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).

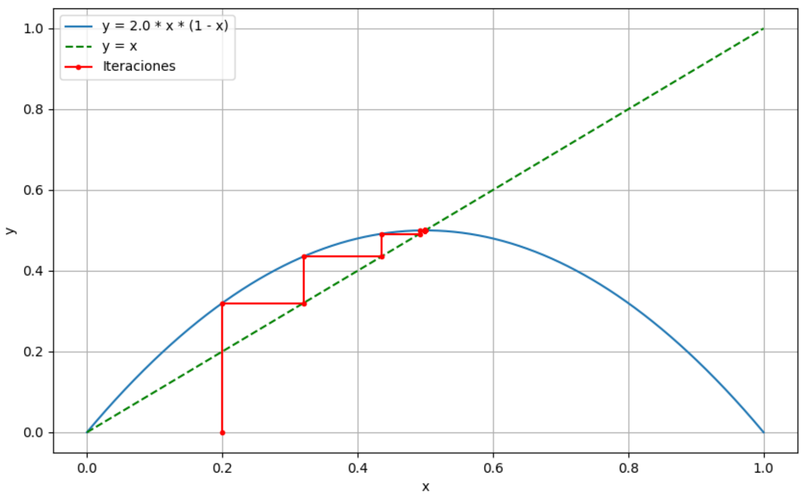


Figure 1: Logistic map for a = 2.0 and $x_0 = 0.2$.

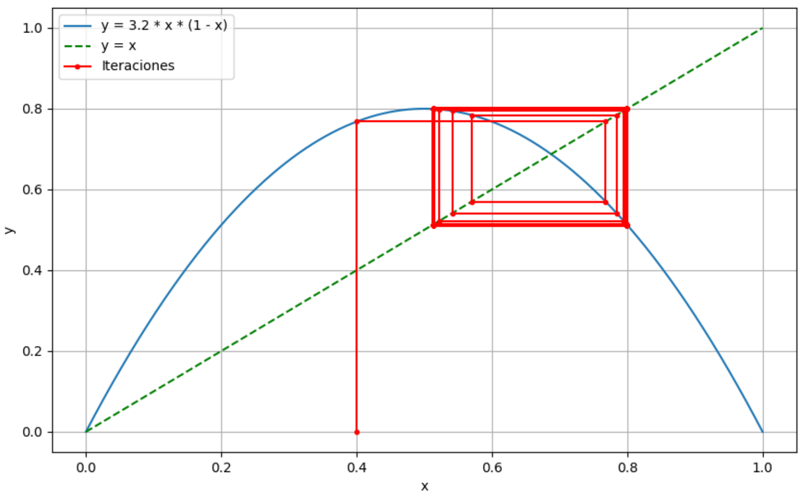


Figure 2: Logistic map for a = 3.2 and x_0 = 0.4.

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 $x_0 \in [-\pi, \pi]$.

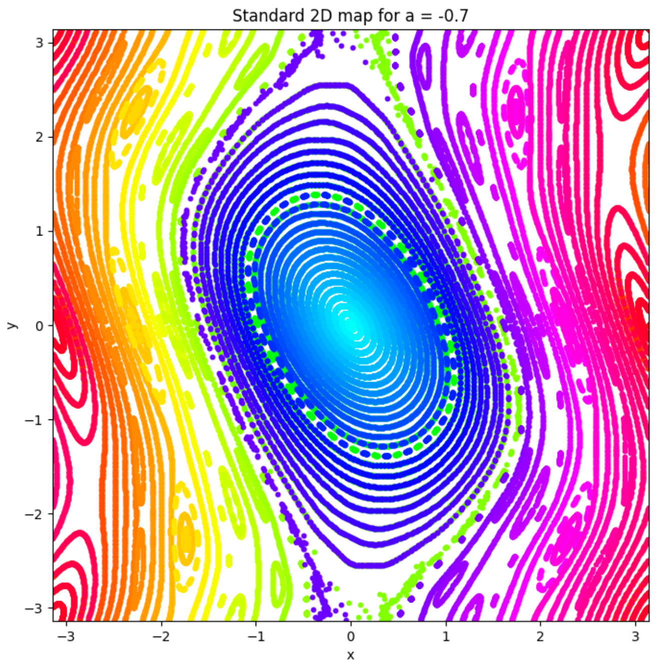


Figure 3: Standard 2D map with 500 iterations and 100 initial conditions for a = -0.7.

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:

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

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

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

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.

