


Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio

Máster Universitario en Sistemas Espaciales

Milestone 5 and 6

Ampliación de Matemáticas 1

11 de diciembre de 2022

Autores:

- Andrés García Serrano

Índice

1. Introducción

En este hito se calcularán las regiones de estabilidad absoluta de distintos esquemas temporales.

1.1. Uso del módulo

Al igual que en los Milestones anteriores, el módulo se puede acceder ejecutando el archivo menu.py situado en la carpeta raíz del proyecto.

El Milestone 5 puede ser configurado con las siguientes variables:

- Numero de cuerpos
- Numero de pasos temporales
- Intervalo entre pasos
- Factor de escalado espacial
- Un boolean que controla si se representan las posiciones absolutas o relativas al centro de masas.

2. Milestone 5

2.1. Problema de los N cuerpos

A continuación se presenta el código necesario para la resolución de un paso del problema de los N cuerpos. Para evitar las copias excesivas se utilizan los métodos internos de numpy para realizar los cálculos 'in-place' dentro de los 'buffers' de las matrices.

```
def NBodyForce(U: ArrayLike, t: ArrayLike):
    # Get the size of the physical world.
    Nb = int( len(U)     )
    Nc = int( len(U[0]) )
    Nd = int( Nc / 2 )

    # Get position and velocity pointers.
    Us = reshape(U, (Nb, Nc))
    r = reshape(Us[:, :Nd], (Nb, Nd))
    v = reshape(Us[:, Nd:], (Nb, Nd))

    # Get derivative pointers.
    F = zeros(size(U))
    Fs = reshape(F, (Nb, Nc))
```

```
    dr = reshape(Fs[:, :Nd], (Nb, Nd))
    dv = reshape(Fs[:, Nd:], (Nb, Nd))

    # Calculate derivatives.
    #dr[:, :] = v[:, :]
    dv[:, :] = 0

    # Start iteration.
    for i in range(Nb):
        #dv[:, :] = 0
        dr[i, :] = v[i, :]

        for j in range(Nb):
            if i == j:
                continue

            D = r[j, :] - r[i, :]
            dv[i, :] = dv[i, :] + D / norm( D ** 3.0 )

    return reshape(F, (Nb, Nc))
```

Para calcular todos los pasos, se genera con la configuración dada una lista de cuerpos para todos los pasos, con posiciones y velocidades iniciales aleatorias, multiplicadas por el factor de escalado espacial.

```
    # Create the array and populate the initial step.
    bodies = empty((steps, num, 6))
    bodies[0] = array( [[(random() - 0.5) * scale for _ in range(6)]\
        for _ in range(num)] )

    # Iterate over time.
    for s in range(1, steps):
        bodies[step] = RungeKutta(bodies[s-1], dt, 0.0, NBodyForce)
```

Una vez se han calculado todos los pasos, se recopilan todas las variables espaciales (X, Y, Z) y la velocidad absoluta (V) de los cuerpos.

```
    # Get the X position.
    X = [ [body[0] for body in step] for step in bodies ]
    Y = [ [body[1] for body in step] for step in bodies ]
    Z = [ [body[2] for body in step] for step in bodies ]
    V = [ [norm(body[3:]) for body in step] for step in bodies ]
```

En caso de haber elegido representar las posiciones relativas al centro de masas, todas las posiciones se normalizan respecto al centro de masas calculado en cada paso temporal, pasando a unos ejes inerciales ligados a dicho centro de masas.

```
    # Renormalize for centre of mass.
    if com:
```

```
    for step in range(steps):
        centre = [0.0, 0.0, 0.0]

        for body in range(num):
            centre[0] += X[step][body]
            centre[1] += Y[step][body]
            centre[2] += Z[step][body]

        centre[0] = centre[0] / num
        centre[1] = centre[1] / num
        centre[2] = centre[2] / num

        for body in range(num):
            X[step][body] = X[step][body] - centre[0]
            Y[step][body] = Y[step][body] - centre[1]
            Z[step][body] = Z[step][body] - centre[2]
```

