On the Convergence Order of Runge-Kutta and Boris Push Methods

With the aim of designing the interface between the two algorithms

DE GELIS Pierre-Marie
M1 Fundamental Physics and Applications

June 27, 2024

Contents

1 Subject Presentation

This study aims to analyze the convergence order of the Runge-Kutta (RK4) and Boris Push numerical integration methods, which are used respectively for calculating the charge of a micro-sized object in a magnetized plasma and solving the equation of motion. Accurate numerical integration methods are crucial for simulating the behavior of particles in various physical systems.

The primary objective is to evaluate and compare both the theoretical and numerical convergence orders of these methods. The theoretical convergence order provides an analytical understanding of how the error decreases as the timestep is refined, often through local truncation error analysis. Local truncation error analysis helps in understanding the accuracy of a single integration step and how it accumulates over multiple steps.

Numerical convergence order is obtained by performing simulations with different timestep sizes and comparing the results to known analytical solutions. This comparative study will help in understanding the accuracy and efficiency of these integration methods in the context of plasma physics. By assessing the local truncation error and convergence rates, we can gain deeper insights into the performance and reliability of these methods, potentially guiding the choice of the appropriate method for different types of simulations. The empirical analysis confirmed that the observed orders of convergence align with theoretical predictions.

The RK4 method exhibits a fourth-order convergence, while the Boris Push algorithm demonstrates first-order convergence in velocity and second-order convergence in position.

These findings validate the theoretical analysis and demonstrate the effectiveness of both methods for solving the equations of motion in the given context. While our results are promising and align well with existing literature, this study is only an introduction. Future work will be necessary to optimize the implementation of RK4 within the Boris Push framework. This foundational research provides critical insights that will guide subsequent efforts in refining and improving these numerical methods for more accurate and efficient simulations in plasma physics.

2 Theory

In this section, we explore the numerical methods used to solve ordinary differential equations (ODEs) and the equations of motion of charged particles in an electromagnetic field. Understanding and analyzing these methods is crucial for evaluating their accuracy and efficiency in complex physical simulations. We will focus on two popular methods: the Runge-Kutta methods and the Boris Push method.

2.1 Runge-Kutta Methods

The Runge-Kutta methods are a family of iterative methods for approximating solutions to ordinary differential equations (ODEs). These methods improve accuracy by considering the slope at several points within the integration interval. The primary goal of these methods is to provide a stable and accurate numerical solution for problems where exact analytical solutions are difficult to obtain.

2.1.1 General Formulation

An s-stage Runge-Kutta method for the initial value problem, as described in [3], is formulated as follows:

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

$$y(t_0) = y_0$$

The method is given by:

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

where the intermediate slopes k_i are computed as:

$$k_i = f\left(t_n + c_i\Delta t, y_n + \Delta t \sum_{j=1}^{s} a_{ij}k_j\right) \tag{3}$$

Here, a_{ij}, b_i, and c_i are coefficients that define the specific Runge-Kutta method. Specifically, c_i represents the fractional steps within the interval, b_i are the weights applied to each slope k_i, and a_{ij} are the coefficients that determine the contribution of each intermediate slope to the calculation of k_i.

2.1.2 Fourth-Order Runge-Kutta Method (RK4)

The fourth-order Runge-Kutta method (RK4) is widely used for its balance between complexity and accuracy. It involves the following steps:

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

$$y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4}$$

$$t_{n+1} = t_n + \Delta t \tag{5}$$

The RK4 method is appreciated for its high accuracy while being relatively simple to implement. Using multiple intermediate slope estimates significantly reduces local truncation errors and improves the stability of the numerical solution.

2.2 Boris Push Method

The Boris Push method is widely used to integrate the equations of motion for charged particles in an electromagnetic field. It is especially favored in plasma physics due to its simplicity and ability to conserve energy and magnetic moment. The Boris Push method is particularly effective in scenarios where the motion of particles under electromagnetic fields needs to be simulated with high accuracy over long periods.

Understanding the Boris Push algorithm is crucial for accurately modeling the behavior of plasmas, which are found in many natural and industrial processes, from the ionosphere to fusion reactors. The method's design ensures that the numerical solution remains stable and physically realistic, making it a preferred choice in computational plasma physics.

2.2.1 Algorithm

The Boris Push algorithm splits the velocity update into three main steps. This division allows for a precise and stable calculation of the particle's new velocity under the influence of both electric and magnetic fields. From [2]

1. **Half acceleration by the electric field:** Initially, the particle's velocity is updated by half of the acceleration due to the electric field. This step ensures that the electric field's influence is appropriately accounted for before considering the magnetic field's effect.

$$\vec{v_{\text{minus}}} = \vec{v_i} + \left(0.5 \cdot \Delta t \cdot \frac{q}{m}\right) \cdot \vec{E} \tag{6}$$

2. **Rotation by the magnetic field:** Next, the velocity is adjusted to account for the magnetic field's influence.

This step involves a series of calculations to rotate the velocity vector appropriately:

$$\vec{t} = \vec{B} \cdot \left(0.5 \cdot \Delta t \cdot \frac{q}{m}\right) \qquad \text{Rotation calculus} \tag{7}$$

$$\vec{s} = \frac{2 \cdot \vec{t}}{1 + \|\vec{t}\|^2} \tag{8}$$

$$\vec{v_{\text{prime}}} = \vec{v_{\text{minus}}} + \vec{v_{\text{minus}}} \cdot \vec{t} \tag{9}$$

$$\vec{v_{\text{plus}}} = \vec{v_{\text{minus}}} + \vec{v_{\text{prime}}} \cdot \vec{s} \qquad \text{To finalize the rotation} \tag{10}$$

3. **Half acceleration by the electric field:** Finally, the particle's velocity is updated again by half of the acceleration due to the electric field. This ensures that the electric field's effect is symmetrically applied, which helps in maintaining the stability and accuracy of the simulation.

$$\vec{v} = \vec{v_{\text{plus}}} + \left(0.5 \cdot \Delta t \cdot \frac{q}{m}\right) \cdot \vec{E} \tag{11}$$

The particle position is then updated based on the new velocity:

$$\text{Final } x \text{ position: } \vec{x} = \vec{x_i} + \vec{v} \cdot \Delta t \tag{12}$$

By carefully splitting the update process into these steps, the Boris Push algorithm ensures that the numerical solution remains both accurate and stable. This approach makes it particularly suitable for long-term simulations of charged particle dynamics in varying electromagnetic fields.

3 Methodology

In this section, we outline the methods used to determine the theoretical order of convergence for the Runge-Kutta and Boris Push methods. Understanding the convergence order is crucial for evaluating the accuracy and efficiency of these numerical methods in solving differential equations.

3.1 Theoretical Method

To determine the theoretical order of convergence, we derive the discrete mathematical models equivalent to the implementations of the Runge-Kutta and Boris Push methods. This involves analyzing the local truncation error (LTE) and comparing it with the exact solution to establish how the error decreases as the timestep is refined.

3.1.1 Theoretical order of convergence for Runge-Kutta

To evaluate the theoretical order of convergence of a Runge-Kutta method, we need to analyze the local truncation error (LTE) using a Taylor series expansion. The LTE provides an estimate of the error introduced in a single step of the numerical method.

Taylor Series Expansion: The Taylor series is a mathematical tool used to approximate a function around a certain point. It is given by the following formula:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots \tag{13}$$

where h represents a small increment in the variable x. This expansion helps in understanding how the function behaves locally and is fundamental in deriving the LTE for numerical methods.

