

Capstone Project Phase B

**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** 6](#_Toc188975438)

[**Key words:** 6](#_Toc188975439)

[**2.** **Introduction** 7](#_Toc188975440)

[**3.** **Background and Related Work** 8](#_Toc188975441)

[3.1. Phase One: Evaluation of Key Algorithms for State Propagation: 8](#_Toc188975442)

[3.2. Phase Two: Refinements and New Contributions: 9](#_Toc188975443)

[3.3. Application of Woodland’s Adaptive Picard-Chebyshev Iteration (APCI): 9](#_Toc188975444)

[3.4. Performance Comparison and Hardware Testing 9](#_Toc188975445)

[**4.** **Research, Development, and Expected Achievements** 10](#_Toc188975446)

[**4.1.** **Research and Development Process** 10](#_Toc188975447)

[**4.2.** **Expected Achievements** 12](#_Toc188975448)

[**5.** **Algorithms Analysis** 13](#_Toc188975449)

[**5.1.** **Runge-Kutta 4th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK4)[3][4]** 14](#_Toc188975450)

[5.1.1. Mathematical Formulas and Coefficients Table**[13]** 14](#_Toc188975451)

[5.1.2. Special Case 16](#_Toc188975452)

[5.1.3. Pseudocode: 17](#_Toc188975453)

[5.1.4. Time Complexity: 18](#_Toc188975454)

[5.1.5. Space Complexity: 18](#_Toc188975455)

[5.1.6. Edge Cases and Limitations 18](#_Toc188975456)

[**5.2.** **Runge-Kutta 8th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK8) [4][6]** 19](#_Toc188975457)

[5.2.1. Mathematical Formulas and Coefficients table**[5][13]** 20](#_Toc188975458)

[5.2.2. Special Case 22](#_Toc188975459)

[5.2.3. Pseudocode: 23](#_Toc188975460)

[5.2.4. Time Complexity: 25](#_Toc188975461)

[5.2.5. Space Complexity: 25](#_Toc188975462)

[5.2.6. Edge Cases and Limitations 25](#_Toc188975463)

[**5.3.** **Dormand-Price Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE45)** **[4][14]** 26](#_Toc188975464)

[5.3.1. Mathematical Formulas and Coefficients Table**[5][13]** 27](#_Toc188975465)

[5.3.2. Special Case 29](#_Toc188975466)

[5.3.3. Pseudocode: 30](#_Toc188975467)

[5.3.4. Time Complexity: 32](#_Toc188975468)

[5.3.5. Space Complexity: 32](#_Toc188975469)

[5.3.6. Edge Cases and Limitations: 32](#_Toc188975470)

[**5.4.** **Verner’s Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE78)[6][12]** 33](#_Toc188975471)

[5.4.1. Mathematical Formulas and Coefficients table**[4][15]** 34](#_Toc188975472)

[5.4.2. Special Case 36](#_Toc188975473)

[5.4.3. Pseudocode: 37](#_Toc188975474)

[5.4.4. Time Complexity: 39](#_Toc188975475)

[5.4.5. Space Complexity: 39](#_Toc188975476)

[5.4.6. Edge Cases and Limitations: 39](#_Toc188975477)

[**5.5.** **Adams-Bashforth-Moulton Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE113)[7]** 40](#_Toc188975478)

[5.5.1. Mathematical Formulas and Coefficients**[16][7]** 41](#_Toc188975479)

[5.5.2. Special Case 43](#_Toc188975480)

[5.5.3. Pseudocode: 44](#_Toc188975481)

[5.5.4. Time Complexity: 46](#_Toc188975482)

[5.5.5. Space Complexity: 46](#_Toc188975483)

[5.5.6. Edge Cases and Limitations: 46](#_Toc188975484)

[**5.6.** **Modified Picard-Chebyshev Iteration (MPCI)[17]** 47](#_Toc188975485)

[5.6.1. Mathematical Formulas and Coefficient 47](#_Toc188975486)

[5.6.2. Pseudocode: 48](#_Toc188975487)

[5.6.3. Time Complexity: 51](#_Toc188975488)

[5.6.4. Space Complexity: 51](#_Toc188975489)

[5.6.5. Edge Cases and Limitations: 51](#_Toc188975490)

[**5.7.** **Adaptive Picard-Chebyshev Iteration with Segmentation and Chebyshev Polynomial Approximation (APCI)** **[8]** 53](#_Toc188975491)

[5.7.1. Mathematical Formulas and Coefficient 54](#_Toc188975492)

[5.7.2. Pseudocode**[18]** 57](#_Toc188975493)

[5.7.3. Time Complexity: 59](#_Toc188975494)

[5.7.4. Space Complexity: 59](#_Toc188975495)

[5.7.5. Edge Cases and Limitations 59](#_Toc188975496)

[**6.** **Development Tools** 61](#_Toc188975497)

[**6.1 Programing Languages** 61](#_Toc188975498)

[**6.2. Tools and Frameworks** 62](#_Toc188975499)

[**6.3.** **Hardware Emulation** 62](#_Toc188975500)

[**6.4.** **Output and Visualization** 63](#_Toc188975501)

[**7.** **Problems and Solutions** 64](#_Toc188975502)

[**7.1.** **Cross-Platform Communication** 64](#_Toc188975503)

[**7.2.** **Algorithm Refinement and Rewriting** 64](#_Toc188975504)

[**7.3.** **Integration of Woodland’s APCI** 64](#_Toc188975505)

[**7.4.** **Long Simulation Times** 65](#_Toc188975506)

[**7.5.** **Resource Constraints** 65](#_Toc188975507)

[**7.6.** **Adaptation and Scheduling** 66](#_Toc188975508)

[**8.** **Testing and Results** 67](#_Toc188975509)

[**8.2. Testing Methodology** 67](#_Toc188975510)

[**8.3.** **Results** 68](#_Toc188975511)

[8.3.1. Graphs 74](#_Toc188975512)

[**9.** **Analysis and Conclusion:** 74](#_Toc188975513)

[**9.1. Execution times conclusion:** 75](#_Toc188975514)

[**9.2. Accuracy conclusion:** 76](#_Toc188975515)

[**9.3.** **Final Conclusion:** 77](#_Toc188975516)

[**10.** **User Guide and Maintenance** 78](#_Toc188975517)

[**10.1. Setting Up the Ubuntu Environment on VirtualBox** 78](#_Toc188975518)

[10.1.1. Download Required Software 78](#_Toc188975519)

[10.1.2. Install and Configure Ubuntu on VirtualBox 78](#_Toc188975520)

[10.1.3. Adjust Virtual Machine Settings 78](#_Toc188975521)

[10.1.4. Install CLion IDE on Ubuntu 79](#_Toc188975522)

[**10.2. Testing a New Satellite Using TLE Data** 79](#_Toc188975523)

[10.2.1. Get TLE Data 79](#_Toc188975524)

[10.2.2. Configure the Code 79](#_Toc188975525)

[**10.3.** **Running the APCI Algorithm** 80](#_Toc188975526)

[10.3.1. Clone the APCI Repository 80](#_Toc188975527)

[10.3.2. Build and Run the Algorithm 80](#_Toc188975528)

[10.3.3. Code and functions details 80](#_Toc188975529)

[**10.4.** **Running Other Propagation Algorithms (RK4, RK8, ODE45, ODE78, ODE113)** 82](#_Toc188975530)

[10.4.1. Clone the Algorithms Repository 82](#_Toc188975531)

[10.4.2. Configure and Run the Code 82](#_Toc188975532)

[10.4.3. Output Files 82](#_Toc188975533)

[10.4.4. Code and functions details 83](#_Toc188975534)

[**10.5.** **Running Final Tests and Generating Graphs** 84](#_Toc188975535)

[10.5.1. Prepare the Data 84](#_Toc188975536)

[10.5.2. Organize Result Files 84](#_Toc188975537)

[10.5.3. Run the Simulation 85](#_Toc188975538)

[10.5.4. Code and functions details 85](#_Toc188975539)

[**Appendix : Phase One** 86](#_Toc188975540)

[ Individual Satellite Results Plot 86](#_Toc188975541)

[ Average Results Plot 88](#_Toc188975542)

[ Summary of Testing 89](#_Toc188975543)

[ Key insights 89](#_Toc188975544)

[ 8.5. Conclusion 89](#_Toc188975545)

[ **Evaluating Algorithms for Implementation** 90](#_Toc188975546)

[**11.** **References** 92](#_Toc188975547)

# **Abstract**

The exponential growth in satellite deployments and space debris has elevated the risk of collisions, underscoring the need for precise state propagation algorithms to ensure safe and autonomous satellite navigation. This study builds upon Phase One, which evaluated six key algorithms (RK4, RK8, ODE45, ODE78, ODE113, and MPCI) under fixed step-size constraints and simplified gravitational models. While Phase One provided valuable insights, limitations in adaptability, accuracy over extended durations, and realism of the forces modeled prompted further investigation.

Phase Two introduces significant enhancements, including dynamic step-size recalculation, the integration of Gauss-Lobatto quadrature for improved stability and precision, and the incorporation of additional forces such as atmospheric drag and perturbations into the second equation of motion. The algorithms were implemented and tested under realistic constraints on a virtual machine configured to emulate satellite onboard computer limitations (single-core processor at 500 MHz, <64 MB memory).

Five satellites, representing diverse orbital regimes, were selected for testing, and performance metrics—position differences compared to a baseline and execution times—were analyzed. Results show that RK8 achieves the best balance between accuracy and computational efficiency, while RK4 demonstrates suitability for time-critical tasks with lower precision requirements. The Adaptive Picard-Chebyshev Iteration (APCI), though highly accurate, proved impractical due to its excessive execution time.

This study highlights the trade-offs between accuracy and computational efficiency for various state propagation algorithms, providing critical insights for real-time collision avoidance and autonomous satellite navigation. Future work includes further optimization of APCI and extending testing to real CubeSat environments.

# **Key words:**

Satellite, Space debris, State propagation, Runge-Kutta, ODE solvers, Numerical methods, Modified Picard-Chebyshev Iteration (MPCI), Satellite navigation, Orbit prediction, SGP4 model, Algorithm evaluation, Position and velocity approximation, Autonomous satellites, Real-time computational efficiency, Adaptive Picard-Chebyshev, gauss lobatto.

# **Introduction**

The exponential increase in satellites and the growing accumulation of space debris have drastically heightened the risk of collisions in space, presenting a significant challenge to the safety and sustainability of space missions. To address this challenge, autonomous satellites require precise, reliable algorithms for propagating the positions of orbiting objects over time. These algorithms enable satellites to calculate distances between themselves and surrounding objects, allowing them to navigate and avoid potential collisions effectively. This study builds upon previous research by focusing on the evaluation and refinement of state propagation algorithms for satellite navigation.

In earlier phases, six key algorithms were evaluated 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 (MPCI) [[see Appendix A for detailed results from Phase One]](#AppendixA). The current study introduces additional improvements: the step size h is no longer fixed but varies dynamically, as it is recalculated based on the space between consecutive time points ​. Additionally, all algorithms now include the capability to calculate positions and velocities at specific points, further enhancing their utility for real-time navigation and collision avoidance. Furthermore, Gauss-Lobatto quadrature points are incorporated to enhance both accuracy and efficiency, particularly over long-time durations.

This phase of the study focuses on implementing everything in C++ and applyies Woodland’s variation of the Adaptive Picard-Chebyshev Iteration (APCI) algorithm. A detailed analysis of Woodland's version will be conducted, comparing it against the other algorithms in terms of running times and accuracy. Accuracy comparisons will include error graphs plotted as a function of time, both at regular intervals (e.g., every second) and at time points corresponding to CATCH.

The initial runs will be performed on a PC to ensure functionality. Moreover, a crucial aspect of this phase involves running the algorithms on a satellite computer emulator. We are explored the possibility of using a virtual machine with CPU and memory limitations, another group's card, or leveraging the existing card in the youth satellite. The choice of hardware will depend on availability.

# **Background and Related Work**

## Phase One: Evaluation of Key Algorithms for State Propagation:

In phase one of this study, six key algorithms were evaluated for their ability to propagate the states of orbiting objects:

* **Runge-Kutta methods (RK4 and RK8):** Both RK4 (a fourth-order method) and RK8 (an eighth-order method) are widely used for their balance of accuracy and computational efficiency.
* **Ordinary Differential Equation (ODE) solvers:** These included ODE45 (Dormand-Prince method), ODE78 (Verner’s method), and ODE113 (Adams-Bashforth-Moulton PECE solver). These solvers offered adaptive step-size control and were evaluated for their handling of both stiff and non-stiff problems.
* **Modified Picard-Chebyshev Iteration (MPCI):** This method combined Picard iteration with Chebyshev polynomials, which enhanced convergence without requiring computationally expensive matrix inversions, making it suitable for satellite onboard computers with limited processing power.

Phase one provided valuable insights into the strengths and weaknesses of these algorithms in simulating satellite motion and predicting potential collisions. However, some limitations were identified [[see Appendix A for an extended evaluation of these algorithms]](#AppendixA), particularly related to the fixed step size in long-duration simulations and the need for more efficient calculations over time. This led to the motivation for further refinement, marking the transition to phase two of the study

## Phase Two: Refinements and New Contributions:

Building on the findings from phase one, phase two of this study introduces several critical improvements to address the challenges identified earlier. The following advancements are the focus of this phase:

* **Dynamic Step Size Adjustment:** Unlike the fixed step size used in phase one, phase two implements a dynamically varying step size h, which is recalculated at each iteration based on the space between consecutive time points ​. This adjustment enhances both accuracy and computational efficiency, especially for long-duration propagation.
* **Capability to Calculate Specific Points:** All algorithms have been enhanced with the ability to calculate satellite positions and velocities at specific time points, providing greater utility for real-time navigation and collision avoidance scenarios.
* **Integration of Gauss-Lobatto Quadrature:** Gauss-Lobatto quadrature points are now incorporated to further improve accuracy and efficiency. This method is particularly beneficial for long-duration simulations, ensuring greater precision without significantly increasing computational cost.

## Application of Woodland’s Adaptive Picard-Chebyshev Iteration (APCI):

A key focus of this phase is the implementation of Woodland’s version of the Adaptive Picard-Chebyshev Iteration (APCI) algorithm. Woodland’s APCI offers an adaptive approach to state propagation, allowing for further optimization of computational resources. A detailed analysis of APCI will be conducted, comparing it against the previously tested algorithms (RK4, RK8, ODE45, ODE78, ODE113) in terms of both running time and accuracy.

## Performance Comparison and Hardware Testing

The algorithms will be evaluated using simulation tools implemented in C++, with accuracy comparisons visualized through error graphs plotted as a function of time. These comparisons will include both regular time intervals (e.g., every second) and specific time points corresponding to critical events, such as CATCH points.

In addition to performance analysis, phase two will explore hardware testing. Initial runs will be performed on a PC to validate the functionality of the algorithms. A crucial aspect of this phase involves running the algorithms on a satellite computer emulator to simulate real-time satellite operations. We are considering options such as using a virtual machine with CPU and memory limitations, another group’s hardware, or the existing card on the youth satellite, depending on availability.

# **Research, Development, and Expected Achievements**

This study employs a systematic research and development approach to evaluate and refine state propagation algorithms for autonomous satellite navigation and collision avoidance. Building upon Phase One’s findings[[Appendix A contains information about phase one]](#AppendixA), this phase incorporates key advancements and sets clear objectives for achieving enhanced performance. The following outlines the research process, methods, and anticipated outcomes

## **Research and Development Process**

1. **Algorithm Selection and Evaluation**:

* Six algorithms were chosen based on their relevance to satellite navigation:
  + Runge-Kutta methods (RK4, RK8)
  + ODE solvers (ODE45, ODE78, ODE113)
  + Modified Picard-Chebyshev Iteration (MPCI)
* Initial evaluations identified strengths and weaknesses, including computational efficiency, accuracy over long durations, and adaptability to onboard computer constraints.

