Using the Theory of Functional Connections to Solve the Restricted 3-Body Two-Point Boundary Value Problem

Juliana Chew, Alex Koenig

May 2021

Contents

Abstract

In this work, we set out to write a method to solve the restricted 3-body two-point boundary value problem: given an initial location, final location, and time of flight, we desire to compute a valid trajectory for the orbit. We rely on the Theory of Functional Connections to split the motion into two classes of equations. The former are fixed equations that satisfy the boundary conditions, and the latter are free equations that are adapted via nonlinear least-squares optimization to best fit the 3-body equations of motion. We then compare our results to a numerical integrator to verify its success and highlight certain solution attributes.

1 Problem Formulation: Restricted 3-Body Problem

This section formulates the relevant equations of motion and details the system variables. In this project we focused on the Earth-Moon system, and used the non-dimensionalized version of the problem where $G = 1$ and $\mu = \frac{M_{moon}}{M_{moon}+M_{Earth}} = 0.01215$. The Earth is located statically at ($-\mu$,0,0), and the moon is located statically at ($1 - \mu$,0,0). An orbital period of the moon is exactly 2π.
Given our problem set-up, the equations of motion are as follows:

$$F_x = \ddot{X}_1 - 2\dot{X}_2 - X_1 + \frac{(1-\mu)(X_1+\mu)}{r_1^3} + \frac{\mu(X_1-1+\mu)}{r_2^3} = 0 \tag{1}$$

$$F_y = \ddot{X}_2 + 2\dot{X}_1 - X_2 + \frac{(1-\mu)X_2}{r_1^3} + \frac{\mu X_2}{r_2^3} = 0 \tag{2}$$

$$F_z = \ddot{X}_3 + \frac{(1-\mu)X_3}{r_1^3} + \frac{\mu X_3}{r_2^3} = 0 \tag{3}$$

Where $r_1 = \sqrt{(X_1+\mu)^2 + X_2^2 + X_3^2}$ (the distance from the object to Earth) and $r_2 = \sqrt{(X_1-1+\mu)^2 + X_2^2 + X_3^2}$ (the distance from the object to the moon).

To find orbital trajectories that satisfy the boundary conditions, we have 4 variables of interest:

1. $\vec{X}_0$, the initial position
2. $\vec{X}_f$, the final position
3. t_0, the initial time (at $\vec{X}_0$), set as 0 throughout the rest of this paper
4. t_f, the final time (at $\vec{X}_f$)

2 Method: Theory of Functional Connections

Several algorithmic methods to solve the 3-body two-point boundary value problem (TPBVP) exist, but we downselected to TFC for its relatively high computational efficiency, as demonstrated by Johnston et al (2021)[1], where TFC was implemented specifically for the Earth-Moon system.

The overarching strategy of the TFC method is to solve the 3-body equations of motion given boundary value constraints by iterating over possible solutions until the orbit sufficiently fits the equations of motion. First, in Section 2.1, the space of possible orbit solutions is defined by specifying separate constraints and free functions — the constraints match the position boundary values and are therefore constant, whereas the free function is adaptable in order to fit the equations of motion. Next, in Section 2.2, the free functions are specified with a set of basis functions and corresponding coefficients;

[1] Johnston et al, Fast 2-impulse non-Keplerian orbit-transfer using the Theory of Functional Connections. https://arxiv.org/abs/2102.11837

it is these coefficients which are iteratively refined via nonlinear least-squares optimization to find a satisfactory orbit. In Section 2.3, the loss function for the equations of motion is defined, which computes the generated orbit's consistency with the equations of motion to be used in the nonlinear least-squares optimizer. In Section 2.4, the nonlinear least-squares optimizer is described, which combines previous components to solve for the orbit.

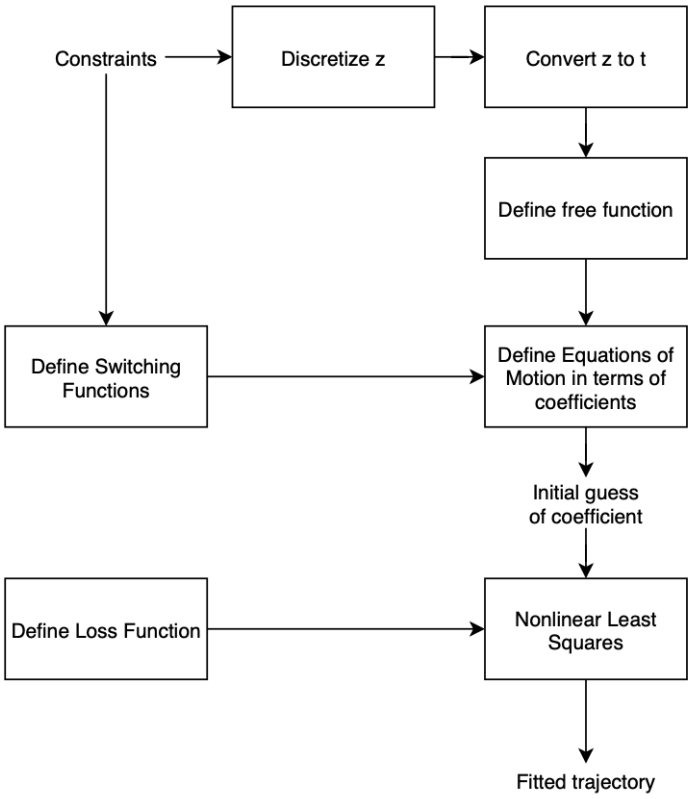

Figure 1: A diagram of our TFC procedure.

