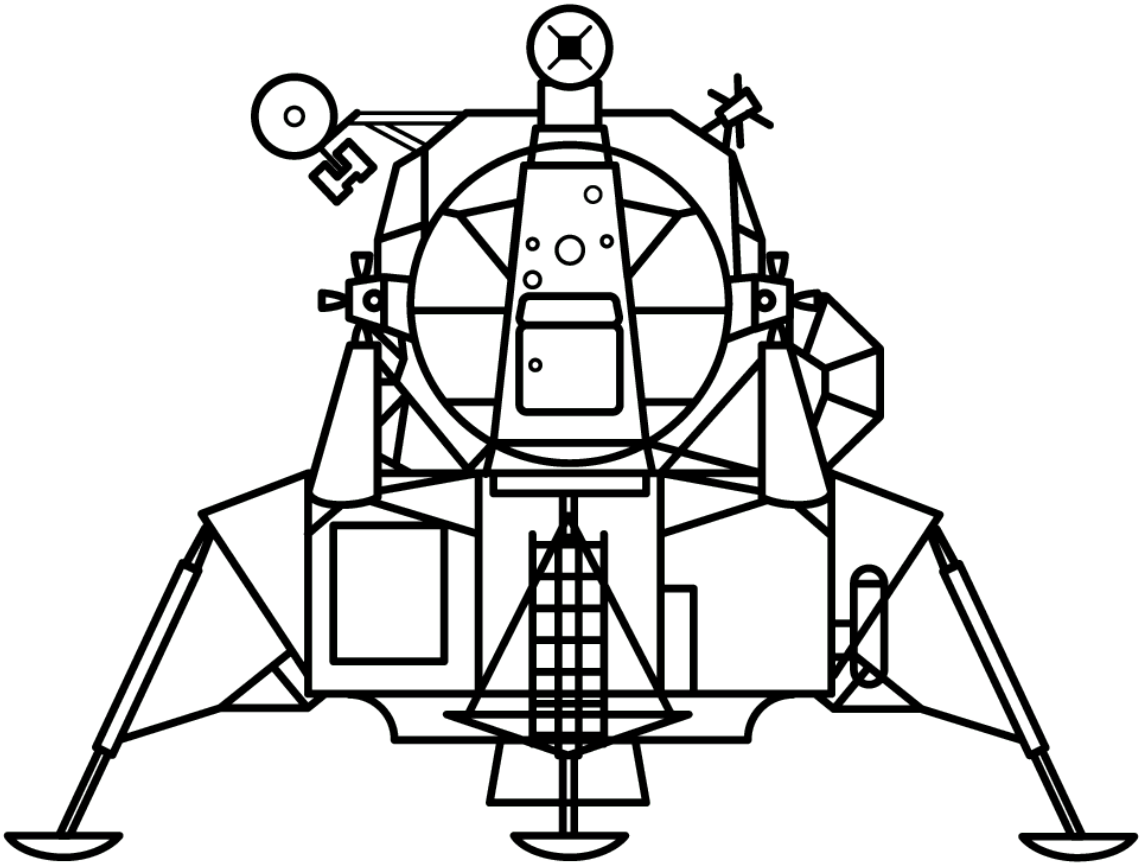
Project 1: Lunar Lander Truss

Aerospace 423: Computational Methods in Aerospace Engineering
Undergraduate Aerospace Engineering
University of Michigan, Ann Arbor

By: Dan Card, dcard@umich.edu
Date: February 14, 2020



Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this project I will simulate the landing of a 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 below.

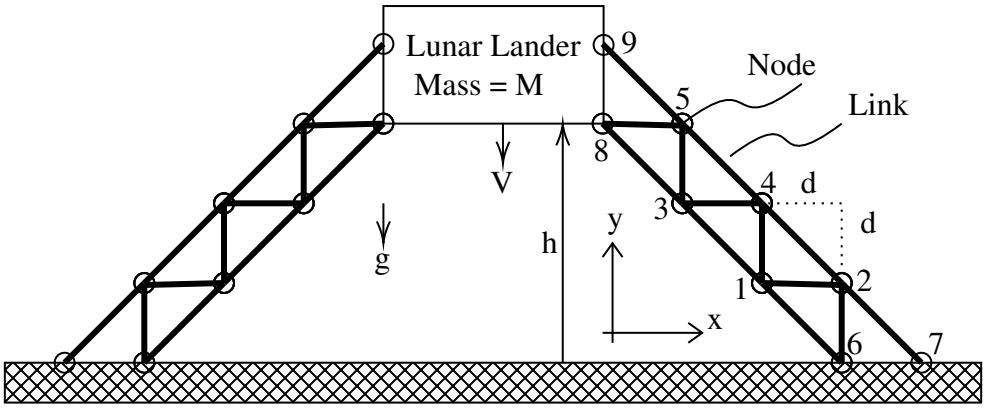


Figure 1: Lunar lander problem setup. The lander has four legs: the two not shown are out of the plane of the page.

The other legs behave symmetrically. In addition, I will assume in-plane deformation, so that the problem is 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.

1.1 Link Force Calculation

The nodes move because of the forces exerted on them by the links. Assume 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 has stretched, δl, and how fast it is being stretched, δl:

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

Taking a closer look at how to calculate the resulting force contributions at the two adjacent nodes – call these i and j. First, a few definitions:

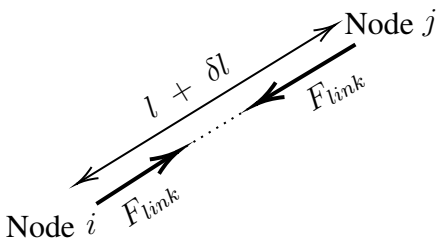


Figure 2: Force in one link.

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

The quantities appearing in Equation 1 can then be calculated as

$$\begin{aligned}
\delta l =& |\vec{x_j} - \vec{x_i}| - l_0 \\
\dot{\delta} l =& (\vec{v_j} - \vec{v_i}) \cdot \vec{e_{ij}}
\end{aligned}$$

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

$$\vec{e_{ij}} = \frac{\vec{x_j} - \vec{x_i}}{|\vec{x_j} - \vec{x_i}|}$$

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

$$\vec{F_{ij}} = F_{link}\vec{e_{ij}} \tag{2}$$

$$\vec{F_{ji}} = -F_{link}\vec{e_{ij}} \tag{3}$$

As shown above, a positive F_{link} corresponds to a tension force that pulls the nodes closer together. δl is positive when the link is stretched longer than its initial equilibrium length, and δl 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.

1.2 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. Its (vertical) acceleration is caused by its 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 .

1.3 Parameters

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

Parameter	Description	Units	Baseline Values
T	Time horizon	s	0.5
m_{node}	Node mass	kg	0.1
M	Lander mass	kg	100
d	Initial truss width/height	m	0.2
k	Link stiffness	N/m	10^5
γ	Link damping	$N \cdot s/m$	200
g	Acceleration due to gravity	m/s^2	1.625
V	Initial impact (descent) speed	m/s	6

Table 1: Simulation Parameters for Lunar Lander.

2 Numerical Approach

To simulate the dynamics of the truss, I will 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

$$\dot{u} = f(u),$$

where f contains the physics (force-balance) calculations.

I will implement three numerical integration techniques for solving this system of ODEs: forward Euler, second-order Adams Bashforth, and fourth-order Runge-Kutta. When discretizing time, I will use a constant time step of $\Delta t = T/N_t$, where N_t is the number of time steps.

3 Simulating a Lunar Lander

3.1 Formulating the Equations of Motion

Firstly to simulate the lunar landing, I will form the equations of motion for the free-nodes of the lunar lander. This is done through simple force balances shown in the sections below.

3.1.1 Equations of Motion for Free-Nodes

Formulating the equations of motion for the free-nodes in the lunar lander truss will be done through the use of force balances. The acceleration experienced on node i is shown below in Equation 4.

$$\begin{bmatrix} \ddot{x}_i \\ \ddot{y}_i \end{bmatrix} = \frac{1}{m_{node}} \sum_{j=1}^{elements} \left(k\delta l + \gamma \dot{\delta l} \right) \vec{e_{i,j}} - \begin{bmatrix} 0 \\ g \end{bmatrix} \tag{4}$$

Where *elements* are the surrounding links that connect node to node and sum to the force affecting the lunar truss's node.