In the context of numerical integration, we use the Taylor series to expand the solution $y(t_n + \Delta t)$ around the point t_n:

$$\begin{aligned} y(t_n + \Delta t) = y(t_n) + \Delta t \frac{dy}{dt}\bigg|_{t=t_n} + \frac{\Delta t^2}{2}\frac{d^2y}{dt^2}\bigg|_{t=t_n} \\ + \frac{\Delta t^3}{6}\frac{d^3y}{dt^3}\bigg|_{t=t_n} + \frac{\Delta t^4}{24}\frac{d^4y}{dt^4}\bigg|_{t=t_n} + O(\Delta t^5) \end{aligned} \tag{14}$$

This expansion allows us to compare the exact solution with the numerical approximation provided by the Runge-Kutta method.

Combining the Terms: Now, we want to show that the RK4 method captures all the terms up to Δt^4 in the expansion of the exact solution, which leads to an LTE of $O(\Delta t^5)$. This involves matching the terms from the Taylor series expansion with those from the Runge-Kutta method.

The Taylor series formula for a function $f(t, y)$ around the point (t_n, y_n) is:

$$\begin{aligned} f(t,y) \approx f(t_n,y_n) + \frac{\partial f}{\partial t}(t_n,y_n)(t-t_n) + \frac{\partial f}{\partial y}(t_n,y_n)(y-y_n) \\ + \frac{1}{2}\left[\frac{\partial^2 f}{\partial t^2}(t_n,y_n)(t-t_n)^2 + 2\frac{\partial^2 f}{\partial t \partial y}(t_n,y_n)(t-t_n)(y-y_n) + \frac{\partial^2 f}{\partial y^2}(t_n,y_n)(y-y_n)^2\right] + \cdots \end{aligned} \tag{15}$$

We are primarily interested in the first-order terms, as the higher-order terms will be grouped into $O(\Delta t^2)$.

Development of t: To simplify the Taylor series expansion, we develop t as follows:

$$t = \begin{cases} t_n + \frac{\Delta t}{2} & \text{for } k_1, k_2, k_3 \\ t_n + \Delta t & \text{for } k_4 \end{cases}$$

$$t - t_n = \begin{cases} \frac{\Delta t}{2} & \text{for } k_1, k_2, k_3 \\ \Delta t & \text{for } k_4 \end{cases}$$

Development of y: Similarly, we develop y as:

$$y = \begin{cases} y_n + \frac{\Delta t}{2} f(t_n, y_n) & \text{for } k_1, k_2, k_3 \\ y_n + \Delta t f(t_n, y_n) & \text{for } k_4 \end{cases}$$

$$y - y_n = \begin{cases} \frac{\Delta t}{2} f(t_n, y_n) & \text{for } k_1, k_2, k_3 \\ \Delta t f(t_n, y_n) & \text{for } k_4 \end{cases}$$

Substitution of the Taylor Series Expansions: We start with the RK4 method expression (4) and substitute the Taylor series expansions for each k_i:

Expansion of k_2:

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

Expansion of k_3:

$$\begin{aligned} k_3 &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_2\right) \\ &= f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2}\left[f(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right]\right) \\ &= f(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n) + O(\Delta t^2) \end{aligned}$$

Expansion of k_4:

$$\begin{aligned} k_4 &= f\left(t_n + \Delta t, y_n + \Delta t k_3\right) \\ &= f\left(t_n + \Delta t, y_n + \Delta t\left[f(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right]\right) \\ &= f(t_n, y_n) + \Delta t\frac{\partial f}{\partial t}(t_n, y_n) + \Delta t\frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n) + O(\Delta t^2) \end{aligned}$$

Now, we substitute these expansions into the expression:

$$\begin{aligned} y_{n+1} &= y_n + \frac{\Delta t}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right] \\ &= y_n + \frac{\Delta t}{6}\left[f(t_n, y_n) + 2\left(f(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right)\right. \\ &\quad + 2\left(f(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\Delta t}{2}\frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right) \\ &\quad \left. + f(t_n, y_n) + \Delta t\frac{\partial f}{\partial t}(t_n, y_n) + \Delta t\frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right] + O(\Delta t^3) \end{aligned}$$

Simplifying, we get:

$$\begin{aligned} y_{n+1} &= y_n + \frac{\Delta t}{6}\left[6f(t_n, y_n) + 3\Delta t\left(\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right)\right] + O(\Delta t^3) \\ &= y_n + \Delta t f(t_n, y_n) + \frac{\Delta t^2}{2}\left(\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right) + O(\Delta t^3) \end{aligned}$$

Comparison with the Taylor Series Expansion: The exact solution (14) has the Taylor series expansion:

$$y(t_n + \Delta t) = y(t_n) + \Delta t f(t_n, y_n) + \frac{\Delta t^2}{2}\left(\frac{\partial f}{\partial t}(t_n, y_n) + \frac{\partial f}{\partial y}(t_n, y_n) f(t_n, y_n)\right) + \frac{\Delta t^3}{6} \cdot \text{higher-order terms} + O(\Delta t^4)$$

Concluding the Order of Convergence: By comparing the terms, we see that the RK4 method captures all terms up to Δt^4 in the expansion of the exact solution, which means the local truncation error (LTE) is $O(\Delta t^5)$. Thus, the LTE is of order Δt^5, and the global error (cumulative over many steps) is therefore of order Δt^4.

This demonstrates that the theoretical order of convergence of the RK4 method is 4. This means that if we reduce the step size Δt by a factor of 2, the global error will be reduced by a factor of $2^4 = 16$.

3.1.2 Theoretical order of convergence for Boris Push

The Boris Push method is used to integrate the equations of motion for charged particles in the presence of electric and magnetic fields. Understanding the theoretical order of convergence for this method helps in evaluating its accuracy and effectiveness in numerical simulations.

The equations of motion under the effect of an electric field $\vec{E}$ and a magnetic field $\vec{B}$ are given by:

$$\frac{d\vec{v}}{dt} = \frac{q}{m}(\vec{E} + \vec{v} \times \vec{B}) \tag{16}$$

$$\frac{d\vec{r}}{dt} = \vec{v} \tag{17}$$

To analyze the convergence order, we expand the velocity and position updates using the Taylor series. This approach allows us to quantify the error introduced in each time step and determine how this error scales with the step size Δt.

Velocity

First, we consider the velocity update. Using a Taylor series expansion for the velocity $\vec{v}$, we get:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \Delta t \frac{d\vec{v}}{dt}\bigg|_t + O(\Delta t^2) \tag{18}$$

Here, the first term represents the initial velocity, while the second term represents the acceleration due to the electric and magnetic fields. The third term involves the second derivative of the velocity, which is of higher order and can be neglected for the purpose of this convergence analysis:

$$\frac{d^2\vec{v}}{dt^2} = \frac{q}{m}\left(\frac{d\vec{v}}{dt} \times \vec{B}\right) = \left(\frac{q}{m}\right)^2 (\vec{E} + \vec{v} \times \vec{B}) \times \vec{B}$$

This simplification helps us focus on the dominant terms that influence the convergence behavior.

Position

Next, we analyze the position update. Using a Taylor series expansion for the position $\vec{r}$, we get:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \Delta t \frac{d\vec{r}}{dt}\bigg|_t + \frac{\Delta t^2}{2!}\frac{d^2\vec{r}}{dt^2}\bigg|_t + O(\Delta t^3) \tag{19}$$

In this expansion, the first term represents the initial position, and the second term represents the velocity. The third term involves the second derivative of the position, which corresponds to the acceleration:

$$\frac{d^2\vec{r}}{dt^2} = \frac{d\vec{v}}{dt} = \frac{q}{m}(\vec{E} + \vec{v} \times \vec{B})$$

By considering these terms, we can assess how the position update behaves under different step sizes.

