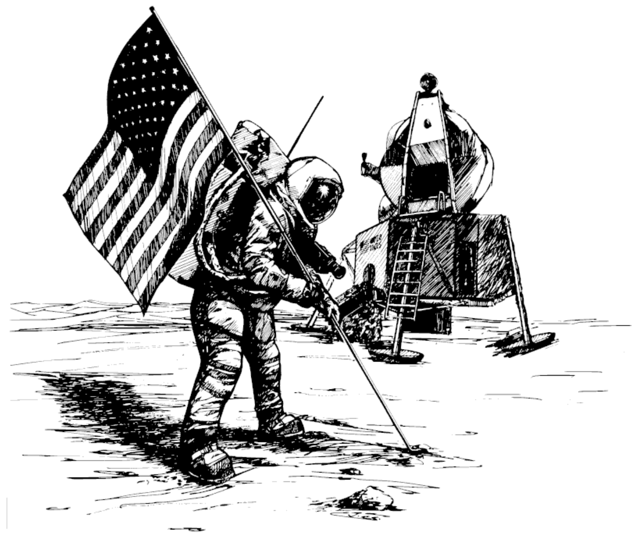

Lunar Lander Truss Project

AERO 423 - Aerospace Computational Methods

University of Michigan - Ann Arbor

February 14, 2020

Prepared by:

Drake Rundell

Image Source: https://www.needpix.com/photo/90155/apollo-moon-landing-nasa-usa-space-travel

Table of Contents

I List of Figures

II List of Equations

III Problem Description

The project will simulate the landing of robotic lunar rover. The landing dynamics are determined by the lander legs, which are flexible trusses that absorb the kinetic energy of a landing. The setup is illustrated in the figure below.

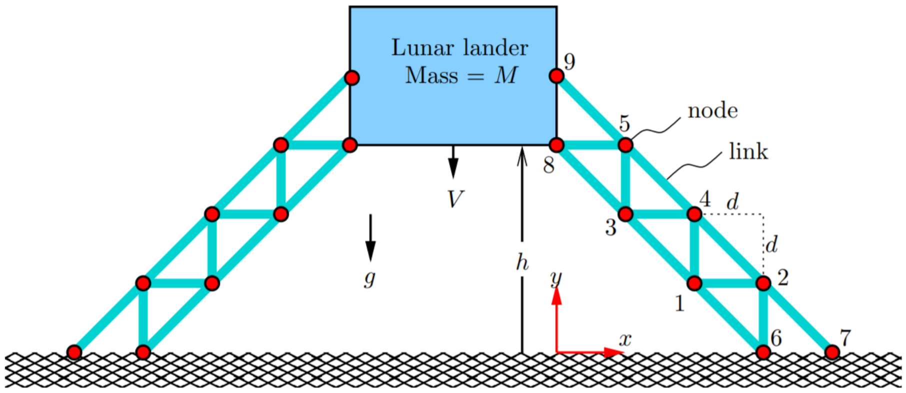

Figure 1: *Lunar Lander problem setup that features a lander with four legs; in the figure only two are shown with the other two out of the plane of the page and not shown.*

The lander has four legs, but only one will be simulated, shown on the right in the above figure. The other legs are calculated from the right leg since they are all symmetric. In addition, in-plane deformation will be assumed so that the problem remains two-dimensional. Each leg consists of a truss of nodes and links, as indicated above. Nodes 8 and 9 are bolted to the lander, and cannot move relative to each other; instead, these move vertically with the lander. In addition, upon landing, nodes 6 and 7 dig into the soil and cannot move.

On the other hand, nodes 1-5 can move arbitrarily, and their motion is dictated by the forces acting through the links to which they are attached. Finally, the motion of the lander is determined by the forces acting on it, which are its weight and the forces of the legs acting through the attachment points.

Link Force Calculation

The nodes move because of the forces exerted on them by the links. It is assumed that these links have no mass (they are very light relative to the nodes) and that the tensile force in a link depends on how much the link is stretched, δl, and how fast it is being stretched, $\dot{\delta} l$:

$$F_{link} = k\delta l + \gamma\dot{\delta} l \tag{1}$$

To calculate the resulting force contributions at the two adjacent nodes - called i and j a few definitions must be made:

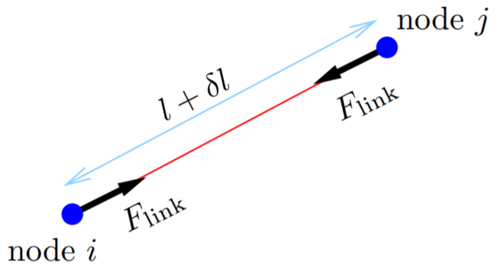

Figure 2: *Force in one link.*

$$
\begin{aligned}
\vec{x}_i = [x_i, y_i] &= \text{position of node } i \\
\vec{x}_j = [x_j, y_j] &= \text{position of node } j \\
\vec{v}_i = [u_i, v_i] &= \text{velocity of node } i \\
\vec{v}_j = [u_j, v_j] &= \text{velocity of node } j \\
l_0 &= \text{initial length of this link}
\end{aligned}
\tag{2}
$$

The quantities appearing in Equation 1 can then be calculated as:

$$\delta l = |\vec{x}_j - \vec{x}_i| - l_0 \tag{3}$$
$$\dot{\delta l} = (\vec{x}_j - \vec{x}_i) \cdot \vec{e}_{ij} \tag{4}$$

where $|\vec{\cdot}|$ refers to the magnitude of a vector and

$$\vec{e}_{ij} = \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|} \tag{5}$$

The force on nodes i and j due to this link are, respectively,

$$\vec{F}_{ij} = F_{link}\vec{e}_{ij} \tag{6}$$
$$\vec{F}_{ji} = -F_{link}\vec{e}_{ij} \tag{7}$$

As shown above, a positive F_{link} corresponds to a tension force that pulls the nodes closer together. δl is a positive when the link is stretched longer than its initial equilibrium length, and $\dot{\delta l}$ is positive when the link is actively being stretched. So $k\delta l$ is then a restorative force, while $\gamma\dot{\delta l}$ is a damping term. At every node, the forces from the links sum together to exert a net force on the node. The node's acceleration is given by this total force divided by the node mass.

Dynamics

Each free node of the truss in the lunar lander leg can move in the x and y direction. The acceleration of a free node is caused by the forces acting on it: its weight and the forces from the adjacent links. The lunar lander moves only vertically, by symmetry of the legs. The lander's (vertical) acceleration is caused by its own weight and the forces of the legs acting through nodes 8 and 9. At $t = 0$, the legs are in their equilibrium configuration, shown in Figure 1, with nodes 6 and 7 having just hit the ground, and with the rest of the nodes, and the lander, moving downward at speed V.

IV Parameters

The parameters relevant to the problem are listed below. Note, the lunar lander mass includes the masses of the affixed nodes, 8 and 9, from each leg.

T	=	0.5	[s]	Time Horizon
m_{mode}	=	0.1	[kg]	Node Mass
M	=	100	[kg]	Lander Mass
d	=	0.2	[m]	Initial Truss Width/Height
k	=	10^5	[N/m]	Link Stiffness
γ	=	200	[Ns/m]	Link Damping
g	=	1.625	[m/s^2]	Acceleration Due to Gravity
V	=	6	[m/s]	Initial Impact (descent) Speed

V Numerical Approach

To simulate the dynamics of the truss, use a state vector of 22 entries: the (x, y) positions and velocities of the 5 free nodes, and the height and vertical velocity of the lunar lander. The equations of motion will be of the form:

$$\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u}) \tag{8}$$

where $\mathbf{f}$ contains the physics (force-balance) calculations.

Three numerical integration techniques will be used to solve this system of ODEs: Forward Euler, second-order Adams Bashforth, and fourth-order Runge Kutta. When discretizing time, a constant time step $\Delta t = T/N_t$ will be used, where N_t is the number of time steps.

1 Equations of Motion for a Free Node and for the Lunar Lander

Using the equations of motion for a free node and for the lunar lander, a function can be programmed that takes input of the state, **u**, and any other required parameters, and computes the vector **f**. Using the state corresponding to the initial equilibrium configuration, $V = 0$, and $g = 0$, the verification can be made that the function returns **f** identically equal to zero.

1.1 Equations of Motion

EOM for Free Nodes (1-5):

$$\begin{bmatrix} \ddot{x}_i \\ \ddot{y}_i \end{bmatrix} = \frac{1}{m_{node}} \left(\sum_{i=1}^{elements} k\delta l + \gamma \dot{\delta l}\, \vec{e}_{ij} \right) - \begin{bmatrix} 0 \\ g \end{bmatrix} \tag{9}$$

EOM for Lunar Lander (Nodes 8-9):

$$\begin{bmatrix} \ddot{x}_i \\ \ddot{y}_i \end{bmatrix} = \frac{8}{M} \left(\sum_{i=1}^{elements} k\delta l + \gamma \dot{\delta l}\, \vec{e}_{ij} \right) - \begin{bmatrix} 0 \\ g \end{bmatrix} \tag{10}$$

1.2 MATLAB Implementation of EOMs

In order to create a function which outputs the vector **f**, a function to calculate the summation of the forces on each node must be created. This function will take in the given node state, a matrix of the position in x and y, and the velocity in u and v, along with the specific node of interest and output the summation of forces. The node state is written as such:

$$\mathbf{u} = \begin{bmatrix} x_1 & y_1 & u_1 & v_1 \\ x_2 & y_2 & u_2 & v_2 \\ x_3 & y_3 & u_3 & v_3 \\ x_4 & y_4 & u_4 & v_4 \\ x_5 & y_5 & u_5 & v_5 \\ x_6 & y_6 & u_6 & v_6 \\ x_7 & y_7 & u_7 & v_7 \\ x_8 & y_8 & u_8 & v_8 \\ x_9 & y_9 & u_9 & v_9 \end{bmatrix} \tag{11}$$

The organization of the node state will allow for the plotting of the node state at given time steps in future numerical schemes. The summation of forces using this node layout is done by creating a `for` loop which loops to the maximum size of the inputted nodes matrix. Within the loop, each node is read against a list of corresponding nodes to determine the initial distance of the links. The links are either initially 0.2 or 0.2828 meters. Once the distance is found, the force of the node of interest and the surrounding nodes is summed together and the function outputs the total force.

