Numerical Methods for Dynamical Systems
Assignment 10
Analysis if the impact of μ on the RTBP
UPC, Fme

Written by
Ole Gunnar Røsholt Hovland

2023

Abstract

In this assignment, we explore the dynamics of the RTBP as the mass ratio μ varies. The assignment involves plotting the unstable manifold of the L3 Lagrange point for different μ values, revealing various behaviors and tendencies in the system. The assignment addresses complexities in plot artifacting and refining accuracy. Further, it examines how increasing μ simplifies the manifold and influences the orbits' predictability. The assignment is supported by code segments demonstrating the computational methods used.

Contents

1 Part A

1.1 6.a

Here we plot the unstable manifold until it crosses the Poincare section for the first time, given that $\mu = 0.008$.

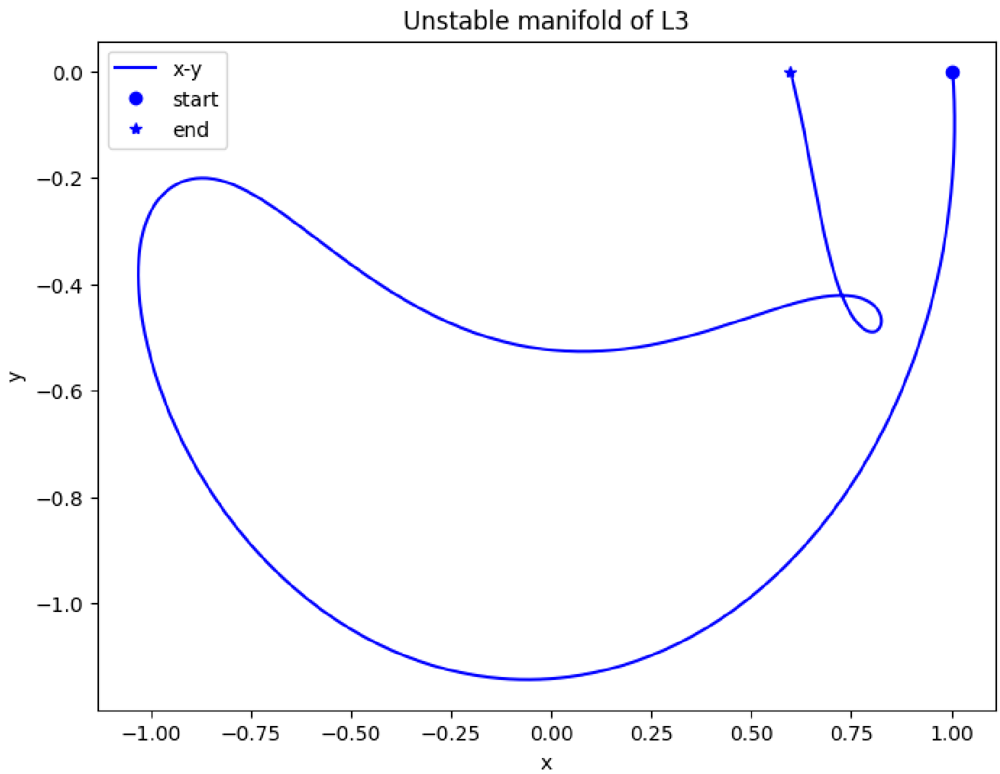

Figure 1.1: The figure shows the plot of the unstable manifold of $L3$ until it crosses the y-axis

1.2 6.b

Here in Figure 1.2 we see the plot for $\mu \in [0.001, 0.5]$. We can see some general tendencies, but a greater accuracy close to zero is needed.