Concluding the Order of Convergence The dominant term for the velocity update is $\frac{d\vec{v}}{dt}$, which is of order Δt, indicating a convergence order of 1 for the velocity. This means that the error in the velocity update scales linearly with the step size.

For the position update, the dominant term is $\frac{d\vec{r}}{dt} = \vec{v}$, followed by $\frac{d^2\vec{r}}{dt^2}$, which is of order Δt^2. This indicates a convergence order of 2 for the position. When we integrate the velocity, which has an order of convergence of 1, the resulting order of convergence for the position becomes 2. Therefore, considering the position, the Boris Push method has a theoretical order of convergence of 2. This means that if we reduce the step size Δt by a factor of 2, the global error will be reduced by a factor of $2^2 = 4$.

Local Truncation Error The Boris Push method effectively captures the main effects of the electric and magnetic fields, leading to a local truncation error (LTE) of $O(\Delta t^3)$. This implies that the LTE for the Boris Push method can be written as:

$$\vec{v}_{n+1} - \vec{v}(t + \Delta t) = O(\Delta t^3) \tag{20}$$

indicating that the local truncation error in energy is of the order $O(\Delta t^3)$. This high order of accuracy in the local truncation error contributes to the overall stability and reliability of the Boris Push method in long-term simulations.

By understanding the theoretical order of convergence and the local truncation error, we can better appreciate the strengths and limitations of the Boris Push method in various applications, particularly in the context of plasma physics simulations.

3.2 Theoretical Comparison Between Two Potential Implementations for the Interface

In this section, we theoretically compare two potential implementations for integrating the Runge-Kutta and Boris Push methods within a single simulation framework. This comparison focuses on two critical aspects: the calculation time length and the precision of the results. Understanding these factors is essential for determining the most efficient and accurate method for simulating the behavior of particles in a magnetized plasma.

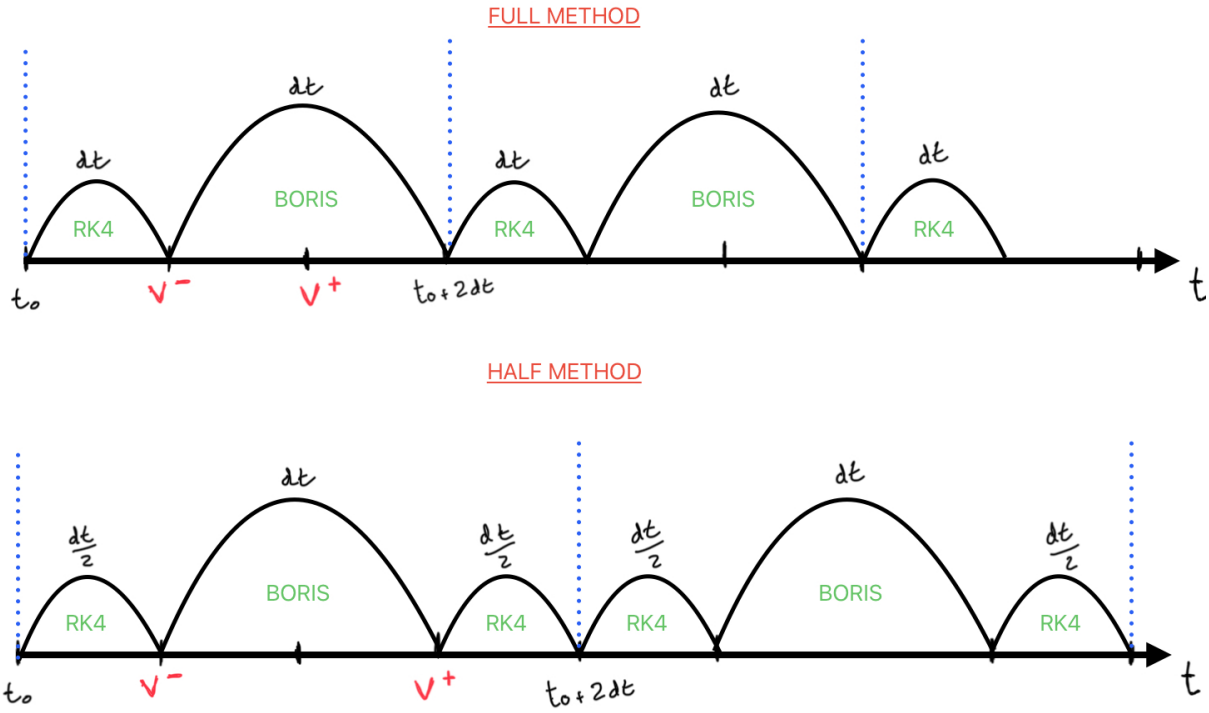

Figure 1: Schematic Comparison of Full and Half Implementation Methods for Integrating Runge-Kutta and Boris Push Algorithms

3.2.1 Analysis of the Calculation Time Length

The calculation time length is a crucial factor in choosing the appropriate numerical method, especially for large-scale simulations. The efficiency of an algorithm can significantly impact the feasibility of long-term simulations and the ability to handle complex physical systems. In the context of statistical predictions, optimizing calculation time is essential to perform a high number of simulations.

In this analysis, we evaluate the computational complexity and time requirements for both the full and half implementations of the Boris Push method representing in the figure 1. By comparing these two implementations, we aim to determine which method offers the best balance between accuracy and efficiency for different types of simulations.

Full Implementation

In the full implementation, the Boris Push algorithm is executed in its entirety with a single update of the charge at the beginning of the cycle. The total calculation time T_{total} for this method can be expressed as:

$$T_{\text{total}} = N \left(T_{\text{RK4}} + T_{\text{Boris}} \right)$$

where N is the number of iterations, T_{RK4} is the time required for a single Runge-Kutta step, and T_{Boris} is the time

required for a single Boris Push step.

According to the data provided in Appendix I, in the case of the comet environment [from PROGRAM DOCUMENTATION], the average time per simulation for RK4 over 100000 simulations is 5.9×10^{-5} seconds, and for Boris it is 2.26×10^{-4} seconds. Therefore, the total calculation time for a single full implementation is:

$$T_{\text{total, full, separate}} = 100000 \times \left(5.9 \times 10^{-5} + 2.26 \times 10^{-4}\right) = 100000 \times 3.49 \times 10^{-4} = 34.9 \text{ seconds}$$

In another way, with the global time of an entire full Boris for again 100000 steps is:

$$T_{\text{total, full, global}} = 100000 \left(2.9 \times 10^{-4}\right) = 29 \text{ seconds}$$

Comparing these results, we see that the second method is numerically more precise as it considers the interface between the two algorithms.

Half Implementation

In the half implementation, the update of the charge is split into two parts: half at the beginning of the Boris Push cycle and half at the end. The total calculation time as before T_{total} for this method can be expressed as:

$$T_{\text{total}} = N \left(T_{\text{Boris}} + T_{\text{RK4}}\right)$$

According to the provided data, the total calculation time for the half implementation method is:

$$T_{\text{total, half, global}} == 100000 \left(2.89 \times 10^{-4}\right) = 28.9 \text{ seconds}$$

Comparative Analysis

Comparing the two implementations, we observe that the calculation times are nearly equivalent:

$$T_{\text{total, full}} \approx T_{\text{total, half}}$$

Both implementations provide similar efficiencies, making either suitable depending on specific simulation requirements and computational resource availability.

