Juan Carlos Díaz Rodríguez
Subgrupo U1
juancdia@ucm.es

Geometría Computacional
Práctica 3

07/04/2023

1. Introducción

En esta práctica estudiaremos la evolución de un sistema dinámico de la forma $\ddot{q} = F(q)$ mediante la discretización del sistema. Con ello representaremos las distintas órbitas para conjuntos de datos iniciales y trataremos de comprobar que se cumple el Teorema de Liouville mediante aproximaciones sencillas de la integral que nos da el área.

2. Material utilizado

El sistema con el que vamos a trabajar es el oscilador no lineal. Comenzamos considerando el hamiltoniano

$$H : \mathbb{R}^2 \to \mathbb{R}, \quad (q,p) \mapsto p^2 + \frac{1}{a}\left(q^2 - b^2\right),$$

con $a, b \in \mathbb{R}$. Siguiendo con las ecuaciones de Hamilton-Jacobi, obtenemos el sistema

$$\ddot{q} = -\frac{8}{a} q \left(q^2 - b\right).$$

Todos nuestros cálculos serán hechos con $a = 3$ y $b = \frac{1}{2}$.

Dicho esto, ahora tenemos que discretizar el problema. Para ello tomamos un parámetro de granularidad δ y representaremos el tiempo como $t = n\delta$.

Una opción es aproximar la evolución del sistema tomando q_0 como el valor inicial dado, $q_1 = q_0 + 2\delta p_0$ y $q_{n+2} = \delta^2 F(q_n) + 2q_{n+1} - q_n$. Una vez tenemos la sucesión q_n podemos obtener las derivadas por cocientes incrementales. Sin embargo, con muy poco esfuerzo, podemos convertir este sistema de segundo orden en un sistema de primer orden e implementar el método de Runge-Kutta de orden 4. Las soluciones que obtengamos aquí serán la sucesión $\{(q_n, 2p_n)\}_{n=0}^{N}$.

Para representar el espacio de fases, y para el resto de apartados, resolveremos una única vez el sistema almacenando todos los datos en un array, dado que el tamaño de este es considerable. Para ello se emplea la librería de `numpy`, así como `matplotlib` para todas las representaciones gráficas y `scipy` para calcular el área del apartado ii). Cabe remarcar que una aproximación sencilla para calcular un área es calcular el área de la envolvente convexa de nuestros puntos. Esto tiene unos evidentes problemas cuando el conjunto considerado no es convexo. Para solventar esto, aunque solo funcione bajo unas condiciones de baja deformación dentro del conjunto, tomaremos subdivisiones del conjunto D_t y sumaremos las áreas de las envolvente convexas de dichas subdivisiones.

3. Resultados

Se presentan a continuación los resultados referentes a los distintos apartados.

Primero mostramos una comparativa de la precisión de resolución para disintos δ empleando los dos métodos descritos.

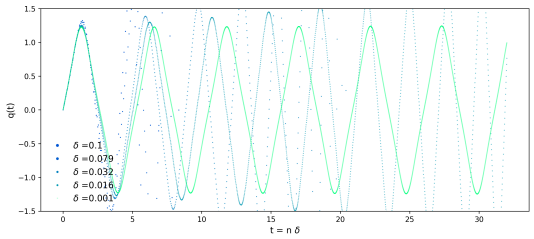


Figura 1: Resolución mediante el primer método

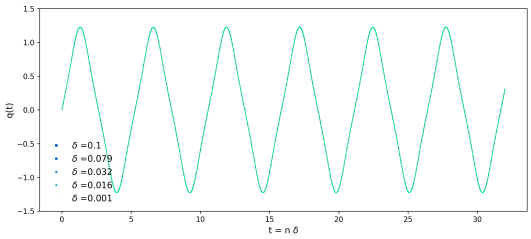
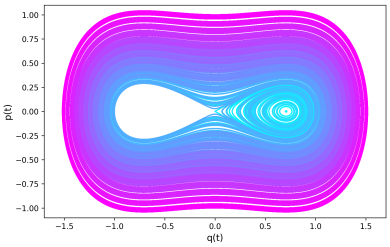


Figura 2: Resolución por Runge-Kutta de orden 4

Podemos apreciar cómo resulta mucho más preciso el segundo enfoque, por lo que serán esos datos los que usaremos para resolver el resto de apartados. Dado que no se requiere un paso muy fino tomamos $\delta = 10^{-3}$.

