

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** 4](#_Toc179875496)

[**Key words:** 4](#_Toc179875497)

[**2.** **Introduction** 4](#_Toc179875498)

[**3.** **Background and Related Work** 5](#_Toc179875499)

[3.1. Phase One: Evaluation of Key Algorithms for State Propagation: 5](#_Toc179875500)

[3.2. Phase Two: Refinements and New Contributions: 6](#_Toc179875501)

[3.3. Application of Woodland’s Adaptive Picard-Chebyshev Iteration (APCI): 6](#_Toc179875502)

[3.4. Performance Comparison and Hardware Testing 6](#_Toc179875503)

[**4.** **Expected Achievements** 7](#_Toc179875504)

[**5.** **Algorithms Analysis** 8](#_Toc179875505)

[**5.1.** **Runge-Kutta 4th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK4)[3][4]** 9](#_Toc179875506)

[5.1.1. Mathematical Formulas and Coefficients Table**[13]** 9](#_Toc179875507)

[5.1.2. Special Case 11](#_Toc179875508)

[5.1.3. Pseudocode: 12](#_Toc179875509)

[5.1.4. Time Complexity: 13](#_Toc179875510)

[5.1.5. Space Complexity: 13](#_Toc179875511)

[5.1.6. Edge Cases and Limitations 13](#_Toc179875512)

[**5.2.** **Runge-Kutta 8th Order with Gauss-Lobatto Quadrature Points and Sectional Time Span (RK8) [4][6]** 14](#_Toc179875513)

[5.2.1. Mathematical Formulas and Coefficients table**[5][13]** 15](#_Toc179875514)

[5.2.2. Special Case 17](#_Toc179875515)

[5.2.3. Pseudocode: 18](#_Toc179875516)

[5.2.4. Time Complexity: 20](#_Toc179875517)

[5.2.5. Space Complexity: 20](#_Toc179875518)

[5.2.6. Edge Cases and Limitations 20](#_Toc179875519)

[**5.3.** **Dormand-Price Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE45)** **[4][14]** 21](#_Toc179875520)

[5.3.1. Mathematical Formulas and Coefficients Table**[5][13]** 22](#_Toc179875521)

[5.3.2. Special Case 24](#_Toc179875522)

[5.3.3. Pseudocode: 25](#_Toc179875523)

[5.3.4. Time Complexity: 27](#_Toc179875524)

[5.3.5. Space Complexity: 27](#_Toc179875525)

[5.3.6. Edge Cases and Limitations: 27](#_Toc179875526)

[**5.4.** **Verner’s Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE78)[6][12]** 28](#_Toc179875527)

[5.4.1. Mathematical Formulas and Coefficients table**[4][15]** 29](#_Toc179875528)

[5.4.2. Special Case 31](#_Toc179875529)

[5.4.3. Pseudocode: 32](#_Toc179875530)

[5.4.4. Time Complexity: 34](#_Toc179875531)

[5.4.5. Space Complexity: 34](#_Toc179875532)

[5.4.6. Edge Cases and Limitations: 34](#_Toc179875533)

[**5.5.** **Adams-Bashforth-Moulton Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE113)[7]** 35](#_Toc179875534)

[5.5.1. Mathematical Formulas and Coefficients**[16][7]** 36](#_Toc179875535)

[5.5.2. Special Case 38](#_Toc179875536)

[5.5.3. Pseudocode: 39](#_Toc179875537)

[5.5.4. Time Complexity: 41](#_Toc179875538)

[5.5.5. Space Complexity: 41](#_Toc179875539)

[5.5.6. Edge Cases and Limitations: 41](#_Toc179875540)

[**5.6.** **Modified Picard-Chebyshev Iteration (MPCI)[17]** 42](#_Toc179875541)

[5.6.1. Mathematical Formulas and Coefficient 42](#_Toc179875542)

[5.6.2. Pseudocode: 43](#_Toc179875543)

