EEE3039 MATLAB Coursework Assignment 2024/25

URN: 6534278
Department of Computer Science and Electronic Engineering

November 22, 2024

Contents

Chapter 1

Two-body Problem

This part of the report is focused on explaining the underlying physics and coding procedure for solving the two-body problem. Figures created by the MATLAB script will be used to visualise the topics discussed. The MATLAB code is included in the submission zip file and the Appendix section. Most of the equations and theory used in this chapter can be found in [1],[2],[3].
Figures that display a detailed 3D mesh of the Earth use an external Matlab function (`make_earth.m`) that was not developed by the author of this report. More information on the custom mesh can be found in [4]. In case the user does not have the function file, a grey mesh of the Earth will be used. All other elements of the plots will stay the same.

1.1 Kepler's Equation

The first task of this assignment is centred around using Kepler's equation to obtain values for the Eccentric E and True anomalies θ. The equation in its basic form can be seen in Equation 1.1 below.

$$M = E - e\ \sin(E) \tag{1.1}$$

For this to become usable in Newton's method for finding E, we need all elements on one side of the equal sign. This can be seen in Equation 1.2.

$$E - e\ \sin(E) - M = 0 \tag{1.2}$$

Having placed all variables on one side of the equal sign, we can use Newton's method seen in Equation 1.3.

$$f(x^{(0)} + \delta x^{(0)}) = f(x^{(0)}) + f'(x^{(0)})\,\delta x^{(0)} + \text{H.O.T.} \tag{1.3}$$

Assuming that $f(x^{(0)} + \delta x^{(0)}) \approx 0$ and H.O.T ≈ 0 this would lead to $\delta(x^0)$ following Equation 1.4.

$$\delta x^{(0)} = -\left(\frac{f(x^{(0)})}{f'(x^{(0)})}\right) \tag{1.4}$$

This naturally means that we need the first derivative of our Equation 1.2 for the Eccentric Anomaly E. This would lead to Equation 1.5.

$$f(E)' = \frac{df(E)}{dE} = 1 - e\ \cos(E) = 0 \tag{1.5}$$

With Equations 1.2,1.5 and 1.4 we can estimate a value for the Eccentric anomaly. A tolerance of $|\delta x^{(0)}| < 10^{-10}$ was used and the initial guess was set at $E = M_0$ where M_0 is the initial mean anomaly. The code for this Matlab function can be found in Appendix B.1. After 4 iterations of the loop, a value of $E = 23.9^o$ was found. For a value of the True anomaly θ to be found we need to rearrange Equation 1.6 into Equation 1.7.

$$\tan\left(\frac{\theta}{2}\right) = \sqrt{\frac{1+e}{1-e}}\tan\left(\frac{E}{2}\right) \tag{1.6}$$

$$\theta = 2\tan^{-1}\left(\sqrt{\frac{1+e}{1-e}}\tan\left(\frac{E}{2}\right)\right) \tag{1.7}$$

Inputting the values obtained in the Kepler function we can get to a value of $\theta = 49.9^o$ which means that the satellite is rather close to the periapsis ($\theta = 0^o$). Matlab offers its inbuilt `fzero` function. It was used to test the validity of our results and computed the same values as our custom function.

1.2 Orbit Propagation

To extrapolate the values of E, M and θ we first need to calculate the period of our orbit. To do so we first calculate the mean motion n of our orbiting satellite through the use of Equation 1.8.

$$n = \sqrt{\frac{\mu}{a^3}} \tag{1.8}$$

where μ is the gravitational parameter of the body being orbited and a is the semi-major axis of the orbit. Knowing what our mean motion we can calculate the period of our orbit by simply multiplying it by 2π. The reasoning behind this equation is that the period is the time it takes for our satellite to complete a full rotation around the central body being orbited. The relation can be seen in Equation 1.9.

$$P = 2\pi\sqrt{\frac{a^3}{\mu}} = \frac{2\pi}{n} \tag{1.9}$$

Knowing both n and P we can extrapolate the data for the Mean Anomaly easily in time by using Equation 1.10.

$$M = M_0 + n(t - t_0) \tag{1.10}$$

Where t_0 is the initial time and M_0 is the Mean anomaly at time t_0.
In the case of the Part 1 main script `cw1.m` (Appendix A.1), it was assumed that $t_0 = 0$ $[s]$ and the t stood for time increments of an n-number of orbits. This produces a range of values for the mean anomaly which is very useful as the previous section discussed how this enables the calculation of both E and θ.
With all three anomalies calculated, a plot of their values can be made in Matlab. Figure 1.1 shows the plotted E, θ and M anomaly data for one orbit and is an important result in validating the work done up to this point.

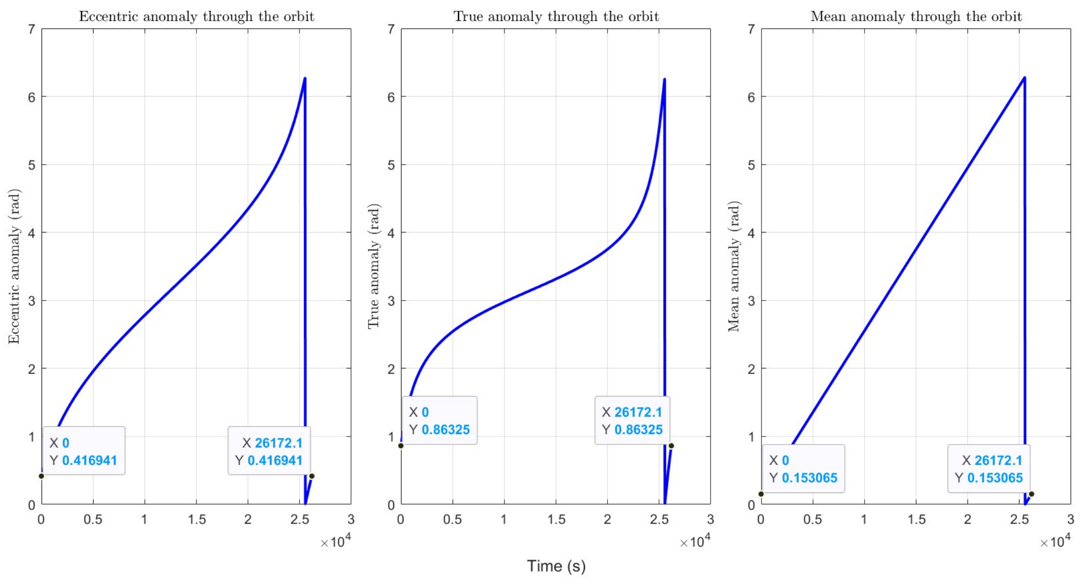

Figure 1.1: All three anomalies side by side. This is a useful result for validation as the shape of the three anomalies aligns with what would be expected of the orbit.

The shape of the curves here is what would be expected from orbital motion theory. The most obvious one is the mean anomaly increasing by a flat amount over time. This is true as M is controlled by the mean motion of the satellite. Both θ and E have more curvy shapes as these take into account the eccentricity of the orbit and change in orbital speed at different points in the trajectory. Another note here is that the start and end points of the graphs align on the same Y-axis values when the data is passed through a computational modulus (%) operator. This translates to the fact that a full orbit has been completed in the simulation.
With this information now available, it is possible to get coordinates and velocity data based on the orbital elements available. A key parameter here is the range of values we have obtained for θ. For any moment in the orbit, we can find the magnitude of the position vector, from the center of the planet to the satellite. We can do so using Equation 1.11.

$$r = \frac{a(1-e^2)}{(1+e\cos\theta)} \tag{1.11}$$

If the magnitude r of the position vector, $\mathbf{r}$, is known then the components of said vector can also be found using Equation 1.12:

$$\mathbf{r} = \begin{bmatrix} r\cos\theta \\ r\sin\theta \\ 0 \end{bmatrix} \tag{1.12}$$

To find the change in the position vector, $\mathbf{r}$, i.e. the velocity of the satellite, we first need the angular momentum. This can be found using Equation 1.13.

$$h = \sqrt{(\mu a(1-e^2))} \tag{1.13}$$

Once this value is available, similarly to the position vector, the velocity vector can be simply found using Equation 1.14:

$$\mathbf{v} = \begin{bmatrix} -\frac{\mu}{h}\sin\theta \\ \frac{\mu}{h}(\cos\theta + e) \\ 0 \end{bmatrix} \tag{1.14}$$

There is one more step needed before these values are useful for us in the ECI frame. These two vectors, $\mathbf{r}$ and $\mathbf{v}$, are in the perifocal frame. A direction cosine matrix (DCM) is used to move from the perifocal to the Earth-centred inertial (ECI) frame. This 3-1-3 $[\mathcal{IP}]$ DCM matrix can be seen below:

$$[\mathcal{IP}] = R_3(\omega)R_1(i)R_3(\Omega) =$$
$$= \begin{bmatrix} \cos\Omega\cos\omega - \sin\Omega\cos i\sin\omega & -\cos\Omega\sin\omega - \sin\Omega\cos i\cos\omega & \sin\Omega\sin i \\ \sin\Omega\cos\omega + \cos\Omega\cos i\sin\omega & -\sin\Omega\sin\omega + \cos\Omega\cos i\cos\omega & -\cos\Omega\sin i \\ \sin i\sin\omega & \sin i\cos\omega & \cos i \end{bmatrix}$$

We can apply this DCM matrix to both our newly defined $\mathbf{r}$ and $\mathbf{v}$ to get the ECI orbital data as seen in Equations 1.15 and 1.16.

$${}^{\mathcal{I}}\mathbf{r} = [\mathcal{IP}]\ {}^{\mathcal{P}}\mathbf{r} \tag{1.15}$$

$${}^{\mathcal{I}}\mathbf{v} = [\mathcal{IP}]\ {}^{\mathcal{P}}\mathbf{v} \tag{1.16}$$

After changing to the appropriate frame and plotting the results in Matlab yields Figures 1.2 and 1.3.

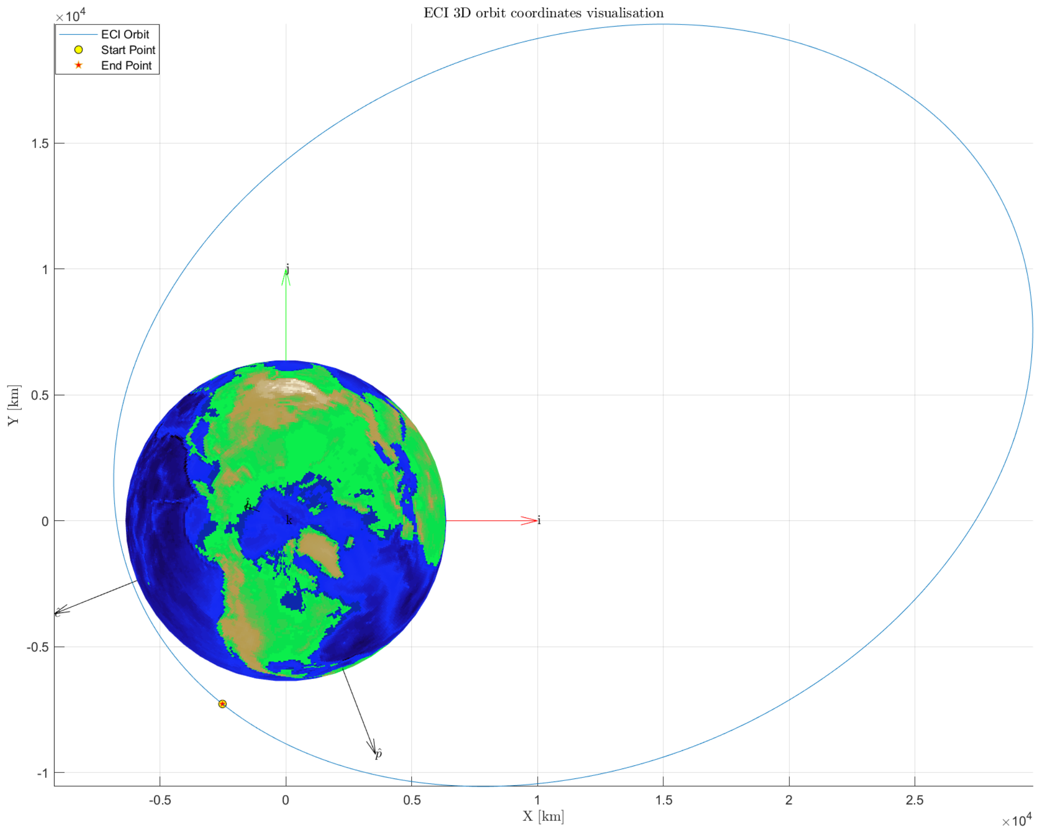

Figure 1.2: 2D view of ECI orbit extracted from orbital elements. Start and End point markers overlap showing a necessary validation of proper full orbital period propagation.