Apartado i)

Figura 3: Espacio de fases



Para representar el diagrama de fases tomamos el cuadrado $[0,1] \times [0,2]$ y lo discretizamos tomando una malla de 15×15 puntos. Denotamos a este conjunto de puntos por D_0 Calculamos las soluciones del sistema hasta $t = 50$ para cada $(q_0, 2p_0) \in D_0$ y las representamos en la figura 3. Podemos apreciar que todas las órbitas calculadas son periódicas. Además, aunque no aparecen representadas, tenemos la misma forma de lágrimas en el lóbulo para $q < 0$ para las órbitas contenidas en esa región, dado que el espacio de fases es simétrico. Puesto que todas las órbitas son periódicas esto nos asegura que para t suficientemente grande (donde $t = 50$ ya lo es) habremos cubierto $D_{(0,\infty)}$.

Apartado ii)

n_{subs}	delta = 0.001
1	1.0835
2	1.0442
3	1.0289
4	1.0232
5	1.0168

Figura 4: $V_{1/4}$ para distintas subdivisiones

Para calcular V_t, realizaremos una subdivisión de D_0 y para cada uno de esos subconjuntos de valores iniciales calculamos la envoltura convexa de su evolución. En la figura 6 podemos ver el mallado empleado para la aproximación del área para $t = \frac{1}{4}$ y en la figura 5 la envoltura convexa sin realizar las subdivisiones. Aunque algunas envolturas se solapan en la figura 6, la aproximación del área es mejor que tomar una única envoltura convexa. De esta forma obtenemos $V_0 = 1$ y $V_{1/4} = 1{,}0168$. Esto, asumiendo que nuestros cálculos tienen un cierto error, nos indica que **se cumple el Teorema de Liouville**, como no podía ser de otra forma. Para tratar de estimar el error podemos variar δ y realizar los mismo pasos. Sin embargo, dado que la resolución por Runge-Kutta ya es bastante excata, el valor de $V_{1/4}$ para $\delta = 10^{-4}$ coincide con el valor anterior hasta el 16º decimal. Podemos ver sin embargo, en la figura 4 cómo se reduce el error según aumentamos la cantidad de subdivisiones que hacemos. Para $n_{subs} = 5$ estamos realizando una malla como la de la figura 6.

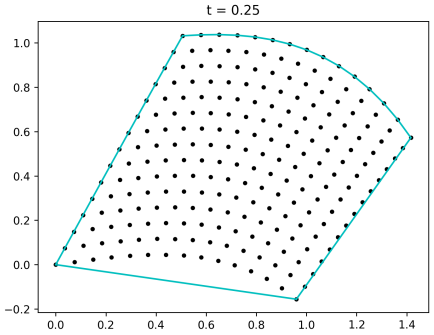


Figura 5: Envoltura convexa de $D_{1/4}$

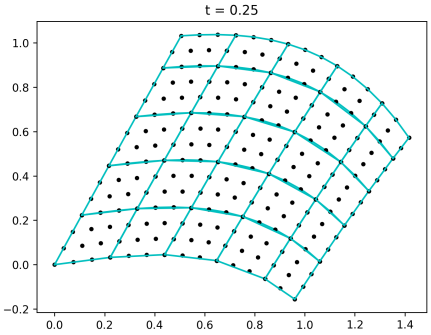


Figura 6: Envoltura convexa de $D_{1/4}$ dividido en 25 subconjuntos

Apartado iii)