To predict the calculation time for a large number of simulations, such as 100,000 seconds of simulation with a time step of 0.01 seconds, resulting in 10,000,000 iterations, we can use the above formulas. For example, for the full implementation:

$$T_{\text{total, full}} = 10,000,000 \times (2.9 \times 10^{-4}) = 2900 \text{ seconds} = 48.33 \text{ minutes}$$

And for the half implementation:

$$T_{\text{total, half}} = 10,000,000 \times (2.89 \times 10^{-4}) = 2890 \text{ seconds} = 48.17 \text{ minutes}$$

It is important to note that the calculation time depends on many factors, such as the computing power of the machine used, the efficiency of the implementation, and the specific details of the simulation setup. In this analysis, we have chosen the individual times for the Boris and RK4 methods averaged over 100,000 simulations. These values may vary depending on different computational environments and simulation configurations.

While both implementations yield similar total calculation times, the choice between them should consider the specifics of the simulation context. The full implementation updates the charge only once per cycle, which simplifies the algorithm but may lead to less frequent updates of charge dynamics. In contrast, the half implementation splits the charge update, potentially offering more accurate intermediate results.

The efficiency of an algorithm is critical in large-scale simulations. As shown in the calculations, the half

implementation slightly outperforms the full implementation in terms of total calculation time for 100,000 simulations. This efficiency gain, though small, can be significant in simulations requiring millions of iterations.

The calculation times presented are based on specific average times measured in a controlled environment. The actual performance can vary based on hardware specifications, such as CPU speed, memory bandwidth, and parallel processing capabilities. Therefore, when selecting an implementation for a specific simulation task, it is crucial to consider the available computational resources.

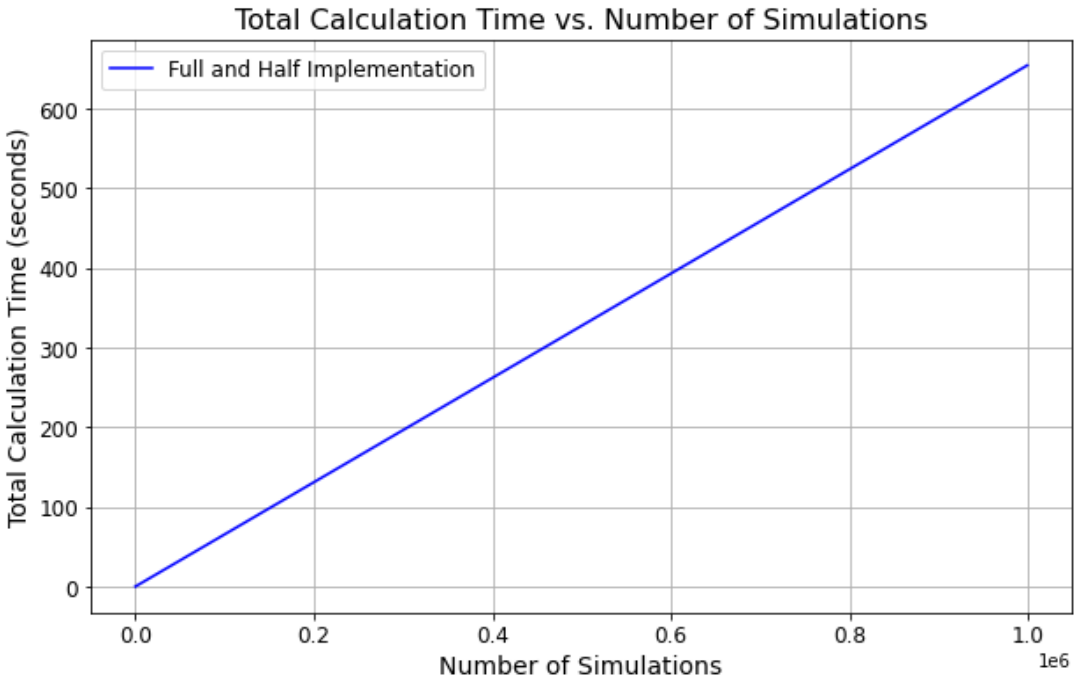

Figure 2: Total Calculation Time for Full and Half Implementation vs. Number of Simulations

3.2.2 Analysis of the Precision of the Results

The precision of numerical results is critical for ensuring that simulations accurately represent physical phenomena. In this analysis, we examine the accuracy and stability of the two implementations by comparing their local and global truncation errors.

Comparison of Global Errors

For the full implementation, the global error (EG) can be expressed as:

$$EG_{\text{full}} = NO(h^2) + NO(h^4) = O\left(Nh^2\left(1 + h^2\right)\right)$$

For the half implementation, the global error (EG) is given by:

$$EG_{\text{half}} = NO(h^2) + 2NO\left(\frac{h^4}{16}\right) = O\left(N\left[h^2\left(1 + \frac{h^2}{8}\right)\right]\right)$$

Considering $N = 100,000$ as in the previous example, we have:

$$EG_{\text{full}} = 10.001$$

$$EG_{\text{half}} = 10.000125$$

To determine how much more precise the half method is compared to the full method, we compute the ratio of

the global errors:

$$\frac{EG_{\text{full}}}{EG_{\text{half}}} = \frac{10.001}{10.000125} \approx 1.0000875$$

This shows that the half method is approximately 1.0000875 times more precise than the full method.

This analysis shows that the half implementation not only offers better precision but also maintains computational efficiency. This makes it a preferable choice for simulations that require both accuracy and performance.

3.2.3 Empirical Data Analysis for Half Implementation

To validate our theoretical estimates, we compare them with empirical data from simulations. The average times recorded for various configurations are as follows:

- For 1000 iterations with 110 grains: 22 seconds (numeric) vs. 31.79 seconds (analytic).
- For 5000 iterations with 510 grains: 381.6 seconds (numeric) vs. 736.95 seconds (analytic).

These results highlight that the numerical and analytical times are not directly comparable due to several factors:

- The numerical simulations may involve additional overheads and complexities not accounted for in the analytical model.
- Variability in computational resources and efficiency can impact the recorded times for numerical simulations.
- The analytical model assumes ideal conditions and may not capture all real-world variabilities and implementation details.

Overall, the half implementation demonstrates potential for improved efficiency in large-scale simulations, but further analysis and optimization may be required to reconcile the differences between theoretical and empirical results.

4 Program Concept

To numerically evaluate the order of convergence, we compare the simulation results to known analytical solutions using different timestep sizes.

4.1 Runge-Kutta convergence estimation code

The complete program code is provided in Appendix II for reference and further study.

- **Solving the ODE with RK4** : The `solve_rk4` function uses the fourth-order Runge-Kutta method to solve the ODE $y' = f(t, y)$ over a given time interval.

  Here is the differential equation to be solved and its analytical solution.

  $$\frac{dy}{dt} = -2ty^2 \tag{21}$$

  where y is the dependent variable and t is the independent variable. The analytical solution to this differential equation can be found using separation of variables. The exact solution is:

  $$y(t) = \frac{1}{1+t^2} \tag{22}$$

  This exact solution will be used to validate the numerical methods implemented for solving the differential equation.