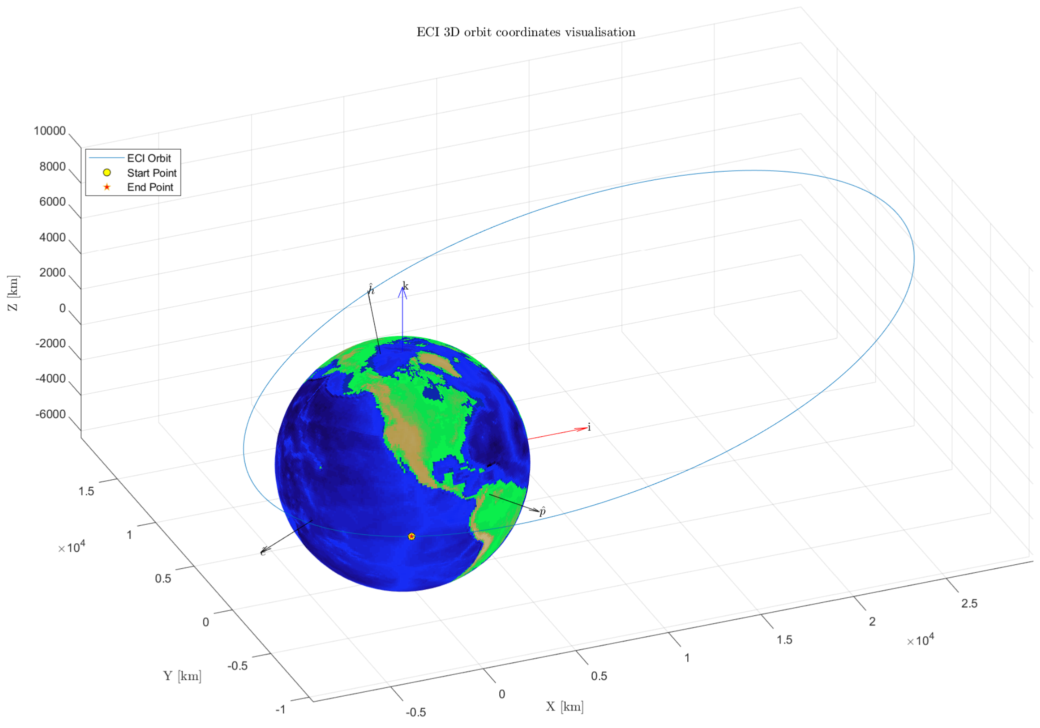

Figure 1.3: 3D view of ECI orbit extracted from orbital elements.

This is an important result as it validates a few key points set out in the assignment. Firstly, as seen with Figure 1.1, we have a full orbit propagation with the starting and end points overlapping. This can further be illustrated with the state vector values seen below:

$$X_{start} = \begin{bmatrix} -2.5202 \text{ km} \\ -7.2774 \text{ km} \\ 0.0031 \text{ km} \\ 0.0070 \text{ km/s} \\ -0.0056 \text{ km/s} \\ 0.0015 \text{ km/s} \end{bmatrix} \times 10^3 \qquad X_{end} = \begin{bmatrix} -2.5202 \text{ km} \\ -7.2774 \text{ km} \\ 0.0031 \text{ km} \\ 0.0070 \text{ km/s} \\ -0.0056 \text{ km/s} \\ 0.0015 \text{ km/s} \end{bmatrix} \times 10^3$$

Additionally, due to the ability to easily plot in 3D space in Matlab, we can see in Figure 1.3 that the orbit matches the orbital elements described in the coursework assignment. All of the described mathematics and logic can be seen in code form under Appendix B.3.

1.3 Numerical Integration in ECI Frame

Matlab offers a powerful tool with its ordinary differential equation (ODE) functions. These are based on the 4th Order Runge-Kutta technique and for this assignment, we are focused on using the `ode45` function [5].
The equation of motion for the acceleration of a body orbiting another is seen in Equation 1.17.

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} \tag{1.17}$$

For this equation to be usable in Matlab however we need to use a "dummy" variable to make it work within the code environment. This means that the second derivative of the position vector $\ddot{\mathbf{r}}$, needs to be expressed as a first derivative. The first derivative of the position vector is the velocity vector $\mathbf{v}$. This gives a simple solution to our problem, and that is simply to use the first derivative of $\dot{\mathbf{v}}$ to represent $\ddot{\mathbf{r}}$. This can be seen formalised in Equation:

$$\ddot{\mathbf{r}} = \dot{\mathbf{v}} \tag{1.18}$$

Knowing Equations 1.17 and 1.18 we can construct an `ode45` function in Matlab. The function would do its numerical integration on the velocity and acceleration components and give back the integrated position and velocity components of the state vector X. The discussed logic and mathematics can be seen in code form under Appendix B.5.
Once the numerical integrator is done with its computation tasks we can plot the corresponding ode state vector data on top of the the original one obtained from the classical orbital elements. This result can be seen in Figures 1.4 and 1.5.

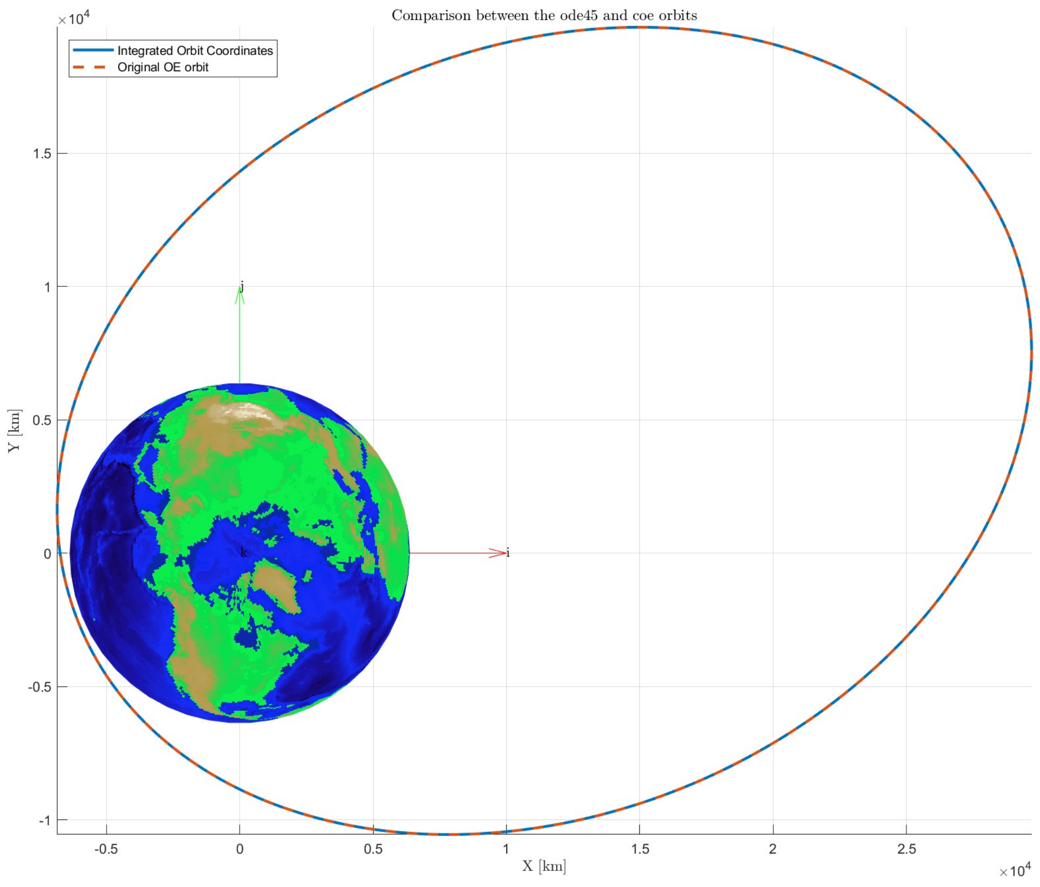

Figure 1.4: 2D view of ECI orbits from COE and ODE integration. There is an almost perfect overlap of the data points which serves as crucial confirmation of correct integration.

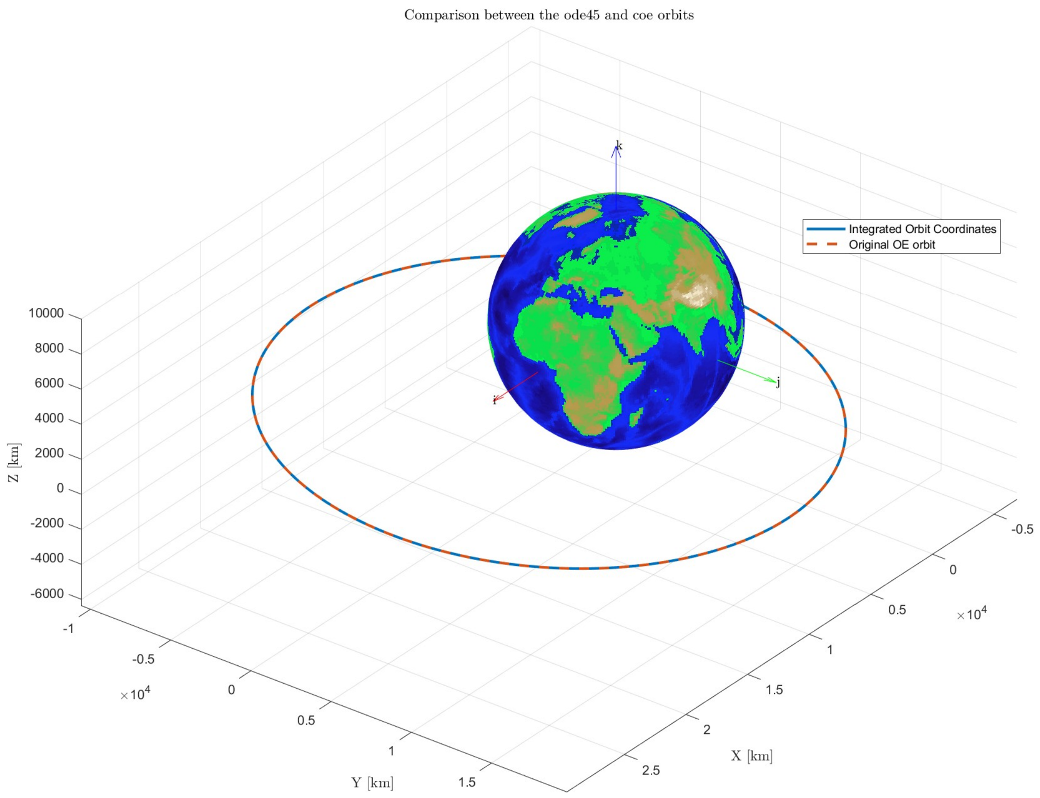

Figure 1.5: 3D view of ECI orbits from COE and ODE integration.

This result is what we would expect to see as both the COE and `ode45` paths of orbital propagation

should give identical results. This serves as a validation of the work done in the previous section as well.
Still, there are some differences in the results between the two orbital propagation methods. Figure 1.6 shows the error between the COE and `ode45` data points obtained. These values are relatively tiny in orders of magnitude and thus the plotted data looks identical. This is expected as the Runge-Kutta Fourth Order method is a numerical integration and would give "perfect" results. Still, this figure shows a remarkably good agreement between theory and numerical tests.

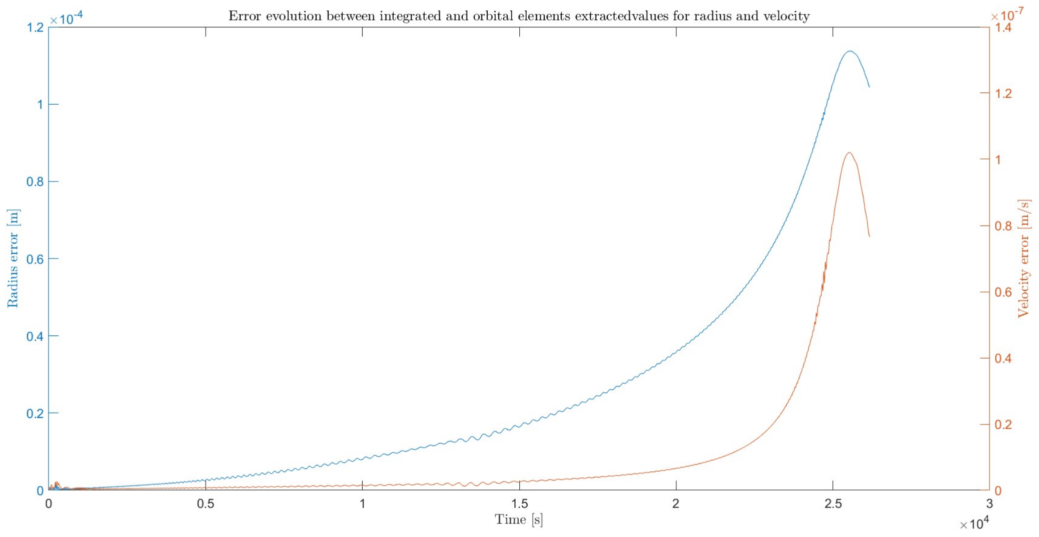

Figure 1.6: Position and velocity differences between COE and ODE45 orbit propagation methods. The differences seen here are very small and can be attributed to the machine accuracy of the numerical integration used when using the ODE45 function.