1. **Integration of Key Enhancements**:

* **Dynamic Step Size Adjustment**: Implemented to replace fixed step sizes, enhancing accuracy and computational efficiency. The step size dynamically adapts to the system’s complexity.
* **Incorporation of Gauss-Lobatto Quadrature**: Used to define time steps within sections of the time span, improving numerical stability and precision, especially for long-duration simulations.
* **Specific Point Calculations**: All algorithms now allow for position and velocity computations at specified time points, enhancing utility for real-time navigation and collision avoidance.

1. **Implementation of Woodland’s APCI Algorithm**:

* Woodland’s Adaptive Picard-Chebyshev Iteration (APCI) was implemented in C++ for evaluation against existing algorithms. This method employs:
  + Adaptive segmentation based on problem complexity.
  + Chebyshev polynomial degree selection according to error tolerances.
* APCI combines Picard iteration with Chebyshev polynomials for precise, efficient state propagation.

1. **Simulation Environment**:

* All algorithms were implemented in C++ for compatibility with satellite onboard computer constraints.
* Tools for error visualization and performance analysis were developed, generating graphs to compare accuracy and efficiency over time.

1. **Hardware Testing**:

* Algorithms were tested on a PC for initial validation and subsequently run on a satellite computer emulator to simulate real-time operations. Emulation options included:
  + Virtual machines with CPU and memory limitations.
  + Existing hardware such as a group’s card or the youth satellite’s onboard computer.

## **Expected Achievements**

* + 1. **Refinement of Algorithms:**
* Enhanced the six algorithms (RK4, RK8, ODE45, ODE78, ODE113, MPCI) by integrating dynamic step size adjustment and Gauss-Lobatto quadrature.
* Improved their performance for long-duration simulations and real-time calculations.
  + 1. **Performance Analysis of Woodland’s APCI Algorithm**:
* Woodland’s APCI will be evaluated for accuracy, computational efficiency, and applicability to autonomous satellite navigation.
* Comparisons will be drawn with other algorithms to identify the most effective propagation method.
  + 1. **Expanded Simulation Capabilities**:
* Introduced the ability to compute positions and velocities at specific time points, enhancing practical applicability.
* Continued refinement of algorithms for compatibility with onboard satellite hardware.
  + 1. **Hardware Testing and Validation**:
* Ensured the algorithms function efficiently on constrained hardware, such as emulated onboard computers.
* Identified the most suitable algorithm(s) for real-time collision avoidance in space.
  + 1. **Comparative Analysis and Documentation**:
* Generated comprehensive error graphs and performance metrics, visualizing results over regular intervals and at critical events like CATCH points.
* Documented methodologies and findings to serve as a reference for future research and development

# **Algorithms Analysis**

In phase two, we continue to evaluate the performance of the selected algorithms: RK4, RK8, ODE45, ODE78, ODE113, and the Modified Picard-Chebyshev Iteration (MPCI), in addition to these algorithms we implemented the Adaptive Picard-Chebyshev Iteration (APCI), The equations of motion remain the same as in phase one:

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.

## **Runge-Kutta 4th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK4)**[**[3]**](#Refrence3)[**[4]**](#Refrence4)

**Purpose:** **RK4 (Runge-Kutta 4th Order)** method is a widely-used technique for solving ordinary differential equations (ODEs), providing a balance between accuracy and computational efficiency. This analysis extends the RK4 method to handle multiple Gauss-Lobatto quadrature sections of a longer time span, which is crucial for applications like satellite orbit propagation around Earth.

In this approach, Gauss-Lobatto points define time steps for integration within smaller sections of the total time span. Each section is computed one after the other, making the method suitable for handling large-scale problems where the total time span is divided into manageable chunks.

**Overview:** The RK4 method computes the solution of an ODE at discrete time steps using four intermediate slope evaluations (k1, k2, k3, k4) at each step. These slopes are combined in a weighted manner to produce an accurate estimate of the solution at the next time step.

In this version, the total time span ​ (e.g., the time it takes for a satellite to orbit Earth) is divided into multiple **sections**, each of which is integrated using Gauss-Lobatto quadrature points to define the time steps within that section. This allows for efficient integration over a large time span without excessive computational cost or error accumulation.

### 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

Here, tn​ and tn+1​ are the Gauss-Lobatto points for the current and next time steps, respectively. The difference between these time points determines the step size h, which can vary from step to step based on the distribution of the Gauss-Lobatto 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.

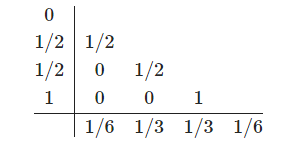
: The current time point.

​: 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:**



### Special Case

In cases where only one Gauss-Lobatto point is provided for a section, the RK4 method cannot proceed as it requires at least two time points to compute a step size h=tn​ – tn+1. In your provided code, this situation is handled by expanding the single Gauss-Lobatto point into a small-time interval. This is done by generating additional time points around the single Gauss-Lobatto point to create a meaningful step size for RK4 integration.

Mathematically, if the single Gauss-Lobatto point is t0​, the expanded time interval is defined as:

tlower = t0 -

tupper =

The new time points are then generated as:

texpanded={tlower, tlower + 0.1,tlower + 0.2,…,tupper}

This ensures that the RK4 method has at least two time points to calculate a meaningful step size h=tn – tn+1 ​, allowing the integration to proceed smoothly.

After expanding the time span, the RK4 method can then proceed with the integration, using the newly generated time points. This ensures that even in cases where only one Gauss-Lobatto point is provided, the algorithm performs meaningful integration and provides useful results.

### Pseudocode:

**Function RK4(odefun, gauss\_lobatto\_points, y0):**

**# Step 1: Handle Single Gauss-Lobatto Point**

**If length of gauss\_lobatto\_points == 1:**

**# Expand the single Gauss-Lobatto point by adding a small interval**

gauss\_lobatto\_points = [gauss\_lobatto\_points[0] - small\_offset, gauss\_lobatto\_points[0] + small\_offset]

**# Step 2: Initialize Arrays for Time and Solution**

tout = gauss\_lobatto\_points

yout = array of zeros with size (length of tout, length of y0)

**# Set the initial condition for the solution**

y = y0

yout[0] = y

**# Step 3: Loop through Gauss-Lobatto Points**

**For i = 1 to length of tout - 1:**

**# Calculate step size**

h = tout[i] - tout[i - 1]

**# Compute the four Runge-Kutta increments**

k1 = h \* odefun(tout[i - 1], y)

k2 = h \* odefun(tout[i - 1] + h / 2, y + k1 / 2)

k3 = h \* odefun(tout[i - 1] + h / 2, y + k2 / 2)

k4 = h \* odefun(tout[i - 1] + h, y + k3)

**# Update the solution y at the next time step**

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

**# Store the updated solution in yout**

yout[i] = y

**# Step 4: Return Time Points and Solution**

**Return tout, yout**

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O (n), where 𝑛 = Number of Gauss-Lobatto points

**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 is n, the time complexity grows linearly with the number of Gauss-Lobatto points in the section. This is because each time step requires the same number of computations for evaluating the slopes and updating the solution.

### Space Complexity:

* **Overall:** O (), where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** The method requires memory to store the time points and the solution values at each time step. This results in a space complexity of O(n × m), where *n* is the number Gauss-Lobatto points, and *m* is the dimension of the solution vector *y*, since *y* is assumed to have a constant dimension (6), the space requirement primarily scales with the number of time steps *n*. Additionally, a fixed amount of space is needed for intermediate calculations, such as the slopes (k1, k2, k3, k4) and the current values of *y* and *t*. However, these constant space requirements do not impact the overall space complexity, which is dominated by the size of the problem.

### Edge Cases and Limitations

If only one Gauss-Lobatto point is provided, the code expands the time span to create a small interval for meaningful integration. Large step sizes can lead to inaccuracies, but dividing the total time span into sections helps reduce this by enabling smaller, more manageable steps. Small step sizes, while improving accuracy, increase computation time, but the use of Gauss-Lobatto points allows for non-uniform time steps that adapt to the complexity of the solution in each section.

**Conclusion:** The RK4 method with Gauss-Lobatto points is a robust and efficient approach for solving ordinary differential equations, particularly in scenarios like satellite motion where non-uniform time steps are beneficial. It offers a good balance between accuracy and computational cost, especially when applied over smaller sections of a larger time span. However, for cases where larger step sizes could introduce inaccuracies, or when dealing with stiff equations.

## **Runge-Kutta 8th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK8)** [**[4]**](#Refrence4)[**[6]**](#Refrence6)

**Purpose:** RK8 (Runge-Kutta 8th Order) method is a higher-order numerical technique for solving ordinary differential equations (ODEs), It offers greater accuracy per step compared to lower-order methods like RK4, making it suitable for applications that require high precision, such as satellite orbit propagation. This analysis extends the RK8 method to handle multiple Gauss-Lobatto quadrature sections of a longer time span, which is crucial for applications like satellite orbit propagation around Earth.

In this approach, Gauss-Lobatto points define time steps for integration within smaller sections of the total time span. Each section is computed one after the other, making the method suitable for handling large-scale problems where the total time span is divided into manageable chunks.

**Overview:** The RK8 method solves ODEs using a set of intermediate stages

(k1, k2, …, k13) derived from the Butcher table coefficients of the Dormand-Prince 8 method (DP8). These stages are combined to achieve an eighth-order accurate solution, providing high precision with a larger number of intermediate evaluations.

This implementation uses Gauss-Lobatto points to define time intervals over which the integration is performed, adapting to the characteristics of the solution. This approach ensures that each step size is suited to the complexity of the dynamics being modeled, making it particularly effective for precise calculations like satellite motion.

### 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

Here, tn​ and tn+1​ are the Gauss-Lobatto points for the current and next time steps, respectively. The difference between these time points determines the step size h, which can vary from step to step based on the distribution of the Gauss-Lobatto 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

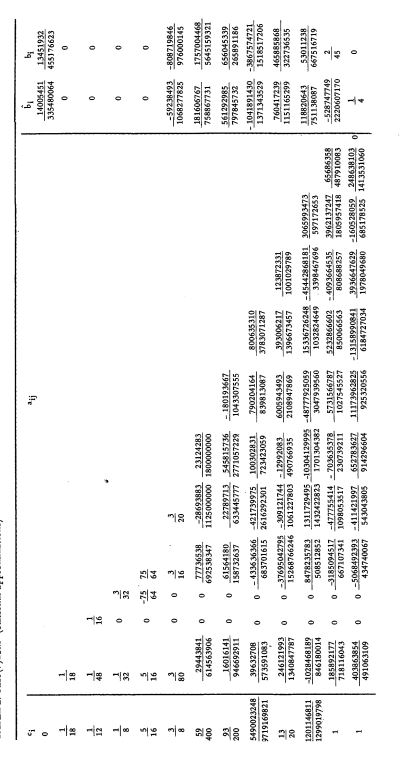
points.

: 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:**

### Special Case

In cases where only one Gauss-Lobatto point is provided for a section, the RK8 method cannot proceed as it requires at least two time points to compute a step size h=tn​ – tn+1. In your provided code, this situation is handled by expanding the single Gauss-Lobatto point into a small-time interval. This is done by generating additional time points around the single Gauss-Lobatto point to create a meaningful step size for RK8 integration.

Mathematically, if the single Gauss-Lobatto point is t0​, the expanded time interval is defined as:

tlower = t0 -

tupper =

The new time points are then generated as:

texpanded={tlower, tlower + 0.1,tlower + 0.2,…,tupper}

This ensures that the RK8 method has at least two time points to calculate a meaningful step size h=tn – tn+1 ​, allowing the integration to proceed smoothly.

After expanding the time span, the RK8 method can then proceed with the integration, using the newly generated time points. This ensures that even in cases where only one Gauss-Lobatto point is provided, the algorithm performs meaningful integration and provides useful results.

### **Pseudocode:**

**Function RK8(f, t\_gauss\_lobatto, Y0):**

**# Step 1: Handle Single Gauss-Lobatto Point**

**If length of t\_gauss\_lobatto == 1:**

**# Expand the single Gauss-Lobatto point into a small interval**

t\_gauss\_lobatto = [t\_gauss\_lobatto[0] - 0.99 \* t\_gauss\_lobatto[0], t\_gauss\_lobatto[0] + small\_offset]

**# Step 2: Initialize Arrays for Time and Solution**

tout = t\_gauss\_lobatto

yout = array of zeros with size (length of tout, length of Y0)

**# Set initial condition for the solution**

y = Y0

yout[0] = y

**# Step 3: Loop through Gauss-Lobatto Points**

**For i = 1 to length of tout - 1:**

**# Calculate step size**

h\_step = tout[i] - tout[i - 1]

**# Initialize k values based on the Butcher table for RK8**

k = array of zeros with size (number of stages from Butcher table, length of Y0)

**# Step 4: Compute k values using Butcher table coefficients**

**For j = 0 to length of Butcher\_table\_DP8['c'] - 1:**

**If j == 0:**

y\_temp = y

**Else:**

y\_temp = y + h\_step \* sum(Butcher\_table\_DP8['a'][j][l] \* k[l] for l in range(0, j))

**# Compute k[j] using function f at the appropriate time step**

k[j] = f(tout[i - 1] + Butcher\_table\_DP8['c'][j] \* h\_step, y\_temp)

**# Step 5: Update the state vector y using Butcher table b coefficients**

y = y + h\_step \* sum(Butcher\_table\_DP8['b'][j] \* k[j] for j in range(0, length of k))

**# Store the updated solution in yout**

yout[i] = y

**# Step 6: Return Time Points and Solution**

**Return tout, yout**

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** Each iteration involves a constant number of operations to compute the 13 intermediate slopes (k1, k2, …, k13) and update the solution. Since the total number of iterations is n, the time complexity grows linearly with the number of Gauss-Lobatto points in the section. Each time step requires the same number of computations for evaluating the slopes and updating the solution.

### Space Complexity:

* **Overall:** O () ,where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** The method requires memory to store the time points and the solution values at each time step. This results in a space complexity of O(n × m), where *n* is the number Gauss-Lobatto points, and *m* is the dimension of the solution vector *y*, since *y* is assumed to have a constant dimension (6), the space requirement primarily scales with the number of time steps *n*. Additionally, a fixed amount of space is needed for intermediate calculations, such as the slopes (k1, k2, …, k13) and the current values of *y* and *t*. However, these constant space requirements do not impact the overall space complexity, which is dominated by the size of the problem.

### Edge Cases and Limitations

When only one Gauss-Lobatto point is provided, the code expands the time span into a small interval to enable meaningful integration. Large step sizes can reduce accuracy, but dividing the problem into sections with smaller intervals helps maintain precision. Conversely, small step sizes increase accuracy but also computational time, while Gauss-Lobatto points enable non-uniform time steps that adapt to the dynamics of the solution. However, RK8 is not ideal for stiff ODEs due to its explicit nature, making implicit methods a better choice for such systems.

**Conclusion:** The RK8 method with Gauss-Lobatto points is a highly accurate and efficient approach for solving ordinary differential equations, particularly in scenarios like satellite motion requiring precise calculations over long durations. It provides superior accuracy compared to lower-order methods, making it ideal for simulations where precision is critical. However, for cases like stiff equations or when adaptive step sizing is needed, alternative methods may be more suitable.

## **Dormand-Price Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE45)** [**[4]**](#Refrence4)[**[14]**](#Refrence14)

**Purpose:** ODE45 method is a popular adaptive step-size integrator for solving ordinary differential equations (ODEs). It uses the Dormand-Prince method (RK45), which provides both fourth-order and fifth-order accurate solutions to estimate errors and adjust the step size accordingly. This method is well-suited for problems where precision and computational efficiency are critical, such as satellite orbit propagation. The analysis extends ODE45 to work with Gauss-Lobatto quadrature points for defining time steps, allowing for non-uniform intervals that adapt to the solution's dynamics. This approach is particularly effective for applications requiring high accuracy over irregular time spans.

In this approach, Gauss-Lobatto points define time steps for integration within smaller sections of the total time span. Each section is computed one after the other, making the method suitable for handling large-scale problems where the total time span is divided into manageable chunks.

**Overview:** ODE45 computes the solution of an ODE by evaluating multiple intermediate stages (k1, k2, …, k7) at each step, using coefficients from the Dormand-Prince Butcher table. These stages yield both fourth-order and fifth-order solutions, which help estimate the local error and adapt the step size to achieve the desired accuracy. The use of **Gauss-Lobatto points** enables the definition of time steps over a specified interval, ensuring that the time steps adapt to the problem's dynamics while maintaining the desired accuracy.