3.1.2 Equations of Motion for Lunar Lander

Formulating the equations of motion for the lunar lander is similar to the forces experienced by the free-nodes with small alterations. The main difference will be including the mass of lander. The equation for the lunar lander is shown below in Equation 5.

$$\begin{bmatrix} \ddot{X}_{lander} \\ \ddot{Y}_{lander} \end{bmatrix} = \begin{bmatrix} 0 \\ \ddot{y}_8 \end{bmatrix} = \frac{4}{m_{lander}} \sum_{j=1}^{elements} \left(k\delta l + \gamma \dot{\delta l} \right) \vec{e_{i,j}} - \begin{bmatrix} 0 \\ g \end{bmatrix} \tag{5}$$

Similar to Equation 4, the equation of motion for the lunar lander are similar but has an additional mass contribution from the lunar lander. Also, the "4" in the numerator that appears from four legs supporting the lunar lander. The acceleration in the x direction can be ignored due to the other lunar legs limiting the motion.

Note: $k\delta l$ is the restorative force, and $\gamma \dot{\delta l}$ is the damping force, and finally $\vec{e_{i,j}}$ is the direction of the tensional force in the link, also noteworthy is that g is the acceleration experienced from the lander. All of which have been defined previously in Equations 1,2,3.

3.2 Determining the Time-Step Restrictions

To determine the time-step restrictions I will numerically approximate the $\frac{\partial f}{\partial \underline{u}}$ matrix to find the eigenvalues and solve for the time-step restrictions.

3.2.1 Constructing the Jacobian Matrix

To determine the time-step restrictions, I will first have to linearize the differential equation.

$$\begin{aligned}\dot{\underline{u}} &= \underline{f}(\underline{u}) \\ \dot{\underline{u}}' &= \underline{\underline{A}}\underline{u}' + \underline{q}\end{aligned}$$

Where $\underline{\underline{A}}$ is defined as,

$$\begin{aligned}\underline{\underline{A}} &= \frac{\partial f}{\partial \underline{u}}|_{\bar{u}} \\ \underline{u} &= [u_1, u_2, \ldots, u_n]^T\end{aligned}$$

So formulating the $\underline{\underline{A}}$ matrix is given to be,

$$\frac{\partial f}{\partial \underline{u}} = \begin{bmatrix} \frac{\partial f}{\partial u_1} & \frac{\partial f}{\partial u_2} & \cdots & \frac{\partial f}{\partial u_1} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{6}$$

And then $\frac{\partial f}{\partial u_n}$ from Equation 6 is defined from the definition of the derivative,

$$\frac{\partial f}{\partial u_n} = \frac{\underline{f}(\underline{u}_0 + \epsilon \underline{e}_n) - \underline{f}(u_0)}{\epsilon}$$

Where ϵ is some small number that I choose to be $\mathcal{O}(10^{-6})$

Once the $\underline{\underline{A}}$ matrix has been formed, the physics of the problem form the eigenvalues that will be used for Forward Euler, Adams-Bashforth, and Forth Order Runge Kutta integration schemes. The implementation of this code can be found in Appendix A.

3.2.2 Determining Time-Step Restrictions

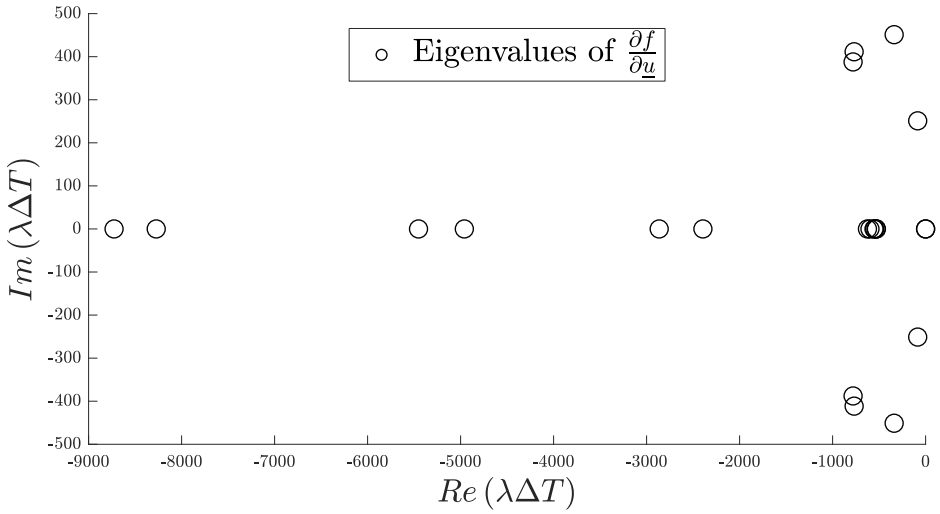


Figure 3: Eigenvalues for governing physics problem.

Above in Figure 3 are the eigenvalues to $\underline{\underline{A}}$, governed by the physics of the lunar landers truss. These are eigenvalues that I will form my ΔT for each integration scheme.

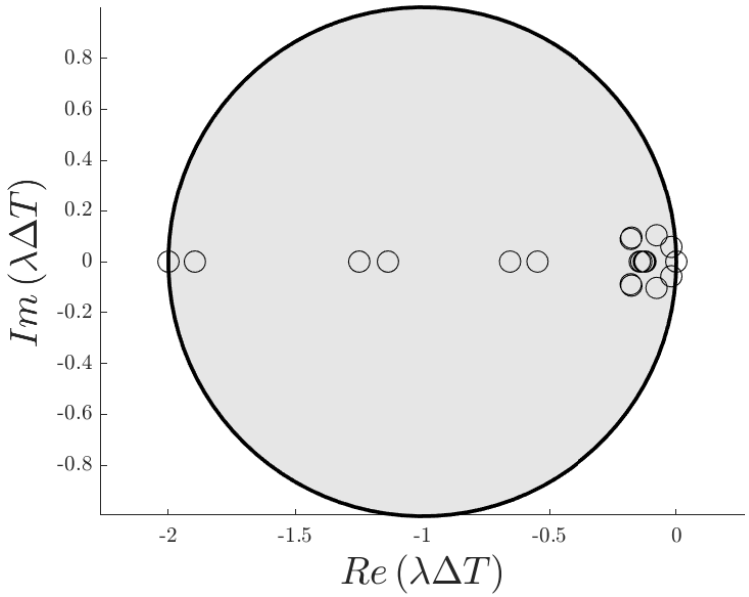


Figure 4: Eigenvalue stability region for Forward Euler.

Shown above in Figure 4 is the stability region for Forward Euler integration scheme. Forming the eigenvalues to this stability region show the minimum and maximum values that ΔT can be to remain an stable integration. Analyzing the data shows that the theoretical maximum ΔT is $2.291 \cdot 10^{-4} s$. However, to ensure that these shifting eigenvalues throughout the deformation of the truss does not incur instabilities I will determine the appropriate ΔT values.

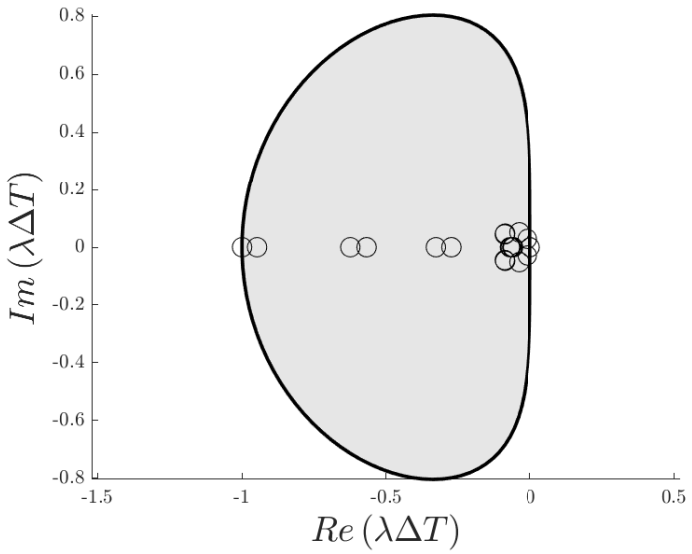


Figure 5: Eigenvalue stability region for Second-Order Adams-Bashforth.

Similar to the Forward Euler, above in Figure 5 is Adams-Bashforth stability region. Analyzing the region of stability shows that the ΔT required for a stable integration is $\Delta T = 1.146 \cdot 10^{-4} s$. However, to ensure that these shifting eigenvalues throughout the deformation of the truss does not incur instabilities I will determine the appropriate ΔT values.