2.1 Separating Constraints from Free Functions

The Theory of Functional Connections performs functional interpolation given a set of constraints. We build upon the definitions given by Johnston et al (2021)[2] to create

[2]Johnston et al, Fast 2-impulse non-Keplerian orbit-transfer using the Theory of Functional Connections. https://arxiv.org/abs/2102.11837

equations for object position in each axis as follows:

$$X_i(t, g_i(t)) = g_i(t) + \sum_{j=1}^{k} \phi_j(t)\rho_j(t, g_i(t)), \tag{4}$$

where X_i defines the location of a body on the i^{th} axis ($X_1 = x$, $X_2 = y$, and $X_3 = z$), and $g_i(t)$ is the free function for the i^{th} axis. The k equations of $\phi_j(t)$ are switching functions for the k given constraints (i.e. they are 1 at the constraint). Each $g_i(t)$ is a free function which is iteratively altered within the nonlinear least squares solver. See 2.2 for more details. The ρ_j equations are the projection functionals that project $g_i(t)$ onto the domain of functions that meet our constraints.

Similar to Johnston et al (2021)[3], we define the projection functionals as follows:

$$\rho_1(t, g_i(t)) = X_{i0} - g_i(t_0) \tag{5}$$
$$\rho_2(t, g_i(t)) = X_{if} - g_i(t_f) \tag{6}$$

Thus, we can rewrite our X_i as follows:

$$X_i(t, g_i(t)) = g_i(t) + \phi_1(t)(X_{i0} - g_i(t_0)) + \phi_2(t)(X_{if} - g_i(t_f)) \tag{7}$$

As we aim to fit our model to the equations of motion, we express our problem as follows:

$$\begin{aligned} &F_i(t, X_j, \dot{X}_j, \ddot{X}_j) = 0 \\ &\text{such that } X_j(t_0) = X_{0,j} \text{ and } X_j(t_f) = X_{f,j} \forall i, j \in \{1, 2, 3\} \end{aligned} \tag{8}$$

Where $X_j(t, g_i(t))$ is defined in equation 7 above.

Given we have 2 boundary points at t_0 and t_f, we thus define our switching functions $\phi_j(t)$ as

$$\phi_1(t_0) = 1 \qquad \phi_1(t_f) = 0 \qquad \phi_2(t_f) = 1 \qquad \phi_2(t_0) = 0 \tag{9}$$

Where each $\phi_j(t)$ can themselves be expressed as a linear combination of coefficients α_{ij} :

$$\phi_1(t_0) = 1 * \alpha_{11} + t_0 * \alpha_{21} = 1 \tag{10}$$

All other $\phi_i(t)$ in the equations above can be described similarly. We can express all four equations using matrices:

$$\begin{bmatrix} 1 & t_0 \\ 1 & t_f \end{bmatrix} \begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{11}$$

[3]Johnston et al, Fast 2-impulse non-Keplerian orbit-transfer using the Theory of Functional Connections. https://arxiv.org/abs/2102.11837

We can thus find all α parameters:

$$\begin{bmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{21} & \alpha_{22} \end{bmatrix} = \begin{bmatrix} \frac{t_f}{t_f - t_0} & \frac{-t_0}{t_f - t_0} \\ \frac{-1}{t_f - t_0} & \frac{1}{t_f - t_0} \end{bmatrix} \tag{12}$$

By doing so, we can rewrite our switching functions as follows:

$$\phi_1(t) = \frac{t_f - t}{t_f - t_0} \qquad \phi_2(t) = \frac{t - t_0}{t_f - t_0} \tag{13}$$

Setting $t_0 = 0$ since we only care about $\Delta t = t_f - t_0$, we can further simplify:

$$\phi_1(t) = \frac{t_f - t}{t_f} \qquad \phi_2(t) = \frac{t}{t_f} \tag{14}$$

When calculating $\dot{X}_i$ and $\ddot{X}_i$, we find that

$$\dot{\phi}_1(t) = -\frac{1}{t_f} \qquad \dot{\phi}_2(t) = \frac{1}{t_f} \qquad \ddot{\phi}_1(t) = 0 \qquad \ddot{\phi}_2(t) = 0 \tag{15}$$

2.2 Defining the Free Functions $\mathbf{g_i(t)}$

To create scalar coefficients which describe the orbits, our free functions $g_i(t)$ can be expressed as

$$g_i(t) = h^T \xi_i \tag{16}$$

Where $h \in \mathbb{R}^m$ is the vector of basis functions (in this case, Chebyshev polynomials of the first kind) defined in z-space such that $z \in \{-1, 1\}$ and $\xi_i \in \mathbb{R}^m$ is the unknown coefficient vector that we are solving for as specified earlier. Each ξ_i corresponds to the i^{th} component. For simplicity, we set $m = 3$, using only the first three Chebyshev polynomials for the vector of basis functions, which means that our entire trajectory is approximated as only a 2nd-order polynomial in each axis — this limitation is discussed further in Section 3 and 4. The Chebyshev polynomials we use are:

$$T(x) = 1 \tag{17}$$

$$T(x) = x \tag{18}$$

$$T(x) = 2x^2 - 1 \tag{19}$$

And therefore h is:

$$h = [1, x, 2x^2 - 1] \tag{20}$$

The z-space for h is separate from our t-space, where the following conversions can be used to switch between the two:

$$z = z_0 + \frac{z_f - z_0}{t_f - t_0}(t - t_0) \qquad t = t_0 + \frac{t_f - t_0}{z_f - z_0}(z - z_0) \tag{21}$$