### 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

Here, tn​ and tn+1​ are the Gauss-Lobatto points for the current and next time steps, respectively. The difference between these time points determines the step size h, which can vary from step to step based on the distribution of the Gauss-Lobatto 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.

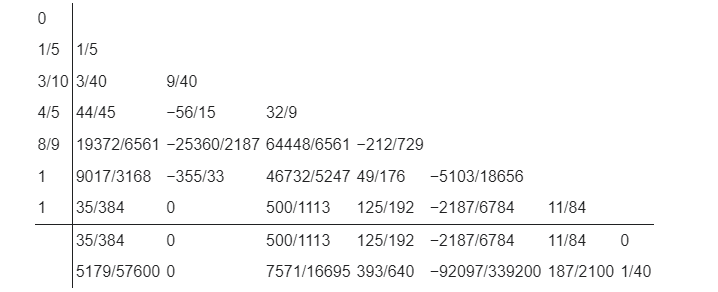
: The current time point.

​: 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:**



### Special Case

In cases where **only one Gauss-Lobatto point** is provided for a section, the ODE45 method cannot proceed as it requires at least two time points to compute a step size h=tn​ – tn+1. In your provided code, this situation is handled by **expanding the single Gauss-Lobatto point** into a small-time interval. This is done by generating additional time points around the single Gauss-Lobatto point to create a meaningful step size for ODE45 integration.

Mathematically, if the single Gauss-Lobatto point is t0​, the expanded time interval is defined as:

tlower = t0 -

tupper =

The new time points are then generated as:

texpanded={tlower, tlower + 0.1,tlower + 0.2,…,tupper}

This ensures that the ODE45 method has at least two time points to calculate a meaningful step size h=tn – tn+1 ​, allowing the integration to proceed smoothly.

After expanding the time span, the ODE45 method can then proceed with the integration, using the newly generated time points. This ensures that even in cases where only one Gauss-Lobatto point is provided, the algorithm performs meaningful integration and provides useful results.

### **Pseudocode:**

**Function rk45\_step(func, t, y, h, rtol, atol)**

**# Step 1: Initialize Butcher Tableau Coefficients**

a = [0, 1/5, 3/10, 4/5, 8/9, 1, 1]

b = matrix of values for intermediate stages

c4 = coefficients for 4th-order solution

c5 = coefficients for 5th-order solution

**# Step 2: Calculate RK45 Stages using Butcher Table Coefficients**

K1 = h \* func(t, y)

K2 = h \* func(t + a[1] \* h, y + b[1][0] \* K1)

K3 = h \* func(t + a[2] \* h, y + b[2][0] \* K1 + b[2][1] \* K2)

K4 = h \* func(t + a[3] \* h, y + b[3][0] \* K1 + b[3][1] \* K2 + b[3][2] \* K3)

K5 = h \* func(t + a[4] \* h, y + b[4][0] \* K1 + b[4][1] \* K2 + b[4][2] \* K3 + b[4][3] \* K4)

K6 = h \* func(t + a[5] \* h, y + b[5][0] \* K1 + b[5][1] \* K2 + b[5][2] \* K3 + b[5][3] \* K4 + b[5][4] \* K5)

K7 = h \* func(t + a[6] \* h, y + b[6][0] \* K1 + b[6][1] \* K2 + b[6][2] \* K3 + b[6][3] \* K4 + b[6][4] \* K5 + b[6][5] \* K6)

**# Step 3: Compute 4th and 5th Order Solutions**

y4 = y + dot product of c4 and [K1, K2, K3, K4, K5, K6, K7]

y5 = y + dot product of c5 and [K1, K2, K3, K4, K5, K6, K7]

**# Step 4: Estimate the Error**

error = norm(y5 - y4) / (atol + rtol \* max(norm(y4), norm(y5)))

**# Step 5: Adjust Step Size Based on Error Estimate**

**If error is not zero:**

h\_new = h \* min(2, max(0.1, 0.9 / error ^ 0.2))

**Else:**

h\_new = h \* 2

**# Step 6: Return the Next Time, Updated Solution, and New Step Size**

**Return t + h, y5, h\_new**

**Function ode45(func, t\_span, y0, rtol, atol)**

**# Step 1: Handle Single Gauss-Lobatto Point**

**If length of t\_span == 1:**

t\_start = t\_span[0] - 0.99 \* t\_span[0]

t\_end = t\_span[0] + small\_offset

t\_span = array of values from t\_start to t\_end with small increments

**# Step 2: Initialize Time and Solution Arrays**

Initialize tout = [t\_span[0]] # List to store time points

Initialize yout = [y0] # List to store state vectors

Set t = t\_span[0] # Current time

Set y = y0 # Initial state vector

**# Step 3: Loop through Gauss-Lobatto Points**

**For each point i in t\_span from 1 to length of t\_span:**

**# Calculate step size**

h = t\_span[i] - t\_span[i-1]

**# Step 4: While the current time is less than the next Gauss-Lobatto point**

**While t < t\_span[i]:**

# Call rk45\_step to compute the next time and state

t\_next, y\_next, h = rk45\_step(func, t, y, h, rtol, atol)

**# Update t and y for the next step**

t = t\_next

y = y\_next

**# Append the current time and state to the result lists**

Append t to tout

Append y to yout

**# Step 5: Convert Output Lists to Arrays**

Convert tout and yout to arrays

**# Step 6: Return Time and Solution in Column Stack Format**

**Return column stack of tout and yout**

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(*n*) where n is the number of adaptive steps taken by the solver based on the initial n (Number of Gauss-Lobatto points)

**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 (), where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** The method requires memory to store the time points and the solution values at each time step. This results in a space complexity of O(n × m), where *n* is the number Gauss-Lobatto points, and *m* is the dimension of the solution vector *y*, since *y* is assumed to have a constant dimension (6), the space requirement primarily scales with the number of time steps *n*. Additionally, a fixed amount of space is needed for intermediate calculations, such as the slopes (k1, k2, …, k7) and the current values of *y* and *t*. However, these constant space requirements do not impact the overall space complexity, which is dominated by the size of the problem.

### Edge Cases and Limitations:

When only one Gauss-Lobatto point is provided, the code expands the time span into a small interval to enable meaningful integration. Large step sizes can reduce accuracy, but dividing the problem into sections with smaller intervals helps maintain precision. Conversely, small step sizes increase accuracy but also computational time, while Gauss-Lobatto points enable non-uniform time steps that adapt to the dynamics of the solution. However, ODE45 is not ideal for stiff ODEs due to its explicit nature, making implicit methods a better choice for such systems.

**Conclusion:** The ODE45 method with Gauss-Lobatto points is an effective numerical integrator for solving ODEs with adaptive time steps. It offers a good trade-off between accuracy and computational efficiency, making it suitable for precise simulations like satellite motion. However, for scenarios involving stiff equations or when very fine error control is required, other methods like implicit solvers or methods with built-in stiffness handling may be more appropriate.

## **Verner’s Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE78)**[**[6]**](#Refrence6)[**[12]**](#Refrence12)

**Purpose:** Verner’s method (ODE78) is a 7th/8th-order Runge-Kutta numerical integrator designed for solving ordinary differential equations (ODEs) with high precision. It is particularly effective for applications requiring accurate solutions over long time spans, such as satellite orbit propagation. This analysis extends Verner’s method to work with Gauss-Lobatto quadrature points, which define time steps over irregular intervals. Using these points, the method divides the total time span into smaller segments, adapting the time steps to the complexity of the system. This approach is especially useful for problems where adaptive step sizes enhance computational efficiency without sacrificing precision.

In this approach, Gauss-Lobatto points define time steps for integration within smaller sections of the total time span. Each section is computed one after the other, making the method suitable for handling large-scale problems where the total time span is divided into manageable chunks.

**Overview:** Verner’s method (ODE78) solves ODEs by evaluating intermediate stages (k1, k2, …, k13) using coefficients from the Butcher tableau. These stages produce a highly accurate solution with an order of 13, minimizing errors in each time step. This implementation employs Gauss-Lobatto points to define time steps, adapting the integration to the system's dynamics over a specified interval. The method’s high order of accuracy and the adaptability provided by the Gauss-Lobatto points make it well-suited for precise and efficient calculations in problems like satellite motion.

### 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

Here, tn​ and tn+1​ are the Gauss-Lobatto points for the current and next time steps, respectively. The difference between these time points determines the step size h, which can vary from step to step based on the distribution of the Gauss-Lobatto 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:**

**c = {**

**1: 0,**

**2: 0.092662,**

**3: 0.1312230361754017604780747799402406525075,**

**…}**

a = {

2: {1: 0.092662},

3: {1: 0.03830746548250284242039554953085778876548, 2: 0.09291557069289891805767923040938286374199},

…}

b = {

1: 0.04625543159712467285354070519930680076661,

2: 0,

3: 0,

…}

bh = {

1: 0.04638504234365210644214797353760063769606,

2: 0,

3: 0,

…}

…

[sfu.ca/~jverner/RKV87.IIa.Efficient.000000011182-240510.FLOAT6040OnWeb](https://www.sfu.ca/~jverner/RKV87.IIa.Efficient.000000011182-240510.FLOAT6040OnWeb)

### Special Case

In cases where only one Gauss-Lobatto point is provided for a section, the ODE78 method cannot proceed as it requires at least two time points to compute a step size h=tn​ – tn+1. In your provided code, this situation is handled by expanding the single Gauss-Lobatto point into a small-time interval. This is done by generating additional time points around the single Gauss-Lobatto point to create a meaningful step size for ODE78 integration.

Mathematically, if the single Gauss-Lobatto point is t0​, the expanded time interval is defined as:

tlower = t0 -

tupper =

The new time points are then generated as:

texpanded={tlower, tlower + 0.1,tlower + 0.2,…,tupper}

This ensures that the ODE78 method has at least two time points to calculate a meaningful step size h=tn – tn+1 ​, allowing the integration to proceed smoothly.

After expanding the time span, the ODE78 method can then proceed with the integration, using the newly generated time points. This ensures that even in cases where only one Gauss-Lobatto point is provided, the algorithm performs meaningful integration and provides useful results.

### **Pseudocode:**

**Function rk78\_step(ode\_func, t, y, h, rtol, atol)**

**# Step 1: Load the coefficients for RK78 (c, a, b, bh from the Butcher tableau)**

c = RK78 coefficients for time nodes

a = RK78 coupling coefficients

b = RK78 weights for 8th-order solution

bh = RK78 weights for 7th-order solution

**# Step 2: Initialize k matrix for storing stage results**

Initialize k as a zero matrix of size (state vector length, 13)

**# Step 3: Calculate the stages for the Runge-Kutta method**

k[1] = h \* ode\_func(t, y) # Compute the first stage

**For each stage i from 2 to 13:**

Initialize a temporary state vector y\_temp as a copy of y

**For each previous stage j from 1 to i-1:**

Update y\_temp with y\_temp += a[i, j] \* k[j]

Compute the next stage: k[i] = h \* ode\_func(t + c[i] \* h, y\_temp)

**# Step 4: Compute the 8th-order solution (y8)**

y8 = y + dot product of b and the k stages

**# Step 5: Compute the 7th-order solution (y7)**

y7 = y + dot product of bh and the k stages

**# Step 6: Estimate the error**

error = norm(y8 - y7) / (atol + rtol \* max(norm(y7), norm(y8)))

**# Step 7: Adjust step size based on the error estimate**

**If error is not zero:**

h\_new = h \* min(2, max(0.1, 0.9 / error^0.2))

**Else:**

h\_new = h \* 2 # Double the step size if error is zero

**# Step 8: Return the next time, updated solution, and new step size**

**Return t + h, y8, h\_new**

**Function ode78(ode\_func, t\_span, y0, rtol, atol)**

**# Step 1: Handle the case of a single time point in t\_span**

**If t\_span has only one time point:**

Expand t\_span to a small range around that point

**# Step 2: Initialize arrays to store time and state values**

Initialize tout as a list containing the first time point in t\_span

Initialize yout as a list containing y0 (the initial state vector)

Set t to the first time point in t\_span

Set y to the initial state vector y0

**# Step 3: Loop through the time points in t\_span**

**For each time point i from 2 to length of t\_span:**

Compute the step size h as the difference between consecutive points in t\_span

**# Step 4: Perform RK78 steps until reaching the next time point**

**While t < t\_span[i]:**

Call rk78\_step(ode\_func, t, y, h, rtol, atol) to compute the next time and state

Update t and y with the results from rk78\_step

**# Step 5: Store the computed time and state**

Append the current time t to tout

Append the current state y to yout

**# Step 6: Convert the time and state lists to arrays**

Convert tout and yout to arrays

**# Step 7: Return the result as a column stack of time and state arrays**

**Return column stack of tout and yout**

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where n is the number of adaptive steps taken by the solver based on the initial n (Number of Gauss-Lobatto points)

**Explanation:** Each iteration involves a constant number of operations to compute the 13 stages (k1, k2, …, k13) and update the solution. Since the total number of iterations is n, the time complexity grows linearly with the number of Gauss-Lobatto points in the section. Each time step requires a consistent set of computations for evaluating the stages and updating the state vector y.

### Space Complexity:

* **Overall:** O (), where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** The method requires memory to store the time points and the solution values at each time step. This results in a space complexity of O(n × m), where *n* is the number Gauss-Lobatto points, and *m* is the dimension of the solution vector *y*, since *y* is assumed to have a constant dimension (6), the space requirement primarily scales with the number of time steps *n*. Additionally, a fixed amount of space is needed for intermediate calculations, such as the slopes (k1, k2, …, k13) and the current values of *y* and *t*. However, these constant space requirements do not impact the overall space complexity, which is dominated by the size of the problem.

### Edge Cases and Limitations:

When only one Gauss-Lobatto point is provided, the code expands the time span into a small interval to enable meaningful integration. Large step sizes can reduce accuracy, but dividing the problem into sections with smaller intervals helps maintain precision. Conversely, small step sizes increase accuracy but also computational time, while Gauss-Lobatto points enable non-uniform time steps that adapt to the dynamics of the solution. Verner’s method (ODE78) is not well-suited for stiff ODEs due to its explicit nature, making implicit methods more suitable for such systems.

**Conclusion:** Verner’s method (ODE78) with Gauss-Lobatto points is a highly accurate and efficient method for solving ODEs, particularly when dealing with long-duration simulations like satellite motion. Its eighth-order accuracy ensures minimal error accumulation across extended time spans, while the use of Gauss-Lobatto points allows the method to adapt to the system's dynamics. However, for scenarios involving stiffness or where adaptive error control beyond eighth-order accuracy is required, alternative methods like implicit solvers may offer better stability.

## **Adams-Bashforth-Moulton Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE113)**[**[7]**](#Refrence7)

**Purpose:** Adams-Bashforth-Moulton (ODE113) method is a multi-step predictor-corrector method designed for solving ordinary differential equations (ODEs) with both efficiency and accuracy. It combines the Adams-Bashforth method for predicting the next time step and the Adams-Moulton method for correcting this prediction, providing a balance between computational speed and precision. This method is particularly suitable for problems requiring long-term integration, such as satellite orbit calculations. The analysis extends ODE113 to work with Gauss-Lobatto quadrature points, which define time steps over irregular intervals, allowing the integration process to adapt to the complexity of the solution within each section.

In this approach, Gauss-Lobatto points define time steps for integration within smaller sections of the total time span. Each section is computed one after the other, making the method suitable for handling large-scale problems where the total time span is divided into manageable chunks.

**Overview:** The ODE113 method begins with fourth-order Runge-Kutta (RK4) steps to initialize the solution at the first few time points, ensuring accurate starting values. After the initial steps, the method transitions to the Adams-Bashforth-Moulton predictor-corrector scheme. The Adams-Bashforth step predicts the next state using previously computed derivatives, while the Adams-Moulton step refines this prediction for greater accuracy. By using Gauss-Lobatto points to define the time steps within a section, the method adapts to varying dynamics, making it effective for scenarios where the total time span is divided into smaller, more manageable segments.

