

Capstone Project Phase A

**Examining the Performance of Satellite Propagators for Autonomous Space-Situational-Awareness Satellites**

24-2-R-13

Supervisor: Elad Denenberg

Mahran AbedEllatif - 209120815 Mahran.Abedellatif@e.braude.ac.il

Shadi AbedAlkream 209120096 Shadi.Abd.Alkream@e.braude.ac.il

Table of Contents

[**1.** **Abstract** 5](#_Toc176425901)

[**2.** **Introduction** 5](#_Toc176425902)

[**3.** **Background and Related Work** 6](#_Toc176425903)

[3.1. Space Environment and Collision Risks: 6](#_Toc176425904)

[3.2. Existing Algorithms for State Propagation: 6](#_Toc176425905)

[3.3. Previous Research on Algorithm Evaluation: 7](#_Toc176425906)

[3.4. Objectives of the Current Study: 7](#_Toc176425907)

[**4.** **Expected Achievements** 8](#_Toc176425908)

[**5.** **Algorithms Analysis** 9](#_Toc176425909)

[**5.1.** **Runge-Kutta 4th Order (RK4)[3][4]** 10](#_Toc176425910)

[5.1.1. Mathematical Formulas and Coefficients Table**[13]** 10](#_Toc176425911)

[5.1.2. Pseudocode: 12](#_Toc176425912)

[5.1.3. Time Complexity: 13](#_Toc176425913)

[5.1.4. Space Complexity: 13](#_Toc176425914)

[5.1.5. Edge Cases and Limitations 13](#_Toc176425915)

[**5.2.** **Runge-Kutta 8th Order (RK8) [4][6]** 14](#_Toc176425916)

[5.2.1. Mathematical Formulas and Coefficients table**[5][13]** 14](#_Toc176425917)

[5.2.2. Pseudocode: 16](#_Toc176425918)

[5.2.3. Time Complexity: 17](#_Toc176425919)

[5.2.4. Space Complexity: 17](#_Toc176425920)

[5.2.5. Edge Cases and Limitations 17](#_Toc176425921)

[**5.3.** **Dormand-Price Method (ODE45)** **[4][14]** 18](#_Toc176425922)

[5.3.1. Mathematical Formulas and Coefficients Table**[5][13]** 18](#_Toc176425923)

[5.3.2. Pseudocode: 20](#_Toc176425924)

[5.3.3. Time Complexity: 21](#_Toc176425925)

[5.3.4. Space Complexity: 21](#_Toc176425926)

[5.3.5. Edge Cases and Limitations: 21](#_Toc176425927)

[**5.4.** **Verner’s Method (ODE78)[6][12]** 22](#_Toc176425928)

[5.4.1. Mathematical Formulas and Coefficients table**[4][15]** 22](#_Toc176425929)

[5.4.2. Pseudocode: 24](#_Toc176425930)

[5.4.3. Time Complexity: 25](#_Toc176425931)

[5.4.4. Space Complexity: 25](#_Toc176425932)

[5.4.5. Edge Cases and Limitations: 25](#_Toc176425933)

[**5.5.** **Adams-Bashforth-Moulton PECE Solver (ODE113)[7]** 26](#_Toc176425934)

[5.5.1. Mathematical Formulas and Coefficients**[16][7]** 26](#_Toc176425935)

[5.5.2. Pseudocode: 28](#_Toc176425936)

[5.5.3. Time Complexity: 30](#_Toc176425937)

[5.5.4. Space Complexity: 30](#_Toc176425938)

[5.5.5. Edge Cases and Limitations: 30](#_Toc176425939)

[**5.6.** **Modified Picard-Chebyshev Iteration (MPCI)[8]** 31](#_Toc176425940)

[5.6.1. Mathematical Formulas and Coefficient 31](#_Toc176425941)

[5.6.2. Pseudocode: 32](#_Toc176425942)

[5.6.3. Time Complexity: 34](#_Toc176425943)

[5.6.4. Space Complexity: 35](#_Toc176425944)

[5.6.5. Edge Cases and Limitations: 35](#_Toc176425945)

[**6.** **Evaluating Algorithms for Implementation** 36](#_Toc176425946)

[**7.** **The Development Process** 38](#_Toc176425947)

[7.1. Workflow 38](#_Toc176425948)

[7.2. Planning the Assemblies 39](#_Toc176425949)

[**8.** **Testing** 40](#_Toc176425950)

[8.1. Objective of the Testing 40](#_Toc176425951)

[8.2. Test Setup 40](#_Toc176425952)

[8.2.1. Algorithms Tested 40](#_Toc176425953)

[8.2.2. Reference Model (SGP4) 40](#_Toc176425954)

[8.2.3. Satellites 41](#_Toc176425955)

[8.3. Testing Procedure 41](#_Toc176425956)

[8.3.1. Algorithm Execution 41](#_Toc176425957)

[8.3.2. Position and Velocity Comparison with SGP4 41](#_Toc176425958)

[8.3.3. Execution Time Measurement 42](#_Toc176425959)

[8.4. Results Recording 42](#_Toc176425960)

[8.4.1. Console Output 42](#_Toc176425961)

[8.4.2. File Logging 42](#_Toc176425962)

[8.5. Visualization 44](#_Toc176425963)

[8.5.1. Individual Satellite Results Plot 44](#_Toc176425964)

[8.5.2. Average Results Plot 46](#_Toc176425965)

[8.6. Summary of Testing 46](#_Toc176425966)

[8.7. Key insights 47](#_Toc176425967)

[**9.** **Conclusion** 47](#_Toc176425968)

[**10.** **References** 48](#_Toc176425969)

# **Abstract**

The growing number of satellites and space debris significantly elevates the risk of collisions, threatening the sustainability of space missions. Autonomous satellites must rely on highly accurate state propagation algorithms to navigate and avoid potential collisions. This project evaluates the performance of six state propagation algorithms for predicting satellite positions: Runge-Kutta methods (RK4, RK8), Dormand-Prince (ODE45), Verner’s Method (ODE78), Adams-Bashforth-Moulton (ODE113), and Modified Picard-Chebyshev Iteration (MPCI). The algorithms were tested using real-world satellite data in a simulated environment to compare accuracy, computational efficiency, and execution time against the Standard General Perturbation (SGP4) model. The results offer insights into the trade-offs between precision and computational cost, guiding the selection of the optimal algorithm for use in autonomous satellite navigation systems.

# **Introduction**

The exponential increase in the number of satellites and the accumulation of space debris have heightened the risk of collisions in space, posing a significant threat to the safety and sustainability of space missions. Autonomous satellites require precise and reliable algorithms to propagate the states of orbiting objects in time, allowing them to calculate distances between themselves and surrounding objects to navigate and avoid potential collisions effectively. This study aims to research and evaluate six specific algorithms for state propagation: Runge-Kutta methods (RK4 and RK8), Ordinary Differential Equation (ODE) solvers (ODE45 - Dormand-Prince method, ODE78 – Verner’s method, and ODE113 - Adams-Bashforth-Moulton PECE solver), and the Modified Picard-Chebyshev Iteration.

The research will involve a thorough literature review to understand existing state propagation techniques and their applications in satellite navigation systems. We will use relevant data on satellite positions and space debris and employ simulation tools, specifically C and C++, to test and evaluate the selected algorithms in various scenarios. The evaluation process will be iterative, allowing us to refine our criteria and improve the accuracy and efficiency of the algorithms. The ultimate goal is to identify the most effective algorithm for autonomous satellite navigation and collision avoidance, contributing to the advancement of space safety and operational efficiency.

# **Background and Related Work**

## Space Environment and Collision Risks:

The rapid increase in satellite deployments and the accumulation of space debris have created a densely populated and hazardous space environment, significantly elevating the risk of collisions. This congestion poses a serious threat to the safety and sustainability of space missions [[1]](#Refrence1). Effective collision avoidance strategies are paramount for the continued viability of satellite operations [[2]](#Refrence2). Autonomous satellites, in particular, need reliable and precise algorithms to calculate distances between themselves and nearby objects to navigate and avoid potential collisions without constant human intervention.

## Existing Algorithms for State Propagation:

Several algorithms have been developed and applied in various fields for solving ordinary differential equations (ODEs) that model the motion of satellites and space debris. Among these, the Runge-Kutta methods (RK4 and RK8) are widely recognized for their balance of accuracy and computational efficiency. RK4, a fourth-order method, is a popular choice for many engineering applications due to its simplicity and robustness [[3]](#Refrence3). RK8, an eighth-order method, offers higher accuracy but at a greater computational cost [[4]](#Refrence4). The difference between these methods lies in the order of accuracy, which refers to how the error decreases as the step size decreases. The Butcher tableau is a matrix used to describe the coefficients of Runge-Kutta methods, providing a structured way to implement these algorithms.

In In addition to the Runge-Kutta methods, various ODE solvers such as Dormand-Prince method (ODE45), Verner’s method (ODE78), and Adams-Bashforth-Moulton (ODE113) have been extensively used in numerical analysis. ODE45 is particularly noted for its adaptive step size control, which enhances computational efficiency by adjusting the step size based on the error estimate [[5]](#Refrence5) . ODE78 and ODE113 are higher-order solvers that provide greater accuracy and are suitable for stiff and non-stiff problems, respectively [[6]](#Refrence6) .

The Modified Picard-Chebyshev Iteration (MCPI) combines the Picard iteration method with Chebyshev polynomials to enhance convergence and accuracy. This method has shown potential in solving complex differential equations with improved precision, making it a candidate for space navigation applications [[8]](#Refrence8). One of its key advantages is the avoidance of inverse matrix calculations, which can be time-consuming.

## Previous Research on Algorithm Evaluation:

Prior research has focused on evaluating these algorithms in various contexts, including orbital mechanics, spacecraft trajectory optimization, and collision avoidance systems. Studies have demonstrated the effectiveness of Runge-Kutta methods and ODE solvers in simulating satellite orbits and predicting close approaches with debris [[10]](#Refrence10). However, the unique requirements of autonomous satellite navigation—such as real-time processing, robustness to uncertainties, and minimal reliance on ground control—necessitate a thorough comparative analysis of these algorithms to identify the most suitable one for this specific application [[11]](#Refrence11).