The **f** function can now be written to take in the node state **u** and output **f**. The function turns a state vector of positions and velocities into a vector of velocities and accelerations using the above equations of motion, while also accounting for the boundary conditions on nodes 6 and 7. Nodes 6 and 7 are not able to move from their bounded state due to their placement into the lunar surface. Therefore, the function sets the **f** output for these nodes to all zeroes. The free node states are calculated using the respective equation of motion. The lunar lander state is also calculated using its equation of motion with respect to the connected nodes of 8 and 9. However, the boundary condition that nodes 8 and 9 must remain 0.2 meters apart in the y direction is enforced through the neglecting of x plane velocities and accelerations.

The output of the **f** function turns the input of a matrix of node positions and velocities into a 36×1 column vector with the following form for all nine nodes:

$$\mathbf{f}(\mathbf{u}) = \begin{bmatrix} u_1 \\ v_1 \\ \dot{u}_1 \\ \dot{v}_1 \\ \vdots \\ u_9 \\ v_9 \\ \dot{u}_9 \\ \dot{v}_9 \end{bmatrix} \tag{12}$$

The function `f(nodes)` in the MATLAB code in the appendix reflects the imposed boundary conditions discussed, as well as the function that calculates the summed force at each node. The force summation function `node_force(nodes,node)` is also shown in the appendix code block.

1.3 Function with Initial Equilibrium Condition

The state corresponding to the initial equilibrium condition where $V = 0$, and $g = 0$, has been verified to output a vector of all zeros. This is expected output since there would be no forces acting on any of the nodes as the lander and truss system would simply be “sitting” in free space. In this equilibrium case, the output is a 36×1 column vector of all zeros.

$$\mathbf{f}(\mathbf{u}) = \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \end{bmatrix}'$$

Note: *this column vector is shown as transposed to allow for better readability.*

2 Approximate Jacobian Matrix for Time-Step Restriction

By perturbing individual components of the state vector **u** from the initial configuration and repeatedly calling the **f** function, an approximate Jacobian Matrix, $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$, can be constructed. The eigenvalues of this 22×22 matrix can be found in the complex number plane and used to graphically determine the time-step restriction for the time stepping methods used.

2.1 Constructed Jacobian Matrix

The $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix can be constructed by perturbing individual components of the state vector **u** by a small amount of 1×10^{-6}. This perturbation will be referred to as h. The matrix will be constructed with the following form:

$$\frac{\delta \mathbf{f}}{\delta \mathbf{u}} = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} & \frac{\delta f_1}{\delta x_3} & \cdots & \frac{\delta f_1}{\delta x_n} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} & \frac{\delta f_2}{\delta x_3} & \cdots & \frac{\delta f_2}{\delta x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\delta f_n}{\delta x_1} & \frac{\delta f_n}{\delta x_2} & \frac{\delta f_n}{\delta x_3} & \cdots & \frac{\delta f_n}{\delta x_n} \end{bmatrix} \tag{13}$$

To find each individual derivative in the matrix Newton's difference quotient will be used. Using a limit as the h value approaches zero, the limit of the derivative of the function is defined as:

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \tag{14}$$

In this case, each part of the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix will need to filled in using this derivative function. In MATLAB, a function called `dfdu_matrix(nodes)` was created to output the 22×22 matrix using an input of the initial node state.

2.2 Eigenvalues Plot of the Jacobian Matrix

The eigenvalues of the now populated 22×22 matrix created using perturbed components can be calculated using the built-in MATLAB function `eig(matrix)`. The eigenvalue output of this function can then be plotted on imaginary versus real axes. Shown below is the plot of the eigenvalues for the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix.

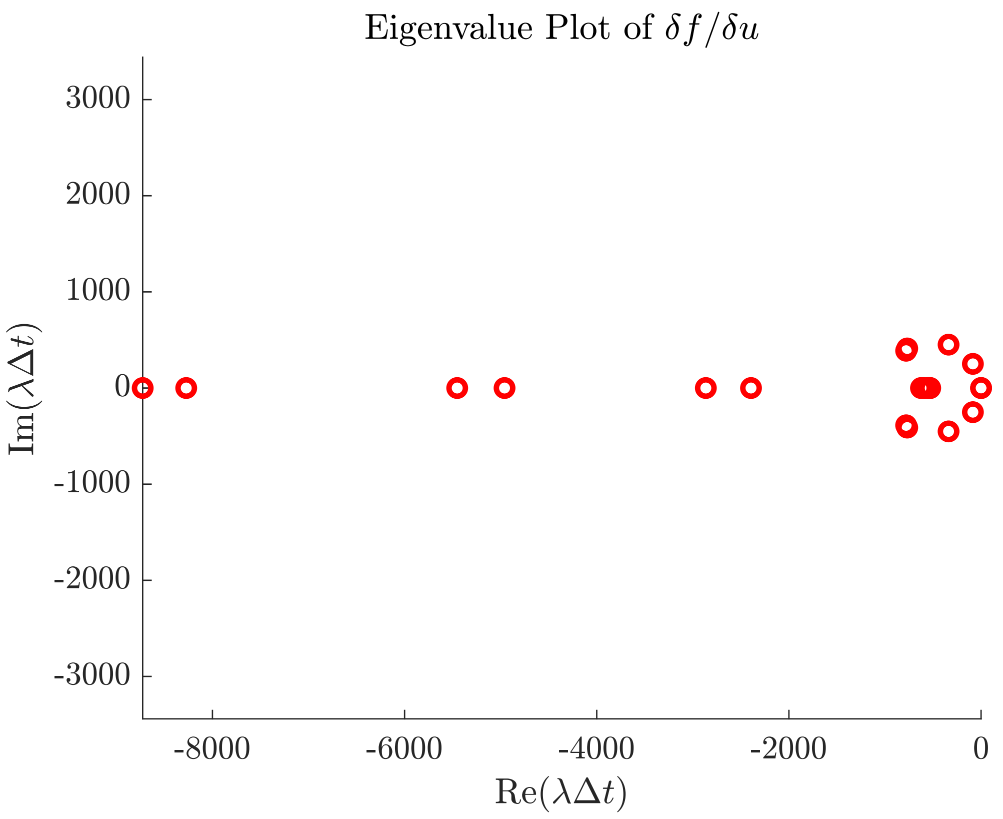

Figure 3: *Plot of the derivative matrix showing the real and imaginary parts of the eigenvalues used to determine the scaling of the maximum time step.*

As seen in the above figure, the smallest real root on the plot is -8726. This value will be important in conjunction with the roots of each eigenvalue stability equation for the numerical methods Forward Euler, Adams Bashforth, and Fourth Order Runge-Kutta. The value will be used in the determination of scaling for the eigenvalues to place them into a stable regime to calculate the maximum allowable time step for the schemes.

2.3 Time-Step Restriction for Numerical Schemes

To find the maximum allowable time step for the numerical schemes in the case of the lunar lander dynamics, the eigenvalue stability plot must first be generated for each scheme. The plot will allow for the scaling of the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ eigenvalues inside the stable regime.

2.3.1 Forward Euler Numerical Scheme Stability

The eigenvalue stability of Forward Euler is given by the following equation:

$$\lambda \Delta t = e^{i\theta} - 1 \tag{15}$$

The equation can be plotted by specifying a θ which ranges from 0 to 2π. Much like the above figure, the real and imaginary parts of the equation are plotted to reveal a stability region. By observing that the smallest real root available on the Forward Euler contour as -2, a scaling factor can be created in order to fit the eigenvalues of the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix inside the stable regime.

$$scale_{FE} = \frac{-8726}{-2}$$

Applying this scaling factor to all the eigenvalues of $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$, the plot below can be created to show all values are now in the stable regime.

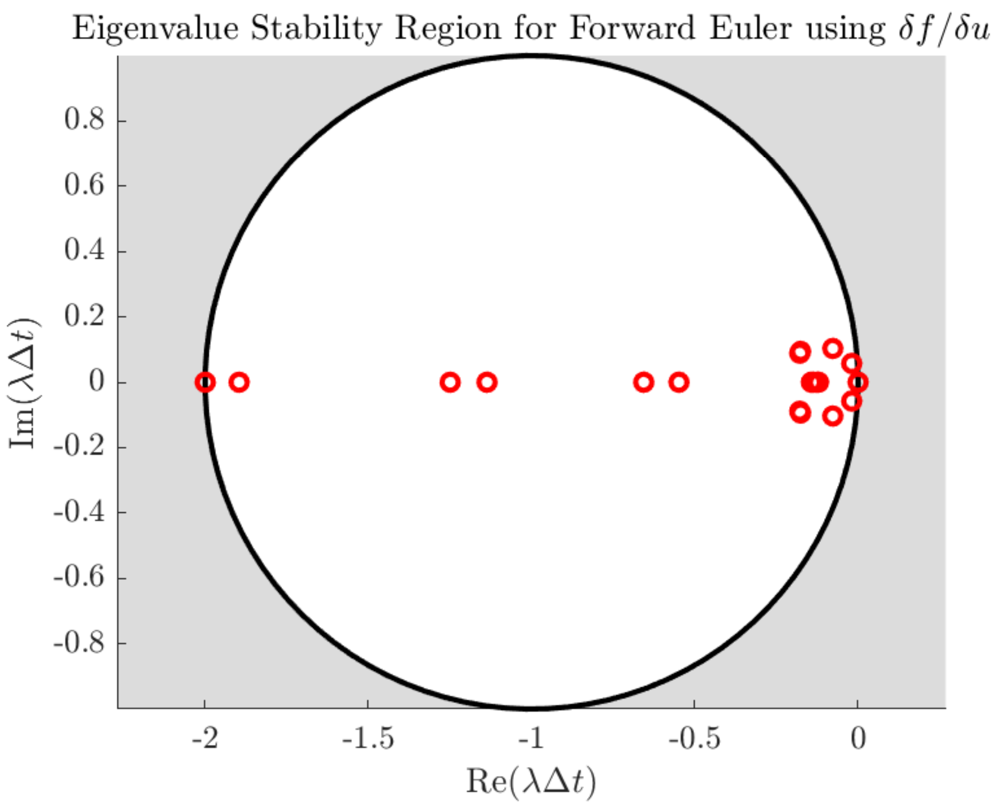

Figure 4: *Plot of the eigenvalues of Forward Euler with the applied scaling factor to fit the eigenvalues of the $\frac{\delta f}{\delta u}$ matrix inside the stability region.*

The inverse of the scaling factor also reveals the maximum allowable time step for the Forward Euler numerical scheme with the given dynamics.

$$\Delta t_{FE} = 2.2914 \times 10^{-4} \text{ s}$$

The restriction on the time step is it must be less than this value or else the time-stepping method will be unstable and result in incorrect computations.