Con todos estos pasos, se crea la animación de las órbitas.

```
# Get the trajectories.
travel = [[] for _ in range(num)]

for step in bodies:
    for (j, data) in enumerate( step ):
        travel[j].append( [data[0], data[1], data[2]] )

travel = array( travel )

# Maximum velocity, X and Y value.
Vmax = max( max(V[:]) )
Xabs = [ [abs(x) for x in step] for step in X ]
Yabs = [ [abs(y) for y in step] for step in Y ]
Zabs = [ [abs(z) for z in step] for step in Z ]

Xmax = max( [max(x) for x in Xabs] )
Ymax = max( [max(y) for y in Yabs] )
Zmax = max( [max(z) for z in Zabs] )

# Display animated problem.
fig = plt.figure()
ax  = fig.add_subplot(projection='3d')

# Initialize the lines.
lines = []

for body in travel[:]:
    l = ax.plot([], [], [])[0]
    l.set_data(body[:0, 0], body[:0, 1])
```

```
        l.set_3d_properties([body[:0, 2]])
        lines.append(l)

    def plot(i):
        for (body, line) in zip( travel, lines ):
            line.set_data(body[:i,0], body[:i,1])
            line.set_3d_properties(body[:i,2])

        ax.set_zlim(-Zmax, Zmax)

    anim = FuncAnimation(fig=fig, func=plot,\
        frames=steps, interval=10, repeat_delay=3000)

    plt.show()
```

3. Milestone 6

3.1. Runge-Kutta embebido

Para el método de Runge-Kutta embebido se crea una clase llamada 'ButcherArray'. Esta clase se crea únicamente como una manera linguísticamente nativa para obtener colecciones de datos que permitirá polimorfismo funcional a la hora de ejecutar distintos esquemas.

En concreto se implementan los esquemas de Heun-Euler, Runge-Kutta 21, Bogacki-Shampine, Cash-Karp y Dormant Prince 54

```
class ButcherArray:
    def __init__(self, a, b, bs, c, q, Ne):
        self.a = a
        self.b = b
        self.bs = bs
        self.c = c
        self.q = q
        self.Ne = Ne

    def parameters(self):
        """Returns the parameters of this Butcher array."""
        return self.a, self.b, self.bs, self.c, self.q, self.Ne
```

Se añade una función para el cálculo dinámico del paso temporal.

```
def StepSize(dU, tol, dt, q):
    """Returns the correct step size."""

    # Error normalizado.
```

```
    n = norm(dU)

    if n > tol:
        return dt * (tol / n) ** (1 / (q + 1))
    else:
        return dt
```

Y también se añade una función Runge-Kutta adpatada a los esquemas Butcher.

```
def RK(U, dt, t, F, method: ButcherArray, first: bool):
    """Performs the given Embedded RK method."""
    # Get the butcher array data.
    a, b, bs, c, q, Ne = method.parameters()

    # Initialize k.
    k = zeros([Ne, len(U)])
    k[0,:] = F(U, t + c[0]*dt)

    for i in range(1, Ne):
        Up = U

        for j in range(i):
            Up = Up + dt * a[i, j] * k[j, :]

        k[i,:] = F(Up, t + c[i] * dt)

    if first:
        return U + dt * matmul(b, k)
    else:
        return U + dt * matmul(bs, k)
```

Por último se crea una función para el cálculo global del Runge-Kutta embebido.

```
def EmbeddedRK(U, dt, t, F, method=default, tol=1e-10):
    # Get the first iteration.
    V1 = RK(U, dt, t, F, method, True )
    V2 = RK(U, dt, t, F, method, False)

    # Get the Butcher array parameters.
    a, b, bs, c, q, Ne = method.parameters()

    # Calculate h.
    h = min(dt, StepSize( V1-V2, tol, dt, min(q) ) )

    # Get the number of intermediate steps.
    Ni = int(dt / h) + 1
    Ndt = dt / Ni
```

```
    # Reset V1, V2
    V1 = U
    V2 = U

    for i in range(Ni):
        time = t + (i * dt / int(Ni))
        V1 = V2
        V2 = RK(V1, time, Ndt, F, method, True)

    return V2
```

