosmologists, offering a digital telescope with which to explore and hypothesize about phenomena far beyond our current observational capabilities. These simulations are pivotal in advancing our understanding of the universe's most enigmatic entities, such as black holes, neutron stars, and the very nature of the cosmos itself. By combining complex theoretical physics with advanced computational techniques, these simulations allow researchers to not only confirm theories but also predict new phenomena and guide future observations.

4.1 Understanding black holes: the Schwarzschild and Kerr metrics

Black holes are among the most fascinating subjects in astrophysics, characterized by gravitational fields so strong that nothing, not even light, can escape their pull once crossed the event horizon. The Schwarzschild metric provides solutions to Einstein's field equations in a vacuum for non-rotating

black holes and is crucial for understanding basic black hole properties. In contrast, the Kerr metric extends this framework to rotating black holes, which are more common in the universe due to angular momentum conservation in collapsing massive stars. This rotation leads to complex phenomena such as frame-dragging, where the spacetime itself is twisted, affecting the trajectories of matter and light near the black hole.

4.2 Equations for simulating photon trajectories

Simulating the environment around black holes is essential for visualizing and predicting the behavior of matter and light under extreme gravitational conditions. The equations derived from the Kerr metric, as mentioned, are vital for modeling photon trajectories around rotating black holes. These trajectories help in predicting the appearance of black holes and the structure of the surrounding accretion disks, crucial for interpreting the observational data from instruments like the Event Horizon Telescope, which captured the first image of a black hole's event horizon. The work Repin et al. 2018 provides us with a system of differential equations that describe this motion:

$$
\begin{aligned}
\frac{dt}{d\sigma} &= -\left(\frac{a^2\sin^2\theta - \xi}{\Delta} + \frac{(r^2+a^2)(r^2+a^2-\xi a)}{\Delta}\right),\\
\frac{dr}{d\sigma} &= r_1,\\
\frac{dr_1}{d\sigma} &= 2r + (a^2 - \xi^2 - \eta)r + (a-\xi)^2 + \eta,\\
\frac{d\theta}{d\sigma} &= \theta_1,\\
\frac{d\theta_1}{d\sigma} &= \cos\theta\left(\frac{\xi^2}{\sin^3\theta} - a^2\sin\theta\right),\\
\frac{d\phi}{d\sigma} &= -\left(\frac{(a-\xi\sin^2\theta)}{\Delta} + \frac{a\Delta}{R^2+a^2-\xi a}\right).
\end{aligned}
$$

You can see my implementation of this system of equations in C++ code in Figure 1

```
struct  KerrTrajectory
{

        const double a = 0.9982;

        /*operator () defines system of differential equations*/
        __device__ void operator()(const double *args, double *k)
        {
                k[0] = args[2];
                k[1] = -2 * a * a * args[7] * args[1] * args[1] * args[1] + 3 * ((a - args[8])*(a - args[8]) + args[7]) * args[1] * args[1] + (a * a - args[8] * args[8] - args[7]) * args[1];
                k[2] = args[4];
                k[3] = cos(args[3]) * (args[8] * args[8] / pow(sin(args[3]), 3) - a * a * sin(args[3]));
                k[4] = -1.0 * (a - args[8] / pow(sin(args[3]), 2)) + a * (1 / (args[1] * args[1]) + a * a - args[8] * a) / (1 / (args[1] * args[1]) - 2 / args[1] + a * a);
                k[5] = -1.0 * a * (a * sin(args[3]) * sin(args[3]) - args[8]) + (1 / (args[1] * args[1]) + a * a) * (1 / (args[1] * args[1]) + a * a - args[8] * a) / (1 / (args[1] * args[1]) - 2 / args[1] + a * a);
                k[6] = 0;
                k[7] = 0;
        }

        __host__ void initial_derivatives(const double* args, double* k)
        {
                k[0] = args[2];
                k[1] = -2 * a * a * args[7] * args[1] * args[1] * args[1] + 3 * ((a - args[8]) * (a - args[8]) + args[7]) * args[1] * args[1] + (a * a - args[8] * args[8] - args[7]) * args[1];
                k[2] = args[4];
                k[3] = cos(args[3]) * (args[8] * args[8] / pow(sin(args[3]), 3) - a * a * sin(args[3]));
                k[4] = -1.0 * (a - args[8] / pow(sin(args[3]), 2)) + a * (1 / (args[1] * args[1]) + a * a - args[8] * a) / (1 / (args[1] * args[1]) - 2 / args[1] + a * a);
                k[5] = -1.0 * a * (a * sin(args[3]) * sin(args[3]) - args[8]) + (1 / (args[1] * args[1]) + a * a) * (1 / (args[1] * args[1]) + a * a - args[8] * a) / (1 / (args[1] * args[1]) - 2 / args[1] + a * a);
                k[6] = 0;
                k[7] = 0;
        }

        /*sets initial values to variables in system according to given distance, deflection on two axes and angle*/
        void initial_conditions(double r_0, double b_y, double b_z, double theta_init, double* y);

};
```

Figure 1. Kerr trajectory implementation

4.3 Challenges of simulating black holes

The non-linear and complex nature of the equations describing photon trajectories near black holes requires advanced numerical methods for accurate simulation, emphasizing the need for high precision and stability due to the extreme spacetime curvature.

5 GPUS IN SCIENTIFIC COMPUTING

Graphics Processing Units (GPUs) have transcended their initial role in rendering graphics for gaming to become pivotal in scientific computing. Their architecture, designed for handling multiple tasks simultaneously, makes them exceptionally efficient for algorithms that can be parallelized. This capability is especially crucial in scientific research, where vast amounts of data and complex calculations are common.

5.1 Parallel processing

The core strength of GPUs lies in their ability to perform thousands of operations concurrently. Unlike Central Processing Units (CPUs), which have a few cores optimized for sequential serial processing, GPUs possess hundreds to thousands of smaller cores designed for parallel processing. This structure allows them to handle multiple calculations at once, drastically reducing the time required for data-intensive tasks. For instance, tasks that would take days on CPUs can be completed in hours or even minutes on GPUs, significantly accelerating the research cycle.

5.2 Complex system simulations

In fields like climate science, physics, and bioinformatics, simulations that model complex systems such as weather patterns, molecular structures, or

astrophysical phenomena are essential. These simulations involve calculations that can be distributed across many threads on a GPU, thus allowing more detailed and extensive simulations to be run faster. This capability not only speeds up research but also improves the accuracy of the models by enabling a higher resolution or more complex dynamics to be included.

6 INTRODUCTION TO CUDA

The rise of parallel computing has significantly transformed the landscape of scientific research, particularly in fields that require the handling of large datasets and complex computational tasks. CUDA (Compute Unified Device Architecture) has emerged as a leading platform enabling researchers and developers to harness the power of graphics processing units (GPUs) for general-purpose computing. This chapter provides an introductory overview of CUDA, its significance in scientific computing, and its applications in various research domains.

6.1 Understanding CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general-purpose processing—an approach known as GPGPU (General-Purpose computing on Graphics Processing Units). CUDA gives direct access to the virtual instruction set and memory of the parallel computational elements in GPUs (Sanders & Kandrot 2010).

Key features of CUDA:

- **Parallel Processing Capabilities:** CUDA enables the execution of thousands of threads simultaneously, making it highly suitable for algorithms that can be parallelized.
- **Memory Hierarchy:** CUDA includes a hierarchy of thread groups, shared memories, and distributed memory. This architecture allows data to be shared efficiently within the GPU, minimizing the need for slow memory access to the main computer memory.
- **Scalable Programming Model:** Whether using a few cores or thousands, CUDA applications scale with the number of cores available.

6.2 CUDA architecture

The CUDA architecture consists of an array of parallel processors grouped into Streaming Multiprocessors (SMs) (Sanders & Kandrot 2010). Each SM

features a set of cores that execute threads in a SIMD (Single Instruction, Multiple Data) fashion, which is efficient for operations that apply the same operation to multiple data points simultaneously. This makes GPUs particularly effective for matrix and vector operations, which are common in scientific computing.