Given that $z_0 = -1$, $z_f = 1$, and $t_0 = 0$, we can further simplify to

$$z = -1 + \frac{2}{t_f}(t) \qquad t = \frac{t_f}{2}(z + 1) \tag{22}$$

As shown in Johnston et al (2021)[4], the derivatives of $g_i(t)$ can thus be written as

$$\begin{aligned} \frac{d^n g_i}{dt^n} &= \Big(\frac{dz}{dt}\Big)^n \frac{d^n h^T}{dz^n}\xi_i \\ \text{where } \frac{dz}{dt} &= \frac{z_f - z_0}{t_f - t_0} = \frac{2}{t_f} \end{aligned} \tag{23}$$

These derivatives are used for finding $\dot{X}_i$ and $\ddot{X}_i$ using equation 4:

$$\begin{aligned} \dot{X}_i(t, g_i(t)) = \dot{g}_i(t) - \dot{g}_i(0)\phi_1(t) - \frac{X_{0i} - g_i(0)}{t_f} \\ - \phi_2(t)\dot{g}_i(t_f) + \frac{X_{fi} - g_i(t_f)}{t_f} \end{aligned} \tag{24}$$

$$\begin{aligned} \ddot{X}_i(t, g_i(t)) = \ddot{g}_i(t) + \frac{\dot{g}_i(0)}{t_f} - \ddot{g}_i(0)\phi_1(t) + \frac{\dot{g}_i(0)}{t_f} \\ - \frac{\dot{g}_i(t_f)}{t_f} - \ddot{g}_i(t_f)\phi_2(t) - \frac{\dot{g}_i(t_f)}{t_f} \end{aligned} \tag{25}$$

2.2.1 Discretization of z

To reduce error at the constrained boundary points, we discretized z (and thus, t) using the Chebyshev-Gauss-Lobatto nodes. When finding N+1 nodes, the discretization scheme is as follows:

$$z_k = -\cos\frac{k\pi}{N} \text{ for k = } 0, 1, 2, ...N \tag{26}$$

By finding z values this way, we are able to perform nonlinear least squares with a higher resolution near the end points, thereby reducing error at the orbit boundaries.

2.3 Defining the Loss Function $\mathbf{L}(\boldsymbol{\Xi})$

For given boundary values, the coefficients Ξ uniquely define an orbit (where Ξ is the concatenation of ξ_1, ξ_2, and ξ_3). The loss function uses the equations of motion

[4]Johnston et al, Fast 2-impulse non-Keplerian orbit-transfer using the Theory of Functional Connections. https://arxiv.org/abs/2102.11837

expressed in terms of these coefficients Ξ — i.e., $\widetilde{F}_i(t, \Xi)$ — to determine the consistency of the orbit with the equations of motion. An orbit perfectly consistent with the equations of motion will have a loss of 0; realistically, due to numerical imprecision, approximation of the orbit via polynomials, and finite computational time, the loss will be a locally minimized non-zero value.

The loss can only be calculated numerically at discretized points in time, so the overall loss for the orbit is computed as a vector of the losses at n discretized time values throughout the orbit.

$$\begin{aligned} L(\Xi) = \Big[&\widetilde{F}_1(t_0, \Xi), ..., \widetilde{F}_1(t_i, \Xi), ..., \widetilde{F}_1(t_f, \Xi), \\ &\widetilde{F}_2(t_0, \Xi), ..., \widetilde{F}_2(t_i, \Xi), ..., \widetilde{F}_2(t_f, \Xi), \\ &\widetilde{F}_3(t_0, \Xi), ..., \widetilde{F}_3(t_i, \Xi), ..., \widetilde{F}_3(t_f, \Xi)\Big] \end{aligned} \tag{27}$$

2.4 Performing Nonlinear Least-Squares Optimization

The nonlinear least-squares optimizer starts with an initial guess of the orbit, Ξ_0, and iteratively updates the guess to reduce loss:

$$\Xi_{k+1} = \Xi_k + \Delta\Xi \tag{28}$$

With $\Delta\Xi$ computed as:

$$\Delta\Xi = -(J^T(\Xi_k)J(\Xi_k))^{-1}J^T(\Xi_k)L(\Xi_k) \tag{29}$$

Where $L(\Xi)$ is the loss function described in Section 2.3, and $J(\Xi)$ is the Jacobian of the loss. For this project, we did not define $J(\Xi)$, nor did we specify the exact update equation, but rather passed on the loss function and variables of interest to the built-in SciPy nonlinear least-squares optimizer which handles the numerical updates on its own.

For our initial guess of the orbit, we simply used $\Xi_0 = \vec{0}$ (i.e., a vector of 0s of the length of Ξ, in our case, 9). This guess represents a linear trajectory from the initial to final location. A better initial guess would allow faster runtime, but we found this guess to be sufficient to find a solution in all explored cases.

3 Results

To analyze results, we chose a few particular orbits that highlight certain attributes of the solver. After finding a particular solution, we extracted the initial velocity from the trajectory and used that within a 3-body numerical integrator to assess the validity of the initial conditions that our method generated.