3.2. Problema de los 3 cuerpos y puntos de Lagrange

Para la resolución del problema de los 3 cuerpos se crea la función 'CR3BP' en la cual se calcula el problema de los 3 cuerpos en base al parámetro gravitacional de los dos cuerpos principales.

```
def CR3BP(U, t, mu=3.0039e-7):
    # Get the dimensionality.
    l = int( len(U) / 2 )

    # Get the position and velocity.
    r = U[:l]
    d = U[l:]

    # Calculate the velocities.
    v = [
        sqrt( ((r[0]     + mu) ** 2) + (r[1]**2) ),
        sqrt( ((r[0] - 1 + mu) ** 2) + (r[1]**2) ),
    ]

    # Calculate the forces.
    f = [
        (-(1-mu) * (r[0] + mu) / (v[0]**3)) -\
        (mu * (r[0] - 1 + mu) / (v[1]**3)),
        (-(1-mu) * (r[1]     ) / (v[0]**3)) -\
        (mu * (r[1]          ) / (v[1]**3)),
    ]

    return array( [
        d[0],
        d[1],

        ( 2 * d[1]) + r[0] + f[0],
        (-2 * d[0]) + r[1] + f[1],
    ] )
```

Para el cálculo de los puntos de Lagrange se utiliza el problema de los 3 cuerpos para calcular los 5 puntos de Lagrange. La estabilidad de estos puntos se extrae del Jacobiano de su matriz (aprovechando la función del Jacobiano creada en Milestones previos).

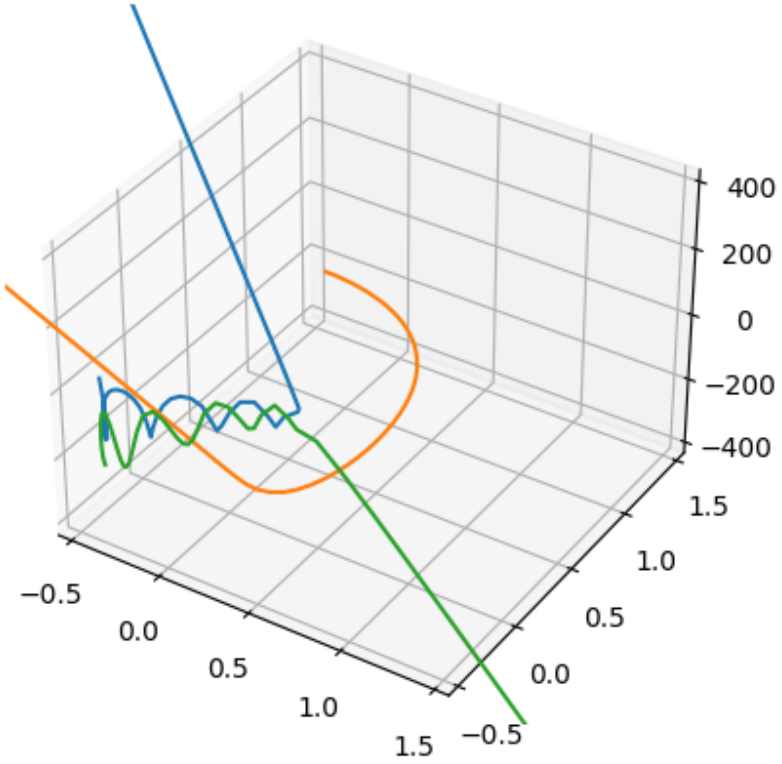
```
def LagrangePoints(U, NL, mu=3.0039e-7):
    # Storage for lagrange points.
    LP = zeros([5,2])

    def F(Y):
        X = zeros(4)
        X[:2] = Y
        X[2:] = 0

        return CR3BP(X, 0, mu)[2:4]

    for i in range(NL):
        LP[i,:] = Newton(F, U[i,:2])

    return LP


def LagrangePointStability(U, mu=3.0039e-7):
    def F(Y):
        return CR3BP(Y, 0, mu)

    A = Jacobiano(F, U)

    values, vectors = eig(A)

    return values
```

