Assignment 1

Pol Navarro Pérez

September 2024

Abstract

This study explores the dynamics of two discrete dynamical systems: the logistic map and the two-dimensional standard map. The logistic map is analyzed for various parameter values and initial conditions, illustrating the emergence of fixed points and periodic behavior. The two-dimensional standard map is investigated for multiple initial conditions, first with a fixed parameter and then as the parameter is varied. We identify periodic points using Newton's method and examine the impact of changing parameters on the system's stability and chaotic behavior. Our results include detailed plots demonstrating the transition from regular to chaotic dynamics in both systems.

1 Logistic Map

Simulate the logistic map $F(x) = ax(1-x)$ for (i) $a = 2.0, x_0 = 0.2$, and $a = 3.2, x_0 = 0.4$. (See plots below).

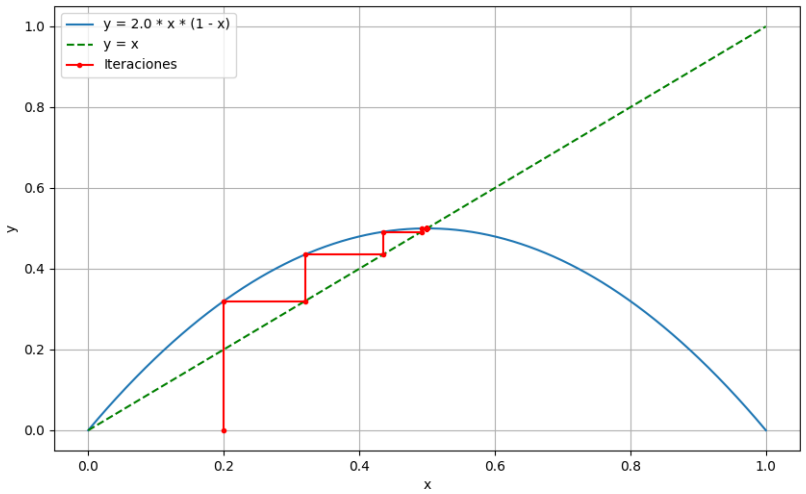


Figure 1: Logistic map for a = 2.0 and $x_0 = 0.2$.

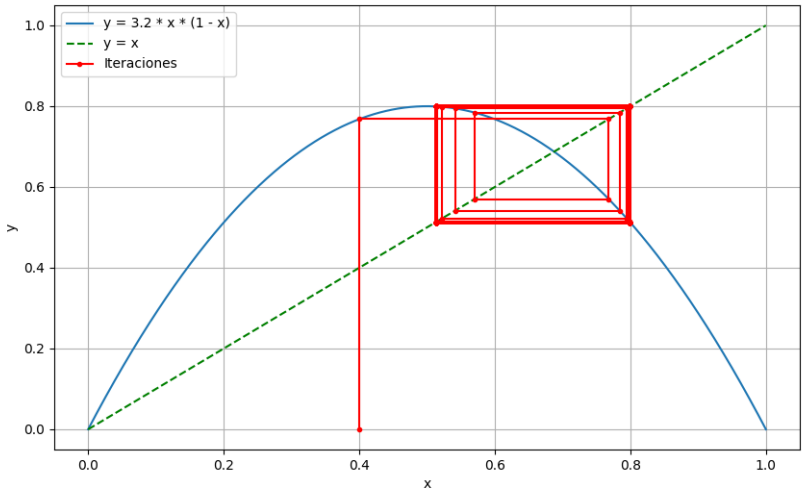


Figure 2: Logistic map for a = 3.2 and $x_0 = 0.4$.

As we can observe in 1 there is a fixed point around x = 0.5, at the top of the parabola. In contrast, there are two periodic points in 2 that alternate constantly.

2 2D Standard map

2.1 Standard map with initial conditions

The second part of the assignment involves a standard map defined by $F(x, y) = (x + a \cdot sin(x + y), x + y)$. We have considered a total of 500 iterations and 100 initial conditions $(x_0, 0)$, where $x_0 \in [-\pi, \pi]$.

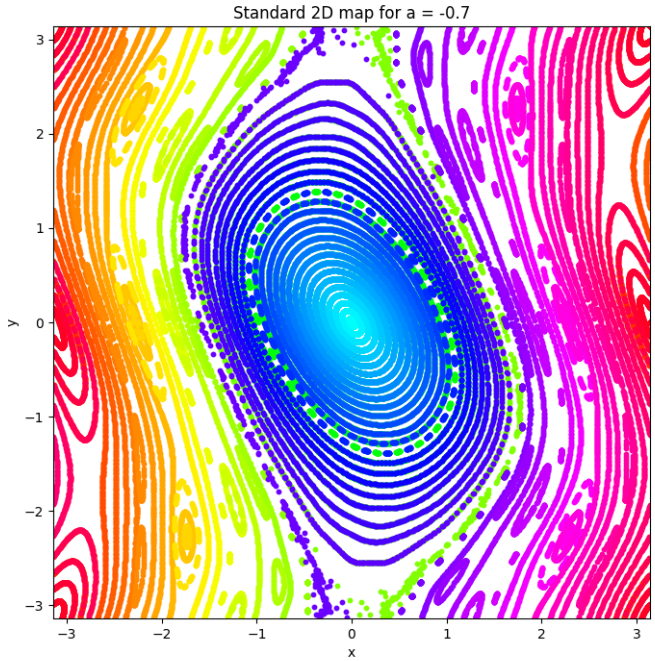


Figure 3: Standard 2D map with 500 iterations and 100 initial conditions for a = -0.7.

2.2 Periodic points

Now, we are asked to find the exact initial conditions for a 2-periodic point.
Analytically, we want for a 2-periodic point:

$$f\left(f\begin{pmatrix} x + a \cdot sin(x+y) \\ x+y \end{pmatrix}\right) = \begin{pmatrix} x \\ y \end{pmatrix} \tag{1}$$

This is satisfied by the point $(\pi, 0)$, where

$$f\left(f\begin{pmatrix} \pi + a \cdot sin(\pi+0) \\ \pi+0 \end{pmatrix}\right) = f\begin{pmatrix} \pi \\ \pi \end{pmatrix} = \begin{pmatrix} \pi \\ 0 \end{pmatrix}$$

The next step is to find an approximate initial condition for a 3-periodic point, or k-periodic points. The method used is a Newton on the function $f^k(x)$- x in order to find the roots.
The Newton method programmed takes two functions and a initial point x_0. These functions are *f*, the function for which we want to find the root, and its Jacobian matrix *Df*. From that point, we just have to update it iteratively, until we get a lower tolerance than the one passed as a paremeter, using the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

As the problem requires finding a 3-periodic point, or a k-periodic point if needed, we need by definition to minimize the funcion $f^3(x) - x = f(f(f(x))) - x$. As the code at the end of the paper shows, we have implemented two functions. A first function *minf*, which creates the function to minimize, depending on the order k of the k-periodic point, and the function *Derminf*, which returns the Jacobian matrix of the k-periodic point function.

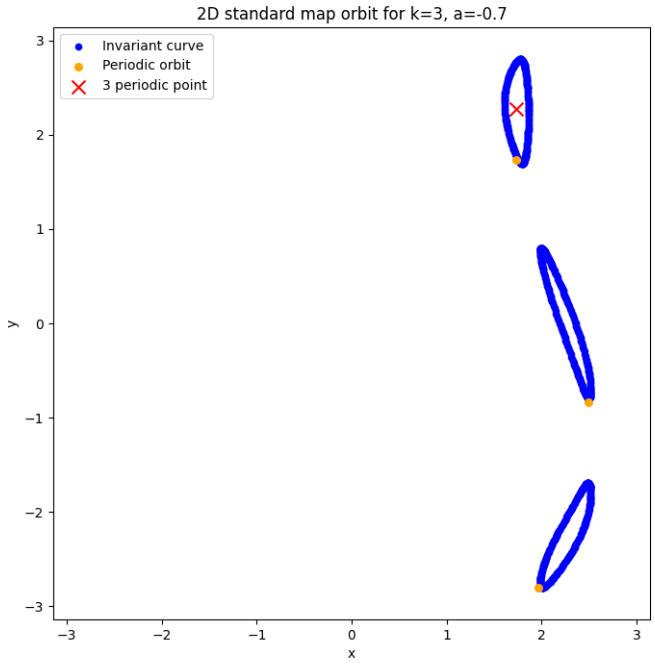


Figure 4: Associated orbit and invariant curve around a 3-periodic point.

2.3 Vertical invariant curves

In this section we are asked about programming a vertical invariant curve and plotting another around the origin. To do that we start at a point we want to study its behavior and then we just perturb the y value. At the origin, we do the same thing but take an arbitrary initial point for x, while setting y = 0, and we perturbate the x value.

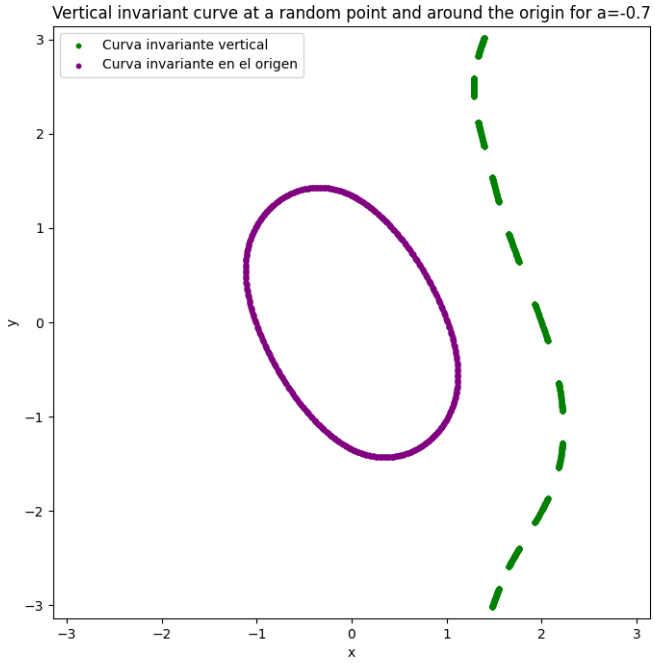
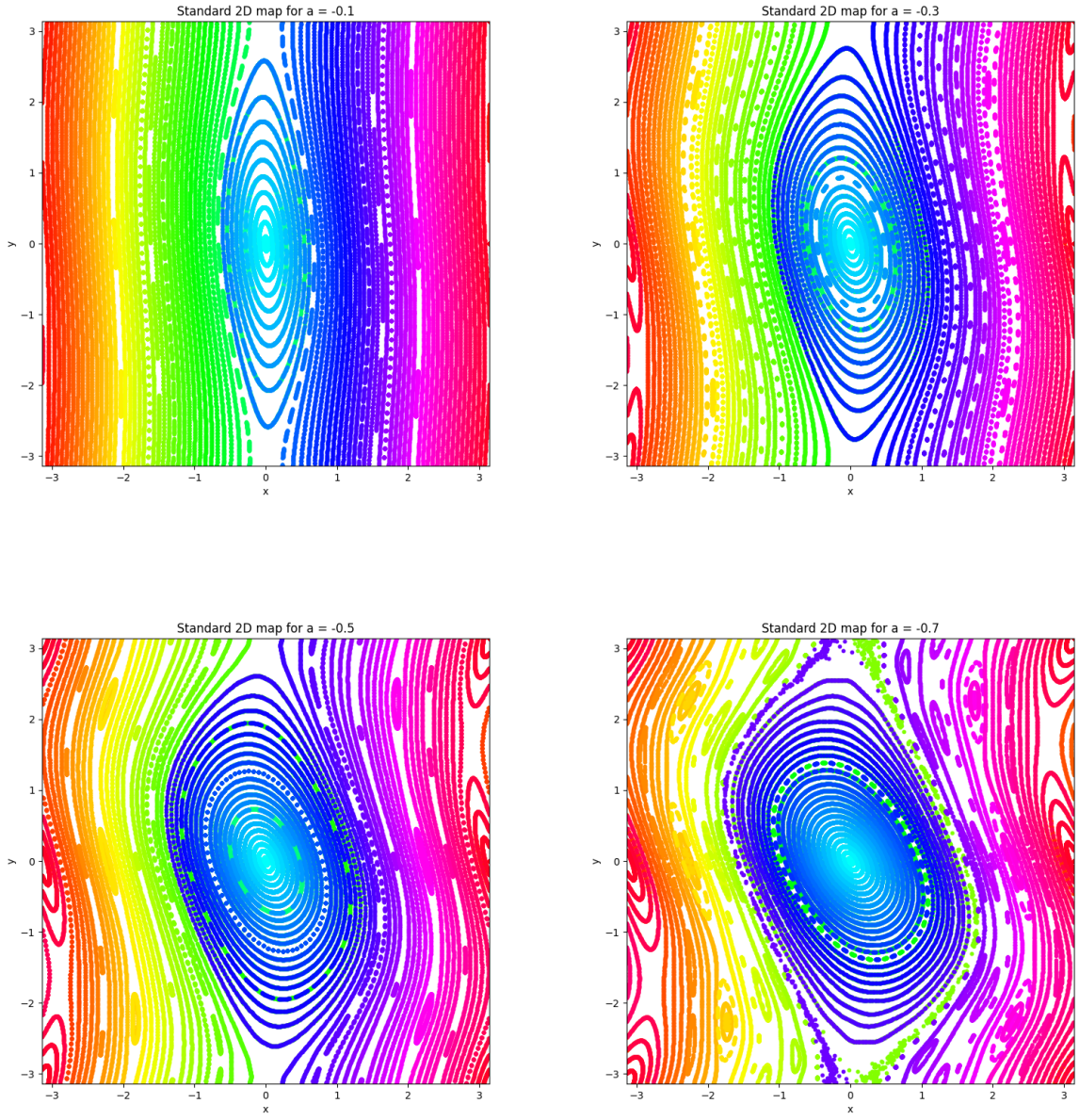


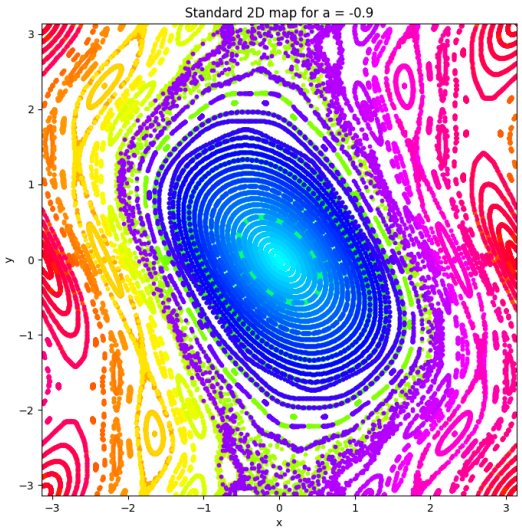
Figure 5: Vertical invariant curve at a random point and around the origin.

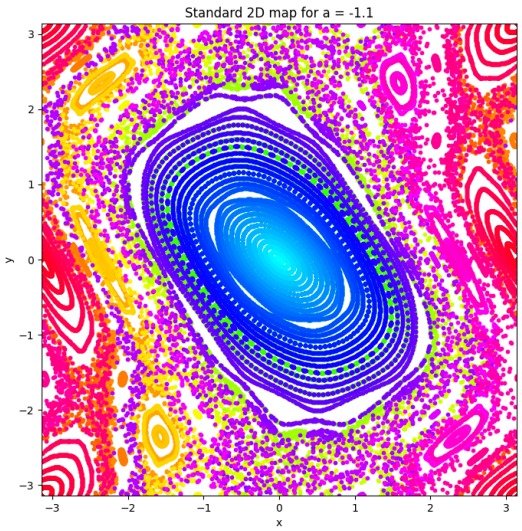
First, we know that the origin is a fixed point. However, if we move the initial condition, the system describes an invariant orbit around it, making it an attractor point. For a body near the origin, the system will keep it near the center, like a stable planetary orbit.
Studying now the 3-periodic point, we can see that if we take a point slightly displaced as initial condition, the systems remains around it tracing an invariant orbit.
Up to this point, we have mentioned about the orbits around periodic point or fixed points. Nevertheless, if we see the plot 3 there are some vertical invariant curves between these orbits of k-periodic points, as the one shown in 5.

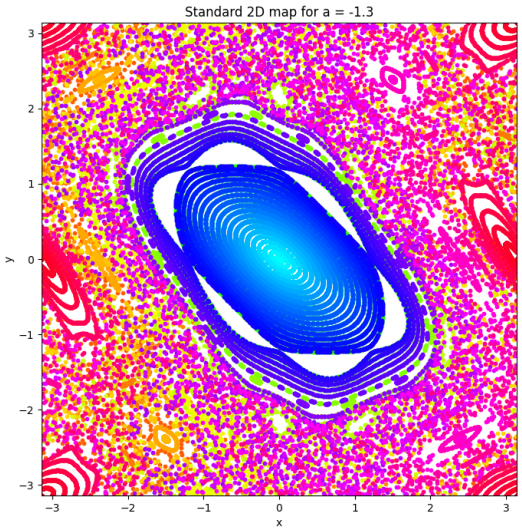
2.4 Simulation varying parameter a

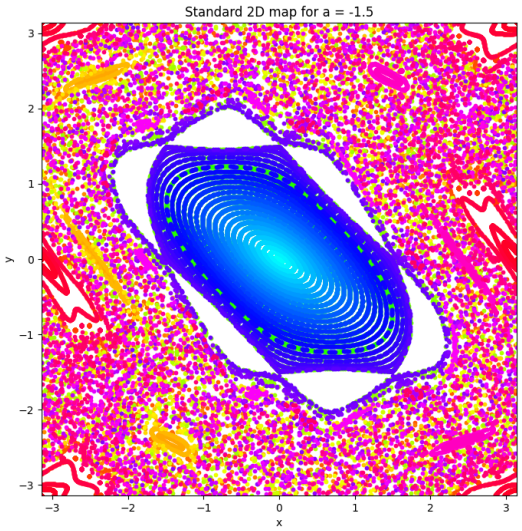
we have worked with a fixed parameter a = -0.7. However, we want to study the behavior of the 2D standard map as a function of the parameter a. The next subplot shows how the plot changes depending on this parameter.




Standard 2D map for a = -0.9
x
y


Standard 2D map for a = -1.1
x
y


Standard 2D map for a = -1.3
x
y


Standard 2D map for a = -1.5
x
y

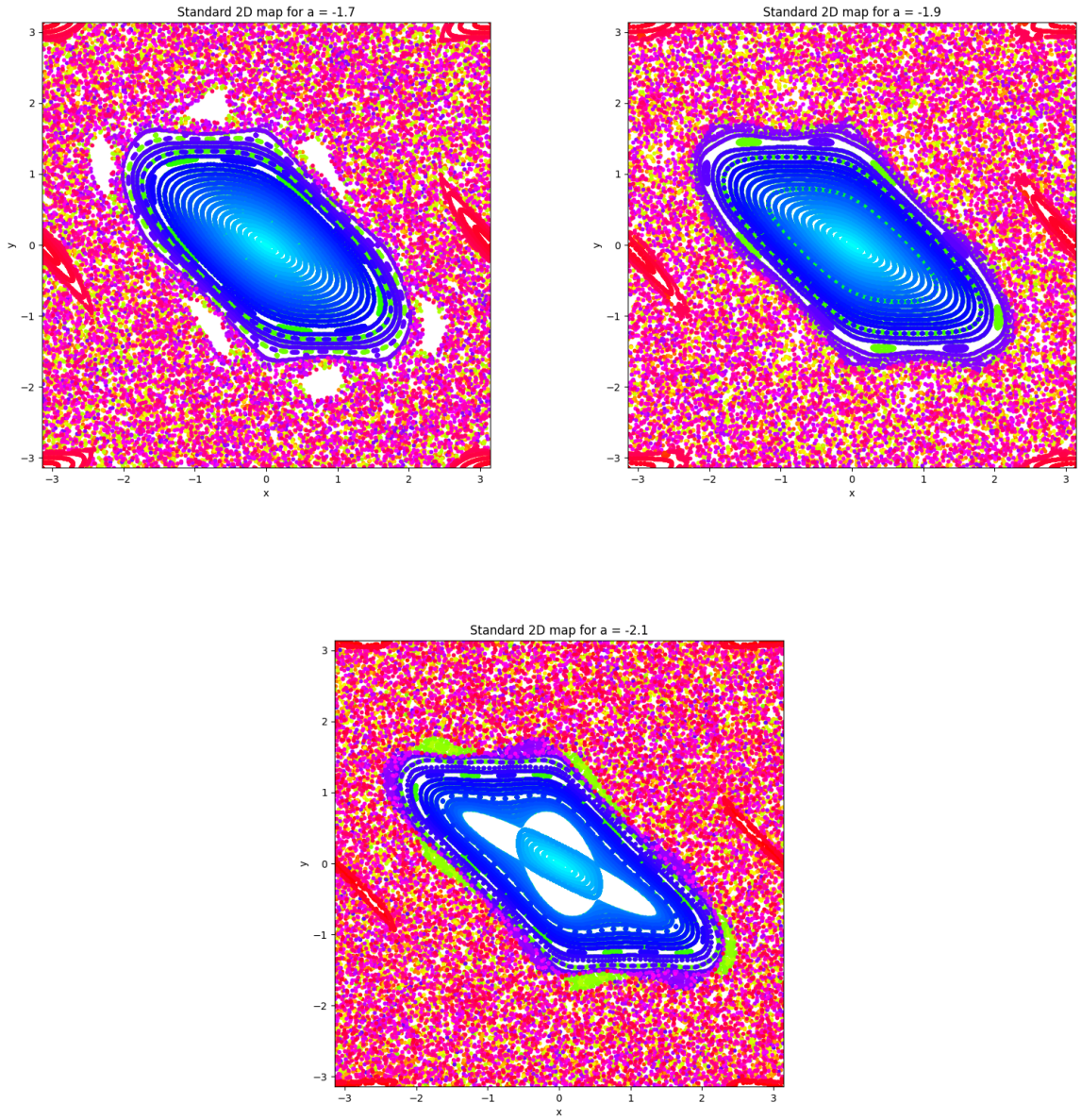


Figure 6: 2D Standard Map for multiple values of a.

As we can observe in 6, the plots are quite different if we change the value of the parameter a. For instance, for the lower value $a = -0.1$, there is a fixed point at the origin and then invariant orbits around it. Next to these orbits, there are vertical invariant curves, as seen previously.
As we progressively decrease the value of a, the vertical curves begin to bend and some periodic points appear, as it is clearly observable at $a = -0.7$.
From that point onward, higher order and more periodic points appear, making a visual effect of more chaos, as we can observe for $a = -1.1$. Another consequence is that the orbits around the origin become closer together and start to rotate. Eventually, for a = -2.1, we encounter complete chaos around the central orbits and a change on these orbits. In addition, there is is a gap between two kinds of central orbits.

CODE

```
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 21 17:22:08 2024

@author: polop
"""

#%%
# PART 1: LOGISTIC MAP
```

```
# valores de las iteraciones (cada iteraci n tiene dos puntos x0 y xf)

import numpy as np
import matplotlib.pyplot as plt

# Par metros
# a = 3.2
a = 2.0
# x0 = 0.4
x0 = 0.2

n = 100  # N mero de iteraciones

# Definir el mapa log stico
def logistic_map(x, a):
    return a * x * (1 - x)

# Par bola
x_parabola = np.linspace(0, 1, 500)
parabola = logistic_map(x_parabola, a)

# Inicializar el array para las iteraciones
iterations = np.zeros((2 * n, 2))
iterations[0, 0] = x0
iterations[0, 1] = 0

# Iterar para llenar el array
i = 0
x = x0
y = 0
while i < 2 * n - 1:
    iterations[i, 0] = x
    iterations[i, 1] = y
    y = logistic_map(x, a)
    i += 1
    iterations[i, 0] = x
    iterations[i, 1] = y
    x = y
    i += 1



# Graficar
plt.figure(figsize=(10, 6))

# Graficar la par bola
plt.plot(x_parabola, parabola, label=f'y = {a} * x * (1 - x)', linewidth=1.5)

# Graficar la recta x = y
plt.plot(x_parabola, x_parabola, label='y = x', linestyle='--', linewidth=1.5, color = "green")

# Graficar los puntos de iteraciones
plt.plot(iterations[:, 0], iterations[:, 1], label='Iteraciones', color='red', marker='o',
    markersize=3)

# A adir etiquetas y t tulo
plt.xlabel('x')
plt.ylabel('y')
# plt.title('Par bola y = a * x * (1 - x) con Iteraciones')
plt.legend()
plt.ylim
plt.show()

#%%

# PART 2: STANDARD MAP

import numpy as np
import matplotlib.pyplot as plt


a = -0.7
# N mero de condiciones iniciales
NP = 100
# N mero de iteraciones
iteraciones = 500

# Dimensiones bidimensionales (x,y) y tantos puntos como iteraciones para cada condici n inicial
xf = np.zeros((iteraciones, 2))
```

```
# Equiespaciadas las condiciones iniciales en el rango requerido
IC = np.linspace(-np.pi, np.pi, NP)

def f(x, a):
    # valor de la funci n F(x,y)
    F_xy = np.zeros(2)
    # valor x y lo envolvemos en el rango de 0 a 2pi
    F_xy[0] = np.mod(x[0] + a * np.sin(x[0] + x[1]), 2 * np.pi)
    F_xy[1] = np.mod(x[0] + x[1], 2 * np.pi)

    # pasamos los puntos a [-pi, pi]
    if F_xy[0] > np.pi and F_xy[1] > np.pi:
        F_xy[0] = -np.pi + np.mod(F_xy[0], np.pi)
        F_xy[1] = -np.pi + np.mod(F_xy[1], np.pi)

    if F_xy[0] > np.pi and F_xy[1] < np.pi:
        F_xy[0] = -np.pi + np.mod(F_xy[0], np.pi)

    if F_xy[0] < np.pi and F_xy[1] > np.pi:
        F_xy[1] = -np.pi + np.mod(F_xy[1], np.pi)

    return F_xy


# Crear un mapa de colores
cmap = plt.cm.get_cmap('hsv', NP)  # Utilizamos 'hsv' para obtener una amplia gama de colores
# Configurar el gr fico
plt.figure(figsize=(8, 8))
plt.xlim([-np.pi, np.pi])
plt.ylim([-np.pi, np.pi])
plt.title("Standard 2D map for a = -0.7")
plt.ylabel("y")
plt.xlabel("x")
# Bucle para todas las initial conditions que tenemos
for i in range(0, NP):
    x0 = np.array([IC[i], 0])
    xf[0, :] = f(x0, a)
    # Empiezo en 1 pq la condici n inicial es el xf[0,:]
    for j in range(1, iteraciones):
        xf[j, :] = f(xf[j-1, :], a)

    # Asignar un color diferente a cada condici n inicial usando cmap
    color = cmap(i)
    plt.scatter(xf[:, 0], xf[:, 1], s=10, color=color)

# # Marcar el 2-periodic point con una cruz negra
# plt.scatter(2.7967814286635737, 1.7432019392580074, color='black', marker='x', s=100, label="2-
    periodic point")

# # Marcar el 3-periodic point con una cruz marron
# plt.scatter(1.7380081367622715, 2.2725885852086543, color='brown', marker='x', s=100, label="3-
    periodic point")
# plt.legend()
plt.show()

#%%
# Find exact initial condition of a 2-periodic point
# Despu s de iterar dos veces obtenemos el mismo punto
# f(f(x0,a), a) = x0

#newton_raphson method, para encontrar raices
def newton_raphson(f, Df, x0, tolerance, a):
    xk = np.array(x0, dtype=float)  # Inicializaci n del punto
    current_tol = 5.0  # Inicializaci n de la tolerancia

    while current_tol > tolerance:
        # Calcular la correcci n dxk usando la derivada de f y el valor actual xk
        # t rmino -f(xk)/(f'(xk))
        dxk = np.linalg.solve(Df(xk, a), -f(xk, a))
        # nuevo xk, me acerco a la ra z
        xk = xk + dxk

        # Calcular la norma del ajuste en L2, estamos usando una tolerancia horizontal
        current_tol = np.linalg.norm(dxk, 2)

        #
        # Ponemos el nuevo punto entre 0 y 2pi
        xk[0] = np.mod(xk[0], 2 * np.pi)
```

```
        xk[1] = np.mod(xk[1], 2 * np.pi)

    return xk



# Definir el mapa est ndar (funci n f)
def f(x, a):
    f_xy = np.zeros(2)
    f_xy[0] = np.mod(x[0] + a * np.sin(x[0] + x[1]), 2 * np.pi)
    f_xy[1] = np.mod(x[0] + x[1], 2 * np.pi)

    # Ajustar los puntos de [0, 2*pi] a [-pi, pi]
    if f_xy[0] > np.pi:
        f_xy[0] -= 2 * np.pi
    if f_xy[1] > np.pi:
        f_xy[1] -= 2 * np.pi

    return f_xy

# Matriz Jacobiana, de las respectivas derivadas
def Df(x, a):
    Df_matrix = np.zeros((2, 2))
    Df_matrix[0, 0] = 1 + a * np.cos(x[0] + x[1])
    Df_matrix[0, 1] = a * np.cos(x[0] + x[1])
    Df_matrix[1, 0] = 1
    Df_matrix[1, 1] = 1
    return Df_matrix


# Defino la funci n a minimizar minf = f^k(x,a) - x para un k periodic point
# Funci n a minimizar
def minf(k, x, a):
    result = f(x, a)
    for _ in range(1, k):
        result = f(result, a)
    return result - x

# Jacobiano de la funci n a minimizar
def Derminf(k, x, a):
    jacobiano = Df(x, a)  # Inicia el Jacobiano con la primera derivada Df(x, a)
    x_copy = np.copy(x)    # Copia de x para no modificar el valor original
    for _ in range(1, k):
        x_copy = f(x_copy, a)  # Aplica f para obtener el siguiente valor de x
        jacobiano = np.dot(Df(x_copy, a), jacobiano)  # Calcula el Jacobiano iterativamente
    return jacobiano - np.eye(2)  # Resta la matriz identidad al final

#%%
# Par metros iniciales
a = -0.7
tolerancia = 1e-15
# Condici ninicial random, mirando la gr fica la alejo un poco del centro
x0 = np.array([1.0, 1.0])
# epsilon le he dado un valor muy peque o porque si no se me desviava mucho de lo esperado
epsilon = np.array([0.01, 0.01])
# k periodic point
k = int(input("Introduce el valor entero de k: "))


# Usar lambdas para no ejecutar las funciones inmediatamente
per_point = newton_raphson(lambda xk, a: minf(k, xk, a), lambda xk, a: Derminf(k, xk, a), x0,
    tolerancia, a)
orbit = np.zeros((k,2))

# Mostrar el punto peri dico encontrado
print(f'The {k} periodic point is located at ({per_point[0]}, {per_point[1]})')


# Plot the associated orbit. Plot an invariant curve around it

# Construir la  rbita  del movimiento en el punto peri dico despu s de las iteraciones
orbit[0, :] = per_point[0]
# Iteramos las k veces para volver al punto
for i in range(1, k):
    # valores de los puntos de la  rbita
    orbit[i, :] = f(orbit[i - 1, :], a)
```

```
# nos desplazamos un epsilon del k-periodic point para ver como es la curva invariante y estudiar el
    k-periodic point
x = per_point[0] + epsilon
# n (iteraciones) puntos, bidimensional
n = 1000
curve = np.zeros((n, 2))
# empezamos en el k periodic point
curve[0, :] = x
for i in range(1, n):
    curve[i, :] = f(curve[i - 1, :], a)

# Graficar la  rbita
plt.figure(figsize=(8, 8))
plt.scatter(curve[:,0], curve[:,1], s=20, color='blue', label='Invariant curve')
# la  rbita  tendr  k puntos
plt.scatter(orbit[:, 0], orbit[:, 1], s=30, color='orange', label='Periodic orbit')
plt.xlim([-np.pi, np.pi])
plt.ylim([-np.pi, np.pi])
plt.title(f"2D standard map orbit for k={k}, a={a}")
plt.xlabel("x")
plt.ylabel("y")
plt.scatter(per_point[0], per_point[1], color='red', marker='x', s=100, label=f'{k} periodic point')
plt.legend()

plt.show()

#%%
 # take a=-0.1,-0.3,-0.5,.....,-2.1
 # (10 different values of a), and for each a, obtain the
 # output data. Plot, as a film, the evolution of the dynamics
 # varying a, that is, plot 10 different plots.


a_list = np.arange(-0.1, -2.2, -0.2, dtype = float)
for el in a_list:
    # Crear un mapa de colores
    cmap = plt.cm.get_cmap('hsv', NP)  # Utilizamos 'hsv' para obtener una amplia gama de colores
    # Configurar el gr fico
    plt.figure(figsize=(8, 8))
    plt.xlim([-np.pi, np.pi])
    plt.ylim([-np.pi, np.pi])
    plt.title(f"Standard 2D map for a = {el:.1f}")
    plt.xlabel("x")
    plt.ylabel("y")
    # Bucle para todas las initial conditions que tenemos
    for i in range(NP):
        x0 = np.array([IC[i], 0])
        xf[0, :] = f(x0, el)
        for j in range(1, iteraciones):
            xf[j, :] = f(xf[j-1, :], el)

        # Asignar un color diferente a cada condici n inicial usando cmap
        color = cmap(i)
        plt.scatter(xf[:, 0], xf[:, 1], s=10, color=color)


    plt.show()
#%%
# 2D MAP with 2 and 3 periodic points


# Crear un mapa de colores
cmap = plt.cm.get_cmap('hsv', NP)  # Utilizamos 'hsv' para obtener una amplia gama de colores
# Configurar el gr fico
plt.figure(figsize=(8, 8))
plt.xlim([-np.pi, np.pi])
plt.ylim([-np.pi, np.pi])
plt.title("Standard 2D map for a = -0.7")
plt.ylabel("y")
plt.xlabel("x")

# Bucle para todas las initial conditions que tenemos
for i in range(NP):
    x0 = np.array([IC[i], 0])
    xf[0, :] = f(x0, a)
    for j in range(1, iteraciones):
        xf[j, :] = f(xf[j-1, :], a)

    # Asignar un color diferente a cada condici n inicial usando cmap
```

```
    color = cmap(i)
    plt.scatter(xf[:, 0], xf[:, 1], s=10, color="black")

# Marcar el 2-periodic point con una cruz negra
plt.scatter(2.7967814286635737, 1.7432019392580074, color='green', marker='x', s=100, label="2-
    periodic point")

# Marcar el 3-periodic point con una cruz marron
plt.scatter(1.7380081367622715, 2.2725885852086543, color='red', marker='x', s=100, label="3-
    periodic point")



plt.legend()
plt.show()

#%%

#  Plot a vertical invariant curve. Plot another one around the origin.

# Par metros
n = 1000  # N mero de puntos en la curva
epsilon = np.array([0.01, 0.0])  # Peque o desplazamiento para la curva vertical

# Curva invariante vertical
vertical_invariant_curve = np.zeros((n, 2))
# La curva invariante vertical empezamos en un punto que queremos estudiar y lo vamos perturbando
    con x constante, solo movimento la y
vertical_invariant_curve[0, :] = [2, 0 + epsilon[1]]
for i in range(1, n):
    vertical_invariant_curve[i, :] = f(vertical_invariant_curve[i - 1, :], a)

# Curva invariante en el origen
origin_curve = np.zeros((n, 2))
# La curva invariante en el origen empiezo en un punto x arbritario y = 0, y lo voy perturbando en x
origin_curve[0, :] = [1 + epsilon[0], 0]
for i in range(1, n):
    origin_curve[i, :] = f(origin_curve[i - 1, :], a)

# Graficar las curvas invariantes junto con las  rbitas
plt.figure(figsize=(8, 8))

plt.scatter(vertical_invariant_curve[:, 0], vertical_invariant_curve[:, 1], s=10, color='green',
    label='Curva invariante vertical')
plt.scatter(origin_curve[:, 0], origin_curve[:, 1], s=10, color='purple', label='Curva invariante en
    el origen')

plt.xlim([-np.pi, np.pi])
plt.ylim([-np.pi, np.pi])
plt.title(f"Vertical invariant curve at a random point and around the origin for a={a}")
plt.xlabel("x")
plt.ylabel("y")

plt.legend()
plt.show()
```

Assignment 3: Computation of a Poincare section and Poincaré map

Pol Navarro Pérez

September 2024

Abstract

In this assignment, we use Poincaré sections to observe the intersection points of a system with a given surface. Additionally, we analyze the behavior of various dynamical systems. We study the behavior of equilibrium points based on the eigenvalues of the Jacobian at those points. This is applied to the Lotka-Volterra model for a dynamical system representing the coexistence of two biological species, and we observe how different possible scenarios emerge.

1 Part A. Poioncaré section

In this first section we are going to implement the poincaré section to the harmonic oscillator problem, described by:

$$\begin{cases} x' = y \\ y' = -x \end{cases} \tag{1}$$

We are asked to compute the first crossing with the Poincaré section y = 0. Then we just have to implement the method for $n_{crossing} > 1$.
Once we have an initial condition (x_0, y_0) we can integrate the solution forward in time *idir = 1* or backward in time *idir = -1*.