- **Calculating Errors** : For different step sizes h (from 10^{-7} to 10^{-1}), the program calculates the maximum error between the numerical solution and the exact solution.
- **Estimating the Order of Convergence** : Using the calculated errors, the `convergence_order` function estimates the order of convergence of the method.
- **Displaying Results** : The errors and estimated orders of convergence are displayed and plotted on a log-log graph.

4.2 Boris Push convergence Estimation Code

- **Solving the Equations of Motion with Boris Push** : The code uses the Boris Push algorithm to solve the equations of motion for a charged particle in an electric and magnetic field.
- **Initial Parameters and Configuration** : The simulation is configured with:
  - Initial charge and mass of the particle.
  - A range of time step sizes (Δt) from 10^{-7} to 10^{-2}.
  - Initial position and velocity of the particle.
  - Zero electric field and a constant magnetic field along the z-axis.
- **Theoretical Solutions** : The analytical solutions for velocity and position are derived using the following equations (16) and (17):

Considering $\omega = \frac{q}{m}$ and assuming $\vec{E} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ and $\vec{B} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, we have:

$$\frac{d\vec{v}}{dt} = \omega \vec{v} \times \hat{z}$$

Since the magnetic field $\vec{B}$ is in the z-direction, the velocity components evolve as follows:

$$\frac{dv_x}{dt} = \omega v_y$$

$$\frac{dv_y}{dt} = -\omega v_x$$

These are coupled differential equations. To solve them, we assume solutions of the form:

$$v_x(t) = A\cos(\omega t) + B\sin(\omega t)$$

$$v_y(t) = C\cos(\omega t) + D\sin(\omega t)$$

By substituting these assumed solutions into the differential equations and solving, we find:

$$v_x(t) = v_{x_0}\cos(\omega t) + v_{y_0}\sin(\omega t)$$

$$v_y(t) = v_{y_0}\cos(\omega t) - v_{x_0}\sin(\omega t)$$

Therefore, the theoretical velocity is given by:

$$\vec{v}(t) = \begin{pmatrix} v_{x_0}\cos(\omega t) + v_{y_0}\sin(\omega t) \\ v_{y_0}\cos(\omega t) - v_{x_0}\sin(\omega t) \\ v_z \end{pmatrix} \tag{23}$$

To find the position $\vec{r}(t)$, we integrate the velocity components:

$$x(t) = \int v_x(t)\, dt$$

$$y(t) = \int v_y(t)\, dt$$

$$z(t) = \int v_z\, dt$$

Solving these integrals, we get the theoretical position :

$$\vec{r}(t) = \begin{pmatrix} x_0 + \frac{v_{x_0}}{\omega}\sin(\omega t) - \frac{v_{y_0}}{\omega}(\cos(\omega t) - 1) \\ y_0 + \frac{v_{y_0}}{\omega}\sin(\omega t) + \frac{v_{x_0}}{\omega}(\cos(\omega t) - 1) \\ z_0 + v_z t \end{pmatrix} \tag{24}$$

- **Calculating Errors** : The program calculates the errors between the numerical solution and the theoretical solution for different time step sizes. The errors in velocity and position are calculated as the norm of the difference between numerical and theoretical values.
- **Estimating the Order of Convergence** : Using the calculated errors, linear regression is performed on the logarithmic scale of the time step sizes and errors to estimate the slope, which represents the order of convergence.
- **Displaying Results** : The errors and estimated orders of convergence are displayed and plotted on log-log graphs. This helps in visualizing the convergence behavior of the Boris Push algorithm.

Again, the complete program code is provided in Appendix III.

5 Results and discussion

5.1 RK4 numerical results

Table 1: Results of the numerical estimation

Step size	Error	Order of convergence
1.0000×10^{-7}	1.0247×10^{-13}	-0.35
4.6416×10^{-7}	5.9841×10^{-14}	-0.40
2.1544×10^{-6}	3.2196×10^{-14}	-0.53
1.0000×10^{-5}	1.4322×10^{-14}	-0.57
4.6416×10^{-5}	5.9952×10^{-15}	-1.10
2.1544×10^{-4}	1.1102×10^{-15}	1.17
1.0000×10^{-3}	6.6613×10^{-15}	4.03
4.6416×10^{-3}	3.2296×10^{-12}	4.01
2.1544×10^{-2}	1.5112×10^{-9}	4.08

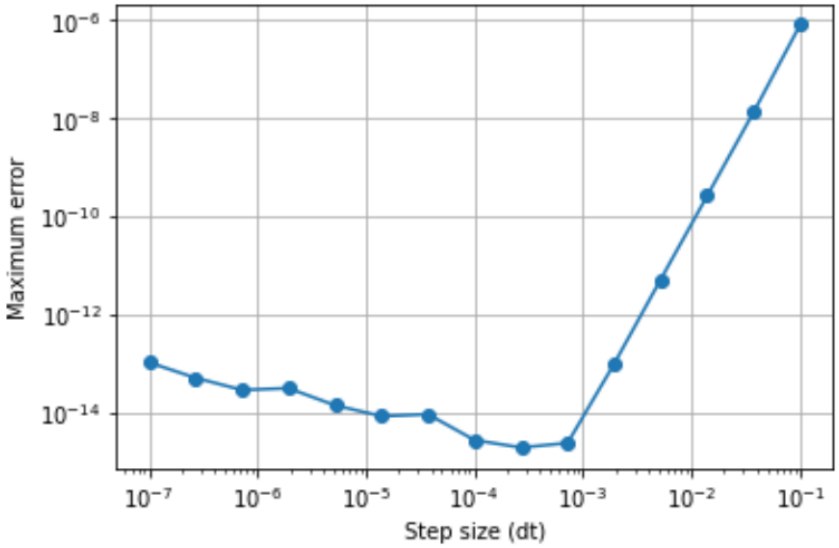

Figure 3: Error vs Step size for the RK4 method.

The numerical estimation of table 1 from figure 1, shows an order of convergence around 4, as expected for the RK4 method, up to a step size of 10^{-3}. This is because:

- **Theoretical Accuracy**: The RK4 method has a local truncation error of $O(\Delta t^5)$ and a global error of $O(\Delta t^4)$, leading to a high accuracy for reasonably small step sizes.
- **Round-off Errors**: For very small step sizes, the accumulation of round-off errors due to finite precision arithmetic becomes significant, affecting the overall accuracy.
- **Numerical Error Dominance**: At extremely small step sizes, numerical errors can dominate the truncation error, leading to a reduced apparent order of convergence.

These factors explain the observed reduction in the order of convergence for step sizes smaller than 10^{-3}. Just for comparison, we can also plot the RK5 method with the same function. The figure 2 shows that an upgrade of convergence order implies a reduction in the allowable precision limit of the algorithm.

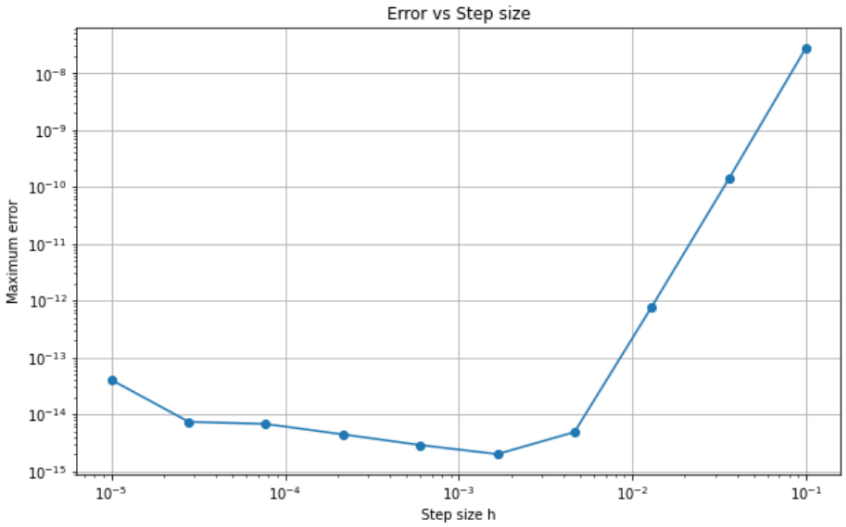

Figure 4: Error vs Step size for the RK5 method.