[5.6.3. Time Complexity: 46](#_Toc179875544)

[5.6.4. Space Complexity: 46](#_Toc179875545)

[5.6.5. Edge Cases and Limitations: 46](#_Toc179875546)

[**5.7.** **Adaptive Picard-Chebyshev Iteration with Segmentation and Chebyshev Polynomial Approximation (APCI)** **[8]** 48](#_Toc179875547)

[5.7.1. Mathematical Formulas and Coefficient 49](#_Toc179875548)

[5.7.2. Pseudocode**[18]** 52](#_Toc179875549)

[5.7.3. Time Complexity: 54](#_Toc179875550)

[5.7.4. Space Complexity: 54](#_Toc179875551)

[5.7.5. Edge Cases and Limitations 54](#_Toc179875552)

[**1.** **Evaluating Algorithms for Implementation** 56](#_Toc179875553)

[**2.** **References** 58](#_Toc179875554)

# **Abstract**

# **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). 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 applying Woodland’s version 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 exploring the possibility of using a Raspberry Pi or 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, 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, and MPCI) 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 Raspberry Pi, another group’s hardware, or the existing card on the youth satellite, depending on availability.

# **Expected Achievements**

In phase two of this study, we anticipate achieving the following milestones:

1. **Refinement of Algorithms from Phase One:**
   * Building on the results of phase one, we will enhance the previously tested algorithms (RK4, RK8, ODE45, ODE78, ODE113, and the Modified Picard-Chebyshev Iteration) by implementing dynamic step size adjustment and incorporating Gauss-Lobatto quadrature for improved accuracy and efficiency over long durations.
2. **Implementation of Woodland’s APCI Algorithm:**
   * Applying Woodland’s version of the Adaptive Picard-Chebyshev Iteration (APCI) to assess its performance against the existing algorithms. Woodland’s APCI will be analyzed for both running time and accuracy, providing a deeper understanding of its applicability in autonomous satellite navigation systems.
3. **Expanded Simulation Setup:**
   * Continuing the use of C and C++ for the simulation environment, we will now integrate the capability to dynamically calculate positions and velocities at specific points, improving the practical utility of the algorithms. This phase will also include testing the integration of Gauss-Lobatto quadrature points in the propagation methods to enhance both accuracy and efficiency.
4. **Algorithm Testing on Emulated Hardware:**
   * In addition to running the algorithms on a PC, we will perform tests on a satellite computer emulator, simulating real-time satellite operations. This is a critical step to ensure that the algorithms can function efficiently on actual satellite hardware. Potential hardware for this testing includes a Raspberry Pi, another group's card, or the youth satellite’s onboard computer.
5. **Comparative Performance Analysis:**
   * Conducting detailed performance comparisons of the enhanced algorithms. We will generate error graphs as a function of time at regular intervals and specific points, such as CATCH points, to assess both accuracy and computational efficiency.
6. **Iterative Refinement and Analysis:**
   * Continuing the iterative analysis from phase one, we will refine the evaluation criteria based on phase two’s results. This process will focus on further optimizing the algorithms for real-time collision avoidance and navigation scenarios, ensuring robustness and efficiency.
7. **Preliminary Findings and Documentation:**
   * Producing comprehensive results that compare the performance of all evaluated algorithms. These findings will identify the most effective state propagation methods for autonomous satellite navigation, and all methodologies and results will be thoroughly documented for future reference and further research.

# **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.

**New Phase Two Contributions:**

* **Dynamic Step Size:** Algorithms now use an adaptive step size ​ to improve accuracy and efficiency, particularly over long durations.
* **Gauss-Lobatto Quadrature:** We integrate Gauss-Lobatto quadrature points to enhance numerical accuracy and stability.
* **One-Point Calculation:** All algorithms now allow for the calculation of positions and velocities at specific points, improving real-time navigation capabilities.
* **Woodland’s APCI Algorithm:** Woodland’s Adaptive Picard-Chebyshev Iteration (APCI) is implemented to optimize satellite motion propagation.