Components of CUDA architecture:

- **Threads and Blocks:** The basic unit of execution in CUDA is the thread. Threads are grouped into blocks, and these blocks are executed within an SM. The programmer defines the number of threads per block and the number of blocks per grid.
- **Memory Management:** CUDA provides several types of memory, each with its own scope and lifetime, including global, local, shared, and constant memory, along with registers for temporary data.
- **Kernels:** A kernel is a function declared in a CUDA program that runs on the GPU. Each thread executes an instance of the kernel.

6.3 Programming with CUDA

Programming in CUDA involves writing parallel kernels, which are functions executed on the GPU. These kernels are typically written in CUDA C/C++, which is an extension of the standard C/C++. A typical workflow includes transferring data from the host (CPU) memory to the device (GPU) memory, executing one or more kernels that process this data, and then transferring results back to the host.

Steps in CUDA programming:

1. Allocate GPU memory and initialize data.
2. Transfer data from CPU to GPU memory.
3. Execute one or more kernels to process the data.
4. Copy the results back from GPU to CPU.

6.4 Applications of CUDA

CUDA has found applications across multiple domains where intensive computations are required:

- **Physical Simulations:** From climate modeling to molecular dynamics, CUDA accelerates simulations by orders of magnitude.

- **Image and Video Processing:** Tasks such as video encoding, image segmentation, and real-time ray tracing are accelerated.
- **Machine Learning:** Deep learning frameworks like TensorFlow and PyTorch utilize CUDA for accelerating neural network training and inference.

7 IMPLEMENTATION DETAILS

This chapter delves into the practical aspects of my implementation the numerical methods providing a detailed look at the structure, design, and code architectures. I have fully designed CUDA kernel, numerical methods, and systems of ODEs code implementations.

7.1 Runge-Kutta method implementation

The Runge-Kutta methods are iterative techniques that provide a series of values which approximate the solution of an ODE at discrete points. In my implementation, templates for second, third, fourth, and eighth order methods are defined under the `Runge_Kutta` namespace. I designed them to be generic, accommodating any system of equations and data types, hence enhancing the flexibility and usability of the implementation.

7.1.1 Structure and design

Each function template within the namespace takes same arguments: a system function, an array of values of all variables at the previous step (`args`), a step size (`dx`), and the number of equations in the system (`n`). The templates use dynamic memory allocation to handle intermediate values and derivatives (k values).

- **Second order Runge-Kutta**: Implements the classical mid-point method. It uses one intermediate step (`k2`) to achieve a second-order accurate solution. This method is particularly effective for problems where higher accuracy than the Euler method is desired without a significant increase in computational complexity. You can see exact implementation in Figure 2

```cpp
template<typename System, typename precision>
__device__ void Order_2(System system, precision *args, const precision dx, const int n, const int args_number)
{
	precision *args_tmp = new precision[args_number];

	// k is different derevatives in different points for calculation
	precision *k1 = new precision[args_number - 1];
	precision *k2 = new precision[args_number - 1];

	system(args, k1);

	args_tmp[0] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] + dx * k1[i - 1];
	}
	system(args_tmp, k2);

	args[n*args_number] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args[n*args_number + i] = args[i] + dx / 2 * k1[i - 1] + dx / 2 * k2[i - 1];
	}

	delete[] args_tmp;
	delete[] k1;
	delete[] k2;
}
```

Figure 2. Runge-Kutta 2 implementation

- **Third order Runge-Kutta**: Uses three stages to calculate the next value. The coefficients are specifically chosen to optimize the error constants, thereby providing better accuracy than the second-order method with a modest increase in computational efforts. You can see exact implementation in Figure 3

```cpp
template<typename System, typename precision>
__device__ void Order_3(System system, precision *args, const precision dx, const int n, const int args_number)
{
	precision *args_tmp = new precision[args_number];

	// k is different derevatives in different points for calculation
	precision *k1 = new precision[args_number - 1];
	precision *k2 = new precision[args_number - 1];
	precision *k3 = new precision[args_number - 1];

	system(args, k1);

	args_tmp[0] = args[0] + dx / 2;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] + dx / 2 * k1[i - 1];
	}
	system(args_tmp, k2);

	args_tmp[0] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] - dx * k1[i - 1] + 2 * dx * (k2[i - 1]);
	}
	system(args_tmp, k3);

	args[n*args_number] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args[n*args_number + i] = args[i] + dx / 6 * k1[i - 1] + dx * 2 / 3 * k2[i - 1] + dx / 6 * k3[i - 1];
	}

	delete[] args_tmp;
	delete[] k1;
	delete[] k2;
	delete[] k3;
}
```

Figure 3. Runge-Kutta 3 implementation

- **Fourth order Runge-Kutta**: This is the most commonly used version due to its balance between accuracy and computational cost. It involves four stages and is known for its accuracy over moderate to long ranges of integration. You can see exact implementation in Figure 4

```cpp
template<typename System, typename precision>
__device__ void Order_4(System system, precision *args, const precision dx, const int n, const int args_number)
{
	precision *args_tmp = new precision[args_number];

	// k is different derevatives in different points for calculation
	precision *k1 = new precision[args_number - 1];
	precision *k2 = new precision[args_number - 1];
	precision *k3 = new precision[args_number - 1];
	precision *k4 = new precision[args_number - 1];

	system(args, k1);

	args_tmp[0] = args[0] + dx / 2;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] + dx / 2 * k1[i - 1];
	}
	system(args_tmp, k2);

	args_tmp[0] = args[0] + dx / 2;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] + dx / 2 * (k2[i - 1]);
	}
	system(args_tmp, k3);

	args_tmp[0] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args_tmp[i] = args[i] + dx * (k3[i - 1]);
	}
	system(args_tmp, k4);

	args[n*args_number] = args[0] + dx;
	for (int i = 1; i < args_number; i++)
	{
		args[n*args_number + i] = args[i] + dx / 6 * k1[i - 1] + dx / 3 * k2[i - 1] + dx / 3 * k3[i - 1] + dx / 6 * k4[i - 1];
	}

	delete[] args_tmp;
	delete[] k1;
	delete[] k2;
	delete[] k3;
	delete[] k4;
}
```