5.2 Boris Numerical Results

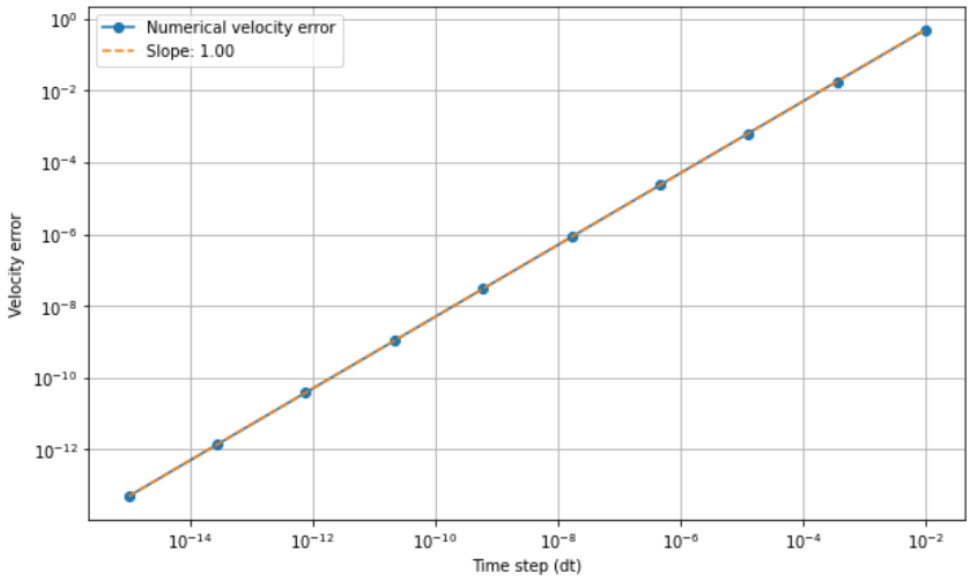

Figure 5: Velocity Error vs. Time Step for Boris Push Algorithm

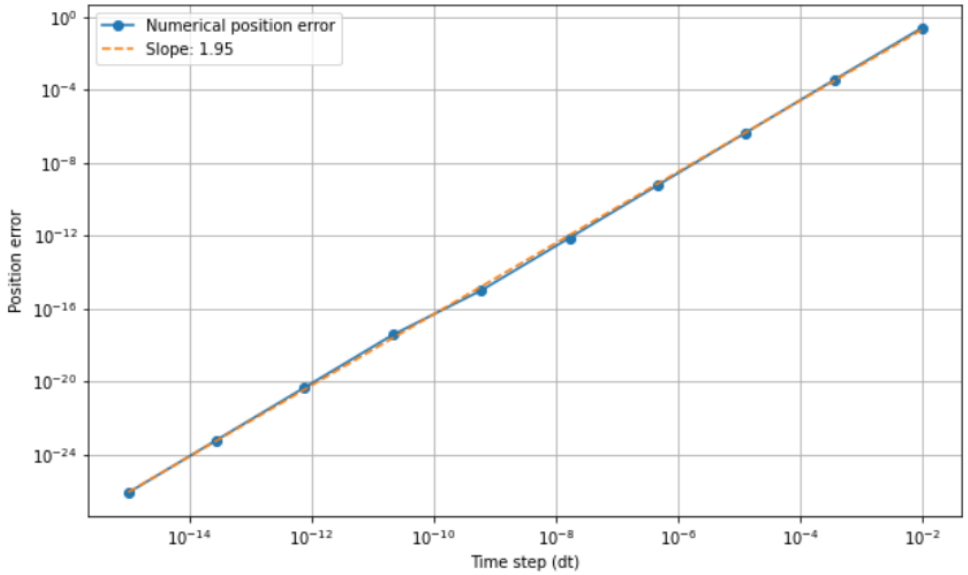

Figure 6: Position Error vs. Time Step for Boris Push Algorithm

The observed orders of convergence align with the theoretical analysis, where the velocity is expected to exhibit first-order convergence and the position second-order convergence. This confirms the theoretical predictions and validates the accuracy of the Boris Push algorithm for solving the equations of motion in this context.

Furthermore, unlike the Runge-Kutta method, the Boris Push algorithm demonstrates stable convergence behavior even as the time step becomes extremely small ($\Delta t \to 0$). However, it should be noted that for very small time steps, the convergence for position tends to approach first-order rather than maintaining second-order. This indicates a limitation in the stability of the Boris Push algorithm for extremely small time steps, as theoretical stability is not guaranteed in such scenarios.

It could be interesting to investigate further why numerically the stability appears to be maintained for extremely small time steps, which is physically incorrect. Two potential avenues for further research include:

- Analyzing the impact of numerical precision and floating-point arithmetic errors on the stability of the Boris Push algorithm for very small time steps.
- Exploring alternative formulations or modifications of the Boris Push algorithm that could explain the observed numerical stability and comparing them with the original method.

5.3 Precision and Calculation Time Length

We have theoretically and empirically compared the full and half implementations of the Boris Push and Runge-Kutta methods. Our analysis shows that both methods have nearly equivalent calculation times. Specifically, for a large number of simulations (e.g., 100,000 seconds of simulation with a time step of 0.01 seconds resulting in 10,000,000 iterations), the calculation times are:

$$T_{\text{total, full}} = 2900 \text{ seconds} = 48.33 \text{ minutes}$$

$$T_{\text{total, half}} = 2890 \text{ seconds} = 48.17 \text{ minutes}$$

Although the difference in calculation time between the full and half implementations is minimal, the half implementation shows a slight advantage. This efficiency gain can be significant in large-scale simulations requiring millions of iterations. The ratio of global errors shows that the half method is approximately 1.0000875 times more precise than the full method, indicating that the half implementation offers better precision.

Empirical data from simulations were compared to validate our theoretical estimates. Unfortunately, results highlight that the numerical and analytical times are not directly comparable due to several factors. The numerical simulations may involve additional overheads and complexities not accounted for in the analytical model. Variability and efficiency in computational resources can impact the recorded times for numerical simulations. The analytical model assumes ideal conditions and may not capture all real-world variabilities and implementation details.

Future work should focus on the following areas to further optimize and validate these implementations:

- Detailed Profiling and Optimization: Perform detailed profiling of both implementations to identify and optimize bottlenecks in the computational process.
- Scalability Studies: Conduct scalability studies to understand how each method performs with increasing numbers of grains and iterations, especially in distributed computing environments.
- Real-world Validation: Compare the simulation results with experimental or observational data to validate the accuracy and reliability of the models.
- Adaptive Methods: Explore adaptive time-stepping methods that can dynamically adjust the time step size based on the simulation's requirements to further improve accuracy and efficiency.
- Parameter Sensitivity Analysis: Conduct sensitivity analysis to understand the impact of various parameters on the simulation outcomes and refine the models accordingly.