Another form of testing can be done when looking at the specific angular momentum of the system. As mentioned in a previous section the specific angular momentum of a two-body system is as seen in Equation 1.13. Similarly, we can find the angular momentum by taking the cross product of the position, **r**, and velocity vector, **v**, of the spacecraft. This is formalised in Equation 1.19.

$$\mathbf{h} = \mathbf{r} \times \mathbf{v} \tag{1.19}$$

The integration process has left us with a state vector for 1000 time increments. Thus we can use Equation 1.19 to find the angular momentum at each increment and plot it against the static theoretical value found in Equation 1.13. Doing so produces Figure 1.7.

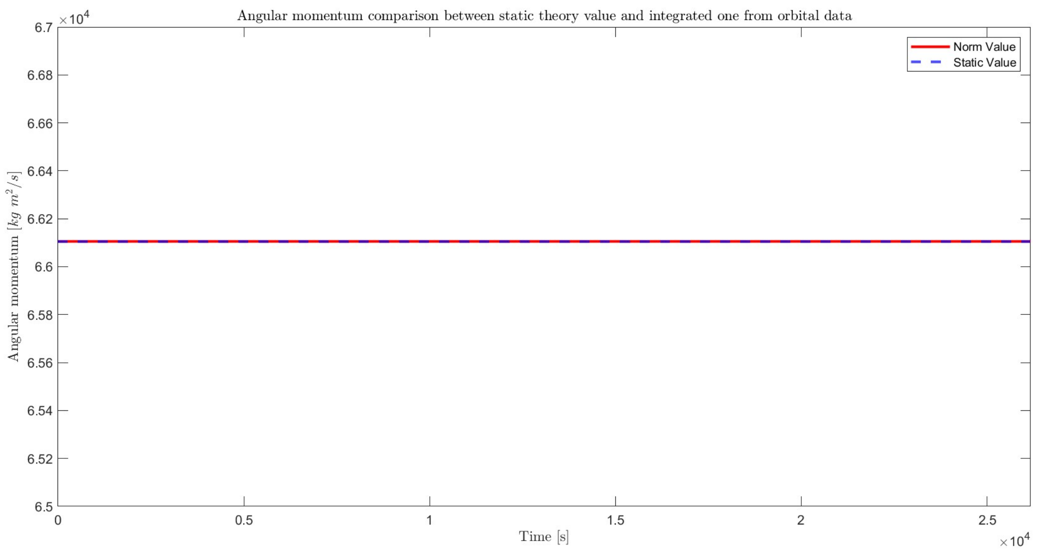

Figure 1.7: Theoretical and numerical (norm) comparison of the specific angular momentum of the system. The result demonstrated here is significant as it demonstrates an agreement between the expected theoretical value and the numerically integrated one.

As can be seen there is near perfect agreement between theory and numerical testing. The use of "near perfect" wording here is deliberate as the figure has "zoomed out" on the y-axis to account for machine accuracy discrepancies that arise from the results seen in Figure 1.6. Still, the error is orders of magnitude smaller than the actual norm-ed values and this is considered a successful test.

1.4 Equations of Motion in the ECEF Frame

To move from the ECI to the ECEF reference frame we can utilise the Transport theorem to convert the elements of the state vector. The transport theorem can be seen in Equation 1.20.

$$^{\mathcal{I}}\ddot{\mathbf{r}} = \left(^{\mathcal{F}}\ddot{\mathbf{r}}\right) + \alpha_{\oplus} \times \mathbf{r} + \mathbf{2}\omega_{\oplus} \times \left(^{\mathcal{F}}\dot{\mathbf{r}}\right) + \omega_{\oplus} \times (\omega_{\oplus} \times \mathbf{r}) \tag{1.20}$$

Where $\alpha_{\oplus}$ and $\omega_{\oplus}$ are the angular acceleration and angular velocity of the Earth, respectively. We can rewrite this equation to show to the ECEF ($\mathcal{F}$) frame rather than the Inertial ($\mathcal{I}$) by rearranging it for $^{\mathcal{F}}\ddot{\mathbf{r}}$. We can also notice that the $\alpha_{\oplus} = 0\ \text{rad/s}^2$ as the Earth spins at a constant rate. Doing the mentioned steps we get to Equation 1.21.

$$^{\mathcal{F}}\ddot{\mathbf{r}} = \left(^{\mathcal{I}}\ddot{\mathbf{r}}\right) - 2\omega_{\oplus} \times \left(^{\mathcal{F}}\dot{\mathbf{r}}\right) - \omega_{\oplus} \times (\omega_{\oplus} \times \mathbf{r}) \tag{1.21}$$

Due to the need to work within the `ode45` framework, it is necessary to represent the equation in the form of a first-order differential equation. To do so we will use the same logic used in Equation 1.18. This translates to Equation 1.22.

$$^{\mathcal{F}}\dot{\mathbf{v}} = -\frac{\mu}{r^3}\mathbf{r} - \mathbf{2}\omega_{\oplus} \times \left(^{\mathcal{F}}\mathbf{v}\right) - \omega_{\oplus} \times (\omega_{\oplus} \times \mathbf{r}) \tag{1.22}$$

The last part of the state vector representation needed is the velocity in the $\mathcal{F}$ frame and this can also be obtained using the transport theorem in Equation

$$^{\mathcal{F}}\mathbf{v} = \left(^{\mathcal{I}}\mathbf{v}\right) - \omega_{\oplus} \times \mathbf{r} \tag{1.23}$$

Using both Equations 1.22 and 1.23 we can construct an `ode45` function and let it loop for the selected amount of orbits and time increments. This can be seen in code form under Appendix B.6.
The result of this operation for 10 orbits can be seen in Figures 1.8 and 1.9.

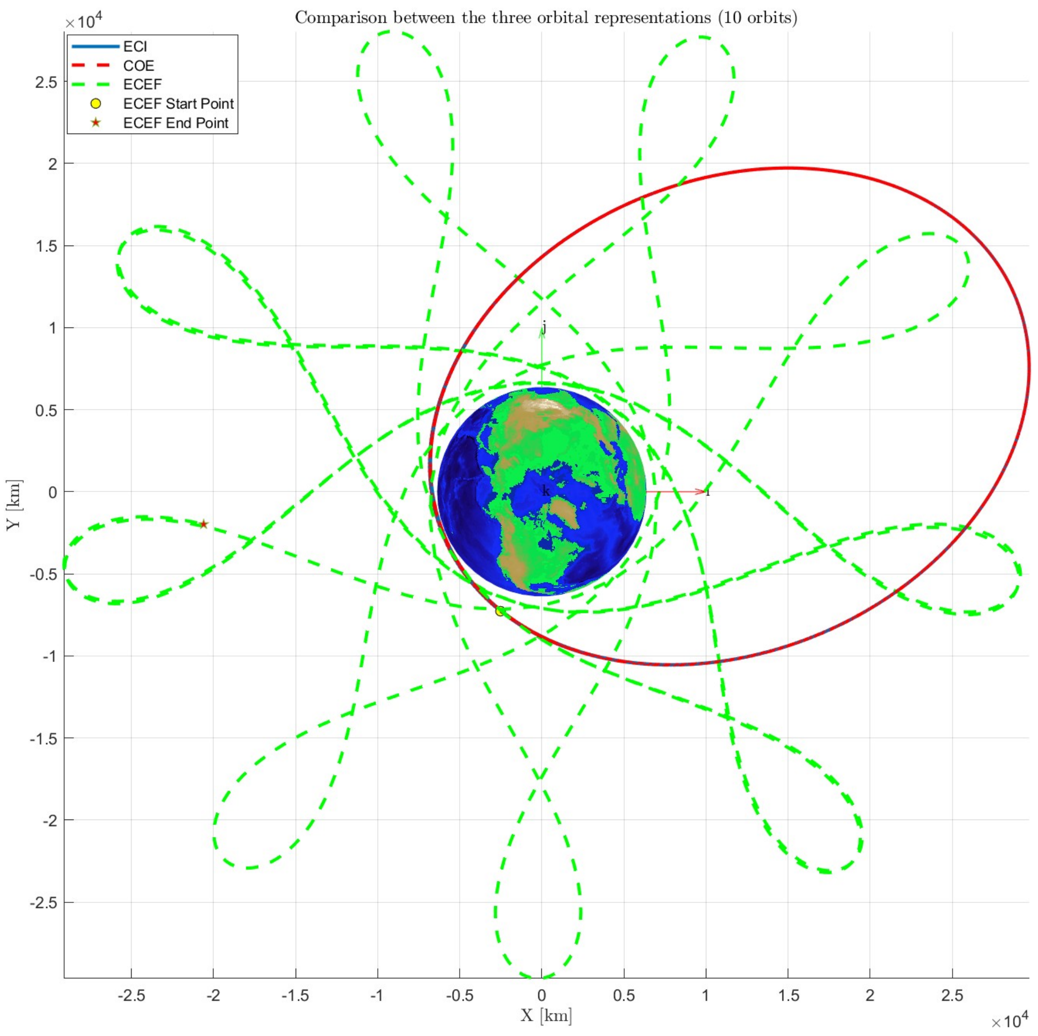

Figure 1.8: Propagation of 10 orbits in the ECEF frame from a top-down view (2D). The previous COE and ECI orbits are present for reference. The ECEF representation is not as intuitive as the ECI one. The "flower" pattern does make sense as the Earth rotates and distorts the otherwise elliptical orbit as seen from the ECI frame.

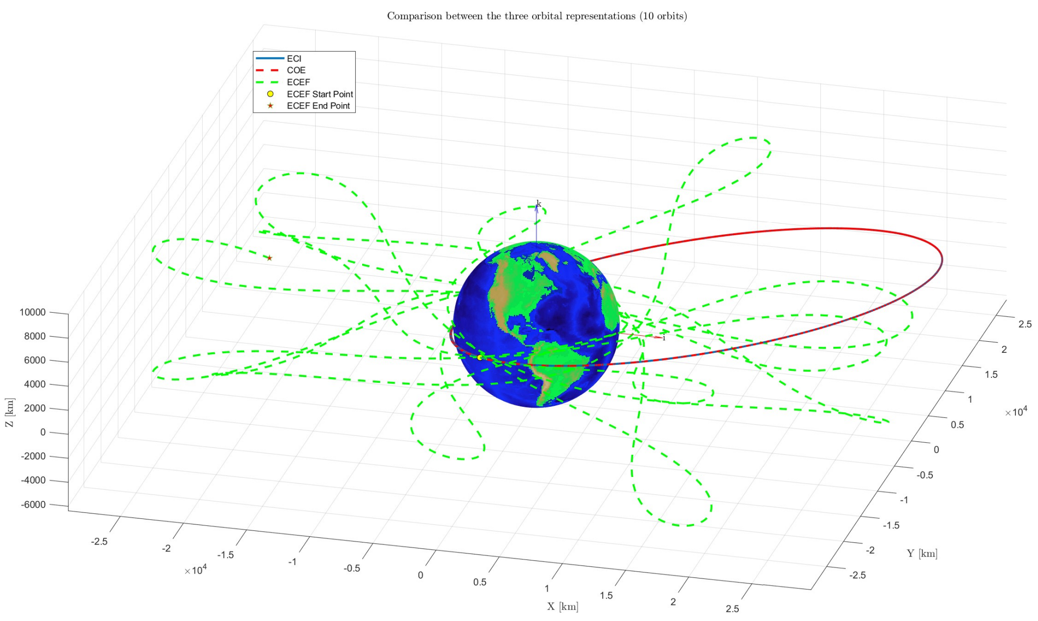

Figure 1.9: Propagation of 10 orbits in the ECEF frame (3D)

The ECEF frame is not as intuitive to parse as the ECI one. Here due to the rotation of the Earth, the orbit forms a "flower" like pattern. Additionally, due to the orbit having a different orbital period than that of Earth's rotational velocity ($P_{\text{satellite}} \neq P_{\omega_\oplus}$), the start and end points of this satellite would not be at the same point. More specifically, the values of the state vector at the beginning and end of the ECEF 10 orbit propagation can be seen below:

$$X_{\text{ECEF}_{\text{start}}} = \begin{bmatrix} -2.5202 \text{ km} \\ -7.2774 \text{ km} \\ 0.0031 \text{ km} \\ 0.0070 \text{ km/s} \\ -0.0056 \text{ km/s} \\ 0.0015 \text{ km/s} \end{bmatrix} \times 10^3 \qquad X_{\text{ECEF}_{\text{end}}} = \begin{bmatrix} -2.0602 \text{ km} \\ -0.1984 \text{ km} \\ 0.4043 \text{ km} \\ 0.0003 \text{ km/s} \\ -0.0002 \text{ km/s} \\ 0.0000 \text{ km/s} \end{bmatrix} \times 10^4$$

To get a better understanding of our result and to make sure that the calculations were correct, there is a need for a sanity check on the ECEF data.