## Objectives of the Current Study:

This study aims to fill this gap by systematically evaluating the performance of RK4, RK8, Dormand-Prince method (ODE45),Verner’s method )ODE78 ), Adams-Bashforth-Moulton (ODE113), and the Modified Picard-Chebyshev Iteration in propagating the states of satellites and space debris. Using existing data on satellite positions and space debris, we will employ simulation tools, specifically C and C++, to test and evaluate these algorithms in various scenarios. Our goal is to determine the most effective algorithm for enhancing the safety and operational efficiency of autonomous satellites.

# **Expected Achievements**

we anticipate achieving the following milestones:

1. **Comprehensive Literature Review:**
   * Gaining a detailed understanding of existing distance calculation algorithms and their applications in satellite navigation systems.
2. **Algorithm Identification:**
   * Selecting six promising algorithms for detailed evaluation: RK4 (Runge-Kutta 4th Order), RK8 (Runge-Kutta 8th Order), Dormand-Prince Method (ODE45), Verner’s Method (ODE78), Adams-Bashforth-Moulton (ODE113), and the Modified Picard-Chebyshev Iteration.
3. **Simulation Setup:**
   * Implementing simulation tools (C and C++) configured for testing the selected algorithms.
   * Implementation includes the possibility to integrate with future algorithms, enhancing the longevity and utility of the simulation setup.
4. **Algorithm Testing:**
   * Conducting initial tests of the selected algorithms in various simulated scenarios to assess their accuracy and efficiency.
   * We have a small CubeSat at Braude College. If time permits, we plan to implement the algorithms and test them in a real-life satellite environment. This will provide practical insights and validate the effectiveness of the algorithms under real operational conditions.
5. **Iterative Analysis:**
   * Performing iterative analysis to refine evaluation criteria and improve algorithm performance.
6. **Preliminary Findings:**
   * Producing preliminary results that identify the most effective algorithms for autonomous satellite navigation and collision avoidance.
7. **Documentation:**
   * Thoroughly documenting all research findings, methodologies, and initial evaluation results for future reference and further research.

# **Algorithms Analysis**

To evaluate the performance of the selected algorithms Runge-Kutta 4th Order - RK4 , Runge-Kutta 8th Order - RK8, Dormand-Price Method - ODE45, Verner’s Method - ODE78, Adams-Bashforth-Moulton - ODE113, and the Modified Picard-Chebyshev Iteration - MPCI, we will solve two primary sets of equations of motion that govern the dynamics of satellites and space debris. The first set includes the equation of motion with only gravitational forces:

where represents the position vector of the satellite or debris, is the standard gravitational parameter.

The second set of equations accounts for gravitational forces along with corrections for additional forces:

Here, represents the acceleration due to atmospheric drag, and denotes the acceleration resulting from perturbations such as the Earth's motion and gravitational influences from other celestial bodies.

Using C and C++ for simulation, we will implement each algorithm to solve these equations of motion iteratively. The algorithms will propagate the state vectors of satellites and debris over time, allowing us to calculate their trajectories under varying conditions. By comparing the results, we will assess each algorithm's accuracy, computational efficiency, and robustness in handling different scenarios, ultimately identifying the most suitable approach for autonomous satellite navigation and collision avoidance.

## **Runge-Kutta 4th Order (RK4)**[**[3]**](#Refrence3)[**[4]**](#Refrence4)

**Purpose:** RK4 is a numerical method used for solving ordinary differential equations (ODEs). It provides a balance between accuracy and computational efficiency and is commonly used in various scientific and engineering applications.

**Overview:** RK4 works by iteratively calculating the solution to an ODE using four intermediate steps (k1, k2, k3, and k4) to estimate the slope of the solution at different points within each step.

### Mathematical Formulas and Coefficients Table[**[13]**](#Refrence13)

**Formulas:**



**Update Formula**

**General Formulation**

​: The current value of the solution.

​: The next value of the solution.

: The step size, defined as the difference between consecutive time points

: The number of stages in the Runge-Kutta method **in this case** **4**.

​: The weights used to combine the intermediate slopes to obtain the

final solution.

​: The intermediate slopes, calculated using the function at different

points.

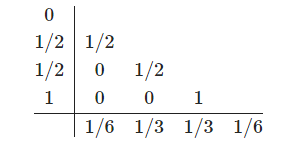
: The current time point.

​: The coefficients that determine the evaluation points within the step.

​: The coefficients that weight the contributions of the intermediate slopes

to calculate the next slope .

**Coefficient Table for RK4:**



### Pseudocode:

**function rk4(func, tspan, y0, h)**

**// Parameters:**

**// func - function defining the ODE (dy/dx = func(x, y))**

**// tspan - tuple (t0, tf) specifying the time range**

**// y0 - initial condition**

**// h - step size**

t0 = tspan[0]

tf = tspan[1]

t\_values = range from t0 to tf with step size h

y\_values = initialize matrix of zeros with dimensions (length of t\_values, length of y0)

y = y0

**for each t in t\_values**

store y in y\_values at current index

k1 = h \* func(t, y)

k2 = h \* func(t + h/2, y + k1/2)

k3 = h \* func(t + h/2, y + k2/2)

k4 = h \* func(t + h, y + k3)

y = y + (k1 + 2\*k2 + 2\*k3 + k4) / 6

results = combine t\_values and y\_values into a single matrix

**return** results

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O (n), where 𝑛 = len(t) – t is array of time steps

**Explanation:** Each iteration involves a constant number of operations to compute the four slopes (k1, K2, k3, k4) and update the solution. Since the total number of iterations 𝑛.

### Space Complexity:

* **Overall:** O ()

**Explanation:** The method requires storage for the time points and solution values across all time steps. This contributes to a space complexity of O () ,where is the number of time steps and is the dimension of the solution vector y. Additionally, a constant amount of space is needed for intermediate slopes (k1​, k2​, k3​, k4​) and the current values of y and t, but this does not affect the overall space complexity, which scales with the problem size.

### Edge Cases and Limitations

Using an excessively small step size in the RK4 method can lead to a large number of iterations and increased computational cost, while a large step size can result in inaccurate solutions and potentially unstable behavior. Additionally, RK4 may not be the best choice for stiff ODEs, where implicit methods or adaptive step size methods may be required for stability and efficiency. Although RK4 is generally accurate, the global error can still accumulate over a large number of steps, making it important to consider error control mechanisms.

**Conclusion:** RK4 is a robust and widely-used method for solving ODEs, offering a good trade-off between accuracy and computational efficiency for many practical problems. However, for specific cases like stiff equations, alternative methods may be more appropriate.

## **Runge-Kutta 8th Order (RK8)** [**[4]**](#Refrence4)[**[6]**](#Refrence6)

**Purpose:** RK8 is a numerical method used for solving ordinary differential equations (ODEs) with higher accuracy than lower-order methods. It is particularly useful in applications requiring precise solutions over long intervals or where the function being integrated has rapid variations.

**Overview:** RK8 works by calculating the solution to an ODE using eight intermediate steps to estimate the slope of the solution more accurately at different points within each step. The method reduces truncation errors by considering more points within each interval.

### Mathematical Formulas and Coefficients table[**[5]**](#Refrence5)[**[13]**](#Refrence13)

**Formulas:**



**Update Formula**

**General Formulation**

​: The current value of the solution.

​: The next value of the solution.

: The step size, defined as the difference between consecutive time points

: The number of stages in the Runge-Kutta method **in this case** **13**.

​: The weights used to combine the intermediate slopes to obtain the

final solution.

​: The intermediate slopes, calculated using the function at different

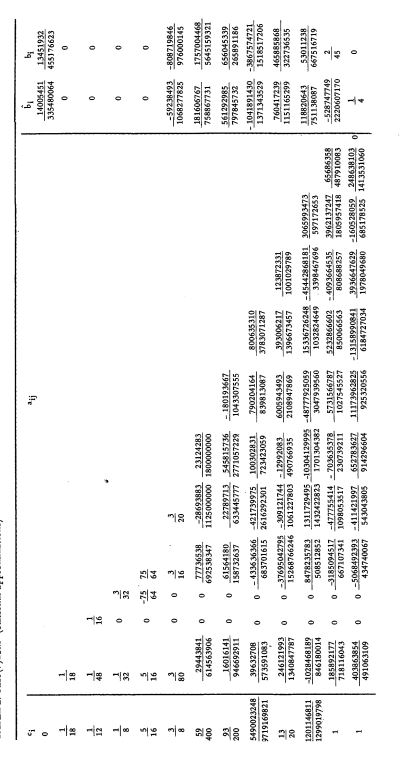
points.

: The current time point.

​: The coefficients that determine the evaluation points within the step.

​: The coefficients that weight the contributions of the intermediate slopes

to calculate the next slope .

**Coefficient Table for RK8:**

### **Pseudocode:**

**function RK8(tspan, r0, h, f)**

**// Parameters:**

**// tspan - tuple with start and end times**

**// r0 - initial state vector**

**// h - step size**

**// f - function that returns derivatives**

t\_start = tspan[0]

t\_end = tspan[1]

n = (t\_end - t\_start) / h + 1

t = linspace(t\_start, t\_end, n)

r = [r0]

**for i from 0 to n-2**

k = initialize matrix of zeros with dimensions (length of Butcher\_table\_DP8['c'], length of r0)

ti = t[i]

ri = r[i]

// Calculate k values using Butcher table coefficients

**for j, cj in enumerate(Butcher\_table\_DP8['c'])**

**if j == 0**

y\_temp = ri // No coefficients for the first step

**else**

y\_temp = ri + h \* sum(Butcher\_table\_DP8['a'][j][l] \* k[l] for l from 0 to min(j, length of

Butcher\_table\_DP8['a'][j]) - 1)

k[j] = f(ti + cj \* h, y\_temp)

// Update the state vector using the k values and the b coefficients

ri\_new = ri + h \* sum(Butcher\_table\_DP8['b'][j] \* k[j] for j from 0 to length of k - 1)

append ri\_new to r

results = combine t and r into a single matrix

**return** results

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where n=len(t) – t is array of time steps

**Explanation:** Each iteration involves a constant number of operations to compute the intermediate slopes (using the Butcher tableau coefficients) and update the solution vector. The RK8 method, although more complex than RK4 due to the higher number of stages, still maintains a constant amount of work per iteration. Since there are n iterations (one for each time step), the total time complexity remains linear in relation to the number of time steps.