### 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

### Special Case

In cases where only one Gauss-Lobatto point is provided for a section, the ODE113 method cannot proceed as it requires at least two time points to compute a step size h=tn​ – tn+1. In your provided code, this situation is handled by expanding the single Gauss-Lobatto point into a small-time interval. This is done by generating additional time points around the single Gauss-Lobatto point to create a meaningful step size for ODE113 integration.

Mathematically, if the single Gauss-Lobatto point is t0​, the expanded time interval is defined as:

tlower = t0 -

tupper =

The new time points are then generated as:

texpanded={tlower, tlower + 0.1,tlower + 0.2,…,tupper}

This ensures that the ODE113 method has at least two time points to calculate a meaningful step size h=tn – tn+1 ​, allowing the integration to proceed smoothly.

After expanding the time span, the ODE113 method can then proceed with the integration, using the newly generated time points. This ensures that even in cases where only one Gauss-Lobatto point is provided, the algorithm performs meaningful integration and provides useful results.

### **Pseudocode:**

**Function ODE113(f, tspan, y0, options=None, additional\_args=None):**

**# Step 1: Extract or Define Default Solver Options**

rel\_tol = options['RelTol'] if options and 'RelTol' in options else 1e-9

abs\_tol = options['AbsTol'] if options and 'AbsTol' in options else 1e-9

hmax = options['hmax'] if options and 'hmax' in options else 0.5

hmin = options['hmin'] if options and 'hmin' in options else 1e-10

**# Step 2: Identify Time Points for Evaluation**

**If tspan has only two elements:**

Set start\_time = tspan[0], end\_time = tspan[1]

Set t\_eval = None # Allow solver to determine points

**Else:**

Set t\_eval = tspan # Specific points provided

Set start\_time = tspan[0], end\_time = tspan[-1]

**# Step 3: Initialize Variables**

current\_time = start\_time

current\_solution = y0

step\_size = initial\_step\_size (e.g., 0.01)

solution\_times = [current\_time]

solution\_values = [current\_solution]

**# Step 4: Main Integration Loop**

**While current\_time < end\_time:**

**# Adjust step size to prevent overshooting**

**If current\_time + step\_size > end\_time:**

step\_size = end\_time - current\_time

**# Predictor Step (Adams-Bashforth)**

predicted\_value = current\_solution + step\_size \* f(current\_time, current\_solution, additional\_args)

**# Corrector Step (Adams-Moulton)**

**Repeat for a few iterations (e.g., 3 times):**

corrected\_value = current\_solution + step\_size \* (f(current\_time, current\_solution, additional\_args) + f(current\_time + step\_size, predicted\_value, additional\_args)) / 2

predicted\_value = corrected\_value

**# Error Estimation**

error\_estimate = Compute error based on corrected\_value, predicted\_value, rel\_tol, and abs\_tol

**If error\_estimate <= rel\_tol:**

**# Accept the step**

current\_time += step\_size

current\_solution = corrected\_value

Append current\_time to solution\_times

Append current\_solution to solution\_values

**# Adjust Step Size Dynamically**

**If error\_estimate is very small:**

step\_size = Min(step\_size \* 2, hmax)

**Else:**

step\_size = Max(Min(step\_size \* 0.9 \* (rel\_tol / error\_estimate)^0.5, hmax), hmin)

**# Step 5: Interpolate for Specific Points if Necessary**

**If t\_eval is not None:**

Interpolate solution at t\_eval points using cubic interpolation

**Return interpolated\_times and interpolated\_values**

**Else:**

Return solution\_times and solution\_values

**End Function**

### Time Complexity:

* **Per Iteration:** O (1)
* **Total Complexity:** O(n), where n is the number of adaptive steps taken by the solver based on the initial n (Number of Gauss-Lobatto points)

**Explanation:** Each iteration involves a fixed number of operations to perform the predictor-corrector steps using the Adams-Bashforth and Adams-Moulton formulas. As the total number of iterations is n, the time complexity is linear with respect to the number of Gauss-Lobatto points..

### Space Complexity:

* **Overall:** O (), where 𝑛 = Number of Gauss-Lobatto points

**Explanation:** The method requires memory to store the time points and the solution values at each time step. This results in a space complexity of O(n × m), where *n* is the number Gauss-Lobatto points, and *m* is the dimension of the solution vector *y*, since *y* is assumed to have a constant dimension (6), the space requirement primarily scales with the number of time steps *n*. Additionally, a fixed amount of space is needed for intermediate calculations, such as the slopes predicator and corrector calculations and the current values of *y* and *t*. However, these constant space requirements do not impact the overall space complexity, which is dominated by the size of the problem.

### Edge Cases and Limitations:

When only one Gauss-Lobatto point is provided, the code expands the time span into a small interval to enable meaningful integration. Large step sizes can introduce inaccuracies, but dividing the time span into smaller sections helps maintain precision. Conversely, small step sizes increase accuracy but also computational time, while Gauss-Lobatto points enable non-uniform time steps that adapt to the dynamics of the solution. ODE113 is not ideal for stiff ODEs due to its explicit nature, making implicit methods more suitable for such problems.

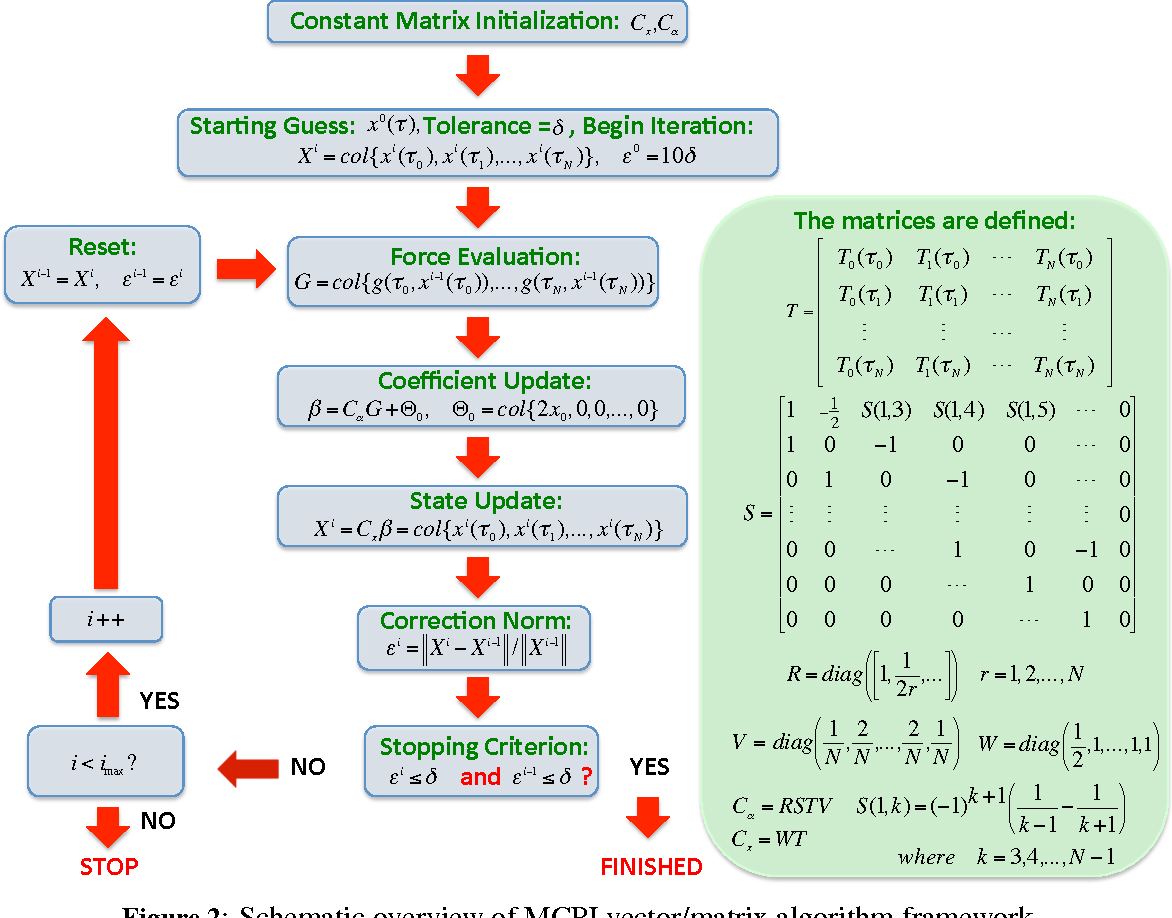
**Conclusion:** The Adams-Bashforth-Moulton method (ODE113) with Gauss-Lobatto points is a robust multi-step integrator for solving ODEs over long durations. Its combination of predictor-corrector steps allows for accurate and efficient calculations, making it ideal for scenarios like satellite motion where precision is critical. While the initial RK4 steps provide accurate starting values, the method's ability to adapt using Gauss-Lobatto points ensures stable time-stepping. However, for scenarios involving stiff equations or where highly adaptive step sizing is needed, alternative methods like implicit solvers may offer better stability.

## **Modified Picard-Chebyshev Iteration (MPCI)**[**[17]**](#Refrence17)

**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

## **Adaptive Picard-Chebyshev Iteration with Segmentation and Chebyshev Polynomial Approximation (APCI)** [**[8]**](#Refrence8)

**Purpose:** The Adaptive Picard-Chebyshev Iteration method is a numerical technique for solving ordinary differential equations (ODEs) that combines Picard iteration with Chebyshev polynomial approximation for high-precision orbital propagation. This method is particularly well-suited for problems like satellite orbit propagation, where accuracy is crucial over long time spans. The analysis extends this approach by incorporating an adaptive scheme, which determines the degree of the Chebyshev polynomial and segmentation based on the problem’s characteristics and tolerance requirements. This adaptive segmentation allows the method to manage large orbital periods effectively by breaking them into smaller, more manageable segments.

**Overview:** The Picard-Chebyshev propagator iteratively solves the ODE describing satellite motion by approximating the solution using Chebyshev polynomials over defined segments. The degree of the Chebyshev polynomial is determined based on the problem’s tolerance, ensuring that the approximation is accurate. The segmentation divides the total time span into smaller intervals, with each segment being solved using the Picard-Chebyshev method. This approach allows the method to adapt to the complexity of the solution, ensuring high accuracy while managing computational costs.

### Mathematical Formulas and Coefficient

**Picard Iteration**

Where:

* is the state vector (e.g., position and velocity).
* is the system of differential equations (the forces acting on the system, such as gravitational forces for satellite motion).

**Chebyshev Polynomial Approximation**

To approximate the function , a series of Chebyshev polynomials ( is used:

Where:

* ​ are the coefficients obtained through a least-squares fit of the function .
* are the Chebyshev polynomials, which form an orthogonal basis over the interval [−1,1].

**Chebyshev Polynomial Basis and Nodes**

The Chebyshev polynomials are sampled at the Chebyshev-Gauss-Lobatto nodes :

This ensures that the approximation is more accurate near the boundaries of the interval.

**Integral Form of Picard Iteration with Chebyshev Polynomials**

Once the approximation is represented as a Chebyshev series, the integral for the Picard iteration becomes:

Where:

* ​ are the Chebyshev coefficients from the previous iteration.
* The integral of the Chebyshev polynomials can be computed analytically.

**Adaptive Segmentation**

The total time interval is divided into smaller segments to ensure accuracy over long periods. Each segment is propagated individually using the Picard-Chebyshev iteration. The degree of the Chebyshev polynomial NNN and the segment length are adaptively chosen based on error criteria, often controlled by the magnitude of the last few Chebyshev coefficients:

Where is a small tolerance value.

**Error Feedback with Quasi-Linearization**

To accelerate convergence, an error feedback term is added to the Picard iteration:

Where:

* is the Jacobian of the system evaluated along the current approximation .
* This error feedback accelerates convergence, especially in the final iterations.

**Second-Order Systems (e.g., Satellite Motion)**

For second-order differential equations (such as those governing satellite motion), the system is written in a cascade form:

**,**

Picard iteration is applied to the velocity update first, and the position update is obtained by integrating the velocity:

this ensures kinematic consistency between the velocity and position.

**Node Adaptation**

The number of nodes N and segment size are adapted based on the nonlinearity of the system over the given segment, ensuring computational efficiency and precision. The coefficients are adjusted accordingly.

### 

### Pseudocode[**[18]**](#Refrence18)

**Adaptive\_Picard\_Chebyshev(r0, v0, t0, tf, dt, deg, tol, soln\_size, Feval, Soln)**

**// Step 1: Determine Degree and Segmentation**

**Call polydegree\_segments to compute polynomial degree (N), number of segments (seg), and time period (Period)**

**// Calculate coefficient array size based on segments and polynomial degree**

**coeff\_size = Calculate\_Coeff\_Size(tf, Period, seg, N)**

**// Step 2: Prepare Propagator**

**Initialize arrays for storing Chebyshev polynomials and time vectors**

**Call prepare\_propagator to setup matrices and segment times based on polynomial degree and segmentation**

**// Step 3: Picard-Chebyshev Propagation**

**Allocate memory for ALPHA and BETA coefficient arrays**

**Initialize total\_seg and segment\_times arrays**

**// Perform Picard iteration for each segment**

**Call picard\_chebyshev\_propagator to iterate and update coefficients for position and velocity**

**// Step 4: Interpolate the Solution**

**Call interpolate to compute the final solution at user-specified times using Chebyshev coefficients**

**Free allocated memory for ALPHA and BETA**

**End Function**

**Key Notes:**

**Functionality**:

**polydegree\_segments**: Determines polynomial degree and number of segments based on the problem's complexity and tolerance.

**prepare\_propagator**: Prepares the propagation environment, including Chebyshev polynomial calculations and segment times.

**picard\_chebyshev\_propagator**: Performs the core propagation using Picard-Chebyshev iterations.

**interpolate**: Uses the computed Chebyshev coefficients to interpolate the final solution.

**Memory Management**: The algorithm dynamically allocates memory for storing Chebyshev coefficients, which are used across different segments for iterative calculations.

**Adaptivity**: The algorithm adapts the segmentation and polynomial degree based on the tolerance specified by the user, ensuring both accuracy and computational efficiency.

**Important Note:**

This pseudocode provides a general overview of the algorithm's structure. The complete implementation includes additional details for managing memory, iterations, and specific calculations, which are too extensive to fully encapsulate here. However, this structure serves as the base for the Adaptive Picard-Chebyshev Numerical Integration process.

### Time Complexity:

* **Per Segment:** O (N)

Each iteration involves computing Chebyshev polynomials and their coefficients over N nodes within a segment. This requires a constant number of operations per node, leading to linear complexity with respect to the number of nodes in the segment.

* Best Case: if N is small and convergence is fast (), O (N) per segment
* Worst Case: If N is large and convergence is slow *,O(I*
* **Total Complexity:** O (), where S is the number of segments is constant, and N is the number of nodes (polynomial degree) in each segment and I for iterations for convergence.

**Explanation**: The total time complexity is influenced by the segmentation of the time span, the polynomial degree for each segment, and the iterations required for convergence. For each segment, the method iterates over N nodes, with a complexity of O*(I*per segment, and propagates the solution across S segments, resulting in a total complexity of O( *)*. The adaptive nature of the algorithm dynamically adjusts the segmentation and polynomial degree based on the system's dynamics, ensuring the method is efficiently applied over manageable segments, balancing computational cost and accuracy.

### Space Complexity:

* **Overall:** O (), The space complexity is proportional to the number of nodes N in each segment and the number of segments S. Memory is required to store the Chebyshev coefficients and intermediate results for position and velocity values in each segment.

**Explanation:** The algorithm stores position and velocity values for each node within a segment, leading to a space complexity of O(N) per segment. Since the solution is propagated across S segments, the overall space complexity becomes O (), The storage requirements are primarily driven by the Chebyshev coefficients and the state variables (position and velocity) at each node. Additional memory for intermediate results, such as the Picard-Chebyshev coefficients, has minimal impact on the overall space complexity.

### Edge Cases and Limitations

When only one time point is provided, the method adjusts by expanding the time interval to allow for meaningful integration. The segmentation scheme helps handle long-duration propagation by breaking down the problem into smaller sections, improving accuracy while controlling computation time. However, the method may struggle with highly stiff systems, where changes occur rapidly over short periods, making adaptive methods with error control potentially more suitable. Furthermore, the degree of the Chebyshev polynomial must be chosen carefully to balance between computational cost and approximation accuracy.