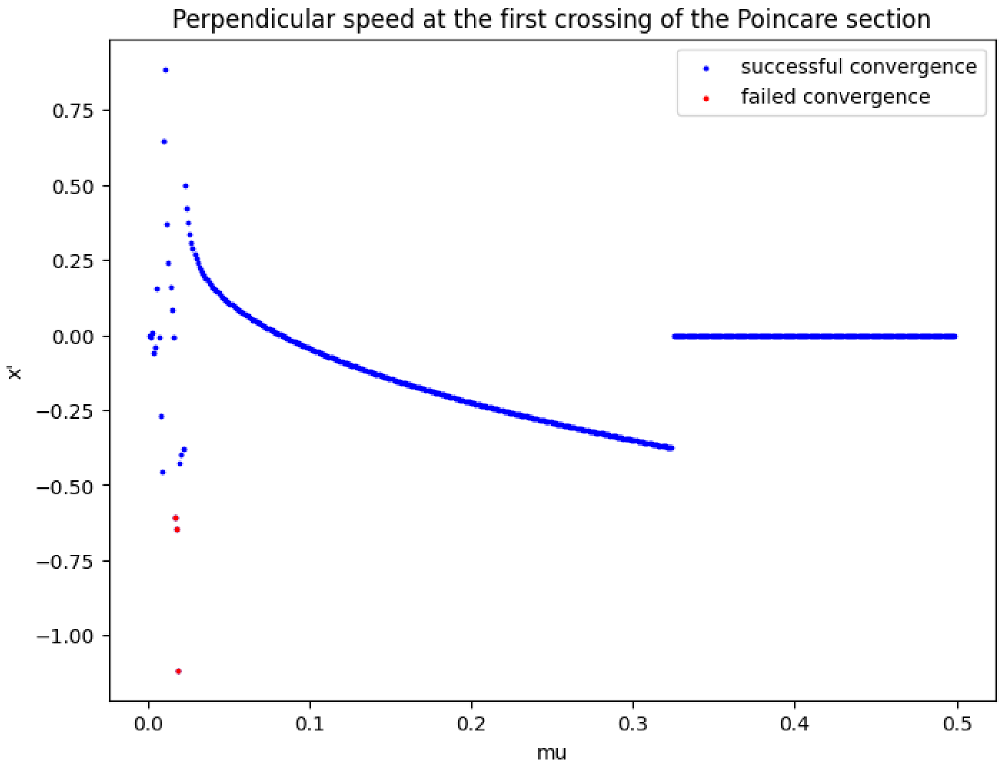

Figure 1.2: Figure showing the perpendicular speed at the first crossing of the Poincare section as a function of mu

1.3 6.c

Here, in Figure 1.3 is a more zoomed in image on the interval $\mu \in [0.001, 0.1]$. There is some unexpected behavior here, that will be addressed, and fixed, by increasing the accuracy of our code.

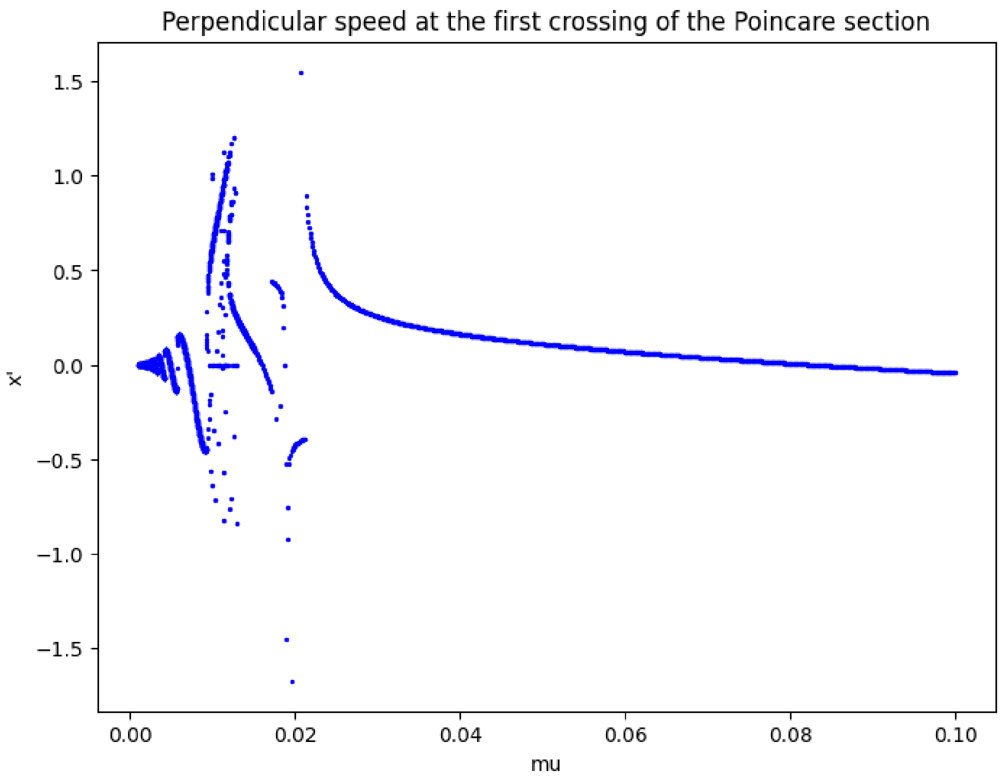

Figure 1.3: Figure showing the perpendicular speed at the first crossing of the Poincare section as a function of mu

The artifacting in our plot stems from a complexity hiccup. Sometimes, the manifold uses very little time between the first and the second crossing of the Poincare section. In our rough search for the crossing, before the application of Newtons method, our interval may be too great to catch the first crossing. Places in the plot where there seems to be more than one continuous line at a given μ is where we sometimes plot the first crossing, and other times the second or third. Bellow is a figure of one such value of μ that slipped past the rough search.