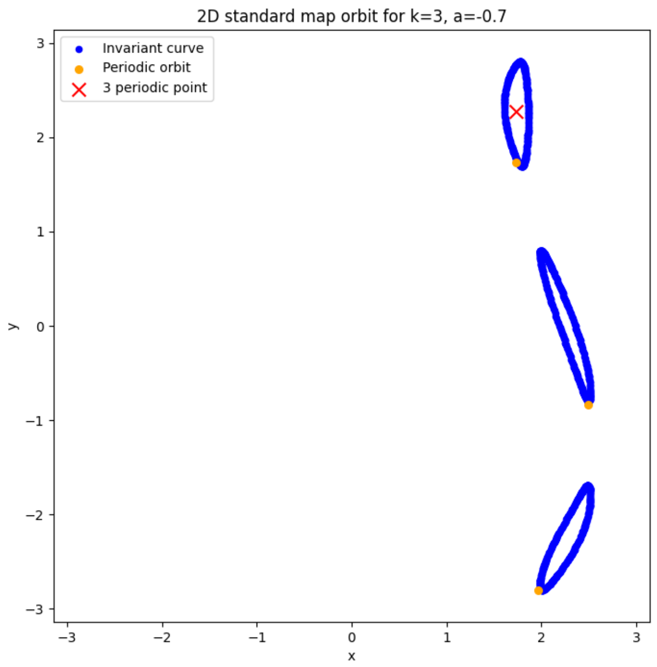


Figure 4: Associated orbit and invariant curve around a 3-periodic point.

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.

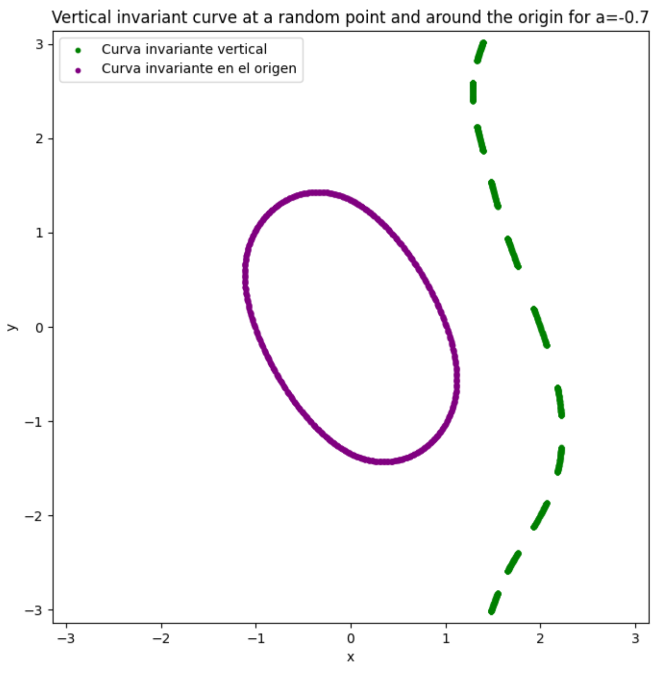
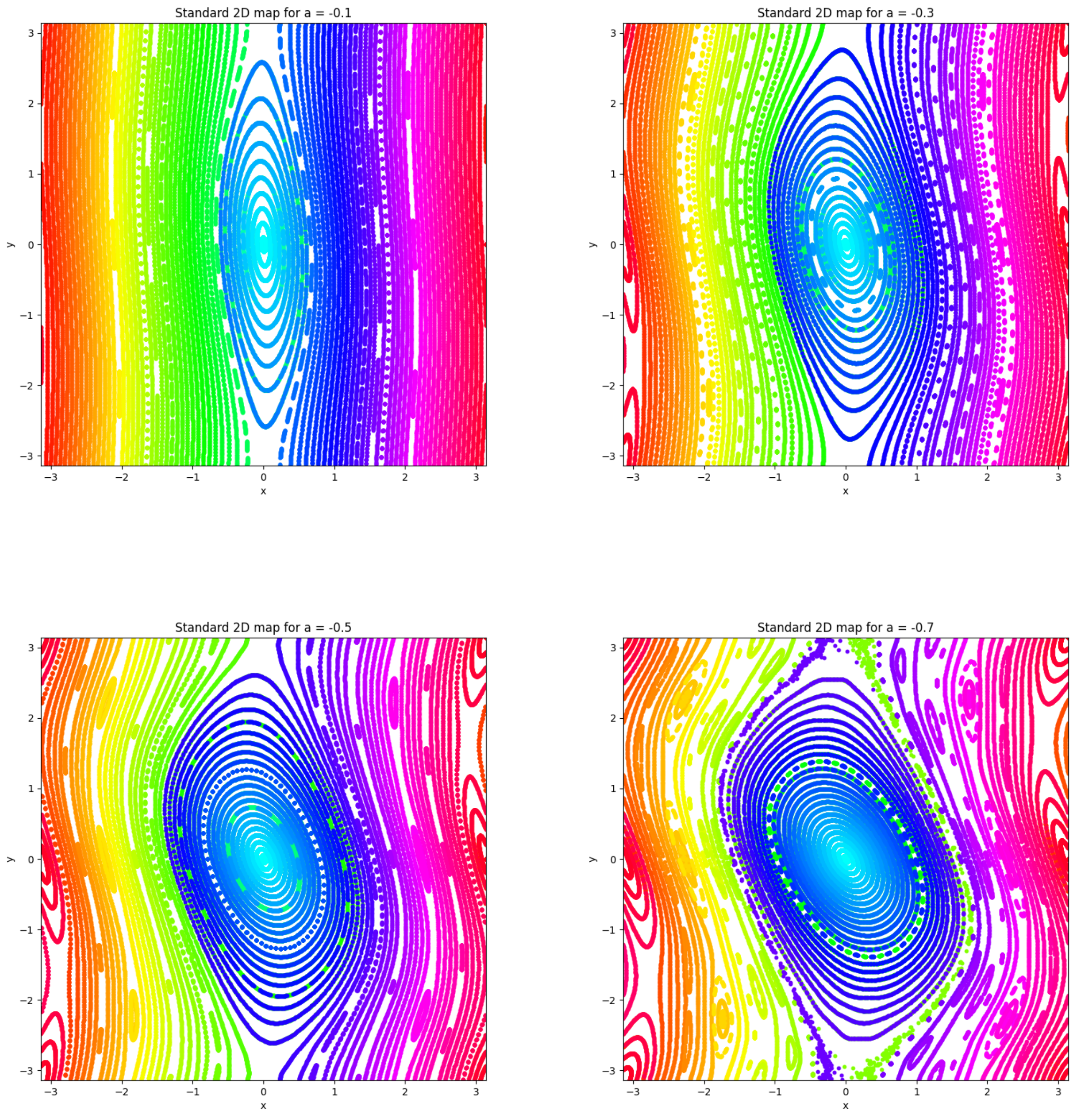


