**Purpose:**

The Modified Picard-Chebyshev Iteration (MPCI) method is a numerical approach used for solving ordinary differential equations (ODEs), particularly for high-precision applications such as satellite orbit propagation. MPCI leverages Chebyshev polynomials and Picard iterations to offer a more accurate solution by iterating over a fixed-point approximation. The method is particularly useful when high precision over long time spans is needed.

**Overview:**

MPCI works by iteratively refining the solution to an ODE using Chebyshev polynomials to approximate the solution over each time interval. Each iteration adjusts the solution based on intermediate states sampled at Chebyshev nodes, aiming to improve convergence with a quasi-linearization feedback mechanism. The method dynamically adjusts the number of nodes and time step based on error estimation to balance accuracy and computational cost.

**Formulas:**

**Chebyshev Polynomial Approximation:**

where Tk(τ) are Chebyshev polynomials, and ak are coefficients.

**Picard Iteration Update**:

where the summation represents the weighted contribution of each Chebyshev node.

**Error Feedback:**

where α is a feedback factor to adjust the correction based on the error.

### **Pseudocode:**

function MPCI(func, tspan, y0, h\_init=0.1, tol=1e-10, max\_iter=100, N\_init=20)

// Parameters:

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

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

// y0 - initial condition

// tol - tolerance for convergence

// max\_iter - maximum number of iterations

// N\_init - initial number of Chebyshev nodes

// h\_init - initial time step size

t0 = tspan[0]

tf = tspan[1]

t\_values = [t0]

y\_values = [y0]

h = h\_init // Start with a smaller time step

N = N\_init // initial number of nodes

M = N + 1

tau = cos(linspace(0, pi, M)) // Chebyshev nodes in [-1, 1]

function adjust\_nodes(error)

// Dynamically adjust the number of Chebyshev nodes based on the error.

if error < tol / 10

return max(10, N - 5)

else if error > tol

return min(50, N + 5)

return N

while t\_values[-1] < tf

t = t\_values[-1]

y = y\_values[-1]

prev\_y = copy of y

for iteration from 0 to max\_iter - 1

Xn = array of zeros with dimensions (M, length of y) // Chebyshev states

xAdd = array of zeros with same shape as Xn

// Map Chebyshev nodes to actual time

for node from 0 to M - 1

tau\_val = tau[node]

tn = t + h \* (tau\_val + 1) / 2 // map Chebyshev nodes to time

xAdd[node] = func(tn, y)

// Integrate using Chebyshev nodes

y\_new = y + h \* dot product of xAdd.T and (1 / M) \* array of ones with length M // weighted sum of Chebyshev nodes

// Calculate error between iterations

error = norm(y\_new - prev\_y)

prev\_y = copy of y\_new

// Adjust number of nodes dynamically based on error

N = adjust\_nodes(error)

M = N + 1

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

if error < tol

break // Converged

append t + h to t\_values

append y\_new to y\_values

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

return results

**Time Complexity:**

* **Per Iteration**: O(M⋅d), where M is the number of Chebyshev nodes and d is the dimension of the solution vector y (both constants). Each iteration requires evaluating the ODE at M nodes and performing matrix-vector operations.
* **Total Complexity**: O(n⋅k), where n is the number of time steps, and k is the average number of iterations required for convergence at each time step. The time complexity grows linearly with the number of time steps n and the number of iterations per time step k, as M and d are constants.

**Explanation:**

Each iteration involves evaluating the ODE at a fixed number of Chebyshev nodes MMM and updating the solution vector y, which has a constant dimension d. Since these values remain constant throughout the algorithm, the per-iteration complexity is constant, i.e., O(M⋅d). Therefore, the overall time complexity is proportional to n⋅k where n is the number of time steps and k is the number of iterations needed to converge at each time step.

.

**Space Complexity:**

**Overall**: O(n)

**Explanation:**

The algorithm stores the time points and the solution vector y at each time step. Since the dimension d of the solution vector is constant, the space complexity grows linearly with the number of time steps n. Additionally, a constant amount of space is used for intermediate Chebyshev node evaluations and feedback computations, but this does not significantly affect the overall space complexity. Therefore, the overall space complexity is O(n).