### Space Complexity:

* **Overall:** O ()

**Explanation:** The method requires storage for the time points and solution vectors across all time steps, leading to a space complexity of O (), where n is the number of time steps and m is the dimension of the state vector r. Additionally, space is required to store the intermediate slopes k, but this space requirement is constant and does not affect the overall space complexity, which scales with the size of the problem.

### Edge Cases and Limitations

Using an excessively small step size in the RK8 method can lead to a large number of iterations and significantly increased computational cost, while a large step size can result in inaccurate solutions and potentially unstable behavior. Additionally, RK8 may not be the best choice for stiff ODEs, where implicit methods or adaptive step size methods may be required for stability and efficiency. Although RK8 is highly accurate, the global error can still accumulate over a large number of steps, necessitating error control mechanisms.

**Conclusion:** RK8 is a robust and highly accurate method for solving ODEs, offering superior accuracy compared to lower-order methods. It is particularly useful for problems requiring precise solutions. However, for specific cases like stiff equations or where computational efficiency is a priority, alternative methods may be more appropriate.

## **Dormand-Price Method (ODE45)** [**[4]**](#Refrence4)[**[14]**](#Refrence14)

**Purpose:** Dormand-Prince (ODE45) is a numerical method used for solving ordinary differential equations (ODEs). It is based on an adaptive Runge-Kutta method of order 4(5). ODE45 is designed to provide a good balance between accuracy and computational efficiency and is commonly used in various scientific and engineering applications.

**Overview:** Dormand-Prince (ODE45) adapts the step size as it integrates an ODE, using a combination of fourth and fifth-order Runge-Kutta formulas to control error. It selects an optimal step size at each iteration to meet specified error tolerances, improving efficiency and stability over fixed-step methods.

### Mathematical Formulas and Coefficients Table[**[5]**](#Refrence5)[**[13]**](#Refrence13)

**Formulas:**



**Update Formula**

**General Formulation**

**Error Estimation**

The Error Estimate can be computed using different set of weights

​: The current value of the solution.

​: The next value of the solution.

: The step size, defined as the difference between consecutive time points

: The number of stages in the Runge-Kutta method **in this case** **7**.

​: The weights used to combine the intermediate slopes to obtain the

final solution.

​: The intermediate slopes, calculated using the function at different

points.

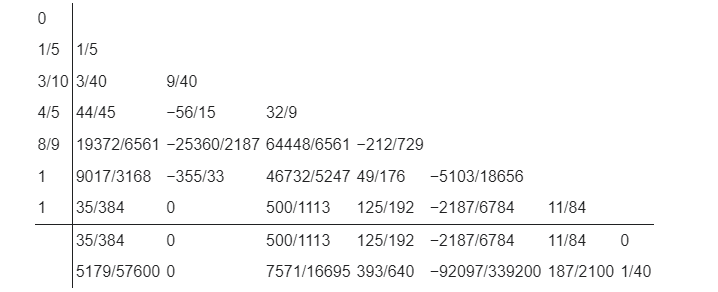
: The current time point.

​: The coefficients that determine the evaluation points within the step.

​: The coefficients that weight the contributions of the intermediate slopes

to calculate the next slope .

**Coefficient Table for ODE45:**



### **Pseudocode:**

// Compute the 4th-order and 5th-order solutions

y4th = y + h \* (b1 \* K1 + b3 \* K3 + b4 \* K4 + b5 \* K5 + b6 \* K6)

y5th = y + h \* (b1p \* K1 + b3p \* K3 + b4p \* K4 + b5p \* K5 + b6p \* K6 + b7p \* K7)

// Estimate the error

error = norm(y5th - y4th)

// Calculate the scaling factor

delta = 0.84 \* (tol / error) ^ (1.0 / 5.0)

**if error < tol**

t += h

y = y5th // Update y with the higher-order estimate (5th-order)

// Adjust step size based on delta

**if delta <= 0.1**

h \*= 0.1

**else if delta >= 4.0**

h \*= 4.0

**else**

h \*= delta

// Ensure the step size stays within bounds

h = max(min(h, hmax), hmin)

// Adjust final step if it overshoots

**if t + h > tf**

h = tf - t

// Convert results to arrays

t\_values = array(t\_values)

y\_values = array(y\_values)

results = combine t\_values and y\_values into a single matrix

**return** results

**function ode45(func, tspan, y0, tol, hmax, hmin)**

**// Parameters:**

**// func - function defining the ODE (dy/dx = func(x, y))**

**// tspan - tuple (t0, tf) specifying the time range**

**// y0 - initial condition**

**// tol - tolerance for adaptive step size**

**// hmax - maximum step size**

**// hmin - minimum step size**

t0 = tspan[0]

tf = tspan[1]

t\_values = empty list

y\_values = empty list

// Initial conditions

t = t0

y = y0

h = hmax

// Coefficients for the Dormand-Prince method are represented with the letter a,b,c

**while t < tf**

append t to t\_values

append y to y\_values

// Compute function values

K1 = func(t, y)

K2 = func(t + c2 \* h, y + h \* (a21 \* K1))

K3 = func(t + c3 \* h, y + h \* (a31 \* K1 + a32 \* K2))

K4 = func(t + c4 \* h, y + h \* (a41 \* K1 + a42 \* K2 + a43 \* K3))

K5 = func(t + c5 \* h, y + h \* (a51 \* K1 + a52 \* K2 + a53 \* K3 + a54 \* K4))

K6 = func(t + h, y + h \* (a61 \* K1 + a62 \* K2 + a63 \* K3 + a64 \* K4 + a65 \* K5))

K7 = func(t + h, y + h \* (a71 \* K1 + a73 \* K3 + a74 \* K4 + a75 \* K5 + a76 \* K6))

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(*n*) where n is the number of adaptive steps taken by the solver

**Explanation:** Each iteration involves a constant number of operations, specifically the computation of seven slopes (K1 through K7) and their corresponding weighted sums to produce 4th and 5th-order estimates. The total complexity depends on the number of steps n taken, which varies with the adaptive step size mechanism. However, each individual step remains O(1).

### Space Complexity:

* **Overall:** O ()

**Explanation:** T he method stores the time points and solution vectors across all time steps, contributing O () to the space complexity, where n is the number of steps and m is the dimension of the state vector y. Additional space is required to store the intermediate slopes (K1 through K7), but these are temporary and do not affect the overall scaling of the space complexity, which is dominated by the storage of the results.

### Edge Cases and Limitations:

ODE45 may not be efficient for stiff ODEs, where specialized methods like implicit Runge-Kutta or adaptive step size methods might be more appropriate. For solutions with very rapid changes, ODE45 may require excessively small step sizes to maintain accuracy, increasing computational cost. Additionally, the adaptive step size control introduces computational overhead, which may be significant for simple problems where fixed-step methods suffice.

**Conclusion:** ODE45 is a robust and widely-used method for solving ODEs, offering a good trade-off between accuracy and computational efficiency. Its adaptive step size control makes it suitable for problems with varying solution behavior. However, for specific cases like stiff equations or simple problems, alternative methods may be more appropriate.

## **Verner’s Method (ODE78)**[**[6]**](#Refrence6)[**[12]**](#Refrence12)

**Purpose:** Verner’s Method (ODE78) is a numerical method used for solving ordinary differential equations (ODEs). It is based on an adaptive Runge-Kutta method of order 7(8). ODE78 is designed to provide very high accuracy by using higher-order methods, making it suitable for solving complex problems with stringent accuracy requirements.

**Overview:** Verner’s Method (ODE78) adapts the step size as it integrates an ODE, using a combination of seventh and eighth-order Runge-Kutta formulas to control error. It selects an optimal step size at each iteration to meet specified error tolerances, improving efficiency and stability over fixed-step methods.

### Mathematical Formulas and Coefficients table[**[4]**](#Refrence6)[**[15]**](#Refrence15)

**Formulas:**

**Update Formula**

**General Formulation**

**Error Estimation**

The Error Estimate can be computed using different set of weights



​: The current value of the solution.

​: The next value of the solution.

: The step size, defined as the difference between consecutive time points

: The number of stages in the Runge-Kutta method **in this case** **13**.

​: The weights used to combine the intermediate slopes to obtain the

final solution.

​: The intermediate slopes, calculated using the function at different

points.

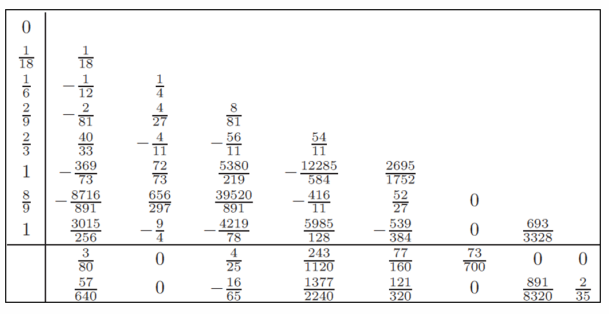
: The current time point.

​: The coefficients that determine the evaluation points within the step.

​: The coefficients that weight the contributions of the intermediate slopes

to calculate the next slope .

**Coefficient Table for ODE78:**



### **Pseudocode:**

**function ode78(func, tspan, y0, h)**

**// Parameters:**

**// func - function defining the ODE (dy/dx = func(x, y))**

**// tspan - tuple (t0, tf) specifying the time range**

**// y0 - initial condition**

**// h - step size**

t0 = tspan[0]

tf = tspan[1]

t = t0

y = y0

**// Butcher tableau coefficients for ODE78 are represented with**

**alpha = [0, 1/18, 1/6, 2/9, 2/3, 1, 8/9, 1]**

**beta = [**

**[], // No coefficients for the first row**

**[1/18],**

**[-1/12, 1/4],**

**[-2/81, 4/27, 8/81],**

**[40/33, -4/11, -56/11, 54/11],**

**[-369/73, 72/73, 5380/219, -12285/584, 2695/1752],**

**[-8716/891, 656/297, 39520/891, -416/11, 52/27, 0],**

**[3015/256, -9/4, -4219/78, 5985/128, -539/384, 0, 693/3328]**

**]**

**chi = [3/80, 0, 4/25, 243/1120, 77/160, 73/700, 0, 0]**

**psi = [57/640, 0, -16/65, 1377/2240, 121/320, 0, 891/8320, 2/35]**

// Arrays to store time and solution values

t\_values = [t]

y\_values = [y]

**while t < tf**

**if t + h > tf**

h = tf - t

// Initialize k array

k = array of zeros with shape (8, length of y0)

// Compute k values

**for i from 0 to 7**

sum\_beta\_k = array of zeros with same shape as y

**for j from 0 to i-1**

sum\_beta\_k += beta[i][j] \* k[j]

k[i] = h \* func(t + alpha[i] \* h, y + sum\_beta\_k)

// Update y using the chi coefficients

y = y + sum of (chi[i] \* k[i] for i from 0 to 7)

// Estimate error (optional, if psi coefficients are given)

error\_est = sum of (psi[i] \* k[i] for i from 0 to 7)

// Update time

t += h

// Store the results

append t to t\_values

append y to y\_values

// Convert results to arrays

t\_values = array(t\_values)

y\_values = array(y\_values)

results = combine t\_values and y\_values into a single matrix

**return** results

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where n is the number of time steps

**Explanation:** Each iteration involves a constant number of operations, specifically the computation of eight stages (k1​ through k8​) using the Butcher tableau coefficients and updating the solution vector. The complexity per iteration is constant, and the total complexity is linear in the number of time steps n.