Figure 4. Runge-Kutta 4 implementation

- **Eighth order Runge-Kutta**: An advanced implementation that uses thirteen intermediate stages. This method is highly accurate and suitable for solving stiff or highly sensitive problems where lower-order methods may fail to provide reliable results. You can see exact implementation in Figure 5

```
template<typename System, typename precision>
__device__ void Order_8(System system, precision* args, const precision dx, const int n, const int args_number)
{
	precision* args_tmp = new precision[args_number];
	precision* k = new precision[args_number * 13];
	const precision a[13][12] = {
	{0},
	{1.0 / 18.0},
	{1.0 / 48.0, 1.0 / 16.0},
	{1.0 / 32.0, 0, 3.0 / 32.0},
	{5.0 / 16.0, 0, -75.0 / 64.0, 75.0 / 64.0},
	{3.0 / 80.0, 0, 0, 3.0 / 16.0, 3.0 / 20.0},
	{29443841.0 / 614563906.0, 0, 0, 77736538.0 / 692538347.0, -28693883.0 / 1125000000.0, 23124283.0 / 1800000000.0},
	{16016141.0 / 946692911.0, 0, 0, 61564180.0 / 158732637.0, 22789713.0 / 633445777.0, 545815736.0 / 2771057229.0, -180193667.0 / 1043307555.0},
	{39632708.0 / 573591083.0, 0, 0, -433636366.0 / 683701615.0, -421739975.0 / 2616292301.0, 100302831.0 / 723423059.0, 790204164.0 / 839813087.0, 800635310.0 / 3783071287.0},
	{246121993.0 / 1340047787.0, 0, 0, -37695042795.0 / 15268766246.0, -309121744.0 / 1061227803.0, -12992083.0 / 490766935.0, 6005943493.0 / 2108947869.0, 393006217.0 / 1396073457.0, 123872331.0 / 1001029789.0},
	{-1028468189.0 / 846180014.0, 0, 0, 8478235783.0 / 508512852.0, 1311729495.0 / 1432422823.0, -10304129995.0 / 1701304382.0, -48777925059.0 / 3047939560.0, 15336726248.0 / 1032824649.0, -45544268181.0 / 3398467696.0, 3065993473.0 / 597172653.0},
	{1858892177.0 / 718116043.0, 0, 0, -3185094517.0 / 667107341.0, -477755414.0 / 1098053517.0, -703635378.0 / 230739211.0, 5731566787.0 / 1027545527.0, 5232866602.0 / 850066563.0, -4093664535.0 / 808688257.0, 3962137247.0 / 1805957418.0, 65086358.0 / 487910083.0},
	{4083663854.0 / 4910063109.0, 0, 0, -5068492393.0 / 434740067.0, -411421997.0 / 543043805.0, 652783627.0 / 914296604.0, 11173962825.0 / 925320556.0, -13158990841.0 / 6184727034.0, 3936647629.0 / 1978049680.0, -160528059.0 / 685178525.0, 248638103.0 / 1413531060.0, 0}
	};
	const precision b[13] = { 14005451.0 / 335480064.0, 0, 0, 0, 0, -59238493.0 / 1068277825.0, 181606767.0 / 758867731.0, 561292985.0 / 797845732.0, -1041891430.0 / 1371343529.0, 760417239.0 / 1151165299.0, 118820643.0 / 751138087.0, -528747749.0 / 2220607170.0, 1.0 / 4.0 };
	const precision c[13] = { 0.0, 1.0 / 18.0, 1.0 / 12.0, 1.0 / 8.0, 5.0 / 16.0, 3.0 / 8.0, 59.0 / 400.0, 93.0 / 200.0, 5490023248.0 / 9719169821.0, 13.0 / 20.0, 1201146811.0 / 1299019798.0, 1.0, 1.0 };

	system(args, k);

	for (int j = 0; j < 13; j++) {
		args_tmp[0] = args[0] + c[j] * dx;
		for (int i = 1; i < args_number; i++) {
			precision sum = 0;
			for (int l = 0; l < j; l++) {
				sum += a[j][l] * k[l * args_number + i - 1];
			}
			args_tmp[i] = args[i] + dx * sum;
		}
		system(args_tmp, &k[j * args_number]);
	}

	args[n * args_number] = args[0] + dx;
	for (int i = 1; i < args_number; i++) {
		precision sum = 0;
		for (int j = 0; j < 13; j++) {
			sum += b[j] * k[j * args_number + i - 1];
		}
		args[n * args_number + i] = args[i] + dx * sum;
	}

	delete[] args_tmp;
	delete[] k;
}
}
```

Figure 5. Runge-Kutta 8 implementation

7.1.2 Code analysis

The implementation of each order involves computing several intermediate derivatives (k values) at strategically chosen points within each integration step. These derivatives are used to estimate the slope that approximates the solution curve of the differential equation.

7.2 Adams-Bashforth method implementation

The Adams-Bashforth methods are explicit multistep techniques used to integrate ODEs. These methods are particularly useful when a series of past derivatives can be utilized to predict future values, thus reducing the number of function evaluations compared to single-step methods like Runge-Kutta.

7.2.1 Structure and design

In the `Adams_Bashforth` namespace, the implementation provides a second-order method. It utilizes a two-step formula which combines the slope at the current point and the slope at the previous point to estimate the next value.

- **Adams-Bashforth 2-Step Method**: This method calculates the solution by leveraging the weighted average of the first derivatives over the last two steps. It is straightforward yet provides a significant improvement over the Euler method in terms of accuracy and stability. You can see exact implementation in Figure 6

```cpp
template<typename System, typename precision>
__device__ void Adams_Bashforth_2(System system, precision* args, precision* previous_k, const precision dx, const int n, const int args_number) {
        precision* k = new precision[args_number - 1];
        precision* args_tmp = new precision[args_number];

        // Compute current derivatives
        system(args, k);

        // Update the arguments based on the Adams-Bashforth formula
        args[n * args_number] = args[0] + dx;
        for (int i = 1; i < args_number; i++) {
                args[n * args_number + i] = args[i] + dx * ((3.0 / 2.0) * k[i - 1] - (1.0 / 2.0) * previous_k[i - 1]);
        }

        // Update previous derivatives for next step
        for (int i = 0; i < args_number - 1; i++) {
                previous_k[n * (args_number-1) + i] = k[i];
        }


        delete[] k;
}
```

Figure 6. Adams-Bashforth 2 implementation

7.2.2 Integration with Runge-Kutta

A unique feature of my implementation is the integration of the Runge-Kutta method to initiate the Adams-Bashforth method. The eighth order Runge-Kutta method is used to populate the initial derivative estimates (`previous_k`), which are then used by the Adams-Bashforth method. This hybrid approach ensures that the Adams-Bashforth method starts with a highly accurate set of initial values, thus enhancing the overall accuracy and stability of the solution. You can see exact implementation in Figure 7

```cpp
//To make the first step before starting Adams-Bashforth
template<typename System, typename precision>
__device__ void RK_8(System system, precision* args, precision* previous_k, const precision dx, const int n, const int args_number)
{
        precision* args_tmp = new precision[args_number];
        precision* k = new precision[args_number * 13];
        const precision a[13][12] = {
        {0},
        {1.0 / 18.0},
        {1.0 / 48.0, 1.0 / 16.0},
        {1.0 / 32.0, 0, 3.0 / 32.0},
        {5.0 / 16.0, 0, -75.0 / 64.0, 75.0 / 64.0},
        {3.0 / 80.0, 0, 0, 3.0 / 16.0, 3.0 / 20.0},
        {29443841.0 / 614563906.0, 0, 0, 77736538.0 / 692538347.0, -28693883.0 / 1125000000.0, 23124283.0 / 1800000000.0},
        {16016141.0 / 946692911.0, 0, 0, 61564180.0 / 158732637.0, 22789713.0 / 633445777.0, 545815736.0 / 2771057229.0, -180193667.0 / 1043307555.0},
        {39632708.0 / 573591083.0, 0, 0, -433636366.0 / 683701615.0, -421739975.0 / 2616292301.0, 100302831.0 / 723423059.0, 790204164.0 / 839813087.0, 800635310.0 / 3783071287.0},
        {246121993.0 / 1340847787.0, 0, 0, -37695042795.0 / 15268766246.0, -309121744.0 / 1061227803.0, -12992083.0 / 490766935.0, 6005943493.0 / 2108947869.0, 393006217.0 / 1396673457.0, 123872331.0 / 1001029789.0},
        {-1020468189.0 / 846180014.0, 0, 0, 8478235783.0 / 508512852.0, 1311729495.0 / 1432422823.0, -10304129995.0 / 1701304382.0, -48777925059.0 / 3047939560.0, 15336726248.0 / 1032824649.0, -45442868181.0 / 3398467696.0, 3065993473.0 / 597172653.0},
        {185892177.0 / 718116043.0, 0, 0, -3185094517.0 / 667107341.0, -477755414.0 / 1098053517.0, -703635378.0 / 230739211.0, 5731566787.0 / 1027545527.0, 5232866602.0 / 850066563.0, -4093664535.0 / 808688257.0, 3902137247.0 / 1805957418.0, 65686358.0 / 487910083.0},
        {403863854.0 / 491065109.0, 0, 0, -5068492393.0 / 434740067.0, -411421997.0 / 543043805.0, 652783627.0 / 914296604.0, 11173962825.0 / 925320556.0, -13158990841.0 / 6184727034.0, 3936647629.0 / 1978049680.0, -160528059.0 / 685178525.0, 248638103.0 / 1413531060.0, 0}
        };
        const precision b[13] = { 14005451.0 / 335480064.0, 0, 0, 0, 0, -59238493.0 / 1068277825.0, 181606767.0 / 758867731.0, 561292985.0 / 797845732.0, -1041891430.0 / 1371343529.0, 760417239.0 / 1151165299.0, 118820643.0 / 751138087.0, -528747749.0 / 2220607170.0, 1.0 / 4.0 };
        const precision c[13] = { 0.0, 1.0 / 18.0, 1.0 / 12.0, 1.0 / 8.0, 5.0 / 16.0, 3.0 / 8.0, 59.0 / 400.0, 93.0 / 200.0, 5490023248.0 / 9719169821.0, 13.0 / 20.0, 1201146811.0 / 1299019798.0, 1.0, 1.0 };

        system(args, k);

        for (int j = 0; j < 13; j++) {
                args_tmp[0] = args[0] + c[j] * dx;
                for (int i = 1; i < args_number; i++) {
                        precision sum = 0;
                        for (int l = 0; l < j; l++) {
                                sum += a[j][l] * k[l * args_number + i - 1];
                        }
                        args_tmp[i] = args[i] + dx * sum;
                }
                system(args_tmp, &k[j * args_number]);
        }

        args[n * args_number] = args[0] + dx;
        for (int i = 1; i < args_number; i++) {
                precision sum = 0;
                for (int j = 0; j < 13; j++) {
                        sum += b[j] * k[j * args_number + i - 1];
                }
                args[n * args_number + i] = args[i] + dx * sum;
        }
        // Update previous derivatives for next step
        for (int i = 0; i < args_number - 1; i++) {
                previous_k[n * (args_number - 1)] = k[i];
        }
        delete[] args_tmp;
        delete[] k;
}
```