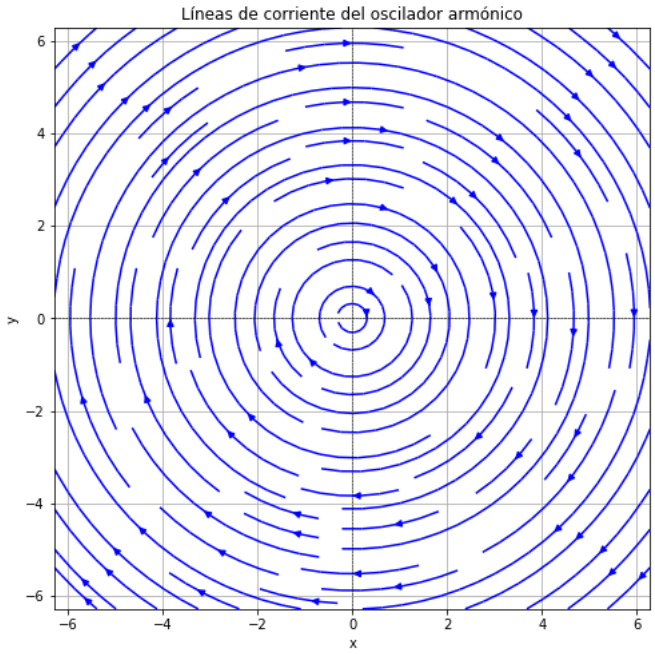


Figure 1: Streamlines of the simple harmonic oscillator system. The arrows indicate the flow of the system, representing the phase space dynamics of the oscillator with respect to position (x) and velocity (y). The system exhibits closed orbits, characteristic of periodic motion.

The figure 1 shows the systems we are working with. It has a 2π period. Therefore, if we work with an initial condition (1,0) the time for a number of crossings 2 should be the same as the period. Using the code attached, we obtain the following results:

- Forward: (x,y,t) = (0.99999999999996, 8.673617379884035e-19, 6.283185307179586)
- Backward (x,y,t) = (0.9999999999999607, 0.0000000000000, -6.283185307179547)

In order to see how good are our result we compare the results with the expected time:

- Forward: Difference $= |2\pi - t_{\text{Forward}}| = 0.0$
- Backward: Difference $= |2\pi - t_{\text{Backward}}| = 3.91 \times 10^{-14}$

Taking 16 digits we can clearly see how the time is considered the expected one.
In a nutshell, we first slightly perturb the initial point to avoid starting at $y = 0$. We then integrate the system until we detect a crossing of $y = 0$, which indicates that we have crossed the Poincaré section. The crossing is detected by a change of sign in the y-coordinate. After detecting the crossing, we apply the Newton-Raphson method, using time as the independent variable, to accurately determine the exact time at which the root of $g(x)$ (i.e., when $y = 0$) occurs. This process is repeated for the desired number of crossings.
Using now (0,1) as the initial point, thinking about our graphic, the first crossing should be around $\frac{2\pi}{4} = \frac{\pi}{2}$ and $\frac{3\pi}{2}$ for the second crossing. With the same computational approach than before the results for the second crossing are the following:

- Forward: (x,y,t) = (-0.9999999999999697, 0.0, 4.712388980384685)
- Backward (x,y,t) = (0.9999999999999699, 0.0, -4.712388980384653)

As we did before the differences with the expected time are:

- Forward: Difference $= \left|\frac{3\pi}{2} - t_{\text{Forward}}\right| = 4.44 \times 10^{-15}$
- Backward: Difference $= \left|\frac{3\pi}{2} - t_{\text{Backward}}\right| = 3.64 \times 10^{-14}$

2 Part B. Integration of a linear system.

In this section, we aim to implement and integrate a 2-dimensional linear system of differential equations with constant coefficients. The general form of such a system is given by:

$$x' = ax + by$$
$$y' = cx + dy$$

where a, b, c, and d are constants that define the behavior of the system. We will implement this system numerically and explore different parameter values to observe the resulting dynamics in the phase space.
We also compute the eigenvalues and eigenvectors of the Jacobian matrix at the point (0,0) and compute the stable and unstable manifold of the origin. In case of a periodic systems we are asked to mention which is the period.

$$J = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \tag{2}$$

2.0.1 Case 1 (Center): $a = 2$, $b = -5$, $c = 1$, $d = -2$

For this case, we have the system matrix:

$$A = \begin{pmatrix} 2 & -5 \\ 1 & -2 \end{pmatrix}$$

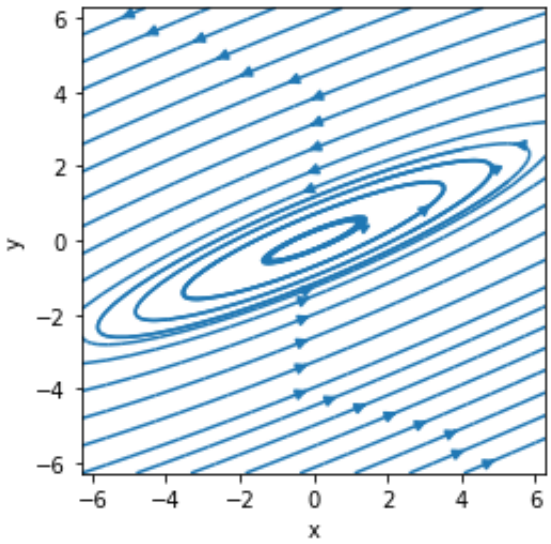


Figure 2: Linear system from case 1.

As we can observe in 2 it is a periodic systems that consist on spiral orbits that move away from the center.

If we compute the eigenvalues of the Jacobian matrix at (0,0) we get two values λ_1 and λ_2 with real part $-6,97 * 10^{-17}$. Therefore, we are dealing with assimptotically stable center. This kind of systems is a 2π periodic systems of orbits around a center.

2.0.2 Case 2 (Unstable center): $a = 3$, $b = -2$, $c = 4$, $d = -1$

For this case, the system matrix is:

$$A = \begin{pmatrix} 3 & -2 \\ 4 & -1 \end{pmatrix}$$

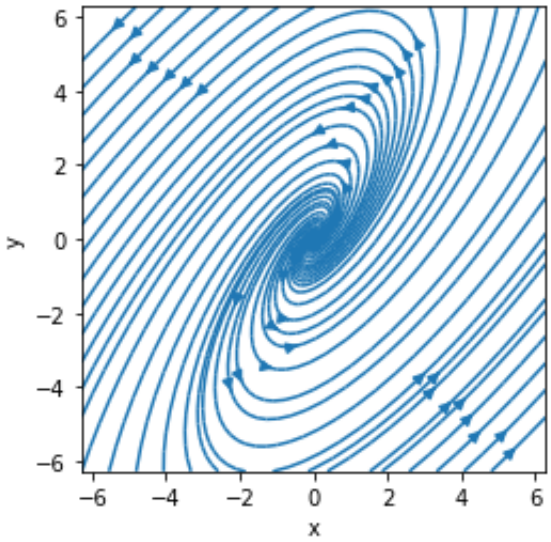


Figure 3: Linear system from case 2.

As we can clearly observe in 3 this system represents spirals that move away from an unstable center.
As we did in case 2, if we compute the eigenvalues we obtain two eigenvalues with real part with value 1, positive then. This means our equilibrium point is unstable, as we can observe in 3. The flows tends to move away points from the center.

2.0.3 Case 3(Saddle): $a = -1$, $b = 0$, $c = 3$, $d = 2$

Here, we analyze the saddle system with the matrix:

$$A = \begin{pmatrix} -1 & 0 \\ 3 & 2 \end{pmatrix}$$

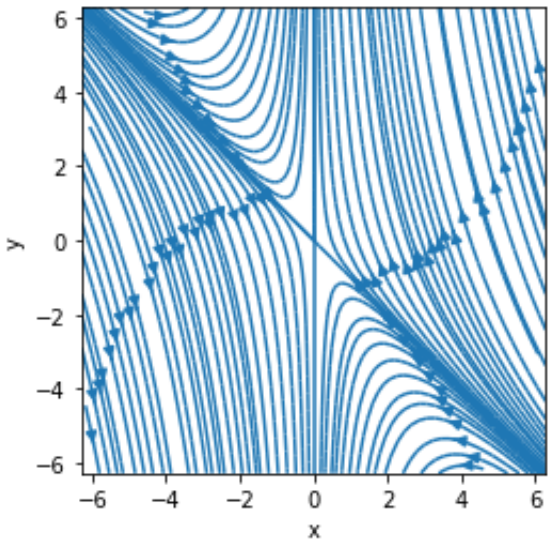


Figure 4: Linear system from case 3.

In the case 3, as seen in 5 we are dealing with a saddle. This type of equilibrium point is characterized by both stable and unstable manifolds, where the system exhibits opposing behaviors depending on the initial conditions.
In this case the eigenvalues are $\lambda_1 = 2$ and $\lambda_2 = -1$, as seen in theory sessions this means the saddle case. The corresponding eigenvectors are $v_1 = (0, 1)$ and $v_2 = (0.707, -0.707)$. With the unstable manifold associated with the positive eigenvector and the stable with the negative one. We want to get them using the computational approximation seen in theory classes. Using $p \pm s\vec{v_i}$ as a initial point, and integrating forward in time or backward in time depending on the manifold we want to obtain.
Hands on work, we use a key parameter $s = 10^{-6}$ and then we get 4 initial points, two for each manifold, and then two more depending if we add or rest the term with $s\vec{V_i}$. The results are the following:

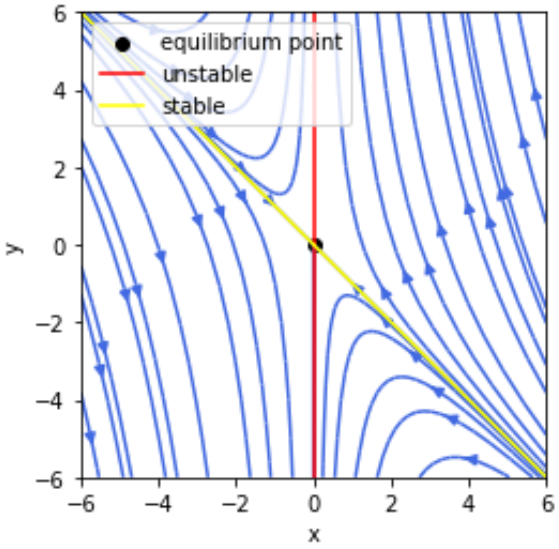


Figure 5: Manifolds for the saddle case.

2.1 PART C. Pendulum.

This section is about ploting the phase portrait of a pendulum and their heteroclinic orbits. So, the start point is the system that describes the pendulum, described by:

$$\begin{cases} \frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0 \\ \frac{d\theta}{dt} = \omega \end{cases} \tag{3}$$

The equilibrium points, calculated by $\dot{\theta} = 0$ and $\dot{\omega} = 0$, are $(0, 0)$ and $(k\pi, 0)$, where $k \in \mathbb{Z}$. To analyze the behaviour of these equilibrium points we need to stud the Jacobian matrix evaluated at this points.

$$J(\theta, \omega) = \begin{pmatrix} 0 & 1 \\ -\frac{g}{L}\cos(\theta) & 0 \end{pmatrix}$$

The Jacobian of the point (0,0) is:

$$J(0, 0) = \begin{pmatrix} 0 & 1 \\ -\frac{g}{L} & 0 \end{pmatrix}$$

As seen in theory sessions, the equilibrium point is a center.
For the second equilibrium point $(\theta, \omega) = (\pm\pi, 0)$, the Jacobian matrix is:

$$J(\pm\pi, 0) = \begin{pmatrix} 0 & 1 \\ \frac{g}{L} & 0 \end{pmatrix}$$

Its eigenvalues are $\lambda = \pm\frac{g}{L}$. As seen in theory this represents a saddle.

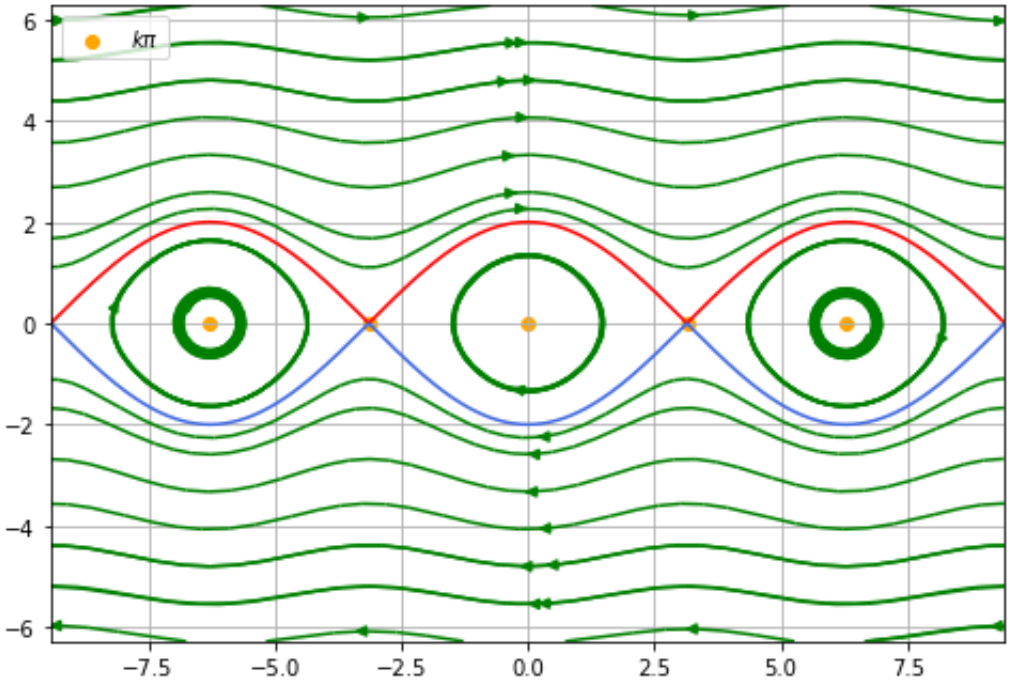


Figure 6: Phase portrait of a pendulum and heteroclinic orbits.

To compute the heteroclinic orbits we need to work with the energy of the system, described by:

$$H(\theta, \omega) = \frac{ml^2\omega^2}{2} + mgl(1 - \cos(\theta))$$

where m is the mass, l is the length of the pendulum, ω is the angular velocity, g is the gravitational constant, and θ is the angular displacement.

Our parameters, in order to make the systems as simpler as possible, are (m, g, l) = (1, 1, 1). Therefore, our H to reach $\theta = \pi$ is H = 2. This is the heteroclinic orbits that connects the two saddle points $(-\pi, 0)$ and $(\pi, 0)$. Remark that we are working with a systems where there exist the first integral.

2.2 PART D. Lotka-Volterra

The last section of this assignment is about the Lotka-Volterra model of competition between two speces, here rabbits and sheeps.
Our system is described by:

$$\begin{cases} x' = x(3 - x - 2y) \\ y' = y(2 - x - y) \end{cases} \tag{4}$$

It is clear that the axis x = 0 and y = 0 are invariant lines of the systems. It means that any trajectory that starts on one of these axes will remain on that axis for all future time.
By calculation $\dot{x} = 0$ and $\dot{y} = 0$, we can obtain the equilibrium points (0,0), (0,2), (3,0) and (1,1). As we did in previous sections, in order to classify them we need to study de eigenvalues of their respective jacobian matrix.

$$J(x, y) = \begin{pmatrix} \frac{\partial}{\partial x}[x(3 - x - 2y)] & \frac{\partial}{\partial y}[x(3 - x - 2y)] \\ \frac{\partial}{\partial x}[y(2 - x - y)] & \frac{\partial}{\partial y}[y(2 - x - y)] \end{pmatrix} = \begin{pmatrix} 3 - 2x - 2y & -2x \\ -y & 2 - x - 2y \end{pmatrix} \tag{5}$$

- **Equilibrium point (0,0)**:
  Jacobian at $(0, 0)$:

  $$J(0, 0) = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

  Eigenvalues: $\lambda_1 = 3$, $\lambda_2 = 2$.
  Since both eigenvalues are positive, $(0, 0)$ is an assymptotically unstable node.

- **Equilibrium point (0,2)**:
  Jacobian at $(0, 2)$:

  $$J(0, 2) = \begin{pmatrix} -1 & 0 \\ 0 & -2 \end{pmatrix}$$

  Eigenvalues: $\lambda_1 = -1$, $\lambda_2 = -2$.
  Theses are the eigenvalues of an attractor, assymptotically stable.

- **Equilibrium point (3,0)**:
  Jacobian at $(3, 0)$:

$$J(3,0) = \begin{pmatrix} -3 & 0 \\ 0 & -1 \end{pmatrix}$$

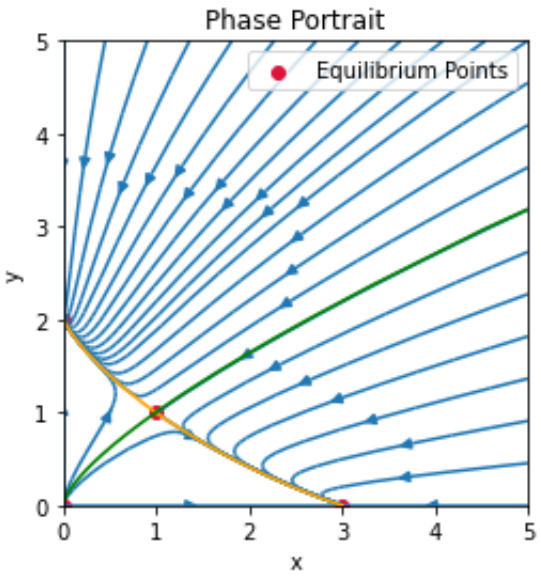
  Eigenvalues: $\lambda_1 = -3$, $\lambda_2 = -1$.
  Since both eigenvalues are negative, $(3, 0)$ is an asymptotically stable node.

- **Equilibrium point (1,1)**:
  Jacobian at $(1, 1)$:

$$J(1,1) = \begin{pmatrix} -\sqrt{2} & -2 \\ -1 & -\sqrt{2} \end{pmatrix}$$

  Eigenvalues: $\lambda_1 = -1 + \sqrt{2}$, $\lambda_2 = -1 - \sqrt{2}$.
  This corresponds to a saddle.

This is going to be clearly observable at the representation of the phase portrait.



There infinitely many heteroclinic orbits

Figure 7: Phase portrait of the Lotka-Volterra system. We have also represented the two manifolds corresponding to the saddle point.

According to the types of equilibrium points there are no homoclinical orbits, in which we return to the equilibrium point when time tends to infinity.
There are some interesting possible scenarios to comment:

- **Extinction of both species** occurs at the equilibrium point $(0, 0)$. This point is unstable, indicating that if both species are driven to near extinction, small changes in the environment or populations will cause one or both species to grow again. Thus, complete extinction of both species is unlikely in the long run.
- **Survival of one species while the other goes extinct** is represented by the points $(0, 2)$ and $(3, 0)$. At these points, one species maintains a stable population while the other is extinct. The points $(3, 0)$ and $(2, 0)$ are stable, meaning that once the second species goes extinct, the first species will maintain its population without further fluctuations.
- **Coexistence of both species** occurs at the point $(1, 1)$. This equilibrium represents a state where both species are present, but the system exhibits cyclical behavior rather than a fixed population size. The populations of both species fluctuate over time, reflecting typical predator-prey or competitive interactions where growth in one population leads to growth or decline in the other.

In summary, this system demonstrates that interactions between species can lead to various long-term outcomes: extinction, survival of one species, or coexistence with population cycles. The absence of homoclinic orbits in the phase portrait indicates that the system does not return to equilibrium after perturbations.

CODE

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  2 15:11:21 2024

@author: polop
"""
# PART A)



#%%

#%%

# PART A)

from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


#POINCAR  SECTIONA AND POINCAR  MAP

# PARTE A
# Poincar  section for y = 0

# The harmonic oscillator function
def oscilador_harmonico(t, pos):
    dx, dy = pos[1], -pos[0]  # Definir las derivadas
    return np.array([dx, dy])

# In case if we need it
def newton_raphson(f, Df, x0, tol, a):
    xk = np.array(x0, dtype=float)  # Inicializaci n del punto
    tolk = 10.0  # Inicializaci n de la tolerancia

    while tolk > tol:
        # Calcular la correcci n dxk usando la derivada de f y el valor actual xk
        # t rmino -f(xk)/(f'(xk))
        dxk = np.linalg.solve(Df(xk, a), -f(xk, a))
        # nuevo xk, me acerco a la ra z
        xk = xk + dxk

        # Calcular la norma del ajuste en L2, estamos usando una tolerancia horizontal
        tolk = np.linalg.norm(dxk, 2)

        #
        # Ponemos el nuevo punto entre 0 y 2pi
        xk[0] = np.mod(xk[0], 2 * np.pi)
        xk[1] = np.mod(xk[1], 2 * np.pi)

    return xk

# As we are working on y = 0, then x' = 0, the corresponding g(x) is defined by:
def g(pos):
    return pos[1]
def grad(g):
    dg = np.array([0,1])
    return dg

# Implementation of the crossing with the section with n_crossing = 1

lim = 2 * np.pi  # Ajuste a 2 pi

# Crear una malla de puntos usando meshgrid
x = np.linspace(-lim, lim, 100)  # 100 puntos en el eje x
y = np.linspace(-lim, lim, 100)  # 100 puntos en el eje y
X0, Y0 = np.meshgrid(x, y)  # Generar la malla

# Vector de las velocidades, paso como argumento todos los puntos de mi cuadr cula
u, v = oscilador_harmonico(0, [X0, Y0])

# Lineas de corriente de nuestro sistema, que forma tiene
plt.figure(figsize=(8, 8))
plt.streamplot(X0, Y0, u, v, color='blue')
plt.title('L neas de corriente del oscilador arm nico')
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.axhline(0, color='black', linewidth=0.5, ls='--')
plt.axvline(0, color='black', linewidth=0.5, ls='--')
plt.grid()
plt.show()
#%%
def Poincar_Map(initial_point, ncrossings, idir, function_to_evaluate):

    crossings = []
    results = []
    absolute_t = 0



    # The process repeats as much as many ncrossing we have
    for j in range(ncrossings):
        # We first integrate until we cross y=0 (backwards or forwards)
        t = 1


        t_periodic = [0, idir*2*np.pi]
        # Remark that for t = 0 we are on y = 0. This is not a good initial point, we need to
            perturbate the system
        perturbation_time = 1E-14
        # the perturbation time goes backward if idir == -1 and forward if idi == +1
        sol = solve_ivp(function_to_evaluate, t_periodic, initial_point, t_eval=np.array([idir *
            perturbation_time]), rtol=3e-14, atol=1e-14)
        # solve_ivp returns a OdeResult with the times t, the y (positions for each time)
        # for the positions the first array [0] is the x-coordinate, [1] y-coordinate. Then we want
            the first point as the initial point

        # Definition of the new initial time, the same as sol.t (this las varaible returnes as an np
            .array)
        absolute_t += perturbation_time

        # Coordinates of the new initial point
        x = sol.y[0][0]
        y = sol.y[1][0]


        # Now we start from an initial point and we know our systems is 2pi periodic
        # Points between the range (0, 2pi) we are gonna calculare the function
        iterations = 500
        teval = np.linspace(0, idir * 2 * np.pi, iterations)
        # Resolvemos para 500 puntos del periodo 2pi, hacia adelante o hacia atr s
        # rtol and atol, value parameter of the tolerance
        sol = solve_ivp(function_to_evaluate, t_periodic, y0=[x, y], t_eval=teval, rtol=3e-14, atol
            =1e-14)
        # Now we want to evaluate when the solution changes sign, that means we have crossed the y =
             0
        t_ini = 0
        crossed = False
        for i in range(iterations):
            point = sol.y[1][i]
            if idir == 1:

                if (y*point)<0:
                    crossed = True
                    # we have a change of sign, y = 0 crossed
                    x_before, y_before = sol.y[0][i-1], sol.y[1][i-1]
                    x_after, y_after = sol.y[0][i], sol.y[1][i]
                    t_before, t_after = sol.t[i-1], sol.t[i]
                    crossx, crossy, t_ini = x_before, y_before, t_before
                    break
            if idir == -1:

                if (y*point)<0:
                    crossed = True
                    # we have a change of sign, y = 0 crossed
                    x_before, y_before = sol.y[0][i-1], sol.y[1][i-1]
                    x_after, y_after = sol.y[0][i], sol.y[1][i]
                    t_before, t_after = sol.t[i-1], sol.t[i]
                    crossx, crossy, t_ini = x_after, y_after, t_after
                    break

        crossings.append([crossx, crossy])
        # Now we apply Newton method from t_before, not the same as initial time, to compure tm+1 =
            tm -G(tm)/G'(tm)
```

```
        tol = 10e-14
        belongs_to = False
        while not belongs_to:
            # we are doing this for each number of crossing times

            sol = solve_ivp(function_to_evaluate, [0,2*np.pi], y0=crossings[j], t_eval = np.array([t
                ]),rtol=3e-14, atol=1e-14)
            t1 = t - (g(sol.y)/ (np.dot(grad(sol.y), function_to_evaluate(0, sol.y))))[0]
            t1 = t1 % (2 * np.pi)
            if np.abs(t - t1) < tol:
                belongs_to = True

            t = t1

        final = solve_ivp(function_to_evaluate, [0,2*np.pi], y0 = crossings[j], t_eval=np.array([t])
            ,rtol=3e-14, atol=1e-14)
        # final point
        x = final.y[0][0]
        y = final.y[1][0]
        initial_point = x,y
        absolute_t += t + t_ini


        results.append([x,y, absolute_t])
    # Nos devuelve una lista con los puntos de corte y el tiempo en el que lo cruza
    return results
initial = [1,0]
# Poincar_Map(initial_point, ncrossings, idir, function_to_evaluate):
print("Forward: ")
result_forward = (Poincar_Map(initial, 2, 1, oscilador_harmonico))
print(result_forward)
time_second_crossing = result_forward[1][2]
print(f"time forward = {time_second_crossing}")
# Calculamos la diferencia con 2pi
difference = np.abs(2 * np.pi - time_second_crossing)

# Imprimir con 16 d gitos de precisi n
print(f"difference with 2pi = {difference:.16f}")

print("Backward: ")
result_backward = (Poincar_Map(initial, 2, -1, oscilador_harmonico))
print(result_backward)
time_second_crossing_back = result_backward[1][2]
print(f"time backward = {time_second_crossing_back}")
# Calculamos la diferencia con 2pi
difference_back = np.abs(-2 * np.pi - time_second_crossing_back)

# Imprimir con 16 d gitos de precisi n
print(f"difference with 2pi = {difference_back:.16f}")
#%%
# Now we want to use a initial point (0,1)

initial = [0,1]
# Poincar_Map(initial_point, ncrossings, idir, function_to_evaluate):
print("Forward: ")
result_forward = (Poincar_Map(initial, 2, 1, oscilador_harmonico))
print(result_forward)
time_second_crossing = result_forward[1][2]
print(f"time forward = {time_second_crossing}")
# Calculamos la diferencia con 2pi
difference = np.abs(3/2*np.pi - time_second_crossing)

# Imprimir con 16 d gitos de precisi n
# print(f"difference with 2pi = {difference:.16f}")

print("Backward: ")
result_backward = (Poincar_Map(initial, 2, -1, oscilador_harmonico))
print(result_backward)
time_second_crossing_back = result_backward[1][2]
print(f"time backward = {time_second_crossing_back}")
# Calculamos la diferencia con 2pi
difference_back = np.abs(-3/2*np.pi- time_second_crossing_back)

print(difference_back)
print(difference)

# # Imprimir con 16 d gitos de precisi n
# print(f"difference with 2pi = {difference_back:.16f}")
```

```
#%%
# PART B
def linear_system(x, y, a, b, c, d):
    result =  np.array([a*x+b*y, c*x+d*y])
    return result

# We are asked to plot the phase space for different parameters a,b,c and d

#%%
# CASE 1:
import numpy as np
import matplotlib.pyplot as plt

# Par metros del sistema
a, b, c, d = 2, -5, 1, -2

# Generaci n de los valores de los ejes con np.linspace
lim = 2 * np.pi
x = np.linspace(-lim, lim, 500)
y = np.linspace(-lim, lim, 500)

# Crear las mallas
X0, X1 = np.meshgrid(x, y)

# Vector field del sistema
u, v = linear_system(X0, X1, a, b, c, d)

# Graficar el campo vectorial
fig = plt.axes()
fig.streamplot(X0, X1, u, v, density=0.6, broken_streamlines=False)
fig.set_aspect('equal', 'box')
plt.ylabel("y")
plt.xlabel("x")
plt.show()

#%%
# CASE 2:
import numpy as np
import matplotlib.pyplot as plt

# Par metros del sistema
a, b, c, d = 3,-2,4, -1

# Generaci n de los valores de los ejes con np.linspace
lim = 2 * np.pi
x = np.linspace(-lim, lim, 500)
y = np.linspace(-lim, lim, 500)

# Crear las mallas
X0, X1 = np.meshgrid(x, y)

# Vector field del sistema
u, v = linear_system(X0, X1, a, b, c, d)

# Graficar el campo vectorial
fig = plt.axes()
fig.streamplot(X0, X1, u, v, density=0.6, broken_streamlines=False)
fig.set_aspect('equal', 'box')
plt.ylabel("y")
plt.xlabel("x")
plt.show()
#%%
# CASE 3:
import numpy as np
import matplotlib.pyplot as plt

# Par metros del sistema
a, b, c, d = -1, 0, 3, 2

# Generaci n de los valores de los ejes con np.linspace
lim = 2 * np.pi
x = np.linspace(-lim, lim, 500)
y = np.linspace(-lim, lim, 500)

# Crear las mallas
X0, X1 = np.meshgrid(x, y)

# Vector field del sistema
u, v = linear_system(X0, X1, a, b, c, d)
```

```
# Graficar el campo vectorial
fig = plt.axes()
fig.streamplot(X0, X1, u, v, density=0.9, broken_streamlines=False)
fig.set_aspect('equal', 'box')
plt.ylabel("y")
plt.xlabel("x")
plt.show()



#%%
# The next part is to compute the eigenvalue and eigenvectors of the Jacobian matrix at (0,0)
# We are also asked to compute the stable and unstable manifold of the center


matrix = np.array([[a, b], [c, d]])
print(np.linalg.eig(matrix))
a, b, c, d = -1, 0, 3, 2
# Our system
def linear_system2(t, x):
    a, b, c, d = -1, 0, 3, 2
    return linear_system(x[0], x[1], a, b, c, d)

# MANIFOLDS
X = [1, 1]
X[1], X[0] = np.mgrid[-6:6:200j, -6:6:200j]
# flow of our phase space
u, v = linear_system(X[0], X[1], a, b, c, d)
fig = plt.axes()
fig.streamplot(X[0], X[1], u, v, density=0.4, broken_streamlines=False, color = "royalblue")
fig.set_aspect('equal', 'box')

fig.scatter(0, 0, label = "equilibrium point", color = "black")
s= 10e-7
# Compute and plot Wu+
x0 = np.array([0, 0]) + s*np.linalg.eig(matrix)[1][:, 0]
sol = solve_ivp(linear_system2, [0, 100], x0, t_eval=np.linspace(0, 10, 200), rtol=3e-14, atol=1e
    -14)
fig.plot(sol.y[0], sol.y[1], color='red', label = "unstable")
# Compute and plot Wu
x0 = np.array([0, 0]) - s*np.linalg.eig(matrix)[1][:, 0]
sol = solve_ivp(linear_system2, [0, 100], x0, t_eval=np.linspace(0, 10, 200), rtol=3e-14, atol=1e
    -14)
fig.plot(sol.y[0], sol.y[1], color='red')
# Compute and plot Ws+
x0 = np.array([0, 0]) + s*np.linalg.eig(matrix)[1][:, 1]
sol = solve_ivp(linear_system2, t_span=(0, -20), y0=x0, t_eval=np.linspace(0, -20, 200), rtol=3e-14,
     atol=1e-14)
fig.plot(sol.y[0], sol.y[1], color='yellow', label = "stable")
# Compute and plot Ws
x0 = np.array([0, 0]) - s*np.linalg.eig(matrix)[1][:, 1]
sol = solve_ivp(linear_system2, t_span=(0, -20), y0=x0, t_eval=np.linspace(0, -20, 200), rtol=3e-14,
     atol=1e-14)
fig.plot(sol.y[0], sol.y[1], color='yellow')

plt.xlim(-6, 6)
plt.ylim(-6, 6)
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc = "upper left")

plt.show()



#%%
# PART C:PENDULUM
# We are asked to plot the phase portrait of a pendulum, the ORBITS and DIRECTIONS
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
# arbritary parameters, the simplest ones
m = 1
g = 1
l = 1
def pendulum(x):
    # x[0] es el  ngulo  theta (posici n angular)
    theta = x[0]
```

```
    # x[1] es la velocidad angular omega
    omega = x[1]

    # dtheta_dt es la derivada de theta con respecto al tiempo, es decir, la velocidad angular (
        omega)
    dtheta_dt = omega

    # domega_dt es la derivada de omega con respecto al tiempo, es decir, la aceleraci n angular
    # Se calcula usando la ecuaci n del movimiento del p ndulo: -g/l * sin(theta)
    domega_dt = -(g / l) * np.sin(theta)

    # Devolver las derivadas como un array, d/dt theta y d/dt omega
    return dtheta_dt, domega_dt



# Par metros
lim = 2 * np.pi
initial_point = [0, 0]
X = initial_point

# all of initial points we are gonna use
X[1], X[0] = np.mgrid[-lim:lim:100j, -2*lim:2*lim:100j]

# derivatives for omega and theta
u, v = pendulum(X)


fig, ax = plt.subplots(figsize=(8, 6))

# we want the phase portrait and their diretions
ax.streamplot(X[0], X[1], u, v, density=0.6, broken_streamlines=False, color="green")


ax.set_aspect('equal', 'box')

x_vals = [0, np.pi, -np.pi, 2*np.pi, -2*np.pi]
y_vals = [0, 0, 0, 0, 0]


ax.scatter(x_vals, y_vals, color='orange', label=r'$k\pi$')


ax.legend(loc='upper left')

ax.set_xlim([-3/2*lim, 3/2*lim])
ax.set_ylim([-lim, lim])
def omega_integral(theta, H):
    omega_p, omega_n =  np.sqrt(2 * (H - m*g*l*(1 - np.cos(theta))) / (m*l**2)), -np.sqrt(2 * (H - m
        *g*l*(1 - np.cos(theta))) / (m*l**2))
    return omega_p, omega_n

# H = T+V
x = np.linspace(-4*np.pi, 4*np.pi, 5000)
# Energy to achieve theta = -pi, calculating with our parameter H = 2 = 2mgl
H = 2
y_p, y_n = omega_integral(x, H)
plt.plot(x,y_p, label = "positive", color = "red")
plt.plot(x,y_n, label = "negative", color = "royalblue")


# Mostrar el gr fico
plt.grid(True)




#%%
# PART D
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Tama o del plot
size = 5
```

```
# Crear una malla de puntos en un rango determinado
x_values, y_values = np.meshgrid(np.linspace(0, size, 100), np.linspace(0, size, 100))

# Definir el sistema de ecuaciones Lotka-Volterra
def lotka_volterra(t, z):
    x, y = z
    dxdt = x * (3 - x - 2 * y)
    dydt = y * (2 - x - y)
    return [dxdt, dydt]

# Evaluar el sistema para cada punto de la malla
def calcular_vector_field(x_values, y_values):
    u = np.zeros_like(x_values)
    v = np.zeros_like(y_values)

    for i in range(x_values.shape[0]):
        for j in range(x_values.shape[1]):
            u[i, j], v[i, j] = lotka_volterra(0, [x_values[i, j], y_values[i, j]])

    return u, v

# Calcular el campo vectorial en la malla
u, v = calcular_vector_field(x_values, y_values)

# Crear la figura y trazar el flujo usando streamplot
fig, ax = plt.subplots()
ax.streamplot(x_values, y_values, u, v, density=0.4, broken_streamlines=False)
ax.set_aspect('equal')

# Equilibrium points at the plot
equilibrium_points = np.array([[0, 0], [0, 2], [3, 0], [1, 1]])
scatter = ax.scatter(equilibrium_points[:, 0], equilibrium_points[:, 1], color = "red", label='
    Equilibrium␣Points')

# Agregar etiquetas para los ejes
ax.set_xlabel('x')
ax.set_ylabel('y')

plt.title('Phase␣Portrait', fontsize=12)

# Configurar l mites de los ejes para que vayan de 0 a 5
plt.xlim(0, 5)
plt.ylim(0, 5)

# A adir una leyenda que explique que los puntos rojos son los puntos de equilibrio


# Definir el Jacobiano
def jacobian_matrix(x):
    return np.array([[3 - 2*x[0] - 2*x[1], -2*x[0]], [-x[1], 2 - x[0] - 2*x[1]]])

# MANIFOLDS

# Wu ( rbita  inestable) - positivo
x0 = np.array([1, 1]) + 1E-7 * np.linalg.eig(jacobian_matrix(np.array([1, 1])))[1][:, 0]
sol = solve_ivp(lotka_volterra, [0, 100], x0, t_eval=np.linspace(0, 100, 200))
ax.plot(sol.y[0], sol.y[1], c='orange', label="Unstable␣Manifold")

# Wu ( rbita  inestable) - negativo
x0 = np.array([1, 1]) - 1E-7 * np.linalg.eig(jacobian_matrix(np.array([1, 1])))[1][:, 0]
sol = solve_ivp(lotka_volterra, [0, 100], x0, t_eval=np.linspace(0, 100, 200))
ax.plot(sol.y[0], sol.y[1], c='orange')