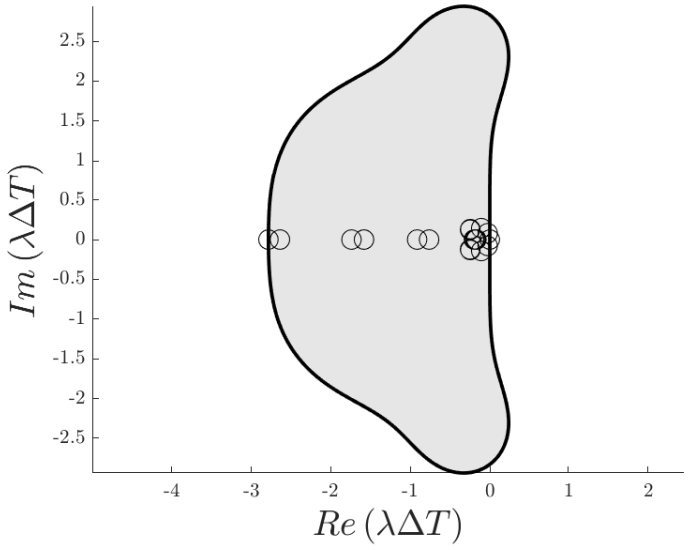


Figure 6: Eigenvalue stability region for Forth-Order Runge Kutta.

Lastly is the stability region for Forth-Order Runge Kutta, shown above in Figure 6 is the stability region. Computing the ΔT required for a stable integration shows that $\Delta T = 3.192 \cdot 10^{-4} s$. However, to ensure that these shifting eigenvalues throughout the deformation of the truss does not incur instabilities I will determine the appropriate ΔT values.

3.3 Verifying Stability Across Integration Schemes

Determining the ΔT that would allow for stability across the integration schemes was accomplished from the $\underline{\underline{A}}$ matrix combined with the stability regions unique to each integration scheme.

$$\Delta T = \min(\lambda \Delta T) / \min(\lambda)$$

Where $\lambda \Delta T$ is governed by the integration scheme being used: Forward Euler, Fourth-Order Runge Kutta, Second-Order Adams-Bashworth, etc. Then λ is the eigenvalues that were approximated from the Jacobian matrix, $\underline{\underline{A}}$ that would approximate the ΔT most appropriate for each integration scheme.

Choosing the approximate values to remain in the stable integration region I choose that the number of steps required would be:

$$N_s = \text{trunc}\left(\frac{T}{\Delta T_{\text{scheme}}}\right) + 50_{steps} \tag{7}$$

Using this relation I have been able to determine the **minimum** steps required for each integration scheme, and allow for a "safety factor" to ensure that the numerical integrations remain stable. The safety factor I choose was an additional 50 steps for each integration scheme, and these next sections I will verify that this safety factor of 50 additional steps will allow for the integration scheme to remain stable.

3.3.1 Forward Euler Stability

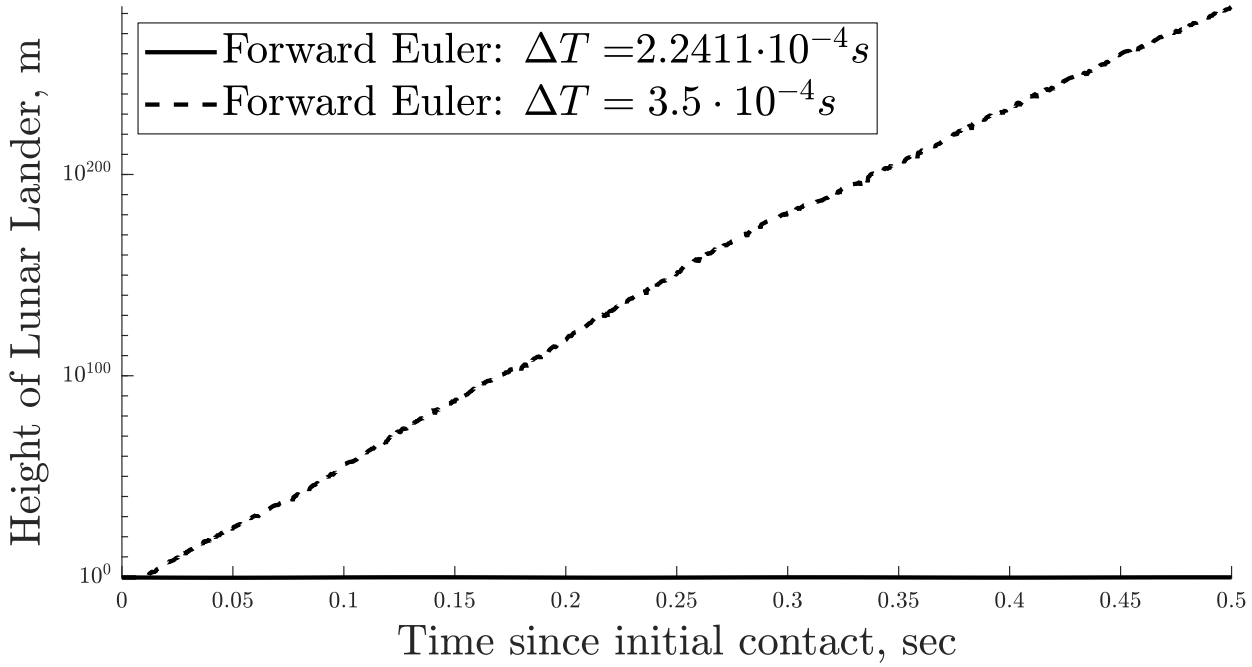


Figure 7: Verifying Forward Euler stability for time-steps approximated.

From the time-step restrictions from Figure 4, the theoretical maximum ΔT is $2.291 \cdot 10^{-4} s$. However, to account for variations in eigenvalue stability as the truss deforms, I will use a ΔT_{FE} value of $2.24 \cdot 10^{-4} s$ from Equation 7. Looking above to Figure 7, a value that is slightly outside of the maximum time-step grows unstable whereas the time-step I denoted above is shown to be stable throughout the integration.

3.3.2 Second-Order Adams-Bashforth Stability

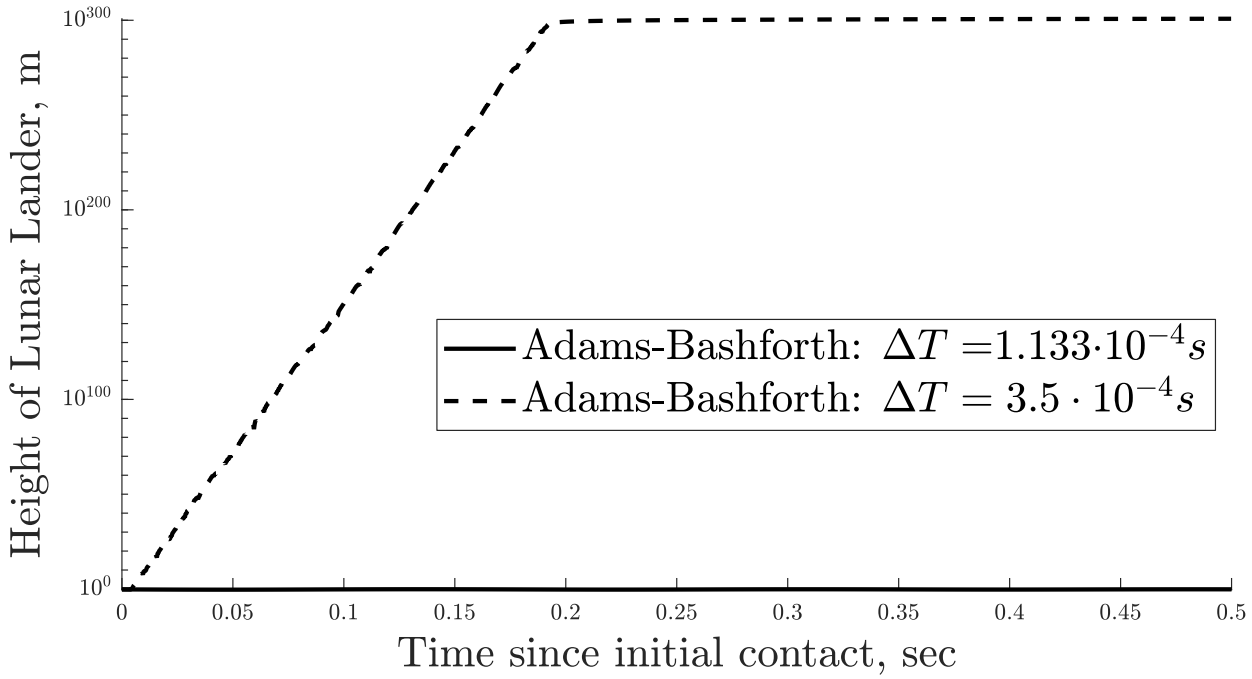


Figure 8: Verifying Second-Order Adams-Bashforth stability for approximated time-steps.