2.3.2 Adams Bashforth Numerical Scheme Stability

The eigenvalue stability of Adams Bashforth is given by the following equation where $g = e^{i\theta}$:

$$\lambda \Delta t = (g-1)\frac{2g}{3g-1} \tag{16}$$

The equation can be plotted by specifying a θ which ranges from 0 to 2π. By observing that the smallest real root available on the Adams Bashforth contour as -0.9997, a scaling factor can be created in order to fit the eigenvalues of the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix inside the stable regime.

$$scale_{AB2} = \frac{-8726}{-0.9997}$$

Applying this scaling factor to all the eigenvalues of $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$, the plot below can be created to show all values are now in the stable regime.

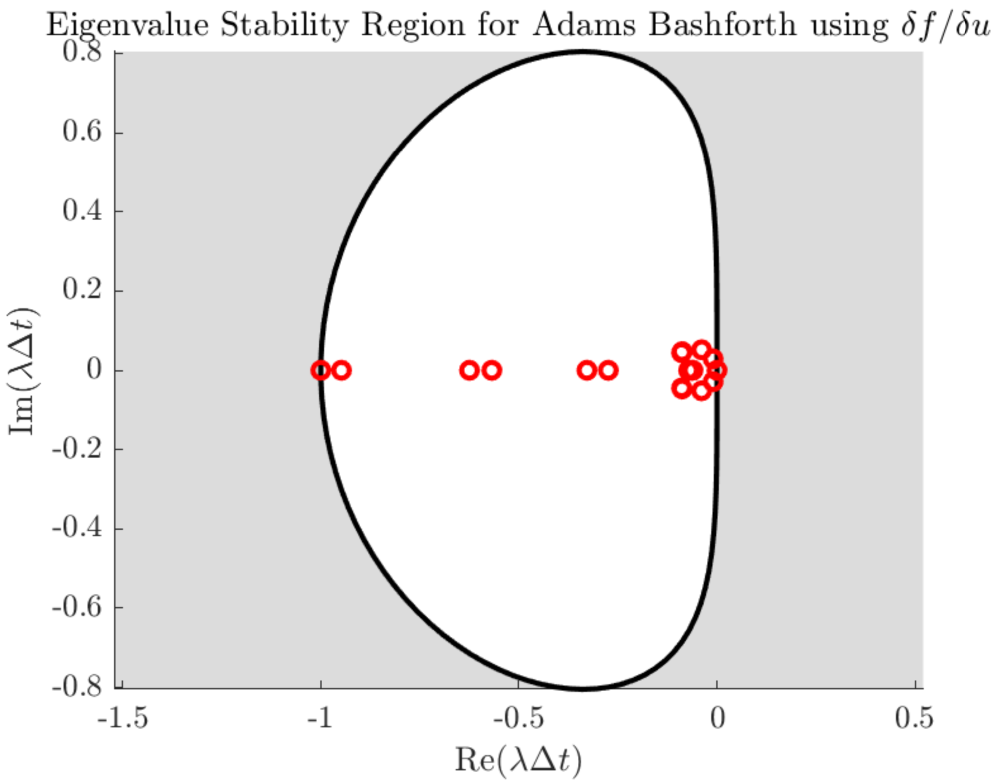

Figure 5: *Plot of the eigenvalues of Adams Bashforth with the applied scaling factor to fit the eigenvalues of the $\frac{\delta f}{\delta u}$ matrix inside the stability region.*

The inverse of the scaling factor also reveals the maximum allowable time step for the Adams Bashforth numerical scheme with the given dynamics.

$$\Delta t_{AB2} = 1.1456 \times 10^{-4} \text{ s}$$

The restriction on the time step is it must be less than this value or else the time-stepping method will be unstable and result in incorrect computations.

2.3.3 Runge-Kutta Numerical Scheme Stability

The eigenvalue stability of 4th Order Runge-Kutta is given by the following equation where $g = e^{i\theta}$:

$$g = 1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t) + \frac{1}{6}(\lambda\Delta t)^3 + \frac{1}{24}(\lambda\Delta t)^4 \tag{17}$$

The equation must be numerically solved for $\lambda\Delta t$ at a θ which ranges from 0 to 2π. Once the system of four equation is solved, the stability plot can be made by plotting the real and imaginary parts of $\lambda\Delta t$. By observing that the smallest real root available on the Runge-Kutta contour as -2.7853, a scaling factor can be created in order to fit the eigenvalues of the $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$ matrix inside the stable regime.

$$scale_{RK4} = \frac{-8726}{-2.7853}$$

Applying this scaling factor to all the eigenvalues of $\frac{\delta \mathbf{f}}{\delta \mathbf{u}}$, the plot below can be created to show all values are now in the stable regime.

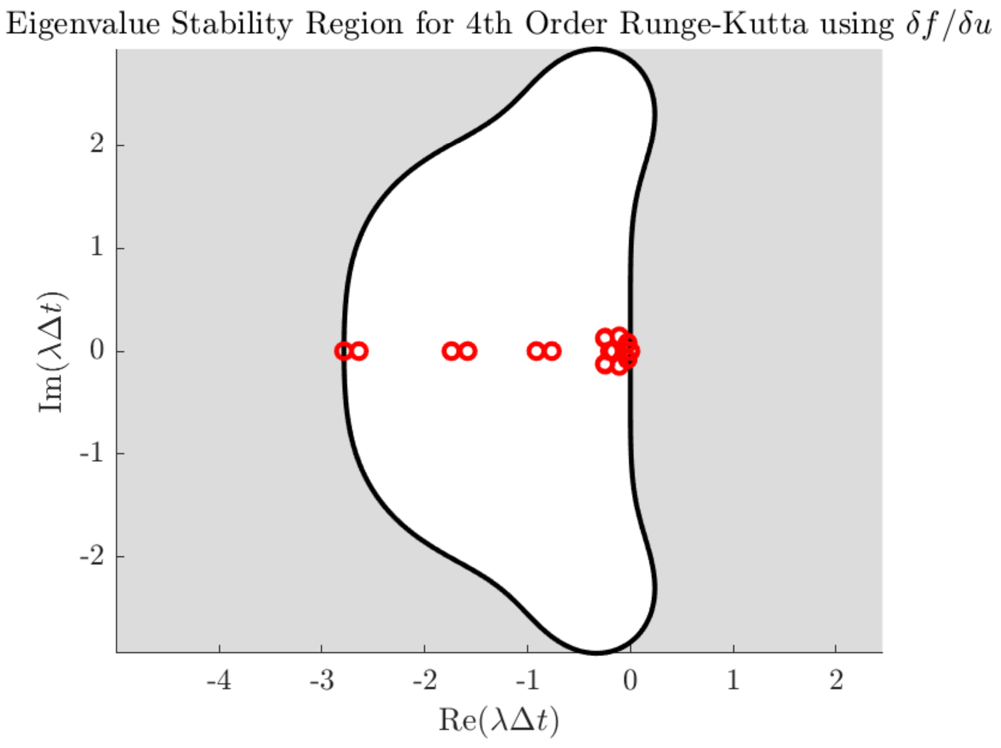

Figure 6: *Plot of the eigenvalues of Runge-Kutta with the applied scaling factor to fit the eigenvalues of the $\frac{\delta f}{\delta u}$ matrix inside the stability region.*

The inverse of the scaling factor also reveals the maximum allowable time step for the Runge-Kutta numerical scheme with the given dynamics.

$$\Delta t_{RK4} = 3.1919 \times 10^{-4} \text{ s}$$

The restriction on the time step is it must be less than this value or else the time-stepping method will be unstable and result in incorrect computations.

3 Implementation of Forward Euler (FE), Adams Bashforth (AB2), and Runge Kutta (RK4)

The implementation of Forward Euler, second-order Adams-Bashforth, and fourth-order Runge-Kutta time integration is detailed in this section. The verification of the maximum time steps calculated in the previous section will take place to ensure stability of the methods. For comparison purposes, a plot of the initial configuration of the lunar lander truss is shown below. This will be used to show that the implementation of these schemes results in a convergent and stable solution.

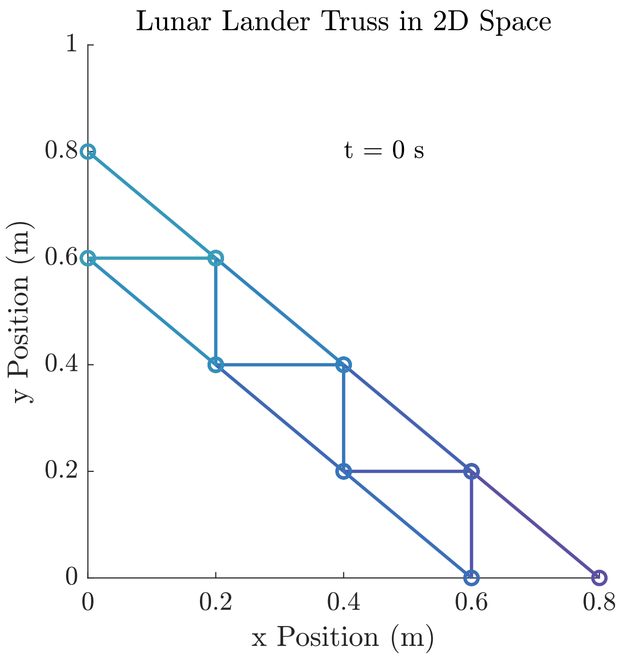

Figure 7: *Plot of the lunar lander truss at $T = 0$ before any numerical scheme is applied in order to show a comparison between dynamic and non-dynamic movement of the truss.*

3.1 Forward Euler Numerical Scheme Implementation

Forward Euler numerical method was implemented based off the following iteration equation:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t\mathbf{f}(\mathbf{u}^n)$$

Using this iteration equation a function called `FE(nodes,t)` was created in MATLAB to repeatedly call the function `f`. The input to the function is the state of the nodes in matrix form as described in the past details, as well as a vector of time that is based off of the time period of 0 to 0.5 seconds using the maximum time step decreased by 3% to ensure stability is maintained. Within the function calculates the Δt value by the difference of the first two time steps in the time vector.