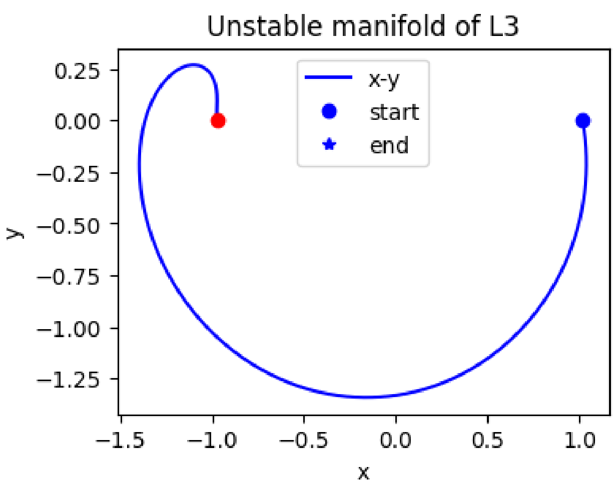

Figure 1.4: The figure shows the plot of the unstable manifold of L3 until it crosses the y-axis twice.

1.4 6.d

The code is at the bottom of the PDF for better readability of the submission.

2 Part B

Looking at the Figure 2.7, noting the enhanced intervals in Figure 2.8 (The intervals are an approximate reference to the continuous line in question):

$[0.001, 0.0085]$: these periodic "snaps" are from small loops like in Figure 1.1. In that figure, there is only one loop left, but the closer we go to zero the more loops we have on the orbit. Close to zero one of the bodies dominates the other. we are essentially almost falling out of the L3 orbit, and into an orbit around the big mass (residing almost at the origin), crashing into the tiny mass.

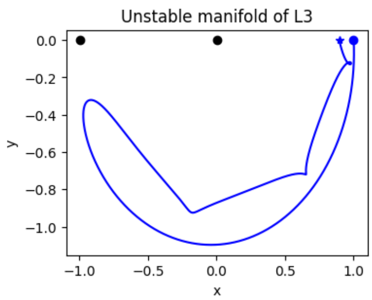

Figure 2.1: The figure shows the plot with mu = 0.004. The black dots are the two masses.

$[0.0085, 0.0116]$: After the last loop snaps, our horseshoe-shaped orbit approaches at its apexes the smaller body. The first y crossing approaches the larger body as we increase mu, until it crashes into the singularity, creating the asymptote on our plot.

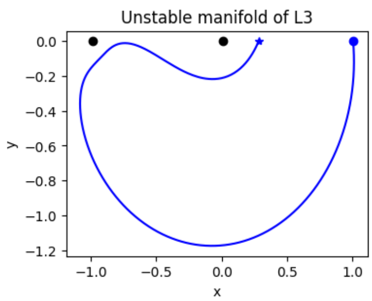

Figure 2.2: The figure shows the plot with mu = 0.0116. The black dots are the two masses.

$[0.0085, 0.016]$: The orbit begins by resembling a heart, but becomes more bean-like as our poincare-crossing approaches the second mass. Finally it collides with the smaller, yet ever increasing mass, to create our second asymptote.

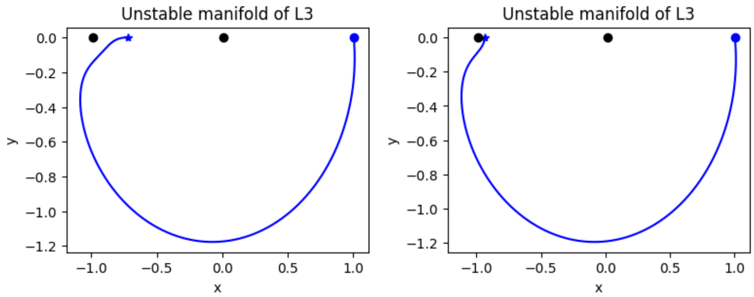

Figure 2.3: The figure shows the plot with mu = 0.01174 and mu = 0.014 respectively. The black dots are the two masses.

$[0.016, 0.02]$: We are now on the other side of the smaller mass, now "bouncing" into the smaller mass. As we approach the small mass we get closer to the axis running through the two masses where the rotational force of the system is the strongest - barely pulling us out of the gravitational pull of the smaller mass.

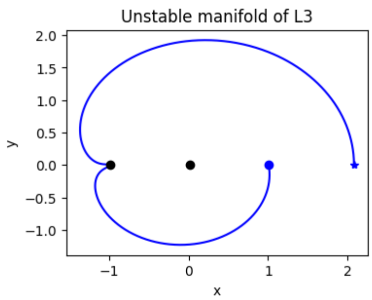

Figure 2.4: The figure shows the plot with mu = 0.0185. The black dots are the two masses.

$[0.021, 0.31]$: After some intense orbits, the unstable manifold is now orbiting outside both of the masses

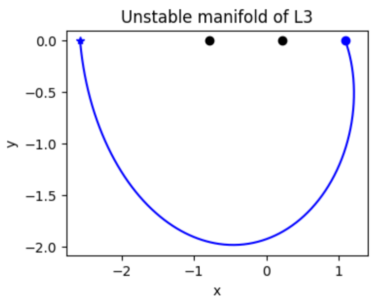

Figure 2.5: The figure shows the plot with mu = 0.22. The black dots are the two masses.