From the time-step restrictions from Figure 5, the theoretical maximum ΔT for Second-Order Adams-Bashforth integration scheme is $\Delta T = 1.146 \cdot 10^{-4} s$. However, to account for variations in eigenvalue stability as the truss deforms, I will use a $\Delta T_{AB2} = 1.1 \cdot 10^{-4} s$ from Equation 7. Looking above to Figure 8, a value that is slightly outside of the maximum time-step grows unstable whereas the time-step I denoted above is shown to be stable throughout the integration.

3.3.3 Fourth-Order Runge Kutta Stability

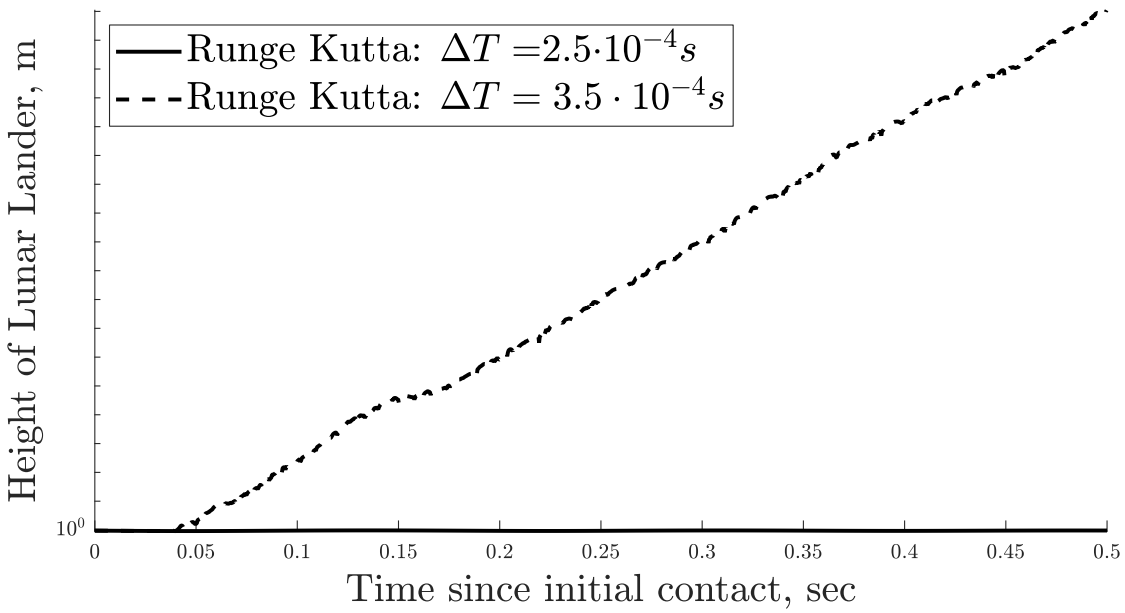


Figure 9: Verifying Fourth-Order Runge Kutta stability for approximated time-steps.

From the time-step restrictions from Figure 6, the theoretical maximum ΔT for Fourth-Order Runge Kutta is $\Delta T = 3.192 \cdot 10^{-4} s$. However, to account for variations in eigenvalue stability as the truss deforms, I will use a $\Delta T_{RK4} = 2.5 \cdot 10^{-4} s$ from Equation 7. Looking above to Figure 9, a value that is slightly outside of the maximum time-step grows unstable whereas the time-step I denoted above is shown to be stable throughout the integration.

3.4 Simulating the Lunar Landing

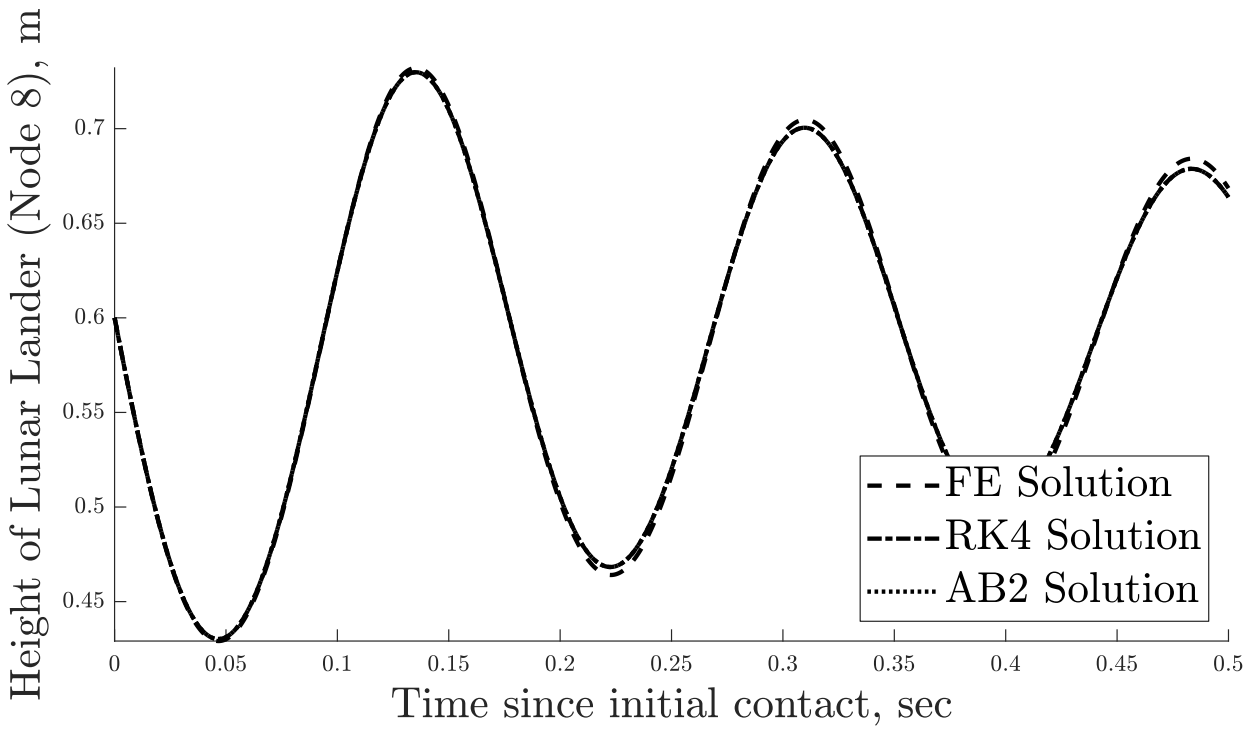


Figure 10: Analysis of the heights across multiple integration schemes.

Discussion of Simulation

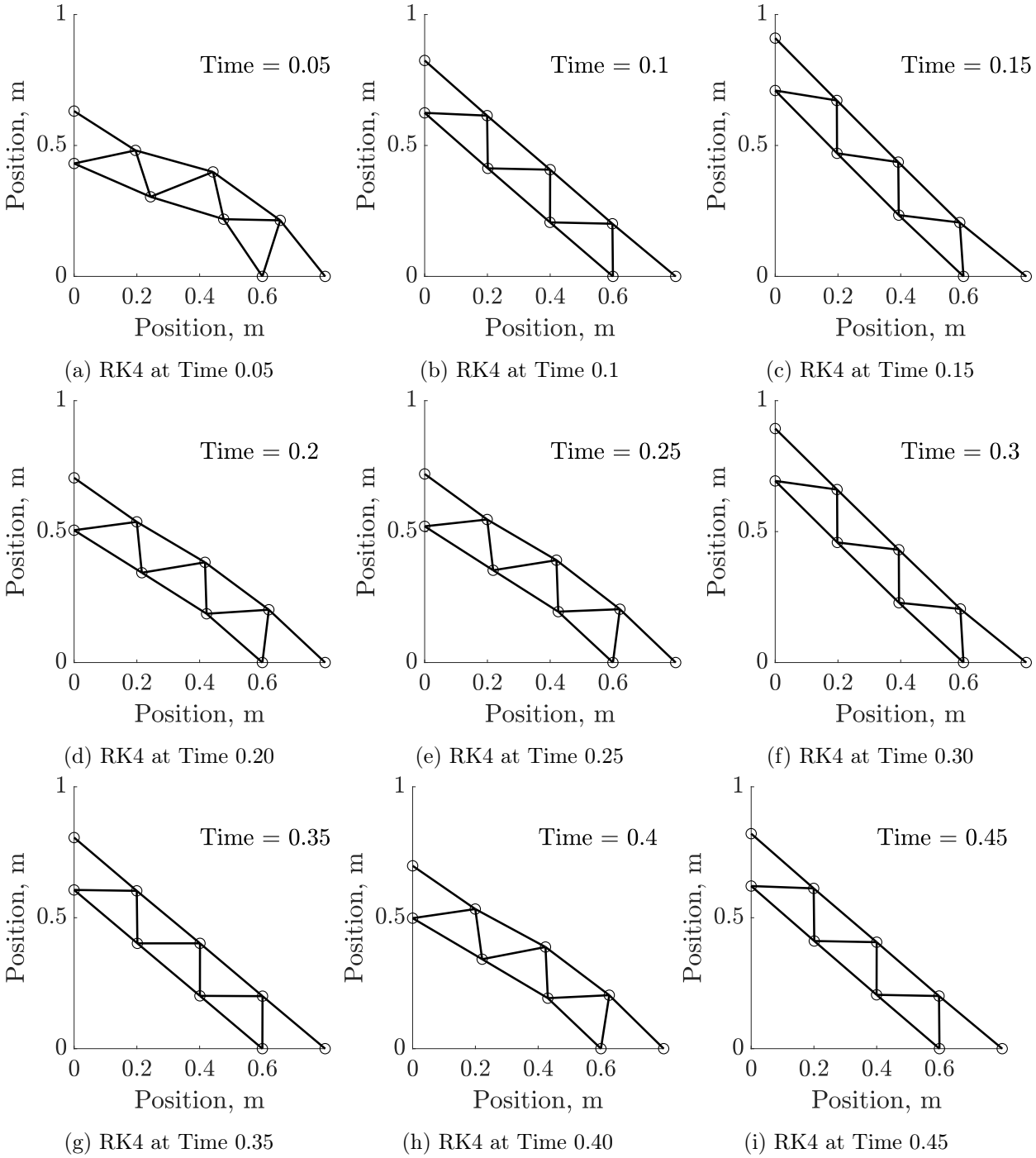
After implementing FE, AB2, and RK4 integration schemes I was able to simulate the height of the lunar lander during the initial impact. An analysis of Figure 10 shows that the truss goes through several dampened oscillations about its equilibrium position. This intuitively matches theory since the links on the truss impede its motion and should drive the lunar lander towards its equilibrium position.