Figure 7. Runge-Kutta 8 integration implementation

8 PERFORMANCE ANALYSIS

In this chapter, I present a thorough examination of numerical methods applied to the classic problem of damped harmonic motion. The system under consideration is a second-order ordinary differential equation representing a

mass-spring-damper model, which is a quintessential system in both physics and engineering disciplines. The model is described by the following set of first-order linear differential equations:

$$\frac{dx}{dt} = y, \tag{1}$$

$$\frac{dy}{dt} = -x - \gamma y, \tag{2}$$

where $x(t)$ represents the displacement of the mass, $y(t)$ denotes its velocity, and γ is the damping coefficient that characterizes the energy dissipation of the system. For our study, we have set $\gamma = 0.1$, which corresponds to a lightly damped regime where the system experiences oscillations with amplitudes gradually decaying over time. The initial conditions for the system are chosen to be $x(0) = 1$ and $y(0) = 0$, initiating the system from a state of maximum displacement and zero velocity.

8.1 Error analysis

The absolute error of each method was calculated by comparing the numerical solutions to the analytical solution of the system. The error trends were plotted and analyzed to evaluate the performance of each method under the two different step sizes. In addition to exploring the properties of the numerical methods with a step size of $h = 0.1$, we juxtapose these findings against simulations run with a significantly larger step size of $h = 1$. The latter serves to accentuate the implications of step size choice on the stability and accuracy of numerical solutions. A step size of $h = 1$ represents a stark contrast to the natural time scale of the system's dynamics and allows for an investigation into the behavior of each numerical method under a regime of coarse temporal resolution.

8.1.1 Step size 0.1

RK2

As shown in Figure 8, the following trends can be observed:

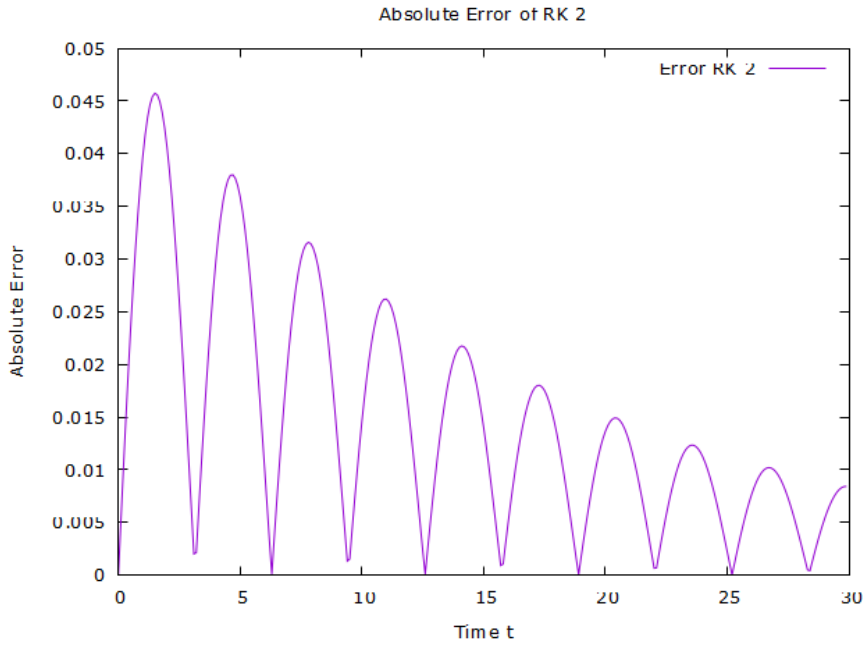

Figure 8. Runge-Kutta 2 order error with step 0.1

- Initial error: The RK2 method began with an absolute error peak around 0.045.
- Error trend: As the system evolved, the error exhibited a damping behavior, decreasing to approximately 0.01 by the end of the interval.

RK8

As shown in Figure 9, the following trends can be observed:

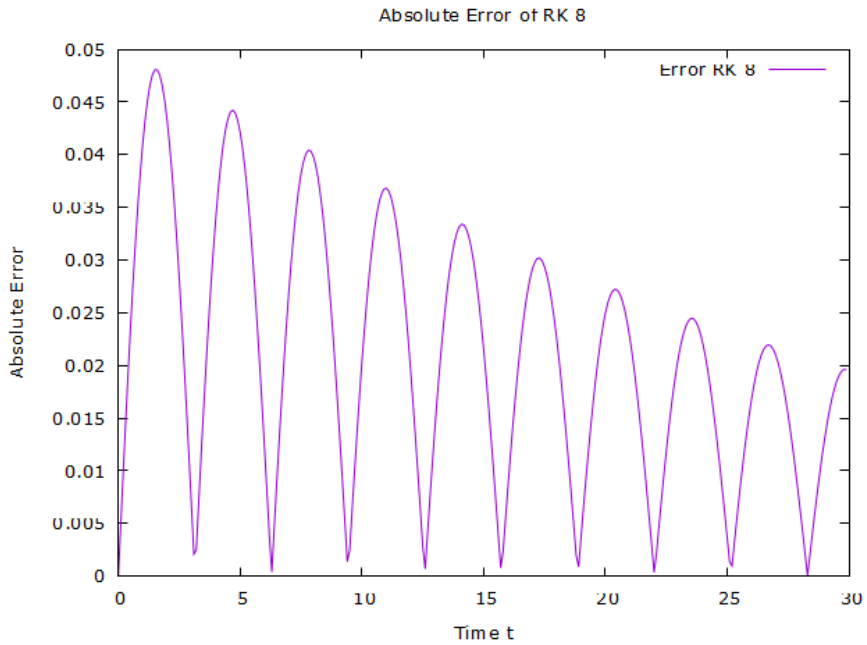

Figure 9. Runge-Kutta 8 order error with step 0.1

- Initial error: The RK8 method also started with a peak error 0.047 similar

to that of the RK2 method.

- Error trend: The error for RK8 decreased over time, maintaining a similar trend to the RK2, albeit with a slightly higher peaks and the error at the end of the interval around 0.02.