# Ws ( rbita  estable) - positivo
x0 = np.array([1, 1]) + 1E-7 * np.linalg.eig(jacobian_matrix(np.array([1, 1])))[1][:, 1]
sol = solve_ivp(lotka_volterra, t_span=(0, -20), y0=x0, t_eval=np.linspace(0, -10, 200))
ax.plot(sol.y[0], sol.y[1], c='green', label="Stable␣Manifold")

# Ws ( rbita  estable) - negativo
x0 = np.array([1, 1]) - 1E-7 * np.linalg.eig(jacobian_matrix(np.array([1, 1])))[1][:, 1]
sol = solve_ivp(lotka_volterra, t_span=(0, -20), y0=x0, t_eval=np.linspace(0, -10, 200))
ax.plot(sol.y[0], sol.y[1], c='green')
plt.legend(loc='upper␣right')
# Mostrar la gr fica
plt.show()
```

Parametrization of a $W^{or/var}(s)$ up to 3rd order including

Pendalum $x'' + \sin x = 0$ or eq $\begin{cases} x_1' = x_2 \\ x_2' = -\sin x_1 \end{cases}$ in $\mathbb{R}^2$

eq point is $(\pi, 0) = z_0$ $z' = G(z)$

$$DG(z_0) = \begin{pmatrix} 0 & 1 \\ -\cos(x_1) & 0 \\ \| & \\ +1 & \end{pmatrix}$$ with eigenvalues $\begin{cases} \lambda_1 = +1 \to \exists\ W^u(z_0) \\ \lambda_2 = -1 \to \exists\ W^s(z_0) \end{cases}$

Goal We want a parametrization $W^{or/var}(s)$ up to 3rd order including

eigenvector v_2

$$P = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \Rightarrow P^{-1} DG(z_0) P = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

↑ eigenvector v_1

$z = z_0 + Pz \to \boxed{z' = F(z)}$ $z = 0$ eq point here

Take $W(s)$ and $f(s)$ as an expansion and then imposing the variational equation s.t $\boxed{F(W(s)) = DW(s) f(s)}$

Then plot in a window $[0, 2\pi] \times [-2.5, 2.5]$ in (x_1, x_2) with two plots. Graph style and normal form style

Change of variables

New variables

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} \pi \\ 0 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \pi \\ 0 \end{pmatrix} + \begin{pmatrix} x+y \\ x-y \end{pmatrix} = \begin{pmatrix} x+y+\pi \\ x-y \end{pmatrix}$$

$-\sin\pi = 0$

We want new variables in terms of the old ones

$$(z - z_0) = P z \to z = P^{-1}(z - z_0) = + \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} z_1 - \pi \\ z_2 \; 0 \end{pmatrix} =$$

(small $\vec{z} = \begin{pmatrix} x \\ y \end{pmatrix}$)

$$= \frac{1}{2}\begin{pmatrix} z_1 - \pi + z_2 \\ z_1 - \pi - z_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$$

$$* \; P^{-1} = \frac{1}{|P|} Adj(A^t) = \frac{1}{-2}\begin{pmatrix} -1 & -1 \\ -1 & +1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Where z' will be $F(z) = P^{-1} G(z_0 + Pz)$

Now lets compute $F(z)$:

$$F(z)\begin{cases} x' = \frac{1}{2}[z_1' + z_2'] = \frac{1}{2}[x_2 - \sin x_1] = \frac{1}{2}[(x-y) + \sin(x+y+\pi)] \\ y' = \frac{1}{2}(z_1' - z_2') = \frac{1}{2}[x_2 + \sin x_1] = \frac{1}{2}[(x-y) + \sin(x+y+\pi)] \end{cases}$$

For $(x, y) = (0,0) \to x' = 0$ ✓ New eq point
$y' = 0$ ✓

We are asked to compute the parametrization of

$$\omega(s) = \sum_{k=1}^{3} \omega_k(s) = \begin{bmatrix} s_1 \\ \sin s \\ 0 \\ 0 \end{bmatrix} + \sum_{k=2}^{3} \omega_k(s)$$

...dd we are going to need $F(z) = \begin{bmatrix} \lambda_1 z_1 \\ \vdots \\ \lambda_n z_n \end{bmatrix} + \sum_{k=2} F_k(z)$

Df(0) z

$$[Dw(s) f(s)]_k = \sum_{e=0}^{k} Dw_{e,1} \cdot f_{k-e} = Dw_1 \cdot f_3 + Dw_3 f_2 + Dw_2 f_2$$

$w_1 = \begin{pmatrix} s \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ $Dw_1 = \begin{pmatrix} I \\ 0 \end{pmatrix}$

$f_1 = \begin{pmatrix} \lambda \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ $f(s) = \lambda_1 s + \sum_{k=2} f_k(z)$ ← only one s, |m|=s

$F\begin{pmatrix} s \\ 0 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} s + \sin(s) \\ s - \sin(s) \end{pmatrix} = \frac{1}{2}\begin{pmatrix} s + s - \frac{s^3}{3!} + \frac{s^5}{5!} + O(s^7) \\ s - s + \dots \end{pmatrix}$ ← Taylor expansion sin(s)

Thus, $[F(w_1(s)]_2 = 0$ and $[F(w_1(s)]_3 = \frac{1}{2}\begin{pmatrix} -1/6 \\ 1/6 \end{pmatrix}$

The left hand of the IE is then:

- For order 2

$[F(w(s))]_2 = [F(w_1(s))]_2 + DF(0) w_2 = \begin{pmatrix} w_2^1 \\ -w_2^2 \end{pmatrix}$ ← First component and second component of second order

and the right hand side for $k=2$

$[Dw(s) f(s)]_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} f_2 + 2 w_2 \cdot f_1 = \begin{pmatrix} f_2 + 2w_2^2 \\ 0 + 2w_2^2 \end{pmatrix}$

$w_1 = \begin{pmatrix} s \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ $Dw_1 = \begin{pmatrix} I \\ 0 \end{pmatrix}$

$f_1 = \lambda_2$ 1dimensional

Wow, the same for order 3

LHS: $[F(w(s))]_3 = \underbrace{[F(w_1(s))]_3}_{\text{Computed before}} + DF(0) w_3 = \begin{pmatrix} -\frac{1}{12} \\ +\frac{1}{12} \end{pmatrix} + \begin{pmatrix} w_3^1 \\ -w_3^2 \end{pmatrix}$

And the RHS is:

$$[Dw(s)f(s)]_3 = \begin{pmatrix}1\\0\end{pmatrix}\cdot f_3 + 3w_3\cdot \cancel{f_2} + 2w_2\cdot f_2 =$$

$$= \left[\begin{pmatrix} 3w_3^1 + 2w_2^2\cdot f_2 + f_3 \\ 3w_3^2 + 2w_2^2\cdot f_2 \end{pmatrix}\right]_3$$

Remark that $f: \mathbb{R}^1 \to \mathbb{R}^1$. $F: \mathbb{R}^2 \to \mathbb{R}^2$ and $w: \mathbb{R}^1 \to \mathbb{R}^2$

We have to use both sides of the invariance equation using the both styles known.

Graph style: $w_{k,m}^i = 0$ ← Because f^1

$k=2 \to \begin{pmatrix}0\\-w_2^2\end{pmatrix} = \begin{pmatrix} f_2 + 2\cancel{w_2}^0 \\ 2w_2^2 \end{pmatrix} \to \boxed{f_2 = 0}$ ← In this case $w_{2,m}^1 = 0$

$-w_2^2 = 2w_2^2 \to \boxed{w_2^2 = 0}$

$k=3 \to \begin{pmatrix} -\frac{1}{12} + \cancel{w_3}^0 \\ +\frac{1}{12} + w_3^2 \end{pmatrix} = \begin{pmatrix} f_3 \\ 3w_3^2 \end{pmatrix} \to \boxed{f_3 = -\frac{1}{12}}$

$\boxed{w_3^2 = \frac{1}{48}}$

$w_3^1 = 0$

And we add the order 1 terms

$$w^{\text{or var}}(s) = \begin{pmatrix} s \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ s^3/48 \end{pmatrix} + O(s^4)$$

Now using the normal form style

$f_k = 0 \quad \forall k \geq 2$.

same as before, order 2 and 3. f_k is now 0

for order 2 → LHS = RHS → $\begin{pmatrix} \omega_2^2 \\ -\omega_2^2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2\omega_2^2 \end{pmatrix}$

We get $\boxed{\omega_2^i = 0}$ for $i = 1, 2$

order 3 ($k=3$) $\begin{pmatrix} -\frac{1}{12} + \omega_3^1 \\ +\frac{1}{12} - \omega_3^2 \end{pmatrix} = \begin{pmatrix} 3\omega_3^1 \\ 3\omega_3^2 \end{pmatrix}$

$\boxed{\omega_3^1 = -\frac{1}{24}}$

$\boxed{\omega_3^2 = +\frac{1}{48}}$

$$\omega(s) = \begin{pmatrix} s \\ 0 \end{pmatrix} + \begin{pmatrix} -\frac{s^3}{24} \\ +\frac{s^3}{48} \end{pmatrix} + O(s^4)$$

Eventually we can compute the $\omega^{orvar}(s)$

For Graph Style: $\omega^{orvar}(s) = \underbrace{\begin{pmatrix} \pi \\ 0 \end{pmatrix}}_{X_0} + P \cdot \omega(s) =$

$$= \begin{pmatrix} \pi \\ 0 \end{pmatrix} + \begin{pmatrix} s + \frac{s^3}{48} \\ s - \frac{s^3}{48} \end{pmatrix} + O(s^4)$$

For normal form style: $\omega^{orvar}(s) = \underbrace{\begin{pmatrix} \pi \\ 0 \end{pmatrix}}_{X_0} + \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \omega(s) =$

$$= \begin{pmatrix} \pi \\ 0 \end{pmatrix} + \begin{pmatrix} s - \frac{s^3}{48} \\ s - \frac{s^3}{16} \end{pmatrix} + O(s^4)$$

Using a one dimensional parameter we have obtained the corresponding two dimensional manifold. Now we can program the code in order to compare the expansion up to third order with the expected plot

Parametrization of pendulum manifolds

Pol Navarro Pérez

October 30, 2024

Abstract

In this class assignment we are asked to compute the parametrization of an unstable manifold for a pendulum system, up to third order. We are working with the equilibrium point (π, 0) and our goal is to obtain $W^{orvar}(s)$ up to k = 3.

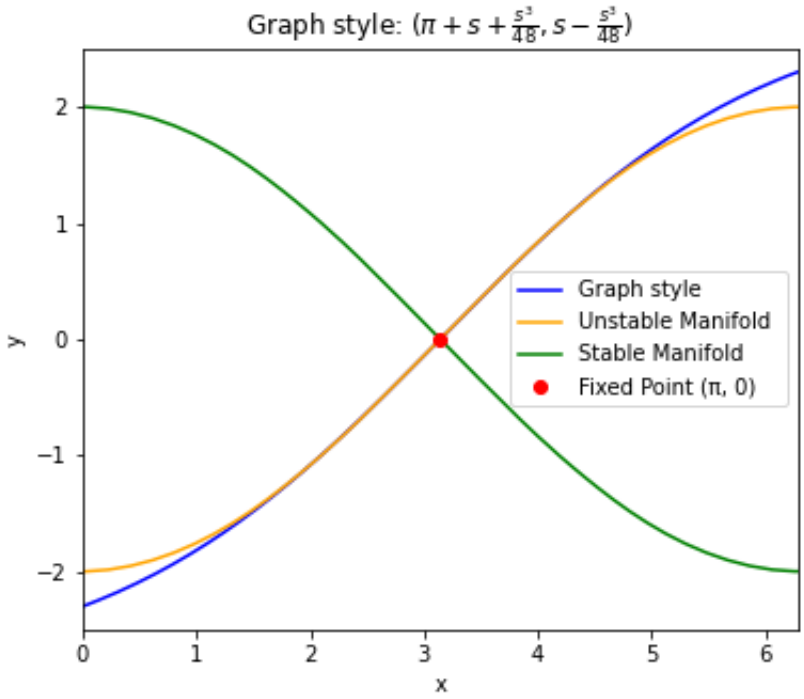


Figure 1: Graph style of the unstable manifold expansion up to order 3 compared to the expected manifold.

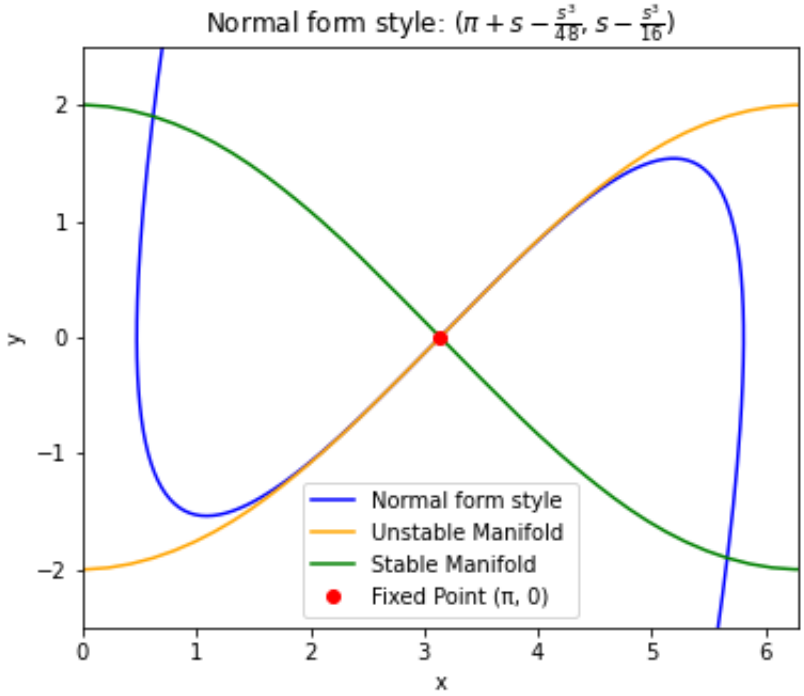


Figure 2: Normal form style of the unstable manifold expansion up to order 3 compared to the expected manifold.

As seen in 1 and 2 the approximations are quite good. The one obtained with the graph style is better suited to the one expected as we move away from the equilibrium point.

1 CODE

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 28 18:43:32 2024

@author: polop
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# Sistema de ecuaciones del p ndulo
def pendulum_system(t, y):
    x1, x2 = y
    return [x2, -np.sin(x1)]

# Matriz jacobiana en el punto (pi, 0)
def jacobian_matrix(point):
    return np.array([[0, 1], [1, 0]])

# Par metros iniciales
point = np.array([np.pi, 0])

# Direcciones de vectores propios (inestable y estable)
eigvals, eigvecs = np.linalg.eig(jacobian_matrix(point))
unstable_dir = eigvecs[:, 0]  # Vector propio correspondiente a    = 1
stable_dir = eigvecs[:, 1]    # Vector propio correspondiente a    = -1

# Integraci n para el manifold inestable (positivo y negativo)
# parametro epsilon (el que llam bamos s antes)
epsilon = 1E-7
x0_unstable_pos = point + epsilon * unstable_dir
x0_unstable_neg = point - epsilon * unstable_dir
sol_unstable_pos = solve_ivp(pendulum_system, [0, 20], x0_unstable_pos, t_eval=np.linspace(0, 20,
    200))
sol_unstable_neg = solve_ivp(pendulum_system, [0, 20], x0_unstable_neg, t_eval=np.linspace(0, 20,
    200))

# Integraci n para el manifold estable (positivo y negativo)
x0_stable_pos = point + epsilon * stable_dir
x0_stable_neg = point - epsilon * stable_dir
```

```
sol_stable_pos = solve_ivp(pendulum_system, [0, -20], x0_stable_pos, t_eval=np.linspace(0, -20, 200)
    )
sol_stable_neg = solve_ivp(pendulum_system, [0, -20], x0_stable_neg, t_eval=np.linspace(0, -20, 200)
    )

# Graficar los manifolds
fig, ax = plt.subplots()
ax.plot(sol_unstable_pos.y[0], sol_unstable_pos.y[1], c='orange')
ax.plot(sol_unstable_neg.y[0], sol_unstable_neg.y[1], c='orange', label="Unstable Manifold ")
ax.plot(sol_stable_pos.y[0], sol_stable_pos.y[1], c='green')
ax.plot(sol_stable_neg.y[0], sol_stable_neg.y[1], c='green', label="Stable Manifold")

# Punto de inter s
ax.plot(point[0], point[1], 'ro', label="Fixed Point (  ,0)")

# Ajustar los l mites de los ejes
ax.set_xlim(0, 2 * np.pi)
ax.set_ylim(-2.5, 2.5)

plt.legend(loc='upper right')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("Stable and Unstable Manifolds of the Pendulum System at $(\pi, 0)$")
plt.grid()
plt.show()

#%%
# Ahora comparo la soluci n anal tica con la parametrizaci n hasta tercer orden obtenida
import numpy as np
import matplotlib.pyplot as plt

# Definici n del rango para el par metro s
s_values = np.linspace(-10, 10, 500)

# Definici n de los manifolds parametrizados
manifold_1_x = np.pi + s_values + (s_values**3) / 48
manifold_1_y = s_values - (s_values**3) / 48

manifold_2_x = np.pi + s_values - (s_values**3) / 48
manifold_2_y = s_values - (s_values**3) / 16

# Graficar el primer manifold en una figura separada
plt.figure(figsize=(6, 5))
plt.plot(manifold_1_x, manifold_1_y, label='Graph style', color='blue')
plt.xlim(0, 2 * np.pi)
plt.ylim(-2.5, 2.5)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Graph style: $(\pi + s + \\frac{s^3}{48}, s - \\frac{s^3}{48})$')
plt.plot(sol_unstable_pos.y[0], sol_unstable_pos.y[1], c='orange')
plt.plot(sol_unstable_neg.y[0], sol_unstable_neg.y[1], c='orange', label="Unstable Manifold ")
plt.plot(sol_stable_pos.y[0], sol_stable_pos.y[1], c='green')
plt.plot(sol_stable_neg.y[0], sol_stable_neg.y[1], c='green', label="Stable Manifold")
plt.plot(point[0], point[1], 'ro', label="Fixed Point (  ,0)")

plt.legend()
plt.show()

# Graficar el segundo manifold en una figura separada
plt.figure(figsize=(6, 5))
plt.plot(manifold_2_x, manifold_2_y, label='Normal form style', color='blue')
plt.xlim(0, 2 * np.pi)
plt.ylim(-2.5, 2.5)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Normal form style: $(\pi + s - \\frac{s^3}{48}, s - \\frac{s^3}{16})$')
plt.plot(sol_unstable_pos.y[0], sol_unstable_pos.y[1], c='orange')
plt.plot(sol_unstable_neg.y[0], sol_unstable_neg.y[1], c='orange', label="Unstable Manifold ")
plt.plot(sol_stable_pos.y[0], sol_stable_pos.y[1], c='green')
plt.plot(sol_stable_neg.y[0], sol_stable_neg.y[1], c='green', label="Stable Manifold")
plt.plot(point[0], point[1], 'ro', label="Fixed Point (  ,0)")
plt.legend()
plt.show()
```

Computation of the symmetric homoclinic orbits of the equilibrium points

Pol Navarro Pérez

November 20, 2024

Abstract

In this work, we study one of the branches of the unstable manifold for an equilibrium point of the Restricted Three-Body Problem (RTBP). Additionally, based on the symmetry of the solution, we search for a parameter μ that yields a derivative x' equal to zero, thereby obtaining a homoclinic orbit. By exploring a range of μ values, we observe different types of discontinuities in the function $x'(\mu)$, caused by changes in the crossing point with the Poincaré section and the intersection with a primary body.

Preliminary

The work starts checking that if (t, x, y, x', y') is a solution then (-t, x, -y, -x', -y') is also a solution. The system of our RTBP is:

$$\begin{cases} x'' - 2y' = \frac{\partial \Omega}{\partial x} \\ y'' + 2x' = \frac{\partial \Omega}{\partial y} \end{cases} \tag{1}$$

where $' = \frac{d}{dt}$, $\quad \Omega(x,y) = \frac{1}{2}(x^2+y^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2} + \frac{1}{2}\mu(1-\mu)$, $\quad \mu = \frac{m_2}{m_1+m_2} \in (0, \frac{1}{2}]$,

$$r_1 = \text{dist}(P_1, P_3), \quad r_2 = \text{dist}(P_2, P_3).$$

Therefore, taking $x_1 = x,$, $x_2 = y$, $x_3 = x'$, $x_4 = y'$ the system becomes:

$$\begin{cases} \frac{dx_1}{dt} = x_1' = x_3 \\ \frac{dx_2}{dt} = x_2' = x_4 \\ \frac{dx_3}{dt} = x_3' = 2x_4 + \Omega_{x_1} \\ \frac{dx_4}{dt} = x_4' = -2x_3 + \Omega_{x_2} \end{cases} \tag{2}$$

As we know Ω, r_1 and r_2 we can substitute and get:

$$\begin{cases} x_1' = x_3 \\ x_2' = x_4 \\ x_3' = 2x_4 + x_1\left(1 - \frac{1-\mu}{r_1^3} - \frac{\mu}{r_2^3}\right) - \frac{\mu(1-\mu)}{r_1^3} - \frac{\mu(1-\mu)}{r_2^3} \\ x_4' = 2x_3 + x_2\left(1 - \frac{1-\mu}{r_2^3} - \frac{\mu}{r_2^3}\right) \end{cases} \tag{3}$$

To check that there is a symmetry in 3 we introduce the change of variables stated, $x_1 = x_1$, $x_2 = -x_2$, $x_3 = -x_3$, $x_4 = -x_4$ and finally $t = -t$, which means the derivative is the opposite sign. For the distance to the equilibrium point, this is r_i, the changes do not affect, since $x_1 = x_{1,symmetric}$ and the y squared inside the square root. This is:

$$\begin{cases} -x_1' = -x_3 \\ x_2' = x_4 \\ x_3' = 2x_4 + x_1\left(1 - \frac{1-\mu}{r_1^3} - \frac{\mu}{r_2^3}\right) - \frac{\mu(1-\mu)}{r_1^3} - \frac{\mu(1-\mu)}{r_2^3} \\ -x_4' = -2x_3 - x_2\left(1 - \frac{1-\mu}{r_2^3} - \frac{\mu}{r_2^3}\right) \end{cases} \tag{4}$$

From Equation 4 we get the same equations as in Equation 3. This checks there is the stated symmetry and it corresponds to a solution.

PART A

The first part is about computing symmetric homoclinic orbits of the L_3 equilibrium point of the RTBP.
Our first step will be then compute L_3 given a value of μ. For the equilibrium points, we know that $F(X) = 0$ and this implies that $x_3 = x_4 = 0$. Following the procedure from theory sessions, we get a value such that $L_3 = (\mu + \xi, 0, 0, 0)$

with $\xi \in (0,1)$. To find the ξ value we consider the equation from the polynomial equation to the one corresponding to a fixed point. This is,

$$\xi = \left[\frac{(1-\mu)(1+\xi)^2}{1+2\mu+\xi(2+\mu+\xi}\right]^{\frac{1}{3}} \tag{5}$$

and we have $\xi = f(\xi)$ and a initial value $\xi_0 = 1 - \frac{7\mu}{12}$. Thereafter, we compute the value of $f(\xi)$ until we get $|f(\xi_n) - \xi_n| < tol$, with tol a certain tolerance.
The final value is computed using this value of ξ and using the L_3 coordinates. For instance, in assignment 8 we used a value of $\mu = 0.1$ and computed the equilibrium point $L_3 = 1.041608908571061$.

Unstable Manifold

Once we have the point L_3 we have generated the unstable manifold, in this case only the branch W_{u+} that starts on the $y < 0$. Using the Poincaré section, we generate it until the first crossing with y = 0.

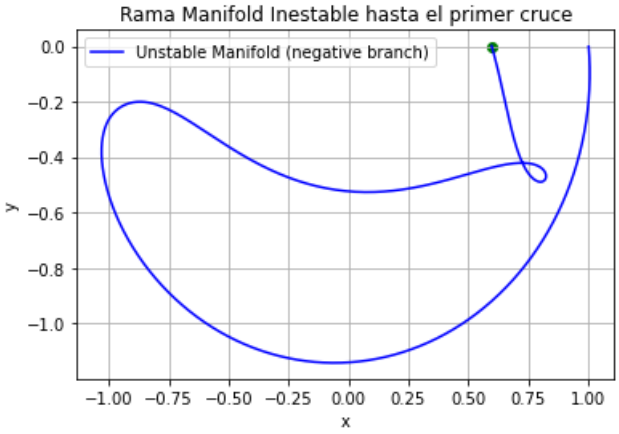


Figure 1: Plot of the $y < 0$ branch from the unstable manifold until the first crossing with y = 0. We have used a value $\mu = 0.008$.

As can be seen in Fig.1 this will not be a homoclinic orbit. From theory classes and the preliminary discussion of the symmetry solution, a homoclinic orbit will be the one that at the first crossing $x' = 0$. Thus, we need to find the values of μ that satisfy this condition and then find the corresponding branch W_{u+}. Then, using the symmetric solution we can find the homoclinic orbit such that:

$$lim_{t\to\pm\infty}\phi(t,x) = L_3 \tag{6}$$

We iterate through multiple parameters of μ using a small increment and then plot the corresponding values of $x'(\mu)$.

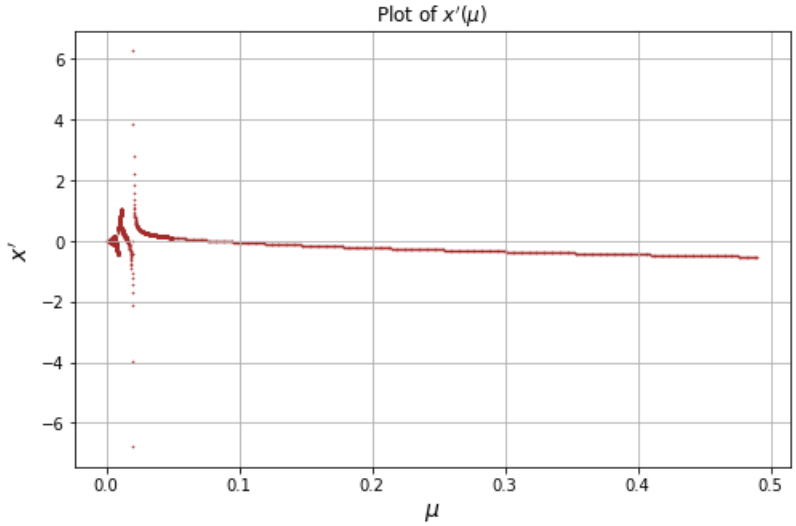
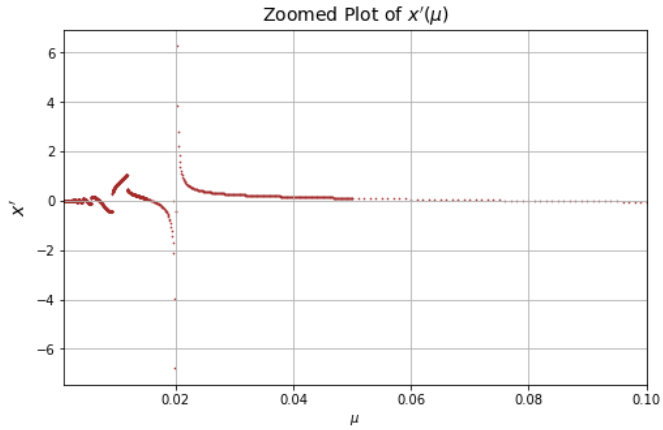
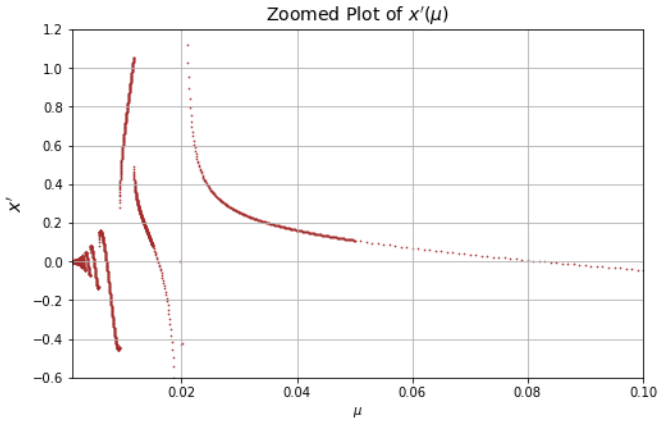


Figure 2: Plot of the $x'(\mu)$ using 1 crossing in the Poincaré section. μ in the range (0.001, 0.5).

PART B



Zoom for μ in (0.001, 0.1).

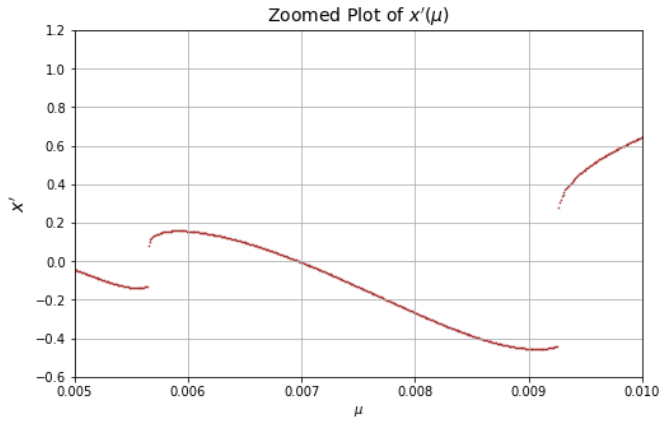


Zoom with x' label limited.

Figure 3: Zoomed plots for $x'(\mu)$

Figures 2 and 3 show that it is not a continuous function, there are some discontinuities. Moreover, there are branch-like structures with the shape of lines with a negative slope, interspersed with discontinuities between them. Furthermore, for a certain value of μ, there is an asymptotic discontinuity.
To study better these discontinuities, around the interval $\mu \in (0.001, 0.1)$, we plot a zoom of the previous figure.

As commented before, we are working with two types of discontinuities, an asymptotic one and many finite jump discontinuities. The best way to approach the study of these is by plotting their respective unstable manifold branch. These singular μ values are for instance, the μ of the finite jump in (0.005, 0.006) and the asymptotic μ around 0.02.



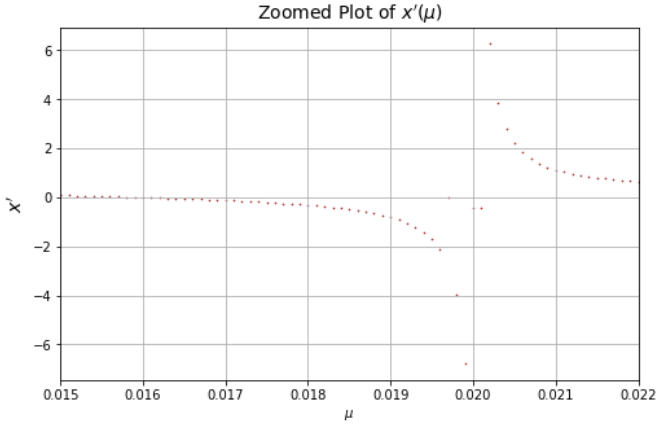


Figure 4: Zoomed plots for $x'(\mu)$ for the singular μ values.

Figure 4 shows the particular singular values of μ we are going to work with. The specific values of μ can be found by searching our data for the typical values of x' on each side and then identifying where a significant change occurs. For one of the finite jump discontinuities, the μ value is around 0.005645. Therefore, we will plot the corresponding branch of the unstable manifold and some close values to look for any difference between these.

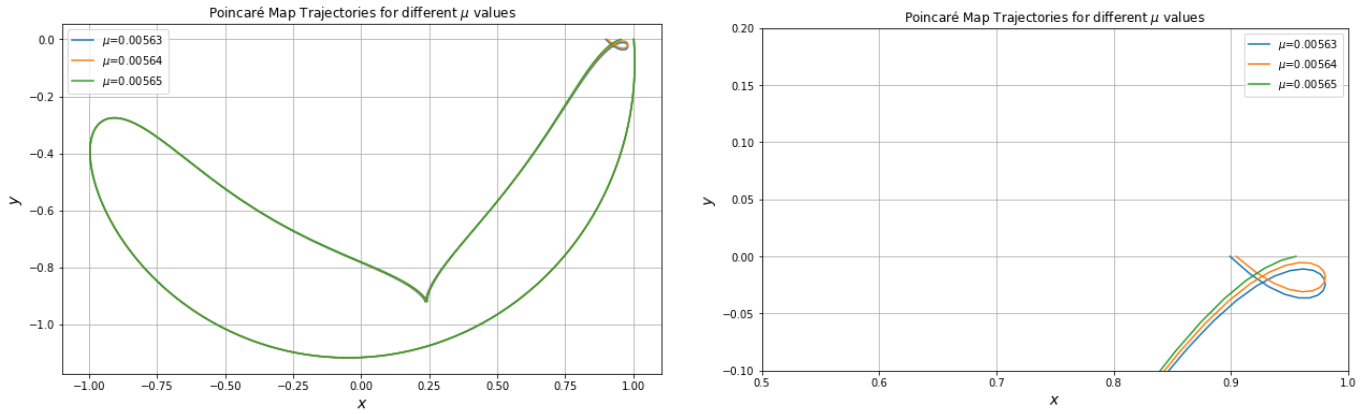


Figure 5: Manifold and zoom of different μ values around one of the finite jump discontinuities.

As it can be seen in Figure, 5 there is an important difference around this value. In this particular case, for $\mu = 0.00563$ the first crossing is at a different point compared to lower values, or better said, long before. The branch is closer to the x-axis around this kind of loop and the crossing point with the Poincaré section is nowhere near. Therefore, we can not expect the x' value to be similar to the previous μ values as the crossing point we are evaluating this derivative is completely different.
Moreover, by examining the general shape of the manifold and Figure 2, we can extract another insight. Specifically, there are different types of finite jump discontinuities. Notably, one discontinuity within the range $\mu \in (0.01, 0.02)$ appears distinct from the others.

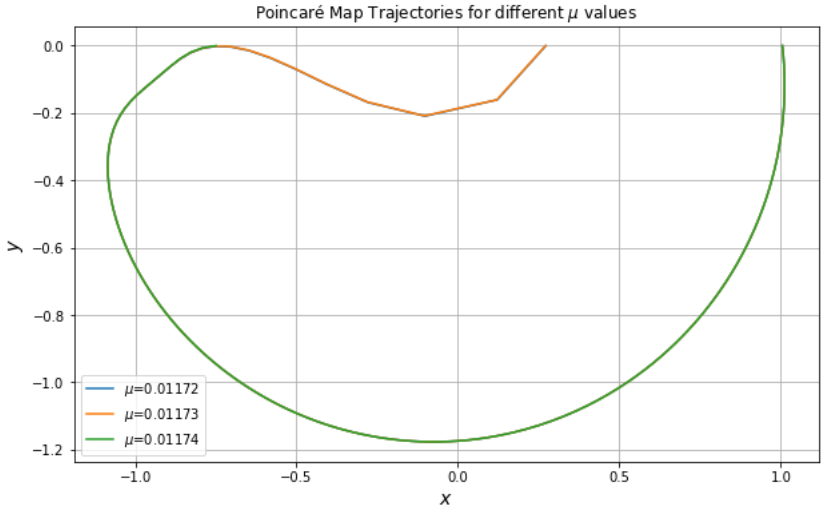


Figure 6: Branch of the unstable manifold for different μ values.

Figure 6 shows a difference from the previous jumps. In this case, the branch is closer to the x-axis, not the little loop, and therefore it reaches the Poincaré section before the previous μ values.

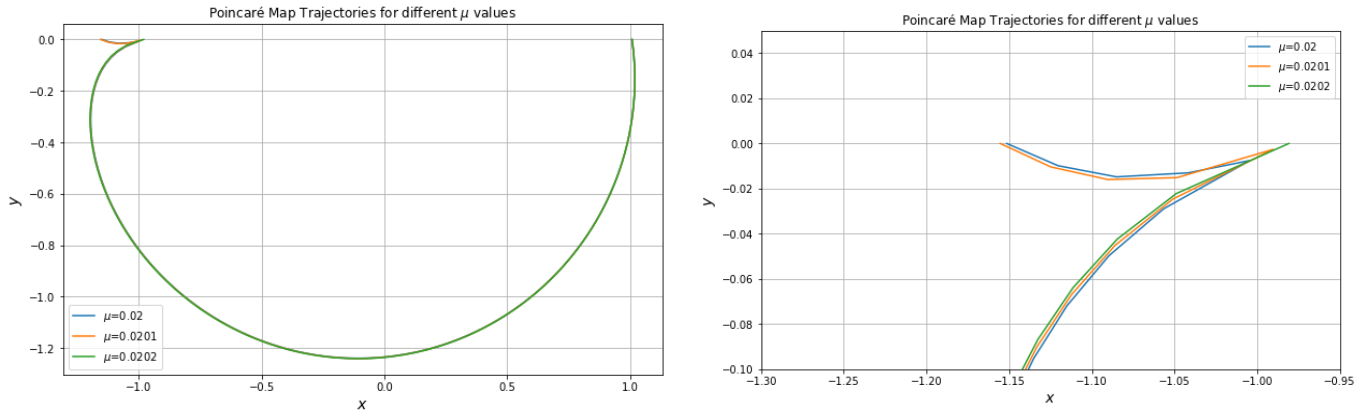


Figure 7: Manifold and zoom of different μ values around the asymptotic discontinuity.

As seen in Figure 7, and as mentioned before, the branch crosses the Poincaré section ($y = 0$) at a different point. In this particular case, for $\mu = 0.0202$, this crossing changes compared to nearby previous μ values. Because of this, we evaluate x' at a different point, and thus the corresponding result may not have any relation to the previous one. In addition, as stated before, this was an asymptotic discontinuity. This has to be studied a little deeply to understand why.