Analysis of Integration Schemes

After confirming the physics of the simulation, I can confirm that each of the integration schemes was able to accurately model the trusses height. However, looking to Figure 10, the Forward Euler integration scheme is shown to have more variation than the second order Adams-Bashforth and fourth-order Runge Kutta integration schemes. Variation is expected from Forward Euler due to the nature of this integration scheme being a single-stage scheme and with a complicated force acting upon the truss.

3.5 Analysis of Lander Leg Truss Deformation

Fourth-Order Runge Kutta Simulations



(a) RK4 at Time 0.05 (b) RK4 at Time 0.1 (c) RK4 at Time 0.15

(d) RK4 at Time 0.20 (e) RK4 at Time 0.25 (f) RK4 at Time 0.30

(g) RK4 at Time 0.35 (h) RK4 at Time 0.40 (i) RK4 at Time 0.45

Figure 11: Deformation of lunar truss at several times using RK4 integration scheme.

Discussion of Simulation and Expected Values

Analyzing Figure 11 and the trusses deformations at multiple times shows how it deforms over the initial impact. The deformations shown in Figure 11 are what I expect since the truss should oscillate after the initial impact before coming to an equilibrium position. This visualization of the truss matches the data calculated from Figure 10 in the sense that there are visible dampened oscillations about some equilibrium position.

3.6 Convergence Study

Performing a convergence study on the integration schemes can validate the numerical solutions because having known the convergence rates of each scheme I can show that these schemes are performing nominally. To determine the convergence I will generate a plot of the height error versus differing the time-steps.

Where error in the height is defined as,

$$\tau_{\text{error}} = |u_{exact} - u_n| \tag{8}$$

However, since there is no numerical solution to this nonlinear problem, to generate an "exact" solution shown in Equation 8, I will perform a numerical integration that has $100,010$ time-steps and allows for a very accurate approximation of the exact solution for a non-linear problem.

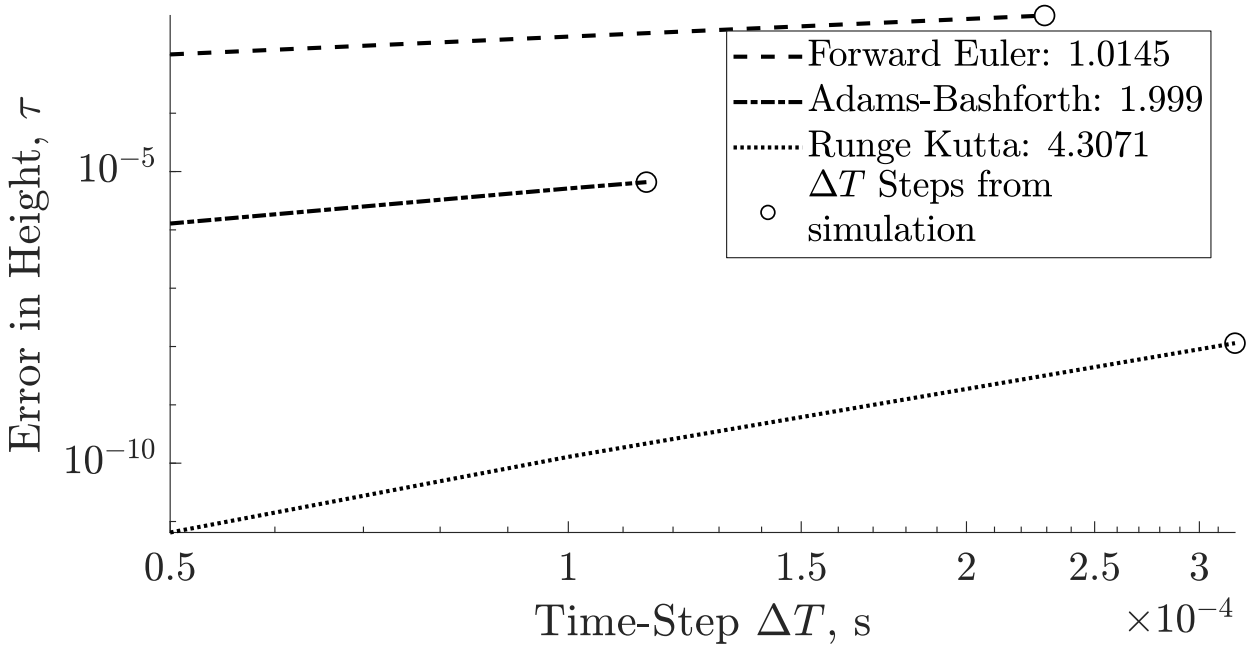


Figure 12: Convergence study of height error between multiple integration schemes.

3.6.1 Analysis of Integration Schemes

Looking to Figure 12, is the detailed analysis of the converge rates associated with each integration scheme. With knowledge of the integration schemes, Forward Euler is expected to have a convergence rate $\mathcal{O}(\Delta T)$, or on the log-log plot denoted as a slope of approximately "1". Then for Second-Order Adams-Bashforth the convergence rate is approximately $\mathcal{O}(\Delta T^2)$ or a slope on the log-log plot of "2". Finally for Fourth-Order Runge Kutta, the convergence rate is approximately $\mathcal{O}(\Delta T^4)$ or a slope of "4" on a log-log plot.

3.6.2 Discussion of Convergence and Expected Values

After knowing the expected values, the legend in Figure 12 shows the slopes of the convergence and it agrees with the theoretical values and verifies that the equations of motion have the correct integration schemes. This Figure shows that the Forward Euler does indeed have a convergence rate of $\mathcal{O}(\Delta T)$, that Second-Order Adams-Bashforth has a convergence rate of $\mathcal{O}(\Delta T^2)$, and finally that Fourth-Order Runge Kutta converges at a rate of $\mathcal{O}(\Delta T^4)$.

4 Discussion and Conclusions

In this project I was able to simulate a non-linear system of differential equations and accurately represent the deformation of a lunar lander. Not only was I able to simulate the landing itself, I was able to determine from the governing physics the most cost-effective way to simulate these integration schemes from the stability regions unique to each integration scheme. From these governing stability regions, I was able to simulate the lander and show how it acts similar to a dampened rigid body oscillating about some equilibrium position. Finally I was able to validate my integration schemes and results and solve for the convergence rates and compare to mathematical theory.

Appendices

A Driving Code for Lunar Lander Simulation

The highlighted code below is the main matlab code that would call all function and generate the plots needed in this project. The functions that are being referenced in this Appendix are shown in the following Appendices.