[0.31, 0.4999.]: The eigenvalues become zero. The model becomes chaotic. The total center of gravity is approaching the exact middle of the two masses, the origin.

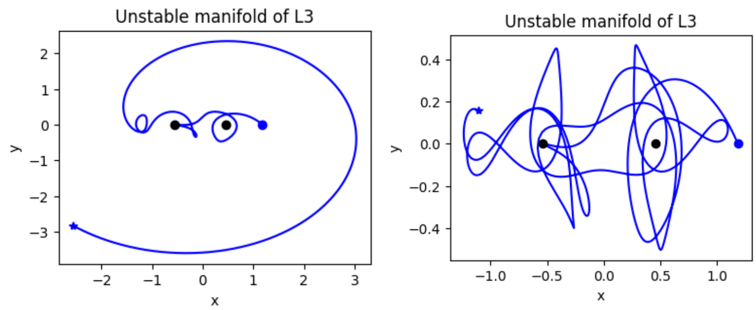

Figure 2.6: The figure shows the plot with mu = 0.45 and mu = 0.46 respectively. The black dots are the two masses.

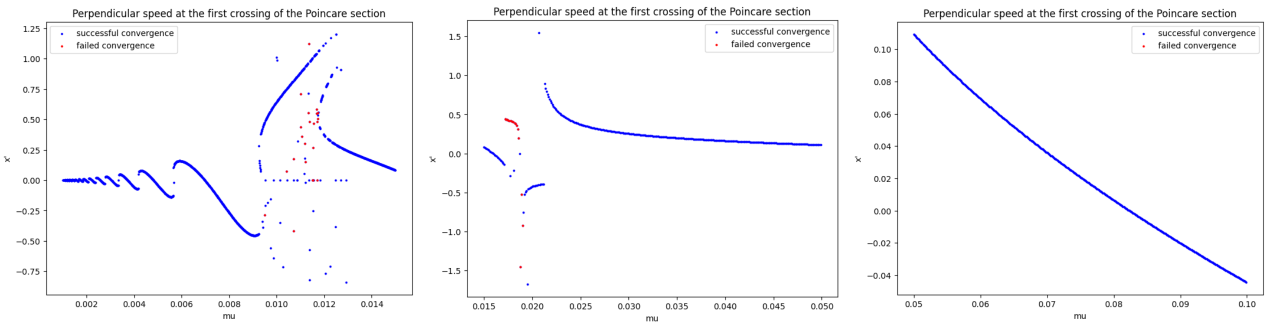

Figure 2.7: Figure showing the perpendicular speed at the first crossing of the Poincare section as a function of mu

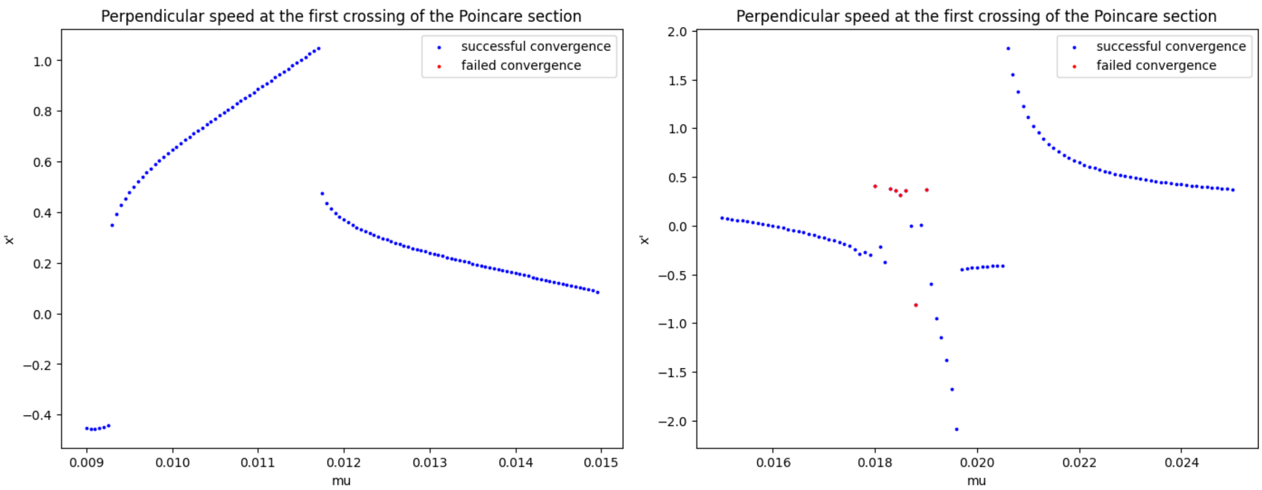

Figure 2.8: Plots of the three 3-periodic points, as well as an orbit around them

3 Code

Because of the scale of the assignment, multiple files were necessary. Below they will be presented top-down. The final file is on the top, and the necessary codes for them below. I will not repeat code to save space, but if needed I can send it all upon request(ie code for the "zoomed in" plots are extremely similar to each other, other than the interval size, and optimization variables).