Para la animación final se computaron soluciones únicamente en el borde del anterior D_0. Se tomaron alrededor de 2000 puntos. La densidad de los puntos tomados depende de cuánto se deforme el conjunto, de modo que la animación sea lo más suave posible en todo el borde. Se pueden encontrar más detalles en la función `phaseSpaceEdge` y la animación en el vídeo adjunto

4. Conclusión

De esta práctica podemos aprender lo siguiente:

- Los métodos explícitos de resolución numérica de ecuaciones diferenciales nos permiten obtener soluciones de alta precisión con poco esfuerzo de programación.
- Es importante que, además de tener soluciones precisas para resolver una ecuación, obtengamos métodos fiables para el resto de análisis que realicemos, como calcular el área del espacio de fases.
- Para una animación, cuantos más puntos mejor, pero si el esfuerzo de cómputo es demasiado grande, se puede interpolar y variar la densidad del muestreo según la deformación del conjunto en cada punto.

5. Código

```
import os
import numpy as np
import matplotlib as mpl
mpl.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'
from matplotlib import pyplot as plt
from matplotlib import animation
from scipy.spatial import ConvexHull
#https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

# Resolutor por Runge-Kutta de orden 4 de una ecuación diferencial
# del tipo \ddot{x} = F(x(t))
def orbRK4(n,q0,dq0,F,d=0.001):
    f = lambda x : np.array([x[1], F(x[0])])

    q_dq = np.empty([n+1, 2])
    q_dq[0] = np.array([q0, dq0])
    for i in range(n):
        eval1 = f(q_dq[i])
        eval2 = f(q_dq[i] + d/2*eval1)
        eval3 = f(q_dq[i] + d/2*eval2)
        eval4 = f(q_dq[i] + d*eval3)
        q_dq[i+1] = q_dq[i] + d/6*(eval1 + 2*eval2 + 2*eval3 + eval4)

    return q_dq

def phaseSpace(D0_q, D0_dq, F, delta, n):
    PS = np.empty((n+1, len(D0_dq) * len(D0_q), 2), dtype=np.float64)
    index = 0
    for i in range(len(D0_dq)):
      for j in range(len(D0_q)):
        PS[:,index] = orbRK4(n, D0_q[j], D0_dq[i], F, d=delta)
        index += 1
    return PS

DPI = 250

# Ajustes iniciales
a = 3.
b = .5
delta = 10.**-3
#Condiciones iniciales:
lenQ = 15
lenDQ = 15
D0_q = np.linspace(0.,1.,num=lenQ)
D0_dq = np.linspace(0.,2.,num=lenDQ)

# Apartado i)
def F_oscilador_no_lineal(q: np.array) -> np.array:
    return -8./a * q * (q*q -  b)

# Carga o cálculo de los datos
loadFile = f'oscilador_no_lineal_a={a}_b={b}_delta={delta}.npy'

Horiz = 50
n = int(Horiz/delta)
if os.path.exists(loadFile):
    print('loading', end='\r')
    phase_space = np.load(loadFile)
else:
```

```
    print('Not found, calculating phase_space...', end='\r')
    phase_space = phaseSpace(D0_q, D0_dq, F_oscilador_no_lineal, delta, n)
    phase_space[:,:,1] /= 2
    np.save(loadFile, phase_space)
print('Finished phase_space, shape=', phase_space.shape, '                              ')

# Representamos el espacio de fases
fig1 = plt.figure(figsize=(8,5))
fig1.subplots_adjust(hspace=0.4, wspace=0.2)
ax = fig1.add_subplot(1,1, 1)
ax.set_xlim(-1.7, 1.7)
ax.set_ylim(-1.1, 1.1)
index = 0
for j in range(len(D0_q)):
    for i in range(len(D0_dq)):
        q = phase_space[:,index,0]
        p = phase_space[:,index,1]
        col = (1+i+j*(len(D0_q)))/(len(D0_q)*len(D0_dq))