The function then preallocates the output array using the size of the input node matrix and time step. Next, the node matrix is organized into a column state vector before that vector is used in the scheme. Since the `f` function takes input as the node matrix, a temporary matrix is created from the column state vector to be used as input. A `for` loop repeats these steps continually until the $N-1$ index is reached, where N is the maximum number of entries in the time vector.

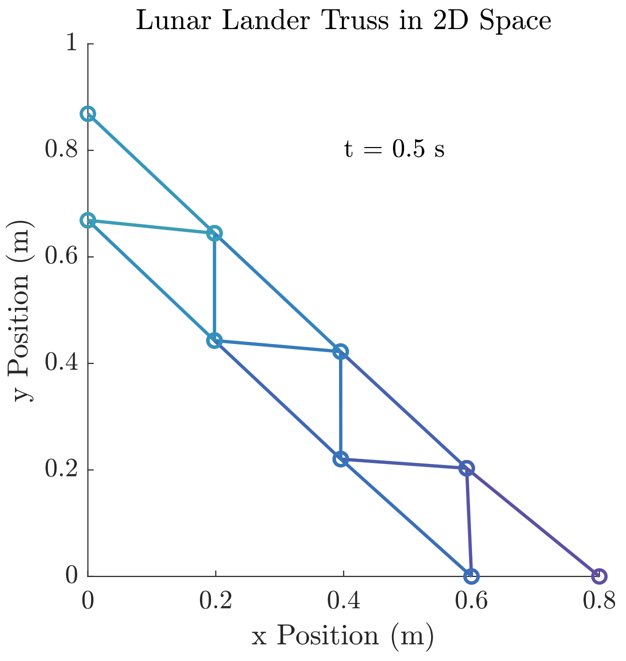

Figure 8: *Plot of the lunar lander truss at $T = 0.5$ after all iterations of the Forward Euler scheme have taken place showing that stability is maintained at the tweaked maximum time-step values.*

The plot of the lunar lander truss shows the iteration successfully converges and results in an expected result for a truss experiencing a load. Therefore, verification of the Forward Euler implementation is completed and can be relied on for the future simulations and results.

3.2 Adams Bashforth Numerical Scheme Implementation

Adams Bashforth numerical method was implemented based off the following iteration equations:

$$\mathbf{u}^{n+1} - \mathbf{u}^{n} = \Delta t\left(\frac{3}{2}\mathbf{f}(\mathbf{u}^{n}, t^{n}) - \frac{1}{2}\mathbf{f}(\mathbf{u}^{n-1}, t^{n-1})\right)$$

Using this iteration equation a function called `AB2(nodes,t)` was created in MATLAB to repeatedly call the function `f`. The input to the function is the state of the nodes in matrix form as described in the past details, as well as a vector of time that is based off of the time period of 0 to 0.5 seconds using the maximum time step decreased by 3% to ensure stability is maintained. Within the function calculates the Δt value by the difference of the first two time steps in the time vector.

The function then preallocates the output array using the size of the input node matrix and time step. Next, the node matrix is organized into a column state vector before that vector is used in the scheme. Since Adams Bashforth relies on two steps in time, an additional step in time was generated using one iteration of Forward Euler. This allows for the AB2 scheme to begin as there are now two known states. The `f` function takes input as the node matrix, therefore a temporary matrix is created from the column state vector to be used as input. A `for` loop repeats these steps continually until the $N-1$ index is reached, where N is the maximum number of entries in the time vector.

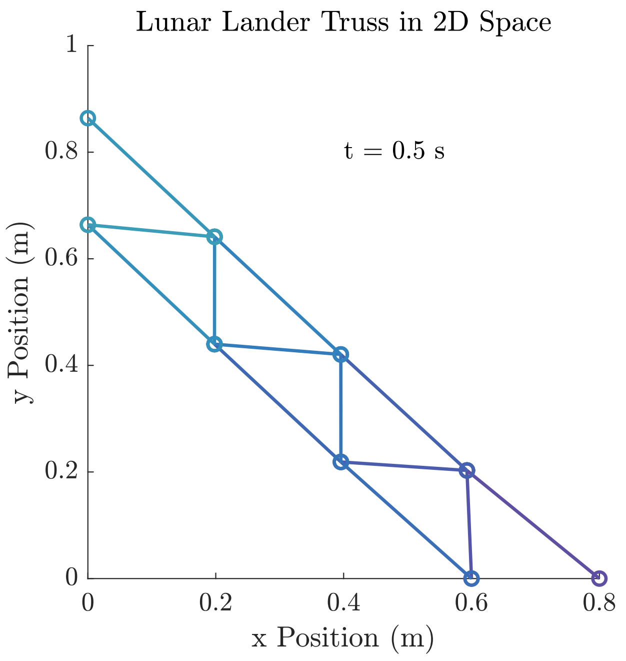

Figure 9: *Plot of the lunar lander truss at $T = 0.5$ after all iterations of the Adams Bashforth scheme have taken place showing that stability is maintained at the tweaked maximum time-step values.*

The plot of the lunar lander truss shows the iteration successfully converges and results in an expected result for a truss experiencing a load. Therefore, verification of the Adams Bashforth implementation is completed and can be relied on for the future simulations and results.

3.3 Runge-Kutta Numerical Scheme Implementation

Fourth Order Runge-Kutta Euler method can be implemented using the following update equation:

$$\begin{aligned}
\mathbf{f}_0 &= \mathbf{f}(\mathbf{u}^n, t^n) \\
\mathbf{f}_1 &= \mathbf{f}(\mathbf{u}^n + \frac{1}{2}\Delta t \mathbf{f}_0, t^n + \frac{\Delta t}{2}) \\
\mathbf{f}_2 &= \mathbf{f}(\mathbf{u}^n + \frac{1}{2}\Delta t \mathbf{f}_1, t^n + \frac{\Delta t}{2}) \\
\mathbf{f}_3 &= \mathbf{f}(\mathbf{u}^n + \Delta t \mathbf{f}_2, t^n + \Delta t) \\
\mathbf{u}^{n+1} &= \mathbf{u}^n + \frac{\Delta t}{6}(\mathbf{f}_0 + 2\mathbf{f}_1 + 2\mathbf{f}_2, +\mathbf{f}_3)
\end{aligned}$$

Using this iteration equation a function called `RK4(nodes,t)` was created in MATLAB to repeatedly call the function `f`. The input to the function is the state of the nodes in matrix form as described in the past details, as well as a vector of time that is based off of the time period of 0 to 0.5 seconds using the maximum time step decreased by 3% to ensure stability is maintained. Within the function calculates the Δt value by the difference of the first two time steps in the time vector.

The function then preallocates the output array using the size of the input node matrix and time step. Next, the node matrix is organized into a column state vector before that vector is used in the scheme. Since the `f` function takes input as the node matrix, a temporary matrix is created from the column state vector to be used as input. The fourth order Runge-Kutta method relies on multiple steps within one time iteration, which deemed it necessary to have each of these steps continually calculated. A `for` loop repeats these steps continually until the $N-1$ index is reached, where N is the maximum number of entries in the time vector.

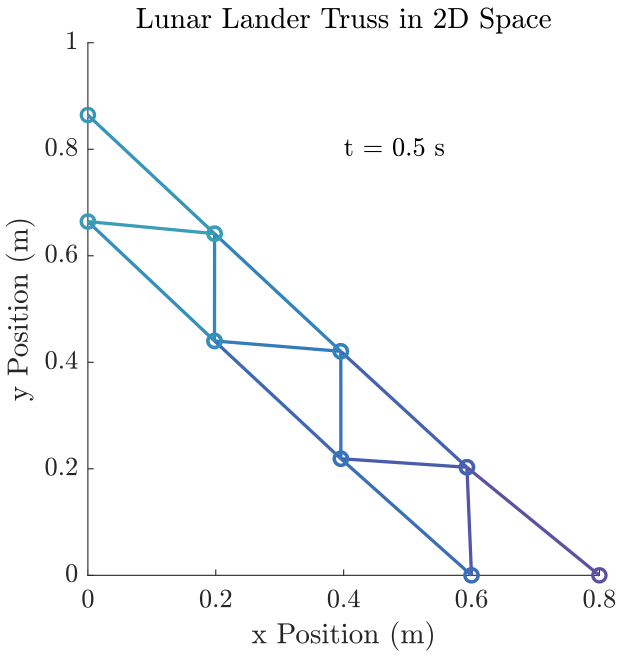

Figure 10: *Plot of the lunar lander truss at* $T = 0.5$ *after all iterations of the Runge-Kutta scheme have taken place showing that stability is maintained at the tweaked maximum time-step values.*

The plot of the lunar lander truss shows the iteration successfully converges and results in an expected result for a truss experiencing a load. Therefore, verification of the Runge-Kutta implementation is completed and can be relied on for the future simulations and results.

3.4 Evidence of Instability at Doubled Time-Steps

Using the height of the lunar lander (node 8) as a function of time, the instability of the numerical scheme when using a time-step double in value of the maximum allowable time-steps respective to the schemes can be seen. This doubled time-step value puts the eigenvalues of the Jacobian matrix out of the stable regime that was previously defined. Therefore, none of the iterative methods converge to a proper solution and the results are astronomically large numbers.