Using C and C++ simulations, we will assess the performance of each algorithm under various conditions, focusing on accuracy, computational efficiency, and robustness, with a special focus on real-time application using emulated satellite hardware.

## **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.

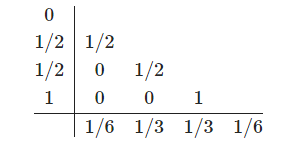
: The current time point.

​: 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

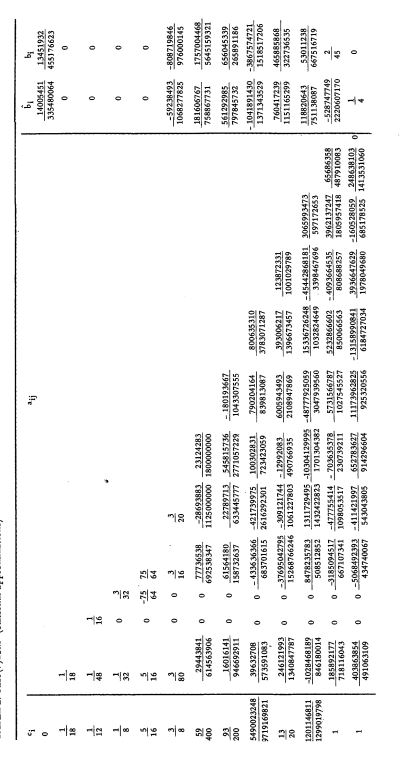
points.

: 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 computationally efficient method for solving ODEs, especially in scenarios requiring precise calculations, such as satellite motion. It provides a good balance between high accuracy and computational cost per step, making it ideal for long-duration simulations. 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.

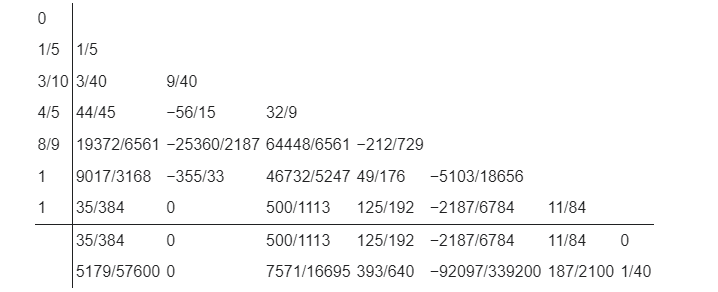
: The current time point.

​: 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, time\_points, y0, tol, hmax, hmin):**

**# Step 1: Handle Single Time Point**

**If length of time\_points == 1:**

# Expand the single time point into a small interval

time\_points = [time\_points[0] - 0.99 \* time\_points[0], time\_points[0] + small\_offset]

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

t\_values\_list = [time\_points[0]] # List for storing time points

y\_values\_list = [y0] # List for storing solution values

**# Step 3: Use RK4 for the Initial Few Steps**

**For i = 0 to 2:**

**If i >= length of time\_points - 1:**

Break # Exit loop if out of time points

t = time\_points[i]

h = time\_points[i + 1] - t # Calculate step size

**# Perform RK4 step to get the next value**

y\_next = rk4\_step(f, t, y\_values\_list[-1], h)

**# Store the time and solution values**

t\_values\_list.append(time\_points[i + 1])

y\_values\_list.append(y\_next)

**# Step 4: Apply Adams-Bashforth-Moulton for Subsequent Steps**

**For i = 3 to length of time\_points - 2:**

t = time\_points[i]

h = time\_points[i + 1] - t # Calculate step size

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

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

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

- 59 \* f(time\_points[i - 1], y\_values\_list[-2])

+ 37 \* f(time\_points[i - 2], y\_values\_list[-3])

- 9 \* f(time\_points[i - 3], y\_values\_list[-4])

)

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

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

9 \* f(time\_points[i + 1], y\_pred)

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

- 5 \* f(time\_points[i - 1], y\_values\_list[-2])

+ f(time\_points[i - 2], y\_values\_list[-3])

)

**# Store the updated time and solution values**

t\_values\_list.append(time\_points[i + 1])

y\_values\_list.append(y\_correct)

**# Step 5: Return Time and Solution Values**

**Return (t\_values\_list, y\_values\_list)**

### 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. The initial RK4 steps also require a constant amount of work, which does not significantly affect the overall complexity.