The orbits we discuss in the examples below are outlined here:

Orbit #	t_f	$\vec{X}_0$	$\vec{X}_f$ (TFC)	$\vec{X}_f$ (Integration)
1	3.45	(1.15, 0, 0)	(0, -1.15, 0)	(0.904, 0.057, 0)
2	2	(5, 0, 0)	(3, -3, 0)	(5.28, -6.19, 0)
3	1.5	(0, -0.8, 0)	(0, 0.8, 0)	(0.484, 1.588, 0)
4	1.8	(0, -0.8, 2)	(0, 0.8, 1)	(1.057, -1.250, 0.586)

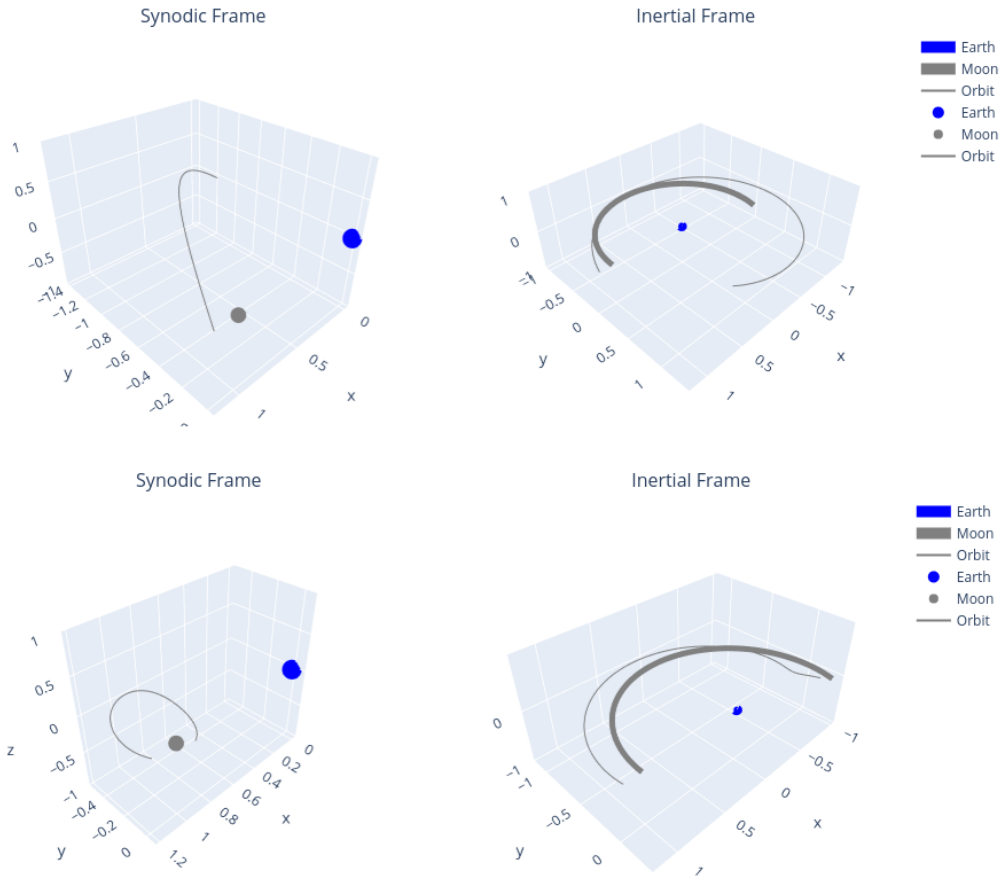

Figure 2: Orbit 1. The top two scenes show the TFC solution within the synodic frame (left) and inertial frame (right). The bottom computes the trajectories via numerical integration from the initial state output by the TFC solver. The trajectories match initially but then quickly diverge; the TFC-computed trajectory escapes lunar influence entirely, whereas numerical integration suggests that the initial conditions provided by TFC actually cause the body to roughly stay in orbit around the moon.

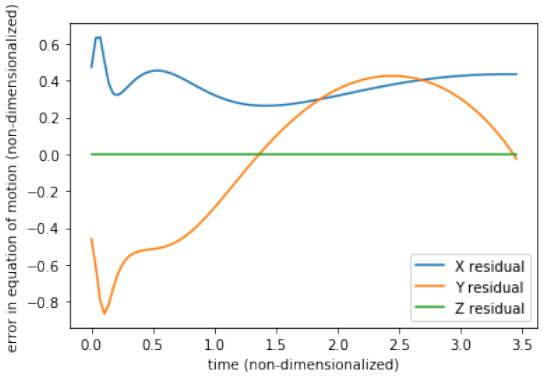

Figure 3: Orbit 1 equation of motion residuals. The high initial residuals at the beginning of the trajectory suggest that the initial velocity and acceleration described by the TFC-derived orbit are not an ideal fit, which helps explain the discrepancy between the TFC-derived orbit and the numerically integrated orbit. Even if the residuals are 0 throughout the rest of the orbit, if the residuals at the start of the orbit are non-zero, the initial velocity estimate will be wrong, and so the TFC solution will not match numerical integration.