### Space Complexity:

* **Overall:** O ()

**Explanation:** The method requires storage for time points and solution vectors across all time steps, leading to a space complexity of O () ,where n is the number of time steps and mmm is the dimension of the state vector y. Additionally, space is needed to store the intermediate slopes (k1​ through k8​), but this requirement is constant and does not affect the overall space complexity.

### Edge Cases and Limitations:

ODE78 may not be efficient for stiff ODEs, where specialized methods like implicit Runge-Kutta or adaptive step size methods might be more appropriate. For solutions with very rapid changes, ODE78 may require excessively small step sizes to maintain accuracy, increasing computational cost. Additionally, the adaptive step size control introduces computational overhead, which may be significant for simple problems where fixed-step methods suffice.

**Conclusion:** ODE78 is a robust and highly accurate method for solving ODEs, offering superior accuracy compared to lower-order methods. Its adaptive step size control makes it suitable for problems with varying solution behavior. However, for specific cases like stiff equations or simple problems, alternative methods may be more appropriate.

## **Adams-Bashforth-Moulton PECE Solver (ODE113)**[**[7]**](#Refrence7)

**Purpose:** Adams-Bashforth-Moulton **(**ODE113) is a numerical method used for solving ordinary differential equations (ODEs). It is based on an Adams-Bashforth-Moulton PECE solver of variable order (1 to 13). ODE113 is designed for non-stiff problems and provides a good balance between accuracy and efficiency by adapting both the step size and the method order.

**Overview:** Adams-Bashforth-Moulton **(**ODE113) uses a predictor-corrector approach with the Adams-Bashforth method for prediction and the Adams-Moulton method for correction. The algorithm dynamically adjusts both the step size and the order of the method to control the local error, making it suitable for problems with smooth solutions over long intervals.

### Mathematical Formulas and Coefficients**[[16]](#Refrence16)**[**[[7]](#Refrence16)**](#Refrence7)

**Adams-Bashforth Method (Predictor)**

The Adams-Bashforth method is an explicit multistep method. The general formula for the k-step Adams-Bashforth method is:

where is the step size, is the current value, and is the function representing the ODE

The coefficients depend on the number of steps k.

**Adams-Moulton Method (Corrector)**

The Adams-Multon method is an implicit multistep method. The general formula for the k- steps Adams-Multon method is:

The coefficients ​ depend on the number of steps k.

**PECE Algorithm**

In the PECE (Predict, Evaluate, Correct, Evaluate) approach, the Adams-Bashforth method is used to predict the value of ​, and the Adams-Moulton method is used to correct this prediction.

* **Predict**: Use the Adams-Bashforth method to predict
* **Evaluate**: Evaluate the function at the predicted point
* **Correct**: Use the Adams-Moulton method to correct
* **Evaluate**: Recompute the function at the corrected point if needed

### **Pseudocode:**

**function adams\_bashforth\_moulton\_adaptive(f, tspan, y0, tol, hmin=1e-6, hmax=1.0)**

**// Parameters:**

**// f - function defining the ODE (dy/dx = f(t, y))**

**// tspan - tuple (t0, tf) specifying the time range**

**// y0 - initial condition**

**// tol - tolerance for adaptive step size**

**// hmin - minimum allowable step size**

**// hmax - maximum allowable step size**

t0 = tspan[0]

tf = tspan[1]

h = (tf - t0) / 1000 // Initial step size estimate

t\_values\_list = [t0]

y\_values\_list = [y0]

// Initial conditions for the first few steps using RK4

t = t0

y = y0

// Compute initial points using RK4

**for \_ from 0 to 2**

y\_next = rk4\_step(f, t, y, h)

t += h

append t to t\_values\_list

append y\_next to y\_values\_list

y = y\_next

// Adams-Bashforth-Moulton method with adaptive step size

**while t < tf**

**if length of t\_values\_list < 4**

y\_pred = rk4\_step(f, t, y, h)

**else**

// Adams-Bashforth predictor

y\_pred = y\_values\_list[-1] + h/24 \* (

55\*f(t\_values\_list[-1], y\_values\_list[-1])

- 59\*f(t\_values\_list[-2], y\_values\_list[-2])

+ 37\*f(t\_values\_list[-3], y\_values\_list[-3])

- 9\*f(t\_values\_list[-4], y\_values\_list[-4])

)

// Adams-Moulton corrector

t\_next = t + h

y\_correct = y\_values\_list[-1] + h/24 \* (

9\*f(t\_next, y\_pred)

+ 19\*f(t\_values\_list[-1], y\_values\_list[-1])

- 5\*f(t\_values\_list[-2], y\_values\_list[-2])

+ f(t\_values\_list[-3], y\_values\_list[-3])

)

// Error estimation

error = abs(y\_correct - y\_pred)

// Step size adjustment

**if any element of error > tol**

h \*= 0.9 \* (tol / max element of error) ^ 0.25

**if h < hmin**

print("Warning: Step size too small. Ending integration.")

**break**

**else**

append t\_next to t\_values\_list

append y\_correct to y\_values\_list

t = t\_next // Update time

h = min(h \* 1.5, hmax) // Increase step size for efficiency

// Check if we're close to the final time to avoid getting stuck

**if tf - t < hmin**

print("Warning: Close to final time. Ending integration.")

**break**

results = combine t\_values\_list and y\_values\_list into a single matrix

**return** result

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where n the number of adaptive steps taken by the solver.

**Explanation:** Each iteration involves computing the predictor and corrector steps, which each require evaluating the function f(t,y) a constant number of times. The complexity per iteration is constant O(1), and the total complexity depends on the number of steps n required, which is adaptive and based on the error tolerance and step size control.

### Space Complexity:

* **Overall:** O ()

**Explanation:** The method stores the time points and solution vectors across all adaptive steps, leading to a space complexity of O () ,where n is the number of steps and m is the dimension of the state vector y. Additional space is used for storing intermediate results (e.g., the predicted and corrected values), but this is constant and does not affect the overall scaling of space complexity.

### Edge Cases and Limitations:

ODE113 may not be efficient for stiff ODEs, where specialized methods like implicit Runge-Kutta or other stiff solvers might be more appropriate. For solutions with high oscillations, ODE113 may require small step sizes to maintain accuracy, increasing computational cost. Additionally, the adaptive step size and variable order control introduce computational overhead, which may be significant for simple problems where fixed-step methods suffice.

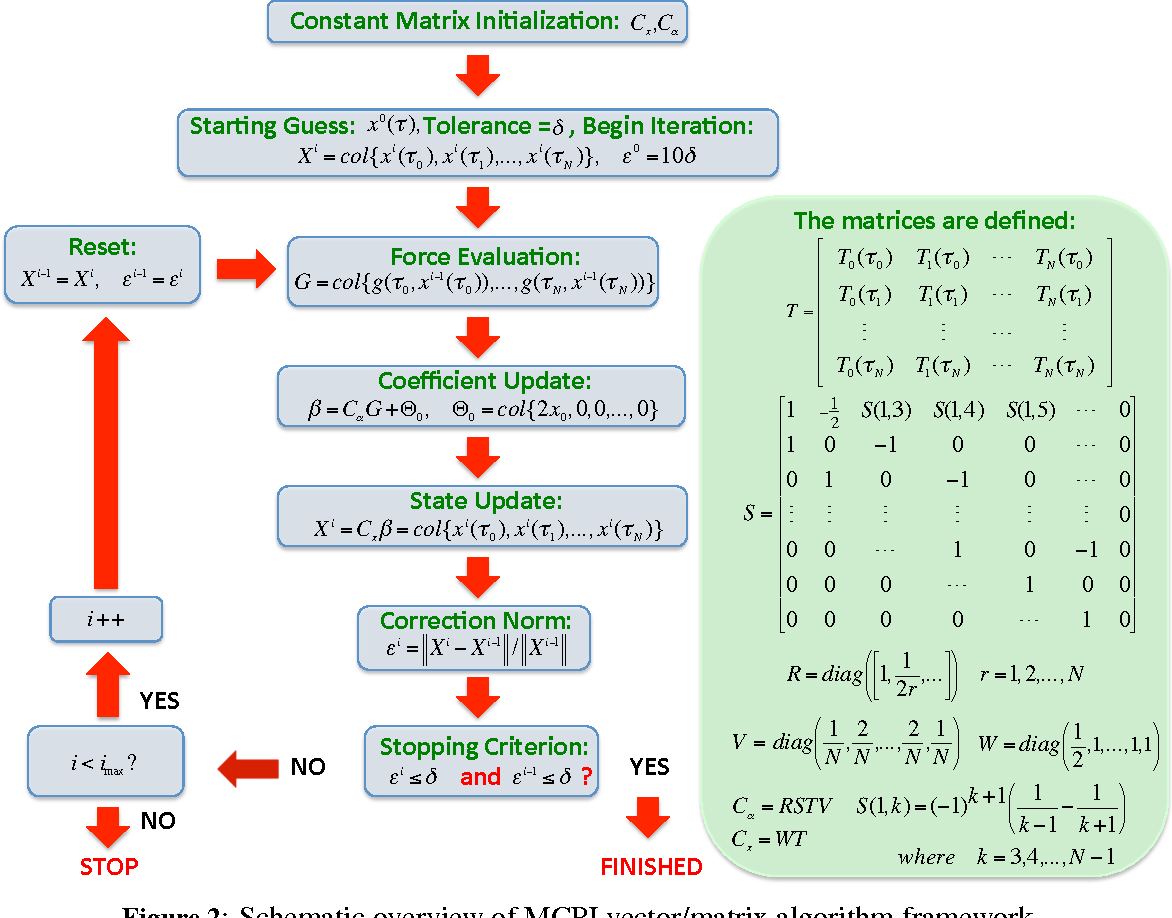
**Conclusion:** ODE113 is a robust and highly accurate method for solving non-stiff ODEs, offering superior accuracy and efficiency compared to fixed-order methods. Its adaptive step size and variable order control make it suitable for problems with varying solution behavior over long intervals. However, for specific cases like stiff equations or simple problems, alternative methods may be more appropriate.