1.4.1 Sanity Check for the ECEF Frame

As stated the ECEF frame representation is not easy to determine if it has been done correctly from intuition as the COE and ECI ones. To test if the integration has been done correctly, we can use a case example where we know what the ECEF frame should look like. The example in question is a Geostationary orbit (GEO). Geostationary satellites rotate with the angular velocity of the Earth and their orbital period is equal to that of a sidereal day. In other words, they stay over the same point in the sky in the ECEF reference frame. If the mathematical logic employed using Equations 1.22 and 1.23 and the code from Appendix B.6 are valid, then the plot of the ECEF orbit would be a stack of points near the initial position of the satellite in the simulation. Figure 1.10 demonstrates this plot.

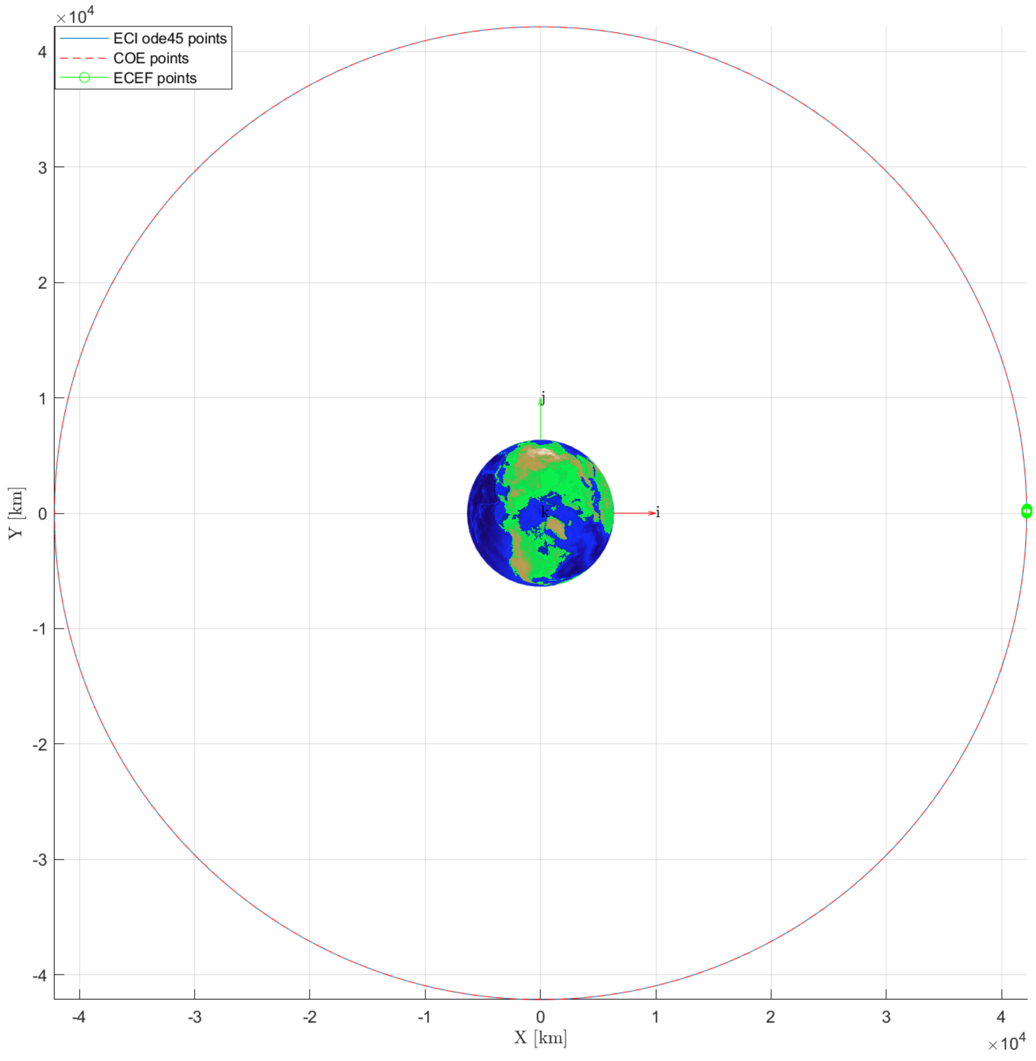

Figure 1.10: Geostationary test for the ECEF numeric integration. As expected the orbit essentially stays in one place for the duration of the orbit. This serves as a validation result for the work done in this specific section

The code for this test can be found in Appendix A.4. The initial orbital elements given in this assignment were simply changed to match those of a geostationary orbit. The script is mostly identical to `cw1.m` (Appendix A.1) aside from this.
The plot of the graph shows the exact behaviour that was expected from the theory. The stack of points does indeed stay stationary for one orbit over the same spot. In reality, it is not a singular point as the numeric integration introduces small errors to the state vector variables. Still, this was a useful test as it shows a successful implementation of the equations of motion and transport theorem for the case of an ECEF reference frame.

Chapter 2

Attitude Representation of Torque-Free Motion

The second part of the coursework is done on a separate script found in Appendix A.2. This is done as orbit propagation and attitude representation are detached problems and can be solved separately. This is not always the case, but in this assignment, it is a valid approach. Additionally, this improves the readability of the scripts.
The assignment also stated that the data below is valid at the beginning of this attitude simulation. Most of the equations and theory mentioned in this chapter can be found in [1],[2],[3].

$$\mathbf{r} = \begin{bmatrix} 5300.64 \\ 17575.73 \\ 138.50 \end{bmatrix} \text{ [km]} \qquad \mathbf{v} = \begin{bmatrix} -4.2880 \\ -1.9373 \\ 0.6026 \end{bmatrix} \text{ [km/s]}$$

2.1 Direction Cosine Matrices

To move from the ECI $\mathcal{I}$ frame to the NTW $\mathcal{T}$ frame we need to understand how the NTW axis unit vectors are constructed. Equation 2.1 demonstrates the tangential component (T) of the frame along the orbit. This unit vector is aligned with the velocity vector of the spacecraft.

$$\hat{e}_t = \frac{\mathbf{v}}{v} \tag{2.1}$$

The second defined vector is the orbital one (W component). This unit vector is aligned with the angular momentum of the spacecraft, which by definition is orthogonal to the orbital plane and therefore to $\hat{e}_t$ (T component). The mathematical definition is seen in Equation 2.2.

$$\hat{e}_h = \frac{\mathbf{h}}{h} \tag{2.2}$$

The final part of the triad is a unit vector created from the cross product of the $\hat{e}_t$ and $\hat{e}_h$. This final unit vector (N component) is seen in Equation 2.3.

$$\hat{e}_n = \hat{e}_t \times \hat{e}_h \tag{2.3}$$

With the definitions seen in Equations 2.1, 2.2, and 2.3 we have the three necessary unit vector definitions for the creation of the $[\mathcal{IT}]/[\mathcal{NTW}]$ frame. These are equivalent as they map the transition between the inertial and NTW frames. This can be seen formalised in Equation 2.4.

$$[\mathcal{NTW}] = [\mathcal{IT}] = [\hat{e}_t|\hat{e}_h|\hat{e}_n] \tag{2.4}$$

Stacking these unit vectors vertically yields the necessary mapping between the two frames. Numerically, this matrix can be observed below:

$$[\mathcal{IT}] = \begin{bmatrix} -0.3949 & -0.9039 & -0.1643 \\ 0.9110 & -0.4084 & 0.0573 \\ -0.1189 & -0.1270 & 0.9847 \end{bmatrix}$$

Due to this being a DCM matrix it should poses the property of orthogonality. This means that the inverse of the matrix is the same as the transpose. This quality is very useful, computationally. Testing this in the Matlab script, available in Appendix A.2, it can be verified that Equation 2.5 is valid.

$$\text{inv}([\mathcal{IT}]) = [\mathcal{IT}]^{\text{T}} = [\mathcal{TI}] = \begin{bmatrix} -0.3949 & 0.9110 & -0.1189 \\ -0.9039 & -0.4084 & -0.1270 \\ -0.1643 & 0.0573 & 0.9847 \end{bmatrix} \tag{2.5}$$

The orthogonality of the DCM is an important validation of the system. This means that we are indeed working with a DCM that can map coordinates between two reference frames.
The assignment supplies an additional DCM matrix that maps between the NTW and body frame of the satellite see below:

$$[\mathcal{BT}] = \begin{bmatrix} 0.7146 & 0.6131 & -0.3368 \\ -0.6337 & 0.7713 & 0.0594 \\ 0.2962 & 0.1710 & 0.9397 \end{bmatrix}$$

To move from the inertial to the body frame of the satellite we can utilise the rule of DCMs where the identical middle frame components effectively "cancel out" during multiplication. The move from inertial to the body frame can be seen in Equation 2.6.

$$[\mathcal{BI}] = [\mathcal{BT}][\mathcal{TI}] = \begin{bmatrix} -0.7810 & 0.3813 & -0.4945 \\ -0.4567 & -0.8889 & 0.0358 \\ -0.4259 & 0.2538 & 0.8684 \end{bmatrix} \tag{2.6}$$

The assignment gave the final result for the $[\mathcal{BI}]$ and the one obtained in the `cw2.m` is identical.

2.2 Attitude Representations

There are many ways to represent the attitude of a body. The most simple way to do so is using Euler angles. While very intuitive and easy to use, Euler angles have some innate disadvantages related to singularities that occur at some specific orientations. These are more commonly referred to as "Gimbal Locks". Another way to describe the orientation and attitude of a body is by using the Euler Principle axis and angle. This concept combines the Euler angle sequential rotations and applies the attitude representation of the body through a single rotation around a specific vector. The principle Euler angle can be defined from DCM components using Equation 2.7.

$$\Phi = \cos^{-} 1 \left(\frac{\text{tr}([\mathcal{BI}]) - 1}{2} \right) \tag{2.7}$$

The separate components of the Euler axis can be found using Equations 2.8, 2.9, 2.10.

$$e_1 = \frac{[\mathcal{BI}]_{2,3} - [\mathcal{BI}]_{3,2}}{2\sin(\Phi)} \tag{2.8}$$

$$e_2 = \frac{[\mathcal{BI}]_{1,3} - [\mathcal{BI}]_{3,1}}{2\sin(\Phi)} \tag{2.9}$$

$$e_3 = \frac{[\mathcal{BI}]_{2,1} - [\mathcal{BI}]_{1,2}}{2\sin(\Phi)} \tag{2.10}$$

To check if the mathematical operations were carried out correctly, we can check if the norm of the principle axis $|\mathbf{e}| = \mathbf{1}$. This would validate the work demonstrated so far. This can be seen formalised in Equation 2.11.

$$|\mathbf{e}| = \sqrt{e_1^2 + e_2^2 + e_3^2} = 1 \tag{2.11}$$

It can be checked in the `cw2.m` script that the value is equal to 1. With this done we can define our Euler Axes and Principle Angle representation of the attitude.

While this attitude representation can offer more flexibility than Euler angles for specific rotations. The principle angle and axes have their own singularity issues, however. For very small rotational changes for $\Phi \approx 0$ rad, this attitude representation can also suffer from mathematical singularities. To remedy both of these attitude representations problems we can use a quaternion solution instead. The quaternion, $\mathbf{q}$ is essentially an imaginary vector, with three imaginary vector components and a single real scalar quantity. The scalar quantity, often denoted as q_0, can be seen calculated from the $[\mathcal{BI}]$ DCM matrix components in Equation 2.12.

$$q_0 = \frac{\sqrt{\mathrm{tr}[\mathcal{BI}] + 1}}{2} \tag{2.12}$$

Where $\mathrm{tr}[\mathcal{BI}]$ is the trace of the matrix. The trace is the sum of the diagonal elements of a matrix. Similarly, the three vector quantities can be obtained through the use of Equation 2.13, 2.14 and 2.15.

$$q_1 = \frac{[\mathcal{BI}]_{2,3} - [\mathcal{BI}]_{3,2}}{4q_0} \tag{2.13}$$

$$q_2 = \frac{[\mathcal{BI}]_{1,3} - [\mathcal{BI}]_{3,1}}{4q_0} \tag{2.14}$$

$$q_3 = \frac{[\mathcal{BI}]_{2,1} - [\mathcal{BI}]_{1,2}}{4q_0} \tag{2.15}$$

Quaternions are very powerful as they do not introduce singularities for any attitude configuration. Additionally, quaternions are computationally efficient, meaning that they are quicker to calculate and implement within an algorithm or program. Unfortunately due to their imaginary components, quaternions are unintuitive and difficult to represent in the physical 3D world. There is some abstraction necessary to fully visualise them for each problem. It should be noted that quaternion notation is not standardised. Different variations of the $\mathbf{q}$ vector can be found including representing the vector quantities as q_1, q_2, q_3 and the scalar quantity as q_4.
A very simple way to check if the quaternion is properly calculated is to check the norm value, $|\mathbf{q}|$, as done in Equation 2.16.

$$|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1 \tag{2.16}$$

It can be verified in `cw2.m` (Appendix A.2) that this value is indeed equal to 1 in the Matlab environment.
Having tested different representations and discussed their advantages and disadvantages we can go back to the Euler angle representation. While the $[\mathcal{BI}]$ does not explicitly state what the Euler angles used for this specific DCM are, we can infer them from a few specific elements in the DCM. As we we are working with a 3-1-3 DCM we can recall its form from Equation 1.2. Here, however, we are using θ_1, θ_2 and θ_3. As the assignment requires it, $\alpha = \theta_1; \beta = \theta_2; \gamma = \theta_3$.