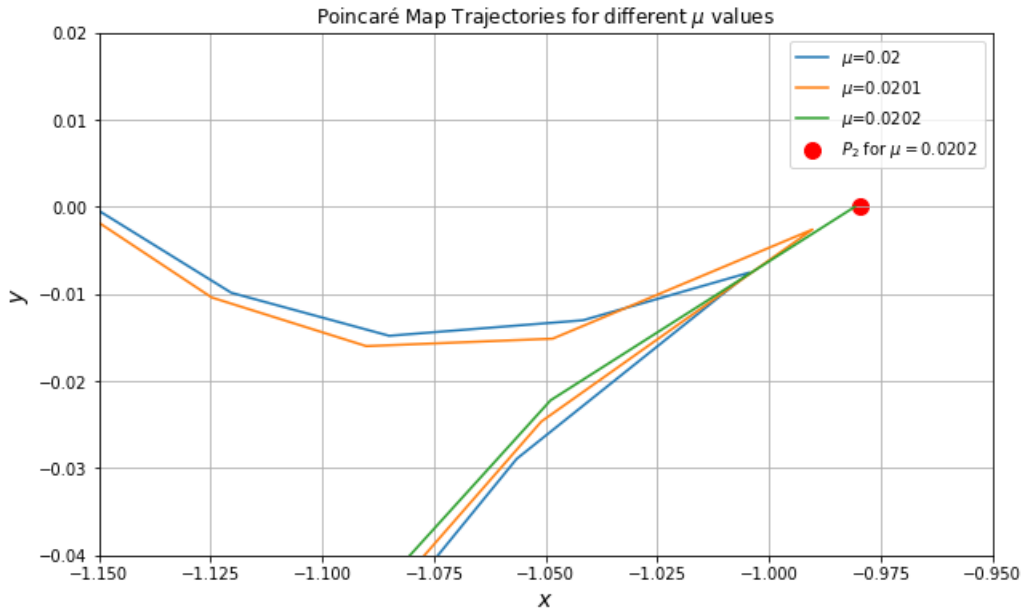


Figure 8: Zoomed plot of the branch from the unstable manifold for some μ values of interest.

As seen in Figure 8, our singular value of $\mu = 0.0202$ not only crosses the poincaré section long before but also intersects our second primary body P_2. By revisiting equation 1 and considering the already mentioned distance $r_2 = \sqrt{(x - \mu + 1)^2 + y^2}$, we see that the value of x' when our system intersects the P_2 becomes a singularity, as the denominator tends to zero.

Physically this says that the velocity is going to be huge as we get closer to a body, intuitively comprehensible if we think about a body attracted by a planet for instance. The RTBP proposed equation allows us to see how, in this situation, both the acceleration and velocity of the object would tend to infinity as we approach a body with significantly greater mass.

To summarize the findings on discontinuities, the function $x'(\mu)$ exhibits singularities or key regions with distinct behaviors. These behaviors depend on the intersections of the branch with the Poincaré section. By varying the value of μ, the branch may reach the Poincaré section earlier or even intersect with one of the primary bodies, as previously analyzed. This has to be taken into account when attempting to find a homoclinic orbit, branch $x' = 0$, considering the problem's symmetry.

CODE

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 15 10:31:22 2024

@author: polop
"""
# pip install rich necessary
from rich.progress import Progress
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import os



# Computation of Li points, using an iteration method for the quintic polynomial equation



def L1(mu):  # Returns the x for L1 given mu
    convergence = 0
    # initial value
    x = (mu / (3 * (1 - mu))) ** (1 / 3)
    tope_iter = 1000
    iteration_count = 0
    # itera method inside the own computation of the equilibrium point
    while convergence == 0 and iteration_count < tope_iter:
        xk = (mu * (1 - x) ** 2 / (3 - 2 * mu - x * (3 - mu - x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu - 1 + x

def L2(mu):  # Returns the x for L2 given mu
    convergence = 0
    tope_iter = 1000
    iteration_count = 0
    # initial value epsilon_0
    x = (mu / (3 * (1 - mu))) ** (1 / 3)
    while convergence == 0 and iteration_count < tope_iter:
        xk = (mu * (1 + x) ** 2 / (3 - 2 * mu + x * (3 - mu + x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu - 1 - x

def L3(mu):  # Returns the x for L3 given mu
    convergence = 0
    x = 1 - 6 * mu / 12
    tope_iter = 1000
    iteration_count = 0
    while convergence == 0 and iteration_count < tope_iter:
        xk = ((1 - mu) * (1 + x) ** 2 / (1 + 2 * mu + x * (2 + mu + x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu + x


# Jacobi constant and derivatives:


# PARTIAL DERIVATIVES
def Omega_x(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return x - (1 - mu) * (x - mu) / (r1 ** 3) - mu * (x - mu + 1) / (r2 ** 3)

def Omega_y(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return y - (1 - mu) * y / (r1 ** 3) - mu * y / (r2 ** 3)
```

```
def Omega_xx(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 - (1 - mu) / r1 ** 3 + (3 * (1 - mu) * (x - mu) ** 2) / (r1 ** 5) - mu / r2 ** 3 + (3 *
        mu * (x - mu + 1) ** 2) / r2 ** 5

def Omega_xy(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return y * ((3 * (1 - mu) * (x - mu)) / r1 ** 5 + (3 * mu * (x - mu + 1)) / r2 ** 5)

def Omega_yy(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 - (1 - mu) / r1 ** 3 - mu / r2 ** 3 + y ** 2 * ((3 * (1 - mu)) / r1 ** 5 + (3 * mu) /
        r2 ** 5)

# OMEGA TOTAL
def Omega(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 / 2 * (x ** 2 + y ** 2) + (1 - mu) / r1 + mu / r2 + 1 / 2 * mu * (1 - mu)

# Function to compute the dynamical_system


def dynamical_system(t, x, mu):
    # C lculo de las distancias r1 y r2
    r1 = np.sqrt((x[0] - mu) ** 2 + x[1] ** 2)
    r2 = np.sqrt((x[0] - mu + 1) ** 2 + x[1] ** 2)

    # C lculo de Omega_x y Omega_y
    Omega_x = x[0] - (1 - mu) * (x[0] - mu) / (r1 ** 3) - mu * (x[0] - mu + 1) / (r2 ** 3)
    Omega_y = x[1] - (1 - mu) * x[1] / (r1 ** 3) - mu * x[1] / (r2 ** 3)

    # C lculo de las derivadas
    df1 = x[2]
    df2 = x[3]
    df3 = 2 * x[3] + Omega_x
    df4 = -2 * x[2] + Omega_y

    return df1, df2, df3, df4

# Jacobian of the RTBP
def Jacobian(x, y, mu):
    return np.array([
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [Omega_xx(x, y, mu), Omega_xy(x, y, mu), 0, 2],
        [Omega_xy(x, y, mu), Omega_yy(x, y, mu), -2, 0]
    ])

# Eigenvalues and eigenvectors of the L3
def get_eigenvalues(mu):
    return np.linalg.eig(Jacobian(L3(mu), 0, mu))[0]

def get_eigenvectors(mu):
    return np.linalg.eig(Jacobian(L3(mu), 0, mu))[1]

# Functions for the Poincar  Map
def g_sigma(x):
    return x[1]

def grad_g_sigma(x):
    return np.array([0, 1, 0, 0])

import numpy as np
from scipy.integrate import solve_ivp

def poincare_map(state, num_crossings, mu):

    # Determinar el l mite de tiempo basado en el valor de mu
    def get_max_time(mu):
        ranges = {
            0.005: 500,
            0.01: 250,
            0.1: 100
        }
        for threshold, time in ranges.items():
```

```
        if mu < threshold:
            return time
    return 50

max_time = get_max_time(mu)

# N mero de puntos en el tiempo para evaluar
num_eval_points = 500000

# Comprobar que la direcci n es v lida
direction_check = 1
if direction_check not in [1, -1]:
    print("-1 BACKWARD , +1 FORWARD")
    return

# Inicializar variables del sistema
accumulated_time = 0
pos_y = state[1]
vel_x = state[2]
vel_y = state[3]
pos_x = state[0]

# Iterar para buscar m ltiples cruces
for crossing in range(num_crossings):
    # Definir el intervalo de tiempo
    integration_span = [0, max_time]

    # Realizar un paso de integraci n corto si no es el primer cruce
    if crossing != 0:
        solution = solve_ivp(
            fun=lambda t, var: dynamical_system(t, var, mu),
            t_span=integration_span,
            y0=[pos_x, pos_y, vel_x, vel_y],
            t_eval=np.array([direction_check * 1E-10]),
            rtol=3e-14,
            atol=1e-14
        )
        accumulated_time += direction_check * 1E-10
        pos_x, pos_y, vel_x, vel_y = solution.y[:, 0]

    # Realizar la integraci n en un rango de tiempo
    evaluation_times = np.linspace(0, max_time, num_eval_points)
    solution = solve_ivp(
        fun=lambda t, var: dynamical_system(t, var, mu),
        t_span=integration_span,
        y0=[pos_x, pos_y, vel_x, vel_y],
        t_eval=evaluation_times,
        rtol=3e-14,
        atol=1e-14
    )

    initial_time = 0
    crossing_found = False

    # Detectar cruce con el eje y=0 hacia adelante o atr s
    if pos_y >= 0:
        for time_step in range(num_eval_points):
            if solution.y[1][time_step] < 0:
                if direction_check == 1:
                    pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step - 1]
                    initial_time = solution.t[time_step - 1]
                    crossing_found = True
                    break
                else:
                    pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step]
                    initial_time = solution.t[time_step]
                    crossing_found = True
                    break

    if pos_y < 0 and not crossing_found:
        for time_step in range(num_eval_points):
            if solution.y[1][time_step] > 0:
                if direction_check == 1:
                    pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step - 1]
                    initial_time = solution.t[time_step - 1]
                    break
                else:
                    pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step]
                    initial_time = solution.t[time_step]
```

```
                        break

        # Ajuste de Newton para refinar el cruce
        convergence_reached = False
        iterations = 0
        current_step_time = 0

        while not convergence_reached and iterations < 1000:
            solution = solve_ivp(
                fun=lambda t, var: dynamical_system(t, var, mu),
                t_span=integration_span,
                y0=[pos_x, pos_y, vel_x, vel_y],
                t_eval=np.array([current_step_time]),
                rtol=3e-14,
                atol=1e-14
            )

            t_adjusted = current_step_time - (g_sigma(solution.y) /
                                              (np.dot(grad_g_sigma(solution.y), dynamical_system(0,
                                                  solution.y, mu))))[0]
            t_adjusted = np.real(t_adjusted) % (2 * np.pi)

            if np.abs(current_step_time - t_adjusted) < 1e-14:
                convergence_reached = True

            current_step_time = t_adjusted
            iterations += 1

            if iterations > 2000:
                print('Error: Too many iterations')

        solution = solve_ivp(
            fun=lambda t, var: dynamical_system(t, var, mu),
            t_span=integration_span,
            y0=[pos_x, pos_y, vel_x, vel_y],
            t_eval=np.array([current_step_time]),
            rtol=3e-14,
            atol=1e-14
        )

        pos_x, pos_y, vel_x, vel_y = solution.y[:, 0]
        accumulated_time += current_step_time + initial_time

    return [solution.y, accumulated_time]




# Definir el rango de mu
# mu_range = [0.005630, 0.005640, 0.005650]

# mu_range = [0.020000
#  ,0.020100
#  ,0.020200 ]
# mu_range = [0.009250,
# 0.009260 ,
# 0.009270 ]
mu_range = [0.011720,0.011730,0.011740]


# mu_range = [0.304]

# Inicializar listas para almacenar resultados
x_list = []
y_list = []
s = 1e-6 # for smaller values i get an error
for mu in mu_range:
    # Encontrar el vector propio para el valor propio positivo
    for i in range(len(get_eigenvalues(mu))):
        if np.isreal(get_eigenvalues(mu)[i]) and get_eigenvalues(mu)[i] > 0:
            v0 = get_eigenvectors(mu)[:, i]

    # Calcular el punto inicial x0
    L3_value = L3(mu)
    x0 = np.array([L3_value, 0, 0, 0]) +  s* v0
    if x0[1] > 0:
        x0 = np.array([L3_value, 0, 0, 0]) - s * v0
```

```
    # Calcular el mapa de Poincar  y la soluci n del sistema
    a = poincare_map(x0, 1, mu)
    sol = solve_ivp(
        fun=lambda t, y: dynamical_system(t, y, mu),
        t_span=[0, a[1]],
        y0=x0,
        t_eval=np.linspace(0, a[1], 1000),
        rtol=3e-14,
        atol=1e-14
    )

    # Almacenar los resultados en las listas
    x_list.append(sol.y[0])
    y_list.append(sol.y[1])


# Graficar los resultados en un solo plot
plt.figure(figsize=(10, 6))
for j in range(len(mu_range)):
    plt.plot(x_list[j], y_list[j], label=f'$\mu$={mu_range[j]}')

plt.xlabel(r"$x$",fontsize = 14)
plt.ylabel(r"$y$",fontsize = 14)
# plt.xlim(-1.15, -0.95)
# plt.ylim(-0.04, 0.02)
plt.title("Poincar  Map Trajectories for different $\mu$ values")

plt.grid(True)
# Calcular el punto L2 para mu = 0.0202

# # Coordenadas del segundo primario P2 para mu = 0.0202
# mu_target = 0.0202
# P2_x = 1 - mu_target
# P2_y = 0  # El segundo primario est  en y = 0

# # Dibujar el punto P2 en el gr fico
# plt.scatter(-P2_x, -P2_y, color='red', marker='o', s=100, label=r'$P_2$ for $\mu=0.0202$')
plt.legend()
plt.show()



#%%

import numpy as np
import matplotlib.pyplot as plt
from rich.progress import Progress
from scipy.integrate import solve_ivp
import os

# Definir intervalos con nombres modificados
range_A = np.linspace(0.001, 0.015, round((0.015 - 0.001) / 0.00001) + 1)
range_B = np.linspace(0.015, 0.05, round((0.05 - 0.015) / 0.0001) + 1)
range_C = np.linspace(0.05, 0.49, round((0.49 - 0.05) / 0.001) + 1)

# Funci n para calcular los cruces
def compute_intersections(mu_val):
    for idx in range(len(get_eigenvalues(mu_val))):
        if np.isreal(get_eigenvalues(mu_val)[idx]) and get_eigenvalues(mu_val)[idx] > 0:
            eig_vector = get_eigenvectors(mu_val)[:, idx]

    initial_state = np.array([L3(mu_val), 0, 0, 0]) + 1E-6 * eig_vector
    if initial_state[1] > 0:
        initial_state = np.array([L3(mu_val), 0, 0, 0]) - 1E-6 * eig_vector

    results = poincare_map(initial_state, 1, mu_val)[0]
    return np.array([results[0][0], results[1][0], results[2][0], results[3][0]])

# Inicializar listas para almacenar resultados
res_A_x, res_A_y, res_A_xp, res_A_yp = [], [], [], []
res_B_x, res_B_y, res_B_xp, res_B_yp = [], [], [], []
res_C_x, res_C_y, res_C_xp, res_C_yp = [], [], [], []

# Procesar intervalos con `rich.progress`
with Progress() as progress:
    task_A = progress.add_task("[blue]Processing Range A...", total=len(range_A))
    for mu_val in range_A:
        results = compute_intersections(mu_val)
```

```
        res_A_x.append(results[0])
        res_A_y.append(results[1])
        res_A_xp.append(results[2])
        res_A_yp.append(results[3])
        progress.update(task_A, advance=1)

    task_B = progress.add_task("[green]Processing Range B...", total=len(range_B))
    for mu_val in range_B:
        results = compute_intersections(mu_val)
        res_B_x.append(results[0])
        res_B_y.append(results[1])
        res_B_xp.append(results[2])
        res_B_yp.append(results[3])
        progress.update(task_B, advance=1)

    task_C = progress.add_task("[magenta]Processing Range C...", total=len(range_C))
    for mu_val in range_C:
        results = compute_intersections(mu_val)
        res_C_x.append(results[0])
        res_C_y.append(results[1])
        res_C_xp.append(results[2])
        res_C_yp.append(results[3])
        progress.update(task_C, advance=1)

# Funci n para guardar los resultados en un archivo .txt
def save_results_to_file(filename, data_ranges, results, labels):
    with open(filename, 'w') as output:
        for range_data, result_data, label in zip(data_ranges, results, labels):
            output.write(f"# Results for {label}\n")
            for val_mu, val_result in zip(range_data, result_data):
                output.write(f"{val_mu:.6f} {val_result:.6f}\n")
            output.write("\n")
    print(f"Data has been saved to {os.path.abspath(filename)}")

# Guardar los resultados para los tres intervalos
output_filename = "computed_intersections.txt"
data_intervals = [range_A, range_B, range_C]
results_list = [res_A_xp, res_B_xp, res_C_xp]
interval_labels = ["Interval A", "Interval B", "Interval C"]

save_results_to_file(output_filename, data_intervals, results_list, interval_labels)

# Graficar los resultados
plt.figure(figsize=(10, 6))
plt.scatter(range_B, res_B_xp, color="brown", label="Range B")
plt.scatter(range_C, res_C_xp, color="brown", label="Range C")
plt.scatter(range_A, res_A_xp, color="brown", label="Range A")
plt.xlabel(r"$\mu$", fontsize=14)
plt.ylabel(r"$x'$", fontsize=14)
plt.title("Combined Plot of $x'$ vs $\mu$")
plt.legend()
plt.grid(True)
plt.show()
```

Assignment 10: COMPUTATION OF PERIODIC ORBITS FOR THE RTBP

Pol Navarro Pérez

November 22, 2024

Abstract

This work investigates periodic orbits in the Restricted Three-Body Problem (RTBP) using numerical methods. A computational routine is implemented to identify initial conditions leading to homoclinic orbits, employing the bisection method for root-finding. By varying the Jacobi constant C, we examine the family of Lyapunov orbits and observe their behavior relative to equilibrium points. Results demonstrate that as C increases, the initial condition converges towards the L_3 equilibrium point, with periodic orbits approaching a period of $T = 5.83$ a.u.

Numerical methods for x' in a RTBP orbit

We first implement a routine that for a fixed value C computes the corresponding y', with the initial condition of the orbit as $(x, 0, 0, y')$.

As commented in theory sessions, we will use this initial state to compute the final value of x'. Then, using the bisection method to find the roots given two initial states, we can find the one that has $x' = 0$ at the Poincaré section.

As we are working with the RTBP we need to remember the dynamical system we are working with:

$$\begin{cases} x'' - 2y' = \frac{\partial \Omega}{\partial x} \\ y'' + 2x' = \frac{\partial \Omega}{\partial y} \end{cases} \tag{1}$$

where $' = \frac{d}{dt}$, $\quad \Omega(x,y) = \frac{1}{2}(x^2+y^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2} + \frac{1}{2}\mu(1-\mu), \quad \mu = \frac{m_2}{m_1+m_2} \in (0, \frac{1}{2}]$,

$$r_1 = \text{dist}(P_1, P_3), \quad r_2 = \text{dist}(P_2, P_3).$$

Therefore, taking $x_1 = x,$, $x_2 = y$, $x_3 = x'$, $x_4 = y'$ the system becomes:

$$\begin{cases} \frac{dx_1}{dt} = x_1' = x_3 \\ \frac{dx_2}{dt} = x_2' = x_4 \\ \frac{dx_3}{dt} = x_3' = 2x_4 + \Omega_{x_1} \\ \frac{dx_4}{dt} = x_4' = -2x_3 + \Omega_{x_2} \end{cases} \tag{2}$$

As we know Ω, r_1 and r_2 we can substitute and get:

$$\begin{cases} x_1' = x_3 \\ x_2' = x_4 \\ x_3' = 2x_4 + x_1\left(1 - \frac{1-\mu}{r_1^3} - \frac{\mu}{r_2^3}\right) - \frac{\mu(1-\mu)}{r_1^3} - \frac{\mu(1-\mu)}{r_2^3} \\ x_4' = 2x_3 + x_2\left(1 - \frac{1-\mu}{r_2^3} - \frac{\mu}{r_2^3}\right) \end{cases} \tag{3}$$

For this system, the given fixed value C refers to the Jacobi constant or first integral

$$C = 2\Omega(x,y) - (x'^2 - y'^2) \tag{4}$$

Using Equation 4 and considering the initial condition $(x, 0, 0, y')$ we can find $y' = -\sqrt{C - 2\Omega(x,0)}$. Then, for a given value x we can compute the orbit until the crossing with a Poincaré section, for instance, $y = 0$, and find the corresponding x' at this crossing.

As commented in the previous assignment, if we want to find the homoclinic orbits, using the symmetry of the solution, we need to find x_0 such that $x'(x_0) = 0$. Therefore, we can implement a routine to find two initial x values with opposite sign derivatives x' and the one with 0 as the derivative.

For the first step we start at a point displaced by 10^{-5} from the L_3 point, this is x_0. Let us call the two interval limits x_1 and x_2, with $x_0 = x_1 = x_2$. Thereafter, we check the respective x' values for both interval limits. If there is no sign change, $f(x_1) \cdot f(x_2) > 0$, we move the whole interval by some value, let us take 10^{-4}. We have fixed a maximum of iterations for this procedure. For instance, with these previous values, we have obtained the corresponding interval $(x_1, x_2, f(x_1), f(x_2)) = (1.060618908571058, 1.061618908571058, -0.0016281794609634447, 0.0026489510496346316)$, with a change of sign as we wanted.

At this point, we could apply the bisection method in order to refine this interval $[x_1, x_2]$ to find x such that $F(x) = 0$.

Variation of the Jacobi constant

The family of Lyapunov orbits can be obtained by implementing the previous procedures and changing the Jacobi constant C.

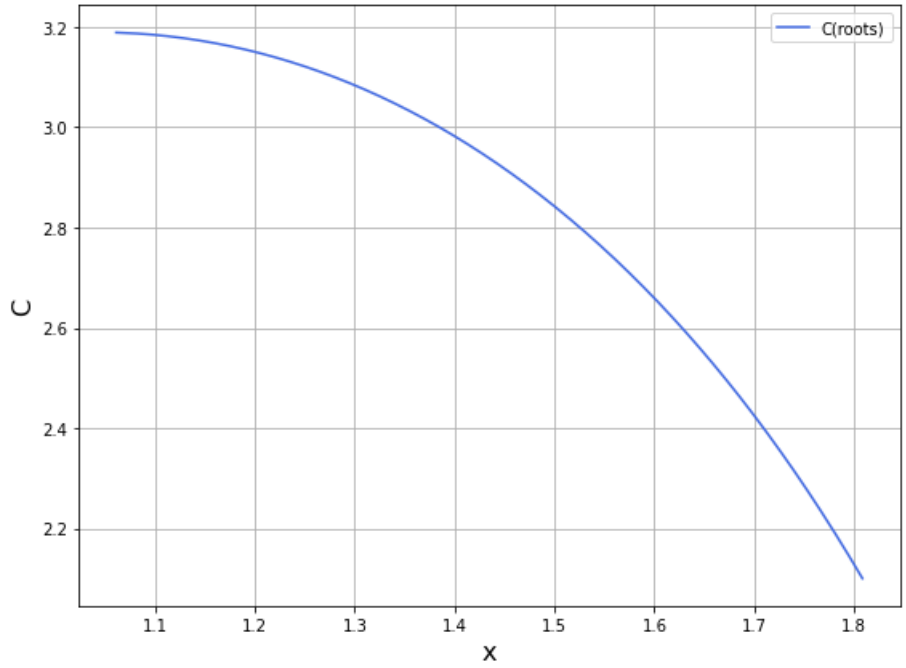


Figure 1: Jacobi constant C as the initial point $(x, 0)$, conditioned by $x' = 0$. $C \in (2.1, 3.189)$ with a $\Delta C = 0,0001$.

As Figure 1 shows, the value of x for the initial condition increases as the Jacobi constant decreases. It is not a straight line but a curved one.

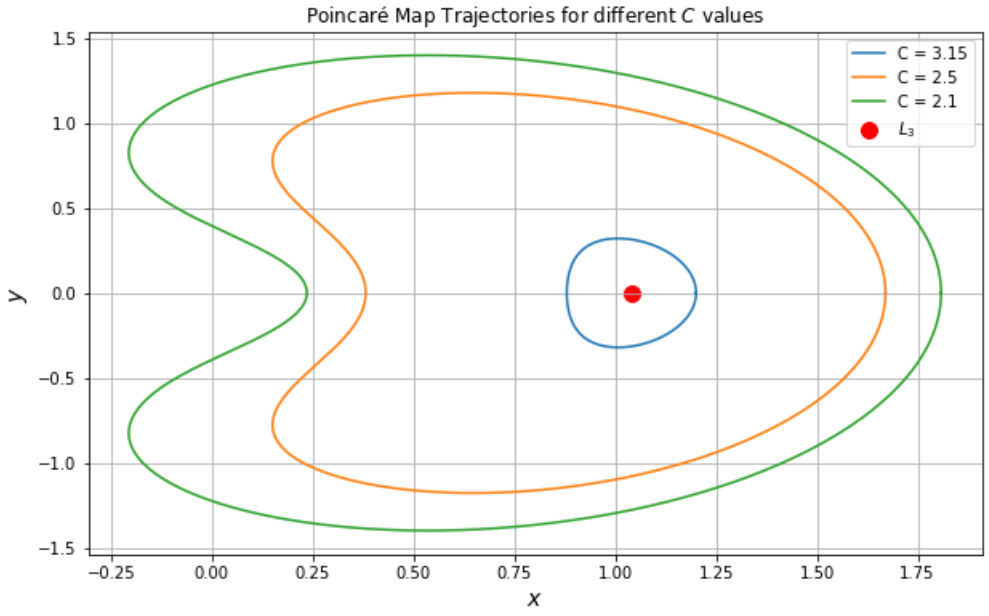


Figure 2: Complete orbits for different C values.

Figure 2 shows how the procedure of finding an initial point $(x_0, 0)$ that gets a $x' = 0$ once it reaches the Poincaré section, implies the acquisition of a homoclinic orbit. This complete orbit has been achieved by using two crossings at the Poincaré section function coded. We can clearly observe how the bigger the C value the closer we are from the equilibrium point L_3.

Possible C values

The main focus of this work has been to study how the Jacobi constant changes the initial point required to get a homoclinic orbit. In the previous section, we have worked with $C \in (2.1, 3.15)$. Nevertheless, this is not an arbitrary range.

For higher values of C, we would get much closer to the equilibrium point so we can intuitively think this would be problematic. In addition, if we see our RTBP systems, for higher values of C the square root would be negative and there would be no solution. On the other hand, for lower C values there is another thing to consider. This is the locations of the two primaries, P_1 and P_2. We could plot again the orbits to observe how these can cause some problems.

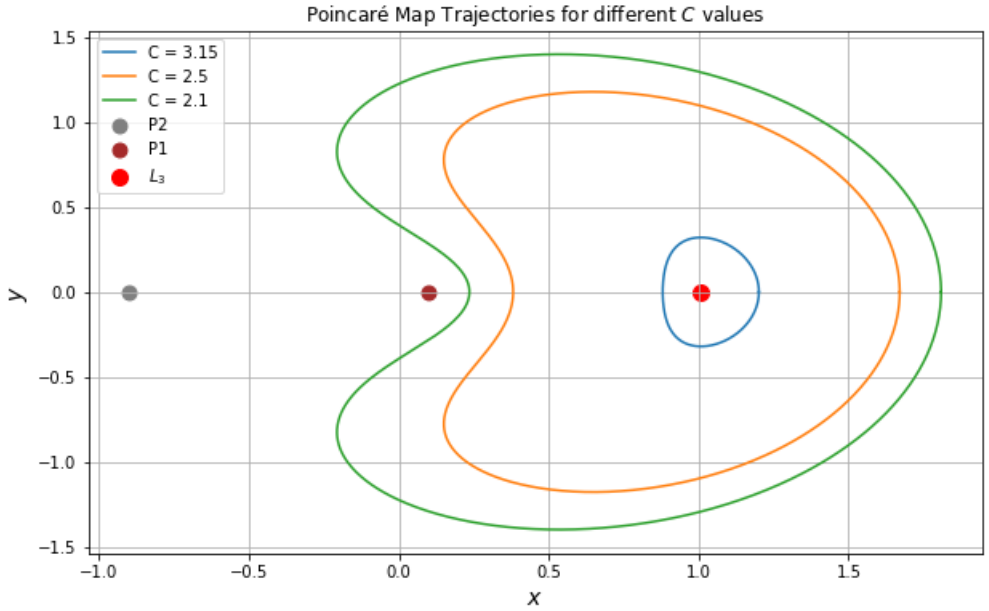


Figure 3: Orbits plotted for some C values and the primaries of our system for $\mu = 0.1$.

Figure 3 shows the problem of working with lower values of C. We would be intersecting one of our primary bodies if we used lower values for the Jacobi constant. Nevertheless, maybe there is some C value for which the corresponding orbits go through the gap between the primaries.

Lyapunov periodic orbits period

The Lyapunov theorem not only states the possibility of a family of periodic orbits under certain conditions but also the exact period these orbits tend to have. These tend to have a period $T = \frac{2\pi}{w}$. Therefore, we need to know the already computed imaginary part of the corresponding L_3 eigenvalues. This is $w = 1.077009310230949$. This means that when the orbit gets closer to the equilibrium point, L_3 in this case for $\mu = 0.1$, then the period will tend to $T = \frac{2\pi}{w} \approxeq 5.83$ a.u.

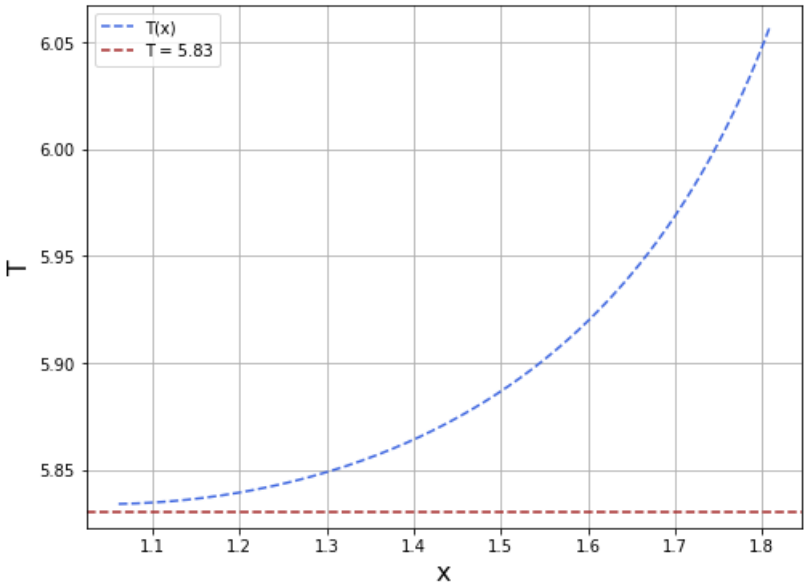


Figure 4: T period depending on the (x,0) initial condition.

As stated before and Figure 4 shows, as we increase the value of the Jacobi constant C, this means the start point closer to L_3, Figure 1, the period tends to the calculated value $T = 5.83$ a.u. It never reaches the exact value but gets closer.

CODE

Listing 1: code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 20 14:40:27 2024

@author: polop
"""

# BISECTION METHOD TO FIND ROOTS

"""Find two points, say a and b such that a < b and f(a)* f(b) < 0
Find the midpoint of a and b, say    t
t is the root of the given function if f(t) = 0; else follow the next step
Divide the interval [a, b]     If f(t)*f(a) <0, there exist a root between t and a
    else if f(t) *f (b) < 0, there exist a root between t and b
Repeat above three steps until f(t) = 0"""


import numpy as np
import matplotlib.pyplot as plt
from rich.progress import Progress
def bisection_method(f, x1, x2, tol=1e-8, max_iter=1000):

    # Caso especial: la ra z es exactamente uno de los l mites
    if f(x1) == 0:
        return x1
    if f(x2) == 0:
        return x2

    # Comprobar condici n de cambio de signo
    if f(x1) * f(x2) > 0:
        raise ValueError("El m todo de bisecci n requiere que f(x1) * f(x2) < 0.")

    iter_count = 0
    while abs(x2 - x1) > tol and iter_count < max_iter:
        c = (x1 + x2) / 2  # Punto medio del intervalo
        if f(c) == 0 or abs(x2 - x1) < tol:  # Ra z encontrada o intervalo suficientemente peque o
            return c
        elif f(x1) * f(c) < 0:
            # raiz en la parte izquierda
            x2 = c  # La ra z est  en [x1, c]
        else:
```

```
            # raiz en la parte derecha
            x1 = c  # La ra z  est  en [c, x2]
        # print(f(c))
        iter_count += 1

    return (x1 + x2) / 2  # Aproximaci n de la ra z despu s de alcanzar la tolerancia


# pip install rich necessary
from rich.progress import Progress
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import os




# Computation of Li points, using an iteration method for the quintic polynomial equation


def L1(mu):  # Returns the x for L1 given mu
    convergence = 0
    # initial value
    x = (mu / (3 * (1 - mu))) ** (1 / 3)
    tope_iter = 1000
    iteration_count = 0
    # itera method inside the own computation of the equilibrium point
    while convergence == 0 and iteration_count < tope_iter:
        xk = (mu * (1 - x) ** 2 / (3 - 2 * mu - x * (3 - mu - x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu - 1 + x

def L2(mu):  # Returns the x for L2 given mu
    convergence = 0
    tope_iter = 1000
    iteration_count = 0
    # initial value epsilon_0
    x = (mu / (3 * (1 - mu))) ** (1 / 3)
    while convergence == 0 and iteration_count < tope_iter:
        xk = (mu * (1 + x) ** 2 / (3 - 2 * mu + x * (3 - mu + x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu - 1 - x

def L3(mu):  # Returns the x for L3 given mu
    convergence = 0
    x = 1 - 6 * mu / 12
    tope_iter = 1000
    iteration_count = 0
    while convergence == 0 and iteration_count < tope_iter:
        xk = ((1 - mu) * (1 + x) ** 2 / (1 + 2 * mu + x * (2 + mu + x))) ** (1 / 3)
        if np.abs(x - xk) < 1e-17:
            convergence = 1
        x = xk
        iteration_count += 1
    return mu + x


# Jacobi constant and derivatives:


# PARTIAL DERIVATIVES
def Omega_x(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return x - (1 - mu) * (x - mu) / (r1 ** 3) - mu * (x - mu + 1) / (r2 ** 3)

def Omega_y(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return y - (1 - mu) * y / (r1 ** 3) - mu * y / (r2 ** 3)