## **Modified Picard-Chebyshev Iteration (MPCI)**[**[8]**](#Refrence8)

**Purpose:** MPCI is a numerical method used for solving ordinary differential equations (ODEs) with high accuracy and stability. It combines the Picard iteration with Chebyshev polynomials to iteratively improve the solution. This method is particularly useful in applications requiring precise solutions over long intervals or where the function being integrated has complex behavior.

**Overview:** MPCI approximates the solution to an ODE by iteratively refining it using Picard iteration and representing the solution in terms of Chebyshev polynomials. Chebyshev polynomials provide a powerful tool for approximating functions with rapid convergence and numerical stability.

### Mathematical Formulas and Coefficient

****

### **Pseudocode:**

// Update using error and convergence check

errorAndUpdate(M, h, length of y0, x0, Xo, Xn, xAdd,

temp)

// Check for convergence

if norm(Xn - Xo) < tol

break

Xo = copy of Xn

// Store the result at the current time step

y\_values[i] = Xn[:length of y0]

results = combine t\_values and y\_values into a single matrix

**return** results

**function MCPI(func, tspan, y0, h, N, tol=1e-10, max\_iter=100)**

**// Parameters:**

**// func - function defining the ODE (dy/dt = func(t, y))**

**// tspan - tuple (t0, tf) specifying the time range**

**// y0 - initial condition**

**// h - step size**

**// N - number of Chebyshev nodes**

**// tol - tolerance for convergence**

**// max\_iter - maximum number of iterations**

t0 = tspan[0]

tf = tspan[1]

t\_values = range from t0 to tf with step size h

M = N + 1

y\_values = initialize matrix of zeros with dimensions

(length of t\_values, length of y0))

y\_values[0] = y0

tau = cos(linspace(0, pi, M))

// Initialize variables for iteration

Xn = array of zeros with length M \* length of y0

Xo = array of zeros with same shape as Xn

xAdd = array of zeros with same shape as Xn

temp = 0.0

// Precompute Chebyshev coefficients

Im = MCPI\_CoeffsI(N, M)

**for i from 1 to length of t\_values - 1**

t = t\_values[i-1]

y = y\_values[i-1]

x0 = y

**for iteration from 0 to max\_iter - 1**

// Update Xn based on the previous step and current estimate

**for node from 0 to M - 1**

tau\_val = tau[node]

tn = t + h \* (tau\_val + 1) / 2

xAdd[node \* length of y0 : (node + 1) \* length

of y0] = func(tn, y)

// BUILD Z MATRIX

**for j from 0 to M - 1**

**for i from 0 to N + 1**

Z[i, j] = (-1) ^ (i + 2)

// BUILD V MATRIX

vElem = 1.0 / N

V[0, 0] = vElem

V[N, N] = vElem

for i from 1 to N - 1

V[i, i] = 2.0 \* vElem

I\_N[0, 1] = 1.0

I\_N[1, 0] = 0.25

I\_N[1, 2] = 0.25

**for ii from 2 to N**

I\_N[ii, ii - 1] = -0.5 / (ii - 1)

**if ii < N + 1**

I\_N[ii, ii + 1] = 0.5 / ii

// Building Cx & Ca matrices

WT = matrix multiplication of W and TT

WTV = matrix multiplication of WT and V

ITZ = matrix multiplication of I\_N and T2Z

Im = matrix multiplication of WTV and ITZ

**return** Im

**function MCPI\_CoeffsI(N, M)**

**W = initialize matrix of zeros with dimensions (M, M)**

**T = initialize matrix of zeros with dimensions (N + 1, M)**

**TT = initialize matrix of zeros with dimensions (M, N + 1)**

**T2 = initialize matrix of zeros with dimensions (N + 2, M)**

**T2Z = initialize matrix of zeros with dimensions (N + 2, M)**

**Z = initialize matrix of zeros with dimensions (M + 2, M)**

**V = initialize matrix of zeros with dimensions (N + 1, N + 2)**

**I\_N = initialize matrix of zeros with dimensions (N + 2, N + 2)**

**tau = initialize array of zeros with length M**

**for i from 0 to M - 1**

**tau[i] = cos(i \* pi / N + pi)**

// BUILD W MATRIX

W[0, 0] = 0.5

for i from 1 to M - 2

W[i, i] = 1.0

W[M - 1, M - 1] = 0.5

**// BUILD T MATRIX**

**for j from 0 to M - 1**

**for i from 0 to N**

T[i, j] = cos(i \* arccos(tau[j]))

// BUILD TT MATRIX

**for j from 0 to N**

**for i from 0 to M - 1**

TT[i, j] = cos(j \* arccos(tau[i]))

// BUILD T2 MATRIX

**for j from 0 to M - 1**

**for i from 0 to N + 1**

T2[i, j] = cos(i \* arccos(tau[j]))

// BUILD T2Z MATRIX

**for j from 0 to M - 1**

**for i from 0 to N + 1**

T2Z[i, j] = cos(i \* arccos(tau[j])) - (-1) ^ (i + 2)

**function errorAndUpdate(MM, timeSub, Nstates2, x0, Xo, Xn, xAdd, temp)**

**for node from 0 to MM - 1**

**for state from 0 to Nstates2 - 1**

indx = node \* Nstates2 + state

Xn[indx] = x0[state] + timeSub \* xAdd[indx]

Err = abs(Xn[indx] - Xo[indx]) / max(1.0, abs(Xo[indx]))

**if state == 0** // Initialize temp with the first state's error

temp = Err

**if Err > temp**

temp = Err

Xo[indx] = Xn[indx]

### Time Complexity:

* **Per Iteration:** O(), where N is the degree of the Chebyshev polynomial used in the approximation.
* **Total Complexity:** O(), where I is the number of iterations performed to reach the desired accuracy, and N is the degree of the Chebyshev polynomial

**Explanation:** The time needed for the MCPI method grows with the square of the polynomial degree N. Each iteration involves several matrix operations, including matrix multiplications and evaluations of the function defining the ODE, which contribute to a complexity of O(), Additionally, at the end of the process, combining the time and state values into the results matrix has a complexity of O(T⋅M), where T is the number of time steps and M is the number of states (related to the polynomial degree N). However, since this operation is linear, it does not affect the overall quadratic nature of the total time complexity, which remains O(I⋅N2).

### Space Complexity:

* **Overall:** O ()

**Explanation:** This space is mainly used to store the matrices T, TT, T2, T2Z, Z, and V, each of size N×N, as well as vectors such as Xn and Xo for each iteration, which have a size of N. Additionally, the final results array, which combines the time points and solution values, has a space complexity of O(T⋅M). Nevertheless, the dominant factor in the overall space complexity is the storage of N×N matrices, leading to a total space complexity of O(N2).

### Edge Cases and Limitations:

The Modified Chebyshev-Picard Iteration (MCPI) method may struggle with stiff differential equations due to numerical instability, and using a very high polynomial degree can lead to excessive computation time due to its quadratic complexity with respect to the degree. Additionally, MCPI’s convergence depends on careful selection of the polynomial degree and step size, and incorrect choices can result in slow convergence or inefficiencies. The polynomial approximations used in MCPI might also fail to capture complex solution behaviors, and very high degrees risk overfitting the solution to noise rather than reflecting the true problem dynamics.

**Conclusion:** The Modified Chebyshev-Picard Iteration (MCPI) method is an effective numerical approach for solving ordinary differential equations (ODEs) that benefits from high accuracy due to its use of Chebyshev polynomials. It is especially useful for problems where high precision is needed, as it balances accuracy and computational efficiency. However, it has limitations, including potential inefficiency for very high polynomial degrees, challenges with stiff ODEs, and sensitivity to the choices of polynomial degree and step size. While MCPI excels for many ODE problems, careful consideration of these factors is crucial for its successful application

# **Evaluating Algorithms for Implementation**

## 

Given the hardware constraints of deploying on an older chip, it is essential to choose algorithms that balance accuracy and computational efficiency while minimizing reliance on intensive matrix operations.

* **Modified Picard-Chebyshev Iteration (MPCI)**: MPCI offers high accuracy and stability through Chebyshev polynomials and iterative refinement via Picard iteration. However, it relies heavily on matrix multiplications and operations, including matrix inversion, which may make it less suitable for environments without powerful optimizations. Despite this, its theoretical strengths and the potential for customization mean MPCI can still be considered if we can optimize specific operations to fit our hardware constraints.
* **Verner's Method (ODE78)**: ODE78 is based on an adaptive Runge-Kutta method of order 7(8), designed for very high accuracy. While it uses higher-order methods, its computational demands may be challenging for older hardware. However, its adaptive nature could potentially make it efficient if the problem's complexity allows for larger step sizes.
* **Dormand-Prince Method (ODE45)**: ODE45 is an adaptive Runge-Kutta method of order 4(5), which adjusts the step size to balance accuracy and computational load dynamically. This adaptability makes ODE45 efficient and reliable, even on less powerful hardware.
* **Runge-Kutta Methods (RK4 and RK8)**: RK4 is well-known for its balance of accuracy and computational simplicity, making it a robust choice for many applications, including our satellite orbit calculations. RK8 provides higher accuracy but at a higher computational cost. Given our hardware limitations, RK4 is likely the more appropriate choice, but RK8 can be considered if additional accuracy is crucial and computational resources allow.
* **Adams-Bashforth-Moulton Method (ODE113)**: ODE113, an adaptive solver suitable for both stiff and non-stiff problems, balances accuracy and efficiency. It is less computationally intensive than some higher-order methods but still requires careful consideration of its computational overhead.