### 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.

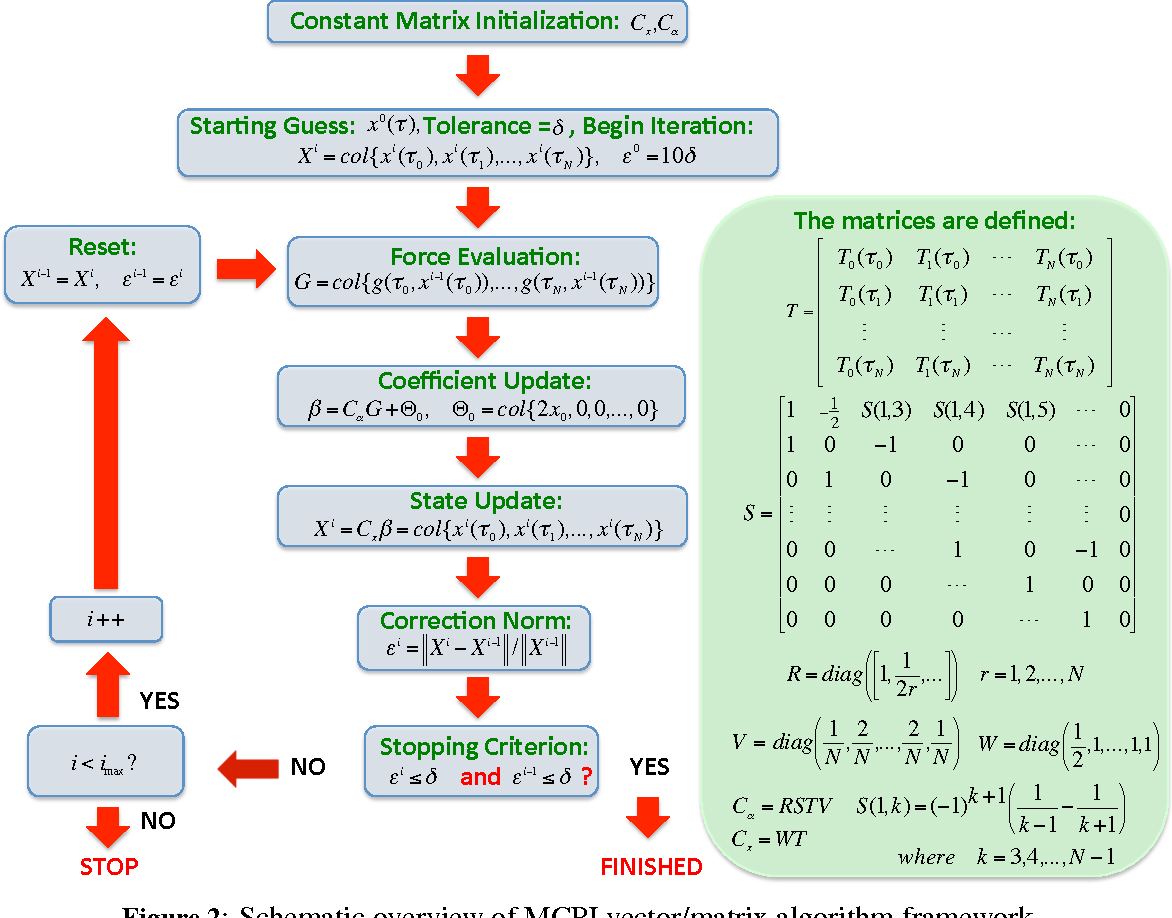
**Conclusion:** The Adams-Bashforth-Moulton method (ODE113) with Gauss-Lobatto points is a robust multi-step integrator for solving ODEs over long durations. Its combination of predictor-corrector steps allows for accurate and efficient calculations, making it ideal for scenarios like satellite motion 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 Iteration:** 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.