```

```python
def Omega_xx(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 - (1 - mu) / r1 ** 3 + (3 * (1 - mu) * (x - mu) ** 2) / (r1 ** 5) - mu / r2 ** 3 + (3
        * mu * (x - mu + 1) ** 2) / r2 ** 5

def Omega_xy(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return y * ((3 * (1 - mu) * (x - mu)) / r1 ** 5 + (3 * mu * (x - mu + 1)) / r2 ** 5)

def Omega_yy(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 - (1 - mu) / r1 ** 3 - mu / r2 ** 3 + y ** 2 * ((3 * (1 - mu)) / r1 ** 5 + (3 * mu) /
        r2 ** 5)

# OMEGA TOTAL
def Omega(x, y, mu):
    r1 = np.sqrt((x - mu) ** 2 + y ** 2)
    r2 = np.sqrt((x - mu + 1) ** 2 + y ** 2)
    return 1 / 2 * (x ** 2 + y ** 2) + (1 - mu) / r1 + mu / r2 + 1 / 2 * mu * (1 - mu)

# Function to compute the dynamical_system


def dynamical_system(t, x, mu):
    # C lculo de las distancias r1 y r2
    r1 = np.sqrt((x[0] - mu) ** 2 + x[1] ** 2)
    r2 = np.sqrt((x[0] - mu + 1) ** 2 + x[1] ** 2)

    # C lculo de Omega_x y Omega_y
    Omega_x = x[0] - (1 - mu) * (x[0] - mu) / (r1 ** 3) - mu * (x[0] - mu + 1) / (r2 ** 3)
    Omega_y = x[1] - (1 - mu) * x[1] / (r1 ** 3) - mu * x[1] / (r2 ** 3)

    # C lculo de las derivadas
    df1 = x[2]
    df2 = x[3]
    df3 = 2 * x[3] + Omega_x
    df4 = -2 * x[2] + Omega_y

    return df1, df2, df3, df4

# Jacobian of the RTBP
def Jacobian(x, y, mu):
    return np.array([
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [Omega_xx(x, y, mu), Omega_xy(x, y, mu), 0, 2],
        [Omega_xy(x, y, mu), Omega_yy(x, y, mu), -2, 0]
    ])

# Eigenvalues and eigenvectors of the L3
# remove [0] to get imaginary part
def get_eigenvalues(mu):
    return np.linalg.eig(Jacobian(L3(mu), 0, mu))[0]

def get_eigenvectors(mu):
    return np.linalg.eig(Jacobian(L3(mu), 0, mu))[1]

# Functions for the Poincar  Map
def g_sigma(x):
    return x[1]

def grad_g_sigma(x):
    return np.array([0, 1, 0, 0])

import numpy as np
from scipy.integrate import solve_ivp

# Given an initial condition compute (x, 0, 0, yp)
def y_p(mu, x, C):
    # print(mu, x, C)
    y_prima = -1*np.sqrt(2*Omega(x, 0, mu)- C)

    x_p = 0
    y = 0
    x = x
    state = [x, y, x_p, y_prima]
```

```
    return state

# Given (x, 0, 0, yp) computes f(x) = x'
def poincare_map(state, num_crossings, mu):

    # Determinar el l mite de tiempo basado en el valor de mu
    def get_max_time(mu):
        ranges = {
            0.005: 500,
            0.01: 250,
            0.1: 100
        }
        for threshold, time in ranges.items():
            if mu < threshold:
                return time
        return 50

    max_time = get_max_time(mu)
    # max_time = 4

    # N mero de puntos en el tiempo para evaluar
    num_eval_points = 500000

    # Comprobar que la direcci n es v lida
    direction_check = 1
    if direction_check not in [1, -1]:
        print("-1 BACKWARD, +1 FORWARD")
        return

    # Inicializar variables del sistema
    accumulated_time = 0
    pos_y = state[1]
    vel_x = state[2]
    vel_y = state[3]
    pos_x = state[0]

    # Iterar para buscar m ltiples cruces
    for crossing in range(num_crossings):
        # Definir el intervalo de tiempo
        integration_span = [0, max_time]

        # Realizar un paso de integraci n corto si no es el primer cruce
        if crossing != 0:
            solution = solve_ivp(
                fun=lambda t, var: dynamical_system(t, var, mu),
                t_span=integration_span,
                y0=[pos_x, pos_y, vel_x, vel_y],
                t_eval=np.array([direction_check * 1E-10]),
                rtol=3e-14,
                atol=1e-14
            )
            accumulated_time += direction_check * 1E-10
            pos_x, pos_y, vel_x, vel_y = solution.y[:, 0]

        # Realizar la integraci n en un rango de tiempo
        evaluation_times = np.linspace(0, max_time, num_eval_points)
        solution = solve_ivp(
            fun=lambda t, var: dynamical_system(t, var, mu),
            t_span=integration_span,
            y0=[pos_x, pos_y, vel_x, vel_y],
            t_eval=evaluation_times,
            rtol=3e-14,
            atol=1e-14
        )

        initial_time = 0
        crossing_found = False

        # Detectar cruce con el eje y=0 hacia adelante o atr s
        if pos_y >= 0:
            for time_step in range(num_eval_points):
                if solution.y[1][time_step] < 0:
                    if direction_check == 1:
                        pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step - 1]
                        initial_time = solution.t[time_step - 1]
                        crossing_found = True
                        break
                    else:
                        pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step]
```

```
                        initial_time = solution.t[time_step]
                        crossing_found = True
                        break

        if pos_y < 0 and not crossing_found:
            for time_step in range(num_eval_points):
                if solution.y[1][time_step] > 0:
                    if direction_check == 1:
                        pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step - 1]
                        initial_time = solution.t[time_step - 1]
                        break
                    else:
                        pos_x, pos_y, vel_x, vel_y = solution.y[:, time_step]
                        initial_time = solution.t[time_step]
                        break

        # Ajuste de Newton para refinar el cruce
        convergence_reached = False
        iterations = 0
        current_step_time = 0

        while not convergence_reached and iterations < 1000:
            solution = solve_ivp(
                fun=lambda t, var: dynamical_system(t, var, mu),
                t_span=integration_span,
                y0=[pos_x, pos_y, vel_x, vel_y],
                t_eval=np.array([current_step_time]),
                rtol=3e-14,
                atol=1e-14
            )

            t_adjusted = current_step_time - (g_sigma(solution.y) /
                                              (np.dot(grad_g_sigma(solution.y), dynamical_system(0,
                                                  solution.y, mu))))[0]
            t_adjusted = np.real(t_adjusted) % (2 * np.pi)

            if np.abs(current_step_time - t_adjusted) < 1e-14:
                convergence_reached = True

            current_step_time = t_adjusted
            iterations += 1

            if iterations > 2000:
                print('Error: Too many iterations')

        solution = solve_ivp(
            fun=lambda t, var: dynamical_system(t, var, mu),
            t_span=integration_span,
            y0=[pos_x, pos_y, vel_x, vel_y],
            t_eval=np.array([current_step_time]),
            rtol=3e-14,
            atol=1e-14
        )

        pos_x, pos_y, vel_x, vel_y = solution.y[:, 0]
        accumulated_time += current_step_time + initial_time

    return [solution.y, accumulated_time]

# Definir el rango de mu
# mu_range = [0.005630, 0.005640, 0.005650]

# mu_range = [0.020000
# ,0.020100
# ,0.020200 ]
# mu_range = [0.009250,
# 0.009260 ,
# 0.009270 ]


mu_range = [0.1]


# mu_range = [0.304]

# Inicializar listas para almacenar resultados
x_list = []
y_list = []
s = 1e-6 # for smaller values i get an error
```

```
i = 0
for mu in mu_range:
    # Encontrar el vector propio para el valor propio positivo
    for i in range(len(get_eigenvalues(mu))):
        if np.isreal(get_eigenvalues(mu)[i]) and get_eigenvalues(mu)[i] > 0:
            v0 = get_eigenvectors(mu)[:, i]

    # Calcular el punto inicial x0
    L3_value = L3(mu)
    x0 = np.array([L3_value, 0, 0, 0]) +  s* v0
    if x0[1] > 0:
        x0 = np.array([L3_value, 0, 0, 0]) - s * v0

    # Calcular el mapa de Poincar  y la soluci n del sistema
    a = poincare_map(x0, 1, mu)
    sol = solve_ivp(
        fun=lambda t, y: dynamical_system(t, y, mu),
        t_span=[0, a[1]],
        y0=x0,
        t_eval=np.linspace(0, a[1], 1000),
        rtol=3e-14,
        atol=1e-14
    )

    # Almacenar los resultados en las listas
    x_list.append(sol.y[0])
    y_list.append(sol.y[1])


# Graficar los resultados en un solo plot
plt.figure(figsize=(10, 6))
for j in range(len(mu_range)):
    plt.plot(x_list[j], y_list[j], label=f'$\mu$={mu_range[j]}')

plt.xlabel(r"$x$",fontsize = 14)
plt.ylabel(r"$y$",fontsize = 14)
# plt.xlim(-1.15, -0.95)
# plt.ylim(-0.04, 0.02)
plt.title("Poincar  Map Trajectories for different $\mu$ values")

plt.grid(True)
# Calcular el punto L2 para mu = 0.0202

# Coordenadas del segundo primario P2 para mu = 0.0202
mu_target = 0.0202
P2_x = 1 - mu_target
P2_y = 0  # El segundo primario est  en y = 0

# Dibujar el punto P2 en el gr fico
plt.scatter(-P2_x, -P2_y, color='red', marker='o', s=100, label=r'$P_2$ for $\mu=0.0202$')
plt.legend()
plt.show()

# Using the poincar  map we want to compute x' for a given x
def F(mu, x, C):
    # x as the initial condition
    #first find the yp to pass as the initial state
    state = y_p(mu, x, C)
    solution = poincare_map(state, 2, mu)
    # sol.y y de aqui quiero la tercera columna x' y el primer valor incial de x'
    x_prima = solution[0][2][0]

    return x_prima

# PQ ES X' = 0 PARA N = 1???
C1 = 2.1
# C2 = 3.189
C2 = 2.99
mu = 0.1
L3_point = L3(mu)
print(L3_point)


#%%
# initial bisection points
x2 = 1.393505608247538
x1 = x2
fx1 = F(mu, x1, C2)
fx2 = F(mu, x2, C2)
```

```
print(x1, fx1)

# he inicializado los dos puntos en el mismo sitio, voy a desplazar uno a la derecha para ir
    iterando con mi m todo
# Limitamos el n mero de iteraciones para evitar bucles infinitos
max_iterations = 10000
for _ in range(max_iterations):
    # busco un cambio de signo en F(x)
    if (fx2 * fx1 <= 0):  # Condici n de salida
        break
    x1 = x2
    x2 = x2+ 1e-3
    fx1 = fx2
    fx2 = F(0.1, x2, C2)
    print(f"x1 = {x1}, x2 = {x2}, fx1 = {fx1}, fx2 = {fx2}")
else:
    print("Error: No se encontr  un cambio de signo en el intervalo dado.")

#%%
root = bisection_method(lambda x: F(0.1, x, C2), x1, x2)
print(root, F(0.1, root, C2))

# 1.0610139281023079 -4.119488789151957e-07

#%%
# # lo pongo para no volvero a correr
# root = 1.0610139281023079
root=  1.393505608247538
from rich.progress import Progress
# ITERATION FOR MULTPLE C, DELTA_C = 1E-4
num_points = int((C2 - C1) / 1e-3) + 1

# Crear un array equiespaciado con linspace
list_C = np.linspace(C2, C1, num_points)
mu = 0.1
# Cuantas orbitas habr
root_orbits = np.zeros(len(list_C))
root_orbits[0] = root
# Proceso principal con medidor de progreso
with Progress() as progress:
    task = progress.add_task("[cyan]Processing...", total=len(list_C)-1)

    for i in range(1, len(list_C)):
        C_actual = list_C[i]
        x1 = root_orbits[i-1]
        x2 = x1
        fx1 = F(mu, x1, C_actual)
        fx2 = fx1

        while fx2*fx1>0:
            x2 = x1
            fx2 = fx1
            # Para C menores me desplazo a x  m  s grandes
            x1 += 1e-3
            fx1 = F(mu, x1, C_actual)

        print(f"INTERVALO CONSEGUIDO: C_actual: {C_actual}, fx2:{ fx2}, fx1: {fx1}")
        nueva_raiz = bisection_method(lambda x: F(mu, x, C_actual), x1, x2)
        root_orbits[i] = nueva_raiz


        progress.update(task, advance=1)


#%%
with open("./results2.txt", "w") as file:
    file.write("roots_orbits\tc_list\n")  # Encabezado
    for root, c in zip(root_orbits, list_C):
        if root != 0:  # Solo guardar si el valor de root_orbits es distinto de 0
            file.write(f"{root}\t{c}\n")
     #%%
# Guardar list_C y root_orbits en un archivo .txt
with open("results2.txt", "w") as file:
    file.write("C values and their corresponding root orbits:\n")
    file.write("C_value\tRoot_orbit\n")
    for C, root in zip(list_C, root_orbits):
        file.write(f"{C:.6f}\t{root:.6f}\n")

print("Results saved to 'results2.txt'")
```

```
import os

print(f"File saved at: {os.path.abspath('results2.txt')}")
  #%%

import os
import numpy as np

# Cambiar al directorio donde se encuentra el script
os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Inicializar listas para almacenar los datos combinados
C_total = []
roots_total = []

# Funci n para leer un archivo y procesar los datos
def read_file(file_path, reverse_columns=False, skip_header=False):
    C_values = []
    roots_values = []
    with open(file_path, "r") as file:
        if skip_header:
            next(file)  # Saltar la primera l nea si es un encabezado
        for line in file:
            try:
                # Leer y convertir valores dependiendo del orden
                val1, val2 = map(float, line.split())
                if reverse_columns:
                    C_values.append(val1)
                    roots_values.append(val2)
                else:
                    roots_values.append(val1)
                    C_values.append(val2)
            except ValueError:
                # Saltar l neas mal formateadas
                print(f"L nea inv lida en {file_path}: {line.strip()}")
    return C_values, roots_values

# Leer el primer archivo (saltando el encabezado)
C1, roots1 = read_file("results.txt", reverse_columns=False, skip_header=False)

# Leer el segundo archivo (columnas invertidas, sin encabezado)
C2, roots2 = read_file("results2.txt", reverse_columns=True)

# Combinar datos
C_total = np.array(C1 + C2)
roots_total = np.array(roots1 + roots2)

# # Imprimir resultados para verificar
# print("C_total:", C_total)
# print("roots_total:", roots_total)
#%%
import matplotlib.pyplot as plt

# Crear el gr fico C(roots)
plt.figure(figsize=(8, 6))
plt.plot(roots_total, C_total, linestyle='-', color='royalblue', label='C(roots)')

# Personalizar el gr fico
# plt.title("Plot of C(initial points x)")
plt.xlabel("x", fontsize = 16)
plt.ylabel("C", fontsize = 16)
plt.grid(True)
plt.legend()
plt.tight_layout()

# Mostrar el gr fico
plt.show()
#%%
import numpy as np

# Inicializar arrays
root_orbits = np.zeros(1000)  # Ajusta el tama o seg n sea necesario
list_C = np.zeros(1000)

# Leer el archivo para cargar valores procesados si existe
try:
    processed_roots = []
    processed_c_list = []

```

```
    with open("results.txt", "r") as file:
        next(file)  # Saltar el encabezado
        for line in file:
            # Manejo de errores por seguridad
            try:
                root, c = map(float, line.split())
                processed_roots.append(root)
                processed_c_list.append(c)
            except ValueError:
                print(f"Error procesando la l nea: {line.strip()}")

    # Convertir a numpy arrays y actualizar root_orbits y list_C
    num_processed = len(processed_roots)
    root_orbits[:num_processed] = np.array(processed_roots)
    list_C[:num_processed] = np.array(processed_c_list)

except FileNotFoundError:
    num_processed = 0  # Si el archivo no existe, empieza desde el principio

#%%
# Proceso principal con medidor de progreso
with Progress() as progress:
    task = progress.add_task("[cyan]Processing...", total=len(list_C) - num_processed)

    for i in range(num_processed, len(list_C)):
        C_actual = list_C[i]
        x1 = root_orbits[i - 1]
        x2 = x1
        fx1 = F(mu, x1, C_actual)
        fx2 = fx1

        # Encontrar intervalo para la ra z
        while fx2 * fx1 > 0:
            x2 = x1
            fx2 = fx1
            x1 += 1e-3
            fx1 = F(mu, x1, C_actual)

        print(f"INTERVALO CONSEGUIDO: C_actual: {C_actual}, fx2: {fx2}, fx1: {fx1}")
        nueva_raiz = bisection_method(lambda x: F(mu, x, C_actual), x1, x2)
        root_orbits[i] = nueva_raiz

        # Guardar en el archivo en cada iteraci n
        with open("./results.txt", "a") as file:
            file.write(f"{nueva_raiz}\t{C_actual}\n")

        progress.update(task, advance=1)


#%%
C =[3.15, 2.5, 2.1]

# Encontramos los  ndices  de los valores de C en C_total
indices = np.where(np.isin(C_total, C))[0]
x_list = []
y_list = []
# Creamos la lista x_iniciales con los valores correspondientes en root_orbits
x_iniciales = roots_total[indices]
mu = 0.1
i = 0
for c_values in C:
    x0 = x_iniciales[i]
    # print(x0)
    state = y_p(mu, x0, c_values)
    val = poincare_map(state, 2, mu)[1]*2
    print(f"val: {val}")
    sol = solve_ivp(
        fun=lambda t, y: dynamical_system(t, y, mu),
        t_span=[0, val],
        y0=state,
        t_eval=np.linspace(0, val, 1000),
        rtol=3e-14,
        atol=1e-14
        )
    x_list.append(sol.y[0])
    y_list.append(sol.y[1])
    i = i+1
#%%
# Graficar los resultados en un solo plot
```

```
plt.figure(figsize=(10, 6))
for j in range(len(C)):
    plt.plot(x_list[j], y_list[j], label = f"C = {C[j]}")

plt.xlabel(r"$x$",fontsize = 14)
plt.ylabel(r"$y$",fontsize = 14)
# plt.xlim(-1.15, -0.95)
# plt.ylim(-0.04, 0.02)
plt.title("Poincar  Map Trajectories for different $C$ values")

plt.grid(True)

plt.scatter(-(1-0.1), 0, color = "gray", label = "P2", s = 80)
plt.scatter(0.1, 0, color = "brown", label ="P1", s =80)
# Coordenadas del segundo primario P2 para mu = 0.0202
mu_target = 0.1
# P2_x = 1 - mu_target
# P2_y = 0  # El segundo primario est  en y = 0

# Dibujar el punto P2 en el gr fico
plt.scatter(L3_value, 0, color='red', marker='o', s=100, label=r'$L_3$')
plt.legend()
plt.show()
#%%
import os
import numpy as np
import matplotlib.pyplot as plt

# Cambiar autom ticamente al directorio del script
script_dir = os.path.dirname(os.path.abspath(__file__))  # Ruta del script
os.chdir(script_dir)  # Cambiar directorio de trabajo al del script
print(f"Directorio cambiado a: {os.getcwd()}")

# Nombre del archivo .txt
txt_file = "variables_datos.txt"

try:
    # Leer datos desde el archivo .txt, separador = tabulador (\t)
    data = np.loadtxt(txt_file, delimiter='\t')

    # Dividir los datos en las columnas correspondientes
    C_t = data[:, 0]  # Primera columna
    x_t = data[:, 1]  # Segunda columna
    T = data[:, 2]    # Tercera columna

    print(f"Archivo '{txt_file}' le do correctamente y procesado.")
except Exception as e:
    print(f"Error al leer el archivo: {e}")
    C_t, x_t, T = None, None, None

# Generar el gr fico de T en funci n de x_t
if C_t is not None and x_t is not None and T is not None:
    plt.figure(figsize=(8, 6))
    plt.plot(x_t, T,  linestyle='--', label='T(x)', color = "royalblue")
    plt.xlabel('x', fontsize = 16)
    plt.ylabel('T', fontsize = 16)
    # plt.title('Gr fico de T en funci n de x_t')
    plt.grid(True)
    # plt.legend()

plt.axhline(y = 5.83, color = "brown", label = "T = 5.83", linestyle = "--")
plt.legend()
plt.show()
```

ASSIGNMENT 11: SYMMETRIC LPO. STABILITY AND COMPUTATION OF THEIR INVARIANT MANIFOLDS.

Pol Navarro Pérez

December 26, 2024

Abstract

In this work, we compute and analyze periodic orbits (PO) and their stable and unstable manifolds within a dynamical system framework. Using Newton's method, we refine initial conditions to satisfy homoclinic symmetry conditions for a fixed period T. Our results show consistent behavior between different methods for $C(x)$ and $T(x)$, and demonstrate the symmetry of the manifolds W^{s+}, W^{s-}, W^{u+}, and W^{u-}. The conservation of the Jacobi constant C_J across all branches is confirmed, with variations observed to be negligible.

PART 1

In this work, we have used a fixed value of the period T to compute the corresponding families of periodic orbits. Unlike we have done in the previous assignment, now we compute the orbit until $\frac{T}{2}$, being T one of the periods found for instance. Then we have to find the corresponding initial point for that period such that $y(\frac{T}{2}, x_0, y_0) = 0$ and $x'(\frac{T}{2}, x_0, y_0) = 0$, conditions to be a homoclinic orbit by the symmetric of the solutions.

This procedure's implementation consists of using Newton's method to find a good approximation of the roots for the functions x' and y, as stated before. We start at an initial point displaced a little bit from the already known solution for a period, then we will have to iterate updating this initial state. Using the Taylor expansion, as discussed in theory sessions we get the system to implement:

$$\begin{pmatrix} \frac{\partial y}{\partial x} & \frac{\partial y}{\partial \dot{x}} \\ \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial \dot{x}} \end{pmatrix} \begin{pmatrix} \Delta x_0^{(0)} \\ \Delta \dot{y}_0^{(0)} \end{pmatrix} = \begin{pmatrix} -y\left(\frac{T}{2}, x^{(0)}, \dot{y}^{(0)}\right) \\ -x'\left(\frac{T}{2}, x^{(0)}, \dot{y}^{(0)}\right) \end{pmatrix} \tag{1}$$

Then we consider the new initial state $(x_0^1 = x_0^0 + \Delta x^0, y_0'^1 = y_0'^0 + \Delta y_0'^0)$. Iterate Newton's method until functions y and x' are less than a certain tolerance, for instance, 10^{-13}.

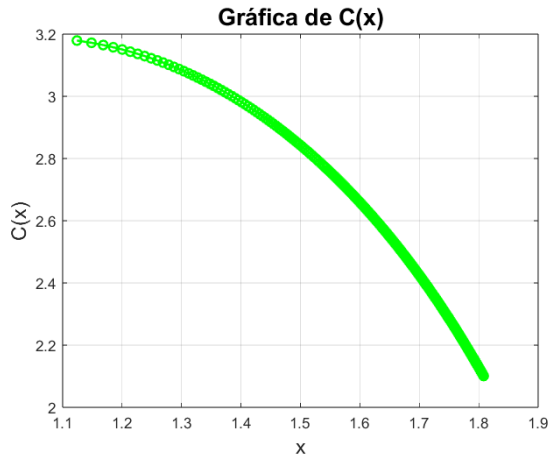
Eventually, we can compare the results obtained from using the values with this method and the ones from the previous assignment, where the period was obtained afterward.

Figure 1 shows, we have obtained the same initial states corresponding to the family of PO using different methods. There is also the same convergence in the period.

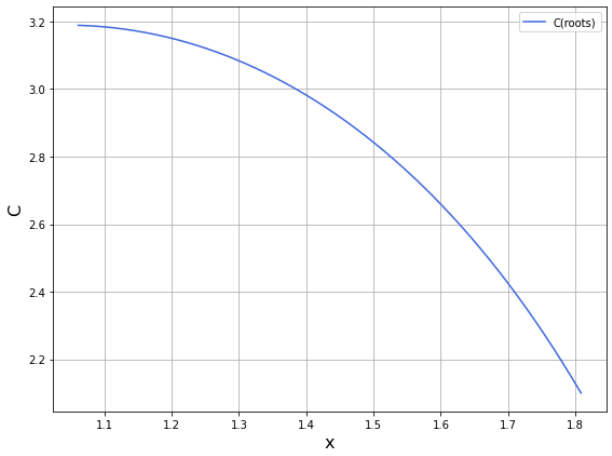
Now let us try again the same procedure but this time with $\mu = 0.01$ and an initial state such that $[x = 1.033366313746765, y = 0, x' = 0, y' = -0.05849376854515592]$ and a period $\frac{T}{2} = 3.114802556760205$. Then, we have to iterate for other periods using $\Delta T = 0.0001$.

Again, Figure 2 illustrates that the shape and consequently, the dependency remains consistent for a different value of μ. Nevertheless, the corresponding values of T and C differ, as does the T value toward which the family of periodic orbits converges. In this case, we have taken an initial value $x = 1.033366313746765$, $y' = -.05849376854515592$ and $T/2 = 3.114802556760205$ then iterated multiple periods to find the corresponding x for the PO.

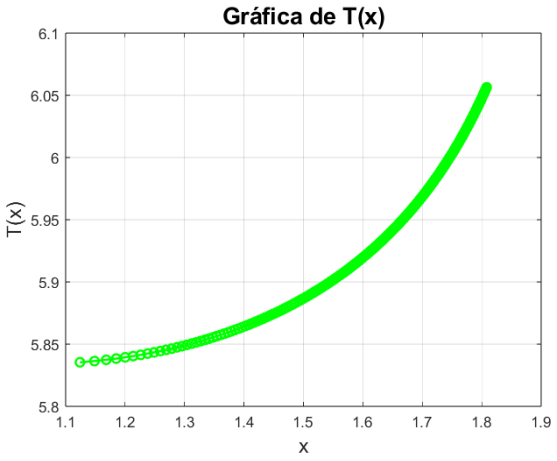
We are asked also to compute the stability parameter k, taking the whole period and plotting the curve (x,k). Figure 3 shows the dependency of k with x. We know that if k ∈ (-2,2) then the PO is linearly stable and if not it is unstable. For $\mu = 0.1$ the stability parameter is always greater than 2, thus it is unstable for every x. Nevertheless, for $\mu = 0.01$ for smaller x, it is greater than 2 but after some x, around $x = 1.9$, it gets smaller. This means it is unstable until some x value, when the orbit becomes stable. Remark that in between, for $k = 2$, there is a critical orbit for which we can not say anything about its behavior.



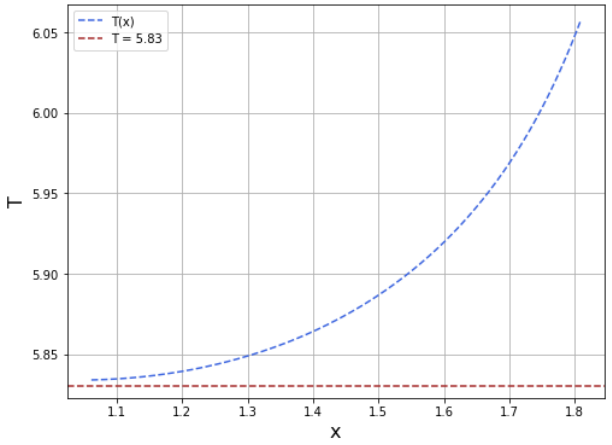
ass11



ass10

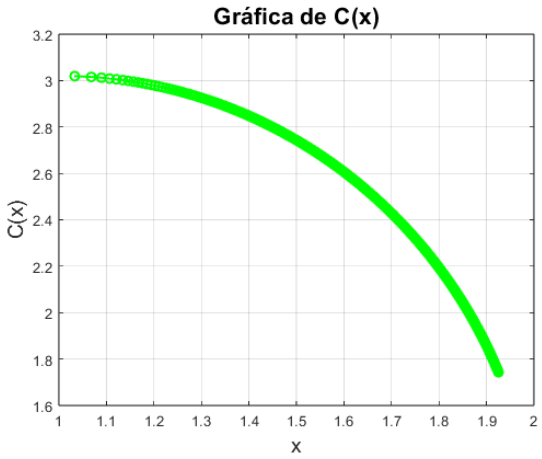


ass11

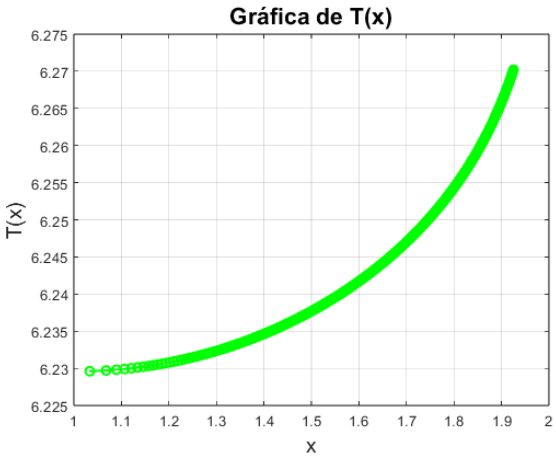


ass10

Figure 1: Comparation of C(x) and T(x) for both methods.

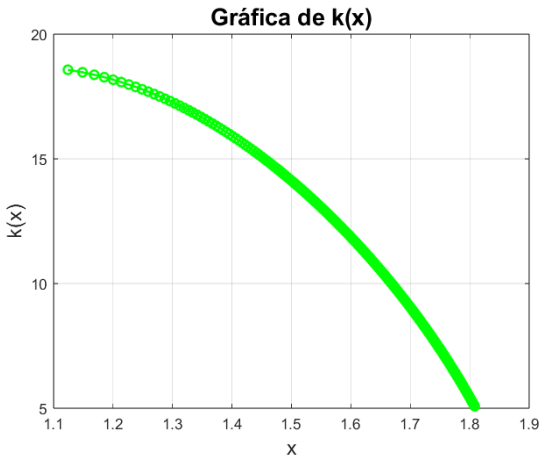


C(x) for $\mu = 0.01$

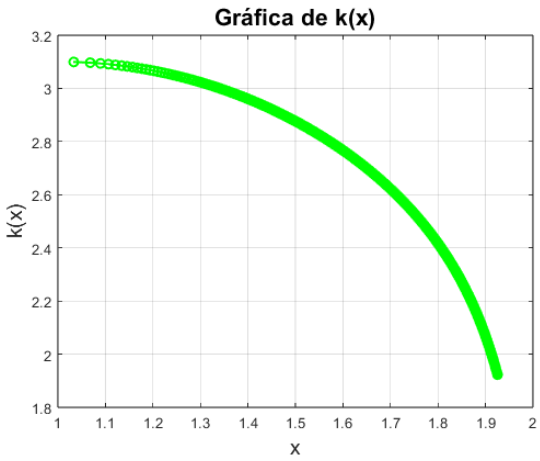


T(x) for $\mu = 0.01$

Figure 2: Comparation of C(x) and T(x) for $\mu = 0.01$.



Stability parameter k for $\mu = 0.1$.



Stability parameter k for $\mu = 0.01$.

Figure 3: Stability parameter depending on the x for different μ values.

PART 2

The next part is about implementing a similar procedure, given an initial condition x and period T compute the 2d unstable and stable invariant manifold.

Before explaining the computational part let's explain the main goal. We want to plot the unstable and stable manifold, but this time for a LPO. This means, given a LPO, we want the orbits that are backward or forward in time to tend to our original orbit.

The procedure can be done using two methods, as seen in theory sessions. In this case, we have implemented the one that parameterizes the PO and the eigenvectors, depending on the parameter time t. The lemma that underpins this procedure is the following:

Lemma: Given $\lambda \in \text{Spec}(D\phi_T(x_0))$, with $\vec{v_u}$ a unitary eigenvector, the function

$$v(t) = \lambda^{-\frac{t}{T}} D\phi_t(x_0) v_u$$

satisfies that:

i) $v(t)$ is T-periodic, i.e., $v(t+T) = v(t)$,

ii) $D\phi_T(\psi(t))v(t) = \lambda v(t+T) = \lambda v(t)$.

This has already been proved in theory sessions.

Using this approach we can implement the corresponding procedure. We begin with an initial state (x_0, y_0, x', y', T) and compute a point that is in the PO to study. Afterward, we get a N number of initial points that will generate their own orbits. These are $p_j = x_0 \pm s\lambda^{-\frac{t_j}{T}} v_{u/s}$.

Remark we have 2 branches for each manifold to compute, consequently we must be careful using $\pm s$ and the correct eigenvector and eigenvalue.

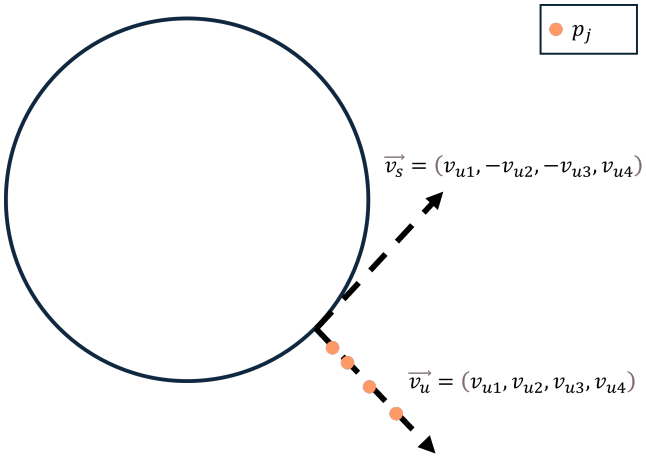
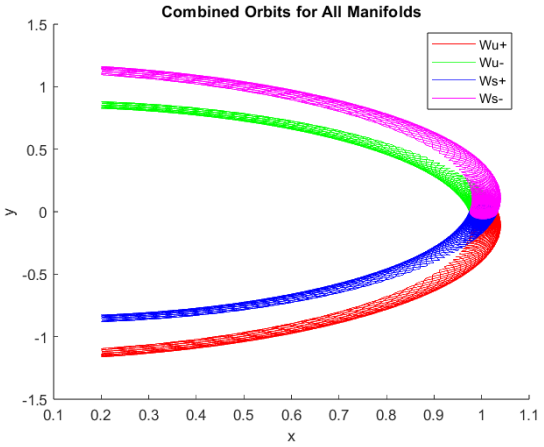
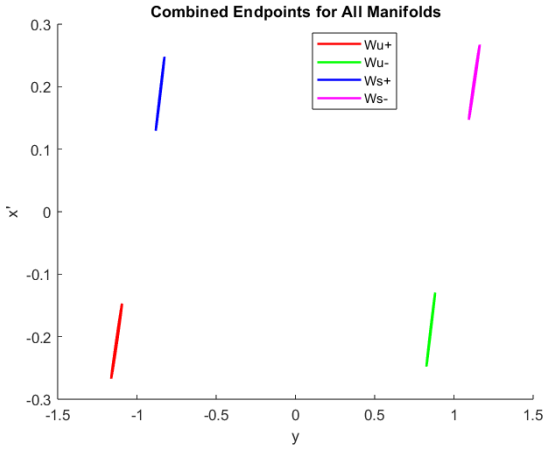


Figure 4: Scheme of the initial points considered

Figure 4 represents the initial points taken for the unstable manifold. The stable one is easily computed using the same procedure but with its corresponding eigenvector and eigenvalues.

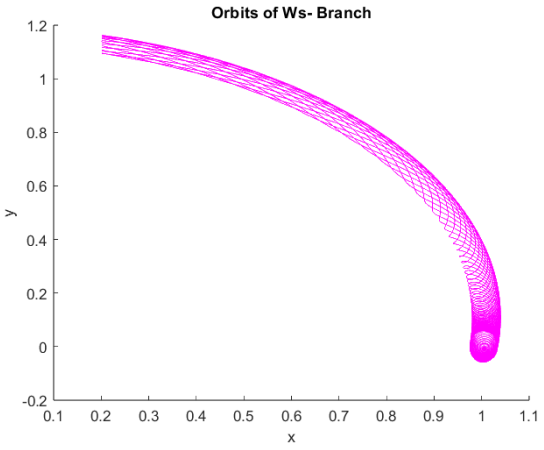


W^{u+}, W^{u-}, W^{s+} and W^{s-} (x,y) projection for N = 15 points.

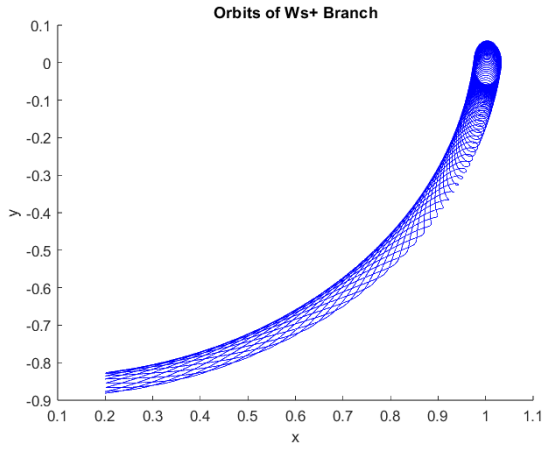


Intersection of the manifolds with the $\sum$ section (y, x') projection.

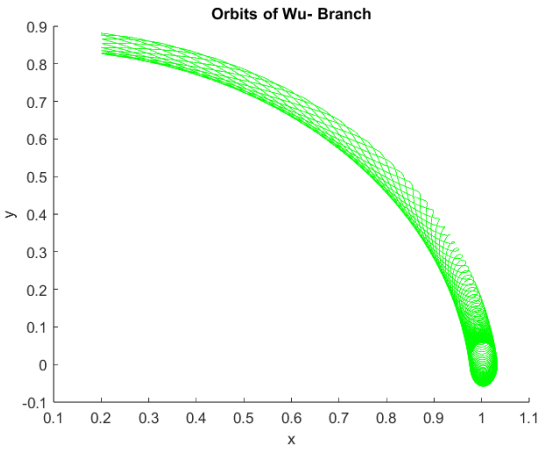
Figure 5: Stable and unstable manifolds. Poincaré section is $\sum : x - 0.2$.



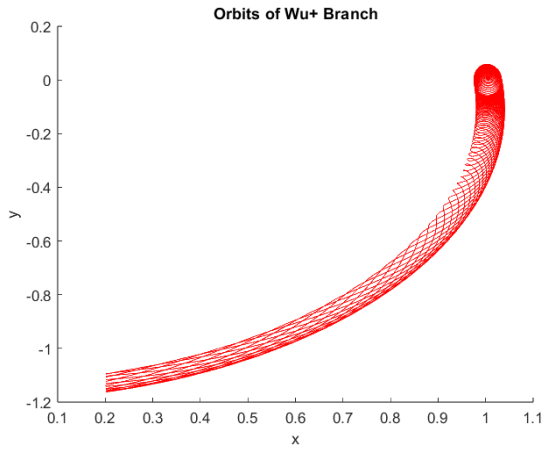
W^{s-} manifold.



W^{s+} manifold.



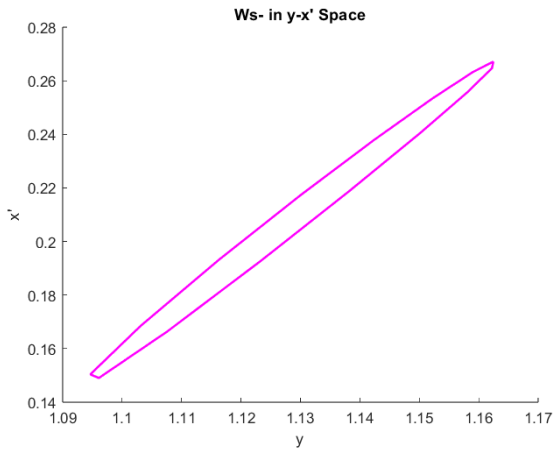
W^{u-} manifold.



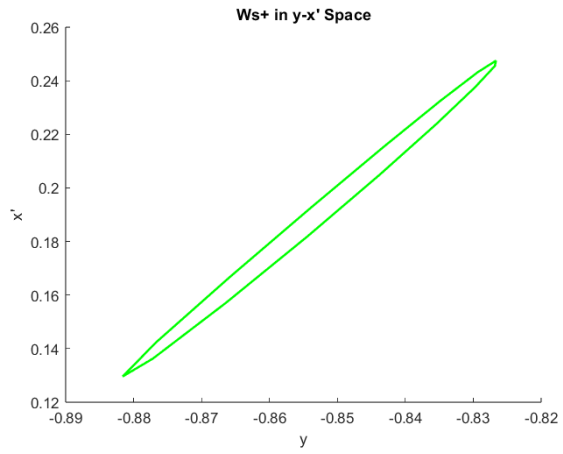
W^{u+} manifold.