Algorithm 1: Main function matlab code for lunar truss.

```
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%      Dan Card, Aero 423 - Project 1: Lunar Lander
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
clear all; clc; close all
set(groot,'defaulttextinterpreter','latex');
set(groot, 'defaultAxesTickLabelInterpreter','latex');
set(groot, 'defaultLegendInterpreter','latex');
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Constants given for project
time_horizon = 0.5;
d0 = 0.2;
v0 = -6;
thetlin = linspace(0, 2*pi, 100);
% Initial Node locations and initial velocities
nodes = [ 2*d0, d0, 0, v0;
          3*d0, d0, 0, v0;
          d0, 2*d0, 0, v0;
          2*d0, 2*d0, 0, v0;
          d0, 3*d0, 0, v0;
          3*d0,0, 0, 0;
          4*d0, 0, 0, 0;
          0, 3*d0, 0, v0;
          0, 4*d0, 0, v0];
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Eigenvalue stability
% Call function to numerically approximate A-matrix, then find eigenvalues
amatrix = makemat(nodes);
eiga = eig(amatrix);
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
figure()
scatter(real(eiga), imag(eiga), 100, 'k')
xlabel('$Re␣\left(␣\lambda␣\Delta␣T␣\right)$','fontsize', 18)
ylabel('$Im␣\left(␣\lambda␣\Delta␣T␣␣\right)$','fontsize', 18)
legend('Eigenvalues␣of␣$\frac{\partial␣f}{\partial␣\underline{u}}$','location', '
    north','fontsize', 18)
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig eigen_physics.eps -native
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Solve for eigenvalue stability region
ldt_fe = exp(thetlin.*1i) - 1;
scale = min(real(eiga))/min(real(ldt_fe)); % Scale the region to fit
dt_fe = min(real(ldt_fe))/min(real(eiga)); % Calculate maximum dT value
fprintf('\n\nMaximum␣dT␣for␣Forward␣Euler␣is␣%.8f␣[sec]', dt_fe)

figure()
hold on
 fill (real(ldt_fe), imag(ldt_fe),[1,1,1],'FaceColor',[.9,.9,.9],'EdgeColor','k','
    linewidth',1.8)
scatter(real(eiga)/scale, imag(eiga)/scale, 100, 'k')
xlim([-2.5, 0])
axis equal
xlabel('$Re␣\left(␣\lambda␣\Delta␣T␣\right)$','fontsize', 18)
ylabel('$Im␣\left(␣\lambda␣\Delta␣T␣␣\right)$','fontsize', 18)
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig eigen_fe.eps -native
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
g = exp(thetlin.*1i);
```

```
56
57 % Solve for eigenvalue stability region
58 ldt_ab2 = (g - 1).*2.*g./(3.*g-1);
59 scale = min(real(eiga))/min(real(ldt_ab2)); % Scale the region to fit
60 dt_ab2 = min(real(ldt_ab2))/min(real(eiga)); % Calculate maximum dT value
61 fprintf('\n\nMaximum␣dT␣for␣Adams-Bashworth␣is␣%.8f␣[sec]', dt_ab2)
62
63 figure
64 hold on
65  fill (real(ldt_ab2), imag(ldt_ab2),[1,1,1],'FaceColor',[.9,.9,.9],'EdgeColor','k','
       linewidth',1.8)
66 scatter(real(eiga)/scale, imag(eiga)/scale, 100, 'k')
67 xlim([-1.5, 0])
68 axis equal
69 xlabel('$Re␣\left(␣\lambda␣\Delta␣T␣\right)$','fontsize', 18)
70 ylabel('$Im␣\left(␣\lambda␣\Delta␣T␣␣\right)$','fontsize', 18)
71 set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
72 export_fig eigen_ab2.eps -native
73 %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
74 % Pre-allocate
75
76 num = max(size(g));
77 dat1 = zeros(1, num);
78 dat2 = zeros(1, num);
79 dat3 = zeros(1, num);
80 dat4 = zeros(1, num);
81
82 % Run through the solutions of g, there are four solutions
83 syms val
84 for i = 1:max(size(g))
85     eqn = g(i) == 1 + val + 1/2.*(val).^2 + 1/6.*(val).^3 + 1/24.*(val).^4;
86     sol = vpasolve(eqn , val);
87     dat1(i) = sol(1);
88     dat2(i) = sol(2);
89     dat3(i) = sol(3);
90     dat4(i) = sol(4);
91 end
92
93 % All solutions diverge at their mid-section, simplifying to one vector
94 ldt_rk4 = [ dat1(1:(num/2)), dat2((num/2 +1):end), dat3(1:(num/2)),...
95            dat4((num/2+1):end), dat4(2:(num/2)), dat3((num/2+1):end),...
96            dat2(2:(num/2)), dat1((num/2+1):end)];
97
98 scale = min(real(eiga))/min(real(ldt_rk4));
99 dt_rk4 = min(real(ldt_rk4))/min(real(eiga));
100 fprintf('\n\nMaximum␣dT␣for␣Forth-Order␣Runge␣Kutta␣is␣%.8f␣[sec]', dt_rk4)
101
102 figure
103 hold on
104  fill (real(ldt_rk4), imag(ldt_rk4),[1,1,1],'FaceColor',[.9,.9,.9],...
105     'EdgeColor','k','linewidth',1.8)
106 scatter(real(eiga)/scale, imag(eiga)/scale, 100, 'k')
107 xlim([-3, 0])
108 axis equal
109 xlabel('$Re␣\left(␣\lambda␣\Delta␣T␣\right)$','fontsize', 18)
110 ylabel('$Im␣\left(␣\lambda␣\Delta␣T␣␣\right)$','fontsize', 18)
111 set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
112 export_fig eigen_rk4.eps -native
113 %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
114 % Verifying Stability
115 tlin_us = 0:0.00035:time_horizon;
116 fe_unstable = euler(nodes, tlin_us);
117 ab2_unstable = ab2(nodes, tlin_us);
118 rk4_unstable = rk4(nodes, tlin_us);
119
120 % Time-Steps that allow for stable-integrations
121 fe_tlin = linspace(0, time_horizon, fix(0.5/dt_fe)+50);
122 ab2_tlin = linspace(0, time_horizon, fix(0.5/dt_ab2)+50);
123 rk4_tlin = 0:0.00025:time_horizon;
124
125 % Call integration schemes
```

```
fe_dat = euler(nodes, fe_tlin);
ab2_dat = ab2(nodes, ab2_tlin);
rk4_dat = rk4(nodes, rk4_tlin);

figure()
hold on
plot(fe_tlin, fe_dat(30,:), 'k-', 'linewidth', 1.8)
plot(tlin_us, abs(fe_unstable(30,:)), 'k--', 'linewidth', 1.8)
axis tight
xlabel('Time since initial contact, sec', 'fontsize', 18)
ylabel('Height of Lunar Lander, m', 'fontsize', 18)
legend({['Forward Euler: $\Delta T = $',num2str((fe_tlin(2)-fe_tlin(1))*10^4),'$\
    cdot 10^{-4}s$'],...
    'Forward Euler: $\Delta T = 3.5\cdot 10^{-4}s$'},'location','best','fontsize',
        18)
set(gca, 'YScale', 'log')
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig fe_unstable.eps -native

figure()
hold on
plot(ab2_tlin, ab2_dat(30,:), 'k-', 'linewidth', 1.8)
plot(tlin_us, abs(ab2_unstable(30,:)), 'k--', 'linewidth', 1.8)
axis tight
xlabel('Time since initial contact, sec', 'fontsize', 18)
ylabel('Height of Lunar Lander, m', 'fontsize', 18)
legend({['Adams-Bashforth: $\Delta T = $',num2str((ab2_tlin(2)-ab2_tlin(1))*10^4),'$
    \cdot 10^{-4}s$'],...
    'Adams-Bashforth: $\Delta T = 3.5\cdot 10^{-4}s$'},'location','best','fontsize',
        18)
set(gca, 'YScale', 'log')
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig ab2_unstable.eps -native

figure()
hold on
plot(rk4_tlin, rk4_dat(30,:), 'k-', 'linewidth', 1.8)
plot(tlin_us, abs(rk4_unstable(30,:)), 'k--', 'linewidth', 1.8)
axis tight
xlabel('Time since initial contact, sec', 'fontsize', 18)
ylabel('Height of Lunar Lander, m', 'fontsize', 18)
legend({['Runge Kutta: $\Delta T = $',num2str((rk4_tlin(2)-rk4_tlin(1))*10^4),'$\
    cdot 10^{-4}s$'],...
    'Runge Kutta: $\Delta T = 3.5\cdot 10^{-4}s$'},'location','best','fontsize', 18)
set(gca, 'YScale', 'log')
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig rk4_unstable.eps -native
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Simulating Lunar Landing

figure()
hold on
plot(fe_tlin, fe_dat(30,:), 'k--', 'linewidth', 1.8)
plot(rk4_tlin, rk4_dat(30,:), 'k-.', 'linewidth', 1.8)
plot(ab2_tlin, ab2_dat(30,:), 'k:', 'linewidth', 1.8)
axis tight
xlabel('Time since initial contact, sec', 'fontsize', 18)
ylabel('Height of Lunar Lander (Node 8), m', 'fontsize', 18)
legend({'FE Solution', 'RK4 Solution','AB2 Solution'}, 'location','southeast','
    fontsize', 18)
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig height_plot.eps -native

final_h = rk4_dat(30,end);
fprintf('\n\nFinal height of the truss is %.5f m', final_h)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Truss deformations
plot_truss(fe_dat, fe_tlin, 0)
plot_truss(ab2_dat, ab2_tlin, 0)
plot_truss(rk4_dat, rk4_tlin, 1)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
% Convergence Error
tlin_md = 0:0.0001:time_horizon;
tlin_high = 0:0.00005:time_horizon;
fe_l2 = [calcerr(euler(nodes, tlin_high)), calcerr(euler(nodes, tlin_md)), calcerr(
    fe_dat)];
ab2_l2 = [calcerr(ab2(nodes, tlin_high)), calcerr(ab2(nodes, tlin_md)), calcerr(
    ab2_dat)];
rk4_l2 = [calcerr(rk4(nodes, tlin_high)), calcerr(rk4(nodes, tlin_md)), calcerr(
    rk4_dat)];

mfe = log10(fe_l2(2)/fe_l2(1))/log10(0.0001/0.00005);
mab2 = log10(ab2_l2(2)/ab2_l2(1))/log10(0.0001/0.00005);
mrk4 = log10(rk4_l2(2)/rk4_l2(1))/log10(0.0001/0.00005);

figure()
hold on
plot([0.00005, 0.0001, dt_fe], fe_l2, 'k--', 'linewidth', 1.8)
plot([0.00005, 0.0001, dt_ab2], ab2_l2, 'k-.', 'linewidth', 1.8)
plot([0.00005, 0.0001, dt_rk4], rk4_l2, 'k:', 'linewidth', 1.8)
scatter(dt_fe, fe_l2(end), 75, 'k')
scatter(dt_ab2, ab2_l2(end), 75, 'k')
scatter(dt_rk4, rk4_l2(end), 75, 'k')
xlabel('Time-Step␣$\Delta␣T$,␣s')
ylabel('␣Error␣in␣Height,␣$\tau$')
legend({['Forward␣Euler:␣',num2str(mfe)],['Adams-Bashforth:␣', num2str(mab2)],...
    ['Runge␣Kutta:␣',num2str(mrk4)],['$\Delta␣T$␣Steps␣from', newline, 'simulation'
        ]}, 'location', 'best')
set(gca, 'Yscale', 'log')
set(gca, 'Xscale', 'log')
set(gca, 'fontsize', 18)
set(gcf, 'Color', 'w', 'Position', [200 200 800 400]);
export_fig l2_error.eps -native
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

B Implementing Force Functions

B.1 Dynamics Function

The dynamics function determines the accelerations experienced at each given node in the lunar lander truss. This function allows for the each and repeatable calculations of the accelerations for each node passed into this function.

Algorithm 2: Matlab Implementation of Dynamics Function.