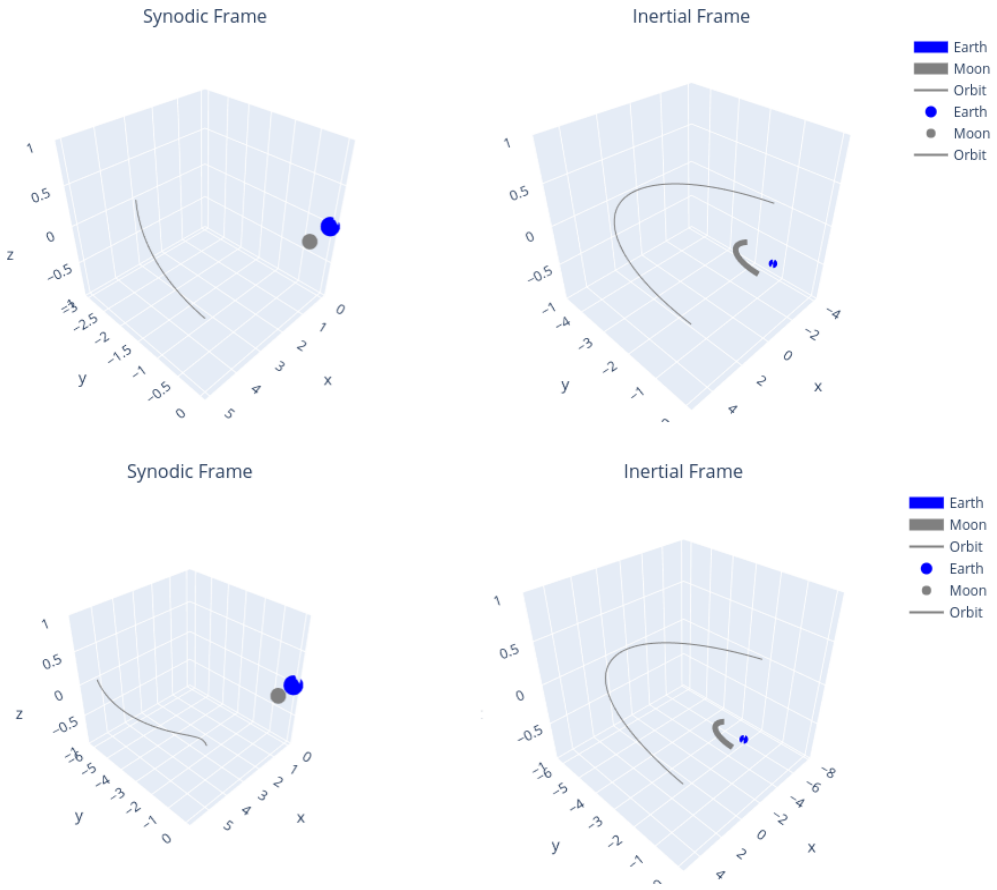

Figure 4: Orbit 2, TFC (top) and integration (bottom). The actual shapes of the orbits match relatively strongly, but the numerically-integrated orbit extends out to a further distance away from the system's center of mass. Since the orbiting body starts far away from the Earth-moon system and thus has near-0 total energy, small errors in the initial velocity result in relatively large position changes.

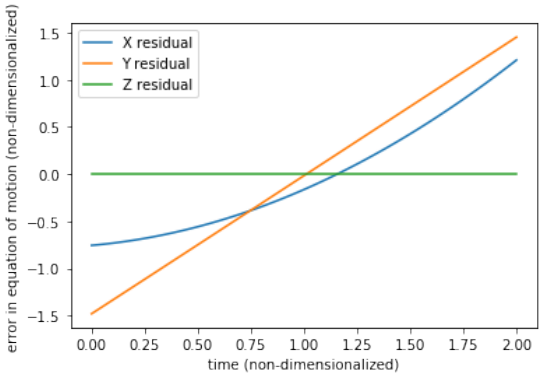

Figure 5: Orbit 2 equation of motion residuals. Although the average residual is close to 0, as desired, there is a consistent upward trend which suggests the orbit solution is non-ideal. As in orbit 1, the residuals at t_0 are particularly extreme, worsening the initial velocity estimate.