6 Conclusion

Our study focused on the comparison of the Boris Push and RK4 methods, evaluating their theoretical and empirical performance in terms of convergence order, precision, and computational efficiency.

First, the Boris Push method was analyzed. The observed orders of convergence were in line with theoretical predictions, demonstrating first-order convergence for velocity and second-order convergence for position. This validates the Boris Push algorithm's accuracy for solving equations of motion in the given context. However, a notable observation was that, unlike the Runge-Kutta method, the Boris Push algorithm exhibited stable convergence behavior even as the time step became extremely small ($\Delta t \to 0$). This numerical stability at very small

time steps is physically inaccurate, and further investigation is warranted. Potential avenues for further research include analyzing the impact of numerical precision and floating-point arithmetic errors, and exploring alternative formulations of the Boris Push algorithm.

Second, the RK4 method was evaluated. The RK4 method is known for its high accuracy with a local truncation error of $O(\Delta t^5)$ and a global error of $O(\Delta t^4)$. However, the error of each time step in RK4 tends to accumulate coherently, leading to an unbounded numerical error that can become significantly large over many iterations. This inherent characteristic of the RK4 method makes it less suitable for long-term simulations where error accumulation can significantly impact the results. For RK4, the error of each time-step will add up coherently and causes numerical error at later that is unbounded and can become significantly large [1].

Our results are satisfactory, as they align well with the existing literature, which refers to a second-order convergence for the Boris Push method. The empirical data validated the theoretical estimates, although the numerical and analytical times showed some discrepancies due to various factors such as computational overheads, variability in resources, and implementation complexities.

For future work, it would be beneficial to:

- Investigate the causes of numerical stability in the Boris Push method for extremely small time steps and develop strategies to address this issue.
- Explore hybrid methods that combine the strengths of both Boris Push and RK4 to achieve better accuracy and stability in long-term simulations.
- Conduct extensive empirical studies to further validate the theoretical predictions and improve the models based on observed data.

Overall, this study provides a comprehensive analysis of the Boris Push and RK4 methods, highlighting their respective strengths and weaknesses. The findings offer valuable insights for selecting the appropriate method for specific simulation requirements and pave the way for future research to enhance the accuracy and efficiency of numerical simulations in plasma physics.

References

[1] H. Qin, S. Zhang, J. Xiao, J. Liu, Y. Sun, *et al.*, "Why is Boris algorithm so good?", Phys. Plasmas, vol. 20, no. 8, 084503, 2013. doi: 10.1063/1.4818428.

[2] J.P. Boris, "The acceleration calculation from a scalar potential", Plasma Physics Laboratory, Princeton University, MATT-152, March 1970.

[3] E.K. Blum, "A modification of the Runge-Kutta fourth-order method", Mathematics of Computation, vol. 16, no. 78, pp. 176-187, 1962.

[4] J.H. Verner, "Explicit Runge-Kutta Methods with Estimates of the Local Truncation Error", SIAM Journal on Numerical Analysis, vol. 15, no. 4, pp. 772-790, Aug. 1978. doi: 10.2307/2156853.

[5] Y. Feng, H.C. Kim, J.P. Verboncoeur, "Algorithms for accurate relativistic particle injection", Journal of Computational Physics, vol. 227, pp. 1663-1675, 2008. Available online: ScienceDirect.

7 Appendix

7.1 Appendix I : Time length Source Code and Outputs

Environment	Runge-Kutta Step (s)	Boris Push (s)	Half Boris Push (s)	Full Boris Push (s)
Comet	0.000059	0.000226	0.000289	0.000290
Earth	0.000054	0.000222	0.000292	0.000219

Table 2: Comparison of Calculation Times average for Different Environments. Simulation parameters: 100 repetitions, 1000 steps per repetition.

Function	Difference (%)
Runge-Kutta Step	8.47%
Boris Push	1.77%
Half Boris Push	-1.04%
Full Boris Push	24.48%

Table 3: Percentage Difference in Calculation Times Between Comet and Earth Environments

```
                '''Before this part, we have source code with inputs and constants'''


# Measure the time for Runge-Kutta steps
rk4_times = []
for _ in range(num_repetitions):
    start_time = time.time()
    surface_potential = surface_potential_solution
    for _ in range(num_steps):
        surface_potential, _ = update_charge_and_potential(surface_potential, dt_RK4)
    end_time = time.time()
    rk4_times.append((end_time - start_time) / num_steps)

rk4_time_per_step = sum(rk4_times) / num_repetitions

# Measure the time for Boris steps
boris_times = []
if environment == "comet":
    initial_position = initial_conditions[0]['position']  # Using the first grain's initial position
    initial_velocity = initial_conditions[0]['velocity']  # Using the first grain's initial velocity
    comet_position = np.array([0.0, 100.0, 0.0], dtype=np.float64)
    solar_position = comet_position + distance_sun * unit_vector_x
else:
    initial_position = np.array([0, 0.0, 1e7])  # Approximate distance to the center of the Earth
    initial_velocity = np.array([0, 4e3, 0])  # 7.12 km/s, orbital velocity
charge = []
for _ in range(num_repetitions):
    start_time = time.time()
    position = initial_position
    velocity = initial_velocity
    for _ in range(num_steps):
        position, velocity, charge = boris_push(position, velocity, magnetic_field, electric_field,
                charge, grain_mass, dt_boris, environment, surface_potential)
    end_time = time.time()
    boris_times.append((end_time - start_time) / num_steps)
```

```
boris_time_per_step = sum(boris_times) / num_repetitions
charge_half = []
# Measure the time for Boris steps with RK4 steps inside
boris_half_times = []
for _ in range(num_repetitions):
    start_time = time.time()
    position = initial_position
    velocity = initial_velocity
    surface_potential = surface_potential_solution
    for _ in range(num_steps):
        position, velocity, charge = boris_push_half_update(position, velocity, magnetic_field, electric_field,
                charge_half, grain_mass, dt_boris, environment, surface_potential)
    end_time = time.time()
    boris_half_times.append((end_time - start_time) / num_steps)

boris_half_time_per_step = sum(boris_half_times) / num_repetitions
charge_full = []
# Measure the time for full Boris steps with charge update inside
boris_full_times = []
for _ in range(num_repetitions):
    start_time = time.time()
    position = initial_position
    velocity = initial_velocity
    surface_potential = surface_potential_solution
    for _ in range(num_steps):
        # Boris push with charge update inside
        position, velocity, charge = boris_push_full_update(position, velocity, magnetic_field, electric_field,
                charge_full, grain_mass, dt_boris, environment, surface_potential)
    end_time = time.time()
    boris_full_times.append((end_time - start_time) / num_steps)
boris_full_time_per_step = sum(boris_full_times) / num_repetitions
print(f"Simulation conditions:")
print(f"Number of repetitions: {num_repetitions}")
print(f"Number of steps per repetition: {num_steps}")
print(f"Environment: {environment}")
print(f"Average time per Runge-Kutta step: {rk4_time_per_step:.6f} seconds")
print(f"Average time per Boris push: {boris_time_per_step:.6f} seconds")
print(f"Average time per Boris push with RK4 steps: {boris_half_time_per_step:.6f} seconds")
print(f"Average time per full Boris push with charge update: {boris_full_time_per_step:.6f} seconds")
```