        plt.plot(q[:], p[:], '-',c=plt.get_cmap("cool")(col))
        index += 1
ax.set_xlabel("q(t)", fontsize=12)
ax.set_ylabel("p(t)", fontsize=12)
fig1.savefig('Simplectic.png', dpi=DPI)

# Apartado ii)
def D_t(phase_space, t, delta):
    n_orig = phase_space.shape[0]
    n = int(t/delta)
    if n >= n_orig:
        print(phase_space.shape)
        print(f't={t}')
        print(f'delta={delta}')
        print(f'n={n}')
        assert(False)
    Dt = phase_space[n]
    return Dt

def divide_D_t(phase_space, t, delta, lenQ, lenDQ, n_subsQ, n_subsDQ):
    d_t = D_t(phase_space, t, delta)
    assert(tuple(d_t.shape) == (lenQ * lenDQ, 2))
    matrix = d_t.reshape((lenDQ, lenQ, 2))
    subsQ = subdivisions(lenQ, n_subsQ)
    subsDQ = subdivisions(lenDQ, n_subsDQ)
    subdivided_d_t = []
    for subDQ in subsDQ:
        for subQ in subsQ:
            aux = matrix[subDQ[0]:subDQ[1], subQ[0]:subQ[1]]
            subdivided_d_t.append(aux.reshape((aux.shape[0]*aux.shape[1], aux.shape[2])))
    return subdivided_d_t

# Función auxiliar que nos da los índices con los que
# subdividir una matriz de puntos para poder calcular
# los convex hull por subconjuntos
def subdivisions(lenQ, n_subsQ):
    assert(lenQ >= n_subsQ)
    if n_subsQ == 1:
        return [(0,lenQ)]
    step = lenQ/n_subsQ
    aux = [int(step*n) for n in range(n_subsQ)]
    subs = [(aux[i],aux[i+1]+1) for i in range(len(aux)-1)]
    if subs[-1][1] != lenQ:
```

```
        subs.append((subs[-1][1]-1, lenQ))
    return subs

# Cálculo de 2-n volumen subdividiendo D_t en n_subsQ x n_subsDQ submatrices
# el conjunto D_t
def V_t(phase_space, t, delta, lenQ, lenDQ, n_subsQ, n_subsDQ):
    D = divide_D_t(phase_space, t, delta, lenQ, lenDQ, n_subsQ, n_subsDQ)
    hulls = [ConvexHull(d_t) for d_t in D]
    return sum([hull.volume for hull in hulls])

# Para evitar copiar y pegar código
def ejecutar_apartado_2(delta):
    for n_subs in range(1,6):
        v_0 = V_t(phase_space, 0, delta, lenQ, lenDQ, n_subs, n_subs)
        v_un_cuarto = V_t(phase_space, .25, delta, lenQ, lenDQ, n_subs, n_subs)

        print(f'V(0) = {v_0} para delta ={delta} subdividido en {n_subs}x{n_subs} subconjuntos')
        print(f'V(1/4) = {v_un_cuarto} para delta = {delta} subdividido en {n_subs}x{n_subs} subconjun

    D_un_cuarto_sub = divide_D_t(phase_space, .25, delta, lenQ, lenDQ, 5, 5)
    Hulls_D_un_cuarto = [ConvexHull(d_t) for d_t in D_un_cuarto_sub]
    fig, ax = plt.subplots()
    for points, hull in zip(D_un_cuarto_sub, Hulls_D_un_cuarto):
        ax.plot(points[:, 0], points[:, 1], '.', color='k')
        ax.set_title('t = 0.25')
        for simplex in hull.simplices:
            ax.plot(points[simplex, 0], points[simplex, 1], 'c')
    fig.savefig(f'divided_hulls_t=0.25_delta={delta}_a={a}_b={b}_1.png', dpi=DPI)

# Calculamos los 2-n volumenes para delta = 10**-3
ejecutar_apartado_2(delta)

# Repetimos los cálculos para un delta más fino, pero menor H para ahorrar tiempo de la computadora
delta = 10.**-4
loadFile = f'oscilador_no_lineal_a={a}_b={b}_delta={delta}.npy'

Horiz = 0.26
n = int(Horiz/delta)
if os.path.exists(loadFile):
    print('loading', loadFile, end='\r')
    phase_space = np.load(loadFile)
else:
    print('Not found, calculating phase_space...', end='\r')
    phase_space = phaseSpace(D0_q, D0_dq, F_oscilador_no_lineal, delta, n)
    phase_space[:,:,1] /= 2
    np.save(loadFile, phase_space)
print('Finished phase_space, shape=', phase_space.shape, '                          ')

ejecutar_apartado_2(delta)


# Apartado iii)
from matplotlib.collections import LineCollection

# Funciones auxiliares para poder guardar las animaciones
# tanto en gif como en mp4 u otros formatos de video similares
def save_gif(fileName, anim, fps):
    writergif = animation.PillowWriter(fps=fps)
    anim.save(fileName, writer=writergif)

def save_video(fileName, anim, fps):
```

```
    try:
        writervideo = animation.FFMpegWriter(fps=fps)
        anim.save(fileName, writer=writervideo)
    except FileNotFoundError as e:
        print(e)
        print("Check if you have downloaded FFMPEG and given the path to executable\
            or binary (depending on OS) to matplotlib on line 4")

# Esta forma de crear el espacio de fases quiza merece una pequeña explicación
def phaseSpaceEdge(n_points, F, delta, n):
    """
    Aquí se crea un "espacio de fases" donde en vez de
    evaluar en una cuadrícula, evaluamos solo en el perímetro
    del cuadrado [0,1]x[0,2]. Sin embargo, lo que vamos a hacer
    es poner más puntos en las regiones que más se deforman a
    la hora de hacer la animación. Esto son las esquina
    inferior izquierda sobre todo y también la inferior derecha.

    El coeficiente primer coeficiente por el que multipliquemos ro
    será lo que estamos aumentando la densidad de punto, y el segundo
    coeficiente la longitud del intervalo que estamos tratando.
    Nótese que en las aristas verticales esta longitud es la mitad, pues
    luego esto se convertirá en el cuadrado [0,1]x[0,1].

    Los colores se añaden para que el gradiente parezca uniforme en el cuadrado
    y no se note a primera vosta dónde hay una mayor concentración de puntos

    El array final no tendrá exactamente n_points soluciones, pero más o menos
    """
    PS = [] # Array con los valores de las soluciones
    ro = n_points / (20*0.2+8*0.8+2*0.5+1.75+2*0.75)
    colors = [] # valores para el gradiente de colores

    def append_linspace(edge, start, end, density, length):
        aux = list(np.linspace(start,end,int(density*length*ro)))
        if aux[-1] == end:
            aux = aux[:-1]
        return edge + aux

    # bottom edge
    bot   = append_linspace([],   0 , .10,20, .10)
    bot   = append_linspace(bot, .10, .25, 8, .15)
    bot   = append_linspace(bot, .25, .75, 2, .50)
    bot   = append_linspace(bot, .75, 1,   8, .25)
    for value in bot:
        colors.append(value / 2)

    # right edge
    right = append_linspace([],     0, .5, 8, .25)
    right = append_linspace(right, .5, 2,  1, .75)
    for value in right:
        colors.append(0.5 + value / 4)

    # top edge
    top   = append_linspace([],    1, 0,   1, 1  )
    for value in top:
        colors.append(0.5 + value / 2)
    # left edge
    left  = append_linspace([],    2, .5,  2, .75)
    left  = append_linspace(left, .5, .2,  8, .15)
    left  = append_linspace(left, .2,  0, 20, .1 )
```

```
    for value in left:
        colors.append(value / 4)

    for j in bot:
        PS.append(orbRK4(n, j, 0.0, F, d=delta))
    for i in right:
        PS.append(orbRK4(n, 1.0, i, F, d=delta))
    for j in top:
        PS.append(orbRK4(n, j, 2.0, F, d=delta))
    for i in left:
        PS.append(orbRK4(n, 0.0, i, F, d=delta))

    PS = np.array(PS).swapaxes(0,1)
    return PS, colors

# Definimos los parámetros con que vamos a calcular los puntos
delta = 10.**-3
t_max = 5
n_max = int(t_max / delta) + 1

# Carga o cálculo de los datos
loadFile = f'oscilador_no_lineal_a={a}_b={b}_delta={delta}_edge.npy'
if os.path.exists(loadFile):
    print('loading', loadFile, end='\r')
    phase_space = np.load(loadFile)
    colors = np.load('colors.npy')
else:
    print('Not found, calculating phase_space...', end='\r')
    phase_space, colors = phaseSpaceEdge(2000, F_oscilador_no_lineal, delta, n_max)
    phase_space[:,:,1] /= 2
    np.save(loadFile, phase_space)
    colors = np.array(colors)
    np.save('colors.npy', colors)
print('Finished phase_space, shape=', phase_space.shape, '                              ')

# Parámetros de la animación
duration = 10
FPS = 30
anim_delta = t_max / (FPS * duration)
t_evaluation = np.arange(0, t_max + anim_delta, anim_delta)
D = np.array([D_t(phase_space, t, delta) for t in t_evaluation])

X_SIZE = 10
Y_SIZE = 10

import matplotlib.cm as cm
# Creamos el array de colores con los valores que nos dio la función de arriba
colors = cm.rainbow(colors)

fig, ax = plt.subplots(figsize=(9,6))

# Tomamos la órbita más externa que acota el espacio de fases y la representamios en negro
outer_orbit = orbRK4(n_max, 1.0, 2.0, F_oscilador_no_lineal, delta)
ax.plot(outer_orbit[:,0], outer_orbit[:,1]/2, '-', color='black')

# Creamos los segmentos que representaremos
d_shape = D.shape
segs = np.empty((d_shape[0], d_shape[1], 2, d_shape[2]))
for i in range(d_shape[1]-1):
    segs[:,i,0] = D[:,i]
```

```
    segs[:,i,1] = D[:,i+1]
segs[:,d_shape[1]-1,0] = D[:,d_shape[1]-1]
segs[:,d_shape[1]-1,1] = D[:,0]
# Se crea el artist, se guarda y se añade a los ejes
print(segs[0].shape)
print(segs.dtype)
lines = LineCollection(segs[0], linewidths=(4, 4, 4, 4),colors=colors, linestyle='solid')
ax.add_collection(lines)

q_min = np.min(D[:,:,0])
q_max = np.max(D[:,:,0])
p_min = np.min(D[:,:,1])
p_max = np.max(D[:,:,1])
ax.set_xlim(q_min - 0.2, q_max + 0.2)
ax.set_ylim(p_min - 0.2, p_max + 0.2)
ax.set_aspect('equal', adjustable='box')
fig.set_dpi(DPI)
fig.set_size_inches(X_SIZE, Y_SIZE)
title = ax.text(0.5,0.05, "", bbox={'facecolor':'w', 'alpha':0.5, 'pad':5},\
            transform=ax.transAxes, ha="center")
plt.show()
assert(False)
# Hay que devolver los artist que hayan cambiado para que se sepa que redibujar
# al pasarle blit=True al animador. Esto es una mera mejora de rendimiento
def init():
    title.set_text("t = 0.00")
    lines.set_verts(segs[0])

    return lines, title,

def animate(step):
    title.set_text("t = " + "{:.2f}".format(step * anim_delta))
    lines.set_verts(segs[step])
    return lines, title,

ani = animation.FuncAnimation(fig, animate, init_func=init, interval=1./FPS, \
    frames=FPS*duration, blit=True, repeat=False)

save_gif(f'animation_a={a}_b={b}_fps={FPS}.gif', ani, FPS)
save_video(f'animation_a={a}_b={b}_fps={FPS}.mp4', ani, FPS)
```