```
function an = f(nodestate)
    % Constants for the lunar lander truss
    mnode = 0.1; lander_mass = 100; g = 1.625;

    % Creates a column vector that consists of
    %      Vel_X; Vel_Y; Accel_X; Accel_Y ...
    %      For all nodes
an = [[nodestate(1, 3:4)';(flink(nodestate, 1)./mnode)'] - [0;0;0;g]; % Node 1
      [nodestate(2, 3:4)';(flink(nodestate, 2)./mnode)'] - [0;0;0;g]; % Node 2
      [nodestate(3, 3:4)';(flink(nodestate, 3)./mnode)'] - [0;0;0;g]; % Node 3
      [nodestate(4, 3:4)';(flink(nodestate, 4)./mnode)'] - [0;0;0;g]; % Node 4
      [nodestate(5, 3:4)';(flink(nodestate, 5)./mnode)'] - [0;0;0;g]; % Node 5
      [0; 0; 0; 0];                                                   % Node 6
      [0; 0; 0; 0];                                                   % Node 7
      ([nodestate(9, 3:4)'.*[0;1]; 8.*(flink(nodestate, 8)./lander_mass)'.*[0;1]] -
          [0;0;0;g]); % Node 8
      ([nodestate(9, 3:4)'.*[0;1]; 8.*(flink(nodestate, 9)./lander_mass)'.*[0;1]] -
          [0;0;0;g])];% Node 9
end
```

B.2 Link Force Function

The link force function, based off Equations 4, 5 determine the forces due to the restoring forces and from the damping force. This function determines what nodes have a link attached between the two and then sums all the forces on the node to determine the total force per node.

Algorithm 3: Numerical Approximation of the Forces at Each Node.

```
function an = flink(nodes, node)
    % Constants for the links
    k = 10^5; gamma = 200;
    an = [0, 0];

    % A matrix of each node and the nodes that it connects to around
    nodecon = [ 2, 3, 4, 6;
                1, 4, 6, 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];

       % Iterate through each of the nodes
    for i = 1:max(size(nodes))
        if ismember(node, nodecon(i,:)) == 1
            % Determines the length of the link
            if node == 1 && (i == 2 || i == 4)
                L_0 = 0.2;
            elseif node == 2 && (i == 1 || i == 6)
```

```
            L_0 = 0.2;
        elseif node == 3 && (i == 4 || i == 5)
            L_0 = 0.2;
        elseif node == 4 && (i == 1 || i == 3)
            L_0 = 0.2;
        elseif node == 5 && (i == 3 || i == 8)
            L_0 = 0.2;
        elseif node == 6 && i == 2
            L_0 = 0.2;
        elseif node == 8 && i == 5
            L_0 = 0.2;
        else
            L_0 = 0.2 * sqrt(2);
        end

        % Gather the state position/velocity at state u
        x1 = nodes(node,1:2);
        x2 = nodes(i,1:2);

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

        % Calculates the direction that the force will be acting in
        eij = (x2 - x1)./(norm(x2-x1));

        % Calculates the portions of the forces on each link
        deltal = norm(x2 - x1) - L_0;
        if abs(deltal) < 10^(-16)
           deltal = 0;
        end

        ddeltal = dot((v2 - v1), eij);
        if abs(ddeltal) < 10^(-16)
           ddeltal =0;
        end

        % Sums all the forces on a given node
        if norm((k*deltal + gamma*ddeltal).*eij) > 10^(-16)
           an = an + (k*deltal + gamma*ddeltal).*eij;
        end
      end
    end
end
```

C Implementations of Integration Schemes

The integration schemes used in this project are Forward Euler, Second-Order Adams-Bashforth, and Fourth-Order Runge Kutta whose implementations are shown below.

C.1 Forward Euler Implementation

The Forward Euler is one of the simplest finite-difference methods to numerically integrate for an approximate solution. The equation for this integration scheme is shown below,

$$u_{n+1} = u_n + \Delta T f(u_n)$$

Where, u_{n+1} is the updated (approximate) value of the next state. Whereas ΔT is the time-step being used through the integration scheme, and $f(u_n)$ is the derivative evaluated at the current time-step.

The implementation of this scheme in this lunar lander truss simulation is shown below.

Algorithm 4: Implementation of a Forward Euler integration schemes.

```
function an = euler(nodes,t)
    % Determine time-step and size of vector
    dt = t(2)-t(1);
    n = max(size(t));

    % Grab initial condition into column vector
    u = zeros(min(size(nodes))*max(size(nodes)), max(size(t)));
    u(:,1) = [nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
        nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)'; nodes(9,:)'];

    for i = 1:(n-1)
        % Forward step in time
        nodetemp = [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(nodetemp);
    end
    an = u;
end
```

C.2 Second-Order Adams-Bashforth Implementation

Second-Order Adams-Bashforth is similar in complexity to Forward Euler, but involves a correcting term to numerically integrate for an approximate solution. The equation for this integration scheme is shown below,

$$u_{n+1} = u_n + \Delta T \left(\frac{3}{2} f(u_n) - \frac{1}{2} f(u_{n-1}) \right)$$

Where, u_{n+1} is the updated (approximate) value of the next state. Whereas ΔT is the time-step being used through the integration scheme, and $f(u_n)$ is the derivative evaluated at the current time-step.

However, unlike Forward Euler this scheme relies on a previously solved for time step, so in order to get the integration started I used Fourth-Order Runge Kutta to approximate u_2 in the integration scheme and then continued with the Second-Order Adams-Bashforth integration nominally from the governing equation shown above.
The implementation of this scheme in this lunar lander truss simulation is shown below.

Algorithm 5: Implementation of a Second-Order Adams-Bashforth integration scheme.