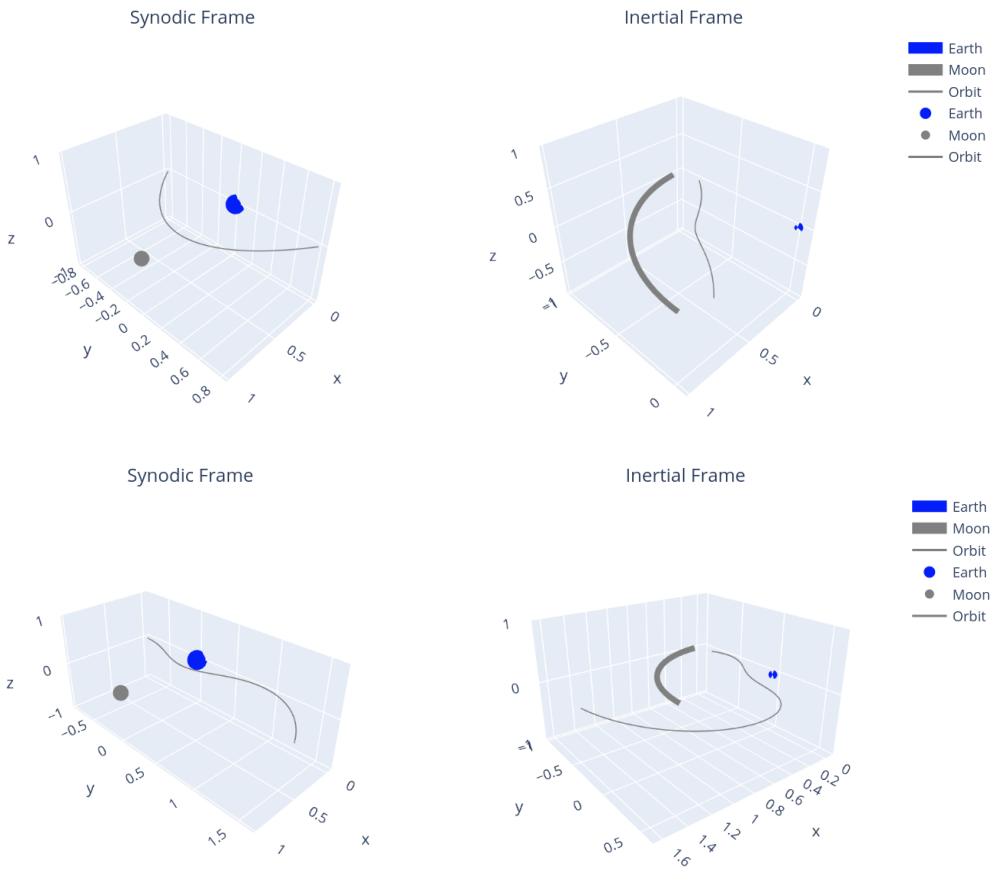

Figure 6: Orbit 3, TFC (top) and integration (bottom). Here, the constraint of the orbit as a quadratic in each dimension clearly constrains the accuracy it can achieve, since the actual orbital shape is rather convoluted due to 3-body effects. Better performance would be expected with a higher-order or piecewise polynomial.

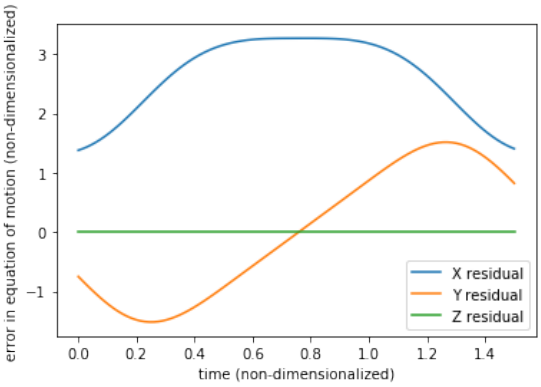

Figure 7: Orbit 3 equation of motion residuals. The residuals average to 0 for the Y axis but not for the X axis.

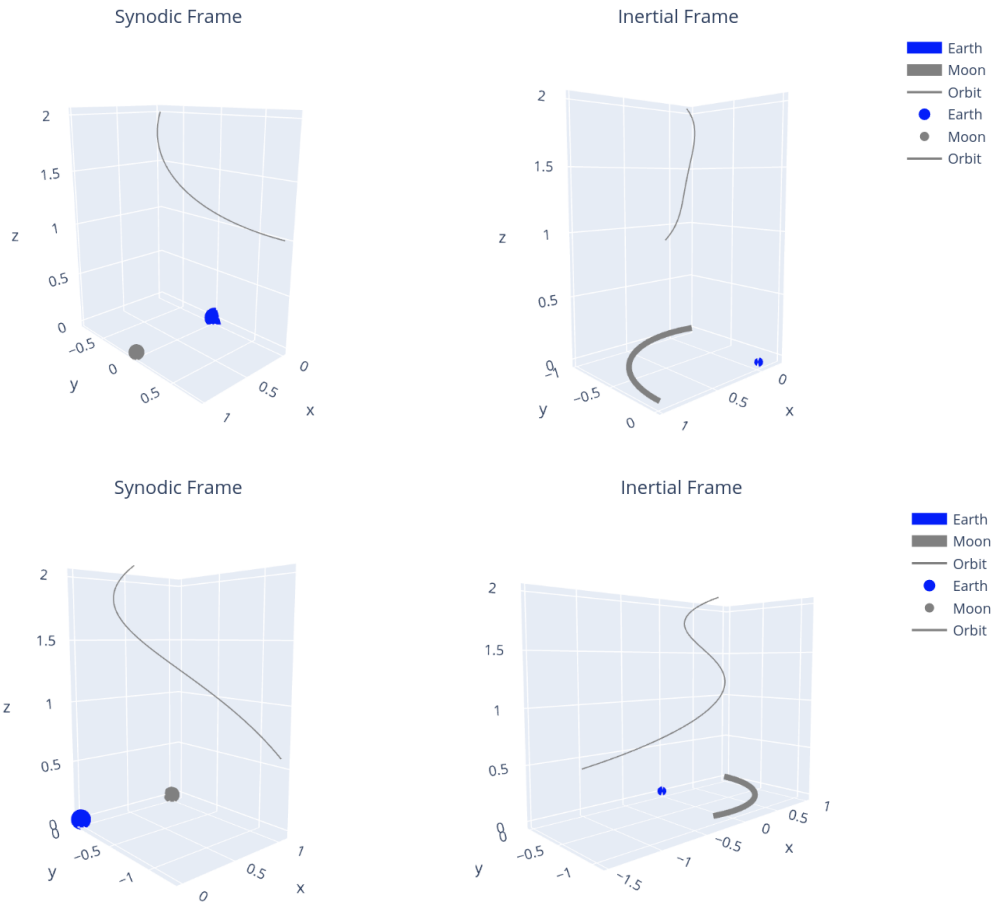

Figure 8: Orbit 4, TFC (top) and integration (bottom). The orbits share similar shapes but vary in orientation, as they appear to be rotated with respect to each other.