$$[\mathcal{BI}] = \begin{bmatrix} \cos\theta_3\cos\theta_1 - \sin\theta_3\cos\theta_2\sin\theta_1 & -\cos\theta_3\sin\theta_1 - \sin\theta_3\cos\theta_2\cos\theta_1 & \sin\theta_3\sin\theta_2 \\ \sin\theta_3\cos\theta_1 + \cos\theta_3\cos\theta_2\sin\theta_1 & -\sin\theta_3\sin\theta_1 + \cos\theta_3\cos\theta_2\cos\theta_1 & -\cos\theta_3\sin\theta_2 \\ \sin\theta_2\sin\theta_1 & \sin\theta_2\cos\theta_1 & \cos\theta_2 \end{bmatrix}$$

It can be seen that the $[\mathcal{BI}]_{3,3} = \cos\theta_2$. This means that we can do an $\cos^{-1}(\theta_2)$ to find the necessary second angle.

$$\theta_2 = \beta = \arccos\left([\mathcal{BI}]_{3,3}\right) = 0.5188 \text{ [rad]} \tag{2.17}$$

Additionally, it can be seen that if $[\mathcal{BI}]_{3,1}/[\mathcal{BI}]_{3,2} = \tan\theta_1$ and $[\mathcal{BI}]_{1,3}/[\mathcal{BI}]_{2,3} = \tan\theta_2$. We can take the arctan of the other side of the equal sign to find our last two Euler angles. This can be seen in Equations 2.18 and 2.19.

$$\theta_1 = \alpha = \arctan\left(\frac{[\mathcal{BI}]_{3,1}}{[\mathcal{BI}]_{3,2}}\right) = -1.0333 \text{ [rad]} \tag{2.18}$$

$$\theta_3 = \gamma = \arctan\left(\frac{[\mathcal{BI}]_{1,3}}{[\mathcal{BI}]_{2,3}}\right) = -1.4984 \text{ [rad]} \tag{2.19}$$

For this attitude coordinate set the kinematic differential relation is seen in Equation 2.20 [6].

$$\begin{bmatrix}\dot{\theta}_1\\ \dot{\theta}_2\\ \dot{\theta}_3\end{bmatrix} = B(\theta)\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix} \tag{2.20}$$

With $B(\theta)$ being equal to:

$$B(\theta) = \frac{1}{\sin(\theta_2)}\begin{bmatrix}\sin(\theta_3) & \cos(\theta_3) & 0\\ \sin(\theta_2)\cos(\theta_3) & -\sin(\theta_2)\sin(\theta_3) & 0\\ -\cos(\theta_2)\sin(\theta_3) & -\cos(\theta_2)\cos(\theta_3) & \sin(\theta_2)\end{bmatrix}$$

The values displayed are taken from the Matlab workspace. With the extracted Euler angle values we can proceed to the final part of this assignment.

2.3 Kinematics and Euler's Equations

The kinematics of our system can be explained by Euler's Equations. This collection of three equations can be seen in the system of Equations 2.21.

$$\begin{aligned} I_1\dot{\omega}_1 &= L_1 + (I_2 - I_3)\omega_2\omega_3,\\ I_2\dot{\omega}_2 &= L_2 + (I_3 - I_1)\omega_1\omega_3,\\ I_3\dot{\omega}_3 &= L_3 + (I_1 - I_2)\omega_1\omega_2. \end{aligned} \tag{2.21}$$

Due to this being a torque-free motion $L_1 = L_2 = L_3 = 0$ J. Additionally, we are interested in the $\dot{\omega}$ values, therefore, we can move the moments of inertia for each axis on the other side of the equal sign. These operations translate to what can be seen in the system of Equations 2.22.

$$\begin{aligned} \dot{\omega}_1 &= \frac{(I_2 - I_3)}{I_1}\omega_2\omega_3,\\ \dot{\omega}_2 &= \frac{(I_3 - I_1)}{I_2}\omega_1\omega_3,\\ \dot{\omega}_3 &= \frac{(I_1 - I_2)}{I_3}\omega_1\omega_2. \end{aligned} \tag{2.22}$$

To be able to fully express the state vector $X_{attitute}$ of the attitude system in Matlab we also need an expression for $\dot{\theta}$. This expression was made available in the previous section and is derived in Equation 2.20. With all six components for all 3 axes of rotation, a state vector for the spacecraft can be made. All of these steps and logic can be seen in code form under Appendix B.7.
Once again the `ode45` function was used to propagate the attitude of the satellite within a time frame between 0 and 3600s. The interval was set at 10 seconds meaning that a total of 360 time increments were generated. The results of the data generated can be seen in Figures 2.1 and Figure 2.2.

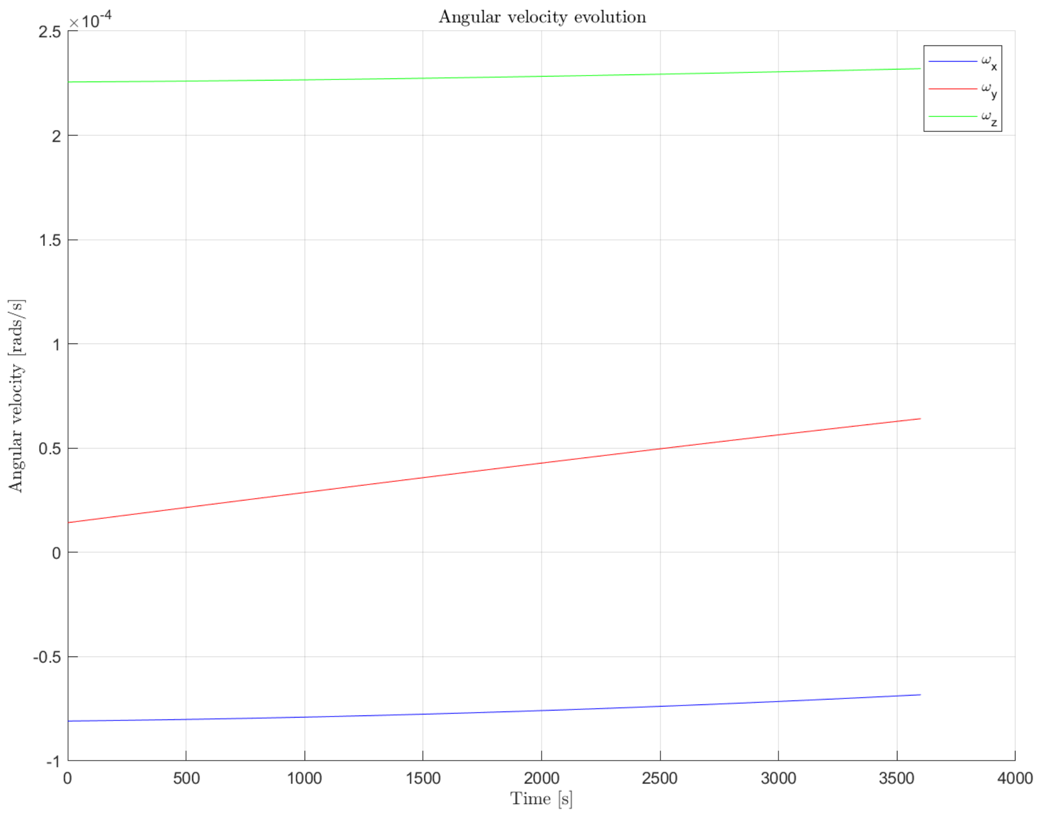

Figure 2.1: Angular velocity evolution.

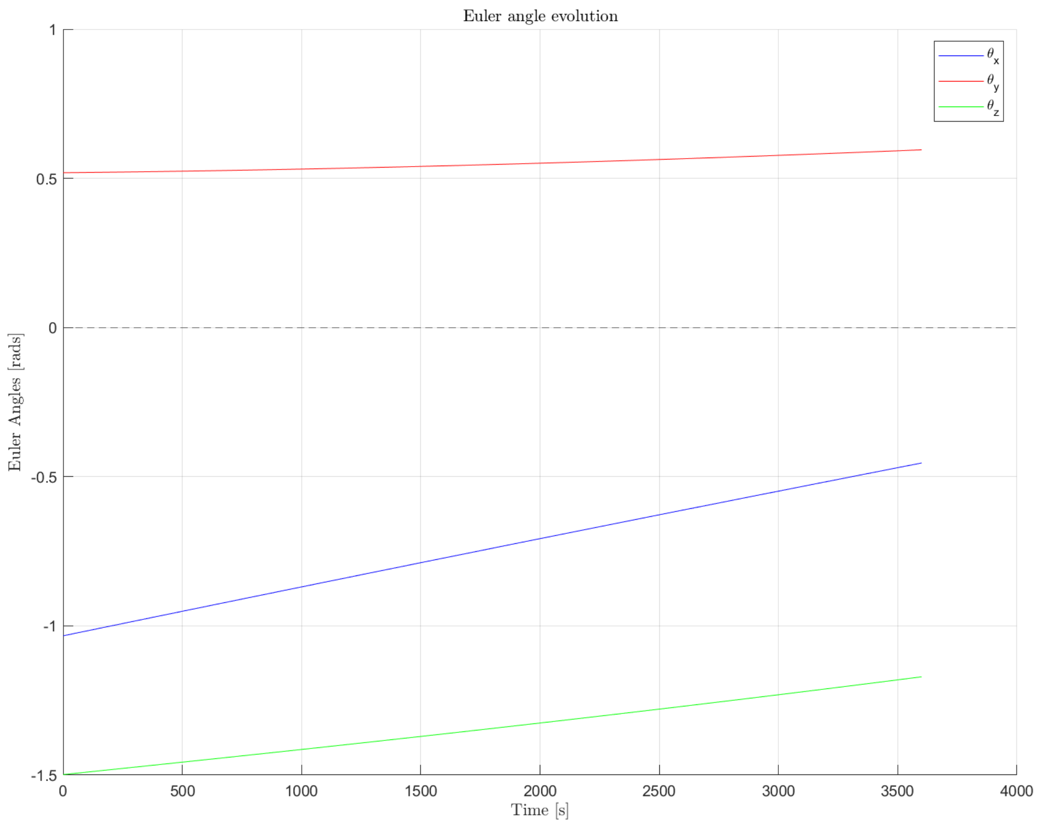

Figure 2.2: Euler angle evolution. θ_2 is not close to approaching the 0 radians line but it is increasing. At some point, if it comes close to π, the representation may break.

To check upon the angular momentum, **H** of the system we should first discuss what it should be in theory. Changes in **H** would be caused by a force, or more specifically torque, being applied to the system as seen in Equation 2.23.

$$\dot{\mathbf{H}} = \mathbf{L} \tag{2.23}$$

Due to this simulation operating in a torque-free environment, $\mathbf{L} = \mathbf{0}$, therefore $\dot{\mathbf{H}} = 0$. This translates to a constant $\mathbf{H}$ through the evolution of the attitude of the system. Knowing this, it can also be shown that the $\mathbf{H}$ can be calculated using Equation 2.24.

$$\mathbf{H} = \begin{bmatrix} H_x \\ H_y \\ H_z \end{bmatrix} = \begin{bmatrix} I_{xx}\omega_x \\ I_{yy}\omega_y \\ I_{zz}\omega_z \end{bmatrix} \tag{2.24}$$

Looking at the kinetic energy T of the system we can also make some mathematical extrapolations. The equation that governs T is seen in Equation 2.25.

$$T = \frac{1}{2}\omega^{\mathbf{T}}\mathbf{H} \tag{2.25}$$

We can take the first derivative of Equation 2.25 and apply the chain rule as seen in Equation 2.26

$$\dot{T} = \frac{1}{2}\mathbf{H}^{\mathbf{T}}\dot{\omega} + \frac{\mathbf{1}}{\mathbf{2}}\omega^{\mathbf{T}}\dot{\mathbf{H}} \tag{2.26}$$

The last term in the equation above goes to 0 as $\dot{\mathbf{H}} = 0$. Additionally, the Euler kinematic differential equations demonstrate that $\dot{\omega} = [I]^{-1}(0 - [\tilde{\omega}][I]\omega)$. Using the mentioned information a more final form of Equation 2.26 can be seen in Equation 2.27.

$$\dot{T} = \frac{1}{2}\omega(-\omega \times \mathbf{H}) \tag{2.27}$$

The cross-product between these two orthogonal vectors would give a 0. This means that rotational kinetic energy is conserved. To calculate its value we can use Equation 2.28

$$T = \frac{1}{2}(H_x\omega_x + H_y\omega_y + H_z\omega_z) \tag{2.28}$$

Using the data extracted from the `AttitudeDynamics.m` (Appendix B.7) function `ode45` loop we can calculate both conserved quantities, $\mathbf{H}$ and T. Plotting the results gives us Figure 2.3.