Assignment 10:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import sys
dir_string = 'C:/Users/rannu/OneDrive␣-␣NTNU/Desktop/VsPython/'+\
                'Spain/NMfDS/Assignments/'
sys.path.append(dir_string + 'Ass4')
sys.path.append(dir_string + 'Ass6')
sys.path.append(dir_string + 'Ass7')
sys.path.append(dir_string + 'Ass8')
sys.path.append(dir_string + 'Ass9')
from RTBP_definitions import r1, r2, OMEGA, ODE_R3BP, Jacobi_first_integral
from Lagrange_computations import compute_Lagrange_pt,
compute_jacobi_const_Li
from custom_ODE_solver import ODE_solver
from PoincareR3BP import poincare_map_solve_ivp_R3BP
from variational_equation_RTBP import variational_eq
from crossings_RTBP import crossings_R3BP_by_mu

mu = 0.008
L3 = [compute_Lagrange_pt(mu, 3), 0, 0, 0]
L3.extend([1, 0, 0, 0,
           0, 1, 0, 0,
           0, 0, 1, 0,
           0, 0, 0, 1])  # initial conditions and identity matrix
time_span = 0

# compute the Jacobian matrix of the RTBP at Li.
# The eigenvalues of this matrix are the frequencies of the periodic orbit

A = variational_eq(time_span, L3, mu, 1)[4:20].reshape(4,4)
print('det(A)␣=␣\n', A)
eigenvalues, eigenvectors = np.linalg.eig(A)
lambda_pos = eigenvalues[3].real
lambda_neg = eigenvalues[2].real
eigvec_pos = eigenvectors[:,3]
eigvec_neg = eigenvectors[:,2]

iregion = 1

A = variational_eq(time_span, L3, mu, iregion)[4:20].reshape(4,4)
eigenvalues, eigenvectors = np.linalg.eig(A)
lambda_pos = eigenvalues[3].real
eigvec_pos = eigenvectors[:,3]
v = -eigvec_pos.real
init_cond = L3[0:4] + v*10**-6


# Find the crossing time of the unstable manifold with the Poincare section
t0 = 0
tmax = 2
dt = 1
t_span = np.arange(t0, tmax, dt)
```

```

# Variables for tweaking the accuracy of the crossing time
refinement = 2000
refinement_fine = 25
tol = 1e-6

newInitial, TimeDuration, _ = \
    poincare_map_solve_ivp_R3BP(lambda t, X: ODE_R3BP(t, mu, X), init_cond,
                                iregion, dt, t_span, mu,
                                init_search = refinement,
                                refinement = refinement_fine,
                                newton_tol = too)

tmax_plot = TimeDuration
t_span_plot = np.arange(t0, tmax_plot, 0.01)

# Solve ode_r3bp using the custom ODE solver,  RETURNS: the solution of the
# system of ODEs as a scipy.integrate.solve_ivp object
sol = ODE_solver(lambda t, X: ODE_R3BP(t, mu, X), init_cond, tmax_plot,
                 len(t_span_plot), tol=1e-12, method='DOP853', hamiltonian=0)

# Extract the solution
x = sol.y[0]
y = sol.y[1]
vx = sol.y[2]
vy = sol.y[3]

# plot the solution
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111)
ax.plot(x, y, 'b', label='x-y')
ax.plot(x[0], y[0], 'bo', label='start')
ax.plot(x[-1], y[-1], 'b*', label='end')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Unstable manifold of L3')
ax.legend()
plt.show()

from tqdm import tqdm # for the progress bar
def crossing_times_and_initials_by_mu(mu_list, Li, iregion, dt, t_span,
                                      refinement, init_tol, refinement_fine,
                                      tol, start_cond_tol):
    crossing_times = np.zeros(len(mu_list))
    crossing_initials = np.zeros((len(mu_list), 4))
    mu_fails = []
    for j, mu_val in tqdm(enumerate(mu_list, start=0)):
        crossing_times_mu, crossing_initials_mu, mu_fail = \
            crossings_R3BP_by_mu(1, Li, lambda t, X: ODE_R3BP(t, mu_val, X),
                                 iregion, dt, t_span, mu_val,
                                 refinement, init_tol, refinement_fine, tol,
                                 start_cond_tol)
        crossing_times[j] = crossing_times_mu
        crossing_initials[j] = crossing_initials_mu
        if mu_fail != -1:
```

```
            mu_fails.append(mu_val)
    return crossing_times, crossing_initials, mu_fails
```

Now for the calculation if the interval. The code is slightly modified to produce the other similar figures.