**Conclusion:** The Adaptive Picard-Chebyshev Iteration method with segmentation and Chebyshev polynomial approximation is a highly accurate and efficient approach for solving ODEs, especially in scenarios like satellite motion. The method's adaptive nature, through segmentation and degree selection, ensures precise results even over long orbital periods. By using Chebyshev polynomials, the method achieves high accuracy with fewer iterations compared to traditional methods. However, it is best suited for problems with smooth dynamics, and alternative methods may be more appropriate for stiff systems or when extremely fine error control is required.

# **Development Tools**

The development and analysis of state propagation algorithms in this study relied on a combination of programming languages, computational frameworks, tools, and hardware emulation. These resources were essential for ensuring accurate implementation, efficient simulation, and compatibility with satellite onboard computer constraints.

## **6.1 Programing Languages**

* **C++**:
  + Most algorithms, including Runge-Kutta methods (RK4, RK8), ODE solvers (ODE45, ODE78, ODE113), and the Modified Picard-Chebyshev Iteration (MPCI), were implemented in C++ to meet the requirements of onboard satellite systems.
  + The final implementations were optimized for constrained environments, such as the emulated virtual machine setup.
* **C**:
  + Woodland’s original implementation of the Adaptive Picard-Chebyshev Iteration (APCI) algorithm, written in C, was utilized for initial testing. A future update to C++ is planned to ensure consistency and further optimization.
* **Python**:
  + Python played a critical role in the development and evaluation process:
  + **Algorithm Prototyping**: Most algorithms were initially written and evaluated in Python to test parameters, identify areas for improvement, and guide their subsequent implementation in C++.
  + **SGP4 Implementation**: Python was used to calculate the initial satellite positions, velocities, orbital periods, and other critical parameters such as drag coefficient (​), cross-sectional area (), and mass ().
  + **Data Visualization**: Python’s libraries, such as Matplotlib and Pandas, were employed to process algorithm outputs stored in CSV files and generate detailed error and comparison graphs.

## **6.2. Tools and Frameworks**

* **SGP4 (Standard General Perturbations Model 4)**
  + Used in Python to compute the initial satellite positions and key parameters necessary for simulations. These values were then utilized as inputs for algorithm evaluations.
  + Given the limitations of the virtual machine, SGP4 was executed externally to the VM for efficiency, and its results were incorporated into the simulation environment.
* **ODE78Baseline**
  + Served as a baseline for comparisons. This algorithm calculated positions over specified intervals using small step sizes and an interpolation function to refine data at specific points.
  + Results from the ODE78Baseline were used to evaluate the accuracy of other algorithms by comparing position differences.

## **Hardware Emulation**

* **Virtual Machine**
  + A virtual machine was configured to emulate the environment of a satellite onboard computer, with stringent resource limitations to test the algorithms' feasibility under realistic conditions.
* **Specifications of the virtual machine for Compiling**
  + **Memory**: Limited to less than 64 MB.
  + **Processor**: Single-core with an estimated frequency of 700 MHz
  + **SGP4 Integration**: Due to the computational limitations of the VM, SGP4 calculations were performed externally using Python, and the results were incorporated into the VM for further simulations.
  + **Stability Configuration**: The VM settings were optimized to ensure continued operation despite resource constraints, allowing algorithms to be tested in a near-realistic satellite environment.

## **Output and Visualization**

* **CSV Files:**
  + Results from algorithm implementations, including positional data, velocity vectors, and error metrics, were stored in CSV format for further analysis.
* **Python for Visualization**:
  + Python was used to read the CSV data and generate graphs illustrating position differences and performance comparisons. These graphs provided insights into accuracy, computational efficiency, and algorithmic behavior over time.

# **Problems and Solutions**

## **Cross-Platform Communication**

* **Problem**: The project involved multiple programming languages C, C++, and Python each with distinct roles in the development process. Ensuring smooth communication and integration between these platforms proved to be a significant challenge.
* **Solution**:
* Standardized data formats (e.g., CSV files) were used to exchange results between platforms, enabling seamless communication.
* Python was leveraged as a bridge for prototyping algorithms, visualizing results, and processing data generated in C and C++ implementations.
* Future plans include consolidating all implementations into C++ for consistency and streamlined integration.

## **Algorithm Refinement and Rewriting**

* **Problem**: Enhancing algorithms from Phase One often required extensive modifications, such as rewriting entire algorithms or updating coefficients. This was both time-consuming and technically demanding.
* **Solution**:
* Phase Two improvements were guided by insights from Phase One, allowing for systematic refinement.
* Algorithms were initially implemented and tested in Python to simplify debugging and parameter tuning before translating them into C++.

## **Integration of Woodland’s APCI**

* **Problem**: Woodland’s Adaptive Picard-Chebyshev Iteration (APCI) algorithm consisted of over 40 interconnected files, making it challenging to analyze, modify, and integrate with the existing framework.
* **Solution**:
* A thorough review of Woodland’s implementation was conducted, requiring significant time to understand its structure and functionality.
* Key sections of the algorithm were isolated and tested independently to ensure they performed as expected.
* Planned future updates include rewriting and optimizing the APCI algorithm in C++ for greater compatibility and manageability.

## **Long Simulation Times**

* **Problem**: Running simulations for satellite propagation was time-intensive. For specific parameter sets, such as segmenting the orbital period into smaller points, simulations could take up to 20 hours per satellite. The entire simulation process spanned about a week due to hardware limitations.
* **Solution**:
* A single virtual machine was used for all evaluations to maintain consistent hardware specifications, ensuring reliable and comparable results.
* Time management was optimized by scheduling other tasks, such as documentation and result analysis, during simulation runs.

## **Resource Constraints**

* **Problem**: The virtual machine used to emulate satellite onboard computers had stringent limitations for compiling, including less than 64 MB of memory, a single-core processor running at 500 MHz, and restricted I/O capacity.
* **Solution**:
* Computationally intensive tasks, such as SGP4 calculations, were performed externally in Python, and results were fed into the virtual machine for further simulations.
* The algorithms were carefully optimized to minimize resource usage while maintaining accuracy and efficiency.

## **Adaptation and Scheduling**

* **Problem**: Managing overlapping tasks and adapting to challenges while adhering to the project timeline was complex, especially with long-running simulations and the need for cross-platform collaboration.
* **Solution**:
* A structured schedule was created, allocating specific time blocks for simulations and parallel tasks, such as debugging, documentation, and algorithm refinement.
* Team efforts were coordinated to focus on complementary sections of the project, ensuring progress across multiple fronts simultaneously.

# **Testing and Results**

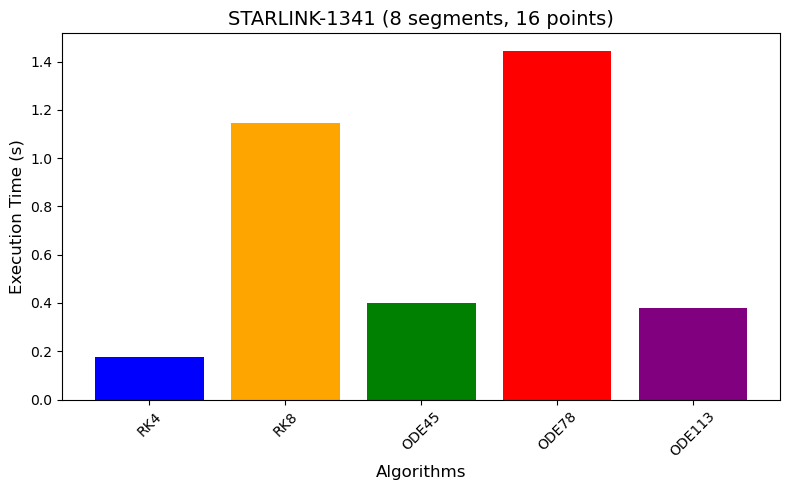
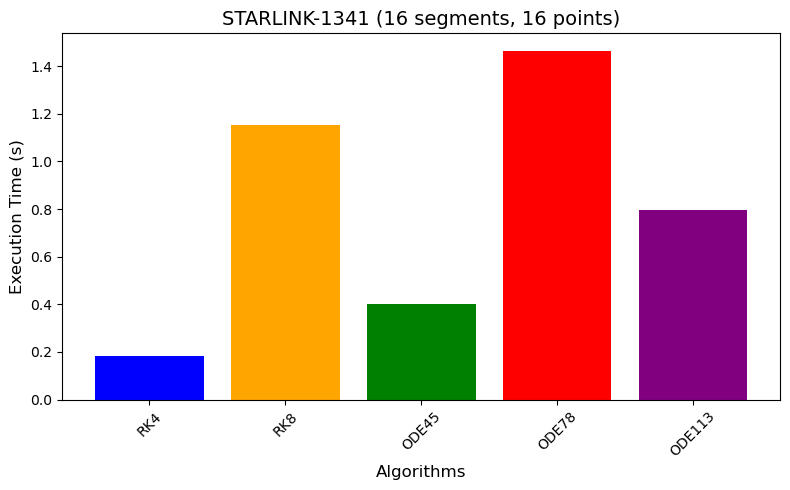
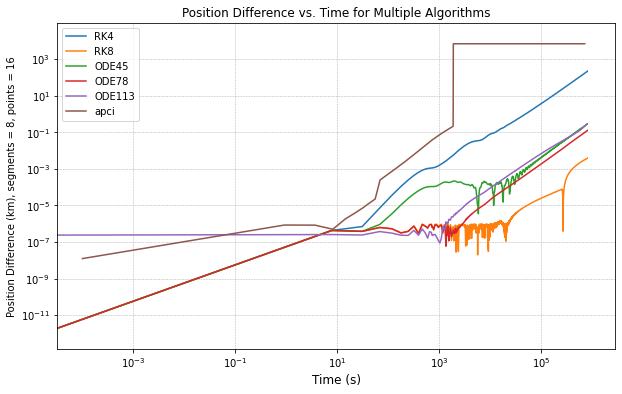
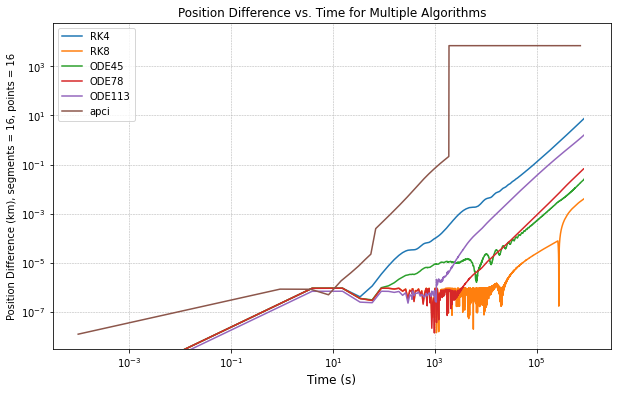
## **8.2. Testing Methodology**

* **Testing Environment**
  + All tests were conducted on a virtual machine (VM) configured to emulate the constraints of satellite onboard computers. The VM specifications included less than 64 MB of memory, a single-core processor running at 500 MHz, and consistent settings across all evaluations to ensure comparability.
* **Equations of Motion**
  + The testing primarily utilized the **second set of equations of motion**, which accounts for gravitational forces as well as corrections for additional forces:
* **Planned Tests for the First Equation of Motion**
* Initially, we planned to test the algorithms using the simpler **first equation of motion**, which accounts only for gravitational forces:
* However, due to time constraints and the long duration of each simulation (up to 20 hours per satellite per configuration), these tests could not be executed in this phase. This limitation highlights the resource-intensive nature of high-fidelity simulations.
* **Simulation Duration**
* Each test simulated satellite motion for approximately 1 million seconds.
* The orbital period of each satellite was divided into segments, with two configurations tested:
  + **8 Segments** and **16 Segments**.
  + Each segment was further divided into **16 Gauss-Lobatto points** for accurate integration.
* **Visualization**
* Python’s Matplotlib library was used to generate detailed graphs showcasing performance metrics such as execution time and algorithm accuracy. The library’s customization features ensured that the graphs were clear, precise, and informative.
* **Satellites Tested**
* Five satellites with varying orbital characteristics were selected:
  + **Low Earth Orbit (LEO)**: STARLINK-1341, IRIDIUM 33 DEB, QIANFAN-4.
  + **High Earth Orbit (HEO)**: SKYNET 4C, ASBM-2.
* These satellites represent a diverse range of orbital regimes, providing insights into algorithm performance across different scenarios.

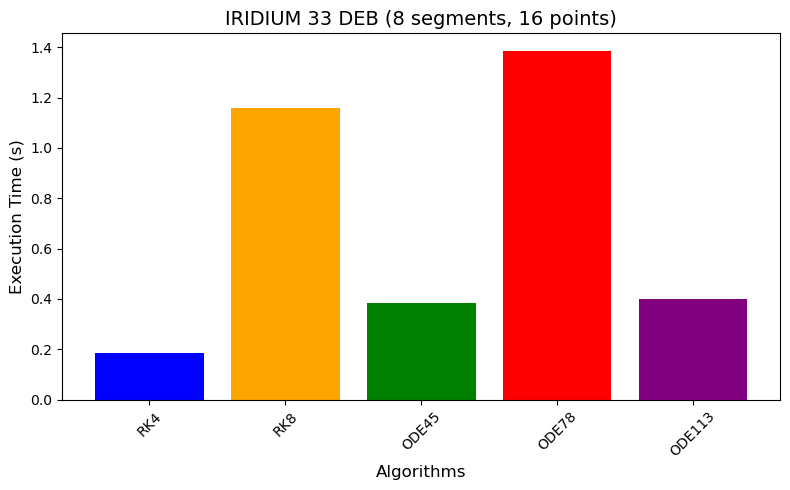
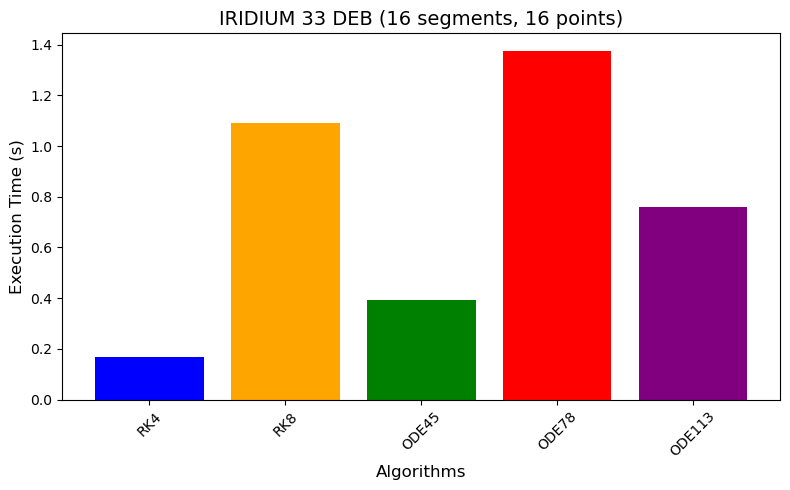
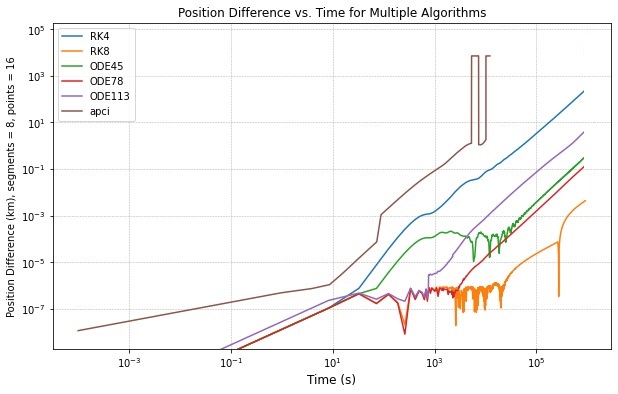
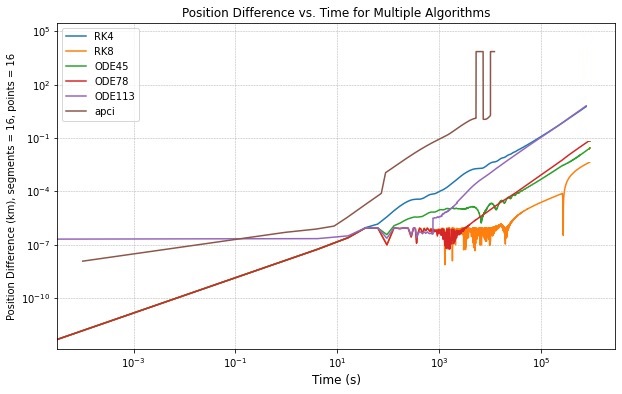
## **Results**

The results are presented in two categories: execution time and accuracy, supported by detailed graphs for each satellite. These graphs provide visual insights into the performance of the algorithms across different orbital regimes, comparing their position differences with the baseline algorithm and their execution times.