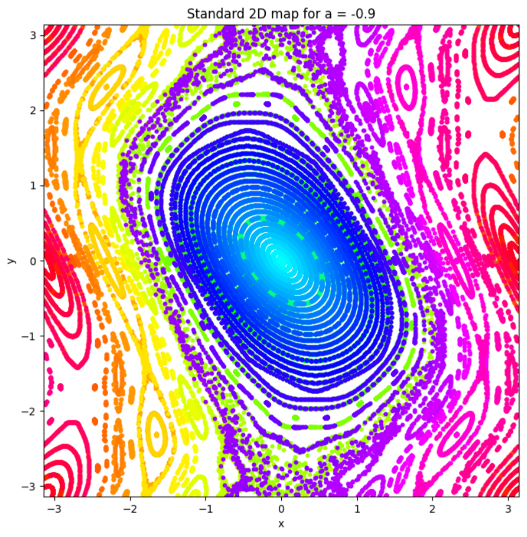
Figure 5: Vertical invariant curve at a random point and around the origin.

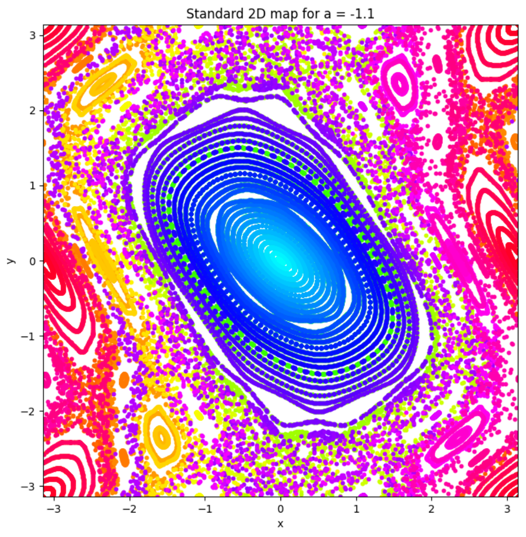
First, we know that the origin is a fixed point. However, if we move the initial condition, the system describes an invariant orbit around it, making it an attractor point. For a body near the origin, the system will keep it near the center, like a stable planetary orbit.
Studying now the 3-periodic 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.