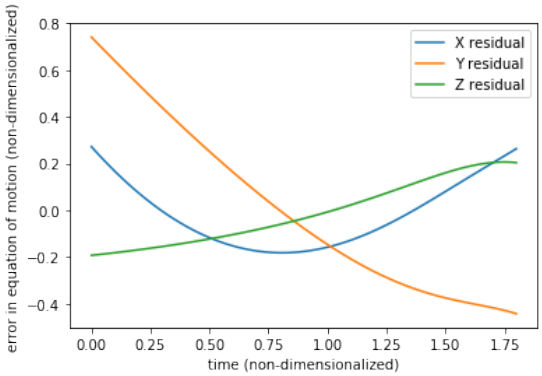

Figure 9: Orbit 4 equation of motion residuals. The residuals are large at the boundaries and relatively small in between.

4 Conclusion

Overall, the performance of our TFC-based method is mixed, and is highly dependent on the selected boundary conditions. Where the actual resulting orbit takes on a more convoluted shape, the TFC solution is generally less accurate because its orbit is only able to be described by a 2nd-order polynomial in each dimension. The TFC solution works better for smoother orbits that are located further away from regions where 3-body effects are dominant (i.e., near the moon and its Lagrange points). Noteworthily, the residuals of the equations of motion tend to be high at the boundaries and low in between; ideally, if anything, the reverse would be true, so that the velocities at the start and end of the orbit are estimated more precisely.

We have several ideas to improve performance. For one, we could use piecewise quadratic approximation, using unique Ξs between time increments to better fit the equations of motion — this would allow for a closer fit of the trajectory to the actual trajectory, and would enable the solver to achieve lower residuals overall. Since we currently use a single Ξ for the entire trajectory, TFC's output is only a 2nd-order polynomial in each axis across the entire trajectory. This limitation is tolerable in certain conditions where the real orbit solution is smooth in nature, but as shown previously, it does affect the general trajectory shape particularly where 3-body effects are dominant. (It is also worthwhile to note that elliptical orbits cannot feasibly be approximated as quadratics, whereas they could be approximated as piecewise quadratics).

Another possibility is to increase the number of basis functions used in the vector of basis functions, h, and correspondingly increase the number of unknown coefficients Ξ. This could be done instead of or in tandem with using piecewise functions to describe the trajectory. If we added n more basis functions, the trajectory could be approximated as a $(2+n)^{th}$-order polynomial rather than a quadratic. This strategy is less desirable

than the piecewise approach because higher-order polynomials will have worse velocity errors at the boundaries, which will negatively impact the estimate of the initial velocity.

TPBVP_TFCSolver

May 19, 2021

0.0.1 Imports

[1]:
```
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

0.0.2 Set boundary conditions

[2]:
```
global mu
mu = 0.01215

global X0
global Xf
global tf

X0 = np.array([1.15,0,0])
Xf = np.array([0,-1.15,0])
tf = 3.45
```

0.0.3 Definitions for g & derivatives

[3]:
```
def g(i,t,Xi):
    h = np.array([1,2*t/tf - 1,2*(2*t/tf - 1)**2 - 1])
    xi = Xi[i:i+3]

    return np.dot(h,xi)

def gdot(i,t,Xi):
    h = np.array([0,2/tf,2/tf*(8*t/tf-4)])
    xi = Xi[i:i+3]

    return np.dot(h,xi)
```

```
def gddot(i,t,Xi):
    h = np.array([0,0,16/tf**2])
    xi = Xi[i:i+3]

    return np.dot(h,xi)
```

0.0.4 Definitions for X & derivatives

[4]:
```
def X(i,t,Xi):
    return g(i,t,Xi) + (1-t/tf)*(X0[i] - g(i,0,Xi)) + (t/tf)*(Xf[i] -␣
↪g(i,tf,Xi))

def Xdot(i,t,Xi):
    #return gdot(i,t,Xi) + (1-t/tf)*(-gdot(i,0,Xi)) + (X0[i] - g(i,0,Xi))*(-1/
↪tf) + (t/tf)*(-gdot(i,tf,Xi)) + (Xf[i] - g(i,tf,Xi))*(1/tf)
    return gdot(i,t,Xi) - gdot(i,0,Xi)*(1-t/tf) - (X(i,0,Xi) - g(i,0,Xi))/tf -␣
↪(t/tf)*gdot(i,tf,Xi) + (X(i,tf,Xi)-g(i,tf,Xi))/tf