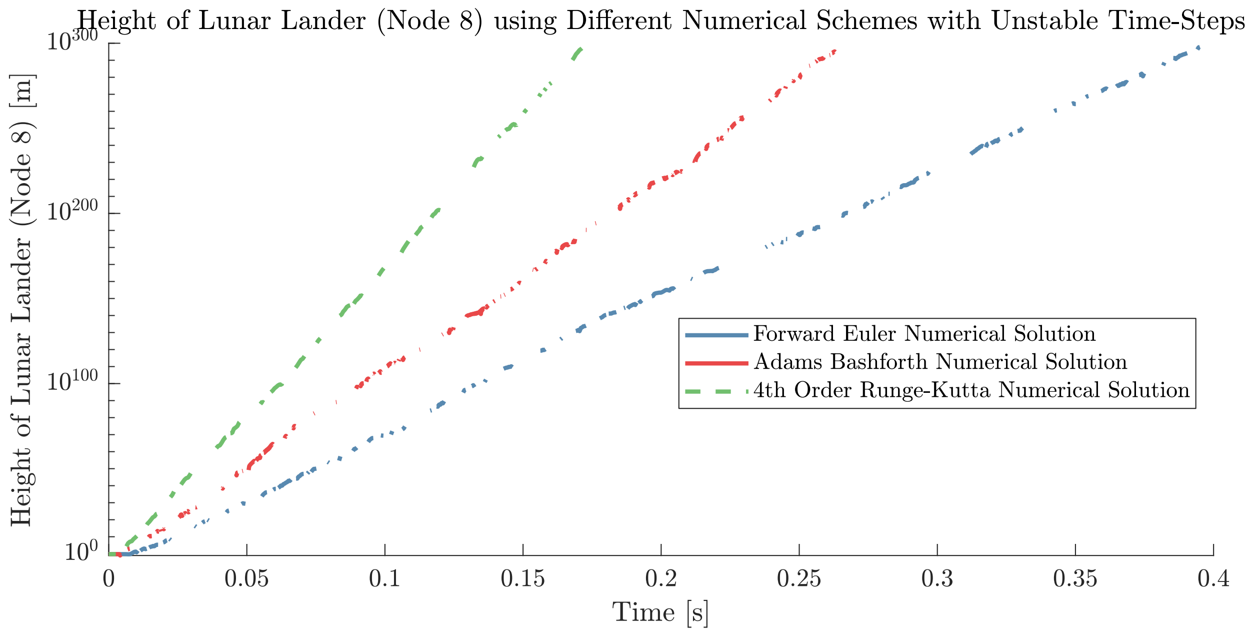

Figure 11: *Plot of the instability associated with a doubled maximum time-step.*

The above plot shows why it is important to determine the maximum allowable time-step and ensure the value used is some percent less than that for the numerical schemes to remain stable. The tweaked value used of a 3% decrease on this maximum time step will continue to be used to keep stability in the schemes.

4 Plot Height of Lander as a Function of Time

Using Forward Euler, Adams Bashforth, and fourth order Runge-Kutta, simulations can be performed up to $T = 0.5$ with the largest possible time step for each method tweaked by a 3% decrease in order to plot the height of the lander. This plot will be height as function of time. The previously discussed functions and implementations are used to find the height of node 8, which corresponds to the actual lander height. After simulations are ran, the y component of node 8 is pulled out of the state vector for every time iteration. This is done for each numerical scheme to create the plot shown below.

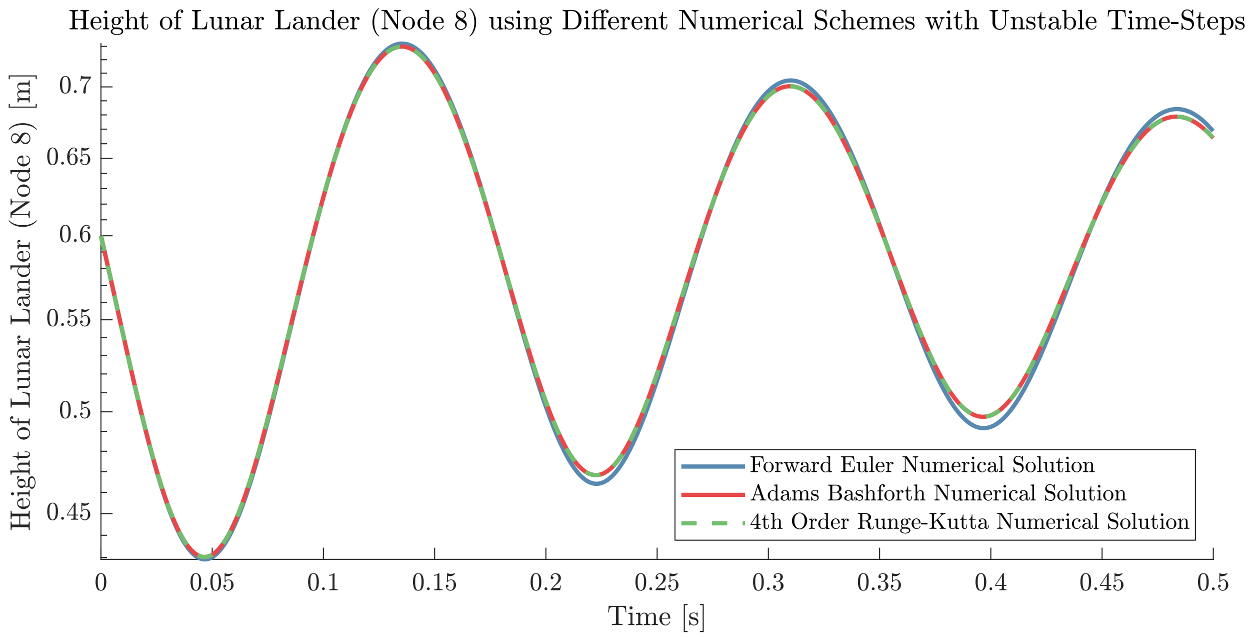

Figure 12: *Plot of the oscillatory lunar lander height from 0 to 0.5 seconds using all three numerical schemes to see the differences imposed by each.*

The differences in the methods can be seen in the plot. The Runge-Kutta and Adams Bashforth implementation are extremely similar and typically produce the same height value at each time interval; however, the Forward Euler differs in height, but still remains within expected error bounds for a first order numerical scheme. In the case a necessary method must be picked to use in future scenarios, Runge-Kutta would be best due to high convergence rate and accuracy as it is a fourth order approximation of the numerical solution.

5 Runge-Kutta Simulation of Lunar Lander Truss

The fourth order Runge-Kutta simulation can be ran at the largest possible time step with the aforementioned 3% tweak to ensure stability remains as the truss deforms. The nodes and links are plotted at $t = 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45$ seconds. These plots are seen below with the labeled time instances in the free space of each plot.

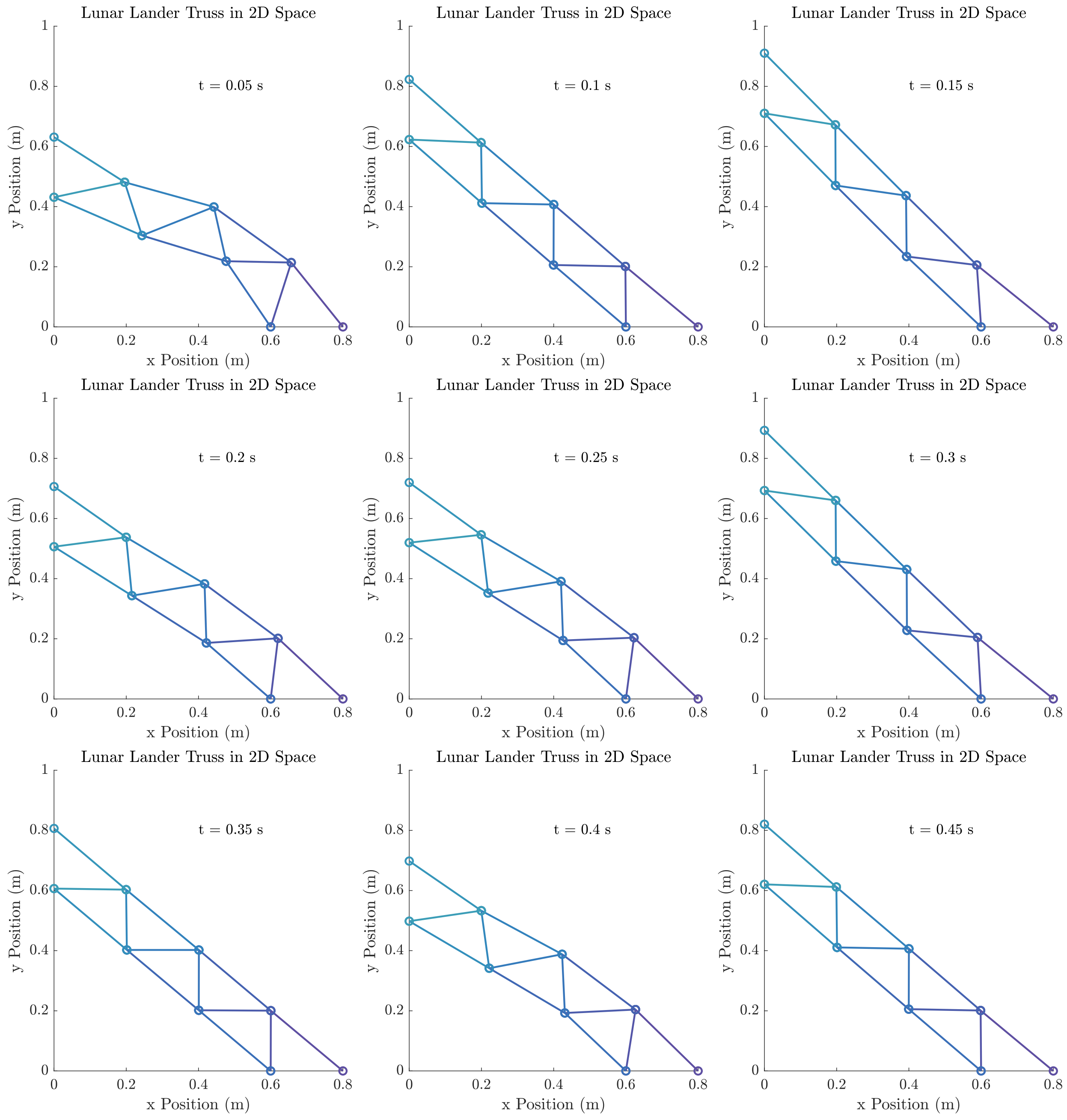

Figure 13: *The nodes and links of the Lunar Lander truss are plotted at $t = 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45$ seconds showing the deformation undergone in the system due to the dynamics in the x and y plane.*

The plots were generated using a function named `plotNodes` which takes in the node state matrix and the current time and outputs a figure of the plotted nodes and links and saves the image file for easy access and publishing. The state space column was converted to the node state matrix using the function `nodeSpace2nodes(nodeSpace)`.