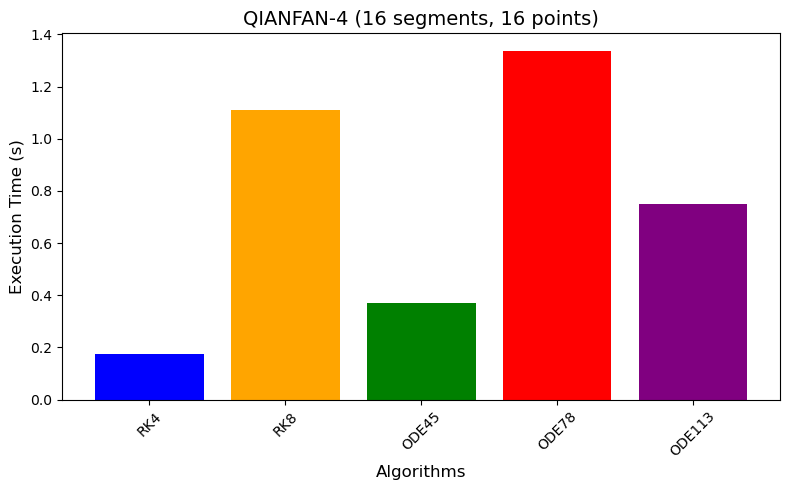
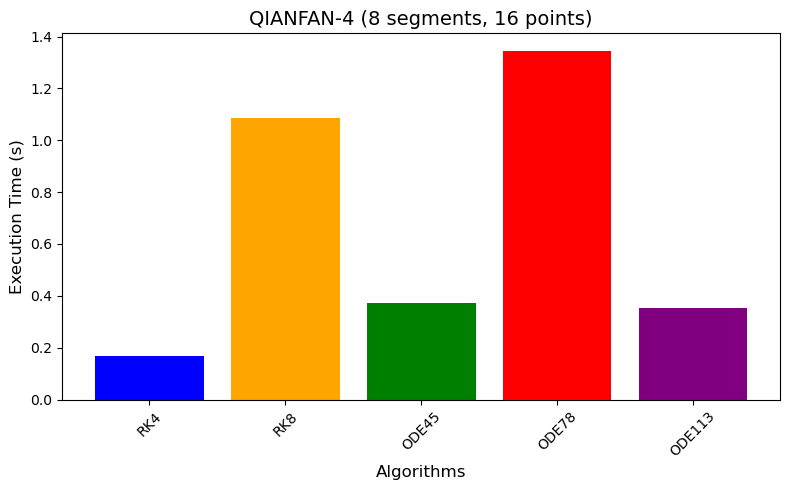
STARLINK-1341:

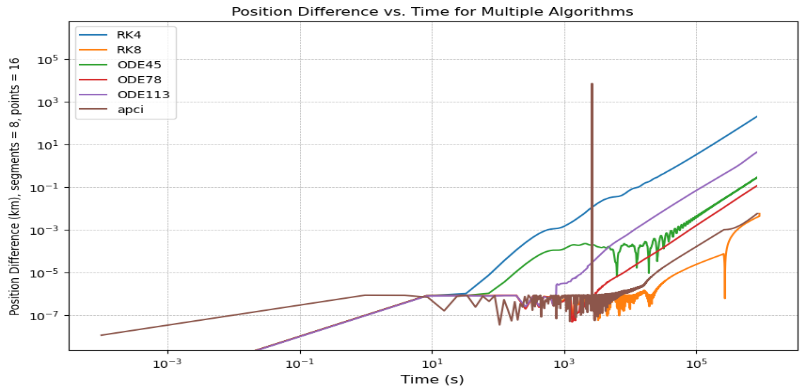
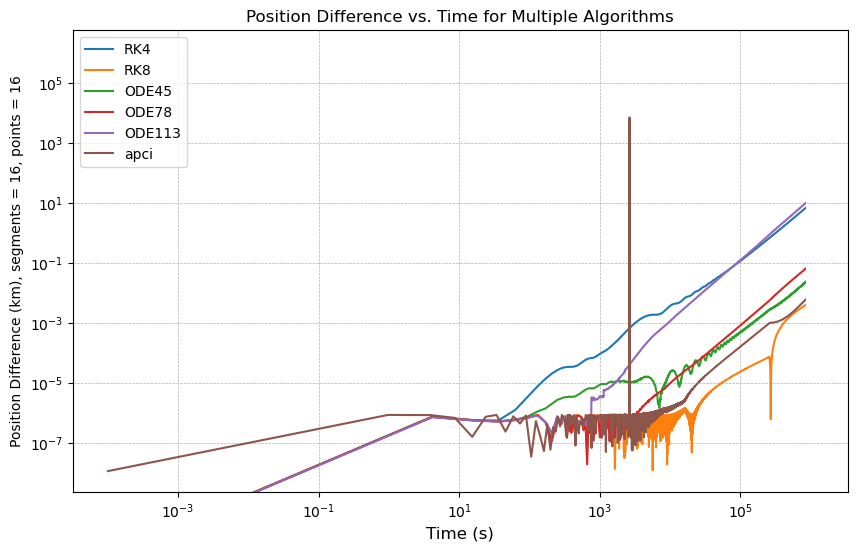


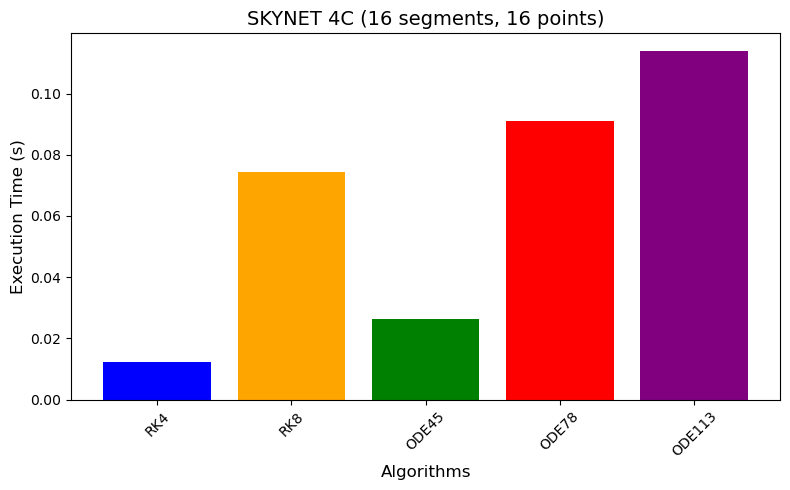
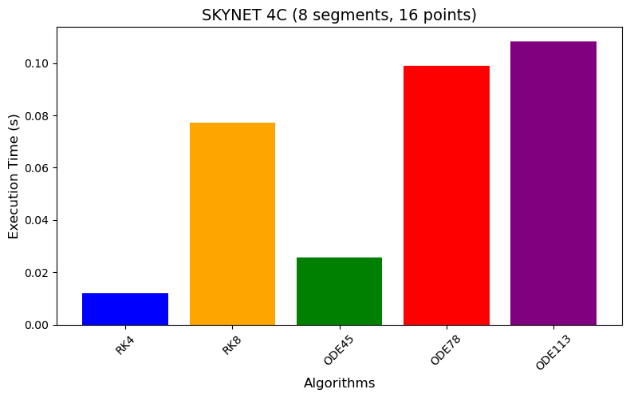
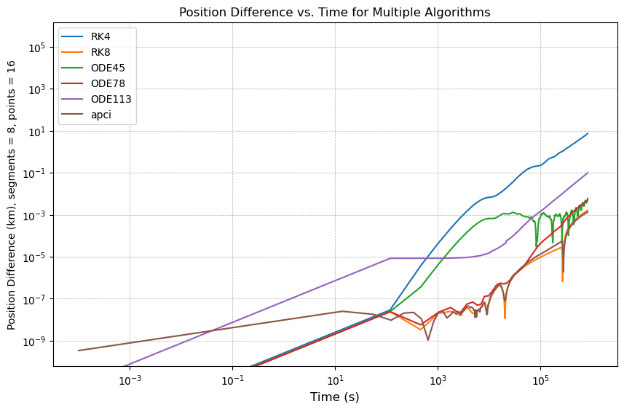
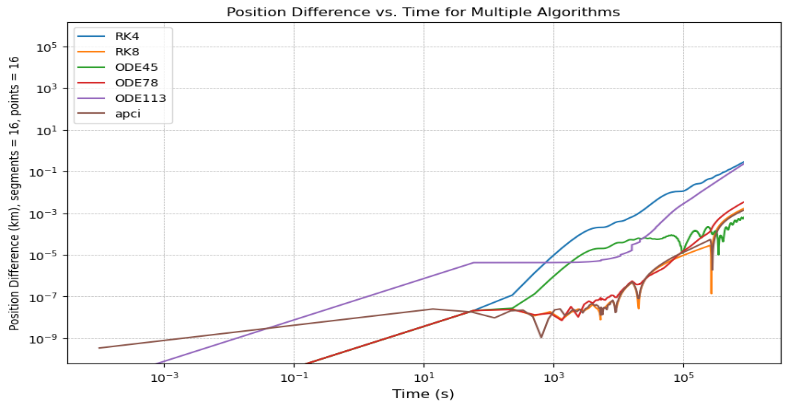
IRIDIUM 33 DEB:

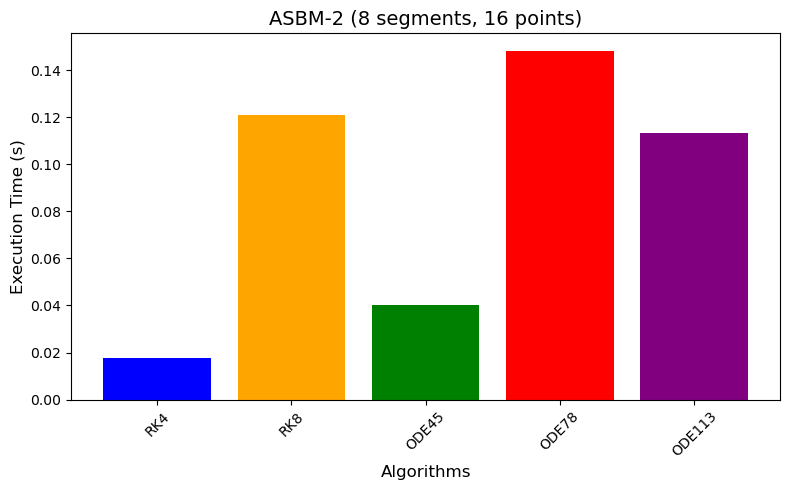
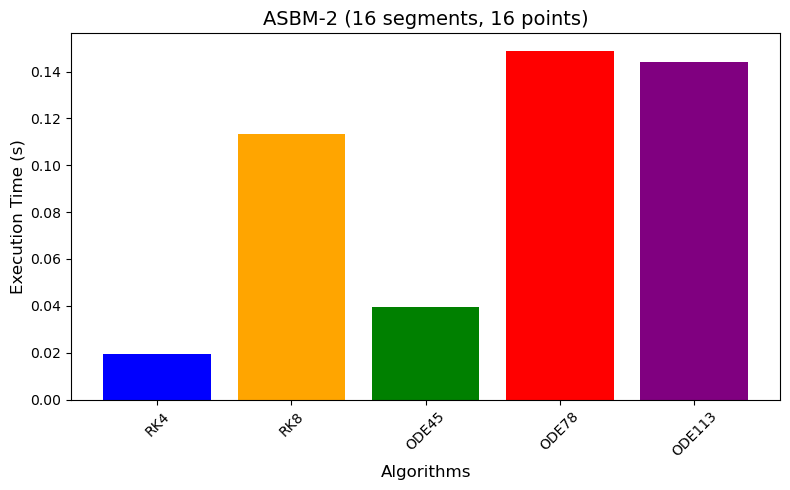
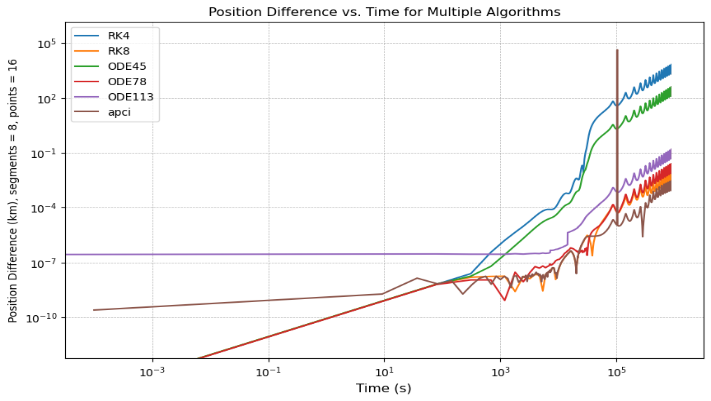
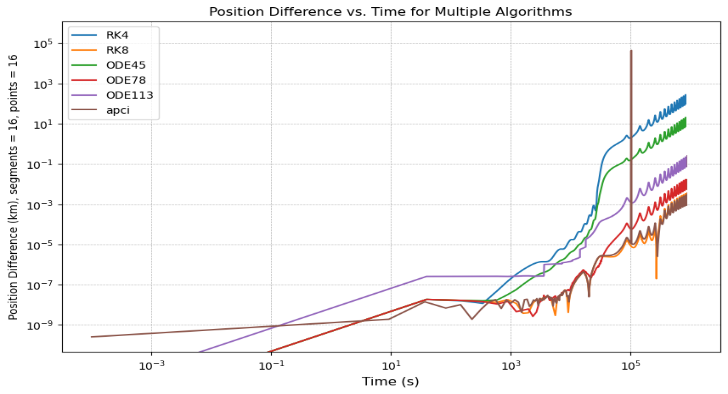


## 

QIANFAN-4:



SKYNET 4C:

ASBM-2:

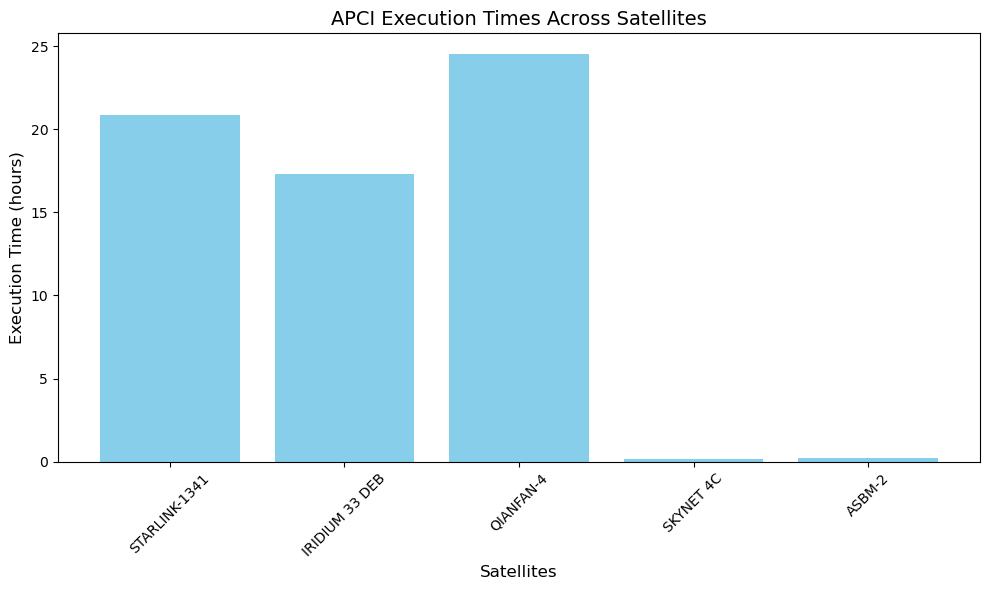
### Graphs

* + - **Position Difference Graphs**  
      These graphs illustrate the accuracy of each algorithm by plotting **position differences as a function of time** against the ODE78Baseline reference. The x-axis represents time (seconds), while the y-axis shows the position differences. They highlight how closely each algorithm's calculated positions align with the baseline over the simulation duration.
    - **Execution Time Graphs**:  
      These graphs compare the **execution times** of the tested algorithms across five satellites. The x-axis lists the algorithms and satellites, while the y-axis shows execution times. They emphasize the computational efficiency of the algorithms within the constraints of the virtual machine.