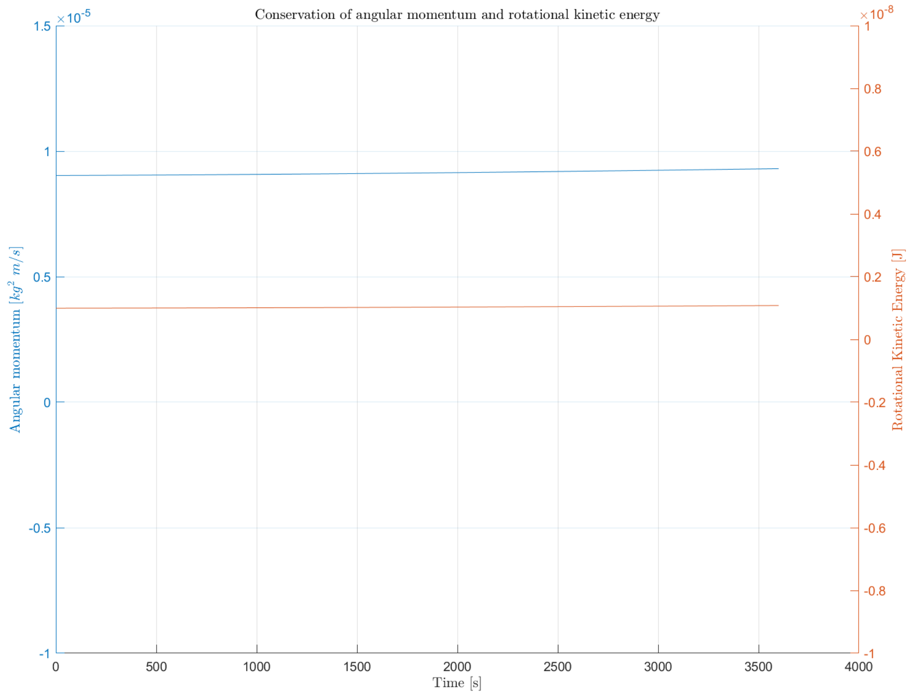

Figure 2.3: Angular momentum and rotational kinetic energy conservation. This is an important result as these quantities are theoretically conserved. As seen in the figure, this is the case here, demonstrating an agreement with numerical testing.

As discussed in the theoretical part above, these quantities inside a torque-free system are conserved and should stay constant. The figure demonstrates exactly that and serves as a validation of proper mathematical and coding procedures.

References

[1] N. Baresi, *Eee3039 space dynamics and missions*, Sep. 2024.

[2] R. R. Bate, D. D. Mueller, J. E. White, and W. W. Saylor, *Fundamentals of Astrodynamics.* Courier Dover Publications, 2020.

[3] H. D. Curtis, *Orbital Mechanics for Engineering Students: Revised Reprint.* Butterworth-Heinemann, 2020.

[4] W. Campbell, *Earth-sized sphere with topography*, `https://www.mathworks.com/matlabcentral/fileexchange/27123-earth-sized-sphere-with-topography`, 2016.

[5] E. Cheever, *Fourth order Runge-Kutta*, `https://lpsa.swarthmore.edu/NumInt/NumIntFourth.html`, Accessed: 2024-11-14.

[6] H. Schaub and J. L. Junkins, *Analytical Mechanics of Space Systems.* AIAA, 2003.

Appendices

Appendix A

MATLAB Scripts

A.1 cw1.m

```
clc;
close all;
clearvars;
clear global;
set(0, 'DefaultTextInterpreter', 'latex');

% This script covers Part 1 of the coursework (Week 2-5)
global kepler_iter

%%%%%%%%%%%%%%% CW data %%%%%%%%%%%%%%%%%
mu = 398600.4418; % [km^3/s^2]

R_e = 6378.137; % [km]
w_e = 7.2921e-5; % [rad/s]


%%%%%%%%%%%%%%%%%%%%%% Part 1 data %%%%%%%%%%%%%%%%%%%%
a = 19052.49; % [km]
e = 0.6516; % eccentricity

i = 10.02; % deg
i = deg2rad(i); % rad

omega = 250.77; % deg
omega = deg2rad(omega); % rad

w = 310.67; % deg
w = deg2rad(w); % rad

M_0 = 8.77; % deg
M_0=deg2rad(M_0); % rad

tol_Kepler = 10e-10;

%%%%%%%%%%%%%%%%%% Week 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%
[E_final, theta_final] = Kepler(e,M_0,tol_Kepler);

% Inbuilt MATLAB fzero function testing for Kepler equation
% [E_fzero, theta_fzero] = Kepler_fzero(e,M_0,tol_Kepler);

E_final = mod(E_final, 2*pi);
% print_kepler = ['Kepler iterations:', kepler_iter];


% Old test values
disp('Kepler E and theta values (in deg):');
disp(rad2deg(E_final));
disp(rad2deg(theta_final)); %49.5deg => periapsis
disp('Kepler iterations needed:');
disp(kepler_iter);
%
% disp('Fzero Kepler');
% disp(rad2deg(E_fzero));
% disp(rad2deg(theta_fzero));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Week 3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% coe = [a, e, i, omega, w, theta_final]';

period = 2*(pi/sqrt(mu))*a^(3/2);

orbits = period*10; % can easily expand the number of rotations around Earth

t_increments = 1000;

% uses orbits to make it easier to increase time
t_period = linspace(0,orbits,t_increments);

n = sqrt(mu/a^3);

M_evolve = M_0+n*t_period;

E_evolve = zeros(1,t_increments);
theta_evolve = zeros(1,t_increments);
kepler_iter_store = zeros(1,t_increments);
%using the M_evolve values in Kepler equations to get E and theta for all t
%points
for k=1:length(t_period)
[E_evolve(k),theta_evolve(k)] = Kepler(e,M_evolve(k),tol_Kepler);
kepler_iter_store(k) = kepler_iter;
end

disp("Average iterations need for Kepler fzero:");
disp(mean(kepler_iter_store));

%%%%%%%% Modulus %%%%%%%%
%Making the data easier to read in a plot
E_evolve = mod(E_evolve,2*pi);
theta_evolve = mod(theta_evolve,2*pi);
M_evolve = mod(M_evolve,2*pi);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Plotting %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%Using tiledlayout to save on space and windows opening, easier to read on
%separate graphs
tiles = tiledlayout(1,3);
tiles.TileSpacing = 'compact';
nexttile;
plot(t_period, E_evolve, "b", "LineWidth",2);
xlabel(tiles,"Time (s)");
ylabel("Eccentric anomaly (rad)");
title("Eccentric anomaly through the orbit");
grid on;
hold off;
nexttile;


plot(t_period, theta_evolve, "b", "LineWidth",2);
ylabel("True anomaly (rad)");
title("True anomaly through the orbit");
grid on;
hold off;
nexttile;

plot(t_period, M_evolve, "b", "LineWidth",2);
ylabel("Mean anomaly (rad)");
title("Mean anomaly through the orbit");
grid on;
hold off;


%%%%%%%%%%%%%%%%%%%%%%%%%%%% COE to RV %%%%%%%%%%%%%%%%%%%%%%%%%%%

% Storage for r and v

r_evolve = zeros(3,length(period));
v_evolve = zeros(3,length(period));

for j=1:length(t_period)

coe = [a, e, i, omega, w, theta_evolve(j)]';
[r_evolve(:,j),v_evolve(:,j)] = coe2rv(coe,mu);

end

X = [r_evolve; %state vector creation; 6x1000 size
v_evolve];
%%%%%%%%%%%%%%%%%%%% Plotting ECI coord %%%%%%%%%%%%%%%%%%%%%%%
disp('ECI first point in orbit state vector:');
disp(X(:,1));
disp('ECI last point in orbit state vector:');
disp(X(:,end));

figure % new figure, stops tiledlayout

% r vector coords plotting
```

```
plot3(X(1,:),X(2,:),X(3,:), 'DisplayName', 'ECI Orbit');
hold on;
grid on;
% first and final r vector points
plot3(X(1,1),X(2,1),X(3,1),'ok','MarkerFaceColor','y', 'DisplayName', ...
'Start Point');
plot3(X(1,end),X(2,end),X(3,end),'pentagram','MarkerFaceColor','r', ...
'DisplayName', 'End Point');
title('ECI 3D orbit coordinates visualisation');
L1 = legend;
L1.AutoUpdate = 'off';
make_earth;

% Orbit unit vectors
r_unit = X(1:3, 1);
v_unit = X(4:6, 1);

h_unit = cross(r_unit,v_unit);
e_unit = cross(v_unit,h_unit)/mu - r_unit/norm(r_unit);

% e,h,p unit vector triad calc
ie = e_unit/norm(e_unit);
ih = h_unit/norm(h_unit);
ip = cross(ih, ie)/norm(cross(ih, ie));

%plotting unit vector arrows for orbital plane
quiver3(0,0,0,ie(1),ie(2),ie(3),1e4, 'k');
text(ie(1)*1e4,ie(2)*1e4,ie(3)*1e4,'$\hat{e}$');
quiver3(0,0,0,ih(1),ih(2),ih(3),1e4, 'k');
text(ih(1)*1e4,ih(2)*1e4,ih(3)*1e4,'$\hat{h}$');
quiver3(0,0,0,ip(1),ip(2),ip(3),1e4, 'k');
text(ip(1)*1e4,ip(2)*1e4,ip(3)*1e4,'$\hat{p}$');


%%%%%%%%%%%%%%%%%%%%% WEEK 4 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ode
options = odeset(RelTol = 1e-10);
[t,X_ode] = ode45(@(t,X_ode) TBP_ECI(t,X_ode,mu) ,t_period,[X(:,1)],options);
X_ode = X_ode'; %from 1000x6 to 6x1000 to match original X vector

figure % this figure is to plot the orbits and compare to the coe one
% r vector coords plotting
plot3(X_ode(1,:),X_ode(2,:),X_ode(3,:), 'DisplayName', ...
'Integrated Orbit Coordinates', 'LineWidth', 2); %integrated r values
hold on;
grid on;
plot3(X(1,:),X(2,:),X(3,:), '--', 'DisplayName', 'Original OE orbit', ...
'LineWidth', 2); %original coe values
title('Comparison between the ode45 and coe orbits');
%Code below generates a legend and stops it from adding the axis vectors
%and earth mesh to the legend
L1 = legend;
```

```
L1.AutoUpdate = 'off';

% first and final r vector points (uncomment to add)
% plot3(X(1,1),X(2,1),X(3,1),'ok','MarkerFaceColor','b');
% plot3(X(1,end),X(2,end),X(3,end),'ok','MarkerFaceColor','r');

%Generate an Earth with ECI axis unit vectors
make_earth;


%%%% Error propagation between integrated and coe r and v
r_error = X(1:3,:) - X_ode(1:3,:);
r_error_store = zeros(1,t_increments);

v_error = X(4:6,:) - X_ode(4:6,:);
v_error_store = zeros(1,t_increments);

for k=1:length(t_period)
r_error_store(k) = norm(r_error(1:3,k));
v_error_store(k) = norm(v_error(1:3,k));
end

%%%%%%%%%%%%%%%%% Plot for error comparison between ode45 and coe %%%%%%%%
figure();
yyaxis left;
plot(t_period, r_error_store);
xlabel('Time [s]');
ylabel('Radius error [m]');
title(['Error evolution between integrated and orbital elements extracted' ...
'values for radius and velocity'])

yyaxis right;
plot(t_period, v_error_store);
ylabel('Velocity error [m/s]');

% adding more orbits makes the orbits to diverge more and more (orbit
% determination problem, needs batch or sequential processors to fix)

h_ode_norm = zeros(1,t_increments);

for k=1:length(t_period)
h_ode = cross(X_ode(1:3,k),X_ode(4:6,k));
h_ode_norm(k) = norm(h_ode);

end

figure()
static_h = sqrt(mu*a*(1-e^2));
plot(t_period,h_ode_norm,'r','LineWidth', 2, 'DisplayName', 'Norm Value');
yline(static_h, 'b--','LineWidth', 2, 'DisplayName', 'Static Value');
xlabel('Time [s]');
ylabel('Angular momentum [$kg~m^2/s$]')
ylim([6.5e4,6.7e4]);
```

```
xlim([0,t(end)]);
legend();
title(['Angular momentum comparison between static theory value and ' ...
'integrated one from orbital data'])

%%%%%%%%%%%%%%%%%%%%% WEEK 5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
options = odeset(RelTol = 1e-10);
[t,X_ode_j] = ode45(@(t,X_ode_j) TBP_ECEF(t,X_ode_j,mu), ...
t_period,[X(:,1)],options);
X_ode_j = X_ode_j'; %from 1000x6 to 6x1000 to match original X vector

figure % this figure is to plot the orbits and compare to the coe one

%%%% r vector coords plotting %%%%%%%%%

%integrated ECI values
plot3(X_ode(1,:),X_ode(2,:),X_ode(3,:),'DisplayName', 'ECI','LineWidth', 2);
hold on;
grid on;
%original coe values
plot3(X(1,:),X(2,:),X(3,:), 'r--','DisplayName', 'COE','LineWidth', 2);
%integrated ECEF values
plot3(X_ode_j(1,:),X_ode_j(2,:),X_ode_j(3,:), 'g--','DisplayName', ...
'ECEF','LineWidth', 2);
title('Comparison between the three orbital representations')

% first and final r vector points
plot3(X_ode_j(1,1),X_ode_j(2,1),X_ode_j(3,1),'ok','MarkerFaceColor','y', ...
'DisplayName','ECEF Start Point');
plot3(X_ode_j(1,end),X_ode_j(2,end),X_ode_j(3,end),'pentagram', ...
'MarkerFaceColor','r','DisplayName','ECEF End Point ');

%skips make_earth elements from being added to the legend
L1 = legend;
L1.AutoUpdate = 'off';
%Generate an Earth with ECI axis unit vectors
make_earth;

%Makes a topdown (2D) view of the figure when generated, comment in to use
% view(2);
disp('ECEF first point in orbit state vector:');
disp(X_ode_j(:,1));
disp('ECEF last point in orbit state vector:');
disp(X_ode_j(:,end));
```