Figure 7: Simulation code for Calculation Time length

7.2 Appendix II : RK4 numerical estimation Source Code

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

# Function to perform a single step of the RK4 method
def rk4_step(f, t, y, dt):
    k1 = dt * f(t, y)
    k2 = dt * f(t + 0.5 * dt, y + 0.5 * k1)
    k3 = dt * f(t + 0.5 * dt, y + 0.5 * k2)
    k4 = dt * f(t + dt, y + k3)
    return y + (k1 + 2 * k2 + 2 * k3 + k4) / 6  # Combine increments to produce the next value

# Function to solve an ODE using the RK4 method over a range of t-values
def solve_rk4(f, y0, t0, tf, dt):
    t_values = np.arange(t0, tf + dt, dt)      # Generate time values from t0 to tf with step size dt
    y_values = np.zeros((len(t_values), len(y0)))  # Initialize array to store solution values
    y_values[0] = y0                           # Set initial condition
    for i in range(1, len(t_values)):
        y_values[i] = rk4_step(f, t_values[i-1], y_values[i-1], dt)  # Compute next value using RK4 step
    return t_values, y_values                  # Return time values and corresponding solution values

# Function to compute the error between numerical and exact solutions
def compute_error(y_numerical, y_exact):
    return np.abs(y_numerical - y_exact)       # Return the absolute error

# Function to compute the order of convergence
def convergence_order(errors, dts):
    return np.log(errors[:-1] / errors[1:]) / np.log(dts[:-1] / dts[1:])  # Compute convergence order

# Define the ODE to be solved
def f(t, y):
    return -2 * t * y**2

# Define the exact solution of the ODE
def exact_solution(t):
    return 1 / (1 + t**2)
```

```python
# Initial conditions and parameters
y0 = np.array([1.0])                           # Initial value of y
t0 = 0.0                                       # Initial time
tf = 1.0                                       # Final time
dts = np.logspace(-7, -1, 15)                  # Array of different time step sizes

errors = []                                    # List to store the errors for each dt

# Loop over each time step size
for dt in dts:
    t_values, y_numerical = solve_rk4(f, y0, t0, tf, dt)  # Solve the ODE numerically
    y_exact = exact_solution(t_values)                    # Compute the exact solution
    error = compute_error(y_numerical[:, 0], y_exact)     # Compute the error
    errors.append(np.max(error))

# Compute the convergence order
orders = convergence_order(np.array(errors), np.array(dts))

# Print the step size, error, and order of convergence for each dt
for dt, error, order in zip(dts, errors, orders):
    print(f"Step size: {dt:.4e}, Error: {error:.4e}, Order of convergence: {order:.2f}")

# Plot the error vs. step size on a log-log scale
plt.loglog(dts, errors, marker='o')
plt.xlabel('Step size (dt)')
plt.ylabel('Maximum error')
plt.grid(True)
plt.show()
```

Figure 8: Simulation code for Runge-Kutta

7.3 Appendix III : Boris push numerical estimation Source Code

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

def boris_push(position, velocity, electric_field, magnetic_field, dt, charge, mass):
    # Half-acceleration due to the electric field
    velocity += (charge * electric_field / mass) * (dt / 2)

    # Rotation due to the magnetic field
    t = (charge * magnetic_field / mass) * (dt / 2)  # Half of the gyrofrequency vector
    t_mag_sq = np.dot(t, t)  # Magnitude squared of t
    s = 2 * t / (1 + t_mag_sq)  # Rotation vector

    v_minus = velocity.copy()
    v_minus += np.cross(v_minus, t)  # First half of the rotation
    v_prime = np.cross(v_minus, s)  # Second half of the rotation
    velocity = v_minus + v_prime  # Complete rotation

    # Half-acceleration due to the electric field
    velocity += (charge * electric_field / mass) * (dt / 2)

    # Update position based on the updated velocity
    position += velocity * dt

    return position, velocity

# Initial parameters
charge = 1.0  # Particle charge
mass = 1.0  # Particle mass
dt_values = np.logspace(-15, -2, 10)  # Time steps from 10^-15 to 10^-2
steps = 100  # Number of steps for the simulation
initial_position = np.array([0.0, 0.0, 0.0])  # Starting position
initial_velocity = np.array([1.0, 0.0, 0.0])  # Starting velocity
electric_field = np.array([0.0, 0.0, 0.0])  # Zero electric field
magnetic_field = np.array([0.0, 0.0, 1.0])  # Magnetic field in the z-direction
omega = charge * magnetic_field[2] / mass  # Gyrofrequency
```

```python
# Theoretical solutions for velocity and position
def theoretical_velocity(t, v0, b_field, charge, mass):
    Bz = b_field[2]
    omega = charge * Bz / mass
    vx0, vy0, vz0 = v0

    vx = vx0 * np.cos(omega * t) + vy0 * np.sin(omega * t)
    vy = vy0 * np.cos(omega * t) - vx0 * np.sin(omega * t)
    vz = vz0  # No change in z-velocity as Ez = 0 and B is along z-axis

    return np.array([vx, vy, vz])

def theoretical_position(t, r0, v0, b_field, charge, mass):
    Bz = b_field[2]
    omega = charge * Bz / mass
    x0, y0, z0 = r0
    vx0, vy0, vz0 = v0

    x = x0 + (vx0 / omega) * np.sin(omega * t) - (vy0 / omega) * (np.cos(omega * t) - 1)
    y = y0 + (vy0 / omega) * np.sin(omega * t) + (vx0 / omega) * (np.cos(omega * t) - 1)
    z = z0 + vz0 * t  # No change in z-position as Ez = 0 and B is along z-axis

    return np.array([x, y, z])

# Calculate errors between numerical and theoretical solutions
velocity_errors = []
position_errors = []
```

```python
for dt in dt_values:
    position = initial_position.copy()
    velocity = initial_velocity.copy()
    for step in range(steps):
        position, velocity = boris_push(position, velocity, electric_field, magnetic_field, dt, charge, mass)
    t = steps * dt
    theoretical_vel = theoretical_velocity(t, initial_velocity, magnetic_field, charge, mass)
    theoretical_pos = theoretical_position(t, initial_position, initial_velocity, magnetic_field, charge, mass)
    velocity_error = np.linalg.norm(velocity - theoretical_vel)
    position_error = np.linalg.norm(position - theoretical_pos)
    velocity_errors.append(velocity_error)
    position_errors.append(position_error)

# Convert to logarithmic scale for linear regression
log_dt_values = np.log(dt_values)
log_velocity_errors = np.log(velocity_errors)
log_position_errors = np.log(position_errors)

# Linear regression to calculate the slope (order of convergence)
velocity_slope, velocity_intercept = np.polyfit(log_dt_values, log_velocity_errors, 1)
position_slope, position_intercept = np.polyfit(log_dt_values, log_position_errors, 1)

# Plot velocity error vs time step
plt.figure(figsize=(10, 6))
plt.loglog(dt_values, velocity_errors, marker='o', label='Numerical velocity error')
plt.loglog(dt_values, np.exp(velocity_intercept) * dt_values**velocity_slope, label=f'Slope: {velocity_slope:.2f}', linestyle='--')
plt.xlabel('Time step (dt)')
plt.ylabel('Velocity error')
plt.grid(True)
plt.legend()
plt.show()
# Plot position error vs time step
plt.figure(figsize=(10, 6))
plt.loglog(dt_values, position_errors, marker='o', label='Numerical position error')
plt.loglog(dt_values, np.exp(position_intercept) * dt_values**position_slope, label=f'Slope: {position_slope:.2f}', linestyle='--')
plt.xlabel('Time step (dt)')
plt.ylabel('Position error')
plt.grid(True)
plt.legend()
plt.show()
```

Figure 9: Simulation code for Boris push