## **Verner’s Method with Gauss-Lobatto Quadrature Points and Sectional Time Span (ODE78)**

**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

**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:**

[**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 = 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.