A.2 cw2.m

```
clc;
close all;
clearvars;
clear global;
set(0, 'DefaultTextInterpreter', 'latex');
```

```
% This script covers Part 2 of the Coursework (Week 6-8)
I_xx = 0.07583; % [kg m^2]
I_yy = 0.05833; % [kg m^2]
I_zz = 0.02916; % [kg m^2]

inert_mat = [I_xx  0     0;
0   I_yy   0;
0    0   I_zz];

r_0 = [5300.64;
17575.73;
-138.50]; % [km]

v_0 = [-4.2880;
-1.9373;
-0.6026]; % [km]


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% WEEK 6 %%%%%%%%%%%%%%%%%%%%%%%%%%%
h_0 = cross(r_0,v_0);

T = v_0/norm(v_0);
W = h_0/norm(h_0);
N = cross(T,W);
NTW_DCM = [N, T, W]; %[IT]
NTW_DCM_inv = inv(NTW_DCM);
NTW_DCM_trans = NTW_DCM';

disp('NTW_DCM');
disp(NTW_DCM);
disp('NTW_DCM intervse');
disp(NTW_DCM_inv);
disp('NTW_DCM transpose');
disp(NTW_DCM_trans);

BT = [0.7146, 0.6131, -0.3368;
-0.6337, 0.7713, 0.0594;
0.2962, 0.1710, 0.9397];

BI = BT*NTW_DCM'; %[BT]*[TI]=[BI]
disp('BI matrix:');
disp(BI);


%%%%%%%%%%%%%%%%%%%%%%%%% WEEK 7 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Princple angle
phi = acos((trace(BI)-1)/2);
% Principle axes
e_1 = (BI(2,3)-BI(3,2))/(2*sin(phi));
e_2 = (BI(3,1)-BI(1,3))/(2*sin(phi));
e_3 = (BI(1,2)-BI(2,1))/(2*sin(phi));

e_test = sqrt(e_1^2+e_2^2+e_3^2); % = 1 (good result)
```

```
% Quaternion conversion

q_0 = 0.5*sqrt(trace(BI)+1);
q_1 = (BI(2,3)-BI(3,2))/(4*q_0);
q_2 = (BI(3,1)-BI(1,3))/(4*q_0);
q_3 = (BI(1,2)-BI(2,1))/(4*q_0);

quat_test = sqrt(q_0^2 + q_1^2 + q_2^2 + q_3^2); % = 1 (good result)


% Getting the Euler angles

theta_1 = atan2(BI(3,1),BI(3,2));
theta_2 = acos(BI(3,3));
theta_3 = atan2(BI(1,3),BI(2,3));


%%%%%%%%%%%%%%%%%%%%%%%%%%% WEEK 8 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

omega_BI = [-8.0862e-5;1.4258e-5;2.2559e-4];

X_omega = [theta_1, theta_2, theta_3, omega_BI(1), omega_BI(2), omega_BI(3)];

t_period = linspace(0,3600,360);
options = odeset(RelTol = 1e-10);
[t_period,X_ode_omega] = ode45(@(t_period,X_ode_omega) ...
AtitittudeDynamics(t_period,X_ode_omega, inert_mat) , ...
t_period,X_omega,options);

figure()
hold on;
grid on;
plot(t_period, X_ode_omega(:,1), 'b');
plot(t_period, X_ode_omega(:,2), 'r');
plot(t_period, X_ode_omega(:,3), 'g');
yline(0, '--');
xlabel('Time [s]');
ylabel('Euler Angles [rads]');
title('Euler angle evolution');
legend('\theta_x', '\theta_y', '\theta_z');

figure()
hold on;
grid on;
plot(t_period, X_ode_omega(:,4), 'b');
plot(t_period, X_ode_omega(:,5), 'r');
plot(t_period, X_ode_omega(:,6), 'g');
xlabel('Time [s]');
ylabel('Angular velocity [rads/s]');
title('Angular velocity evolution');
legend('\omega_x', '\omega_y', '\omega_z');
```

```
% Angular momentum and rotational kinetic energy
H_store = zeros(1,length(t_period));
T_store = zeros(1,length(t_period));

for jj=1:length(t_period)
H_x = I_xx*X_ode_omega(jj,4);
H_y = I_yy*X_ode_omega(jj,5);
H_z = I_zz*X_ode_omega(jj,6);
H_store(jj) = norm([H_x;H_y;H_z]);

T_1 = H_x*X_ode_omega(jj,4);
T_2 = H_y*X_ode_omega(jj,5);
T_3 = H_z*X_ode_omega(jj,6);
T_store(jj) = 0.5*(T_1+T_2+T_3);
end

figure();
hold on;
grid on;
yyaxis left;
plot(t_period, H_store);
xlabel('Time [s]');
ylabel('Angular momentum [$kg^2~m/s$]');
title('Conservation of angular momentum and rotational kinetic energy')
ylim([-1e-5,1.5e-5]);
yyaxis right;
plot(t_period, T_store);
ylabel('Rotational Kinetic Energy [J]');
ylim([-1e-8,1e-8]);
```

A.3 make_earth.m

```
% This is a simple script that is used throughout the coursework main
% scripts to generate a realistic looking Earth mesh. It tries to generate
% the custom mesh from a file obtained from the mathworks website. If the
% file is not present it will make a simple gray mesh, in case the user
% does not want to download any additional materials. This script was done
% mainly for convinience and save on space in the main files.

% Earth Mesh taken from  Will Campbell (2024). Earth-sized Sphere with Topography
% (https://www.mathworks.com/matlabcentral/fileexchange/27123-earth-
% sized-sphere-with-topography)

%Try/catch statement in place in case user does not have the custom Earth
%mesh. Instead a gray Earth-sized sphere is generated.
try
earth_sphere('km')
catch
disp(['Error when trying to create textured Earth mesh. Using generic' ...
' 3D mesh to represent the Earth instead. earth_sphere potentially ' ...
'missing. Please download the file from the link in the make_earth.m ' ...
'subscript.'])
earth_radius_km = 6371; % [km]
```

```
[X_mesh, Y_mesh, Z_mesh] = sphere(50); % making mesh
X_mesh = X_mesh * earth_radius_km; % scaling all axis by Earth radius
Y_mesh = Y_mesh * earth_radius_km;
Z_mesh = Z_mesh * earth_radius_km;
earth_mesh = mesh(X_mesh,Y_mesh,Z_mesh);
daspect([1 1 1]);
earth_mesh.FaceColor = [0.5, 0.5, 0.5]; % Grey color for the faces
earth_mesh.EdgeColor = 'none'; % remove the mesh lines
end

% ECI unit vectors arrows
quiver3(0,0,0,1,0,0,1e4, 'r');
text(1e4,0,0,'i');
quiver3(0,0,0,0,1,0,1e4, 'g');
text(0,1e4,0,'j');
quiver3(0,0,0,0,0,1,1e4, 'b');
text(0,0,1e4,'k');
```

A.4 geo_test.m

```
clc;
close all;
clearvars;
clear global;
set(0, 'DefaultTextInterpreter', 'latex');


% This is a sanity check script that is a copy of cw1.m. The only changes
% here are done to the initial orbital conditions to mimic those of a
% geostationary satellite. The theory is that if the calculation of the ECI
% to ECEF frame is done correctly, the plot of the ECEF trajectory would
% appear as "stationary points" in the orbital plane, close to the initial
% starting point. This is done for verification of Week 5 task as ECEF
% trajectories can be rather unintuitive. The formatting is not as well
% presented as in cw1.m as this uses an older version of the script.


global kepler_iter

%%%%%%%%%%%%%%% CW data %%%%%%%%%%%%%%%%%
mu = 398600.4418; % [km^3/s^2]

R_e = 6378.137; % [km]
w_e = 7.2921e-5; % [rad/s]
I_xx = 0.07583; % [kg m^2]
I_yy = 0.05833; % [kg m^2]
I_zz = 0.02916; % [kg m^2]

inert_mat = [I_xx  0     0;
0    I_yy   0;
0     0   I_zz];

%%%%%%%%%%%%%%%%%%%%%%% GEOSTATIONARY DATA %%%%%%%%%%%%%%%%%%%%
```

```
a = 42164; % [km]
e = 0.0; % eccentricity

i = 0; % deg
i = deg2rad(i); % rad

omega = 0; % deg
omega = deg2rad(omega); % rad

w = 0; % deg
w = deg2rad(w); % rad

M_0 = 0; % deg
M_0=deg2rad(M_0); % rad

tol_Kepler = 10e-10;

%%%%%%%%%%%%%%%%% Week 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%
[E_final, theta_final] = Kepler(e,M_0,tol_Kepler);

% Inbuilt MATLAB fzero function testing for Kepler equation
% [E_fzero, theta_fzero] = Kepler_fzero(e,M_0,tol_Kepler);

E_final = mod(E_final, 2*pi);
% print_kepler = ['Kepler iterations:', kepler_iter];


% Old test values
% disp('Custom Kepler');
% disp(rad2deg(E_final));
% disp(rad2deg(theta_final)); %49.5deg => periapsis
%
% disp('Fzero Kepler');
% disp(rad2deg(E_fzero));
% disp(rad2deg(theta_fzero));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Week 2 End %%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Week 3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% coe = [a, e, i, omega, w, theta_final]';

period = 2*(pi/sqrt(mu))*a^(3/2);

orbits = period*1; % can easily expand the number of rotations around Earth

t_increments = 1000;
t_period = linspace(0,orbits,t_increments); % uses orbits to make it easier
%                                              to increase time

n = sqrt(mu/a^3);

M_evolve = M_0+n*t_period;
```

```
E_evolve = zeros(1,t_increments);
theta_evolve = zeros(1,t_increments);
kepler_iter_store = zeros(1,t_increments);
%using the M_evolve values in Kepler equations to get E and theta for all t
%points
for k=1:length(t_period)
[E_evolve(k),theta_evolve(k)] = Kepler(e,M_evolve(k),tol_Kepler);
kepler_iter_store(k) = kepler_iter;
end

disp("Average iterations need for Kepler fzero:");
disp(mean(kepler_iter_store));

%%%%%%%% Modulus %%%%%%%%
%Making the data easier to read in a plot
E_evolve = mod(E_evolve,2*pi);
theta_evolve = mod(theta_evolve,2*pi);
M_evolve = mod(M_evolve,2*pi);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Plotting %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Using tiledlayout to save on space and windows opening, easier to read on
%separate graphs
tiles = tiledlayout(1,3);
tiles.TileSpacing = 'compact';
nexttile;
plot(t_period, E_evolve, "b", "LineWidth",2);
xlabel(tiles,"Time (s)");
ylabel("Eccentric anomaly (rad)");
title("Eccentric anomaly through the orbit");
grid on;
hold off;
nexttile;


plot(t_period, theta_evolve, "b", "LineWidth",2);
ylabel("True anomaly (rad)");
title("True anomaly through the orbit");
grid on;
hold off;
nexttile;

plot(t_period, M_evolve, "b", "LineWidth",2);
ylabel("Mean anomaly (rad)");
title("Mean anomaly through the orbit");
grid on;
hold off;


%%%%%%%%%%%%%%%%%%%%%%%%%%%% COE to RV %%%%%%%%%%%%%%%%%%%%%%%%%%%

% Storage for r and v

r_evolve = zeros(3,length(period));
```

```
v_evolve = zeros(3,length(period));

for j=1:length(t_period)

coe = [a, e, i, omega, w, theta_evolve(j)]';
[r_evolve(:,j),v_evolve(:,j)] = coe2rv(coe,mu);

end

X = [r_evolve; %state vector creation; 6x1000 size
v_evolve];
%%%%%%%%%%%%%%%%%%%% Plotting ECI coord %%%%%%%%%%%%%%%%%%%%%%%

figure % new figure, stops tiledlayout

% r vector coords plotting
plot3(X(1,:),X(2,:),X(3,:));
hold on;
% first and final r vector points
plot3(X(1,1),X(2,1),X(3,1),'ok','MarkerFaceColor','b');
plot3(X(1,end),X(2,end),X(3,end),'ok','MarkerFaceColor','r');
make_earth;

% Orbit unit vectors
r_unit = X(1:3, 1);
v_unit = X(4:6, 1);

h_unit = cross(r_unit,v_unit);
e_unit = cross(v_unit,h_unit)/mu - r_unit/norm(r_unit);

% e,h,p unit vector triad calc
ie = e_unit/norm(e_unit);
ih = h_unit/norm(h_unit);
ip = cross(ih, ie)/norm(cross(ih, ie));

%plotting unit vector arrows for orbital plane
quiver3(0,0,0,ie(1),ie(2),ie(3),1e4, 'k');
text(ie(1)*1e4,ie(2)*1e4,ie(3)*1e4,'$\hat{e}$');
quiver3(0,0,0,ih(1),ih(2),ih(3),1e4, 'k');
text(ih(1)*1e4,ih(2)*1e4,ih(3)*1e4,'$\hat{h}$');
quiver3(0,0,0,ip(1),ip(2),ip(3),1e4, 'k');
text(ip(1)*1e4,ip(2)*1e4,ip(3)*1e4,'$\hat{p}$');


%%%%%%%%%%%%%%%%%%%%% WEEK 4 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ode
options = odeset(RelTol = 1e-10);
[t,X_ode] = ode45(@(t,X_ode) TBP_ECI(t,X_ode,mu) ,t_period,[X(:,1)], ...
options);
X_ode = X_ode'; %from 1000x6 to 6x1000 to match original X vector

figure % this figure is to plot the orbits and compare to the coe one
```

```
% r vector coords plotting
plot3(X_ode(1,:),X_ode(2,:),X_ode(3,:)); %integrated r values
hold on;
plot3(X(1,:),X(2,:),X(3,:), '--'); %original coe values

% first and final r vector points (uncomment to add)
% plot3(X(1,1),X(2,1),X(3,1),'ok','MarkerFaceColor','b');
% plot3(X(1,end),X(2,end),X(3,end),'ok','MarkerFaceColor','r');

make_earth;


%%%% Error propagation between integrated and coe r and v
r_error = X(1:3,:) - X_ode(1:3,:);
r_error_store = zeros(1,t_increments);

v_error = X(4:6,:) - X_ode(4:6,:);
v_error_store = zeros(1,t_increments);

for k=1:length(t_period)
r_error_store(k) = norm(r_error(1:3,k));
v_error_store(k) = norm(v_error(1:3,k));
end

%%%%%%%%%%%%%%%%% Plot for error comparison between ode45 and coe %%%%%%%%
figure();
yyaxis left;
plot(t_period, r_error_store);
xlabel('Time [s]');
ylabel('Radius error [m]');
title(['Error evolution between integrated and orbital elements extracted' ...
'values for radius and velocity'])

yyaxis right;
plot(t_period, v_error_store);
ylabel('Velocity error [m/s]');

% adding more orbits makes the orbits to diverge more and more (orbit
% determination problem, needs batch or sequential processors to fix)

h_ode_norm = zeros(1,t_increments);

for k=1:length(t_period)
h_ode = cross(X_ode(1:3,k),X_ode(4:6,k));
h_ode_norm(k) = norm(h_ode);

end

figure()
plot(t_period, h_ode_norm);
static_h = sqrt(mu*a*(1-e^2));
yline(mean(h_ode_norm),'r','LineWidth', 2);
yline(static_h, 'b--','LineWidth', 2);
```

```
ylim([6.5e4,6.7e4]);
xlim([0,t(end)]);

%%%%%%%%%%%%%%%%%%%% SANITY CHECK %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
options = odeset(RelTol = 1e-10);
[t,X_ode_j] = ode45(@(t,X_ode_j) TBP_ECEF(t,X_ode_j,mu) ,t_period, ...
[X(:,1)],options);
X_ode_j = X_ode_j'; %from 1000x6 to 6x1000 to match original X vector

figure % this figure is to plot the orbits and compare to the coe one

% r vector coords plotting
plot3(X_ode(1,:),X_ode(2,:),X_ode(3,:),'DisplayName', ...
'ECI ode45 points'); %integrated r values
hold on;
grid on;
plot3(X(1,:),X(2,:),X(3,:), 'r--', 'DisplayName', ...
'COE points'); %original coe values
plot3(X_ode_j(1,:),X_ode_j(2,:),X_ode_j(3,:), 'g-o', 'DisplayName', ...
'ECEF points'); %original coe values
L1 = legend;
L1.AutoUpdate = 'off';
% first and final r vector points (uncomment to add)
% plot3(X(1,1),X(2,1),X(3,1),'ok','MarkerFaceColor','b');
% plot3(X(1,end),X(2,end),X(3,end),'ok','MarkerFaceColor','r');


make_earth;
view(2)
```