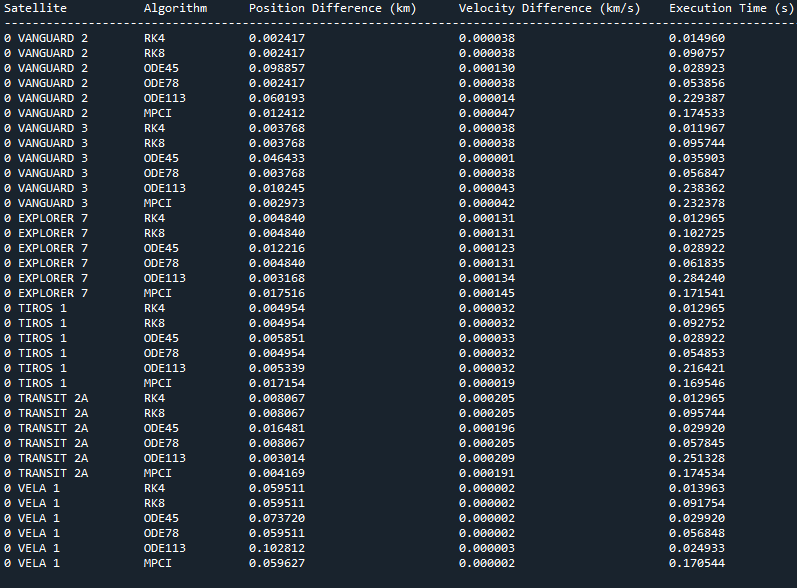
**Edge Cases and Limitations:**

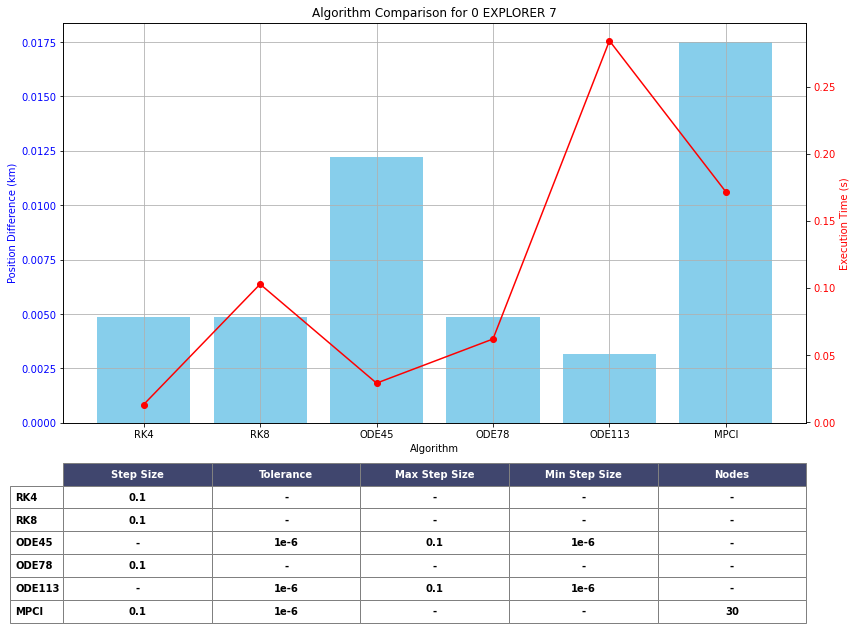
1. **Large Step Size**: If the time step h is too large, the algorithm may fail to converge or provide inaccurate results, especially for highly nonlinear ODEs.
2. **Small Step Size**: Using an excessively small step size will increase the number of iterations and computational cost, possibly leading to inefficiency.
3. **Stiff ODEs**: MPCI may not be suitable for stiff ODEs, where implicit methods or other adaptive approaches are more efficient.
4. **Chebyshev Node Adjustment**: Incorrect handling of node adjustments can lead to instability or poor convergence, especially for highly dynamic systems.