Figure 6: Individual manifolds (x,y) projection.

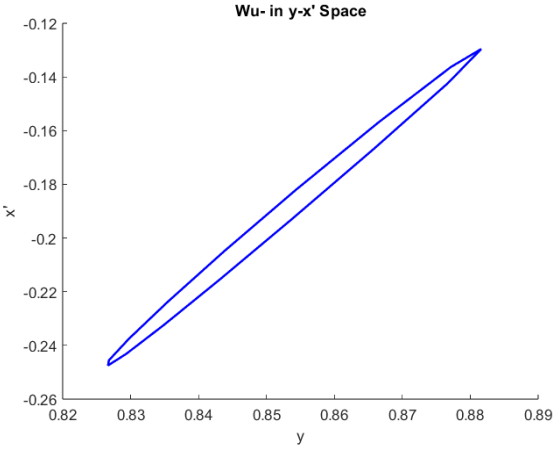
The results of Figures 5, 6 and 7 show the expected results. There is a clear symmetry between W^{s-} and W^{u+} and on the other hand between W^{s+} and W^{u-}. As we observe there is some symmetry on these plots. There is no need on each individual point to be symmetric to the commented symmetric branch of the manifold, but the entire orbits are indeed. As a final comment and Figure 8 shows, the Jacobi constant is conserved along the integration. The variation is observed to be negligible, within a scale for instance until 10^{-9}, and the same happens for the other branches.



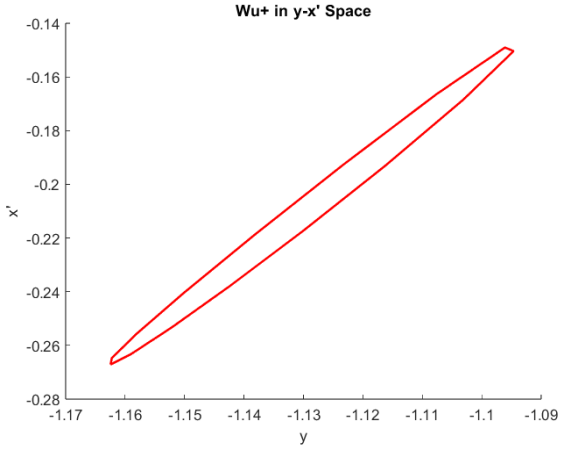
W^{s-} intersection with $\sum : x - 0.2$ (x',y) projection.



W^{s+} intersection $\sum$ (x',y) projection.



W^{u-} intersection $\sum$ (x',y) projection.



W^{u+} intersection $\sum$ (x',y) projection.

Figure 7: Individual $\sum$ intersections (x',y) projections for each manifold branch. The poincaré section is $\sum : x = 0.2$.

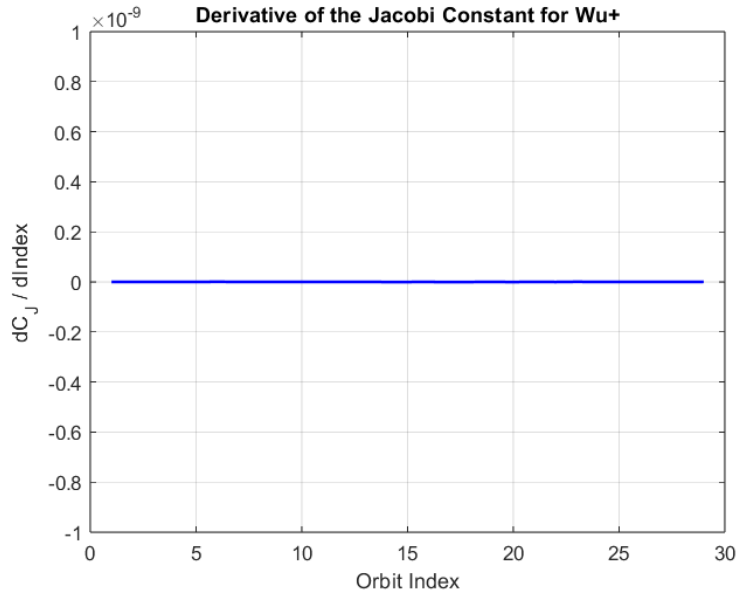


Figure 8: Variation of the Jacobi constant C_J across all points of the orbit for initial conditions along the unstable manifold W_u^+.

CODE

Listing 1: code

```
format long;
close all;
clear all;
clc

tol=1e-13;
s=1e-6;
max_counter=1000;
xmu = 0.1;
ysign=-1;
delta=1e-3;
h=1e-3;

struct = load('data.mat');
data = struct.data;
[L3,C3,eigL3,veigL3]=RTBP_eq_points(xmu,tol);
x0=L3(1);

T_i=data(100,3):0.001:round(data(end,3),3);
size_T=size(T_i);
x0=1.123959614214410;
y0_prime=-0.175736293599871;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);  % ODE solver options for accuracy
orbits_data=zeros(size_T(2),5);
counter=1;

for T=T_i
    [x,y_prime]=newton_variational(x0,y0_prime,xmu,T,tol);
    [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T], [x 0 0
        y_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
    r1 = sqrt((x-xmu)^2);
    r2 = sqrt((x-xmu+1)^2);
    C=2*Omega([x,0],xmu,r1,r2) - y_prime^2;
    k=X(end,5)+X(end,10)+X(end,15)+X(end,20)-2;
    orbits_data(counter,1)=C;
    orbits_data(counter,2)=x;
    orbits_data(counter,3)=y_prime;
    orbits_data(counter,4)=T;
    orbits_data(counter,5)=k;
    counter=counter+1;
    % Imprime los valores actuales de T y x
    fprintf('T: %.6f, x: %.6f\n', T, x);

    x0=x;
    y0_prime=y_prime;
end

% Extrae los datos necesarios desde la matriz orbits_data
x_values = orbits_data(:, 2); % x est  en la segunda columna de orbits_data
T_values = orbits_data(:, 4); % T est  en la cuarta columna de orbits_data
C_values = orbits_data(:, 1); % C est  en la primera columna de orbits_data
k_values = orbits_data(:, 5); % k est  en la quinta columna de orbits_data

% Grafica T(x)
figure;
plot(x_values, T_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('T(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de T(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;
saveas(gcf, 'Grafica_Tx.png'); % Guarda como PNG

% Grafica C(x)
figure;
plot(x_values, C_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
```

```
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('C(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de C(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;
saveas(gcf, 'Grafica_Cx.png'); % Guarda como PNG

% Grafica k(x)
figure;
plot(x_values, k_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('k(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de k(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;
saveas(gcf, 'Grafica_kx.png'); % Guarda como PNG

%%
xmu = 0.01;
T_i=3.114802556760205*2:0.0001:3.114802556760205*2+0.0001*1000;
size_T=size(T_i);
x0=1.033366313746765;
y0_prime=-.05849376854515592;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);  % ODE solver options for accuracy
orbits_data=zeros(size_T(2),5);
counter=1;

for T=T_i
    [x,y_prime]=newton_variational(x0,y0_prime,xmu,T,tol);
    [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T], [x 0 0
        y_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
    r1 = sqrt((x-xmu)^2);
    r2 = sqrt((x-xmu+1)^2);
    C=2*Omega([x,0],xmu,r1,r2) - y_prime^2;
    k=X(end,5)+X(end,10)+X(end,15)+X(end,20)-2;
    orbits_data(counter,1)=C;
    orbits_data(counter,2)=x;
    orbits_data(counter,3)=y_prime;
    orbits_data(counter,4)=T;
    orbits_data(counter,5)=k;
    counter=counter+1;
    % Imprime los valores actuales de T y x
    fprintf('T: %.6f, x: %.6f\n', T, x);

    x0=x;
    y0_prime=y_prime;
end
%%
% Extrae los datos necesarios desde la matriz orbits_data
x_values = orbits_data(1:407, 2); % x est  en la segunda columna de orbits_data
T_values = orbits_data(1:407, 4); % T est  en la cuarta columna de orbits_data
C_values = orbits_data(1:407, 1); % C est  en la primera columna de orbits_data
k_values = orbits_data(1:407, 5); % k est  en la quinta columna de orbits_data

% Grafica T(x)
figure;
plot(x_values, T_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('T(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de T(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;
saveas(gcf, 'Grafica_Tx.png'); % Guarda como PNG

% Grafica C(x)
figure;
plot(x_values, C_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('C(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de C(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;
saveas(gcf, 'Grafica_Cx.png'); % Guarda como PNG
```

```
133
134 % Grafica k(x)
135 figure;
136 plot(x_values, k_values, 'g-o', 'LineWidth', 1.5); % L nea verde con marcadores
137 xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
138 ylabel('k(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
139 title('Gr fica de k(x)', 'FontSize', 16); % T tulo con fuente m s grande
140 grid on;
141 saveas(gcf, 'Grafica_kx.png'); % Guarda como PNG
142
143 %% STATE 2
144 tol=1e-13;
145 s=1e-6;
146 max_counter=10000;
147 xmu = 0.01;
148 ysign=-1;
149 delta=1e-3;
150 h=1e-3;
151
152 T_i=(2*3.114802556760205):0.0001:2*3.125;
153 size_T=size(T_i);
154 x0=1.033366313746765;
155 y0_prime=-0.05849376854515592;
156 options = odeset('RelTol',1e-13,'AbsTol',1e-13);  % ODE solver options for accuracy
157 orbits_data=zeros(size_T(2),5);
158 counter=1;
159
160
161
162
163 el = 0;
164 for T = T_i
165     % Imprimir valores antes de llamar a newton_variational
166     fprintf('Antes de Newton:\n x0 = %.8f\n y0_prime = %.8f\n\n, T = %.6f\n\n', x0,
        y0_prime, T);
167
168     [x_new, y_prime] = newton_variational(x0, y0_prime, xmu, T, tol);
169
170     [~, X] = ode45(@(t, y) f_variational_eq(t, y, xmu, 1), [0, T], [x_new, 0, 0, y_prime,
        1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], options);
171
172     r1 = sqrt((x_new - xmu)^2);
173     r2 = sqrt((x_new - xmu + 1)^2);
174     C = 2 * Omega([x_new, 0], xmu, r1, r2) - y_prime^2;
175     k = X(end, 5) + X(end, 10) + X(end, 15) + X(end, 20) - 2;
176
177     orbits_data(counter, 1) = C;
178     orbits_data(counter, 2) = x_new;
179     orbits_data(counter, 3) = y_prime;
180     orbits_data(counter, 4) = T;
181     orbits_data(counter, 5) = k;
182     counter = counter + 1;
183
184     % Incrementar el contador de iteraciones
185     el = el + 1;
186
187     % Actualizar valores
188     x0 = x_new;
189     y0_prime = y_prime;
190
191     % Imprimir valores despu s de la actualizaci n
192     fprintf('Despu s de actualizar:\n x0 = %.8f\n y0_prime = %.8f\n\n', x0, y0_prime);
193
194     % Salir del bucle si se cumplen las condiciones
195
196 end
197
198
199 %%
```

```
% Extrae los datos necesarios desde la matriz orbits_data
x_values = orbits_data(:, 2); % x est  en la segunda columna de orbits_data
T_values = orbits_data(:, 4); % T est  en la cuarta columna de orbits_data
C_values = orbits_data(:, 1); % C est  en la primera columna de orbits_data
k_values = orbits_data(:, 5); % k est  en la quinta columna de orbits_data

% Grafica T(x)
figure;
plot(x_values, T_values, 'm-o', 'LineWidth', 1.5); % L nea magenta con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('T(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de T(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;


% Grafica C(x)
figure;
plot(x_values, C_values, 'm-o', 'LineWidth', 1.5); % L nea magenta con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('C(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de C(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;


% Grafica k(x)
figure;
plot(x_values, k_values, 'm-o', 'LineWidth', 1.5); % L nea magenta con marcadores
xlabel('x', 'FontSize', 14); % Etiqueta del eje x m s grande
ylabel('k(x)', 'FontSize', 14); % Etiqueta del eje y m s grande
title('Gr fica de k(x)', 'FontSize', 16); % T tulo con fuente m s grande
grid on;


%% PART 2


T_i = 2 * 3.114802556760205;
xmu = 0.01;
h = 1e-3;
ysign = -1;
delta=1e-3;
tol=1e-13;

% INITIAL CONDITIONS
T = 6.22960511352041;
x0 = 1.0333662809371982;
y0_prime = -0.05849370327223067;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
counter=1;
delta=0.0001:0.0001:0.001;
size_delta=size(delta);
matrix=zeros(size_delta(2),8);

% Imprimir valores
fprintf('T = %.15f\n', T);
fprintf('x0 = %.15f\n', x0);
fprintf('y0_prime = %.15f\n', y0_prime);

for deltaS = delta
    [x,y_prime]=newton_variational(x0,y0_prime,xmu,T,tol);
    [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T], [x 0 0
        y_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
    DG = [X(end,5)      X(end,6)  X(end,7)  X(end,8);
            X(end,9)  X(end,10) X(end,11) X(end,12);
            X(end,13) X(end,14) X(end,15) X(end,16);
            X(end,17) X(end,18) X(end,19) X(end,20)];


    [veps,vaps]=eig(DG);
```

```
    tol2=1e-3;
    unstable_vap=0;
    unstable_vep=0;
    stable_vap=0;
    stable_vep=0;
    for i=1:4
        if real(vaps(i,i))>1+tol2
            unstable_vap=real(vaps(i,i));
            unstable_vep=sign(veps(2,i))*real(veps(:,i));
        end
        if real(vaps(i,i))<1-tol2
            stable_vap=real(vaps(i,i));
            stable_vep=sign(veps(2,i))*real(veps(:,i));
        end
    end
    s=1e-6+deltaS;
    x0_unstable_plus=[x0 0 0 y0_prime]'+s*unstable_vep;
    x0_unstable_minus=[x0 0 0 y0_prime]'-s*unstable_vep;
    x0_stable_plus=[x0 0 0 y0_prime]'+s*stable_vep;
    x0_stable_minus=[x0 0 0 y0_prime]'-s*stable_vep;

    [cross_points1,total_time1,cross_times1,orbit1]=poincare(1,x0_unstable_plus',xmu,1,h,tol);
    [cross_points2,total_time2,cross_times2,orbit2]=poincare(1,x0_unstable_minus',xmu,1,h,tol);
    [cross_points3,total_time3,cross_times3,orbit3]=poincare(1,x0_stable_plus',xmu,-1,h,tol);
    [cross_points4,total_time4,cross_times4,orbit4]=poincare(1,x0_stable_minus',xmu,-1,h,tol);

    figure(1)
    hold on;
    plot(orbit1(1,:),orbit1(2,:),"red")
    legend('Wu+')
    hold off;
    figure(2)
    hold on;
    plot(orbit2(1,:),orbit2(2,:),"blue")
    legend('Wu-')
    hold off;
    figure(3)
    hold on;
    plot(orbit3(1,:),orbit3(2,:),"green")
    legend('Ws+')
    hold off;
    figure(4)
    hold on;
    plot(orbit4(1,:),orbit4(2,:),"magenta")
    legend('Ws-')
    hold off;

    figure(5)
    hold on;
    plot(orbit1(1,:),orbit1(2,:),"red")
    hold on;
    plot(orbit2(1,:),orbit2(2,:),"blue")
    hold on;
    plot(orbit3(1,:),orbit3(2,:),"green")
    hold on;
    plot(orbit4(1,:),orbit4(2,:),"magenta")
    legend('Wu+','Wu-','Ws+','Ws-')
    hold off;

    matrix(counter,:)=[orbit1(2,end),orbit1(3,end),orbit2(2,end),orbit2(3,end),orbit3(2,end),orbit
    counter=counter+1;
end
    figure(6)
    hold on
    plot(matrix(:,1),matrix(:,2),"red")
    hold on
    plot(matrix(:,3),matrix(:,4),"blue")
    hold on;
    plot(matrix(:,5),matrix(:,6),"green")
```

```
    hold on;
    plot(matrix(:,7),matrix(:,8),"magenta")
    legend('Wu+','Wu-','Ws+','Ws-')
    xlabel('y')
    ylabel('x''')
    %%
    {
    figure(7)
    plot(matrix(:,1),matrix(:,2),"red")
    legend('Wu+')
    xlabel('y')
    ylabel('x''')
    }

%%
T_i = 2 * 3.114802556760205;
xmu = 0.01;
h = 1e-3;
ysign = -1;
delta=1e-3;
tol=1e-13;

% INITIAL CONDITIONS
T = 6.22960511352041;
N = 1e-3; % Step size for t
time_points = linspace(0, T, floor(T / N) + 1); % Creates the interval

x0 = 1.0333662809371982;
y0_prime = -0.05849370327223067;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
counter=1;
% delta=0.0001:0.0001:0.001;
% size_delta=size(delta);
% Parameters (Ensure x0, y0_prime, xmu, T, and tol are defined)

[x,y_prime]=newton_variational(x0,y0_prime,xmu,T,tol);
[~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T], [x 0 0
    y_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
DG = [X(end,5)      X(end,6)  X(end,7)  X(end,8);
        X(end,9)  X(end,10) X(end,11) X(end,12);
        X(end,13) X(end,14) X(end,15) X(end,16);
        X(end,17) X(end,18) X(end,19) X(end,20)];


[veps,vaps]=eig(DG);
tol2=1e-3;
unstable_vap=0;
unstable_vep=0;
stable_vap=0;
stable_vep=0;
for i=1:4
    if real(vaps(i,i))>1+tol2
        unstable_vap=real(vaps(i,i));
        unstable_vep=-sign(veps(2,i))*real(veps(:,i));
    end
    if real(vaps(i,i))<1-tol2
        stable_vap=real(vaps(i,i));
        stable_vep=-sign(veps(2,i))*real(veps(:,i));
    end
end
% s=1e-6+deltaS;
% Print results
fprintf('Unstable Eigenvalue: %.15f\n', unstable_vap);
fprintf('Unstable Eigenvector:\n');
disp(unstable_vep);

fprintf('Stable Eigenvalue: %.15f\n', stable_vap);
fprintf('Stable Eigenvector:\n');
disp(stable_vep);
```

```
405
406
407
408
409
410
411
412 % Parameters
413 N_points = 30;  % Number of points along each direction
414 s = 1e-6;  % Scaling factor
415 x_initial = [x, 0, 0, y_prime]';  % Starting point of the periodic orbit
416 T = 6.22960511352041;  % Period of the orbit
417 options = odeset('RelTol', 1e-10, 'AbsTol', 1e-10);  % ODE solver options
418
419 % Storage for endpoints
420 endpoints = zeros(N_points, 8);  % [Wu+ y, Wu+ x', Wu- y, Wu- x', Ws+ y, Ws+ x', Ws- y,
        Ws- x']
421
422 % Plot 1: All endpoints together
423 figure(1);
424 hold on;
425 title('Endpoints for All Manifolds');
426 xlabel('y'); ylabel('x''');
427
428 % Individual Figures for Orbits
429 figure(2); hold on; title('Orbits of Wu+ Branch'); xlabel('x'); ylabel('y');
430 figure(3); hold on; title('Orbits of Wu- Branch'); xlabel('x'); ylabel('y');
431 figure(4); hold on; title('Orbits of Ws+ Branch'); xlabel('x'); ylabel('y');
432 figure(5); hold on; title('Orbits of Ws- Branch'); xlabel('x'); ylabel('y');
433
434 % Combined Plot for All Manifolds
435 figure(6);
436 hold on;
437 legend_handles = [];  % To store plot handles for the legend
438
439 % Loop for Wu+ (positive unstable branch)
440 figure(2);  % For individual plot of Wu+
441 for j = 1:N_points
442     factor = (j - 1) / N_points;  % j/N (from 0 to 1)
443     p_j = x_initial + s * unstable_vap^factor * unstable_vep;  % Initial point along
            unstable eigenvector
444
445     % Compute orbit for Wu+
446     [cross_points, total_time, cross_times, orbit] = poincare(1, p_j', xmu, 1, h, tol);
447
448     % Store endpoint
449     endpoints(j, 1:2) = [orbit(2, end), orbit(3, end)];
450
451     % Plot endpoints for Wu+
452     figure(1);
453     plot(orbit(2, end), orbit(3, end), 'ro');  % Red dots for Wu+ endpoints
454
455     % Plot individual Wu+
456     figure(2);
457     plot(orbit(1, :), orbit(2, :), 'red');
458
459     % Combined plot Wu+
460     figure(6);
461     if j == 1
462         handle = plot(orbit(1, :), orbit(2, :), 'red', 'DisplayName', 'Wu+');
463         legend_handles(1) = handle;  % Store handle for Wu+
464     else
465         plot(orbit(1, :), orbit(2, :), 'red');
466     end
467 end
468
469 % Loop for Wu- (negative unstable branch)
470 figure(3);  % For individual plot of Wu-
471 for j = 1:N_points
```

```
    factor = (j - 1) / N_points;  % j/N (from 0 to 1)
    p_j = x_initial - s * unstable_vap^factor * unstable_vep;  % Initial point along
        unstable eigenvector

    % Compute orbit for Wu-
    [cross_points, total_time, cross_times, orbit] = poincare(1, p_j', xmu, 1, h, tol);

    % Store endpoint
    endpoints(j, 3:4) = [orbit(2, end), orbit(3, end)];

    % Plot endpoints for Wu-
    figure(1);
    plot(orbit(2, end), orbit(3, end), 'go');  % Blue dots for Wu- endpoints

    % Plot individual Wu-
    figure(3);
    plot(orbit(1, :), orbit(2, :), 'green');

    % Combined plot Wu-
    figure(6);
    if j == 1
        handle = plot(orbit(1, :), orbit(2, :), 'green', 'DisplayName', 'Wu-');
        legend_handles(2) = handle;  % Store handle for Wu-
    else
        plot(orbit(1, :), orbit(2, :), 'green');
    end
end

% Loop for Ws+ (positive stable branch)
figure(4);  % For individual plot of Ws+
for j = 1:N_points
    factor = (j - 1) / N_points;  % j/N (from 0 to 1)
    p_j = x_initial + s * stable_vap^(-factor) * stable_vep;  % Initial point along
        stable eigenvector

    % Compute orbit for Ws+ (backward in time)
    [cross_points, total_time, cross_times, orbit] = poincare(1, p_j', xmu, -1, h, tol);

    % Store endpoint
    endpoints(j, 5:6) = [orbit(2, end), orbit(3, end)];

    % Plot endpoints for Ws+
    figure(1);
    plot(orbit(2, end), orbit(3, end), 'bo');  % Green dots for Ws+ endpoints

    % Plot individual Ws+
    figure(4);
    plot(orbit(1, :), orbit(2, :), 'blue');

    % Combined plot Ws+
    figure(6);
    if j == 1
        handle = plot(orbit(1, :), orbit(2, :), 'blue', 'DisplayName', 'Ws+');
        legend_handles(3) = handle;  % Store handle for Ws+
    else
        plot(orbit(1, :), orbit(2, :), 'blue');
    end
end

% Loop for Ws- (negative stable branch)
figure(5);  % For individual plot of Ws-
for j = 1:N_points
    factor = (j - 1) / N_points;  % j/N (from 0 to 1)
    p_j = x_initial - s * stable_vap^(-factor) * stable_vep;  % Initial point along
        stable eigenvector

    % Compute orbit for Ws- (backward in time)
    [cross_points, total_time, cross_times, orbit] = poincare(1, p_j', xmu, -1, h, tol);

```

```
    % Store endpoint
    endpoints(j, 7:8) = [orbit(2, end), orbit(3, end)];

    % Plot endpoints for Ws-
    figure(1);
    plot(orbit(2, end), orbit(3, end), 'mo');  % Magenta dots for Ws- endpoints

    % Plot individual Ws-
    figure(5);
    plot(orbit(1, :), orbit(2, :), 'magenta');

    % Combined plot Ws-
    figure(6);
    if j == 1
        handle = plot(orbit(1, :), orbit(2, :), 'magenta', 'DisplayName', 'Ws-');
        legend_handles(4) = handle;  % Store handle for Ws-
    else
        plot(orbit(1, :), orbit(2, :), 'magenta');
    end
end

% Combined plot legend
figure(6);
legend(legend_handles, {'Wu+', 'Wu-', 'Ws+', 'Ws-'}, 'Location', 'best');
title('Combined Orbits for All Manifolds');
xlabel('x');
ylabel('y');
hold off;

% Plot 7: Continuous Wu+ in y-x' space
figure(7);
hold on;
title('Wu+ in y-x'' Space');
xlabel('y');
ylabel('x''');
% Directly append the first point for continuity
plot([endpoints(:, 1); endpoints(1, 1)], [endpoints(:, 2); endpoints(1, 2)], 'r-',
    'LineWidth', 1.5);  % Red line
hold off;

% Plot 8: Continuous Wu- in y-x' space
figure(8);
hold on;
title('Wu- in y-x'' Space');
xlabel('y');
ylabel('x''');
plot([endpoints(:, 3); endpoints(1, 3)], [endpoints(:, 4); endpoints(1, 4)], 'b-',
    'LineWidth', 1.5);  % Blue line
hold off;

% Plot 9: Continuous Ws+ in y-x' space
figure(9);
hold on;
title('Ws+ in y-x'' Space');
xlabel('y');
ylabel('x''');
plot([endpoints(:, 5); endpoints(1, 5)], [endpoints(:, 6); endpoints(1, 6)], 'g-',
    'LineWidth', 1.5);  % Green line
hold off;

% Plot 10: Continuous Ws- in y-x' space
figure(10);
hold on;
title('Ws- in y-x'' Space');
xlabel('y');
ylabel('x''');
plot([endpoints(:, 7); endpoints(1, 7)], [endpoints(:, 8); endpoints(1, 8)], 'm-',
    'LineWidth', 1.5);  % Magenta line
hold off;
```

```
603
604 % Plot 11: Combined Continuous Endpoints for All Manifolds
605 figure(11);
606 hold on;
607 title('Combined Endpoints for All Manifolds');
608 xlabel('y');
609 ylabel('x''');
610 plot([endpoints(:, 1); endpoints(1, 1)], [endpoints(:, 2); endpoints(1, 2)], 'r-',
        'LineWidth', 1.5, 'DisplayName', 'Wu+');  % Red line
611 plot([endpoints(:, 3); endpoints(1, 3)], [endpoints(:, 4); endpoints(1, 4)], 'g-',
        'LineWidth', 1.5, 'DisplayName', 'Wu-');  % Blue line
612 plot([endpoints(:, 5); endpoints(1, 5)], [endpoints(:, 6); endpoints(1, 6)], 'b-',
        'LineWidth', 1.5, 'DisplayName', 'Ws+');  % Green line
613 plot([endpoints(:, 7); endpoints(1, 7)], [endpoints(:, 8); endpoints(1, 8)], 'm-',
        'LineWidth', 1.5, 'DisplayName', 'Ws-');  % Magenta line
614 legend('Location', 'best');
615 hold off;
616
617
618
619 %%
620 % Loop for Wu+ (positive unstable branch)
621 Jacobi_constants = zeros(N_points, 1); % Pre-allocate storage for Jacobi constants
622
623 for j = 1:N_points
624     factor = (j - 1) / N_points;  % j/N (from 0 to 1)
625     p_j = x_initial + s * unstable_vap^factor * unstable_vep;  % Initial point along
            unstable eigenvector
626
627     % Compute orbit for Wu+
628     [~, ~, ~, orbit] = poincare(1, p_j', xmu, 1, h, tol);
629
630     % Calculate Jacobi constant
631     x = orbit(1, end);  % Position x
632     y = orbit(2, end);  % Position y
633     x_dot = orbit(3, end);  % Velocity in x
634     y_dot = orbit(4, end);  % Velocity in y
635     r1 = sqrt((x - xmu)^2 + y^2);
636     r2 = sqrt((x - xmu + 1)^2 + y^2);
637     Omega_value = 0.5 * (x^2 + y^2) + (1 - xmu) / r1 + xmu / r2;  % Potential energy
638     Jacobi_constants(j) = 2 * Omega_value - (x_dot^2 + y_dot^2);
639 end
640
641 % Compute the derivative of the Jacobi constant
642 Jacobi_derivative = diff(Jacobi_constants);
643
644 % Plot the derivative
645 figure;
646 plot(1:N_points-1, Jacobi_derivative, 'b-', 'LineWidth', 1.5);
647 title('Derivative of the Jacobi Constant for Wu+');
648 xlabel('Orbit Index');
649 ylabel('dC_J / dIndex');
650 grid on;
651 % Set y-axis limits for a range around 10^-6
652 ylim([-1e-9, 1e-9]); % Adjust the y-axis to range [-5e-13, 5e-13]
653
654
655 %%
656 function [x0,y0_prime]=newton_variational(x0,y0_prime,xmu,T,tol)
657     deltaT=0.0001;
658     options = odeset('RelTol',1e-13,'AbsTol',1e-13);  % ODE solver options for accuracy
659     [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T/2], [x0 0 0
            y0_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
660     x_prime=X(end,3);
661     y=X(end,2);
662     counter=1;
663     while abs(x_prime)>tol || abs(y)>tol
664         A=[X(end,9),X(end,12);X(end,13),X(end,16)];
665         b=[-y;-x_prime];
```

```
        delta=A\b;
        x0=x0+delta(1);
        y0_prime=y0_prime+delta(2);
        [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T/2], [x0 0 0
            y0_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
        x_prime=X(end,3);
        y=X(end,2);
    end
end


% Function defining the system of differential equations
function df = f_variational_eq(~,x,mu,dir)
  df = zeros(20,1);
  r1 = sqrt((x(1)-mu)^2 + x(2)^2);
  r2 = sqrt((x(1)-mu+1)^2 + x(2)^2);

  df(1) = x(3);
  df(2) = x(4);
  df(3) = 2*x(4) + x(1) - ((1-mu)*(x(1)-mu)/(r1^3)) - mu*(x(1)-mu+1)/(r2^3);
  df(4) = -2*x(3) + x(2)*(1 - (1-mu)/(r1^3) - mu/(r2^3));

  df(5)=x(13);
  df(6)=x(14);
  df(7)=x(15);
  df(8)=x(16);
  df(9)=x(17);
  df(10)=x(18);
  df(11)=x(19);
  df(12)=x(20);
  df(13)=Omegaxx([x(1) x(2)],mu)*x(5)+Omegaxy([x(1) x(2)],mu)*x(9)+2*x(17);
  df(14)=Omegaxx([x(1) x(2)],mu)*x(6)+Omegaxy([x(1) x(2)],mu)*x(10)+2*x(18);
  df(15)=Omegaxx([x(1) x(2)],mu)*x(7)+Omegaxy([x(1) x(2)],mu)*x(11)+2*x(19);
  df(16)=Omegaxx([x(1) x(2)],mu)*x(8)+Omegaxy([x(1) x(2)],mu)*x(12)+2*x(20);
  df(17)=Omegaxy([x(1) x(2)],mu)*x(5)+Omegayy([x(1) x(2)],mu)*x(9)-2*x(13);
  df(18)=Omegaxy([x(1) x(2)],mu)*x(6)+Omegayy([x(1) x(2)],mu)*x(10)-2*x(14);
  df(19)=Omegaxy([x(1) x(2)],mu)*x(7)+Omegayy([x(1) x(2)],mu)*x(11)-2*x(15);
  df(20)=Omegaxy([x(1) x(2)],mu)*x(8)+Omegayy([x(1) x(2)],mu)*x(12)-2*x(16);
  if dir == -1
      df=-df;
  end
end

function df = f(~,x,mu,dir)
  df = zeros(4,1);
  r1 = sqrt((x(1)-mu)^2 + x(2)^2);
  r2 = sqrt((x(1)-mu+1)^2 + x(2)^2);

  df(1) = x(3);
  df(2) = x(4);
  df(3) = 2*x(4) + x(1) - ((1-mu)*(x(1)-mu)/(r1^3)) - mu*(x(1)-mu+1)/(r2^3);
  df(4) = -2*x(3) + x(2)*(1 - (1-mu)/(r1^3) - mu/(r2^3));

  if dir == -1
      df=-df;
  end
end

function [L3,C3,eigL3,veigL3]=RTBP_eq_points(xmu,tol)
    %%%%%L3%%%%%
    xi=1-(7*xmu)/(12);
    aux=0;
    while(abs(xi-aux)>tol)
        aux=xi;
        xi=F3(xi,xmu);
    end
    L3=[xmu+xi,0,0,0];
    xL3=L3(1);
    C3 = 2*Omega([xL3 0],xmu,abs(xL3-xmu),abs(xL3-xmu+1));
```

```
    DG3 = zeros(4,4); DG3(1,3)=1; DG3(2,4)=1; DG3(3,4)=2; DG3(4,3)=-2;
    DG3(3,1) = Omegaxx([xL3 0],xmu);
    DG3(3,2) = Omegaxy([xL3 0],xmu);
    DG3(4,1) = Omegaxy([xL3 0],xmu);
    DG3(4,2) = Omegayy([xL3 0],xmu);
    [veigL3,eigL3] = eig(DG3);
    eigp3=zeros(4,1);
    for i = 1:4
        eigp3(i)=eigL3(i,i);
    end
    eigL3=eigp3;
end


function res = F(x,xmu,dy)
  [cross_points,~,~,~]=poincare(1,[x 0 0 dy],xmu,1,1e-2,1e-13);
  res = cross_points(3);
end


function dy=dy_i(x,xmu,C,ysign)
    r1 = sqrt((x-xmu)^2);
    r2 = sqrt((x-xmu+1)^2);
    dy= ysign*sqrt(2*Omega([x 0], xmu, r1, r2)-C);
end

function new=F1(old,xmu)
    new = ((xmu*(1-old)^2)/(3-2*xmu-old*(3-xmu-old)))^(1/3);
end
function new=F2(old,xmu)
    new = ((xmu*(1+old)^2)/(3-2*xmu+old*(3-xmu+old)))^(1/3);
end
function new=F3(old,xmu)
    new = (((1-xmu)*(1+old)^2)/(1+2*xmu+old*(2+xmu+old)))^(1/3);
end

function res = Omega(x,mu,r1,r2)
  res = (1/2)*(x(1)^2 + x(2)^2) + (1-mu)/r1 + mu/r2 + (1/2)*(mu*(1-mu));
end

function res = Omegaxx(x,mu)
  res =
      (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)-(3*(2*mu-2*x(1))^2*(mu-1))
      ...
      +x(2)^2)^(5/2))+(3*mu*(2*x(1)-2*mu+2)^2)/(4*((x(1)-mu+1)^2+x(2)^2)^(5/2))+1;
end

function res = Omegayy(x,mu)
  res =
      (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)+(3*mu*x(2)^2)/((x(1)-mu+1)
      ...
      +x(2)^2)^(5/2)-(3*x(2)^2*(mu-1))/((mu-x(1))^2+x(2)^2)^(5/2)+1;
end

function res = Omegaxy(x,mu)
  res = (3*x(2)*(2*mu-2*x(1))*(mu-1))/(2*((mu-x(1))^2+x(2)^2)^(5/2)) ...
      +(3*mu*x(2)*(2*x(1)-2*mu+2))/(2*((x(1)-mu+1)^2+x(2)^2)^(5/2));
end

function val = g(x)
%NEW POINCAR  SECTION, THIS IS X = 0.2, g(x) must be zero
    val = x(1)-0.2;
end


function [x1,x2]=varying(x,xmu,C,ysign,delta)
    x1=x;
    x2=x+delta;
    dy1=dy_i(x1,xmu,C,ysign);
```

```
    dy2=dy_i(x2,xmu,C,ysign);
    while F(x1,xmu,dy1)*F(x2,xmu,dy2)>=0
        x1=x2;
        x2=x2+delta;
        dy1=dy2;
        dy2=dy_i(x2,xmu,C,ysign);
        %F(x1,xmu,dy1)
    end
end

function [c,t]=bisection_method(x1,x2,xmu,C,ysign,tol,max_counter)
    dy1=dy_i(x1,xmu,C,ysign);
    dy2=dy_i(x2,xmu,C,ysign);
    bool=0;
    if F(x1,xmu,dy1)==0
        c=x1;
    bool=1;
    end
    if F(x2,xmu,dy2)==0 && bool==0
        c=x2;
    bool=1;
    end
    counter=0;
    while abs(x2-x1)>tol && counter<max_counter && bool==0
        c=(x2+x1)/2;
        dy1=dy_i(x1,xmu,C,ysign);
        dyc=dy_i(c,xmu,C,ysign);
        if F(c,xmu,dyc)==0 && bool==0
            bool=1;
        end
        if bool==0
        if F(x1,xmu,dy1)*F(c,xmu,dyc) <0
            x2=c;
        else
            x1=c;
        end
        end
        counter=counter+1;
    end
    c=(x2+x1)/2;
    [~,t,~,~]=poincare(2,[c 0 0 dy_i(c,xmu,C,ysign)],xmu,1,1e-4,1e-13);
end

function
    [cross_points,total_time,cross_times,orbit]=poincare(n_crossing,x0,xmu,idir,h,tol)

options = odeset('RelTol',1e-10,'AbsTol',1e-10);  % ODE solver options for accuracy

% Initialize arrays to store crossing points and times
cross_points = zeros(n_crossing, 4);  % Pre-allocate for speed: stores [position,
    velocity]
total_time = 0;                       % Initialize total time counter
cross_times = zeros(n_crossing, 1);   % Pre-allocate for speed: stores crossing times
orbit=[x0'];
for i = 1:n_crossing
    % For subsequent crossings, the initial condition is the previous crossing point
    if i ~= 1
        x0 = cross_points(i-1, :);  % Set x0 to the previous crossing point
    end
    val=6;
    found=0;
    while found==0
    % Solve the ODE using ode45 from t=0 to t=h with initial condition x0
    [t, x1] = ode45(@(t,y) f(t,y,xmu,idir), [total_time,total_time+val], x0, options);
    index=2;
    size_sol=size(x1);
    while index<=size_sol(1)-1 && found==0
        if g(x1(index,:))*g(x1(index+1,:))<0
            found=1;
```

```
        else
        index=index+1;
        end
    end
    orbit=[orbit,x1(1:index,:)'];
    total_time=t(index);

    x0=x1(index,:);
    end
    % Approximate the crossing point
    approx = x1(index, :);
    % Refine the crossing point using Newton's method until the error is within tolerance
    counter=0;
    while abs(g(approx)) > tol
        counter=counter+1;
        % Compute correction delta using Newton's method
        delta = -g(approx) / approx(3);
        total_time = total_time + idir * delta;  % Update the total time
        if counter==300 || abs(approx(4))<1e-8
            total_time=-1;
            return
        end
        % Choose the direction for ODE integration based on the sign of delta
        if delta < 0
            [t, new_approx] = ode45(@(t,y) f(t,y,xmu,-1), [0, abs(delta)], approx,
                options);
        end
        if delta > 0
            [t, new_approx] = ode45(@(t,y) f(t,y,xmu,1), [0, abs(delta)], approx,
                options);
        end
        approx = new_approx(end, :);  % Update the approximation

    end
    % Store the refined crossing point and time
    cross_points(i, :) = approx;
    cross_times(i) = total_time;
    orbit=[orbit,approx'];
    end
end
```

ASSIGNMENT 12: COMPUTATION OF POINCARÉ SECTION PLOTS (PSP) FOR THE RTBP

Pol Navarro Pérez

December 26, 2024

Abstract

In this work, we analyze the dynamics of the Restricted Three-Body Problem (RTBP) by computing Zero Velocity Curves (ZVCs) and Poincaré Section Plots (PSPs). Using numerical continuation methods, ZVCs are derived for varying Jacobi constant values, partitioning the configuration space into regions of allowed and forbidden motion. PSPs are constructed for selected initial conditions, illustrating periodic and quasiperiodic orbits as well as chaotic dynamics in different regions. Finally, we compute the monodromy matrix and the stability parameter for specific initial conditions, extracting valuable insights into the stability properties of the system.