Appendix B

MATLAB Functions

B.1 Kepler.m

```
function [E_final,theta_final] = Kepler(e, M, tol)
% Uses Newton-Ralphson method to calculate the eccentric and
% true anomaly based on supplied eccentricity and initial mean anomaly.
% Inputs: e - Eccentricity
%         M - Initial Mean Anomaly
%         tol - tolerance for the Newton-Ralphson method
% Outputs: E_final - Eccentric anomaly
%          theta_final - Mean anomaly


% Just outputting the iter value out the function is the better solution.
% The assignment however does not include kepler_iter as an output so
% obvious solution is to export it as a global
global kepler_iter


% first guess for the eccentric anomaly is the mean anomaly's value
E_guess = M;

% first iteration of the function with all variables on one side of the
% equals sign
func_loop = E_guess-e*sin(E_guess)-M;
kepler_iter = 0;
% using absolute value for the function as it can go into negatives
while abs(func_loop)>tol
func_loop = E_guess-e*sin(E_guess)-M;
func_p_loop = 1-e*cos(E_guess); %first derivative of func with respect to E
delta = - (func_loop/func_p_loop);
E_guess = E_guess + delta;
kepler_iter = kepler_iter + 1;
end

% Outputs calculation
E_final = E_guess;
theta_final = 2*atan2(sqrt(1+e)*tan(E_final/2), sqrt(1-e));
```

B.2 Kepler_fzero.m

```
function [E_final,theta_final] = Kepler_fzero(e, M, tol)
% This is a validation function used to test the custom Kepler.m. It uses
% the inbuilt fzero function of Matlab to do so

E_guess = M;
func = @(E) E-e*sin(E)-M;
options = optimset('TolX', tol);
E_final = fzero(func, E_guess, options);
E_final_1 = mod(E_final,2*pi);

theta_final = 2*atan(sqrt((1+e)/(1-e))*tan(E_final/2));
theta_final_1 = 2*atan(sqrt((1+e)/(1-e))*tan(E_final_1/2));
```

B.3 coe2rv.m

```
function [r,v] = coe2rv(coe, mu)
% Converts orbital elements to cartesian coordinates
% for position and velocity
% Accepts an array for orbital elements as seen below
%
% Inputs: coe = [a, e, i, omega, w, theta]'
%         mu  = gravitational parameter of the Earth
% Outputs: r - position vector
%          v - velocity vector

% separate elements into variables for ease of use and readibility
a = coe(1);
e = coe(2);
i = coe(3);
omega = coe(4);
w = coe(5);
theta = coe(6);

% r vector defininition and specific angular momentum
r_mag = (a*(1-e^2))/(1+e*cos(theta));
h = sqrt(mu*a*(1-e^2));
r = [r_mag*cos(theta),r_mag*sin(theta),0]';

% Rotation matrix creation
% Note: Line 22 uses a separate function where rotations around the X,Y,Z
% axes is done via a switch case
coe_rv_mat = (rot_mat(w,3)*rot_mat(i,1)*rot_mat(omega,3))';

r = coe_rv_mat*r; %New reference frame R after rotation

v = coe_rv_mat*[-mu/h*sin(theta), mu/h*(cos(theta)+e), 0]'; %New v vector
```

B.4 rot_mat.m

```
function [rot_mat]=rot_mat(theta,rot_mode)
% DCM for sequential rotiations selection function. It accepts the angle
% and the axis around which it happens. The notation is kept to the
% standard 1-2-3. In other words for a rotation of 3-1-3 the function would
% be called three times with the corresponding angles and the numbers
% 3,1,3.
% Input: theta - angle by which the vector/matrix will be rotated
%        rot_mode - around which axis will the rotation be done. It uses
%                   the numerical representation 1,2,3 rather than using
%                   x,y,z.
% Output: rot_mat- rotation matrix for selected axis and angle

% simple switch case for ease of use in larger scripts
switch rot_mode
case 1
%around the x-axis
rot_mat = [1     0           0
0 cos(theta) sin(theta)
0 -sin(theta) cos(theta)];
case 2
%around the y-axis
rot_mat = [cos(theta)  0  -sin(theta)
0           1       0
sin(theta)  0  cos(theta)];
case 3
%around the z-axis
rot_mat = [cos(theta)     sin(theta)  0
-sin(theta)    cos(theta)  0
0                   0       1];
end
```

B.5 TBP_ECI.m

```
function [dXdt] = TBP_ECI(t, X, mu)
% This function is made with the intent of being supplied to ode45.
% The inputs and the calculations below represent the equations of motion
% of the satellite.
% Inputs: t - time period for the differential solver
%         X - state vector of the satellite. Rows 1:3 are position; Rows
%         4:6 are velocity
%         mu - gravitational parameter of the Earth
% Outputs: dXdt - this is the first order derivative of the state vector.
%                 When passed into the ode, the output of the
%                 differentiation will be position and velocity data


r = X(1:3,:);
v = X(4:6,:);
r_norm = norm(r);
% acceleration calculation
v_dot = -mu/r_norm^3 * r;

dXdt = [v;v_dot];
```

B.6 TBP_ECEF.m

```
function [dXdt] = TBP_ECEF(t,X,mu)
% ode45 function that accepts ECI coordinates and velocity data and gives
% back the ECEF equivalent. Uses the transport theorem to go from one
% reference frame to the next.
% Inputs: t - time period of integration [s] Example: [0, 1000]
%         X - initial state vector guess (1x6 shape): Example: [r_i; v_i]
%         mu - gravitational parameter of the planet
% Output: dXdt - the derivates of the position and velocity in the ECEF
%                frame. The configuration should be the same as the intial
%                guess state vector [vel_j; accel_j]

%ECI frame data
r_i = X(1:3,:);
v_i = X(4:6,:);

r_norm_i = norm(r_i);
a_i = -mu/r_norm_i^3 * r_i;

earth_rot = (2*pi)/(24*60*60); %rads/s
omega_j = [0;0;earth_rot];

%%%
% rotate the r vector to ECEF by doing a euler rotation around the 3/z axis
r_j = eye(3)*r_i;

%%% velocity
v_j  = v_i - cross(omega_j,r_j);
% a_i_j = a_i - cross(omega_j,v_i);


%%% acceleration
% euler = cross(a_i,r_i);
% coriolis = 2*cross(omega_j,v_i);
centrif = cross(omega_j,cross(omega_j,r_j));

coriolis = 2*cross(omega_j,v_j);

a_j = a_i - coriolis - centrif;

dXdt = [v_j;a_j];

%vdot_j = vdot_i + vdot_i x r_i + 2*omega_j x v_j - omega_j x omega_j x r_i
```

B.7 AttitudeDynamics.m

```
function [dXdt] = AttittudeDynamics(t, X, I)
% ode 45 function. Calculates the torque-free motion of a rigid body. This
% is done specifically for a 3-1-3 rotation in Euler rotations.
% Inputs: t - time period of integration
%         X - state vector of rotational elements of the body. The
%         structure takes the form [theta_1; theta_2; theta_3; omega_x;
%          omega_y; omega_z], where the theta angles are the Euler rotations
%          and the omegas are the angular velocities in the three axes.
%          I - inertia matrix of the body. This assumes no products of
%          inertia, i.e principle axes.
% Outputs: dXdt - derivative values of the inputs. This is done using the
%                 Euler equations for change in theta angles and angular
%                 velocity for the 3-1-3 rotation.

% Extracting the diagonal inertia terms
I_1 = I(1,1);
I_2 = I(2,2);
I_3 = I(3,3);
% Extracting the Euler angles
theta_1 = X(1);
theta_2 = X(2);
theta_3 = X(3);
% Extracting the angular velocities
omega_x = X(4);
omega_y = X(5);
omega_z = X(6);

% Euler equations for torque-free motion (L=0)
w_x_dot = ((I_2-I_3)*omega_y*omega_z)/I_1;
w_y_dot = ((I_3-I_1)*omega_z*omega_x)/I_2;
w_z_dot = ((I_2-I_1)*omega_x*omega_y)/I_3;
% B matrix for change in angular vel
B_theta_mat = (1/sin(theta_2)) * [sin(theta_3), cos(theta_3), 0;
sin(theta_2)*cos(theta_3), -sin(theta_2)*sin(theta_3),     0;
-cos(theta_2)*sin(theta_3), -cos(theta_2)*cos(theta_3),  sin(theta_2)];
% Change in angular vel
theta_dot = B_theta_mat*[omega_x;omega_y;omega_z];

% Output vector
dXdt = [theta_dot(1);theta_dot(2);theta_dot(3);w_x_dot;w_y_dot;w_z_dot];
```