* **Total Complexity:** O (), where S is the number of segments, and N is the number of nodes (polynomial degree) in each segment.

**Explanation:** The total time complexity is determined by the segmentation of the time span and the polynomial degree for each segment. Since the method performs iterations over N nodes for each segment and propagates the solution across S segments, the total complexity scales with . The adaptive nature of the algorithm ensures that the method is applied over manageable segments, improving accuracy and computational efficiency.

### 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.

# **Evaluating Algorithms for Implementation**

## 

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

* **Modified Picard-Chebyshev Iteration (MPCI)**: While MPCI offers precise trajectory analysis and can calculate specific points, it remains unchanged in phase two and relies on heavy matrix operations, making it less suitable for older hardware. Although it shows potential in theory, its computational demands and lack of optimization make it less favorable than simpler and more efficient algorithms like RK4 and RK8 in this context. Therefore, MPCI is unlikely to outperform RK4 and RK8 for our hardware limitations.
* **Woodland's Adaptive Picard-Chebyshev Iteration (APCI):** Introduced in phase two, APCI builds upon the Picard-Chebyshev approach but with adaptive step size and the use of Gauss-Lobatto quadrature points. These improvements help APCI achieve better accuracy and efficiency for satellite state propagation, particularly over long durations. APCI is a more suitable alternative to MPCI for this phase, due to its adaptive nature and computational optimizations.
* **Verner's Method (ODE78)**: ODE78, an adaptive Runge-Kutta method of order 7(8), offers high accuracy but with higher computational demands. The dynamic step size introduced in phase two helps balance efficiency and accuracy. ODE78 can be effective for hardware that can accommodate larger step sizes, but it might still be more resource-intensive than RK4 and RK8.
* **Dormand-Prince Method (ODE45)**: ODE45 continues to be an excellent choice due to its adaptive step size, which dynamically adjusts to reduce the computational load while maintaining accuracy. Its efficiency on less powerful hardware makes ODE45 a strong contender for implementation in this phase.
* **Runge-Kutta Methods (RK4 and RK8)**: RK4 is favored for its balance of simplicity, accuracy, and low computational demand, making it ideal for older hardware. RK8 provides higher accuracy, though with greater computational cost, but phase two’s dynamic step size adjustments can help reduce this overhead. Given the hardware limitations, RK4 remains the most appropriate choice, with RK8 being a secondary option if additional accuracy is crucial and resources allow.
* **Adams-Bashforth-Moulton Method (ODE113)**: ODE113, an adaptive solver, balances accuracy and efficiency, though it still requires careful consideration of its computational overhead. Phase two’s refinements improve its performance, but like other higher-order methods, it may be too resource-intensive for older hardware.

**Conclusion:** Given the hardware constraints and phase two’s improvements, the most suitable algorithms to implement are:

1. **Dormand-Prince Method (ODE45)**: For its adaptive step size and efficient performance on constrained hardware
2. **Woodland's APCI:** For its adaptive nature, precision, and improvements through Gauss-Lobatto quadrature points and dynamic step sizing.
3. **Runge-Kutta Methods (RK4 and RK8):** RK4 remains the most efficient choice for older hardware, with RK8 as an option for increased accuracy when resources allow

Although MPCI offers precise trajectory calculations, its reliance on matrix operations and lack of phase two enhancements make it less suited to this phase. APCI, ODE45, and RK4/8 are expected to outperform MPCI given our current hardware limitations..

# **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>