PART 1

In this first section, we use the pseudo-archlength method to compute the zero velocity curves. For a given value of the Jacobi constant C, the ZVCs are defined by the equation:

$$G(x,y) = 2\left(\frac{x^2+y^2}{2} + \frac{1-\mu}{\sqrt{(x-\mu)^2+y^2}} + \frac{\mu}{\sqrt{(x-(1-\mu))^2+y^2}}\right) - C, \tag{1}$$

where μ is the mass ratio of the two primary bodies.

The ZVCs partition the configuration space into regions of allowed and forbidden motion. We compute these curves for specific values of C: $C > C_1$, $C = C_1$, $C_2 < C < C_1$, $C = C_2$, $C_3 < C < C_2$, $C = C_3$, and $C_5 = C_4 < C < C_3$.

Starting from an initial solution $X(s_0) = (x_0, y_0)$, where $G(x_0, y_0) = 0$, the system is augmented with an additional constraint:

$$\begin{cases} G(x_1, y_1) = 0, \\ (x_1 - x_0)x_0' + (y_1 - y_0)y_0' - \Delta s = 0, \end{cases} \tag{2}$$

where Δs is the step size, and (x_0', y_0') is the tangent vector to the solution branch.

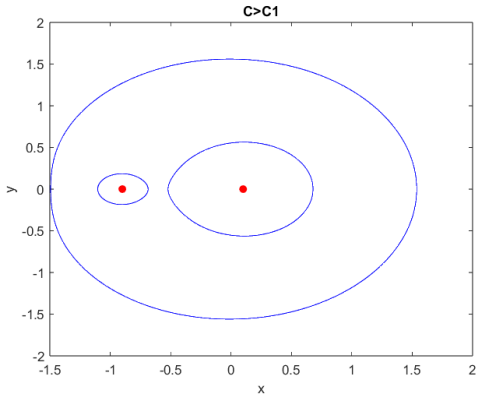
The system is solved iteratively using Newton's Method. The initial guess for (x_1, y_1) is predicted along the tangent direction:

$$(x_1^{(0)}, y_1^{(0)}) = (x_0, y_0) + \Delta s(x_0', y_0'). \tag{3}$$

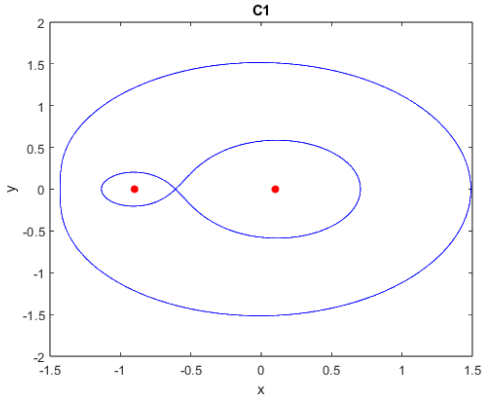
After each step, the tangent vector at the new point is updated by solving the linearized system:

$$\begin{bmatrix} G_x & G_y \\ x_0' & y_0' \end{bmatrix} \begin{bmatrix} x_1' \\ y_1' \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{4}$$

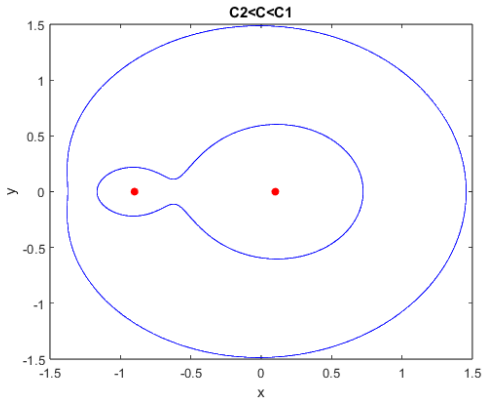
The resulting vector is normalized to ensure a unit tangent direction. This normalization ensures that each step along the solution branch maintains consistent spacing, defined by Δs.



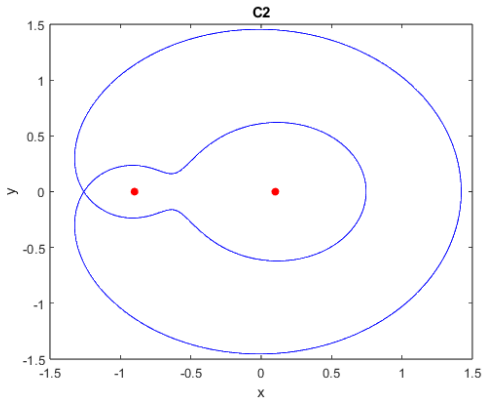
$\mu = 0.1,\ C > C_1$



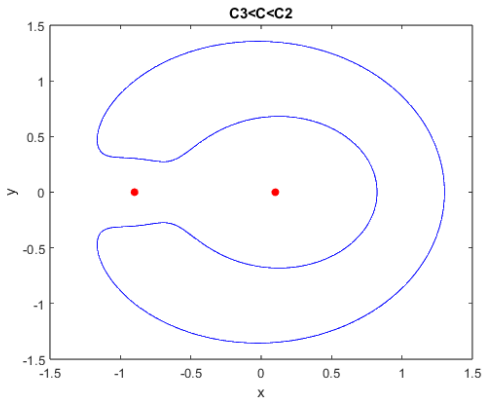
$\mu = 0.1,\ C = C_1$



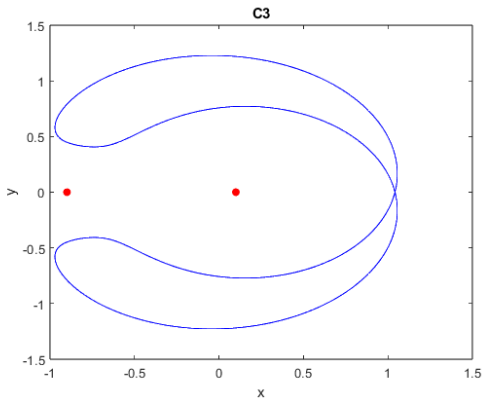
$\mu = 0.1,\ C_2 < C < C_1$



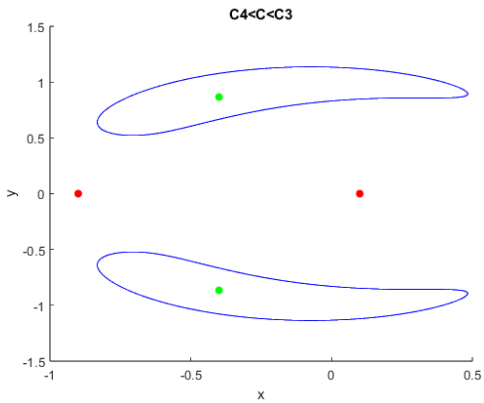
$\mu = 0.1,\ C = C_2$



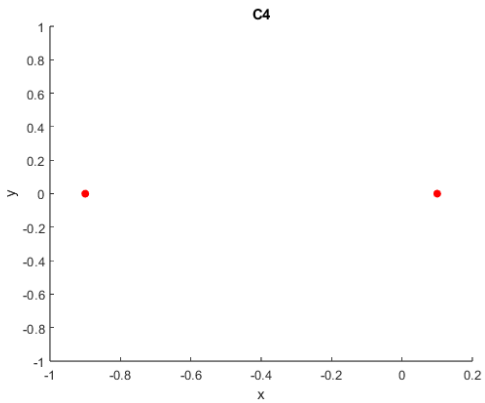
$\mu = 0.1,\ C_3 < C < C_2$



$\mu = 0.1,\ C = C_3$

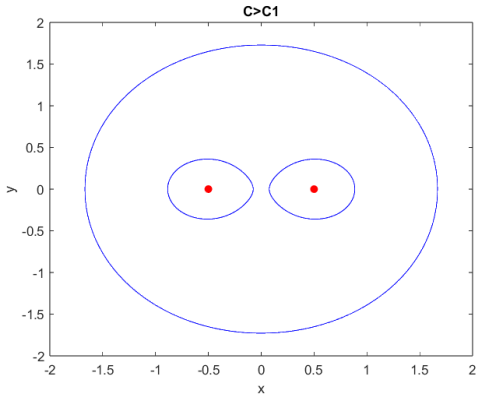


$\mu = 0.1,\ C_4 < C < C_3$

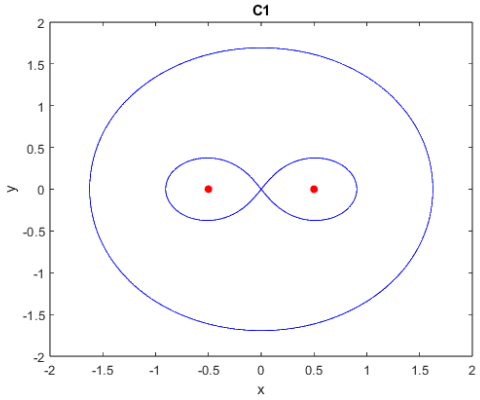


$\mu = 0.5,\ C = C_4$

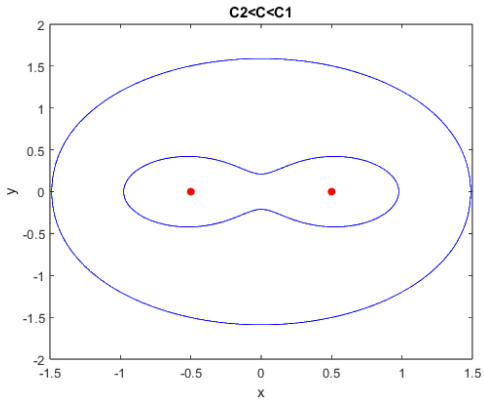
Figure 1: Zero Velocity Curves for $\mu = 0.1$ and varying C values. Primary bodies are the red dots and L_4 and L_5 equilibrium points the green ones.



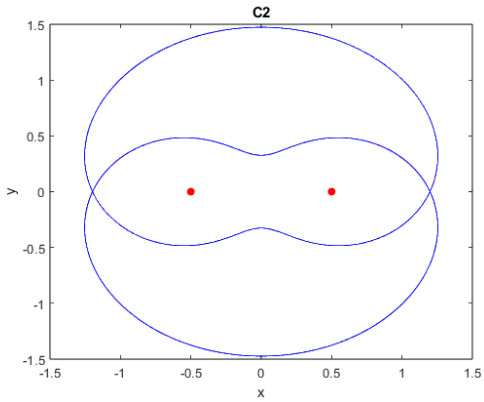
$\mu = 0.5,\ C = C_1$



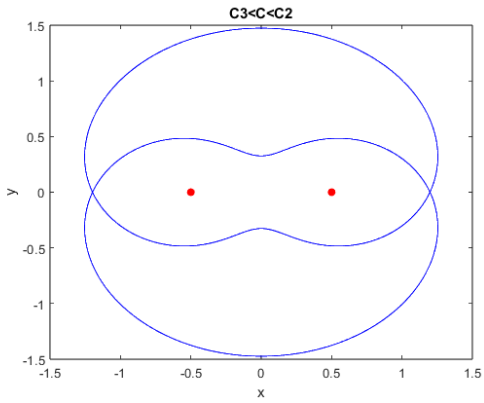
$\mu = 0.5,\ C = C_1$



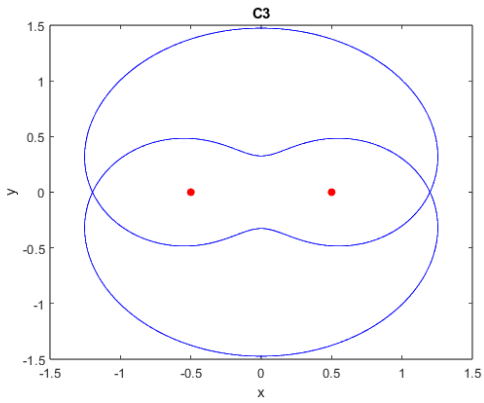
$\mu = 0.5,\ C_2 < C < C_1$



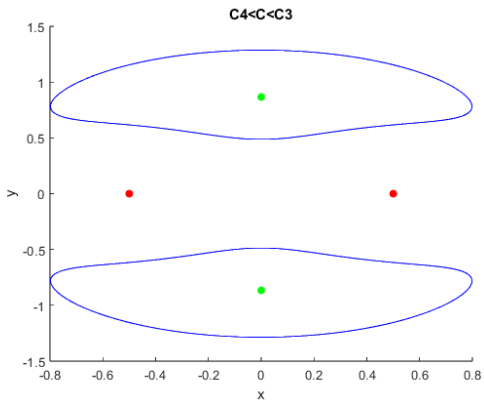
$\mu = 0.5,\ C = C_2$



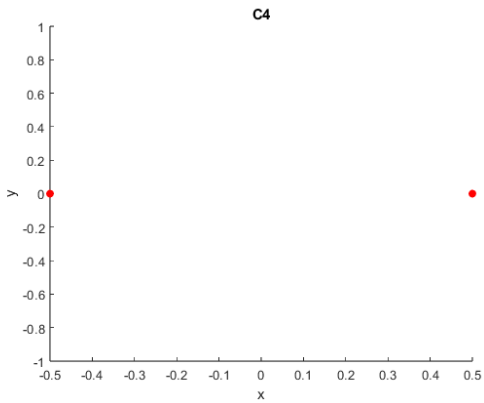
$\mu = 0.5,\ C_3 < C < C_2$



$\mu = 0.5,\ C = C_3$



$\mu = 0.5,\ C_4 < C < C_3$



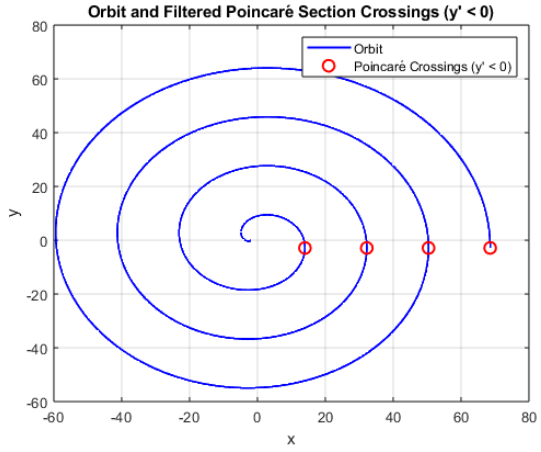
$\mu = 0.5,\ C = C_4$

Figure 2: Zero Velocity Curves for $\mu = 0.5$ and varying C values.

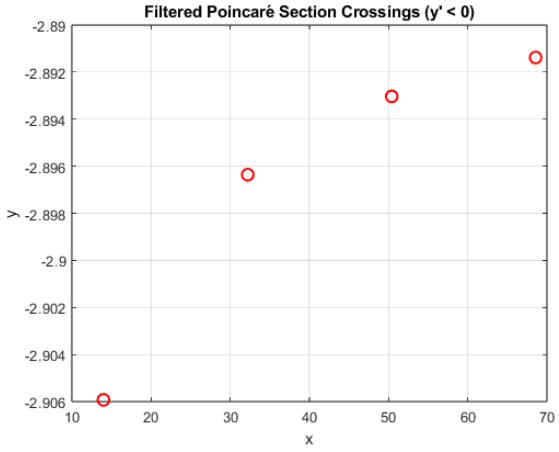
Figure 1 shows the ZVC for multiple values of C, related to the Jacobi constant value at each equilibrium point. Same for Figure 2, this time for a higher value of μ. As seen in both figures for $C = C_4$ all the phase space is allowed.

PART 2

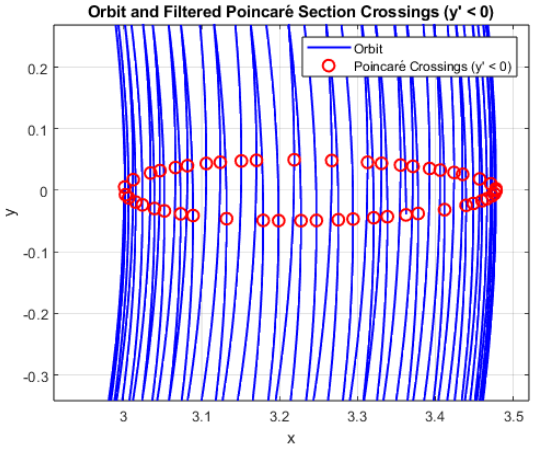
In this section, we are going to work with the Poincaré section plots (PSP). First, we start with some initial conditions and we compute the successive crossing with the Poincaré section $x' = 0$, $y' < 0$. Afterward, we plot the corresponding (x, y) values for this crossing.



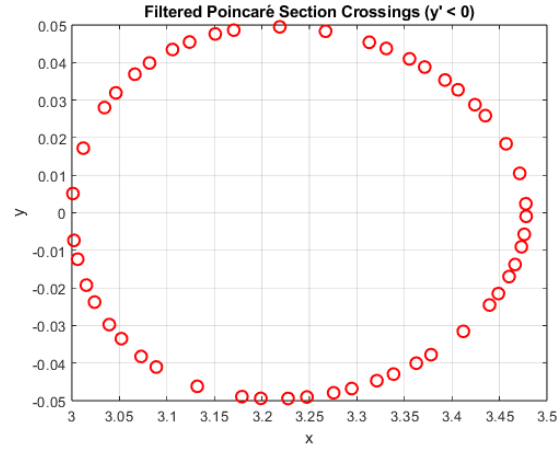
PSP for initial condition $x = -2$, $\mu = 0.1$, $C = 4$ and 4 crossing points.



Crossing points for the initial condition $x = -2$ in the (x, y) space.



PSP for initial condition $x = 3$, $\mu = 0.1$, $C = 4$ and 50 crossing points.



Crossing points for the initial condition $x = 3$ in the (x, y) space.

Figure 3: PSP for $\mu = 0.1$, $C = 4$ and some initial values.

Figure 3 shows the difference of the PSP depending on the initial value taken for the orbit. In the case of $x = -2$ and 4 iterations the crossing points are increasing in both (x, y) coordinates. The second case, $x = 3$ and 50 iterations, shows the crossing points make some ellipse.

Now we can try to observe the behavior of the PSP inside the allowed motion regions computed before, the ZVC for this $C = 4$ value. Therefore, we take some initial values on the x values, and for each of these plot the corresponding crossings.

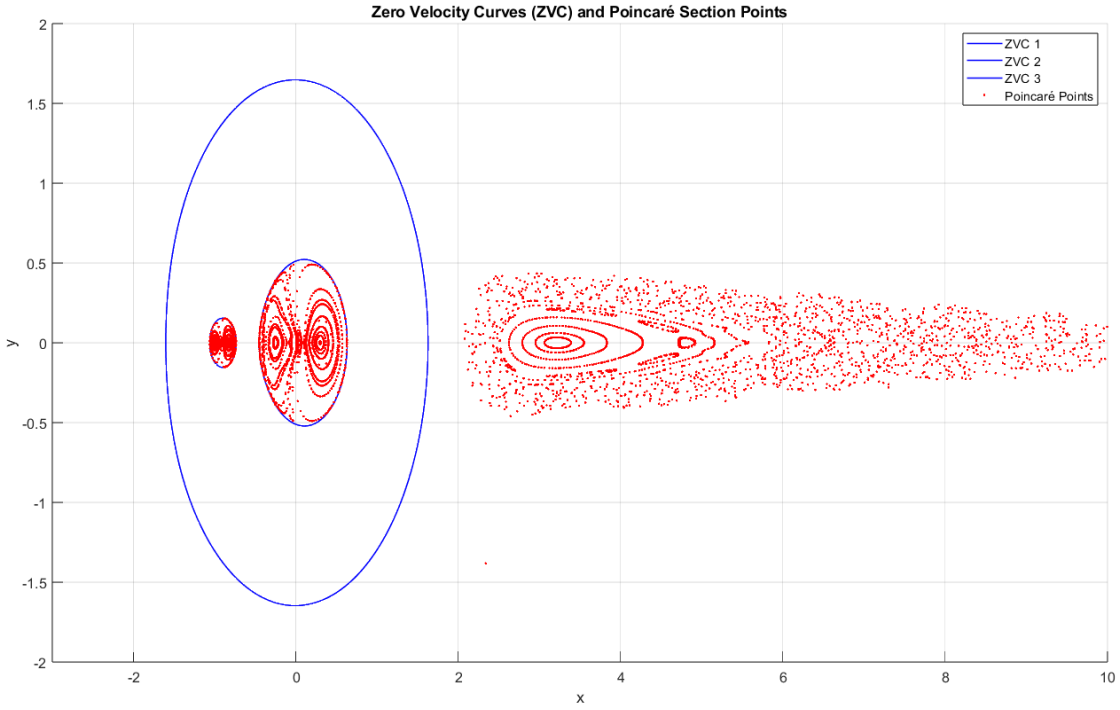


Figure 4: PSP for multiple initial conditions inside the allowed regions of motion and the ZVC for $C = 4$.

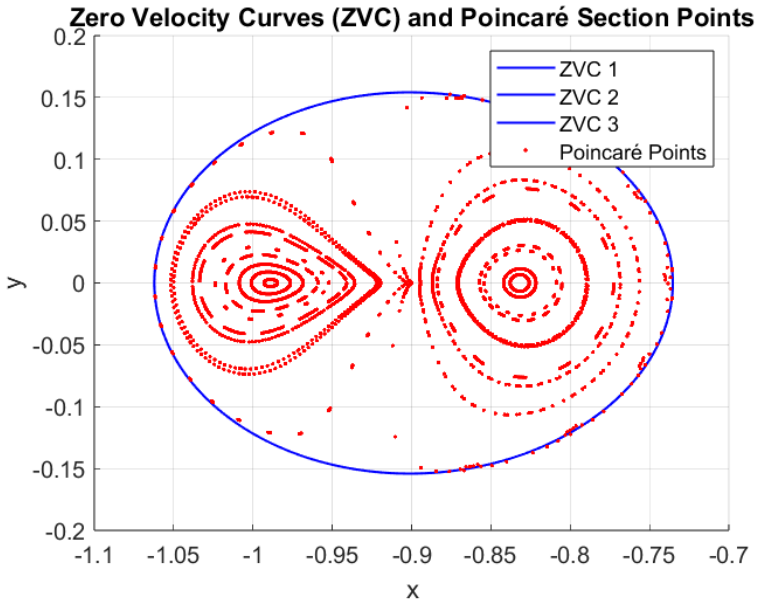


Figure 5: PSP around a region of interest. This is around $P_1(\mu - 1, 0)$.

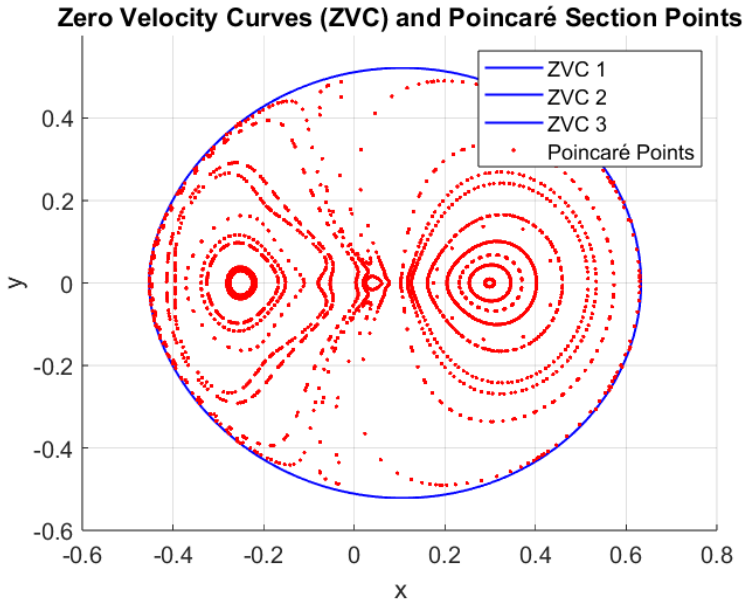


Figure 6: PSP around a region of interest. This is around $P_2(\mu, 0)$.

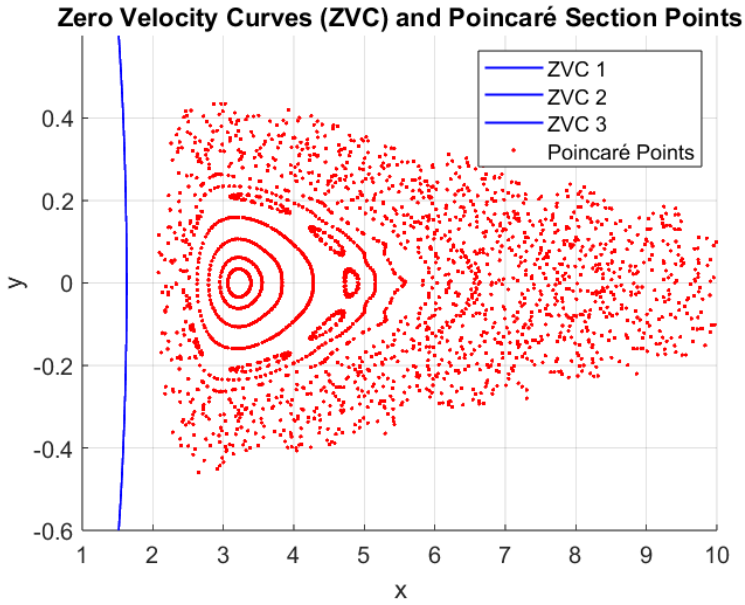


Figure 7: PSP for the outer region.

Figure 4 illustrates the phase-space dynamics of the system. The blue curves, labeled as ZVC 1, ZVC 2, and ZVC 3, correspond to the zero velocity curves, which act as boundaries for the regions of possible motion. These are computed as the solutions of $C - 2\Omega(x, y) = 0$, given a fixed value for the Jacobi constant. In other applications this C could be, for instance, a fixed value of the Hamiltonian system h.

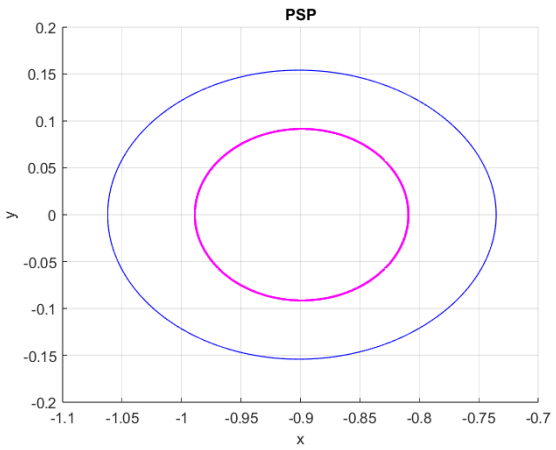
The red dots indicate the points where trajectories intersect the Poincaré section, providing insights into the dynamic behavior of the system. Figures 5, 6 and 7 are plotted to help in understanding the motion of our dynamical system in the different regions.

- Figure 5 shows two sets of periodic orbits inside a bounded ZVC. This is around the first primary body, located at $(-1 + \mu, 0)$. These sets are on the left and on the right of the mentioned primary body.

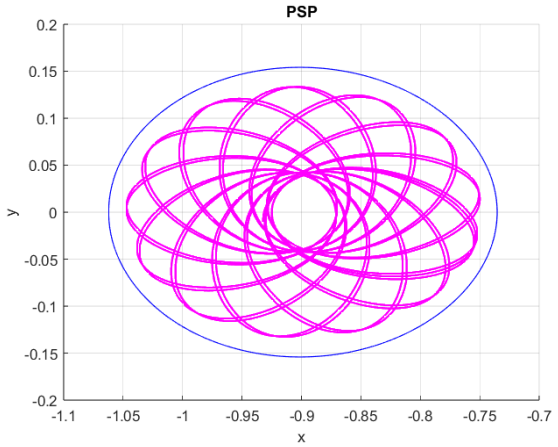
  As Figure 8 shows, the fixed point in the PSP corresponds to a PO. Similar happens with points from the closed orbits in the PSP. In this case, these are quasiperiodic orbits. In addition, periodic orbits give rise to a family of nearby quasiperiodic orbits.

- Regarding the second bounded ZVC_2, there is a similar structure than before. In this case, the dynamics are around the second primary body P_2. As in the surrounding of the other primary, there seems to be two fixed points surrounded by some PO in this region of the PSP, on both sides of $(\mu, 0)$. Similar to what we did previously Figure 9 shows how the fixed points in the PSP are PO and the closed orbits are quasiperiodic orbits in the (x,y) projection.

- The final region to analyze is the outer region. Outside the ZVC, the body is too far from the primary masses, and therefore, no bounded structures are present, unlike in the previous cases. Moreover, in this region there is no planet inside the studied region, this is also a remarkable point to analyze the system. Figure 10 shows the (x,y) projection for a fixed point from the outside regions of the PSP, as expected, being a PO. In Figure 11 we can see the quasiperiodic orbit taking a point from the closed curves from the outer region in the PSP. In this case, we have some new points to discuss, the ones from the chaotic region. As Figure 12 shows, these points of the chaotic regions represent unbounded orbits. This means that we could have escaping orbits when taking initial conditions from the outer regions.

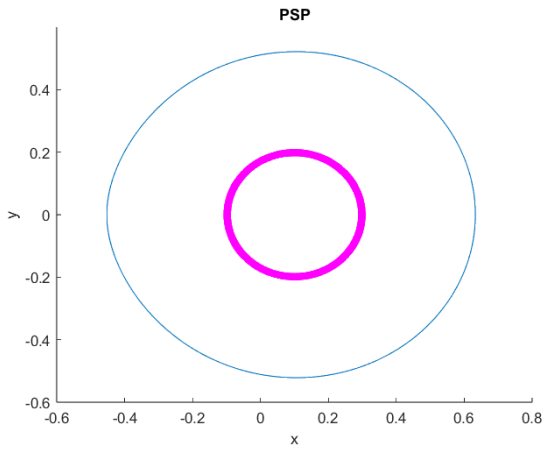


Orbit (x,y) for a fixed point in the PSP, x = -0,99.

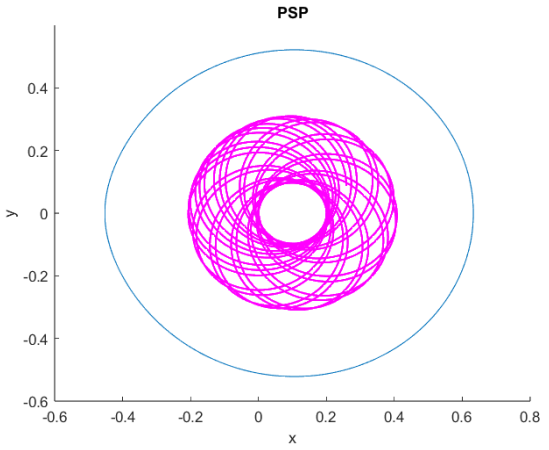


Orbits (x,y) for a point of the invariant curve in the PSP, x = -0.925

Figure 8: Orbits (x,y) for some interesting points from the PSP plot.



Orbit (x,y) for a fixed point in the PSP, x = 0.29.



Orbits (x,y) for a point of the invariant curve in the PSP, x = 0.20.

Figure 9: Orbits (x,y) for some interesting points from the PSP plot.

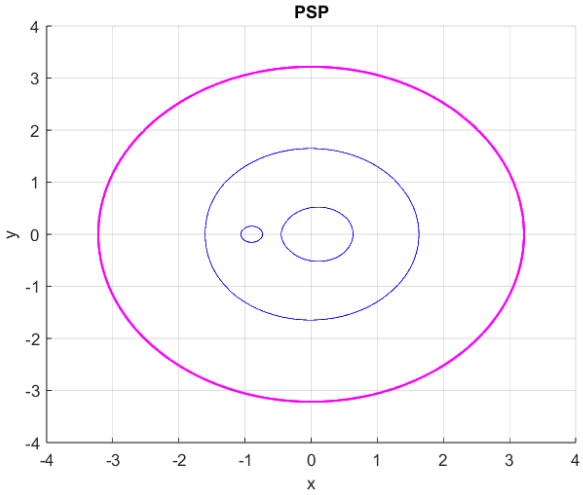


Figure 10: Orbits of a fixed point from the PSP outer region.

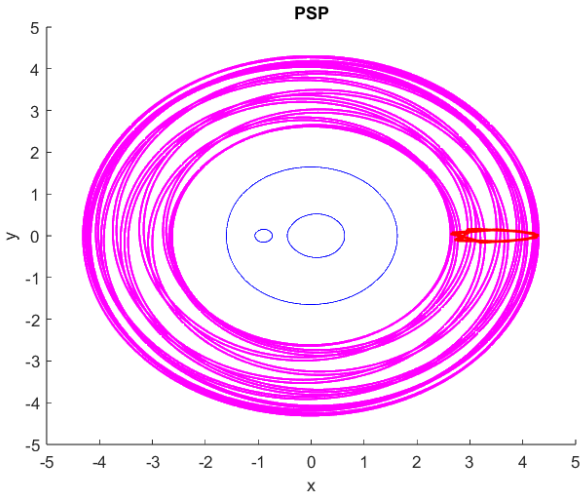


Figure 11: Orbits of a point of the invariant curve from the PSP outer region.

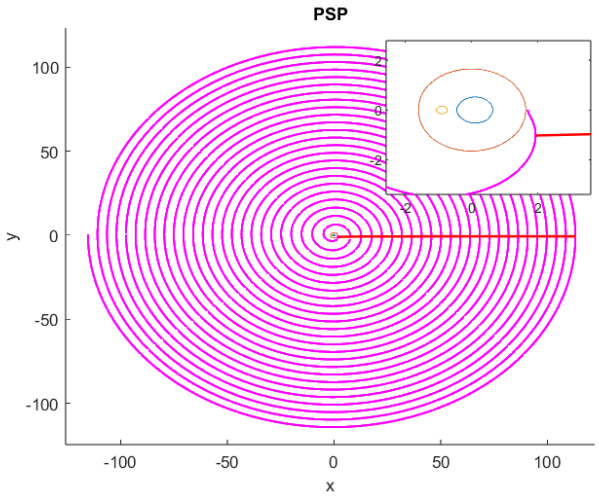


Figure 12: Orbit of a point from the chaotic region in the PSP plot.

For the last part of the assignment, we start at the point $x_0 = 3.2171$ and compute the time to reach the first crossing. Then, with this time as a fixed period find the initial condition (x, y) to get the PO.

The values found for the PO are $(x, y') = (3.2906, -2.7381)$ with a Jacobi constant value of 4.033150586970220.

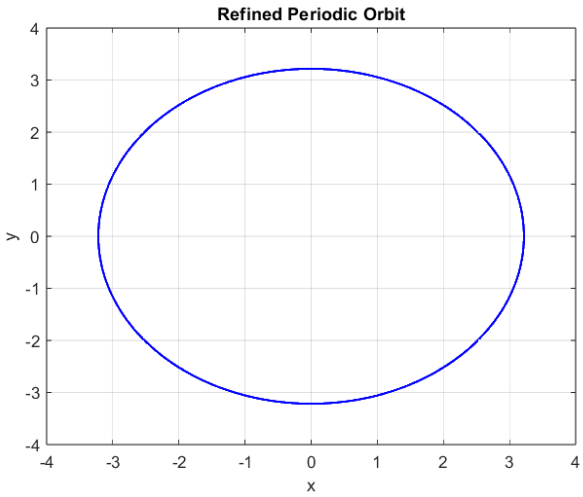


Figure 13: Periodic orbit for the initial state found x = 3.2906.