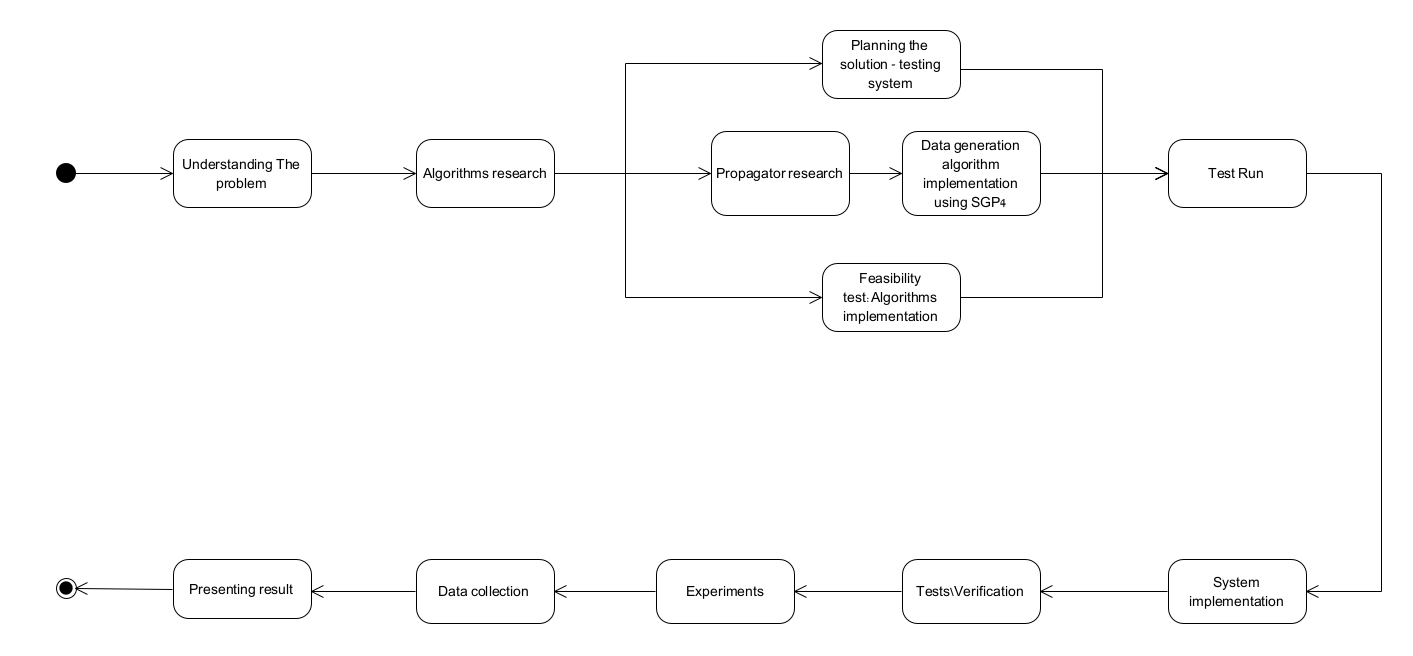
**Conclusion:** Considering our hardware constraints, we believe the best algorithms to implement based on our analysis and intuition, are:

1. **Verner's Method (ODE78)**: For its adaptive step size and high accuracy.
2. **Dormand-Prince Method (ODE45)**: For its adaptive step size and efficient performance.

Despite potential challenges with hardware limitations, the inclusion of **Modified Picard-Chebyshev Iteration (MPCI)** remains advantageous. MPCI uses Chebyshev-Gauss-Lobatto nodes, a feature shared by other advanced algorithms like CATCH used in calculating satellite close approaches. This approach allows for more precise integration over intervals, making it invaluable for detailed trajectory analysis where accuracy is paramount. Hence, we aim to refine and adapt MPCI to meet our hardware's capabilities, leveraging its unique advantages to enhance the robustness and precision of our satellite navigation solutions.

# **The Development Process**

## Workflow



Our project workflow begins with a comprehensive understanding of the problem to establish a solid foundation. We then conduct thorough research on state propagation algorithms, specifically Runge-Kutta methods (RK4 and RK8), ODE solvers (ODE45, ODE78, and ODE113), and the Modified Picard-Chebyshev Iteration, while investigating propagators to understand data propagation and component interactions. Next, we plan the testing system, design test cases, define success metrics, and set up the necessary infrastructure. Implementing data generation algorithms using a suitable framework ensures robust data generation, followed by conducting feasibility tests to verify algorithm functionality and effectiveness. Successful feasibility tests lead to test runs to identify issues or areas for improvement before system implementation. During system implementation, we integrate all components to ensure functionality. Post-implementation, we conduct thorough testing and verification to ensure the system meets all requirements and performs reliably. We then conduct experiments to validate the system’s performance and explore potential optimizations. Throughout the experiments and testing phases, we collect and analyze data to identify any issues. Finally, we compile our findings, insights, and recommendations into a comprehensive report to effectively communicate the project outcomes. Regular weekly review meetings are held to discuss progress, challenges, and next steps, with adjustments made as needed based on ongoing findings and challenges.

## Planning the Assemblies

Given the Constrains of a two-person team and the short timeframe, the planning and workflow will be streamlined to ensure efficient use of time and resources. This plan includes testing all algorithms but focusing on the Modified Picard-Chebyshev Iteration (MCPI), Verner’s Methods (ODE78) and Dormand-Prince Method (ODE45).

**Algorithm Selection and Implementation:**

* **Responsibilities:** Both team members will collaborate on the implementation of all six algorithms (Runge-Kutta 4th Order – RK4, Runge-Kutta 8th Order – RK8, Dormand-Prince Method – ODE45, Verner’s Method – ODE78, Adams-Bashforth-Moulton Solver – ODE113 and the Modified Picard-Chebyshev Iteration – MPCI in C and C++. One member will focus on coding and testing the algorithms, while the other will handle data integration, validation, and documentation.
* **Initial Setup:** Define the scope of testing for each algorithm, including specific criteria for evaluating accuracy and computational efficiency.

**Simulation Environment Setup:**

* **Development:** Create a comprehensive simulation environment in C and C++ that includes the implementation of all six algorithms.
* **Integration:** Incorporate existing data on satellite positions and space debris into the simulation setup.
* **Validation:** Perform preliminary checks to ensure that the simulation environment and data are correctly set up.

**Testing and Evaluation:**

* **Scenario Design:** Develop a set of test scenarios to evaluate the performance of all six algorithms in various conditions.
* **Execution:** Carry out tests in a small CubeSat at Braude College to compare the accuracy and efficiency of each algorithm.

**Iterative Analysis and Refinement:**

* **Testing Cycles:** Conduct iterative cycles of testing, focusing on refining the MCPI,ODE45 and ODE78 algorithms based on the initial findings.
* **Meetings:** Hold regular meetings to review progress, address issues, and adjust testing strategies.

# **Testing**

## Objective of the Testing

The primary objective of the testing process was to evaluate the accuracy and computational efficiency of various numerical algorithms in predicting satellite positions using **the equation of motion with only gravitational forces**. The algorithms were tested against the SGP4 model, a widely accepted standard in orbital prediction, and their results were compared in terms of both position differences and execution times. The algorithms included in the testing are:

* Runge-Kutta 4th Order (**RK4**)
* Runge-Kutta 8th Order (**RK8**)
* Dormand-Price Method (**ODE45**)
* Verner’s Method (**ODE78**)
* Adams-Bashforth-Moulton (**ODE113**)
* Modified Picard-Chebyshev Iteration (**MPCI**)

## Test Setup

### Algorithms Tested

Each of the six algorithms was tested using the following parameters:

* **Initial State Vector**: Derived from Two-Line Element (TLE) data of several satellites.
* **Time Span**: The simulations were run for a period of 30 seconds to observe the evolution of the satellite’s position and velocity over time.
* **Step Size and Tolerances**: Each algorithm was configured with similar step size and tolerance values to test accuracy and computational efficiency.

### Reference Model (SGP4)

The **SGP4** model was used as a reference for comparison. After running the 30-second simulation, the predicted results from each algorithm were compared to the SGP4 model after a simulated 30 seconds period. The SGP4 model provided the ground truth for the satellite’s position and velocity.

### Satellites

The testing involved the use of multiple satellites(Low Earth Orbit and High Elliptical Orbit), such as:

* 0 VANGUARD 2 (LOW)
* 0 VANGUARD 3 (LOW)
* 0 EXPLORER 7 (LOW)
* 0 TIROS 1 (LOW)
* 0 TRANSIT 2A (LOW)
* 0 VELA 1 (HIGH)

Each satellite's TLE data was used to initialize the position and velocity vectors for the algorithms. The results of each test were individually analyzed.

## Testing Procedure

The testing procedure involved the following steps:

### Algorithm Execution

Each algorithm was run independently for each satellite's TLE data. The time span for the simulation was set to 30 seconds. For each time step, the algorithms calculated the satellite’s:

* Position (X, Y, Z)
* Velocity (X, Y, Z)

The results were printed for the last second of the simulation (t = 29 to 30 seconds), allowing a closer look at the satellite’s final predicted state.

### Position and Velocity Comparison with SGP4

After each algorithm completed the 30-second simulation, the final predicted position and velocity were compared with the results obtained from the SGP4 model. The key performance metrics were:

* **Position Difference**: The absolute difference in kilometers between the algorithm’s predicted final position and SGP4’s position.
* **Velocity Difference**: The difference between the predicted and actual velocities, measured in kilometers per second.

### Execution Time Measurement

In addition to accuracy, the execution time for each algorithm was measured. The goal was to determine the computational efficiency of each method, making the results relevant for real-time or computationally constrained systems.