```
function an = ab2(nodes,t)
    % Determine time-step and size of vector
    dt = t(2)-t(1);
    n = max(size(t));

    % Grab initial condition into column vector
    u = zeros(min(size(nodes))*max(size(nodes)), max(size(t)));
    u(:,1) = [ nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
               nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)';
               nodes(9,:)'];


    node_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)'];
    %Runge-Kutta Functions
    f0 = f(node_temp);
    n0  = [ f0(1:4)';f0(5:8)';f0(9:12)';f0(13:16)';f0(17:20)';f0(21:24)';
           f0(25:28)';f0(29:32)'; f0(33:36)'];

    f1 = f(node_temp +1/2*dt*n0);
    n1  = [ f1(1:4)';f1(5:8)';f1(9:12)';f1(13:16)';f1(17:20)';f1(21:24)';
           f1(25:28)';f1(29:32)'; f1(33:36)'];

    f2 = f(node_temp+1/2*dt*n1);
    n2  = [ f2(1:4)';f2(5:8)';f2(9:12)';f2(13:16)';f2(17:20)';f2(21:24)';
           f2(25:28)';f2(29:32)'; f2(33:36)'];

    f3 = f(node_temp+dt*n2);
    u(:,2) = u(:,1) + 1/6*dt *(f0+2*f1+2*f2+f3);

    for i = 2:(n-1)
        % Forward step in time
        nodetemp = [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)'];
        nodetemp_prime = [u(1:4,(i-1))';u(5:8,(i-1))';u(9:12,(i-1))';
        u(13:16,(i-1))';u(17:20,(i-1))';u(21:24,(i-1))';u(25:28,(i-1))';
        u(29:32,(i-1))'; u(33:36,(i-1))'];

        u(:,i+1) = u(:,i) + dt.*(3/2.*f(nodetemp) -1/2.*f(nodetemp_prime));
    end
    an = u;
end
```

C.3 Fourth-Order Runge Kutta Implementation

The Forward Euler is one of the simplest finite-difference methods to numerically integrate for an approximate solution. The equation for this integration scheme is shown below,

$$
\begin{aligned}
f_0 &= f\left(u_n\right) \\
f_1 &= f\left(u_n + \frac{1}{2}\Delta T f_0\right) \\
f_2 &= f\left(u_n + \frac{1}{2}\Delta T f_1\right) \\
f_3 &= f\left(u_n + \Delta T f_2\right) \\
u_{n+1} &= u_n + \frac{\Delta T}{6}\left(f_0 + 2f_1 + 2f_2 + f_3\right)
\end{aligned}
$$

Where, like the other integration schemes u_{n+1} is the updated (approximate) value of the next state. Whereas ΔT is the time-step being used through the integration scheme, and $f(u_n)$ is the derivative evaluated at the current time-step.

The implementation of this scheme in this lunar lander truss simulation is shown below.

Algorithm 6: Implementation of a Fourth-Order Runge Kutta scheme.

```
function an = rk4(nodes,t)
    % Determine time-step and size of vector
    dt = t(2)-t(1);
    n = max(size(t));

    % Grab initial condition into column vector
    u = zeros(min(size(nodes))*max(size(nodes)), max(size(t)));
    u(:,1) = [ nodes(1,:)'; nodes(2,:)'; nodes(3,:)'; nodes(4,:)';
               nodes(5,:)'; nodes(6,:)'; nodes(7,:)'; nodes(8,:)';
               nodes(9,:)'];

    for i = 1:(n-1)
        nodetemp = [  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)'];
        %Runge-Kutta Functions
        f0 = f(nodetemp);
        n0  = [ f0(1:4)';f0(5:8)';f0(9:12)';f0(13:16)';f0(17:20)';
                f0(21:24)';f0(25:28)';f0(29:32)'; f0(33:36)'];

        f1 = f(nodetemp + 1/2*dt*n0);
        n1  = [ f1(1:4)';f1(5:8)';f1(9:12)';f1(13:16)';f1(17:20)';
                f1(21:24)';f1(25:28)';f1(29:32)'; f1(33:36)'];

        f2 = f(nodetemp + 1/2*dt*n1);
        n2  = [ f2(1:4)';f2(5:8)';f2(9:12)';f2(13:16)';f2(17:20)';
                f2(21:24)';f2(25:28)';f2(29:32)'; f2(33:36)'];

        f3 = f(nodetemp + dt*n2);

        u(:,i+1) = u(:,i) + 1/6*dt *(f0+2*f1+2*f2+f3);
    end
    an = u;
end
```

D Function to Approximate Jacobian Matrix

This function allows for the generation of the Jacobian matrix of the function **f**. This function implements Equation 6 and numerically approximates the corresponding Jacobian to determine the governing eigenvalues behind the physics of this truss deformation.

Algorithm 7: Approximating the Jacobian Matrix of **f**.

```
function an = makemat(nodes)
    % Constants and pre-allocation
    epsi = 10^(-6);
    dfdu = zeros(22);
    fu0 = f(nodes);

    % Iterate through the five free-nodes over the x,y, velx, vely
    k = 1;
    for i = 1:5
        for j = 1:4
            % Create temporary variable
            nodetemp = nodes;

            % Add small perturbation to temporary value
            nodetemp(i,j) = nodetemp(i, j) + epsi;
            fd = f(nodetemp);

            % Numerical derivative
            dfdu(:,k) = (fd(1:22) - fu0(1:22))/epsi;
            k = k + 1;
        end
    end
    an = dfdu;
end
```

E Convergence Study Function

This function determines the total error in the height of the lunar lander to determine the order of accuracy for each integration scheme. This function calls to a previously generated "exact" solution that is a numerical approximation that has $100,010$ steps using Fourth-Order Runge Kutta integration scheme.

Algorithm 8: Determining Convergence of Integration schemes.

```
function an = calcerr(u)
    % Find the error, where calcexact is a function that create the exact
    %   solution for what ever numerical answer is in the function

    % uex is the "exact" solution, or an rk4 integration with 100k steps
    uex = cell2mat(struct2cell(load('smalldt_ex.mat')));

    an = abs(uex(30, end) - u(30,end));
end
```

F Miscellaneous Function

This function was generated for ease of use to generate plots of the lunar landers truss and show how it deforms.

Algorithm 9: General Function to Generate the Plots of the Truss.

```
function plot_truss(dat, tlin, savefig)
    % Determine the size of the matrix
    [~, num] = size(dat);
    k = 0;

    if savefig == 0

        vals = 1:100:num;
    else
        vals = (1:9).*200 + ones(1,9);
    end

    % Plot nodes/links
    figure()
    for i = vals
        nodes = [dat(1:2,i)'; dat(5:6,i)'; dat(9:10,i)';
            dat(13:14,i)'; dat(17:18,i)'; dat(21:22,i)';
            dat(25:26,i)'; dat(29:30,i)'; dat(33:34,i)'];
        links = [nodes(1,1:2), nodes(2,1:2);
         nodes(1,1:2), nodes(3,1:2);
         nodes(1,1:2), nodes(4,1:2);
         nodes(1,1:2), nodes(6,1:2);
         nodes(2,1:2), nodes(4,1:2);
         nodes(2,1:2), nodes(6,1:2);
         nodes(2,1:2), nodes(7,1:2);
         nodes(3,1:2), nodes(4,1:2);
         nodes(3,1:2), nodes(5,1:2);
         nodes(3,1:2), nodes(8,1:2);
         nodes(4,1:2), nodes(5,1:2);
         nodes(5,1:2), nodes(8,1:2);
         nodes(5,1:2), nodes(9,1:2)];

        scatter(nodes(:,1), nodes(:,2), 75, 'ko')
        hold on
        xlim([0, 0.8])
        ylim([0, 1])
        ax = gca;
        ax.FontSize = 20;
        xlabel('Position, m')
        ylabel('Position, m')
        for j = 1: max(size(links))
         plot([links(j,1), links(j,3)], [links(j,2),links(j,4)],'k', 'linewidth',
             1.8)
        end
        text(0.3,0.8,['Time = ',num2str(tlin(i))],'fontsize', 20)
        hold off
        if savefig == 1
            set(gcf, 'Color', 'w', 'Position', [400 400 400 400]);
            export_fig(['deform_',num2str(k), '.eps'])

            k = k + 1;
        end

        pause(0.05)
    end
end
```