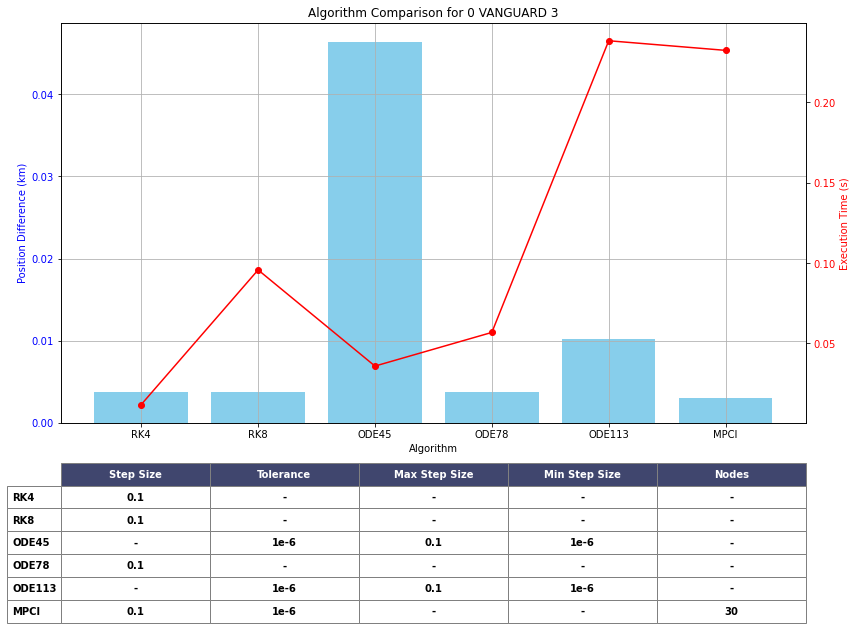
**Conclusion:**

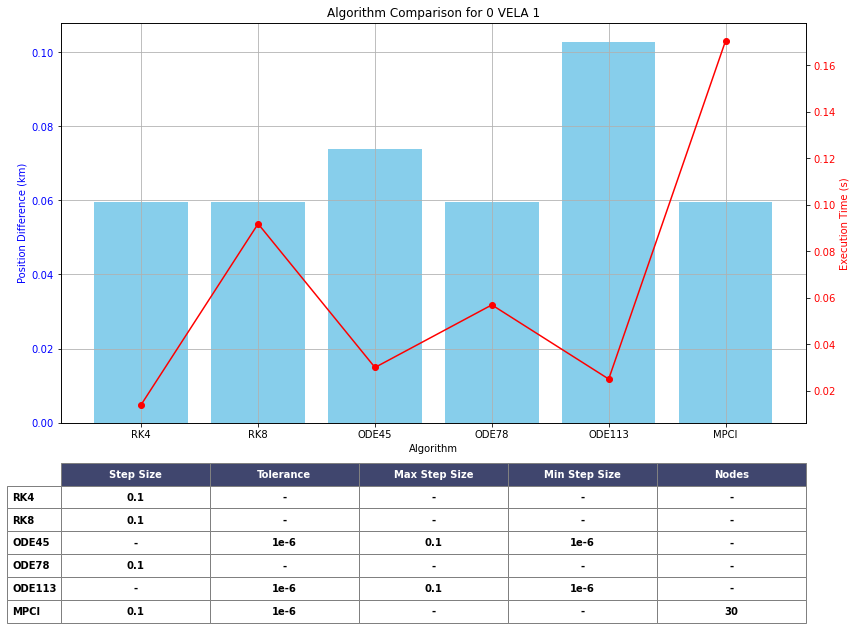
MPCI is a highly accurate method for solving ODEs, particularly for problems requiring long-term propagation with minimal error accumulation, such as satellite motion. However, it requires careful tuning of parameters like the number of Chebyshev nodes and step size to achieve the desired balance between accuracy and computational efficiency. While MPCI offers precision over long time spans, it may not be as efficient for stiff or highly nonlinear ODEs without further adaptation.