## Results Recording

### Console Output

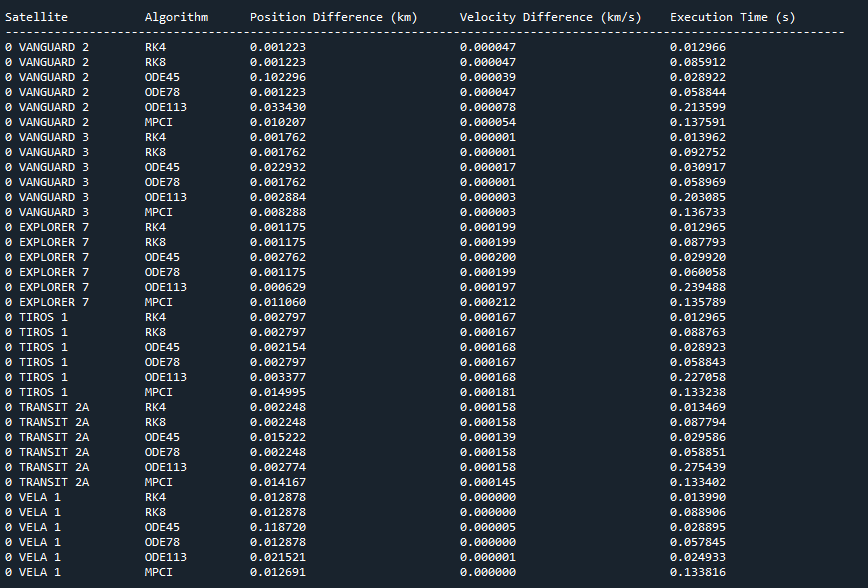
For each algorithm and satellite, the testing process printed the detailed results to the console, including:

* The final position and velocity for the last second.
* The position and velocity differences compared to the SGP4 model.
* The execution time of each algorithm.

### File Logging

In addition to console output, the results for each algorithm were saved to a separate text file. Each file contains:

* The algorithm name.
* The detailed position and velocity values for the last second.
* The calculated position and velocity differences compared to the SGP4 model.
* The execution time for the algorithm.



## Visualization

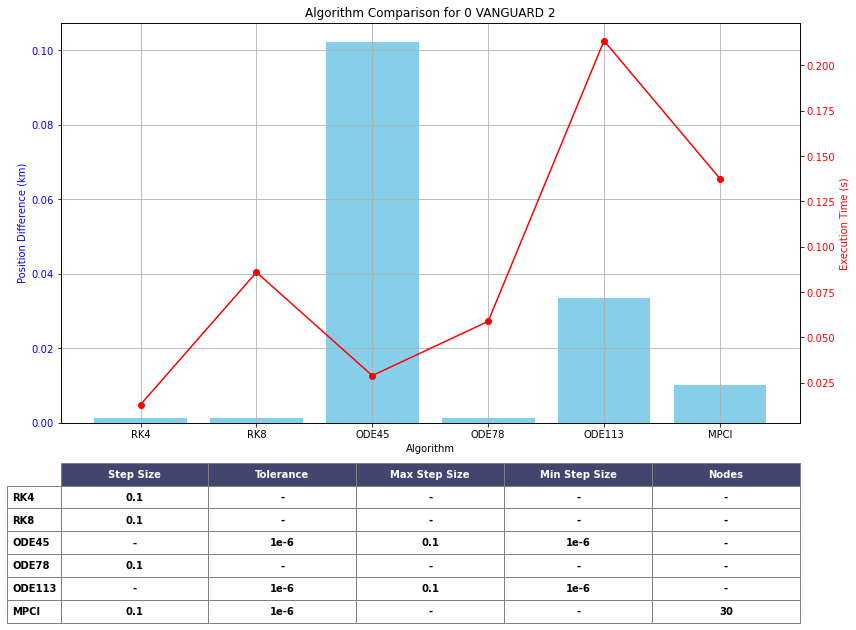
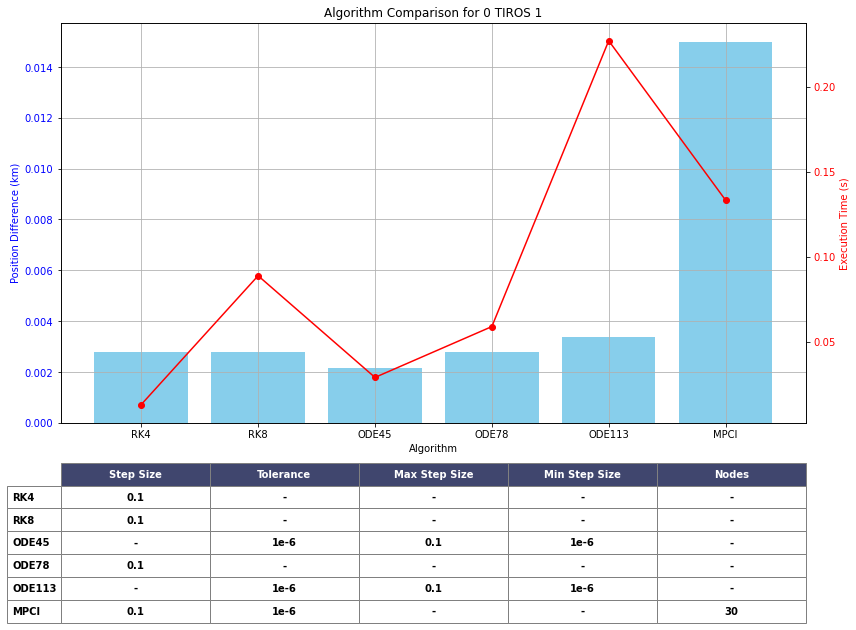
## Individual Satellite Results Plot

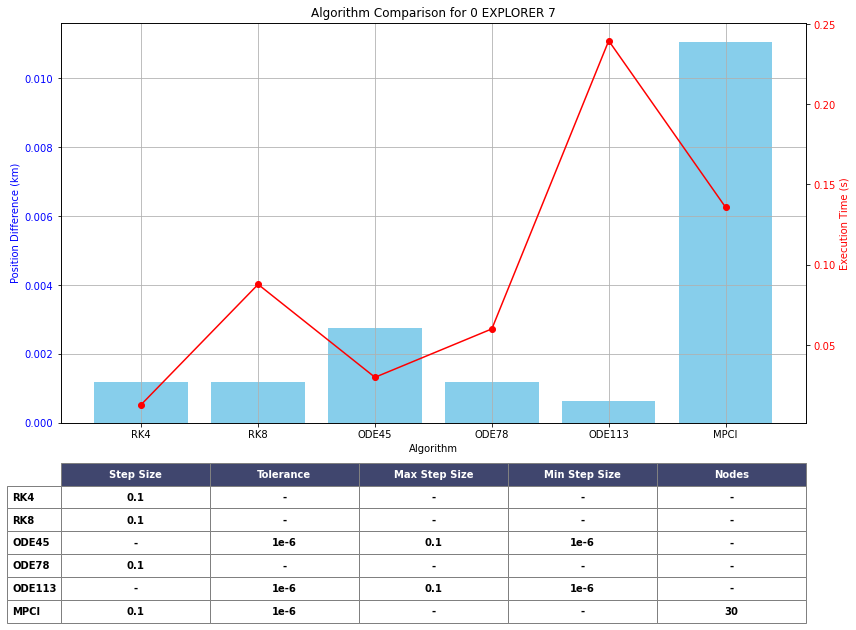
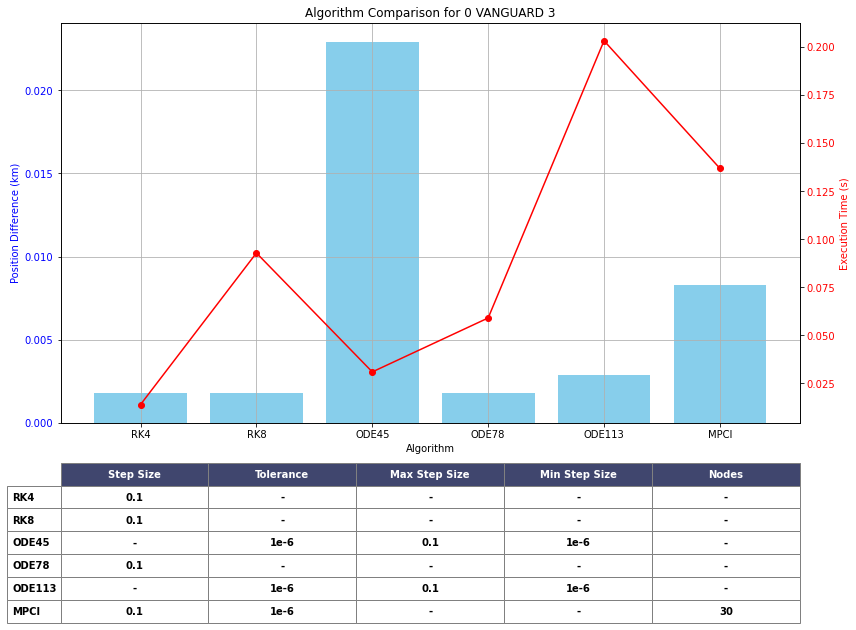
For each satellite, a detailed graph was generated to visualize the performance of the algorithms. These graphs show how each algorithm predicted the satellite’s position over time(execution time), allowing for a direct comparison between different methods.