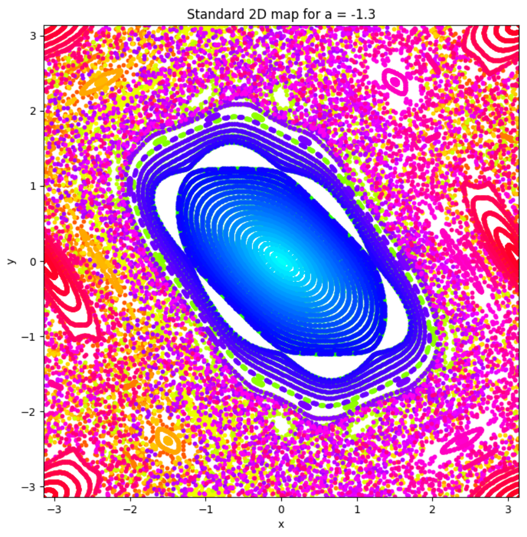
2.4 Simulation varying parameter a

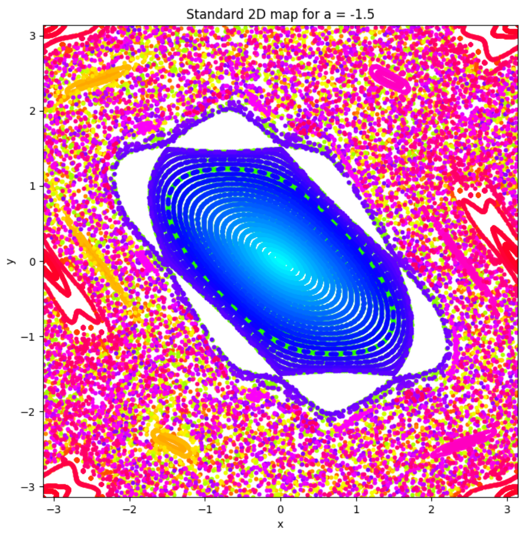
we have worked with a fixed parameter a = -0.7. However, we want to study the behavior of the 2D standard map as a function of the parameter *a*. The next subplot shows how the plot changes depending on this parameter.

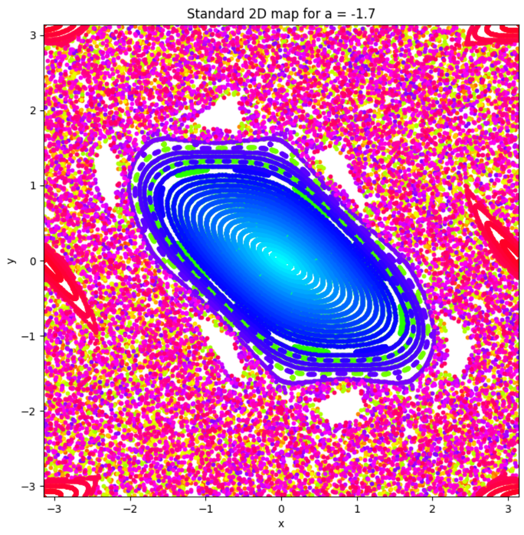


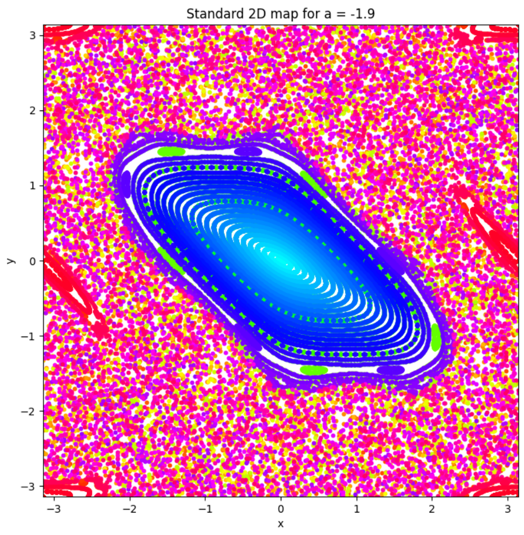

Standard 2D map for a = -0.9
y
x
−3
−2
−1
0
1
2
3


Standard 2D map for a = -1.1
y
x
−3
−2
−1
0
1
2
3


Standard 2D map for a = -1.3
y
x
−3
−2
−1
0
1
2
3


Standard 2D map for a = -1.5
y
x
−3
−2
−1
0
1
2
3





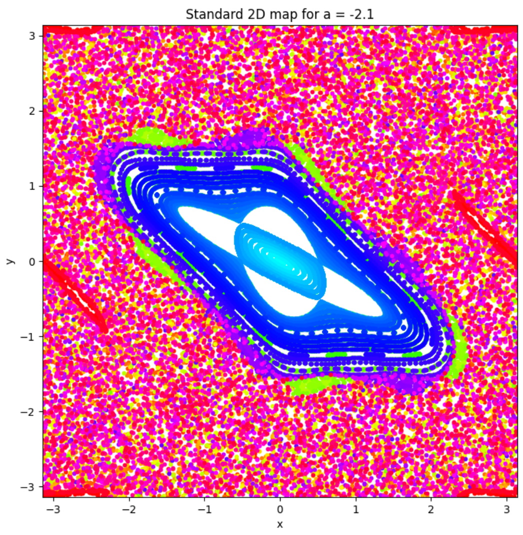


Figure 6: 2D Standard Map for multiple values of a.

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()
```