# **Analysis and Conclusion:**

We will analyze the results in two stages. First, we will draw conclusions based solely on the execution times of the algorithms for each satellite. Next, we will assess the accuracy of the algorithms independently. Finally, we will combine these two analyses to provide a comprehensive conclusion that considers both execution time and accuracy, enabling us to identify the most suitable algorithm for the given problem.

However, we have excluded the **APCI** algorithm from our final conclusion. While in some cases it demonstrated excellent accuracy, its performance time was exceptionally slow for low attitude satellites, requiring up to one day of execution time. This makes it impractical for real-world applications where computational efficiency is a critical factor.



## **9.1. Execution times conclusion:**

**Algorithm Performance Consistency**:

* The execution time ranking is consistent across all satellites, with the algorithms generally performing in the following order (from fastest to slowest): **RK4**, **ODE45**, **RK8**, **ODE113**, and **ODE78**.

**Execution Time Variability**:

* The execution times of the algorithms vary depending on the satellite, but **ODE78** consistently takes the longest time, suggesting it may have a higher computational cost due to its high-order nature.

**RK4** has the shortest execution time across all cases, making it the most computationally efficient algorithm for the given problem.

**Algorithm Scalability**:

* While **RK4** and **ODE45** are faster, their accuracy might be lower compared to higher-order methods like **ODE78** and **ODE113**, which trade off speed for precision.

The execution times for **ODE113** are close to **ODE78**, but with a slight advantage in efficiency.

**Satellite-Specific Trends**:

* There is no significant deviation in algorithm behavior across satellites, indicating the algorithms scale similarly across both LEO (e.g., STARLINK-1341, IRIDIUM 33 DEB) and HEO (e.g., SKYNET 4C, ASBM-2) cases.

**Optimal Algorithm Selection**:

* For time-critical operations, **RK4** and **ODE45** are preferable.

For scenarios where higher precision is critical, **ODE78** and **ODE113** may be better suited despite their longer execution time

## **9.2. Accuracy conclusion:**

**Algorithm Accuracy Hierarchy**:

* **RK8** demonstrates the highest accuracy among the algorithms analyzed, consistently maintaining smaller position differences over time.

**ODE78** follows closely but exhibits slightly larger deviations compared to **RK8**, particularly over extended durations.

**ODE113** provides intermediate accuracy, with deviations increasing steadily as time progresses.

**ODE45** offers moderate accuracy but performs better than **RK4**.

**RK4** shows the largest position differences, making it the least accurate of the algorithms.

**Accuracy over Time**:

* All algorithms experience growing position differences as time progresses due to cumulative numerical errors.

Higher-order methods like **RK8** and **ODE78** perform significantly better over longer durations compared to lower-order methods like **RK4**.

**Satellite Variations**:

* The accuracy trends are consistent across satellites, irrespective of their orbital regimes (LEO or HEO). This consistency highlights the robustness of higher-order methods like **RK8** across different orbital scenarios.

**Overall Insights**:

* **RK8** is the most accurate algorithm for scenarios requiring high precision over both short and long propagation durations.

While **ODE78** is close in performance, **RK8** consistently outperforms it, making it the preferred choice when accuracy is critical.

**RK4**, while efficient in computation, shows significant accuracy limitations, making it suitable only for less precision-critical tasks.

## **Final Conclusion:**

* **RK8** emerges as the most accurate algorithm, consistently maintaining smaller position differences over time. This makes it the best choice for scenarios requiring high precision, especially for long-duration satellite propagations. Additionally, its execution time is reasonable compared to other high-accuracy algorithms, such as ODE78 and ODE113, making it a balanced choice between performance and computational cost.
* **ODE78** follows closely in accuracy but has the longest execution times among the tested algorithms. It is better suited for situations where computational resources are less constrained and accuracy is paramount.
* **RK4**, while the fastest algorithm, shows the largest position differences, making it suitable only for time-critical operations or applications where lower accuracy is acceptable.
* **ODE45** and **ODE113** provide intermediate performance, with ODE45 leaning more toward computational efficiency and ODE113 offering better accuracy. They are viable options for scenarios that require a balance between execution time and precision.

Based on the analysis of both accuracy and execution time, RK8 stands out as the best algorithm among all those tested. It consistently delivers the highest accuracy while maintaining a reasonable execution time, making it the most well-rounded choice. RK8 strikes an optimal balance between precision and computational efficiency, outperforming other algorithms in scenarios where both accuracy and speed are critical. This makes RK8 the preferred algorithm for a wide range of satellite propagation tasks.

# **User Guide and Maintenance**

## **10.1. Setting Up the Ubuntu Environment on VirtualBox**

### 10.1.1. Download Required Software

* Ubuntu ISO: [Download Ubuntu](https://ubuntu.com/download/desktop)
* Oracle VirtualBox: [Download VirtualBox](https://www.virtualbox.org/wiki/Downloads)

### 10.1.2. Install and Configure Ubuntu on VirtualBox

* Install VirtualBox after downloading it.
* Create a new Ubuntu virtual machine in VirtualBox.
* Need help? Follow this video tutorial: [Watch on YouTube](https://www.youtube.com/watch?v=8mns5yqMfZk)

### 10.1.3. Adjust Virtual Machine Settings

* RAM: 2.1 GB

| **Component** | **Total RAM Allocated** |
| --- | --- |
| **Ubuntu OS** | ~1 GB |
| **CLion IDE** | ~1 GB |
| **Compiler Simulation** | 64 MB |
| **Total VM RAM** | **2.5** |

* Processor Cores: 1 Core
* CPU Execution Cap: 40%(depends on the cpu you are using)

**Example Calculation**

| **Component** | **Host CPU Resources** | **Execution Cap** | **Allocated CPU** |
| --- | --- | --- | --- |
| **Ubuntu + CLion** | 3.4 GHz | 35%-40% | ~1.2 GHz |
| **Compiler Simulation** | 3.4 GHz | 10%-15% | ~0.5 GHz |
| **Total VM** | 3.4 GHz | 50% | ~1.7 GHz |

### 10.1.4. Install CLion IDE on Ubuntu

* Open the Ubuntu Terminal and run the following commands:
* sudo apt update
* sudo apt install snapd curl
* sudo snap install clion --classic
* clion

## **10.2. Testing a New Satellite Using TLE Data**

### 10.2.1. Get TLE Data

* Visit [N2YO.COM](https://www.n2yo.com/).
* Search for the satellite (e.g., "ASBM-2").
* Scroll down to the Two Line Element Set (TLE).
* Copy the two TLE lines.

Note: You can use our TLE data TLE file

### 10.2.2. Configure the Code

* Open: Final\_Main\_make\_sgp4\_results.csv.
* Update: the tle\_data\_list variable with the new TLE data.
* Run the code.
* Output: A file named sgp4\_results.csv will be generated.

## **Running the APCI Algorithm**

### 10.3.1. Clone the APCI Repository

* Open **CLion IDE**.
* Clone the repository:  
  🔗 [APCI Repository](https://github.com/shadi26/APCI.git)

### 10.3.2. Build and Run the Algorithm

Open Terminal in the /src folder and run:

* Compile the Matrix Builder:
* make matrix\_builder
* Build the Picard-Chebyshev Matrices:
* ./matrix\_builder
* Compile the Propagator:
* make
* Run the Propagation:
* ./test

Output: final\_positions\_results.csv

Note: save the Execution time for each satellite name in csv file(apci\_algorithm\_execution\_times satelite\_name, Execution Time (hours)) and save it in csvfiles

### 10.3.3. Code and functions details

* **adaptive\_picard\_chebyshev.c**: Handles all operations required for performing the Adaptive Picard-Chebyshev numerical integration method.
* **c\_functions.c**: Performs simple vector-matrix operations.
* **chebyshev.c**: Generates Chebyshev polynomials of the first kind.
* **clenshaw\_curtis\_II.c**: Generates constant matrices for second-order Clenshaw-Curtis quadrature.
* **ecef2eci.c**: Converts states from the Earth-Centered Earth-Fixed (ECEF) frame to the Earth-Centered Inertial (ECI) frame.
* **eci2ecef.c**: Converts states from the Earth-Centered Inertial (ECI) frame to the Earth-Centered Earth-Fixed (ECEF) frame.
* **EGM2008.c**: Computes the spherical harmonic gravity for a specified degree and order using the EGM2008 model.
* **FandG.c**: Computes the analytical solution to the two-body problem in celestial mechanics.
* **Interpolate.c**: Interpolates the solution to user-specified output intervals (ephemeris).
* **lsq\_chebyshev\_fit.c**: Constructs the least squares operator and Chebyshev matrix for fitting data.
* **makefile**: A script to compile all the source code for the Adaptive Picard-Chebyshev integration.
* **matrix\_builder.c**: Builds and stores constant matrices required for the Adaptive Picard-Chebyshev method.
* **matrix\_loader.c**: Loads the constant matrices required for the Adaptive Picard-Chebyshev method.
* **perigee\_approx.c**: Computes the approximate Keplerian perigee when the initial conditions provided by the user do not correspond to perigee.
* **perturbed\_gravity.c**: Computes gravity using terminal convergence approximations to enhance efficiency.
* **picard\_chebyshev\_propagator.c**: Propagates the solution segment by segment from the initial to the final time.
* **picard\_error\_feedback.c**: Computes the linear error correction term to accelerate Picard iteration convergence.
* **picard\_iteration.c**: Iterates one segment at a time until the solution converges.
* **polydegree\_segments.c**: Determines the number of segments per orbit and the Chebyshev polynomial degree required to meet the desired precision.
* **prepare\_propagator.c**: Loads matrices for the selected polynomial degree and calculates start and end times for each segment.
* **radial\_gravity.c**: Computes the gravity degree required based on the satellite's distance from Earth's surface.
* **reosc\_perigee.c**: Reosculates the Keplerian perigee at the end of each orbit.
* **rv2elm.c**: Converts Cartesian coordinates into Keplerian orbital elements.

## **Running** **Other Propagation Algorithms (RK4, RK8, ODE45, ODE78, ODE113)**

## 

### 10.4.1. Clone the Algorithms Repository

* Open CLion IDE.
* Clone the repository:  
  🔗 [SatPropagatorAnalysis Repository](https://github.com/shadi26/SatPropagatorAnalysis.git)

### 10.4.2. Configure and Run the Code

* Open Main\_to\_print\_executable\_time.cpp.
* Update the following variables:
* r0: Initial Position
* v0: Initial Velocity
* orbital\_period: Satellite’s Orbital Period
* total\_time: Total Simulation Time
* A: Cross-sectional area
* m: Mass
* C\_D: Drag coefficient
* Number of segments in each orbital period
* NUM\_GAUSS\_LOBATTO\_POINTS: Number of points in each segment
* Run the code.

### 10.4.3. Output Files

* Satellite\_ODE113\_final\_positions\_8\_16.csv
* Satellite\_ODE45\_final\_positions\_8\_16.csv
* Satellite\_ODE78\_final\_positions\_8\_16.csv
* Satellite\_RK8\_final\_positions\_8\_16.csv
* Satellite\_RK4\_final\_positions\_8\_16.csv
* Algorithm\_excution\_time.csv

Note : the algorithm name would be Satellite\_ODE113\_final\_positions\_{number of segments}\_{number of gauss lobatto points}

### 10.4.4. Code and functions details

* **SatelliteUtils.cpp**: Provides utility functions for satellite propagation, including vector operations, acceleration calculations, and integration of satellite motion dynamics.
* **ODE113.cpp**: Implements the ODE113 adaptive step-size numerical integrator with error estimation for solving satellite propagation equations.
* **ODE45.cpp**: Provides an implementation of the Runge-Kutta-Fehlberg 4(5) method for solving differential equations with error control.
* **ODE78.cpp**: Implements the Runge-Kutta 7(8) method for precise satellite motion propagation using adaptive step sizing.
* **RK4.cpp**: Implements the 4th-order Runge-Kutta method for solving satellite equations of motion over Gauss-Lobatto points.
* **RK8.cpp**: Implements the Dormand-Prince 8th-order Runge-Kutta method for highly accurate satellite motion propagation.
* **coefficients78.cpp**: Contains coefficients and data structures for the 7(8) Runge-Kutta method using Butcher tableau representations.
* **CommonFunctions.cpp**: Provides reusable mathematical operations such as vector manipulation, norm calculation, and helper functions for integrators.
* **Makefile**: Automates the compilation process for building satellite propagation analysis tools.
* **Main.cpp**: Serves as the entry point for testing and comparing satellite propagation algorithms using various numerical integrators.

## **Running Final Tests and Generating Graphs**

### 10.5.1. Prepare the Data

* Install this libraries to your ide that uses python {pip install (numpy,sgp4,matplotlib,pandas,scipy)}
* Open main.py.
* Update the following variables:
* Tle\_data: Satellite TLE data.
* Add the tle data (in the suitable section “tle\_data=[]” and make sure you add the name exactly as the satellite folder name)

Example” (

ASBM-2",

"1 60423U 24143B 24286.24012071 .00000053 00000-0 00000-0 0 9996",

"2 60423 62.3237 68.6665 5330361 267.8786 33.3421 1.50421044  2382")

)

* Tspan: Time span for the simulation.
* Adjust the time in function tspan (to run the baseline module “[0.1,000,000])
* Make sure you adjust the number of segments and node numbers
* Total\_time: Should be less than 1000000 (adjust if needed).

### 10.5.2. Organize Result Files

* Copy and paste the following files into the folder csvfiles->satellite name (example:csvfiles\ASBM-2)
* final\_positions\_results.csv → Rename to apci\_final\_positions\_results.csv
  + Satellite\_ODE113\_final\_positions\_8\_16.csv
  + Satellite\_ODE45\_final\_positions\_8\_16.csv
* Satellite\_ODE78\_final\_positions\_8\_16.csv
* Satellite\_RK8\_final\_positions\_8\_16.csv
* Satellite\_RK4\_final\_positions\_8\_16.csv
* Algorithm\_excution\_time

Note: all csv files must be in folder name “csvfiles” under the satellite name folder for example   
“asbm-2”

### 10.5.3. Run the Simulation

* Execute the Main.py script.
* Execute Main\_Exution\_times.py

Note: This simulation may take several hours to complete.

### 10.5.4. Code and functions details

***Python code functions :***

* **eci\_to\_geodetic(x, y, z):** Converts Earth-Centered Inertial (ECI) coordinates into geodetic coordinates (latitude, longitude, and altitude).
* **a\_c\_func(t, y, A, m, C\_D=2.2):** Calculates the total acceleration acting on a satellite, considering gravity, J2 perturbation, and atmospheric drag.
* **atmospheric\_density(altitude):** Determines atmospheric density based on altitude using an exponential model with predefined data for different altitude layers.
* **get\_satellite\_params(sat\_name):** Retrieves the cross-sectional area, mass, and drag coefficient of a satellite based on its name.
* **compute\_position\_velocity\_and\_geodetic(tle\_data, year, month, day, hour, minute, second):** Computes the position, velocity, and geodetic coordinates of a satellite using its TLE data and the given timestamp.
* **gauss\_lobatto\_points(n, a, b):** Generates scaled Gauss-Lobatto quadrature points within the interval [a, b].
* **compute\_orbital\_period\_in\_seconds(tle\_data**): Calculates the orbital period of a satellite in seconds using its semi-major axis derived from TLE data and Kepler's Third Law.
* **compare\_algorithms(...):** Compares the position differences of various orbital propagation algorithms against baseline results and records the execution time.
  + **read\_results(file\_name)**: Reads a CSV file containing time and position data, validates its structure, and returns the data as a NumPy array for further analysis; used in compare\_algorithms to process algorithm output files.
* **plot\_all\_algorithms(...):** Plots the position differences over time for multiple orbital propagation algorithms, segmented by configuration.
* **satellite\_motion\_only\_gravity(t, y, mu=398600):** Simulates satellite motion influenced solely by Earth's gravitational force using Newton's second law.