Comment

Despite expectations for higher-order numerical methods to outperform their lower-order counterparts, the RK8 method does not exhibited only improvement over the RK2 method when utilizing a step size of 0.1. This anomaly in the RK8's performance, where it behaved similarly but slightly worse than RK2, may indicate that the step size of 0.1 is already sufficiently small to bring the RK2's solution within an acceptable error threshold for this particular system. Given the light damping in the oscillator system, the RK8's additional stages and computational complexity do not translate into a significant accuracy advantage. This suggests that the benefits of higher-order methods like RK8 become more apparent when facing bigger steps, steeper gradients or more volatile system dynamics than those presented by the lightly damped oscillator.

It can be easily seen that on a big scale solution obtained by RK2 (Figure 10)

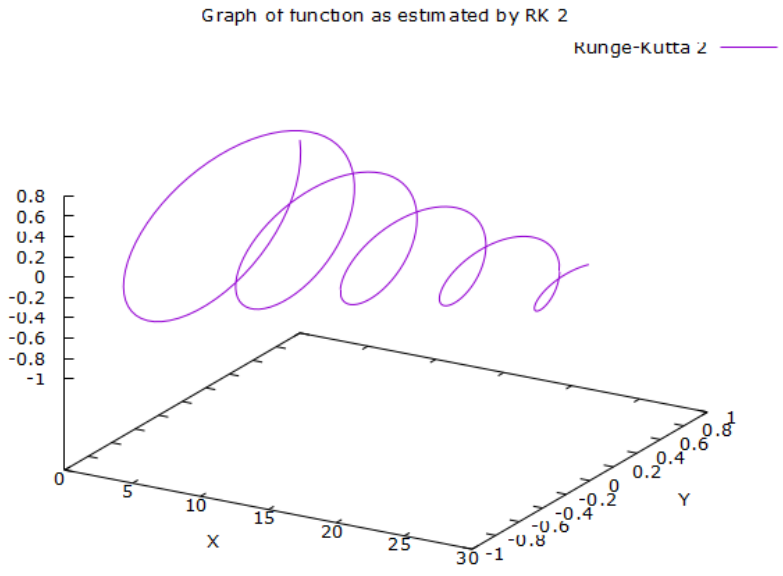

Figure 10. Graph of the solution produced by Runge-Kutta 2 with step 0.1

does not differ much from the solution obtained by RK8(Figure 11).

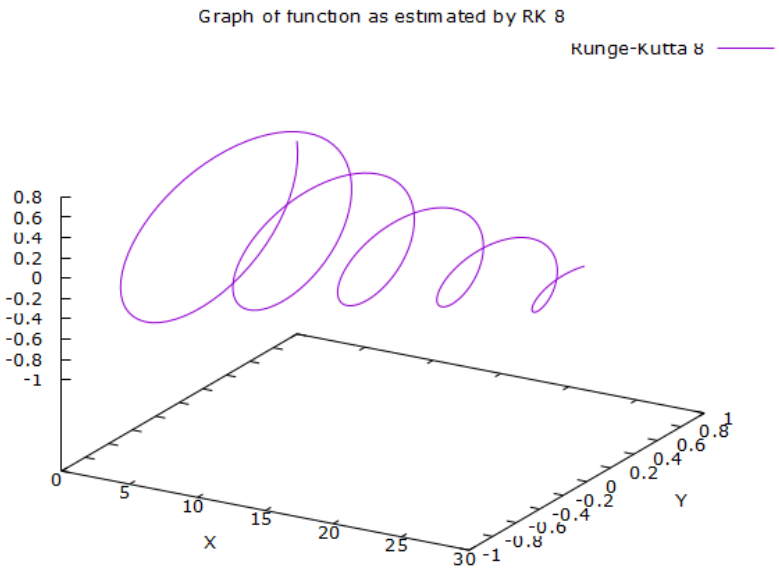

Figure 11. Graph of the solution produced by Runge-Kutta 8 with step 0.1

8.1.2 Step size 1

RK2

Figure 12 shows behaviour of RK2 when step size is increased to 1. The following trends can be observed:

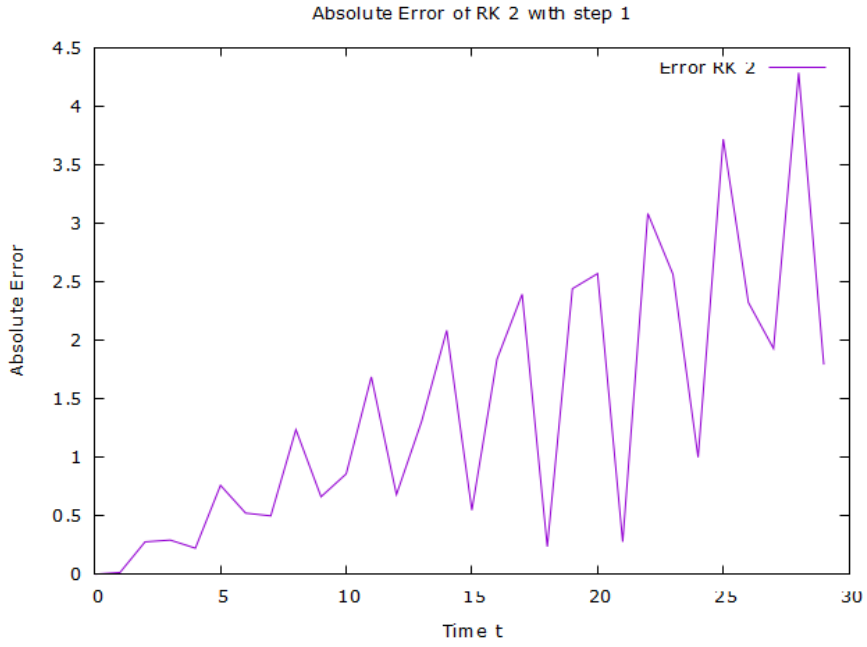

Figure 12. Runge-Kutta 2 order error with step 1

- Initial error: A significant increase in initial error to approximately 0.25.
- Error trend: A marked error growth was observed, with the final error reaching around 4.25, indicating a pronounced sensitivity to step size.

RK8

On the contrary, for RK8(Figure 13) under the same conditions:

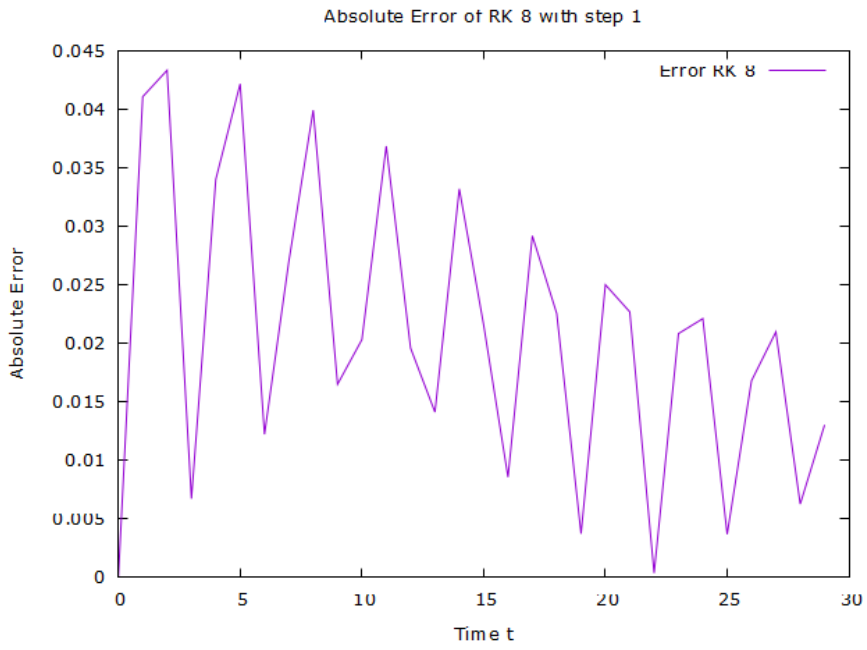

Figure 13. Runge-Kutta 8 order error with step 1

- Initial error and trend: The RK8 method's error pattern remained almost identical to the scenario with a smaller step size, indicating a robustness to step size variation.