def Xddot(i,t,Xi):
    return gddot(i,t,Xi) + 2*gdot(i,0,Xi)/tf + (1-t/tf)*(-gddot(i,0,Xi)) -␣
↪gdot(i,tf,Xi)*2/tf + (t/tf)*(-gddot(i,tf,Xi))
```

[5]:
```
def r(i,t,Xi):
    if i==1:
        return ((X(0,t,Xi)+mu)**2 + X(1,t,Xi)**2 + X(2,t,Xi)**3)**0.5
    if i==2:
        return ((X(0,t,Xi)-1+mu)**2 + X(1,t,Xi)**2 + X(2,t,Xi)**3)**0.5
```

0.0.5 Equations of Motion

[6]:
```
def F(i,t,Xi):
    if i==1:
        x1ddot = Xddot(0,t,Xi)
        x2dot = Xdot(1,t,Xi)
        x1 = X(0,t,Xi)
        r1 = r(1,t,Xi)
        r2 = r(2,t,Xi)

        return x1ddot - 2*x2dot - x1 + (1-mu)*(x1+mu)/r1**3 + mu*(x1-1+mu)/r2**3

    if i==2:
        x2ddot = Xddot(1,t,Xi)
        x1dot = Xdot(0,t,Xi)
        x2 = X(1,t,Xi)
```

```
        r1 = r(1,t,Xi)
        r2 = r(2,t,Xi)

        return x2ddot + 2*x1dot - x2 + (1-mu)*x2/r1**3 + mu*x2/r2**3

    if i==3:
        x3ddot = Xddot(2,t,Xi)
        x3 = X(2,t,Xi)
        r1 = r(1,t,Xi)
        r2 = r(2,t,Xi)

        return x3ddot + (1-mu)*x3/r1**3 + mu*x3/r2**3
```

0.0.6 Loss function

[7]:
```
def Loss(Xi):
    loss = []
    n = 100
    for i in range(n):
        t = tf*i/(n-1)
        loss.append(F(1,t,Xi))
    for i in range(n):
        t = tf*i/(n-1)
        loss.append(F(2,t,Xi))
    for i in range(n):
        t = tf*i/(n-1)
        loss.append(F(3,t,Xi))

    return np.asarray(loss)
```

0.0.7 Run the solver on an initial guess

[8]:
```
guess = np.array([0,0,0,0,0,0,0,0,0])
result = scipy.optimize.least_squares(Loss,guess)
Xi = result.x
```

0.0.8 Plot the resulting orbit

[9]:
```
def plot(Xi,flattenZ=True):
    def rotate_vector(vec, axis, angle_deg):
        proj = np.dot(vec, axis)*axis
        return proj + np.cos(angle_deg*np.pi/180)*(vec - proj) + np.
 ↪sin(angle_deg*np.pi/180)*np.cross(vec,axis)
```

```
    rs = [rotate_vector(np.array([X(0,t,Xi),X(1,t,Xi),X(2,t,Xi)]),np.
↪array([0,0,1]),57.3*t) for t in np.linspace(0,tf,100)]
    xsi = [r[0] for r in rs]
    ysi = [r[1] for r in rs]
    zsi = [r[2] for r in rs]
    rsE = [rotate_vector(np.array([-mu,0,0]),np.array([0,0,1]),57.3*t) for t in␣
↪np.linspace(0,tf,100)]
    xsEi = [r[0] for r in rsE]
    ysEi = [r[1] for r in rsE]
    zsEi = [r[2] for r in rsE]
    rsM = [rotate_vector(np.array([1-mu,0,0]),np.array([0,0,1]),57.3*t) for t␣
↪in np.linspace(0,tf,100)]
    xsMi = [r[0] for r in rsM]
    ysMi = [r[1] for r in rsM]
    zsMi = [r[2] for r in rsM]

    xs = [X(0,t,Xi) for t in np.linspace(0,tf,100)]
    ys = [X(1,t,Xi) for t in np.linspace(0,tf,100)]
    zs = [X(2,t,Xi) for t in np.linspace(0,tf,100)]
    xsE,ysE,zsE = [-mu],[0],[0]
    xsM,ysM,zsM = [1-mu],[0],[0]

    if flattenZ:
        zs = [0 for t in np.linspace(0,tf,100)]
        zsi = [0 for t in np.linspace(0,tf,100)]

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scene'}, {'type':␣
↪'scene'}]], subplot_titles=("Synodic Frame", "Inertial Frame"))
    fig.add_trace(
        go.Scatter3d(x=xsEi, y=ysEi,␣
↪z=zsEi,mode='lines',marker=dict(color='blue'),line=dict(width=15),name='Earth'),
        row=1, col=2)
    fig.add_trace(
        go.Scatter3d(x=xsMi, y=ysMi,␣
↪z=zsMi,mode='lines',marker=dict(color='gray'),line=dict(width=10),name='Moon'),
        row=1, col=2)
    fig.add_trace(
        go.Scatter3d(x=xsi, y=ysi,␣
↪z=zsi,mode='lines',marker=dict(color='gray'),name='Orbit'),
        row=1, col=2)
    fig.add_trace(
        go.Scatter3d(x=xsE, y=ysE,␣
↪z=zsE,mode='markers',marker=dict(color='blue',size=10),name='Earth'),
        row=1, col=1)
    fig.add_trace(
```

```
        go.Scatter3d(x=xsM, y=ysM,␣
↪z=zsM,mode='markers',marker=dict(color='gray'),name='Moon'),
        row=1, col=1)
    fig.add_trace(
        go.Scatter3d(x=xs, y=ys,␣
↪z=zs,mode='lines',marker=dict(color='gray'),name='Orbit'),
        row=1, col=1)
    #fig.write_html('orbit.html', auto_open=True)
    fig.show()
```

[10]:
```
plot(Xi,flattenZ=True)
```