***C++ code functions :***

* **save\_results\_to\_csv(satellite\_name, algorithm\_name, time\_points, position\_differences)**Saves the position differences over time for a specific satellite and algorithm to a CSV file.
* **save\_final\_positions\_to\_csv(satellite\_name, algorithm\_name, time\_points, positions)**Saves the final computed positions (X, Y, Z) of a satellite at different time points into a CSV file.
* **writeToCSV(tout, yout, filename, write\_header=false)**Writes satellite state information (time, position) to a CSV file, allowing data to be appended or written from scratch.
* **compare\_RK4\_algorithm\_with\_segments(algorithm\_name, algorithm\_func, r0, v0, total\_time, orbital\_period, A, m, C\_D)**Compares the RK4 propagation method over multiple orbital segments using Gauss-Lobatto time points and records execution time.
* **compare\_RK8\_algorithm\_with\_segments(algorithm\_name, algorithm\_func, r0, v0, total\_time, orbital\_period, h, A, m, C\_D)**Compares the RK8 propagation method over multiple segments using Gauss-Lobatto points and records execution time.
* **compare\_ODE45\_algorithm\_with\_segments(algorithm\_name, algorithm\_func, r0, v0, total\_time, orbital\_period, A, m, C\_D, rtol, atol)**Compares the ODE45 propagation algorithm, applying numerical integration over orbital segments and recording execution time.
* **compare\_ODE78\_algorithm\_with\_segments(algorithm\_name, algorithm\_func, r0, v0, total\_time, orbital\_period, A, m, C\_D, rtol, atol)**Runs the ODE78 propagation algorithm using Gauss-Lobatto points, applying adaptive numerical integration**.**
* **compare\_ODE113\_algorithm\_with\_segments(algorithm\_name, r0, v0, tol, hmax, hmin, mu, orbital\_period, total\_time, A, m, C\_D)**Compares the ODE113 algorithm with adaptive step-size control to propagate satellite motion while tracking execution time.

***C code functions :***

* **adaptive\_picard\_chebyshev(initial\_positions, initial\_velocities, t0, tf, dt, deg, tol, soln\_size, Feval, Soln)**

 Calls the **Adaptive Picard-Chebyshev (APC) method** to compute satellite motion for a given time step.

 Uses initial position and velocity extracted from the CSV file.

 Applies **high-order Picard iteration with Chebyshev polynomial approximation** for precise trajectory propagation.

 Outputs position and velocity at discrete time points stored in Soln.

* **ID2(soln\_size, i, soln\_size) :** Computes the index mapping for extracting position values from the Soln array at the final time step.

# **Appendix : Phase One**

## 

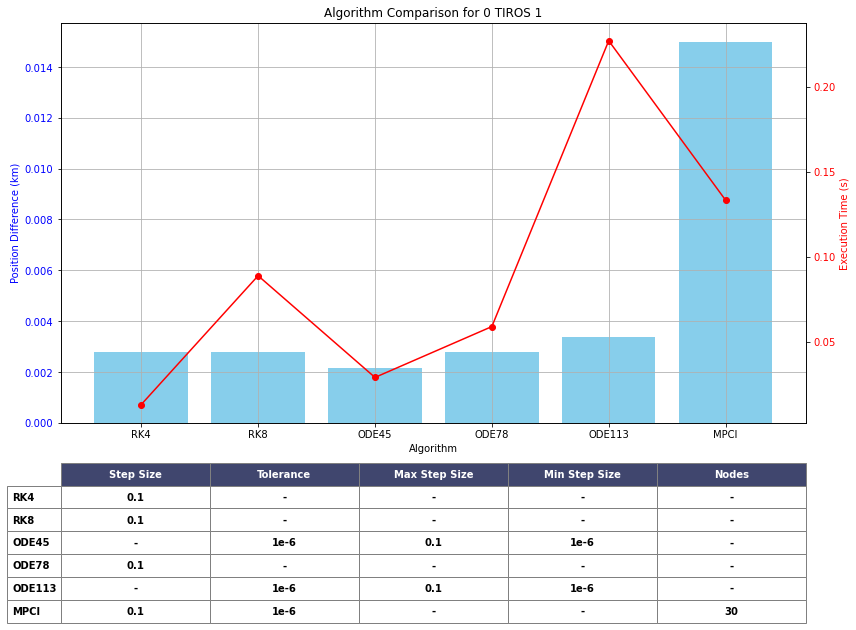
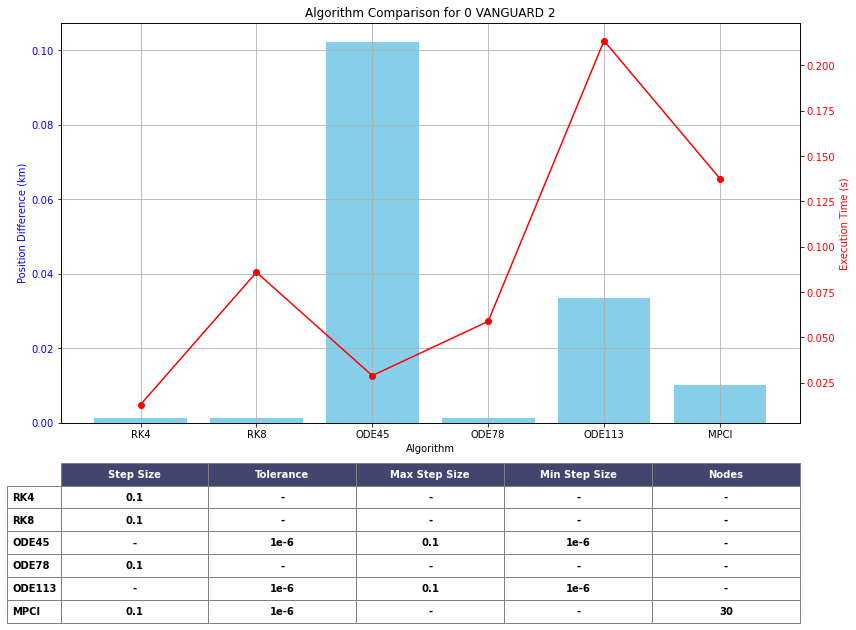
## Individual Satellite Results Plot

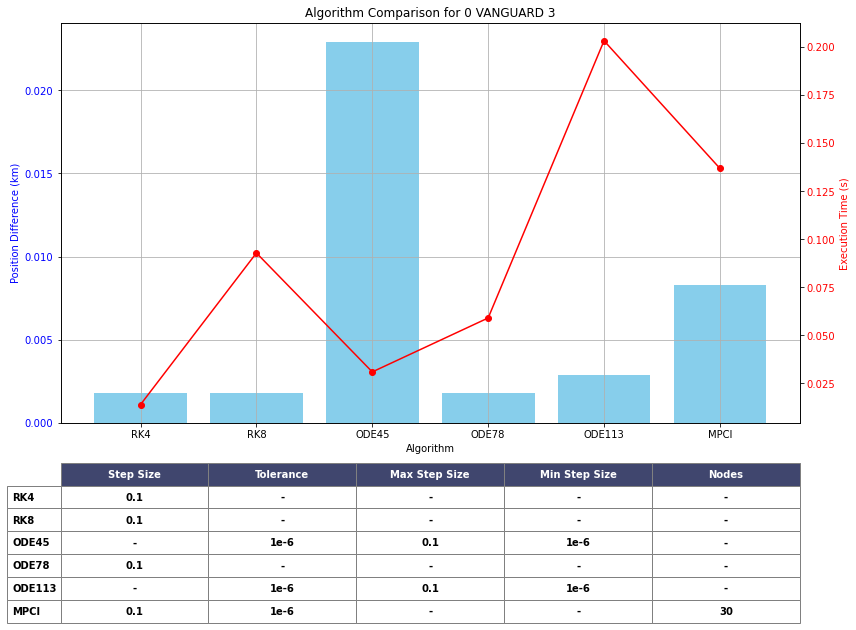
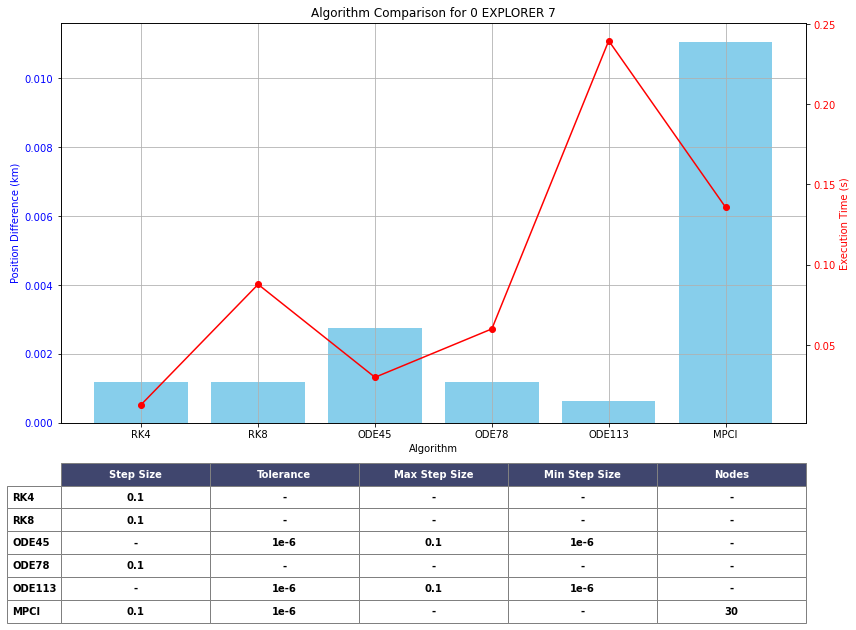
For each satellite, a detailed graph was generated to visualize the performance of the algorithms. These graphs show how each algorithm predicted the satellite’s position over time(execution time), allowing for a direct comparison between different methods.

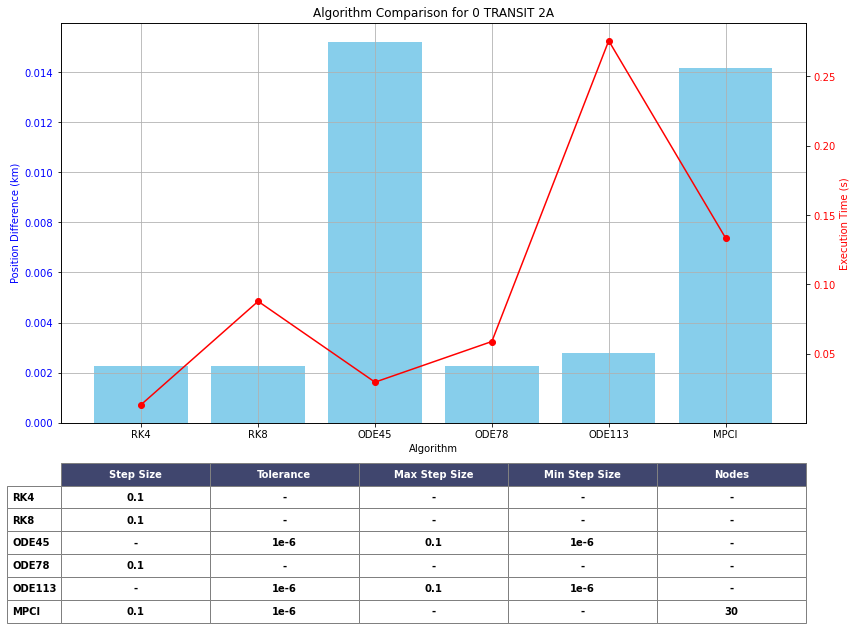
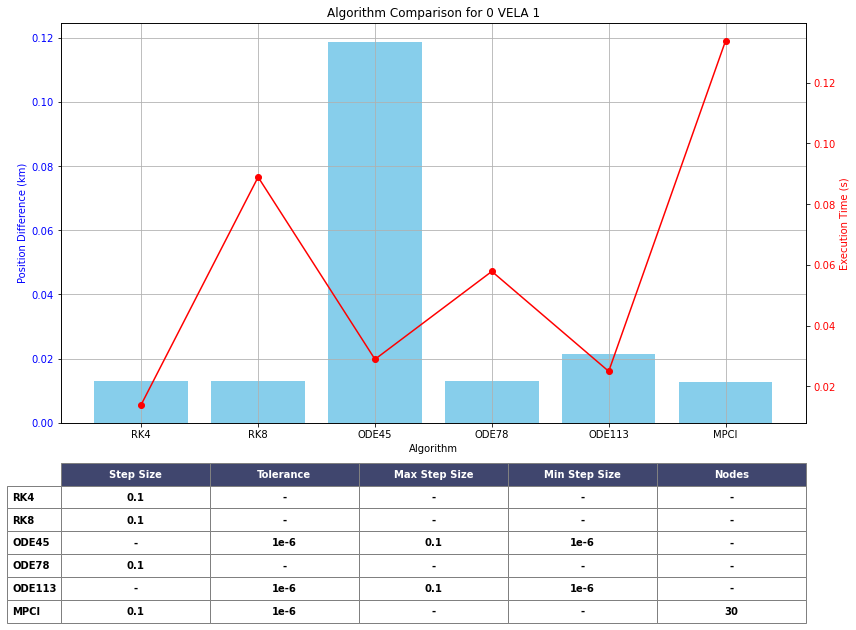
Each satellite’s graph includes:

* **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.

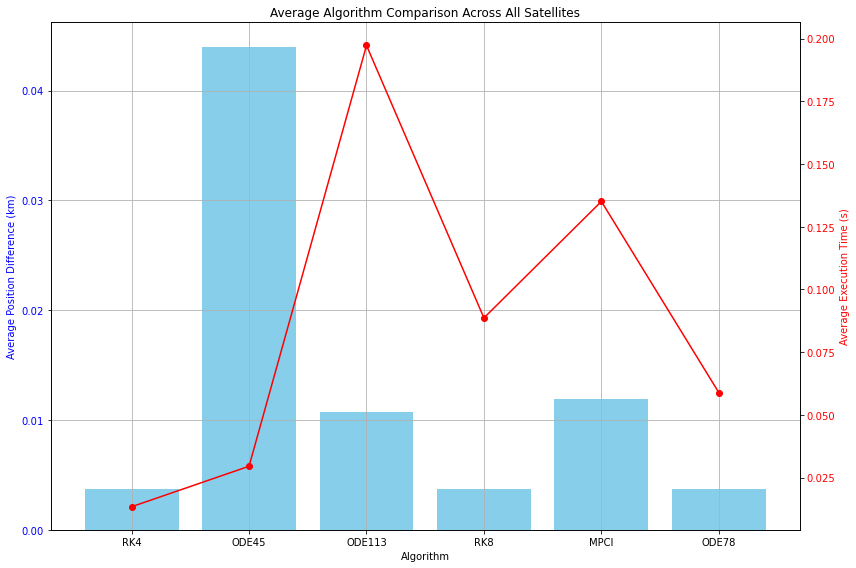






### 

### 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.

## 8.5. 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.

## **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.

# **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>

1. B ai, X., Junkins, J. L., & Frisbee, J. H. (2010). Modified Chebyshev-Picard Iteration Methods for Solution of Ordinary Differential Equations.  
   <https://oaktrust.library.tamu.edu/server/api/core/bitstreams/3a59777c-e5cd-4681-b971-d4dd61fb22d5/content>
2. Adaptive Picard Chebyshev code from woodland  
   <https://github.com/woollands/adaptive_picard_chebyshev_C>