Results of comparison between RK2 and RK8

Impact of step size on RK2

The RK2 method's sensitivity to step size is pronounced. A larger step size results in a substantial increase in both the initial error and the error growth rate. This sensitivity suggests that the RK2 method is appropriate for systems where a finer resolution is required to capture dynamic behavior accurately.

Performance of RK8 under step size variations

Conversely, the RK8 method's error displayed minimal sensitivity to the change in step size, highlighting its advanced approximation capabilities. This robustness is attributed to the inclusion of higher-order terms, which allow the method to maintain accuracy over larger step increments.

Analysis of error for RK2, RK3, RK4, RK8 and Adams-Bashforth

In an extensive comparison of numerical methods for a damped oscillator system with a step size of h=0.1, it was observed that the RK2, RK3, RK4, as

well as the Adams-Bashforth second-order method, all exhibited similar error magnitudes throughout the simulation(Figure 14).

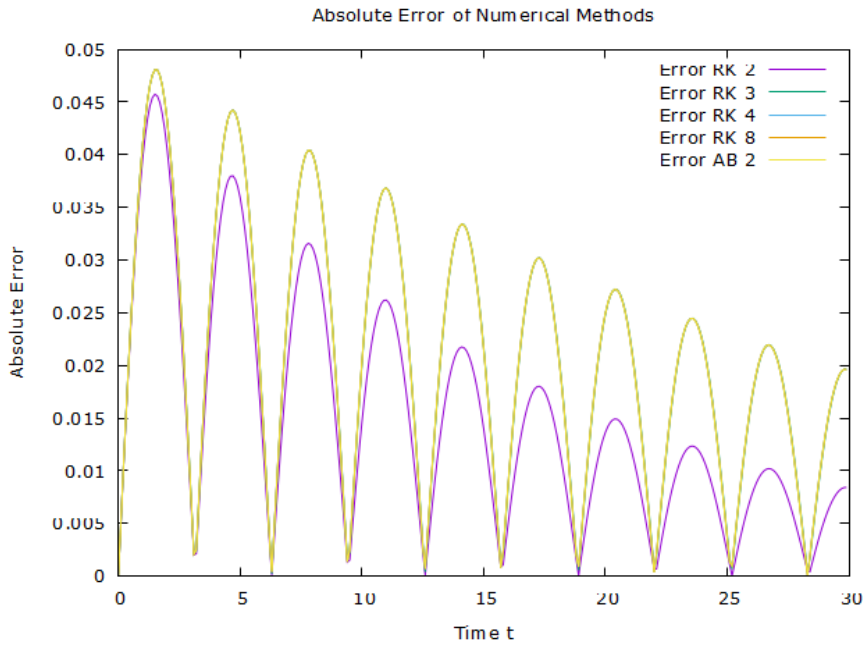

Figure 14. Absolute sizes of errors

Also it can be clearly seen on the solution plot(Figure 15).

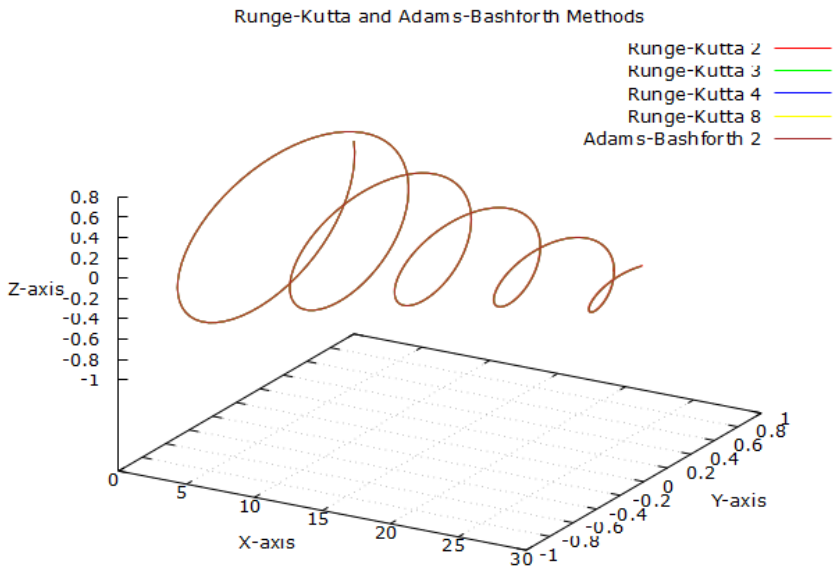

Figure 15. Graph of solutions obtained by different methods

Surprisingly, the RK2 method demonstrated the lowest error among them. This convergence in performance suggests that for the specific system dynamics at the chosen step size, the complexity of higher-order methods does not confer a significant advantage in terms of accuracy. The result emphasizes the notion

that lower-order methods can sometimes provide the most efficient solution without compromising precision, particularly in systems where the solution varies gradually and the error introduced by a smaller step size is already minimized.

8.2 Time performance analysis

In computational mathematics, the efficiency of numerical methods is paramount when solving systems of ordinary differential equations (ODEs). This chapter presents a comparative time performance analysis of various Runge-Kutta methods and the Adams-Bashforth second-order method in solving a damped harmonic oscillator system. The analysis is based on the average computation time over 10,000 steps, providing insights into the trade-offs between computational speed and the accuracy of the methods discussed earlier.

8.3 Computational time analysis

8.3.1 Runge-Kutta methods

The Runge-Kutta family of methods displayed a range of computation times, which were somewhat inversely related to their order:

- **Eighth Order Runge-Kutta (RK8)**: Despite being a higher-order method expected to have more computational overhead, RK8 recorded an average time of 526 microseconds, which is surprisingly efficient given its complexity.
- **Fourth Order Runge-Kutta (RK4)**: A standard in many applications due to its balance between accuracy and computational demand, RK4 took slightly longer, averaging 580 microseconds.
- **Third Order Runge-Kutta (RK3)**: Slightly faster, RK3 averaged 487 microseconds, demonstrating less computational overhead while still providing an acceptable accuracy level.
- **Second Order Runge-Kutta (RK2)**: The quickest among the Runge-Kutta methods with an average time of 402 microseconds, RK2 proved to be the most computationally efficient within the family, aligning with its lower complexity.

8.3.2 Adams-Bashforth method

- **Adams-Bashforth Second Order**: As shown in Figure 16, the data indicates that the Adams-Bashforth method outperformed all Runge-Kutta methods in terms of speed, with an average computation time of 278

microseconds. This method's efficiency is notable, particularly given its predictive nature and lower complexity relative to higher-order methods.