The truss deforms as expected for a system taking a large load with a high downward velocity. The oscillatory nature of the truss is shown through the bouncing effect of the lunar lander and respective nodes. Looking at node 8 shows the height of the lander itself is changing as a function of time which corresponds to what was observed in the height as a function of time plot in the above section. This nature is common among dynamically damped systems and coincides with what is to be expected in a low gravity atmosphere, given link damping coefficient, and link stiffness.

Not included in this report due to the technicality of the file format was the creation of a .gif file encapsulating the motion of truss from 0.01 to 0.5 seconds with a frame taken every 0.05 seconds. The animated image showed the oscillatory motion of the truss and how the damping of the system begins as the oscillations slow down and decrease in magnitude. The .gif is a great way of observing the nature of the lunar lander truss as it allows for another visual aspect to verify the simulations were ran satisfactory and produced valid results.

6 Time-Step Convergence Study to Determine Theoretical Convergence Rate

Using the lander position at $T = 0.5$ as the output, a time-step convergence study can be completed to demonstrate that the theoretical convergence rate was achieved for each method. The exact solution was determined by using the Runge-Kutta function with 100,000 time steps in order to provide an extremely accurate solution. The other refined time steps for Forward Euler, Adams Bashfroth, and Runge-Kutta were then simulated and the output of the height of the lander at the final time was extracted and the error was calculated with the simple equation:

$$e_h = |h_{exact} - h_{simulated}| \tag{18}$$

The error was then plotted on a loglog scale to compare the time-steps with the calculated error. This plot is shown below and features a legend with the slope of each error line. The slope is equivalent to the approximate convergence rate of the utilized method.

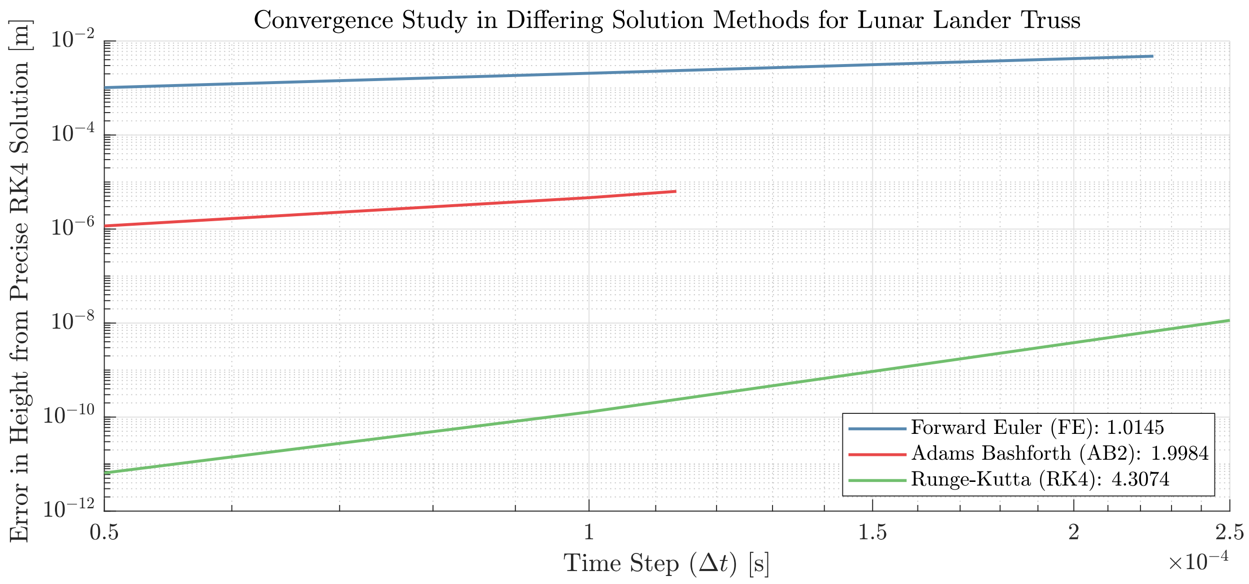

Figure 14: *Plot of the convergence of the numerical methods used to determine if the theoretical convergence rate was achieved.*

As shown by the figure, the convergence rate achieved for Forward Euler with the given dynamics was 1.0145, this coincides with the theoretical expected rate of a first order iterative scheme denoted by $\mathcal{O}(\Delta t)$. For the Adams Bashforth method the slope was observed as 1.9984, which is approximately equal to the second order theoretical solution of a second order scheme, $\mathcal{O}(\Delta t^2)$. The theoretical fourth order Runge-Kutta method is also shown as the numerical scheme reaches a slope of 4.3074 for the implemented dynamics, showcasing a convergence rate of $O(\Delta t^4)$. Therefore, all schemes have been demonstrated to output at the theoretical convergence rate

7 Conclusion

The following document details the process used to construct the simulation of a lunar lander using differential equations based on the physics of damped oscillations of stiff links connected to nodes in a truss. The stability regimes of the equations of motion were found for three numerical schemes, Forward Euler, Adams Bashforth, and Runge-Kutta. The maximum time-step value was then used to produce a simulated height plot as function of time to show the oscillatory nature of the dynamic equations of motion. The schemes were compared against themselves to ensure quality results were obtained. The fourth order Runge-Kutta scheme was used to generate plots of the truss at given time intervals to ensure the deformations coincide with physical intuition. The convergence of each implemented scheme was verified to ensure an approximate theoretical convergence order was achieved. The simulation of the lunar lander allowed for a real life application of a physical system using numerical methods on partial differential equations.

A Appendix

The MATLAB code ran to generate the plots and numerical results in this document is attached below. Some function calls have been commented out in order to save time when running the code as locally stored matrices of values resulted in much better performance run time.

A.1 List of Algorithms

A.2 MATLAB Script Implementation

```matlab
%Drake Rundell
%AE 423 - Project 1
%Due: 2/14/2020

clear; clc; close all;
set(groot,'DefaultTextInterpreter','LaTeX');
set(groot,'DefaultAxesTickLabelInterpreter','LaTeX');
set(groot,'DefaultLegendInterpreter','LaTeX');
set(groot,'defaultLegendFontSize',10);
set(groot,'defaultAxesFontSize',12);
set(groot,'defaultLineLineWidth',1.5);
set(groot,'defaultFigureColormap',linspecer);

%Parameters
T = 0.5;      %units: [s]
V = -6;     %units: [m/s]
d = 0.2;    %units: [m]

%Initial node Position and Velocity (x,y,u,v for all nine nodes)
nodes = [2*d,     d, 0, V; ...
         3*d,     d, 0, V; ...
           d,   2*d, 0, V; ...
         2*d,   2*d, 0, V; ...
           d,   3*d, 0, V; ...
         3*d,     0, 0, 0; ...
         4*d,     0, 0, 0; ...
           0,   3*d, 0, V; ...
           0,   4*d, 0, V];

%% Eigenvalue Stability for Schemes using df/du matrix
dfdu = dfdu_matrix(nodes);

%Get eigenvalues of df/du matrix
dfdu_eig = eig(dfdu);