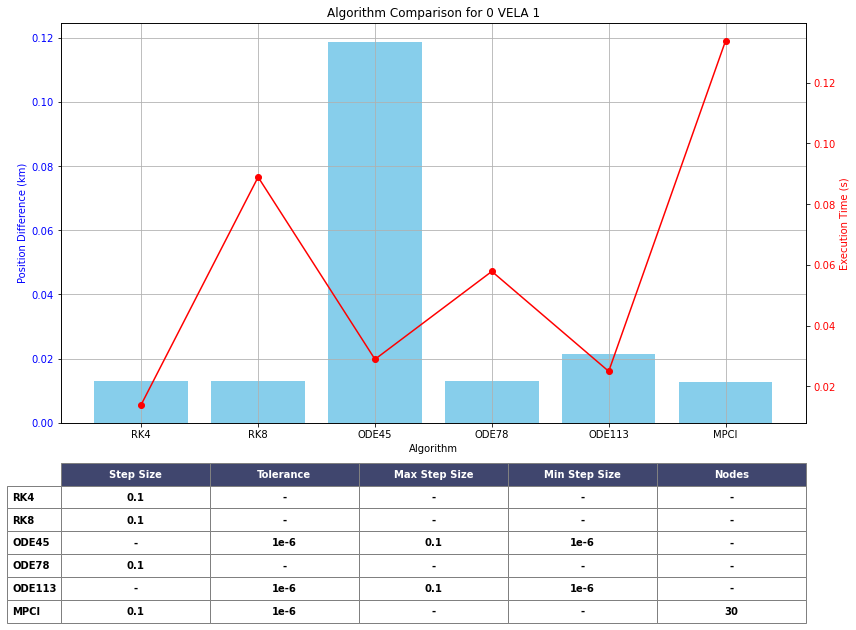
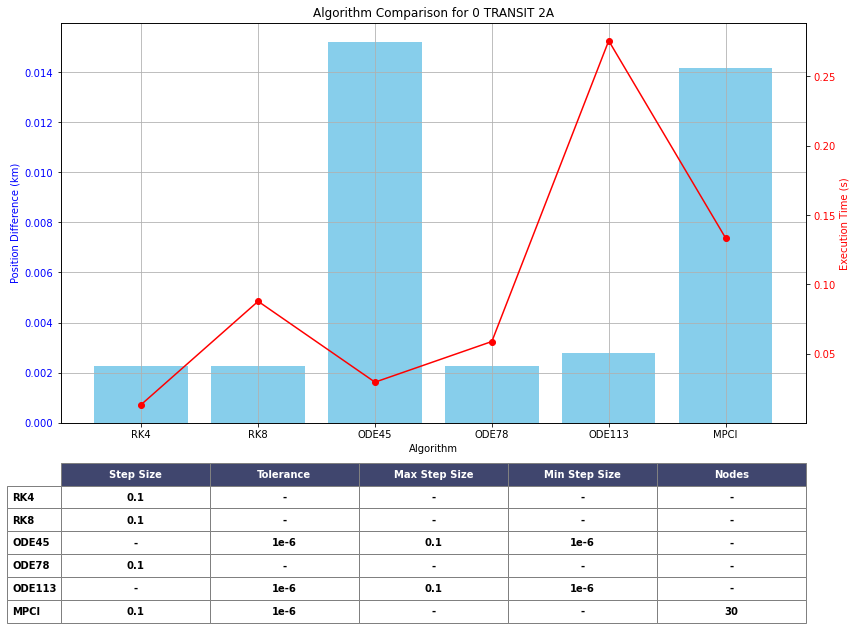
Each satellite’s graph includes:

* **Multiple Algorithms**: The predicted positions from each algorithm (RK4, RK8, ODE45, ODE78, ODE113, MPCI) were plotted to highlight their differences over time.
* **Position Accuracy**: The accuracy of the position predictions is visually displayed as the gap between each algorithm’s result and the reference SGP4 model.
* **Execution Time Consideration**: In addition to position accuracy, these graphs help illustrate the trade-offs between faster algorithms (e.g., RK4) and more precise but slower ones (e.g., ODE113).

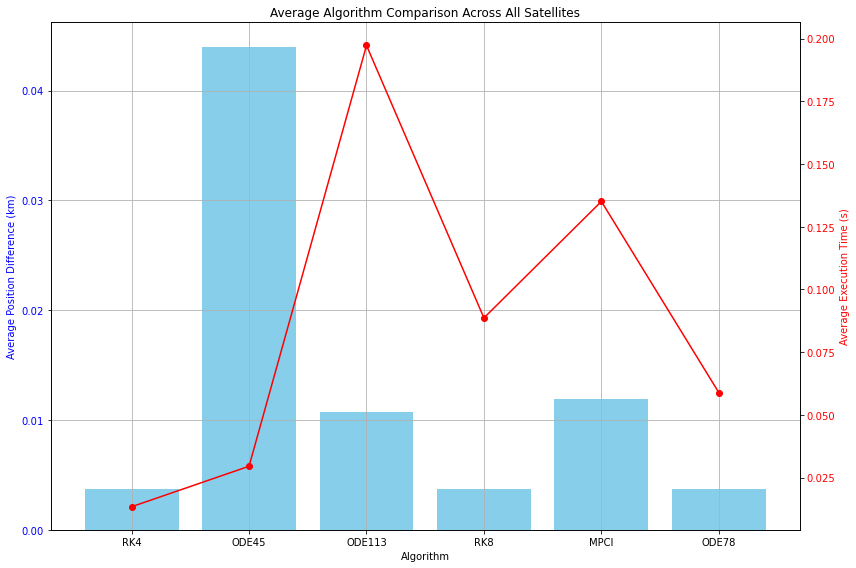
These graphs provide a visual comparison, making it easy to observe how the accuracy of each algorithm evolves during the 30-second simulation.







### Average Results Plot

After running the simulations for all satellites, the average position difference and execution time for each algorithm were calculated.

This plot provided insights into which algorithms achieved the best balance between accuracy and computational speed across all satellites.

## Summary of Testing

The testing provided valuable insights into the trade-offs between accuracy and computational efficiency for each algorithm. The comparison against the SGP4 model allowed us to objectively assess each algorithm's precision in predicting satellite positions, while the execution time measurements helped to gauge their practicality in real-time scenarios. The results were captured in both console output and text files for further analysis.

## Key insights

Key insights:

* **RK4** and **RK8** show the smallest average position differences, indicating the highest accuracy among the algorithms tested. RK4, in particular, has almost no position deviation, while RK8 also performs exceptionally well.
* **ODE45**, on the other hand, exhibits the largest position difference, highlighting its relatively lower accuracy for satellite propagation tasks in this test.
* **MPCI** and **ODE113** offer intermediate accuracy but are not as accurate as RK4 or RK8.
* Regarding execution time, **ODE45** and **MPCI** have the longest execution times, whereas **RK8** and **ODE78** are among the fastest.

# **Conclusion**

From the analysis, **RK4** emerges as the most balanced algorithm, combining high accuracy with moderate execution time, making it a strong candidate for real-time applications where precision is critical. **ODE45**, despite its longer execution time, underperforms in terms of accuracy, making it less suitable for applications demanding precise satellite positioning. **RK8** also shows impressive accuracy with faster execution time, suggesting its effectiveness for scenarios where both speed and precision are required. **MPCI** and **ODE113** offer a trade-off between accuracy and computational cost but fall short compared to RK4 and RK8. Lastly, **ODE78** provides a good balance with a fast execution time and acceptable accuracy, potentially making it suitable for less demanding satellite tracking applications.

# **References**

1. Kessler, D. J., & Cour-Palais, B. G. (1978). Collision frequency of artificial satellites: The creation of a debris belt.

<https://www.researchgate.net/publication/23888776_Collision_frequency_of_artificial_satellites_-_The_creation_of_a_debris_belt>

1. Liou, J. C., & Johnson, N. L. (2009). A sensitivity study of the effectiveness of active debris removal in LEO.

<https://www.researchgate.net/publication/252313802_A_sensitivity_study_of_the_effectiveness_of_active_debris_removal_in_LEO>

1. Runge, C. (1895). Über die numerische Auflösung von Differentialgleichungen. Mathematische Annalen.

<https://www.researchgate.net/publication/230873346_Uber_die_numerische_Auflosung_von_Differentialgleichungen>

1. Kutta, W. (1901). Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. Zeitschrift für Mathematik und Physik.

<https://www.researchgate.net/publication/200175415_Beitrag_zur_Naherungsweisen_Integration_totaler_Differentialgleichungen>

1. Dormand, J. R., & Prince, P. J. (1980). A family of embedded Runge-Kutta formulae. Journal of Computational and Applied Mathematics.

<https://www.researchgate.net/publication/222465749_A_Family_of_Embedded_Runge-Kutta_Formulae>

1. Fehlberg, E. (1968). Classical Fifth-, Sixth-, Seventh-, and Eighth-Order Runge-Kutta Formulas with Stepsize Control.

<https://www.researchgate.net/publication/24331587_Classical_fifth-_sixth-_seventh-_and_eighth-order_Runge-Kutta_formulas_with_stepsize_control>

1. Shampine, L. F., & Gordon, M. K. (1975). Computer Solution of Ordinary Differential Equations: The Initial Value Problem. W.H. Freeman.

<https://books.google.co.il/books/about/Computer_Solution_of_Ordinary_Differenti.html?id=3TFPAQAAIAAJ&redir_esc=y>

1. Woodland, R. A., & Junkins, J. L. (2007). Modified Picard-Chebyshev Iteration Methods for Solving Differential Equations. Journal of Computational Physics.

<https://www.researchgate.net/publication/330092401_Nonlinear_Differential_Equation_Solvers_via_Adaptive_Picard-Chebyshev_Iteration_Applications_in_Astrodynamics>

1. Butcher, J. C. (2003). Numerical Methods for Ordinary Differential Equations. John Wiley & Sons.

<https://www.google.com.lb/books/edition/Numerical_Methods_for_Ordinary_Different/opd2NkBmMxsC?hl=ar&gbpv=1&dq=Numerical+Methods+for+Ordinary+Differential+Equations.+John+Wiley+%26+Sons.&printsec=frontcover>

1. Vallado, D. A. (2001). Fundamentals of Astrodynamics and Applications. Springer.

<https://www.google.com.lb/books/edition/Fundamentals_of_Astrodynamics_and_Applic/PJLlWzMBKjkC?hl=ar&gbpv=1&dq=Fundamentals+of+Astrodynamics+and+Applications.+Springer&printsec=frontcover>

1. Chobotov, V. A. (2002). Orbital Mechanics. AIAA Education Series.

<https://www.google.com.lb/books/edition/Orbital_Mechanics/SuPQmbqyrFAC?hl=ar&gbpv=1&dq=Orbital+Mechanics.+AIAA+Education+Series.&printsec=frontcover>

1. Verner, J. L. (1988). A family of explicit Runge-Kutta methods for the numerical solution of ordinary differential equations ACM Transactions on Mathematical Software

<https://dl.acm.org/doi/10.1145/285861.285863>

1. List of Runge–Kutta methods and Coefficient table

<https://en.wikipedia.org/w/index.php?title=List_of_Runge%E2%80%93Kutta_methods&oldid=896594269>

1. Dormand–Prince method Coefficient table

<https://en.wikipedia.org/wiki/Dormand%E2%80%93Prince_method>

1. Verner’s method Coefficient table

<https://rotordynamics.wordpress.com/2014/05/13/the-method-of-verner/>

1. Linear multistep method (Adam-bushforth-multon Formulas)

<https://en.wikipedia.org/wiki/Linear_multistep_method>