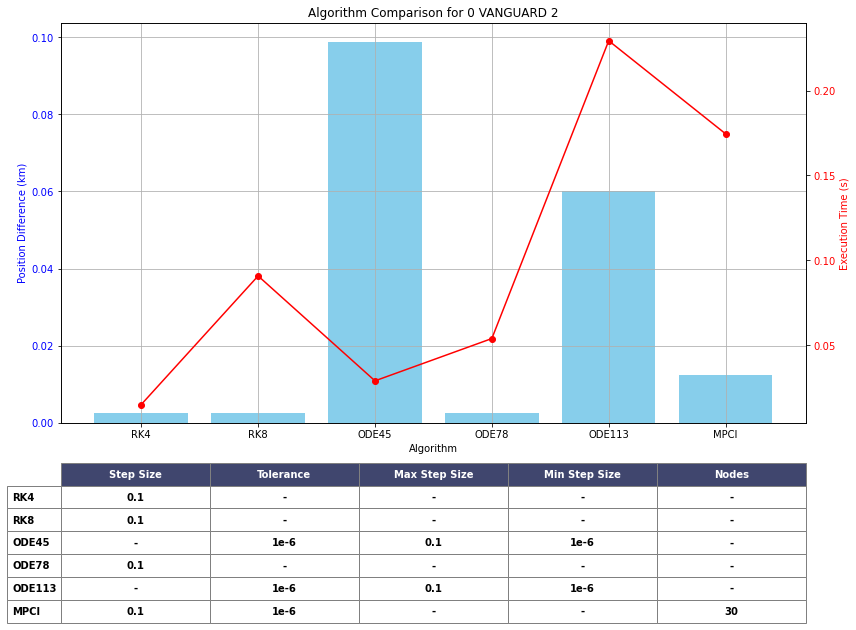
**Results**

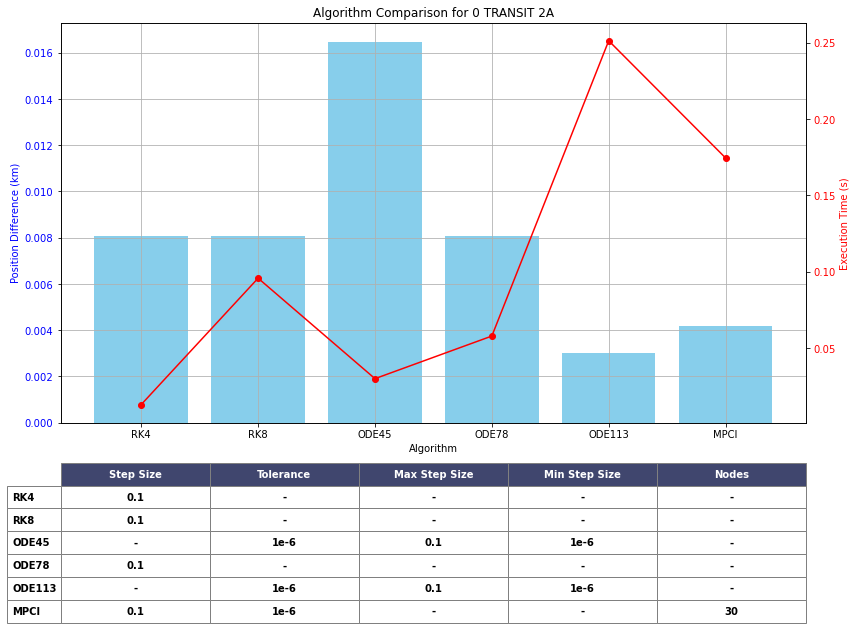
****

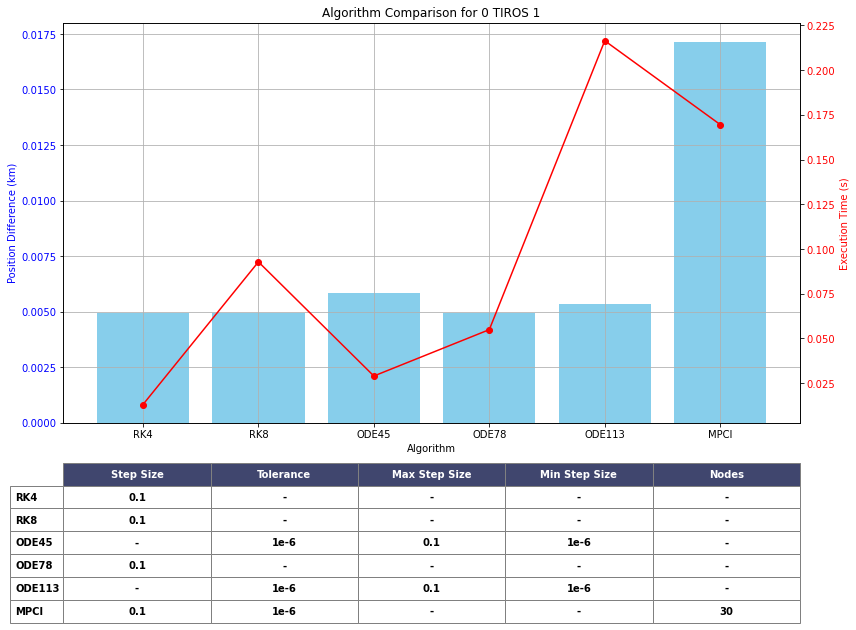


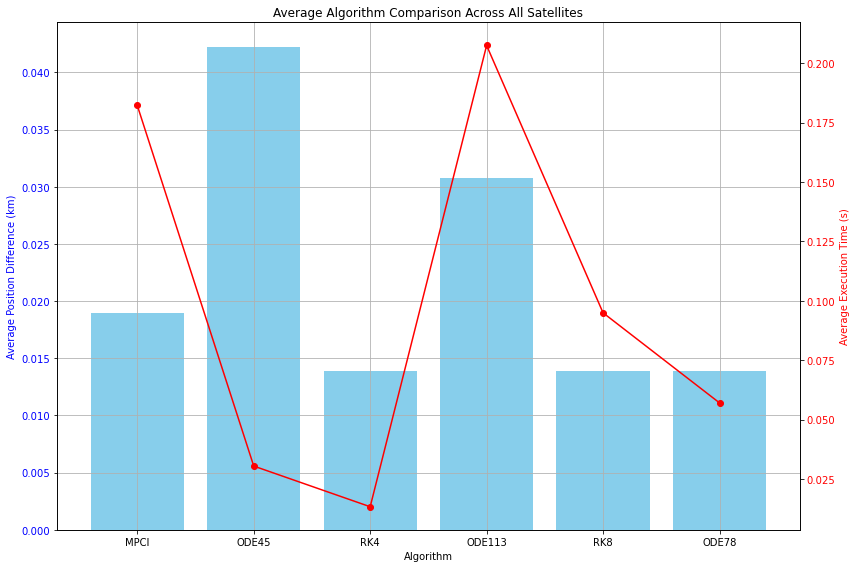












**8.7. Key Insights**

Key insights from the graph:

* **MPCI** has a moderate average position difference and a relatively long execution time compared to other algorithms.
* **ODE45** has the **largest average position difference**, indicating lower accuracy, and also shows a significant execution time, making it one of the slowest algorithms.
* **RK4** has the **smallest average position difference**, indicating the highest accuracy, and its execution time is quite fast, making it one of the most efficient algorithms in this comparison.
* **ODE113** shows a larger position difference compared to most other algorithms (except ODE45), and its execution time is the longest among all algorithms.
* **RK8** exhibits both low position difference and relatively fast execution time, demonstrating a good balance between accuracy and speed.
* **ODE78** also performs well in terms of both accuracy (low position difference) and execution time, making it one of the faster algorithms.

**9. Conclusion**

From this analysis, **RK4** emerges as the best algorithm, providing the highest accuracy (smallest position difference) with a fast execution time. It is an excellent candidate for real-time applications where precision is critical.

**ODE45**, on the other hand, performs poorly in terms of both accuracy and execution time, making it the least suitable algorithm for this task.

**RK8** shows a good balance between accuracy and speed, making it a strong contender for scenarios that require both quick calculations and precise results.

**MPCI** and **ODE113** offer a trade-off between accuracy and execution time but fall short in comparison to **RK4** and **RK8**. **ODE113** particularly stands out as having the longest execution time, which may not be suitable for time-sensitive applications.

**ODE78** offers a solid balance between execution time and accuracy, making it a viable choice for less demanding satellite tracking applications where a reasonable trade-off is acceptable.

This updated analysis better reflects the data presented in the graph, particularly the relationship between accuracy (position difference) and execution time for each algorithm.