```
# test mu of only 5 different mu values
mu_listw = np.arange(0.001, 0.5, 0.001)
print("number of mu values = ", len(mu_listw))

iregion = 1

t0 = 0
tmax = 1
dt = 0.1
t_span = np.arange(t0, tmax, dt)

# Variables for tweaking the accuracy of the crossing time
refinement = 20000
refinement_fine = 25
init_tol = 1e-7
start_cond_tol = 10**-4

# tol for newton solver:
tol = 1e-6 # even a relatively large tolerance gives good results

# Test the function
crossing_timesw, crossing_initialsw, mu_failsw = \
    crossing_times_and_initials_by_mu(mu_listw, 3, 1, dt, t_span,
                                      refinement, init_tol, refinement_fine,
                                      tol, start_cond_tol)

# plot the solution
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111)
ax.scatter(mu_listw, crossing_initialsw[:,2], color='b',
           label='successful convergence', s=3)
ax.set_xlabel('mu')
ax.set_ylabel("x'")
ax.set_title('Perpendicular speed at the first crossing of the Poincare section')
# color points where mu fails red
# find the index of the mu values that failed
mu_fails_index = []
for mu_fail in mu_failsw:
    mu_fails_index.append(np.where(mu_listw == mu_fail)[0][0])
ax.scatter(mu_failsw, crossing_initialsw[mu_fails_index,2], color='r',
           label='failed convergence', s=3)
# legend for both red and blue points
ax.legend()
plt.show()
```

Similarly, code from the previous assignments, utilized in assignment 10 is written below. These are not jupyter notebook files. but .py files I import to the assignment 10 jupyter environment.

Assignment 9:

```
import numpy as np
import sys
dir_string = 'C:/Users/rannu/OneDrive␣-␣NTNU/Desktop/VsPython/'+\
                'Spain/NMfDS/Assignments/'
sys.path.append(dir_string + 'Ass4')
sys.path.append(dir_string + 'Ass6')
sys.path.append(dir_string + 'Ass7')
sys.path.append(dir_string + 'Ass8')
#from RTBP_definitions import r1, r2, OMEGA, ODE_R3BP, Jacobi_first_integral
from Lagrange_computations import compute_Lagrange_pt, compute_jacobi_const_Li
#from custom_ODE_solver import ODE_solver
from PoincareR3BP import poincare_map_solve_ivp_R3BP
from variational_equation_RTBP import variational_eq

def crossings_R3BP(no_crossings, ODE_R3BP, initial_conditions, dir,
                                step, t_span, mu, init_search=100,
                                init_tol=1e-12,
                                refinement=100, newton_tol = 1e-15):
    # Procedure to compute when the x-axis is crossed
    # for no_crossings times
    # returns an array of the crossing times
    # and a 2d array of the initial conditions at all the crossings
    crossing_times = np.zeros(no_crossings)
    crossing_initials = np.zeros((no_crossings, 4))
    mu_fails = np.zeros(no_crossings)
    new_initial, time_duration, mu_fail = poincare_map_solve_ivp_R3BP\
                                                (ODE_R3BP, initial_conditions, dir,
                                                step, t_span, mu, init_search,
                                                init_tol,
                                                refinement, newton_tol)
    crossing_times[0] = time_duration
    crossing_initials[0] = new_initial
    mu_fails[0] = mu_fail

    for i in range(1, no_crossings):
        new_initial, time_duration, mu_fail = poincare_map_solve_ivp_R3BP\
                                        (ODE_R3BP, crossing_initials[i-1], dir,
                                        step, t_span, mu, init_search,
                                        init_tol,
                                        refinement, newton_tol)
        crossing_times[i] = time_duration
        crossing_initials[i] = new_initial
        mu_fails[i] = mu_fail

    return crossing_times, crossing_initials, mu_fails

def crossings_R3BP_by_mu(no_crossings, L123, ODE_R3BP, dir,
                                step, t_span, mu,
                                init_search=100, init_tol=1e-12,
                                refinement=100, newton_tol = 1e-15,
                                start_cond_tol = 10**-6):
    Li = [compute_Lagrange_pt(mu, L123), 0, 0, 0]
    Li.extend([1, 0, 0, 0,
                0, 1, 0, 0,
```

```
                0, 0, 1, 0,
                0, 0, 0, 1])  # initial conditions and identity matrix
    time_span = 0

    # compute the Jacobian matrix of the RTBP at Li.
    # The eigenvalues of this matrix are the frequencies of the periodic orbit

    A = variational_eq(t_span[0], Li, mu, 1)[4:20].reshape(4,4)
    eigenvalues, eigenvectors = np.linalg.eig(A)
    lambda_pos = eigenvalues[3].real
    lambda_neg = eigenvalues[2].real
    eigvec_pos = eigenvectors[:,3].real
    eigvec_neg = eigenvectors[:,2].real
    if lambda_pos < lambda_neg:
        print("Warning: eigenvalues are not ordered")
        temp = eigvec_pos
        eigvec_pos = eigvec_neg
        eigvec_neg = temp
    if dir == 1:
        if eigvec_pos[1] > 0:
            eigvec_pos = -eigvec_pos
            #eigvec_pos[0] = -eigvec_pos[0]
        v = eigvec_pos
    elif dir == -1:
        v = eigvec_neg
    else:
        raise ValueError("Direction must be 1 or -1")
    init_cond = Li[0:4] + v*start_cond_tol

    crossing_times, crossing_initials, mu_fails = \
        crossings_R3BP(no_crossings, ODE_R3BP, init_cond, dir, step, t_span,
                        mu, init_search, init_tol, refinement, newton_tol)
    return crossing_times, crossing_initials, mu_fails
```

Assignment 8:

```
import numpy as np
import sys
dir_string = 'C:/Users/rannu/OneDrive - NTNU/Desktop/VsPython/'+\
                'Spain/NMfDS/Assignments/'
sys.path.append(dir_string + 'Ass4')
from RTBP_definitions import r1, r2


# Define the variation equation as a function of time, x, mu and direction
def variational_eq(t, x, mu, dir):
    var_eq = np.zeros(20)
    # Defining these as variables so solve_ivp can handle them
    r1_val = r1(mu, x[0], x[1])
    r2_val = r2(mu, x[0], x[1])

    var_eq[0] = x[2]
    var_eq[1] = x[3]
    var_eq[2] = 2 * x[3] + x[0] - (1 - mu) * (x[0] - mu) / r1_val**3 \
                - mu * (x[0] - mu + 1) / r2_val**3
```

```
    var_eq[3] = -2 * x[2] + x[1] * (1 - (1 - mu) / r1_val**3 - mu / r2_val**3)

    Omegaxx = 1 - (1 - mu) / r1_val**3 \
            + 3 * (1 - mu) * (x[0] - mu)**2 / r1_val**5 - mu / r2_val**3 \
            + 3 * mu * (x[0] - mu + 1)**2 / r2_val**5
    Omegayy = 1 - (1 - mu) / r1_val**3 - mu / r2_val**3 \
            + (3 * (1 - mu) * x[1]**2) / r1_val**5 \
            + (3 * mu * x[1]**2) / r2_val**5
    Omegaxy = 3 * (1 - mu) * x[1] * (x[0] - mu) / r1_val**5 \
            + 3 * mu * x[1] * (x[0] - mu + 1) / r2_val**5

    for i in range(4, 12):
        var_eq[i] = x[i + 8]

    for i in range(12, 16):
        var_eq[i] = Omegaxx * x[i - 8] + Omegaxy * x[i - 4] + 2 * x[i + 4]

    for i in range(16, 20):
        var_eq[i] = Omegaxy * x[i - 12] + Omegayy * x[i - 8] - 2 * x[i - 4]

    # Flip the sign of the variation equation if we are looking at the
    # stable manifold
    if dir == -1:
        var_eq = -var_eq

    return var_eq
```

Assignment 7

```
import sys
import scipy as sp
import numpy as np
sys.path.append('C:/Users/rannu/OneDrive - NTNU/Desktop/VsPython/' +
                'Spain/NMfDS/Assignments/Ass4')
sys.path.append('C:/Users/rannu/OneDrive - NTNU/Desktop/VsPython/' +
                'Spain/NMfDS/Assignments/Ass6')
from RTBP_definitions import r1, r2, OMEGA, ODE_R3BP, \
                            Jacobi_first_integral
from custom_ODE_solver import ODE_solver

def poincare_map_solve_ivp_R3BP(ODE_R3BP, initial_conditions, dir,
                                step, t_span, mu, init_search=100,
                                init_tol=1e-12, refinement=100,
                                newton_tol = 1e-15):
    product = 1
    time = 0
    startPoint = np.array(initial_conditions)
    initial_conditions = initial_conditions
    failed_mu = -1

    while product >= 0 and abs(time) < abs(step*init_search):
        solution = ODE_solver(ODE_R3BP, startPoint, t_span[1], 1000,
                              t_min=t_span[0], tol = init_tol,
            hamiltonian=lambda X: Jacobi_first_integral(mu, X[0], X[1],
                                                        X[2], X[3]))
```

```
        Y = solution.y.T  # Transposing to match previous structure
        product = Y[1, 1] * Y[-1, 1]  # Check if x-axis is crossed
        startPoint = Y[-1, :]
        t_span = [t_span[0] + dir * step, t_span[1] + dir * step]
        time += step*dir
    # make an error if the x-axis is never crossed
    if abs(time) >= abs(step*init_search):
        raise ValueError("No␣crossing␣found,␣initial␣search␣failed" +
                          "\ntime:␣␣" + str(time) +
                         "\nproduct␣␣" + str(product) + "\nstep␣␣"
                         + str(step) +
                            "\ninit_search␣␣" + str(init_search))
    # Procedure to compute the exact time of the crossing
    for i in range(refinement):
        solution = ODE_solver(ODE_R3BP, initial_conditions, time, 1000,
                              tol = init_tol,
                              hamiltonian=lambda X: \
                                Jacobi_first_integral(mu, X[0], X[1],
                                                          X[2], X[3]))
        Y = solution.y.T
        # One iteration of Newton's method.
        difference = Y[-1, 1] / ODE_R3BP(0, Y[-1, :])[1]
        time -= difference

        if i == refinement - 1:
            # raise ValueError("No convergence, refinement failed" +
            #                  "\ntime:  " + str(time) +
            #                  "\ndifference  " + str(difference))
            #print("No convergence, refinement failed " + "mu: " + str(mu))
            failed_mu = mu
        if abs(difference) < newton_tol or abs(Y[-1, 1]) < newton_tol:
            # print("Convergence after", i, "iterations")
            break