```

```
%Plot eigenvalues of df/du matrix
figure
hold on
scatter(real(dfdu_eig),imag(dfdu_eig),'or','LineWidth',2)
axis equal
title('Eigenvalue Plot of $\delta f/\delta u$','Interpreter','LaTeX')
    ;
xlabel('Re($\lambda\Delta t$)','Interpreter','LaTeX');
ylabel('Im($\lambda\Delta t$)','Interpreter','LaTeX');
set(gcf, 'Color', 'w','Position',[200 200 800 400]);
export_fig Figures/eig_dfdu.eps -native

%Space out theta values for Stability Equations
theta = linspace(0,2*pi);

%Eigenvalue Stability for Forward Euler Scheme
lambda_dt_FE = exp(1i.*theta) - 1;

%Scale df/du matrix to fit in FE Scheme Eigenvalue Stability Region
scaleFE = min(real(dfdu_eig))/min(real(lambda_dt_FE));

%Plot scaled eigenvalues to plot of stable region
figure
hold on
fill(real(lambda_dt_FE),imag(lambda_dt_FE),[1,1,1],'Facecolor','w','
    EdgeColor','k','Linewidth',2);
scatter(real(dfdu_eig)/scaleFE,imag(dfdu_eig)/scaleFE,'or','LineWidth
    ',2)
axis equal
title('Eigenvalue Stability Region for Forward Euler using $\delta f
    /\delta u$','Interpreter','LaTeX');
xlabel('Re($\lambda\Delta t$)','Interpreter','LaTeX');
ylabel('Im($\lambda\Delta t$)','Interpreter','LaTeX');
set(gca,'Color',[0.8627 0.8627 0.8627])
set(gcf, 'Color', 'w','Position',[200 200 800 400]);
export_fig Figures/eig_FE.eps -native

%Time Restriction for Forward Euler
dt_FE = 1/scaleFE;

%Eigenvalue Stability for AB2 Scheme
g = exp(1i.*theta);
lambda_dt_AB2 = (g-1) .* 2.*g./(3.*g-1);

%Scale df/du matrix to fit in AdamsBashforth Scheme Eigenvalue
    Stability Region
scaleAB2 = min(real(dfdu_eig))/min(real(lambda_dt_AB2));

%Plot scaled eigenvalues to plot of stable region
figure
hold on
fill(real(lambda_dt_AB2),imag(lambda_dt_AB2),[1,1,1],'Facecolor','w',
    'EdgeColor','k','Linewidth',2);
scatter(real(dfdu_eig)/scaleAB2,imag(dfdu_eig)/scaleAB2,'or','
    LineWidth',2)
axis equal
title('Eigenvalue Stability Region for Adams Bashforth using $\delta
    f/\delta u$','Interpreter','LaTeX');
xlabel('Re($\lambda\Delta t$)','Interpreter','LaTeX');
```

```
ylabel('Im($\lambda\Delta t$)','Interpreter','LaTeX');
set(gca,'Color',[0.8627 0.8627 0.8627])
set(gcf, 'Color', 'w','Position',[200 200 800 400]);
export_fig Figures/eig_AB2.eps -native

%Time Restriction for RK4
dt_AB2 = 1/scaleAB2;

%Eigenvalue Stability for RK4 Scheme
syms lambda_dt
lambda_dt_RK4_1 = zeros(1,max(size(theta)));
lambda_dt_RK4_2 = zeros(1,max(size(theta)));
lambda_dt_RK4_3 = zeros(1,max(size(theta)));
lambda_dt_RK4_4 = zeros(1,max(size(theta)));

for i = 1:max(size(theta))
    eqn = exp(1i.*theta(i)) == 1 + lambda_dt + .5*lambda_dt^2 + 1/6 *
     lambda_dt^3 + 1/24*lambda_dt^4;
    sol = vpasolve(eqn,lambda_dt);
    lambda_dt_RK4_1(i) = sol(1);
    lambda_dt_RK4_2(i) = sol(2);
    lambda_dt_RK4_3(i) = sol(3);
    lambda_dt_RK4_4(i) = sol(4);
end
n = max(size(theta));
lambda_dt_RK4 = [lambda_dt_RK4_1(1:(n/2)) lambda_dt_RK4_2(((n/2)+1):
    end) ...
                 lambda_dt_RK4_3(1:(n/2)) lambda_dt_RK4_4(((n/2)+1):
    end) ...
                 lambda_dt_RK4_4(2:(n/2)) lambda_dt_RK4_3(((n/2)+1):
    end) ...
                 lambda_dt_RK4_2(2:(n/2)) lambda_dt_RK4_1(((n/2)+1):
    end)];

%Scale df/du matrix to fit in RK4 Scheme Eigenvalue Stability Region
scaleRK4 = min(real(dfdu_eig))/min(real(lambda_dt_RK4));

%Plot scaled eigenvalues to plot of stable region
figure
hold on
fill(real(lambda_dt_RK4),imag(lambda_dt_RK4),[1,1,1],'Facecolor','w',
    'EdgeColor','k','Linewidth',2);
scatter(real(dfdu_eig)/scaleRK4,imag(dfdu_eig)/scaleRK4,'or','
    LineWidth',2)
axis equal
title('Eigenvalue Stability Region for 4th Order Runge-Kutta using $\
    delta f/\delta u$','Interpreter','LaTeX');
xlabel('Re($\lambda\Delta t$)','Interpreter','LaTeX');
ylabel('Im($\lambda\Delta t$)','Interpreter','LaTeX');
set(gca,'Color',[0.8627 0.8627 0.8627])
set(gcf, 'Color', 'w','Position',[200 200 800 400]);
export_fig Figures/eig_RK4.eps -native

%Time Restriction for RK4
dt_RK4 = 1/scaleRK4;


%% Height of Lander with FE, AB2, and RK4
```

```
%Run simulation with FE using largest time step with 3.0% adjustment
    to
%prevent instability
dt_FE = dt_FE * 0.97;
t_FE = 0:dt_FE:T;
nodeSpaceFE = FE(nodes,t_FE);
h_FE = nodeSpaceFE(30,:);

%Final Result for Verification
nodesFE_final = nodeSpace2nodes(nodeSpaceFE);
figure
plotNodes(nodesFE_final,T);
export_fig 'Figures/final_FE.eps'

%Run simulation with AB2 using largest time step with 3.0% adjustment
    to
%prevent instability
dt_AB2 = dt_AB2 * 0.97;
t_AB2 = 0:dt_AB2:T;
nodeSpaceAB2 = AB2(nodes,t_AB2);
h_AB2 = nodeSpaceAB2(30,:);

%Final Result for Verification
nodesAB2_final = nodeSpace2nodes(nodeSpaceAB2);
figure
plotNodes(nodesAB2_final,T);
export_fig 'Figures/final_AB2.eps'

%Run simulation with RK4 using largest time step with 3.0% adjustment
    to
%prevent instability
dt_RK4 = dt_RK4 * 0.97;
t_RK4 = 0:dt_RK4:T;
nodeSpaceRK4 = RK4(nodes,t_RK4);
h_RK4 = nodeSpaceRK4(30,:);

%Final Result for Verification
nodesRK4_final = nodeSpace2nodes(nodeSpaceRK4);
figure
plotNodes(nodesRK4_final,T);
export_fig 'Figures/final_RK4.eps'

figure
axes('NextPlot','replacechildren','ColorOrder',linspecer(3));
semilogy(t_FE,h_FE,t_AB2,h_AB2,t_RK4,h_RK4,'--','Linewidth',2);
title('Height of Lunar Lander (Node 8) using Different Numerical
    Schemes with Unstable Time-Steps');
legend('Forward Euler Numerical Solution','Adams Bashforth Numerical
    Solution','4th Order Runge-Kutta Numerical Solution','Location','
    Best')
xlabel('Time [s]');
ylabel('Height of Lunar Lander (Node 8) [m]');
set(gcf, 'Color', 'w','Position',[200 200 900 400]);
export_fig Figures/lander_height.eps -native

%% Simulations with FE, AB2, and RK4 up to T = 0.5

%Initial Node Position
figure
```

```
plotNodes(nodes,0);

%RK4 Simulation for T = 0.5 at varying times
times = 0.05:0.05:0.5;
set(0,'DefaultFigureVisible','off')
for i = 1:max(size(times))
    nodeSpaceRK4 = RK4(nodes,0:dt_RK4:times(i));
    nodesRK4 = nodeSpace2nodes(nodeSpaceRK4);
    figure
    plotNodes(nodesRK4,times(i));
end
set(0,'DefaultFigureVisible','on')

%% Error Analysis

%First run RK4 with high resolution to define "correct" answer for
    height at
%T = 5;
t_RK4 = linspace(0,T,100000);
%nodeSpaceRK4 = RK4(nodes,t_RK4); disabled for time reaons
nodeSpaceRK4 = cell2mat(struct2cell(load('RK4_fine_simulation.mat')))
    ;
h_actual = nodeSpaceRK4(30,:);
h_actual = h_actual(end);

%Three Time Steps
t_3_FE = 0:2.2411e-4:T;
t_3_AB2 = 0:1.133e-4:T;
t_3_RK4 = 0:2.5e-4:T;
t_2 = 0:0.0001:T;
t_1 = 0:0.00005:T;

%FE Error Calculations at Three Points
nodeSpaceFE_3 = FE(nodes,t_3_FE);
h_FE_3 = nodeSpaceFE_3(30,:);
error_FE_3 = abs(h_actual - h_FE_3(end));

nodeSpaceFE_2 = FE(nodes,t_2);
h_FE_2 = nodeSpaceFE_2(30,:);
error_FE_2 = abs(h_actual - h_FE_2(end));

nodeSpaceFE_1 = FE(nodes,t_1);
h_FE_1 = nodeSpaceFE_1(30,:);
error_FE_1 = abs(h_actual - h_FE_1(end));

%AB2 Error Calculations at Three Points
nodeSpaceAB2_3 = AB2(nodes,t_3_AB2);
h_AB2_3 = nodeSpaceAB2_3(30,:);
error_AB2_3 = abs(h_actual - h_AB2_3(end));

nodeSpaceAB2_2 = AB2(nodes,t_2);
h_AB2_2 = nodeSpaceAB2_2(30,:);
error_AB2_2 = abs(h_actual - h_AB2_2(end));

nodeSpaceAB2_1 = AB2(nodes,t_1);
h_AB2_1 = nodeSpaceAB2_1(30,:);
error_AB2_1 = abs(h_actual - h_AB2_1(end));

%RK4 Error Calculations at Three Points
```

```
nodeSpaceRK4_3 = RK4(nodes,t_3_RK4);
h_RK4_3 = nodeSpaceRK4_3(30,:);
error_RK4_3 = abs(h_actual - h_RK4_3(end));

nodeSpaceRK4_2 = RK4(nodes,t_2);
h_RK4_2 = nodeSpaceRK4_2(30,:);
error_RK4_2 = abs(h_actual - h_RK4_2(end));

nodeSpaceRK4_1 = RK4(nodes,t_1);
h_RK4_1 = nodeSpaceRK4_1(30,:);
error_RK4_1 = abs(h_actual - h_RK4_1(end));

%Put time steps in vector for plotting
timestepsFE = [2.2411e-4 0.0001 0.00005];
timestepsAB2 = [1.133e-4 0.0001 0.00005];
timestepsRK4 = [2.5e-4 0.0001 0.00005];

%Calculate Slope on Loglog Plot
slope_FE = log10(error_FE_2/error_FE_1) / log10(0.0001/0.00005);
slope_AB2 = log10(error_AB2_2/error_AB2_1) / log10(0.0001/0.00005);
slope_RK4 = log10(error_RK4_2/error_RK4_1) / log10(0.0001/0.00005);

figure
axes('NextPlot','replacechildren','ColorOrder',linspecer(3));
loglog(timestepsFE,[error_FE_3 error_FE_2 error_FE_1],timestepsAB2,[
    error_AB2_3 error_AB2_2 error_AB2_1],timestepsRK4,[error_RK4_3
    error_RK4_2 error_RK4_1]);
title('Convergence Study in Differing Solution Methods for Lunar
    Lander Truss','Interpreter','LaTeX');
xlabel('Time Step ($\Delta t$) [s]','Interpreter','LaTeX');
ylabel('Error in Height from Precise RK4 Solution [m] ','Interpreter'
    ,'LaTeX');
legend(['Forward Euler (FE): ', num2str(slope_FE)],['Adams Bashforth
    (AB2): ', num2str(slope_AB2)],['Runge-Kutta (RK4): ', num2str(
    slope_RK4)],'Interpreter','LaTeX','Location','Southeast');
grid on
grid minor
set(gcf, 'Color', 'w','Position',[200 200 1000 400]);
export_fig Figures/lander_error.eps -native
```

Algorithm 1: *MATLAB Script used to call implemented functions.*

A.3 MATLAB Function Implementation

```matlab
function nodes = nodeSpace2nodes(nodeSpace)
    %Convert column state space vector to node matrix in order to
    plot
    nodes = [nodeSpace(1:4,length(nodeSpace))'; nodeSpace(5:8,length(
    nodeSpace))'; nodeSpace(9:12,length(nodeSpace))'; nodeSpace
    (13:16,length(nodeSpace))'; nodeSpace(17:20,length(nodeSpace))';
    ...
              nodeSpace(21:24,length(nodeSpace))'; nodeSpace(25:28,
    length(nodeSpace))'; nodeSpace(29:32,length(nodeSpace))';
    nodeSpace(33:36,length(nodeSpace))'];
end
```