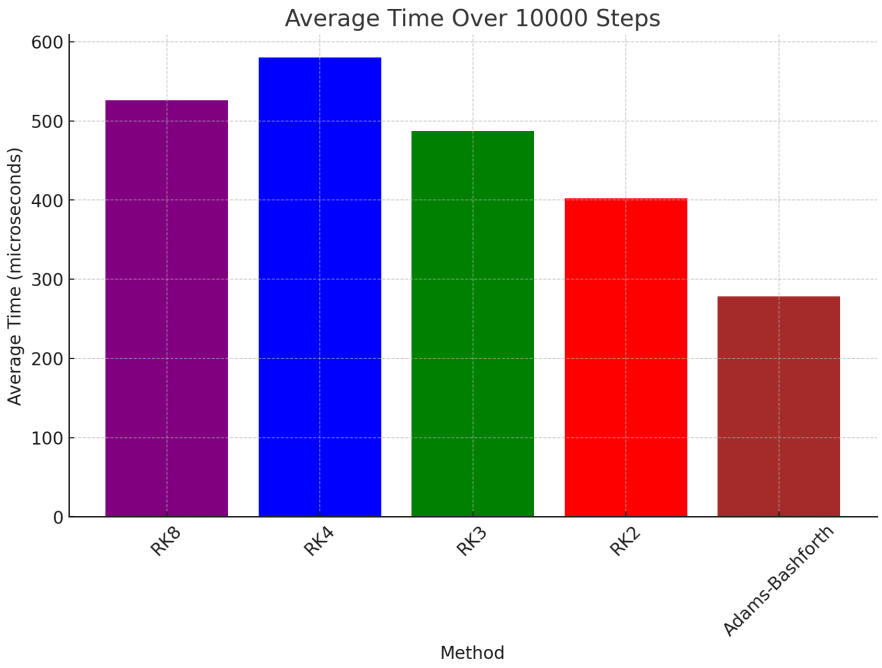

Figure 16. Average time over 10000 steps

8.3.3 Conclusions

The findings reveal intriguing aspects of the computational efficiency of each method:

- **Higher-order versus lower-order**: Higher-order methods did not necessarily correspond to longer computation times compared to their lower-order counterparts. Especially it is to be noted that RK4 unexpectedly took longer than RK8. This could be due to the efficiency of the implementation or the nature of the system being solved.
- **Efficiency of RK2**: The RK2's performance, in terms of speed, was exemplary among the Runge-Kutta methods, suggesting that for certain systems, lower-order methods can be optimized to rival even explicit multistep methods like Adams-Bashforth.
- **Superiority of Adams-Bashforth**: The Adams-Bashforth method demonstrated the best computational time, which aligns with theoretical expectations as it uses information from previous steps to predict future values without the need for intermediate calculations like those required by Runge-Kutta methods.

8.4 Main results of comparison tests

The comparative analysis of various numerical techniques for ODEs highlights the potential advantages of using higher-order Runge-Kutta methods in specific cases. Given that the RK8 method has the same computational duration as the RK4 method in systems involving damped oscillators, it may enhance accuracy when addressing the system from Repin et al. 2018.

9 PERFORMANCE EVALUATION OF NUMERICAL METHODS FOR SOLVING ASTROPHYSICS ODES ON GPU

The resolution of ordinary differential equations (ODEs) using numerical methods is a cornerstone of computational physics, particularly for systems requiring precise and rapid computations. Previous studies on a damping system suggested minimal performance differences between RK4 and RK8, indicating that higher-order numerical methods might offer comparable time efficiency. This intriguing outcome led me to a hypothesis that applying higher-order methods to more complex systems might not necessarily result in efficiency loss. To investigate this hypothesis, this chapter explores the implementation and performance of several numerical methods, including RK2, RK3, RK4, RK8, and Adams-Bashforth second-order method (AB), in solving a system of four equations that model the dynamics of photons in in the gravitation field of a rotating black hole. The analysis aims to determine if the enhanced accuracy provided by higher-order methods can be achieved without sacrificing computational speed, leveraging the parallel processing capabilities of GPUs.

9.1 Methodology and system setup

The ODE system studied models the dynamics of particles in a physical system described by four equations. Using CUDA for parallel computations, the system was simulated on a GPU where each numerical method was implemented and tested under the same initial conditions and computational settings. The performance was measured in terms of the total time taken to complete 1200 steps of the simulation.

9.2 Results of performance evaluation

The computational performance and accuracy of each numerical method were evaluated in the context of solving a system of four equations on NVIDIA RTX 4060 GPU. The results from the simulation, including computation times(Figure 17)

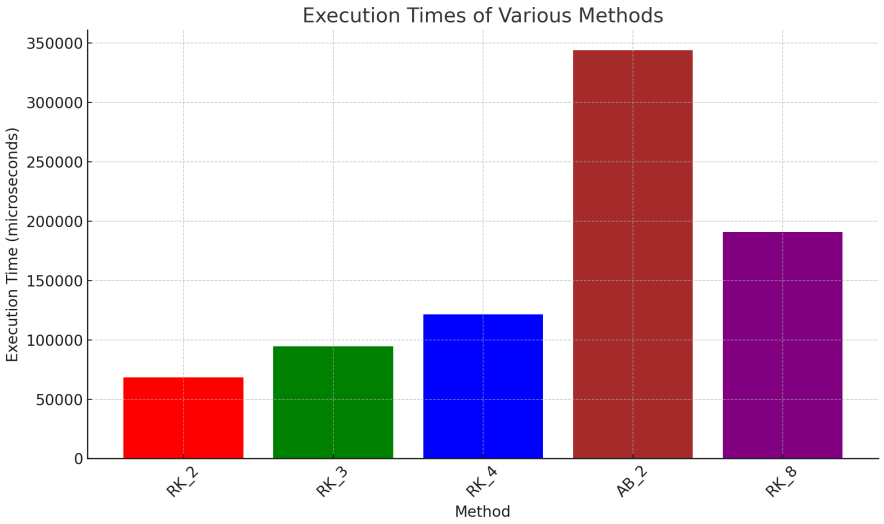

Figure 17. Execution times of various methods

and the average distance errors(Figure 18)

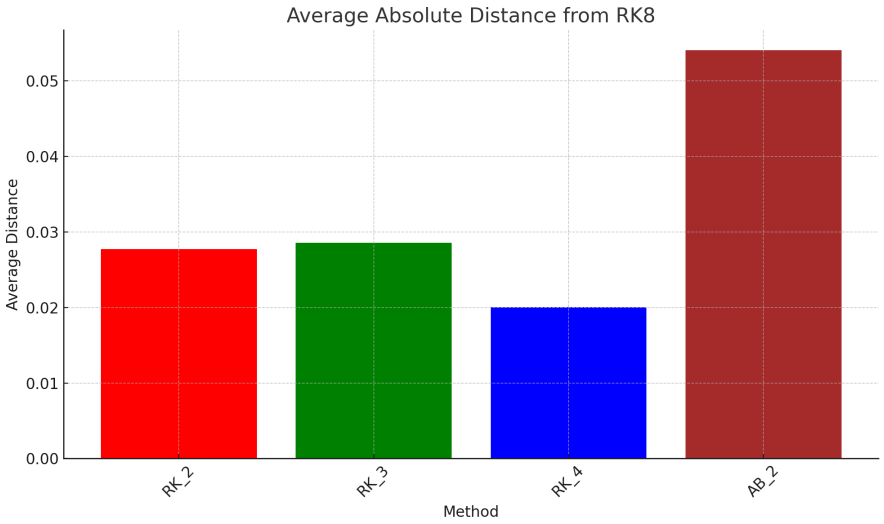

Figure 18. Average absolute distance from RK 8 results

observed for each method, are detailed below:

- **Adams-Bashforth (AB)**: Computation time was 344,011 microseconds, with an average distance error of 0.0540800297467361. This error magnitude suggests that while AB is fast, it may not deliver the same accuracy as some higher-order methods.
- **Eighth order Runge-Kutta (RK8)**: Computation time was 191,049 microseconds. The error data for RK8 is not available because the system does not have analytical solution so error was measured against RK8.
- **Fourth order Runge-Kutta (RK4)**: Computation time was 121,315 microseconds, with an average distance error of

0.020074345609930554. This shows that RK4 offers a good balance between computational speed and accuracy.

- **Third order Runge-Kutta (RK3)**: Computation time was 94,661 microseconds, with an average distance error of 0.028547150213701397. This result indicates that RK3 is faster than RK4 but slightly less accurate.
- **Second order Runge-Kutta (RK2)**: Computation time was 68,294 microseconds, with an average distance error of 0.027735712411645835. RK2 is the fastest among the methods tested and provides accuracy comparable to that of RK3.

The computation time increases with the order of the Runge-Kutta methods, with higher-order methods taking more time. The Adams-Bashforth method, despite being a different approach, has the highest computation time among the tested methods.