    TimeDuration = time
    newInitial = Y[-1, :]
    return newInitial, TimeDuration, failed_mu
```

Assignment 6

```
import numpy as np
import scipy as sp

def ODE_solver(func, x0, t_max, eval_pts, tol=1e-12, t_min=0,
                method='DOP853', hamiltonian=0):
    t_eval = np.linspace(t_min, t_max, eval_pts)

    sol = sp.integrate.solve_ivp(
        func, [t_min, t_max], x0, method=method,
        t_eval=t_eval, atol=tol, rtol=tol)
    return sol
```

Assignment 4

```
import numpy as np
from scipy.optimize import fsolve
```

```
from RTBP_definitions import r1, r2, OMEGA, ODE_R3BP, \
                              Jacobi_first_integral

# Compute x coordinate of the Lagrange points L1, L2, L3
# mu is the mass ratio of the two bodies with mass.
def compute_L1(mu):
    """
    Compute the L1 Lagrange point for the given mass ratio mu.
    For all the mathematics, see L13.
    """
    def polynomial(x):
        return x**5 - (3 - mu)*x**4 + (3 - 2*mu)*x**3 - mu*x**2 + 2*mu*x - mu
    # Estimation for eps_0
    eps = (mu / (3 * (1- mu)))**(1/3) # See L13
    L1 = fsolve(polynomial, eps)
    return mu-1+L1[0] # See L13

def compute_L2(mu):
    """
    Compute the L2 Lagrange point for the given mass ratio mu.
    For all the mathematics, see L13.
    """
    def polynomial(x):
        return x**5 + (3 - mu)*x**4 + (3 - 2*mu)*x**3 - mu*x**2 - 2*mu*x - mu
    # Estimation for eps_0
    eps_0 = (mu / (3 * (1- mu)))**(1/3) # See L13
    L2 = fsolve(polynomial, eps_0)
    return mu-1-L2[0] # See L13

def compute_L3(mu):
    """
    Compute the L3 Lagrange point for the given mass ratio mu.
    For all the mathematics, see L13.
    """
    def polynomial(x):
        return x**5 + (2 + mu)*x**4 + (1 + 2*mu)*x**3 - (1 - mu)*x**2 - \
               2*(1 - mu)*x - (1 - mu)
    # Estimation for eps_0
    eps_0 = 1 - (7 / 12)*mu # See L13
    L3 = fsolve(polynomial, eps_0)
    return mu + L3[0] # See L13

# Wrapper function
# Compute the position of the Lagrange point L for the given mu.
def compute_Lagrange_pt(mu, L):
    if L == 1:
        return compute_L1(mu)
    elif L == 2:
        return compute_L2(mu)
    elif L == 3:
        return compute_L3(mu)
    else:
        raise ValueError("Lagrange point must be 1, 2 or 3")

def compute_jacobi_const_Li(mu, L):
```

```
    '''
    Compute the Jacobi constant for the Lagrange point 1, 2 or 3
    '''
    xLi = compute_Lagrange_pt(mu, L)
    # from RTBP_definitions.py:
    C = Jacobi_first_integral(mu, xLi, 0, 0, 0)
    return C
```

Assignment 4

```
import numpy as np
# This file is used to define the functions used in the restricted
# three body problem

# r1, r2, OMEGA, ODE_R3BP, Jacobi_first_integral

def r1(mu, x, y):
    return np.sqrt((x - mu)**2 + y**2)

def r2(mu, x, y):
    return np.sqrt((x-mu+1)**2 + y**2)

def OMEGA(mu, x, y):
    return 0.5 * (x**2 + y**2) + (1 - mu) / r1(mu,x,y) + mu / r2(mu, x, y) \
           + 0.5 * (1 - mu) * mu

def ODE_R3BP(t, mu, X):
    # ODEs of the restricted three body problem
    return [X[2], X[3], 2*X[3] + X[0] - (1 - mu)*(X[0] - mu) / \
            r1(mu,X[0],X[1])**3 - mu * (X[0] - mu + 1) / \
            r2(mu,X[0],X[1])**3, -2*X[2] + X[1] - (1 - mu) * X[1] \
            / r1(mu,X[0],X[1])**3 - mu * X[1] / r2(mu,X[0],X[1])**3]

def Jacobi_first_integral(mu, x, y, vx, vy):
    # Function to compute the Jacobi first integral
    # This should be constant for a given value of mu
    return 2*OMEGA(mu, x, y) - (vx**2 + vy**2)
```