Figure 13 shows the PO found by refinement. The Monodromy matrix M for the given periodic orbit is:

$$M = \begin{bmatrix} 10.0952 & -4.7501 & 5.7054 & 8.3858 \\ -1.8563 & 7.9817 & -8.3858 & 0.0531 \\ 0.5351 & 6.3911 & -6.6765 & 1.9626 \\ -10.6181 & 5.5455 & -6.6608 & -8.7900 \end{bmatrix}.$$

The eigenvalues of M are:

$$\lambda_1 = 0.30516851548374 + 0.95229836561731i$$
$$\lambda_2 = 0.30516851548374 - 0.95229836561731i$$
$$\lambda_3 = 1.0000165320082$$
$$\lambda_4 = 0.99999834680205$$

From the eigenvalues, we notice there are two with only real parts, equal to 1. In addition, there are two complex eigenvalues, conjugated to each other. These have a norm of value 1. The stability parameter is $k = 0.610337030970339$, this value is inside the interval (-2,2). Therefore, the orbit is linearly stable, meaning that the mentioned closeby orbits will tend to this plotted orbit of this particular initial state.

CODE

Listing 1: code

```
format long;
close all;
clear all;
clc

% Parameters
tol = 1e-13;
s = 1e-6;
max_counter = 1000;
xmu = 0.1;
ysign = -1;
delta = 1e-3;
h = 1e-3;
% forward
idir = 1;

% Initial conditions
x0 = 3.0;
C = 4.033150586970220;

% Compute initial y_prime
y_prime = dy_i(x0, xmu, C, ysign);
disp(y_prime);

% Poincar  options and computation
options = odeset('RelTol', 1e-13, 'AbsTol', 1e-13);
[cross_points, ~, ~, orbit1] = poincare(100, [x0, 0.0, 0.0, y_prime], xmu, idir, h, tol);

% Filter crossing points where y' < 0
% % even cross points
% cross_points_filtered = cross_points(2:2:end,:);
% Filtrar puntos de cruce donde y' < 0
cross_points_filtered = cross_points(cross_points(:,4) < 0, :);
% Figure 1: Plotting the orbit and filtered crossing points
figure;
plot(orbit1(1,:), orbit1(2,:), 'b-', 'LineWidth', 1.5); % Orbit in blue
hold on;
plot(cross_points_filtered(:,1), cross_points_filtered(:,2), 'ro', 'MarkerSize', 8,
    'LineWidth', 1.5); % Filtered points in red
xlabel('x');
ylabel('y');
title('Orbit and Filtered Poincar  Section Crossings (y'' < 0)');
legend('Orbit', 'Poincar  Crossings (y'' < 0)');
grid on;
hold off;

% Figure 2: Plotting only the filtered Poincar  crossing points
figure;
plot(cross_points_filtered(1:end,1), cross_points_filtered(1:end,2), 'ro', 'MarkerSize',
    8, 'LineWidth', 1.5);
xlabel('x');
ylabel('y');
title('Filtered Poincar  Section Crossings (y'' < 0)');
grid on;


%%
% Cargar datos
orbit1 = readmatrix('zvc_orbit1.txt');
orbit2 = readmatrix('zvc_orbit2.txt');
orbit3 = readmatrix('zvc_orbit3.txt');

orbits = {orbit1, orbit2, orbit3};

% Figura principal
```

```
figure;
hold on;
colors = lines(length(orbits)); % Generar colores distintos

% Dibujar  rbitas  principales
for i = 1:length(orbits)
    plot(orbits{i}(:, 1), orbits{i}(:, 2), 'DisplayName', ['Orbit ' num2str(i)], 'Color',
        colors(i, :));
end

% Dibujar  rbita  y puntos Poincar
x0 = 2.48448275;
C = 4;

% Calcular y_prime
y_prime = dy_i(x0, xmu, C, ysign);
disp(y_prime);

% Opciones de Poincar  y c lculo
options = odeset('RelTol', 1e-13, 'AbsTol', 1e-13);
[cross_points, ~, ~, orbit1] = poincare(500, [x0, 0.0, 0.0, y_prime], xmu, idir, h, tol);

plot(orbit1(1, :), orbit1(2, :), 'magenta', 'LineWidth', 1.5);

xlabel('x');
ylabel('y');
title('PSP');
% xlim([-0.6, 0.8])
% ylim([-0.6, 0.6])

% % Filtrar puntos cruzados
% cross_points_filtered = cross_points(cross_points(:,4) < 0, :);
% plot(cross_points_filtered(:,1), cross_points_filtered(:,2), 'r-', 'MarkerSize', 8,
    'LineWidth', 1.5);
%
% % Sub-ejes para el zoom con superposici n
% zoom_axes = axes('Position', [0.6, 0.6, 0.3, 0.3]); % Ajusta la posici n y el tama o
% set(zoom_axes, 'Color', 'white'); % Fondo opaco blanco
% box on;
% hold on;
%
% % Dibujar el zoom
% for i = 1:length(orbits)
%     plot(zoom_axes, orbits{i}(:, 1), orbits{i}(:, 2), 'Color', colors(i, :));
% end
%
% plot(zoom_axes, orbit1(1, :), orbit1(2, :), 'magenta', 'LineWidth', 1.5);
% plot(zoom_axes, cross_points_filtered(:,1), cross_points_filtered(:,2), 'r-',
    'MarkerSize', 8, 'LineWidth', 1.5);
%
% % Configuraci n de los l mites del zoom
% xlim(zoom_axes, [-8, 8]);
% ylim(zoom_axes, [-8, 8]);
%
% hold off;

%%


% Parameters
tol = 1e-13;
s = 1e-6;
max_counter = 1000;
xmu = 0.1;
ysign = -1;
delta = 1e-3;
h = 1e-3;
idir = 1; % Forward integration
C = 4;
```

```
130
131 ranges = {
132     [-1.05, -0.74],    % Range 1
133     [-0.452, 0.63],    % Range 2
134     [1.63, 10]         % Range 3
135 };
136
137 % Generate approximately 10 points for each range
138 x_vals = [];
139 for i = 1:length(ranges)
140     x_vals = [x_vals, linspace(ranges{i}(1), ranges{i}(2), 20)];
141 end
142 orbits_poincare = {};
143 cross_points_all = [];
144
145 % Load ZVCs
146 orbit1 = readmatrix('zvc_orbit1.txt');
147 orbit2 = readmatrix('zvc_orbit2.txt');
148 orbit3 = readmatrix('zvc_orbit3.txt');
149 orbits = {orbit1, orbit2, orbit3};
150
151 % Loop through initial conditions and compute Poincar  crossings
152 for i = 1:length(x_vals)
153     x0 = x_vals(i);
154
155     % Compute initial y_prime
156     y_prime = dy_i(x0, xmu, C, ysign);
157
158     % Compute Poincar  Section
159     options = odeset('RelTol', 1e-13, 'AbsTol', 1e-13);
160     [cross_points, ~, ~, orbit] = poincare(300, [x0, 0.0, 0.0, y_prime], xmu, idir, h,
        tol);
161
162     % Filter crossing points where y' < 0
163     cross_points_filtered = cross_points(cross_points(:, 4) < 0, :);
164     cross_points_all = [cross_points_all; cross_points_filtered(:, 1:2)]; % Collect all
        crossings
165     orbits_poincare{i} = orbit; % Save orbit for this initial condition
166 end
167
168 %%
169 figure;
170 hold on;
171
172 % Plot ZVCs
173 for i = 1:length(orbits)
174     plot(orbits{i}(:, 1), orbits{i}(:, 2), 'Color', 'blue', 'LineWidth', 1.2,
        'DisplayName', ['ZVC ' num2str(i)]);
175 end
176
177 % Plot PSP Points
178 plot(cross_points_all(:, 1), cross_points_all(:, 2), 'r.', 'LineWidth', 1.5,
    'DisplayName', 'Poincar  Points');
179 xlabel('x', 'FontSize', 14); % Make x-label larger
180 ylabel('y', 'FontSize', 14); % Make y-label larger
181 title('Zero Velocity Curves (ZVC) and Poincar  Section Points', 'FontSize', 16); % Make
    title larger
182 legend show;
183
184 % Set axis limits
185 xlim([-0.6, 0.8]); % Set x-axis range
186 ylim([-0.6, 0.6]);  % Set y-axis range
187
188 % Adjust tick sizes
189 set(gca, 'FontSize', 12); % Make tick labels larger
190
191 grid on;
192 hold off;
193
```

```
%%
% Parameters
tol = 1e-13; % Tolerance for Newton's method and integration
xmu = 0.1;   %    value
C = 4;       % Jacobi constant
h = 1e-3;    % Step size for integration
ysign = -1;  % y' sign
idir = +1;   % Forward integration
x0 = 3.2171; % Initial x value

% Step 1: Compute initial y_prime
y0_prime = dy_i(x0, xmu, C, ysign);

% Step 2: Compute time to Poincar  section and full period
[cross_points, times, ~, ~] = poincare(1, [x0, 0.0, 0.0, y0_prime], xmu, idir, h, tol);
T = times(1) * 2; % Full period

% Step 3: Refine x0 and y0_prime using Newton's method
[x, y_prime] = newton_variational(x0, y0_prime, xmu, T, tol);

% Step 4: Integrate the periodic orbit with variational equations
options = odeset('RelTol', tol, 'AbsTol', tol);
[~, X] = ode45(@(t, y) f_variational_eq(t, y, xmu, 1), [0, T], ...
    [x, 0, 0, y_prime, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], options);

% Step 5: Extract Monodromy Matrix (M)
M = reshape(X(end, 5:end), [4, 4]);

% Step 6: Compute Eigenvalues of M
eigenvalues = eig(M);

% Step 7: Display Results
fprintf('Refined Initial Conditions:\n');
fprintf('x0 = %.6f, y_prime = %.6f\n', x, y_prime);
fprintf('Full Period (T): %.6f\n', T);
fprintf('Monodromy Matrix (M):\n');
disp(M);
fprintf('Eigenvalues of M:\n');
disp(eigenvalues);
fprintf('Trace of M: %.6f\n', trace(M));
fprintf('Determinant of M: %.6f\n', det(M));

% Step 8: Plot Periodic Orbit
[t_orbit, orbit] = ode45(@(t, y) f(t, y, xmu, 1), [0, T], [x, 0, 0, y_prime], options);
figure;
plot(orbit(:, 1), orbit(:, 2), 'b-', 'LineWidth', 1.5);
xlabel('x');
ylabel('y');
title('Refined Periodic Orbit');
grid on;
%%
%%
function [x0,y0_prime]=newton_variational(x0,y0_prime,xmu,T,tol)
    deltaT=0.0001;
    options = odeset('RelTol',1e-13,'AbsTol',1e-13);  % ODE solver options for accuracy
    [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T/2], [x0 0 0
        y0_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
    x_prime=X(end,3);
    y=X(end,2);
    counter=1;
    while abs(x_prime)>tol || abs(y)>tol
        A=[X(end,9),X(end,12);X(end,13),X(end,16)];
        b=[-y;-x_prime];
        delta=A\b;
        x0=x0+delta(1);
        y0_prime=y0_prime+delta(2);
        [~,X]=ode45(@(t,y) f_variational_eq(t,y,xmu,1), [0,T/2], [x0 0 0
            y0_prime,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1], options);
```

```
        x_prime=X(end,3);
        y=X(end,2);
    end
end


% Function defining the system of differential equations
function df = f_variational_eq(~,x,mu,dir)
  df = zeros(20,1);
  r1 = sqrt((x(1)-mu)^2 + x(2)^2);
  r2 = sqrt((x(1)-mu+1)^2 + x(2)^2);

  df(1) = x(3);
  df(2) = x(4);
  df(3) = 2*x(4) + x(1) - ((1-mu)*(x(1)-mu)/(r1^3)) - mu*(x(1)-mu+1)/(r2^3);
  df(4) = -2*x(3) + x(2)*(1 - (1-mu)/(r1^3) - mu/(r2^3));

  df(5)=x(13);
  df(6)=x(14);
  df(7)=x(15);
  df(8)=x(16);
  df(9)=x(17);
  df(10)=x(18);
  df(11)=x(19);
  df(12)=x(20);
  df(13)=Omegaxx([x(1) x(2)],mu)*x(5)+Omegaxy([x(1) x(2)],mu)*x(9)+2*x(17);
  df(14)=Omegaxx([x(1) x(2)],mu)*x(6)+Omegaxy([x(1) x(2)],mu)*x(10)+2*x(18);
  df(15)=Omegaxx([x(1) x(2)],mu)*x(7)+Omegaxy([x(1) x(2)],mu)*x(11)+2*x(19);
  df(16)=Omegaxx([x(1) x(2)],mu)*x(8)+Omegaxy([x(1) x(2)],mu)*x(12)+2*x(20);
  df(17)=Omegaxy([x(1) x(2)],mu)*x(5)+Omegayy([x(1) x(2)],mu)*x(9)-2*x(13);
  df(18)=Omegaxy([x(1) x(2)],mu)*x(6)+Omegayy([x(1) x(2)],mu)*x(10)-2*x(14);
  df(19)=Omegaxy([x(1) x(2)],mu)*x(7)+Omegayy([x(1) x(2)],mu)*x(11)-2*x(15);
  df(20)=Omegaxy([x(1) x(2)],mu)*x(8)+Omegayy([x(1) x(2)],mu)*x(12)-2*x(16);
  if dir == -1
      df=-df;
  end
end

function df = f(~,x,mu,dir)
  df = zeros(4,1);
  r1 = sqrt((x(1)-mu)^2 + x(2)^2);
  r2 = sqrt((x(1)-mu+1)^2 + x(2)^2);

  df(1) = x(3);
  df(2) = x(4);
  df(3) = 2*x(4) + x(1) - ((1-mu)*(x(1)-mu)/(r1^3)) - mu*(x(1)-mu+1)/(r2^3);
  df(4) = -2*x(3) + x(2)*(1 - (1-mu)/(r1^3) - mu/(r2^3));

  if dir == -1
      df=-df;
  end
end

function [L3,C3,eigL3,veigL3]=RTBP_eq_points(xmu,tol)
    %%%%%L3%%%%%
    xi=1-(7*xmu)/(12);
    aux=0;
    while(abs(xi-aux)>tol)
        aux=xi;
        xi=F3(xi,xmu);
    end
    L3=[xmu+xi,0,0,0];
    xL3=L3(1);
    C3 = 2*Omega([xL3 0],xmu,abs(xL3-xmu),abs(xL3-xmu+1));
    DG3 = zeros(4,4); DG3(1,3)=1; DG3(2,4)=1; DG3(3,4)=2; DG3(4,3)=-2;
    DG3(3,1) = Omegaxx([xL3 0],xmu);
    DG3(3,2) = Omegaxy([xL3 0],xmu);
    DG3(4,1) = Omegaxy([xL3 0],xmu);
    DG3(4,2) = Omegayy([xL3 0],xmu);
```

```
330     [veigL3,eigL3] = eig(DG3);
331     eigp3=zeros(4,1);
332     for i = 1:4
333         eigp3(i)=eigL3(i,i);
334     end
335     eigL3=eigp3;
336 end
337
338
339 function res = F(x,xmu,dy)
340   [cross_points,~,~,~]=poincare(1,[x 0 0 dy],xmu,1,1e-2,1e-13);
341   res = cross_points(3);
342 end
343
344
345 function dy=dy_i(x,xmu,C,ysign)
346     r1 = sqrt((x-xmu)^2);
347     r2 = sqrt((x-xmu+1)^2);
348     dy= ysign*sqrt(2*Omega([x 0], xmu, r1, r2)-C);
349 end
350
351 function new=F1(old,xmu)
352     new = ((xmu*(1-old)^2)/(3-2*xmu-old*(3-xmu-old)))^(1/3);
353 end
354 function new=F2(old,xmu)
355     new = ((xmu*(1+old)^2)/(3-2*xmu+old*(3-xmu+old)))^(1/3);
356 end
357 function new=F3(old,xmu)
358     new = (((1-xmu)*(1+old)^2)/(1+2*xmu+old*(2+xmu+old)))^(1/3);
359 end
360
361 function res = Omega(x,mu,r1,r2)
362   res = (1/2)*(x(1)^2 + x(2)^2) + (1-mu)/r1 + mu/r2 + (1/2)*(mu*(1-mu));
363 end
364
365 function res = Omegaxx(x,mu)
366   res =
      (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)-(3*(2*mu-2*x(1))^2*(mu-1))/
      ...
367       +x(2)^2)^(5/2))+(3*mu*(2*x(1)-2*mu+2)^2)/(4*((x(1)-mu+1)^2+x(2)^2)^(5/2))+1;
368 end
369
370 function res = Omegayy(x,mu)
371   res =
      (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)+(3*mu*x(2)^2)/((x(1)-mu+1)
      ...
372       +x(2)^2)^(5/2)-(3*x(2)^2*(mu-1))/((mu-x(1))^2+x(2)^2)^(5/2)+1;
373 end
374
375 function res = Omegaxy(x,mu)
376   res = (3*x(2)*(2*mu-2*x(1))*(mu-1))/(2*((mu-x(1))^2+x(2)^2)^(5/2)) ...
377       +(3*mu*x(2)*(2*x(1)-2*mu+2))/(2*((x(1)-mu+1)^2+x(2)^2)^(5/2));
378 end
379
380 function val = g(x)
381 % in this case we want poincar  section to be x' = 0 and y'<0 (checked
382 % later)
383     val = x(3);
384 end
385
386
387 function [x1,x2]=varying(x,xmu,C,ysign,delta)
388     x1=x;
389     x2=x+delta;
390     dy1=dy_i(x1,xmu,C,ysign);
391     dy2=dy_i(x2,xmu,C,ysign);
392     while F(x1,xmu,dy1)*F(x2,xmu,dy2)>=0
393         x1=x2;
394         x2=x2+delta;
```

```
        dy1=dy2;
        dy2=dy_i(x2,xmu,C,ysign);
        %F(x1,xmu,dy1)
    end
end

function [c,t]=bisection_method(x1,x2,xmu,C,ysign,tol,max_counter)
    dy1=dy_i(x1,xmu,C,ysign);
    dy2=dy_i(x2,xmu,C,ysign);
    bool=0;
    if F(x1,xmu,dy1)==0
        c=x1;
    bool=1;
    end
    if F(x2,xmu,dy2)==0 && bool==0
        c=x2;
    bool=1;
    end
    counter=0;
    while abs(x2-x1)>tol && counter<max_counter && bool==0
        c=(x2+x1)/2;
        dy1=dy_i(x1,xmu,C,ysign);
        dyc=dy_i(c,xmu,C,ysign);
        if F(c,xmu,dyc)==0 && bool==0
            bool=1;
        end
        if bool==0
        if F(x1,xmu,dy1)*F(c,xmu,dyc) <0
            x2=c;
        else
            x1=c;
        end
        end
        counter=counter+1;
    end
    c=(x2+x1)/2;
    [~,t,~,~]=poincare(2,[c 0 0 dy_i(c,xmu,C,ysign)],xmu,1,1e-4,1e-13);
end

function
    [cross_points,total_time,cross_times,orbit]=poincare(n_crossing,x0,xmu,idir,h,tol)

options = odeset('RelTol',1e-10,'AbsTol',1e-10);  % ODE solver options for accuracy

% Initialize arrays to store crossing points and times
cross_points = zeros(n_crossing, 4);  % Pre-allocate for speed: stores [position,
    velocity]
total_time = 0;                       % Initialize total time counter
cross_times = zeros(n_crossing, 1);   % Pre-allocate for speed: stores crossing times
orbit=[x0'];
for i = 1:n_crossing
    % For subsequent crossings, the initial condition is the previous crossing point
    if i ~= 1
        x0 = cross_points(i-1, :);  % Set x0 to the previous crossing point
    end
    val=6;
    found=0;
    while found==0
    % Solve the ODE using ode45 from t=0 to t=h with initial condition x0
    [t, x1] = ode45(@(t,y) f(t,y,xmu,idir), [total_time,total_time+val], x0, options);
    index=2;
    size_sol=size(x1);
    while index<=size_sol(1)-1 && found==0
        if g(x1(index,:))*g(x1(index+1,:))<0
            found=1;
        else
        index=index+1;
        end
    end
```

```
    orbit=[orbit,x1(1:index,:)'];
    total_time=t(index);

    x0=x1(index,:);
    end
    % Approximate the crossing point
    approx = x1(index, :);
    % Refine the crossing point using Newton's method until the error is within tolerance
    counter=0;
    while abs(g(approx)) > tol
        counter=counter+1;
        % Compute correction delta using Newton's method
         r1 = sqrt((x0(1)-xmu)^2 + x0(2)^2);
         r2 = sqrt((x0(1)-xmu+1)^2 + x0(2)^2);

        derivative = 2*x0(4) + x0(1) - ((1-xmu)*(x0(1)-xmu)/(r1^3)) -
            xmu*(x0(1)-xmu+1)/(r2^3);
        delta = -g(approx) / derivative;
        total_time = total_time + idir * delta;  % Update the total time
        if counter==300 || abs(approx(4))<1e-8
            total_time=-1;
            return
        end
        % Choose the direction for ODE integration based on the sign of delta
        if delta < 0
            [t, new_approx] = ode45(@(t,y) f(t,y,xmu,-1), [0, abs(delta)], approx,
                options);
        end
        if delta > 0
            [t, new_approx] = ode45(@(t,y) f(t,y,xmu,1), [0, abs(delta)], approx,
                options);
        end
        approx = new_approx(end, :);  % Update the approximation

    end
    % Store the refined crossing point and time
    cross_points(i, :) = approx;
    cross_times(i) = total_time;
    orbit=[orbit,approx'];
    end
end
```

Listing 2: code

```
clc
clear all
close all
format long

xmu = 0.5;
ysign=-1;
delta=1e-3;
h=1e-3;
tol=1e-13;
max_iter=100;
n=200;
delta_s=1e-2;

[L1,C1,eigL1,L2,C2,eigL2,L3,C3,eigL3]=RTBP_eq_points(xmu,tol);
L4 =[ -0.5 + xmu,sqrt(3)/2];
C4= 2*Omega(L4,xmu,sqrt((L4(1)-xmu)^2 + L4(2)^2),sqrt((L4(1)-xmu+1)^2 + L4(2)^2));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C>C1%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C=C1+0.1;
C = C1+0.1;
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
```

```
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end
aux1=0;
aux2=0;
for y0=y
    aux2=2*Omega([xmu-1,y0],xmu,sqrt((xmu-1-xmu)^2 + y0^2),sqrt((xmu-1-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end
ic1=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C,tol,100);
ic2=bisection_method(xmu,list_changes(2),list_changes(2)-(y(2)-y(1)),xmu,C,tol,100);
ic3=bisection_method(xmu-1,list_changes(3),list_changes(3)-(y(2)-y(1)),xmu,C,tol,100);
orbit1=pseudoarc(xmu,ic1,xmu,C,tol, max_iter,delta_s,2000);
orbit2=pseudoarc(xmu,ic2,xmu,C,tol, max_iter,delta_s,2000);
orbit3=pseudoarc(xmu-1,ic3,xmu,C,tol, max_iter,delta_s,2000);

figure
plot(orbit1(:,1),orbit1(:,2),'blue')
hold on;
plot(orbit2(:,1),orbit2(:,2),'blue')
hold on;
plot(orbit3(:,1),orbit3(:,2),'blue')
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C>C1')


writematrix(orbit1, 'zvc_orbit1.txt', 'Delimiter', 'tab');
writematrix(orbit2, 'zvc_orbit2.txt', 'Delimiter', 'tab');
writematrix(orbit3, 'zvc_orbit3.txt', 'Delimiter', 'tab');



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C1%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C1;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

ic4=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C1,tol,100);
ic5=bisection_method(xmu,list_changes(2),list_changes(2)-(y(2)-y(1)),xmu,C1,tol,100);
orbit4=pseudoarc(xmu,ic4,xmu,C1,tol, max_iter,delta_s,2000);
orbit5=pseudoarc(xmu,ic5,xmu,C1,tol, max_iter,delta_s,2000);

figure
plot(orbit4(:,1),orbit4(:,2),'blue')
hold on;
plot(orbit5(:,1),orbit5(:,2),'blue')
```

```
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C1')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C2<C<C1%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C=(C2+C1)/2;
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

ic6=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C,tol,100);
ic7=bisection_method(xmu,list_changes(2),list_changes(2)-(y(2)-y(1)),xmu,C,tol,100);
orbit6=pseudoarc(xmu,ic6,xmu,C,tol, max_iter,delta_s,2000);
orbit7=pseudoarc(xmu,ic7,xmu,C,tol, max_iter,delta_s,2000);

figure
plot(orbit6(:,1),orbit6(:,2),'blue')
hold on;
plot(orbit7(:,1),orbit7(:,2),'blue')
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C2<C<C1')


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C2%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C2;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

ic8=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C2,tol,100);
orbit8=pseudoarc(xmu,ic8,xmu,C2,tol, max_iter,delta_s,3000);

figure
plot(orbit8(:,1),orbit8(:,2),'blue')
hold on;
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C2')



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C3<C<C2%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C=(C2+C3)/2;
number=10000;
```

```
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

ic9=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C,tol,100);
orbit9=pseudoarc(xmu,ic9,xmu,C,tol, max_iter,delta_s,5000);
figure
plot(orbit9(:,1),orbit9(:,2),'blue')
hold on;
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C3<C<C2')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C3%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C=C3;
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

ic10=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C,tol,100);
orbit10=pseudoarc(xmu,ic10,xmu,C,tol, max_iter,delta_s,5000);
figure
plot(orbit10(:,1),orbit10(:,2),'blue')
hold on;
scatter([xmu,xmu-1],0,'filled','red')
xlabel('x')
ylabel('y')
title('C3')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C4<C<C3%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C=(C3+C4)/2;
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end

y2=linspace(-5,0,number);
i=1;
```

```
aux1=0;
aux2=0;
for y0=y2
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end
ic11=bisection_method(xmu,list_changes(1),list_changes(1)-(y(2)-y(1)),xmu,C,tol,100);
orbit11=pseudoarc(xmu,ic11,xmu,C,tol, max_iter,delta_s,2000);
ic12=bisection_method(xmu,list_changes(end),list_changes(end)-(y2(2)-y2(1)),xmu,C,tol,100);
orbit12=pseudoarc(xmu,ic12,xmu,C,tol, max_iter,delta_s,2000);
figure
hold on;
plot(orbit11(:,1),orbit11(:,2),'blue')
hold on;
plot(orbit12(:,1),orbit12(:,2),'blue')
hold on;
scatter([xmu,xmu-1],0,'filled','red')
scatter([xmu-0.5,xmu-0.5],[sqrt(3)/2,-sqrt(3)/2],'filled','green')
xlabel('x')
ylabel('y')
title('C4<C<C3')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%C4%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C=C4;
number=10000;
y=linspace(0,5,number);
i=1;
aux1=0;
aux2=0;
list_changes=[];
for y0=y
    aux2=2*Omega([xmu,y0],xmu,sqrt((xmu-xmu)^2 + y0^2),sqrt((xmu-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end
y2=linspace(-5,0,number);
i=1;
aux1=0;
aux2=0;
for y0=y2
    aux2=2*Omega([0,y0],xmu,sqrt((0-xmu)^2 + y0^2),sqrt((0-xmu+1)^2 + y0^2))-C;
    if aux1*aux2<0
        list_changes=[list_changes, y0];
    end
    aux1=aux2;
end
figure
hold on;
scatter([xmu,xmu-1],0,'filled','red')
%scatter([xmu-0.5,xmu-0.5],[sqrt(3)/2,-sqrt(3)/2],'filled','green')
xlabel('x')
ylabel('y')
title('C4')
%%
xmu = 0.1;
C = 4;
tol = 1e-13;
max_iter = 500;
delta_t = 1e-3; % Time step for integration
num_iterates = 250; % Number of iterates for PSP


x_vals = linspace(-3, 3, 10); % Vary x along x-axis
```

```
initial_conditions = [];
for x = x_vals
    % Solve for y' using energy relation
    r1 = sqrt((x - xmu)^2);
    r2 = sqrt((x - xmu + 1)^2);
    y_prime_squared = 2 * Omega([x, 0], xmu, r1, r2) - C;
    if y_prime_squared > 0
        y_prime = -sqrt(y_prime_squared); % Select negative y'
        initial_conditions = [initial_conditions; [x, 0, 0, y_prime]];
    end
end


all_iterates = cell(size(initial_conditions, 1), 1);
for i = 1:size(initial_conditions, 1)
    ic = initial_conditions(i, :);
    iterates = compute_psp(ic, xmu, C, tol, max_iter, delta_t, num_iterates);
    all_iterates{i} = iterates;
end


figure;
hold on;
for i = 1:length(all_iterates)
    iterates = all_iterates{i};
    plot(iterates(:, 1), iterates(:, 2), ".");
end
xlabel('x');
ylabel('y');
title('Poincare Section Plot for C = 4');
grid on;
hold off;

%%

function c=bisection_method(x,y1,y2,xmu,C,tol,max_counter)
    bool=0;
    if G(x,y1,xmu,C)==0
        c=x1;
    bool=1;
    end
    if G(x,y2,xmu,C)==0 && bool==0
        c=x2;
    bool=1;
    end
    counter=0;
    while abs(y1-y2)>tol  && bool==0 && counter<max_counter
        c=(y2+y1)/2;
        if G(x,c,xmu,C)==0 && bool==0
            bool=1;
        end
        if bool==0
        if G(x,y1,xmu,C)*G(x,c,xmu,C) <0
            y2=c;
        else
            y1=c;
        end
        end
        counter=counter+1;
    end
    c=(y2+y1)/2;
end


function orbit = pseudoarc(x0,y0, xmu,C,tol, max_iter,delta_s,n)
    % Initialize variables
    r1 = sqrt((x0-xmu)^2 + y0^2);
    r2 = sqrt((x0-xmu+1)^2 + y0^2);
    x0_prime_aux=-(2*Omegay([x0,y0],xmu,r1,r2))/(2*Omegax([x0,y0],xmu,r1,r2));
```

```
    y0_prime_aux=1;
    x0_prime=x0_prime_aux/(norm([x0_prime_aux,y0_prime_aux]));
    y0_prime=y0_prime_aux/(norm([x0_prime_aux,y0_prime_aux]));
    x1=x0+delta_s*x0_prime;
    y1=y0+delta_s*y0_prime;
    orbit=zeros(n,2);
    for i=1:n %Newton_for

    % Initialize variables

    if i~=1
    x0=x1;
    y0=y1;
    x_prime_aux=x0_prime;
    y_prime_aux=y0_prime;
    r1 = sqrt((x0-xmu)^2 + y0^2);
    r2 = sqrt((x0-xmu+1)^2 + y0^2);
    x0_prime_aux=-(2*Omegay([x0,y0],xmu,r1,r2))/(2*Omegax([x0,y0],xmu,r1,r2));
    y0_prime_aux=1;
    x0_prime=x0_prime_aux/(norm([x0_prime_aux,y0_prime_aux]));
    y0_prime=y0_prime_aux/(norm([x0_prime_aux,y0_prime_aux]));
    x0_prime=x0_prime;
    y0_prime=y0_prime;
    if dot([x0_prime,y0_prime],[x_prime_aux,y_prime_aux])<0
        x0_prime=-x0_prime;
        y0_prime=-y0_prime;
    end
    x1=x0+delta_s*x0_prime;
    y1=y0+delta_s*y0_prime;

    end

    iter = 0; % Iteration counter
    not_finished=true;
    while not_finished && iter < max_iter
        % Evaluate F and J at current X
        FX = F(x1,y1,x0,y0,x0_prime,y0_prime,xmu,C,delta_s);
        JX = J(x1,y1,x0,y0,x0_prime,y0_prime,xmu,C,delta_s);

        % Solve the linear system J(X) * Delta = -F(X) for Delta
        Delta = -JX \ FX; % MATLAB's backslash operator for solving linear systems
        % Update the solution
        x1 = x1 + Delta(1);
        y1= y1+Delta(2);

        % Check for convergence
        if norm(FX, 'fro') < tol
            not_finished=false;
        end

        % Increment iteration counter
        iter = iter + 1;
    end
    % If max iterations are reached without convergence, warn the user
    orbit(i,:)=[x1,y1];
    end
end

function res=G(x,y,xmu,C)
    r1 = sqrt((x-xmu)^2 + y^2);
    r2 = sqrt((x-xmu+1)^2 + y^2);
    res=2*Omega([x y],xmu,r1,r2)-C;
end
function matrix = F(x1,y1,x0,y0,x0_prime,y0_prime,xmu,C,delta_s)
    matrix=[G(x1,y1,xmu,C);((x1-x0)*x0_prime+(y1-y0)*y0_prime)-delta_s];
end
function matrix=J(x1,y1,x0,y0,x0_prime,y0_prime,xmu,C,delta_s)
r1 = sqrt((x1-xmu)^2 + y1^2);
r2 = sqrt((x1-xmu+1)^2 + y1^2);
```

```
matrix=[2*Omegax([x1,y1],xmu,r1,r2),2*Omegay([x1,y1],xmu,r1,r2);x0_prime,y0_prime];
end




function [L1,C1,eigL1,L2,C2,eigL2,L3,C3,eigL3]=RTBP_eq_points(xmu,tol)
    %%%%%L1%%%%%
    xi=(xmu/(3*(1-xmu)))^(1/3);
    aux=0;
    while(abs(xi-aux)>tol)
        aux=xi;
        xi=F1(xi,xmu);
    end
    L1=[xmu-1+xi,0,0,0];

    %%%%%L2%%%%%
    xi=(xmu/(3*(1-xmu)))^(1/3);
    aux=0;
    while(abs(xi-aux)>tol)
        aux=xi;
        xi=F2(xi,xmu);
    end
    L2=[xmu-1-xi,0,0,0];

    %%%%%L3%%%%%
    xi=1-(7*xmu)/(12);
    aux=0;
    while(abs(xi-aux)>tol)
        aux=xi;
        xi=F3(xi,xmu);
    end
    L3=[xmu+xi,0,0,0];
    xL1=L1(1);
    xL2=L2(1);
    xL3=L3(1);

    C1 = 2*Omega([xL1 0],xmu,abs(xL1-xmu),abs(xL1-xmu+1));
    C2 = 2*Omega([xL2 0],xmu,abs(xL2-xmu),abs(xL2-xmu+1));
    C3 = 2*Omega([xL3 0],xmu,abs(xL3-xmu),abs(xL3-xmu+1));

    DG1 = zeros(4,4); DG1(1,3)=1; DG1(2,4)=1; DG1(3,4)=2; DG1(4,3)=-2;
    DG2 = DG1; DG3 = DG1;
    DG1(3,1) = Omegaxx([xL1 0],xmu);
    DG1(3,2) = Omegaxy([xL1 0],xmu);
    DG1(4,1) = Omegaxy([xL1 0],xmu);
    DG1(4,2) = Omegayy([xL1 0],xmu);
    eigL1 = eig(DG1);
    DG2(3,1) = Omegaxx([xL2 0],xmu);
    DG2(3,2) = Omegaxy([xL2 0],xmu);
    DG2(4,1) = Omegaxy([xL2 0],xmu);
    DG2(4,2) = Omegayy([xL2 0],xmu);
    eigL2 = eig(DG2);
    DG3(3,1) = Omegaxx([xL3 0],xmu);
    DG3(3,2) = Omegaxy([xL3 0],xmu);
    DG3(4,1) = Omegaxy([xL3 0],xmu);
    DG3(4,2) = Omegayy([xL3 0],xmu);
    eigL3 = eig(DG3);
end

function res=Omegax(x,mu,r1,r2)
res=x(1) - ((1-mu)*(x(1)-mu)/(r1^3)) - mu*(x(1)-mu+1)/(r2^3);
end
function res=Omegay(x,mu,r1,r2)
res=+ x(2)*(1 - (1-mu)/(r1^3) - mu/(r2^3));
end

function new=F1(old,xmu)
    new = ((xmu*(1-old)^2)/(3-2*xmu-old*(3-xmu-old)))^(1/3);
```

```
end
function new=F2(old,xmu)
    new = ((xmu*(1+old)^2)/(3-2*xmu+old*(3-xmu+old)))^(1/3);
end
function new=F3(old,xmu)
    new = (((1-xmu)*(1+old)^2)/(1+2*xmu+old*(2+xmu+old)))^(1/3);
end

function res = Omega(x,mu,r1,r2)
  res = (1/2)*(x(1)^2 + x(2)^2) + (1-mu)/r1 + mu/r2 + (1/2)*(mu*(1-mu));
end

function res = Omegaxx(x,mu)
  res =
    (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)-(3*(2*mu-2*x(1))^2*(mu-1))
end

function res = Omegayy(x,mu)
  res =
    (mu-1)/((mu-x(1))^2+x(2)^2)^(3/2)-mu/((x(1)-mu+1)^2+x(2)^2)^(3/2)+(3*mu*x(2)^2)/((x(1)-mu+1)
end

function res = Omegaxy(x,mu)
  res =
    (3*x(2)*(2*mu-2*x(1))*(mu-1))/(2*((mu-x(1))^2+x(2)^2)^(5/2))+(3*mu*x(2)*(2*x(1)-2*mu+2))/(2*
end
```