10 ANALYSIS OF RESULTS

These results provide valuable insights into the trade-offs between speed and accuracy across different numerical methods when utilized on a GPU. The data suggests that:

- RK 2 and RK 3 had the same error rates. It make RK 2 preferable due to lower computation time. These lower-order methods are highly efficient on GPUs, offering the quickest computation times with reasonably low error rates. This efficiency makes them attractive for simulations where speed is a critical factor.
- The higher-order method, RK4, presents a compelling case for applications requiring a balanced approach to speed and accuracy. Its moderate computational demand combined with lower error rates makes it suitable for more precise simulations. RK 4 provides the best time-to-error ratio.
- Adams-Bashforth's higher error rate might limit its applicability for precision-critical simulations, despite its lower computational time. This indicates that while it is a fast method, its predictive nature may not always align with accuracy demands in complex systems.

The investigation into the performance of various numerical methods for solving a system of four equations on a GPU yielded conclusive results that diverge from the initial hypothesis. Contrary to the assumption that higher-order methods like RK8 could match the time efficiency of lower-order methods based on previous findings with a damping system, the actual

computation times recorded during this study indicate a clear trend: higher-order methods consistently took longer to compute than their lower-order counterparts.

10.1 Additional insights from physical application

In terms of the physical application of these methods to astrophysical phenomena, the results from Figure 19

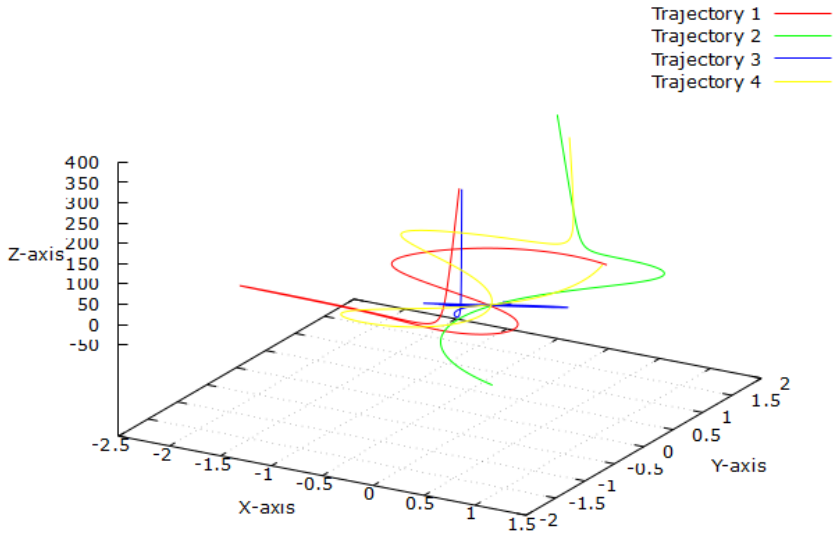

Figure 19. Trajectories visualisation

and Figure 20 are revealing.

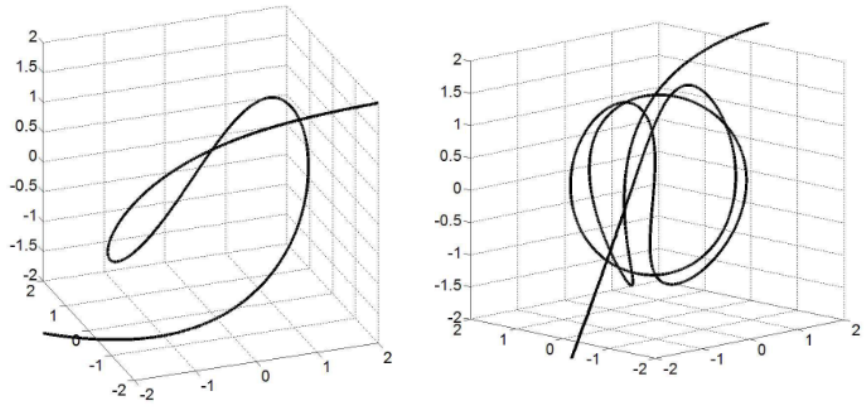

Figure 20. Trajectories from the Repin et al. 2018

Overall he trajectories of photons simulated show behavior similar to that described in the research from Repin et al. 2018, which underscores the effectiveness of these numerical methods in replicating scientifically observed phenomena. However, an exception was noted in trajectory 3, which is

identified as degenerate. Unlike the expected behavior, this particular photon trajectory suggests that the particle would have fallen into the black hole, but the simulation was not designed to halt under such circumstances. This highlights a limitation in the current simulation setup where extreme physical scenarios, such as crossing the event horizon, are not adequately managed.

11 CONCLUSIONS

The core objective of this thesis was to evaluate the efficiency and accuracy of various numerical methods for simulating photon trajectories in astrophysical environments, particularly near a Kerr black hole. This involved integrating numerical methods with NVIDIA CUDA technology to solve a system of ordinary differential equations (ODEs). This research was motivated by findings from previous studies on damping systems, suggesting that higher-order methods could potentially offer computational time performance comparable to RK4 in more complex astrophysical simulations.

11.1 Main results

- **Implementation and comparison of numerical methods**:
  Various Runge-Kutta methods (second, third, fourth, and eighth order) and the Adams-Bashforth second-order method were implemented and compared against each other in terms of accuracy and computation time.
- **Computational efficiency**:
  - Contrary to initial assumptions, higher-order methods did not improve computational efficiency for this specific astrophysical simulation.
  - RK2 demonstrated the quickest computation times among the Runge-Kutta methods.
  - Adams-Bashforth method, despite being a different approach, did not prove efficient in terms of computation time.
- **GPU performance**:
  - The use of CUDA technology enabled efficient parallel processing, highlighting the potential of GPUs in handling complex numerical simulations.
  - Despite the computational capabilities of GPUs, higher-order methods still showed increased computation times.

11.2 Solving the case

The thesis successfully addressed the problem of comparing numerical methods for simulation of photon trajectories near Kerr black holes. While the higher-order methods provided greater accuracy, they did not offer the expected computational efficiency. Instead, lower-order methods like RK2 and RK3 proved to be more suitable for large-scale simulations where computational speed is critical. The study clarified the computational trade-offs involved in using advanced numerical methods for simulating complex astrophysical phenomena, contributing valuable insights to the field.

11.3 Future work and applications

- **Optimization of numerical methods**:
  - Future research could explore hybrid methods combining the accuracy of higher-order methods with the efficiency of lower-order methods.
  - Adaptive step-size control could be implemented to enhance the accuracy and efficiency of numerical simulations.
- **Software development**:
  - Developing user-friendly software tools based on the findings of this study could help researchers in various fields perform efficient and accurate simulations without deep expertise in numerical methods or parallel programming.
  - Such tools could include modules for different numerical methods, allowing users to choose the most appropriate one for their specific applications.

In conclusion, this thesis has demonstrated the importance of choosing appropriate numerical methods for specific simulation scenarios and the potential benefits of leveraging parallel computing technologies like CUDA. While higher-order methods offer greater accuracy, their computational efficiency must be carefully evaluated in the context of the specific application, with lower-order methods often providing a more balanced solution for large-scale simulations.

REFERENCES

Butcher, J. C. 2016. Numerical methods for ordinary differential equations. Third Edition. John Wiley and Sons.

Gilat, A. & Subramaniam, V. 2000. Numerical methods for engineers and scientists. First Edition. John Wiley and Sons.

Hairer, E. & Wanner, G. 1993. Solving ordinary differential equations i: nonstiff problems. Second Revised Edition. Springer.

Repin, S. et al. 2018. Shadow of rotating black holes on a standard background screen. In: *Arxiv:1802.04667 [astro-ph.he]*.

Sanders, J. & Kandrot, E. 2010. Cuda by example: an introduction to general-purpose gpu programming. Addison-Wesley Professional.

LIST OF FIGURES