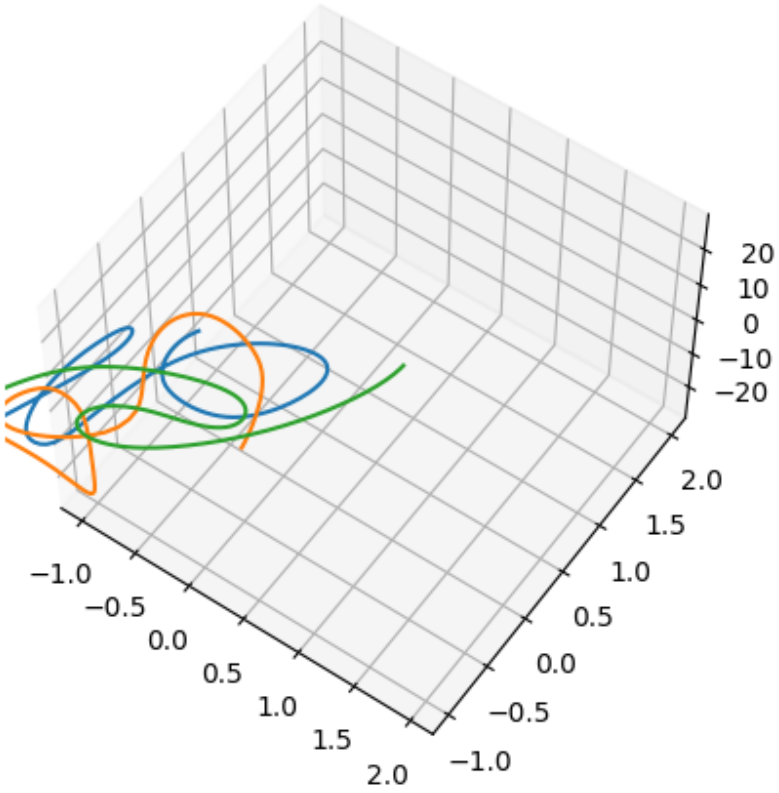
4. Resultados

4.1. Milestone 5

Debajo se muestran dos ejemplos del problema de los N cuerpos para el caso de 3 cuerpos libres. En el primer ejemplo se pueden observar dos cuerpos que han sido generados muy cerca uno del otro. Tras varias oscilaciones rápidas entre ellos, se separan a alta velocidad tras casi chocar.

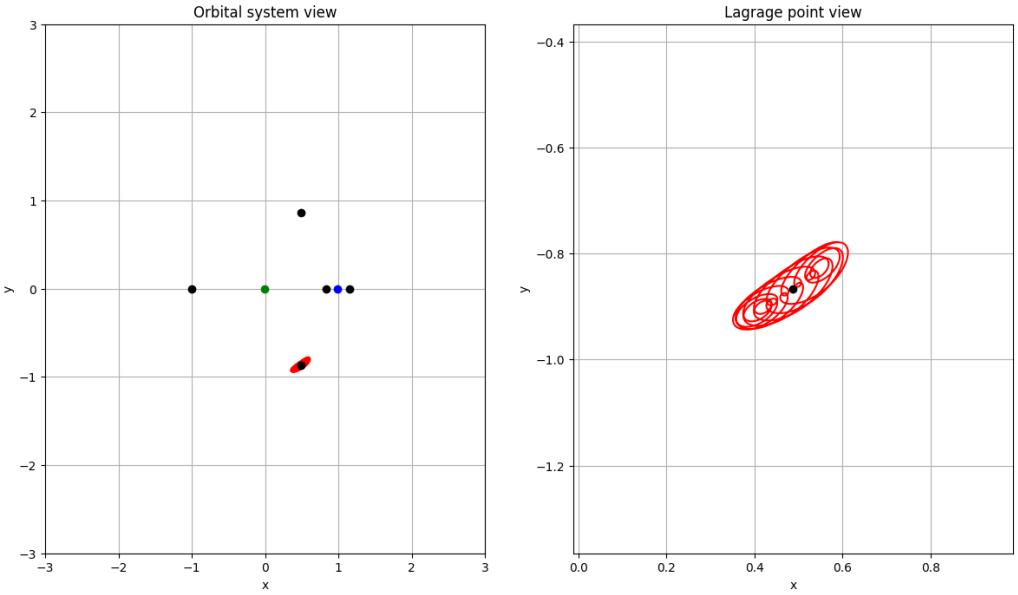
En el segundo ejemplo se demuestra como el centro de masa se desplaza con el tiempo, al no ser la velocidad inicial nula. Se puede apreciar como los objetos enseñados orbitan alrededor de dicho centro de masas, desplazándose con él hasta salir del espacio observado.