Algorithm 2: *Function responsible for converting column space vector into node matrix*

```matlab
function dfdu = dfdu_matrix(nodes)
    %Small Perturbation
    h = 1e-6;
    %Preallocate 22x22 Matrix
    dfdu = zeros(22);
    %Take initial derivative
    f_0 = f(nodes);
    %Begin array indexing
    n = 1;
    for i = 1:5
        for j = 1:4
            %Store node input as temporary to keep it from changing
    at
            %every iteration
            node_temp = nodes;
            %Add perturbation to temp node matrix
            node_temp(i,j) = node_temp(i,j) + h;
            %Put temp node matrix into f function
            df = f(node_temp);
            %Take derivative using classic equation with h
    perturbation
            dfdu(:,n) = (df(1:22) - f_0(1:22))/h;
            n = n + 1;
        end
    end
end
```

Algorithm 3: *Function responsible for outputing Jacobian matrix*

```
function output = RK4(nodes,t)
    %Setup RK4
    dt = t(2) - t(1);
    u = zeros(max(size(nodes))*min(size(nodes)),max(size(t)));
    N = max(size(t));

    %Organize column vector from node matrix
    u(:,1) = [nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
    nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)'; nodes(9,:)'];

    %Perform RK4 method using column vector form
    for i = 1:(N-1)
        temp = [u(1:4,i)'; u(5:8,i)'; u(9:12,i)'; u(13:16,i)'; u
    (17:20,i)'; ...
                u(21:24,i)'; u(25:28,i)'; u(29:32,i)'; u(33:36,i)'];
        k_0 = f(temp);

        temp_k_1 = [k_0(1:4)'; k_0(5:8)'; k_0(9:12)'; k_0(13:16)';
    k_0(17:20)'; ...
                k_0(21:24)'; k_0(25:28)'; k_0(29:32)'; k_0(33:36)'];
        k_1 = f(temp + 0.5 .* dt .* temp_k_1);

        temp_k_2 = [k_1(1:4)'; k_1(5:8)'; k_1(9:12)'; k_1(13:16)';
    k_1(17:20)'; ...
                k_1(21:24)'; k_1(25:28)'; k_1(29:32)'; k_1(33:36)'];
        k_2 = f(temp + 0.5 .* dt .* temp_k_2);

        temp_k_3 = [k_2(1:4)'; k_2(5:8)'; k_2(9:12)'; k_2(13:16)';
    k_2(17:20)'; ...
                k_2(21:24)'; k_2(25:28)'; k_2(29:32)'; k_2(33:36)'];
        k_3 = f(temp + dt .* temp_k_3);

        u(:,i+1) = u(:,i) + dt/6 .* (k_0 + 2.*k_1 + 2.*k_2 + k_3);
    end

    %output is matrix of x1 y1 v1 u1 for each node in column form for
     each
    %time step
    output = u;
end
```

Algorithm 4: *Function responsible for Runge-Kutta Implementation*

```
function output = AB2(nodes, t)
    %Setup method
    dt = t(2) - t(1);
    u = zeros(max(size(nodes))*min(size(nodes)),max(size(t)));
    N = max(size(t));

    %Organize column vector from node matrix
    u(:,1) = [nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
    nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)'; nodes(9,:)'];
    temp = [u(1:4,1)'; u(5:8,1)'; u(9:12,1)'; u(13:16,1)'; u(17:20,1)
    '; ...
                u(21:24,1)'; u(25:28,1)'; u(29:32,1)'; u(33:36,1)'];

    %Take initial step using Forward Euler
    u(:,2) = u(:,1) + dt .* f(temp);

    %Perform AB2 method using column vector form
    for i = 2:(N-1)
        temp1 = temp;
        temp = [u(1:4,i)'; u(5:8,i)'; u(9:12,i)'; u(13:16,i)'; u
    (17:20,i)'; ...
                u(21:24,i)'; u(25:28,i)'; u(29:32,i)'; u(33:36,i)'];
        u(:,i+1) = u(:,i) + dt .* (1.5 .* f(temp) - .5 .*f (temp1));
    end

    %output is matrix of x1 y1 v1 u1 for each node in column form for
     each
    %time step
    output = u;
end
```

Algorithm 5: *Function responsible for Adams Bashforth Implementation*

```
function output = FE(nodes,t)
    %Setup Forward Euler
    dt = t(2) - t(1);
    u = zeros(max(size(nodes))*min(size(nodes)),max(size(t)));
    N = max(size(t));

    %Organize node matrix into column vector
    u(:,1) = [nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
    nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)'; nodes(9,:)'];

    %Implement Forward Euler Scheme
    for i = 1:(N-1)
        temp = [u(1:4,i)'; u(5:8,i)'; u(9:12,i)'; u(13:16,i)'; u
    (17:20,i)'; ...
                u(21:24,i)'; u(25:28,i)'; u(29:32,i)'; u(33:36,i)'];
        u(:,i+1) = u(:,i) + dt .* f(temp);
    end

    %output is matrix of x1 y1 v1 u1 for each node in column form for
     each
    %time step
    output = u;
end
```

Algorithm 6: *Function responsible for Forward Euler Implementation*

```
function udot = f(nodes)
    %Constants
    m_node = 0.1;
    M = 100;
    g = 1.625;

    %Equation of Motions for all nine nodes organized into u,v,dot u,
     and dot v form
    udot = [[nodes(1,3:4)'; node_force(nodes,1)'/m_node] - [0 0 0 g
    ]';
            [nodes(2,3:4)'; node_force(nodes,2)'/m_node] - [0 0 0 g
    ]';
            [nodes(3,3:4)'; node_force(nodes,3)'/m_node] - [0 0 0 g
    ]';
            [nodes(4,3:4)'; node_force(nodes,4)'/m_node] - [0 0 0 g
    ]';
            [nodes(5,3:4)'; node_force(nodes,5)'/m_node] - [0 0 0 g
    ]';
            [0 0 0 0]';
            [0 0 0 0]';
            ([(nodes(9,3:4))' .* [0;1] ; 8.*(node_force(nodes,8)./M)'
     .* [0;1]] - [0 0 0 g]');
            ([(nodes(9,3:4))' .* [0;1] ; 8.*(node_force(nodes,9)./M)'
     .* [0;1]] - [0 0 0 g]')];
end
```

Algorithm 7: *Function responsible for output of accelerlation and velocity state space*

```
function sum = node_force(nodes, node)
    %Constants
    gamma = 200;
    k = 10^5;
    sum = [0 0];
    %Each node (1-9) with connections
    links = [2 3 4 6;
            1 6 4 7;
            1 4 5 8;
            1 2 3 5;
            3 4 8 9;
            1 2 NaN NaN;
            2 NaN NaN NaN;
            3 5 NaN NaN;
            5 NaN NaN NaN];

    %Test if connecting node exists to given node, and if it does
    loop
    %through to gather all forces
    for i = 1:max(size(nodes))
        if ismember(node, links(i,:)) == 1
            if (node == 1 && (i == 2 || i == 4))
                d = 0.2;
            elseif (node == 2 && (i == 6 || i == 1))
                d = 0.2;
            elseif (node == 3 && (i == 4|| i == 5))
                d = 0.2;
            elseif (node == 4 && (i == 3 || i == 1))
                d = 0.2;
            elseif (node == 5 && (i == 3 || i == 8))
```

```
                d = 0.2;
            elseif (node == 6 && (i == 2))
                d = 0.2;
            elseif (node == 8 && (i == 5))
                d = 0.2;
            else
                d = 0.2 * sqrt(2);
            end

            %Equation for link forces from project statement
            x1 = nodes(node,1:2);
            x2 = nodes(i,1:2);

            v1 = nodes(node,3:4);
            v2 = nodes(i,3:4);

            e = (x2 - x1) ./ norm(x2 - x1);

            %Delta l
            dl =  norm(x2 - x1) - d;
            dl(abs(dl) < 1e-16) = 0;

            %Dot Delta l
            ddl = dot((v2 - v1), e);
            ddl(abs(ddl) < 1e-16) = 0;

            Fij = (k .* dl + gamma .* ddl) .* e;

            %Sum all forces
            if (norm(Fij) > 1e-16)
                sum = sum + Fij;
            end
        end
    end
end
```

Algorithm 8: *Function responsible for summation of forces on a node*

```
function plotNodes(nodes,t)
    %Make each link
    L_72 = [nodes(7,1:2); nodes(2,1:2)];
    L_62 = [nodes(6,1:2); nodes(2,1:2)];
    L_12 = [nodes(1,1:2); nodes(2,1:2)];
    L_42 = [nodes(4,1:2); nodes(2,1:2)];
    L_13 = [nodes(1,1:2); nodes(3,1:2)];
    L_16 = [nodes(1,1:2); nodes(6,1:2)];
    L_14 = [nodes(1,1:2); nodes(4,1:2)];
    L_43 = [nodes(4,1:2); nodes(3,1:2)];
    L_45 = [nodes(4,1:2); nodes(5,1:2)];
    L_35 = [nodes(3,1:2); nodes(5,1:2)];
    L_38 = [nodes(3,1:2); nodes(8,1:2)];
    L_59 = [nodes(5,1:2); nodes(9,1:2)];
    L_58 = [nodes(5,1:2); nodes(8,1:2)];

    %Organize into x and y space
    x = [L_72(:,1) L_62(:,1) L_12(:,1) L_42(:,1) L_13(:,1) L_16(:,1)
    ...
        L_14(:,1) L_43(:,1) L_45(:,1) L_35(:,1) L_38(:,1) L_59(:,1)
    L_58(:,1)];
    y = [L_72(:,2) L_62(:,2) L_12(:,2) L_42(:,2) L_13(:,2) L_16(:,2)
    ...
        L_14(:,2) L_43(:,2) L_45(:,2) L_35(:,2) L_38(:,2) L_59(:,2)
    L_58(:,2)];

    axes('NextPlot','replacechildren','ColorOrder',linspecer(max(size
    (x))*7));
    plot(x,y,'-o')
    title('Lunar Lander Truss in 2D Space');
    xlabel('x Position (m)');
    ylabel('y Position (m)');
    xlim([0 0.8]);
    ylim([0 1]);
    set(gcf, 'Color', 'w','Position',[200 200 400 400]);
    text(0.4,0.8,['t = ' num2str(t) ' s'],'Fontsize',12,'Interpreter'
    ,'LaTeX');
    export_fig (['Figures/lander_truss_t_', num2str(t*100),'.eps'])
end
```

Algorithm 9: *Function responsible for plotting truss layout in 2D*