400
200
0
−200
−400
−0.5
0.0
0.5
1.0
1.5
1.5
1.0
0.5
0.0
−0.5

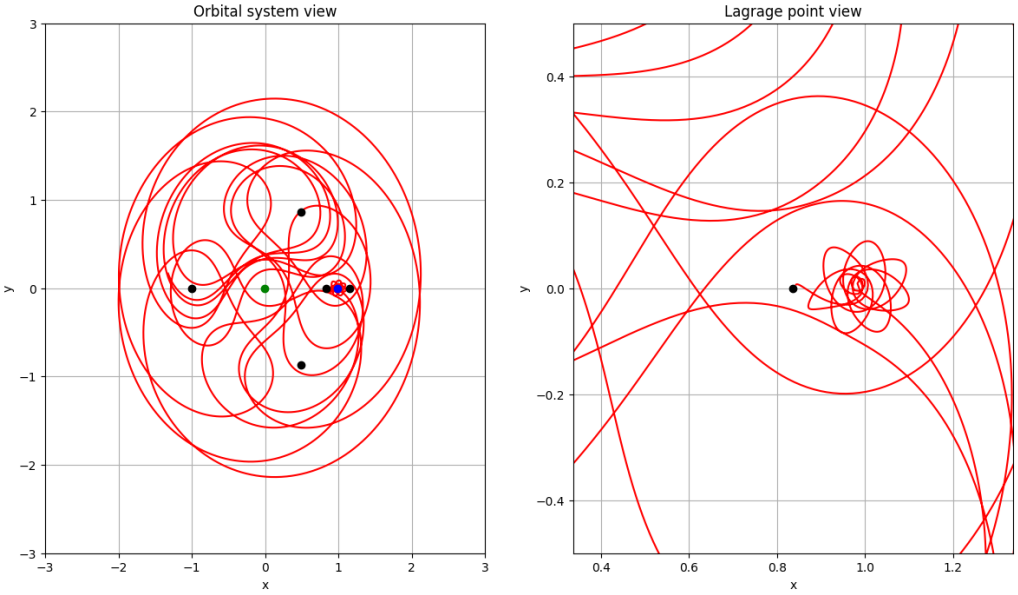


4.2. Milestone 6

En la siguiente figura se puede observar un cálculo de estabilidad sobre el punto de Lagrange L4. En este gráfico se aprecia que, aunque el objeto no se mantiene sobre el punto L4, orbita alrededor de él, demostrando la estabilidad de éste.



A continuación se muestra un ejemplo de la capacidad de cambio de órbita entre los puntos de Lagrange con cambios de estado mínimos. En la figura de la izquierda se aprecian oscilaciones iniciales alrededor de L2, hasta que se escapa de él y se empieza a mover el objeto en órbitas mayores, moviéndose entre los disintos puntos de manera oscilatoria.



5. Conclusiones

Considero que este Milestone es el punto donde el paradigma funcional empieza a brillar. La posibilidad de realizar polimorfismo con las distintas configuraciones para los problemas reduce la complejidad del código para los desarrolladores de alto nivel.

Sin embargo, para el bajo nivel, el uso de Python como lenguaje de programación resulta, en mi opinión, una gran desventaja a la hora de depurar código, desarrollar código robusto y velocidad de computación.

Los problemas escondidos dentro de la facilidad de Python son a su vez una ventaja y un inconveniente. Aunque su polimorfismo y la falta de sistema tipado fuerte ayuden al desarrollo rápido, a la hora de generar código funcional resultan un peligro, ya que las comprobaciones de tipo entre funciones no son realizadas 'ahead of time' (AOT), sino que son realizadas al momento de realizar la operación.

Considero que, para entornos no educativos, la manera más robusta de realizar estos Milestones sería con 'wrappers' profesionales a librerías de cálculo y físicas, y utilizando Python simplemente cómo el último nivel de desarrollo, prácticamente como el 'end-user'.

No todo son cosas malas. Python brilla mucho en, sorprendentemente, sus partes más funcionales. El mapeado funcional, los generadores, el polimorfismo y la composisción, tanto de clases como de funciones, produce un código de alto nivel muy fácil de interpretar y muy flexible. Además, si se trata a las clases no como objetos, sino como colleciones de datos con tipo asociado, se puede generar un paradigma cuasi-funcional a la hora de escribir código, aunque, al final del día, cuando se interactúa con código externo, suele acabando ser necesario el uso de objectos.