

## Métodos Numéricos

# Trabajo práctico N°4

### Ecuaciones diferenciales ordinarias

Grupo 11:    Martin Amagliani  
                 Olivia De Vincenti  
                 Nicolás Fernandez Pelayo

Profesor:    Pablo Ignacio Fierens

Fecha de entrega: 15 de octubre de 2020

Buenos Aires, Argentina

# Consigna

## Grupo impar

1. Escriba una función que implemente el método de Heun para resolver el sistema de ecuaciones diferenciales:

$$\begin{aligned}\dot{\vec{x}} &= f(t, \vec{x}) & t > t_0, \\ \vec{x}(t_0) &= \vec{x}_0\end{aligned}\tag{1}$$

La función debe tener el nombre **kheun** y recibir el handle a la función  $f$ , el tiempo inicial  $t_0$ , el tiempo final, el paso de integración y la condición inicial  $\vec{x}_0$  (como arreglo de **Numpy**). La función debe devolver un arreglo con los instantes de tiempo y otro con las aproximaciones numéricas a  $\vec{x}$  (una fila por cada instante de tiempo).

2. Utilice la función **kheun** para resolver el sistema de ecuaciones:

$$\begin{aligned}\dot{s} &= v_0 - 0,23 \cdot s \cdot p^2 & t > t_0, \\ \dot{p} &= 0,23 \cdot s \cdot p^2 - 0,4 \cdot p & t > t_0.\end{aligned}\tag{2}$$

Esta ecuación diferencial se corresponde con el modelo de glucólisis de Higgins-Selkov. Este modelo describe la glucólisis gobernada por la fosfofrutoquinasa (PFK). Primero, la glucosa se convierte en fructosa-6-fosfato (F6P). Luego la F6P es convertida en fructosa-1,6-bisfosfato (FBP), con intermediación de la PFK. Luego, la FBP posibilita la formación de adenosina trifosfato (ATP), las moléculas que dan energía a las células. En la reacción en que interviene la PFK, una molécula de ATP es convertida en adenosina difosfato (ADP). En las ecuaciones diferenciales,  $s$  está relacionada con la cantidad *F6P* y  $p$  con la de *ADP*.

Los valores habituales de  $v_0$  están en el rango 0,48 - 0,60. Lo interesante es que, cuando  $v_0 < v_c$ ,  $s$  y  $p$  entran en un régimen oscilatorio permanente que puede ser reconocido simulando unos, digamos, 600 segundos ( $t \in [0, 600]$ ).

Presente gráficos que muestren este hecho. ¿Puede determinar el valor de  $v_c$  aproximadamente?

Todo esto debe estar en una función denominada **higgins-selkov**.

3. Escriba una función que pruebe el correcto funcionamiento de todas la demás funciones implementadas. La función debe llamarse **test**, sin argumentos.

Todas las funciones deben estar en un archivo denominado **energiza.py**.

Este problema fue tomado de: Alan Garfinkel, Jane Shevtsov, y Yina Guo.

*Modeling Life. The Mathematics of Biological Systems*. Springer. 2017.

# Funciones

<b>1. Ejercicio 1</b>	<b>1</b>
1.1. kheun . . . . .	1
<b>2. Ejercicio 2</b>	<b>2</b>
2.1. higginselkov . . . . .	2
2.2. f . . . . .	2
2.3. plotfun . . . . .	3
2.4. check_perm . . . . .	3
2.5. findvc . . . . .	4
<b>3. Ejercicio 3</b>	<b>5</b>
3.1. test . . . . .	5
3.2. testfla . . . . .	5
3.3. testflb . . . . .	5
3.4. solfl . . . . .	5

# Índice de Códigos

1. Definición de kheun . . . . .	1
2. Definición de higginselkov . . . . .	2
3. Definición de f . . . . .	2
4. Definición de plotfun . . . . .	3
5. Definición de check_perm . . . . .	3
6. Definición de findvc . . . . .	4
7. Definición de test . . . . .	5
8. Definición de testfla . . . . .	5
9. Definición de testflb . . . . .	5
10. Definición de solfl . . . . .	6

# 1. Ejercicio 1

## 1.1. kheun

Recibe la función que se va a calcular, el tiempo inicial y el final, los valores iniciales y el paso. Dados el paso y el intervalo de tiempo crea la lista  $t$  que tendrá todos los tiempos que se evaluarán. Luego en cada uno de estos tiempos se calcula la solución de la función utilizando el método de Heun para resolver sistemas de EDOs. Por último, la función devuelve la lista  $t$  con todos los tiempos evaluados y el arreglo  $x$  con las aproximaciones numéricas a  $\vec{x}$

Código 1: Definición de kheun

```
1 def kheun(f, x0, t0, tf, h):
2     N = int((tf - t0) / h)           # Número de puntos
3     t = np.linspace(t0, tf, N + 1)   # Creo vector con los valores de tiempo
4     n = x0.shape[0]                  # Dimensión del problema
5     x = np.zeros((n, N + 1))         # Creo vector nulo con las dimensiones que debe
6     #                               ↪ tener
7     x[:, 0] = x0                     # Le doy los valores iniciales
8
9     for k in range(N):
10        f1 = h * f(t[k], x[:, k])     # Calculo de la primer derivada
11        f2 = h * f(t[k] + h, x[:, k] + f1) # Calculo de la segunda derivada
12        x[:, k + 1] = x[:, k] + (f1 + f2) / 2.0 # Se hace promedio de las derivadas
13
14    return t, x                       # Devuelve la lista con los tiempos tomados y el
15    #                               ↪ array con los valores obtenidos
```

## 2. Ejercicio 2

### 2.1. higginselkov

El objetivo de esta función es resolver el sistema de EDOS (2) utilizando el método de heun, que es de la forma Se eligió  $t \in [0, 600]$ ,  $v_0$  entre 0.48 y 0.6 y condiciones iniciales  $s(0) = 2, p(0) = 3$  por recomendación de la cátedra. Se busca elegir un salto  $h$  tal que haya un error pequeño sin realizar una excesiva cantidad de operaciones. De esta forma llegamos a que  $h$  debería ser 0.01 dando como resultado un error global de  $7,5 \times 10^{-5}$ . Para este cálculo se usó algoritmo dado en clase:

$$\begin{aligned}
 (I) : E_k^{(1)}(\Delta t) &= X^{(1)}(t_k) - X_k^{(1)} \approx C((\Delta t)^2) \\
 (II) : E_{2k}^{(2)}\left(\frac{\Delta t}{2}\right) &= X^{(2)}(t_{2k}) - X_{2k}^{(2)} \approx C\left(\left(\frac{\Delta t}{2}\right)^2\right) \\
 \Rightarrow (I - II) : \cancel{X^{(1)}(t_k)} - X_k^{(1)} - \cancel{X^{(2)}(t_{2k})} + X_{2k}^{(2)} &\approx C\left((\Delta t)^2 - \left(\frac{\Delta t}{2}\right)^2\right) \\
 \Rightarrow X_{2k}^{(2)} - X_k^{(1)} &\approx C\left((\Delta t)^2 \cdot \frac{3}{4}\right)
 \end{aligned} \tag{3}$$

Nota: decidimos no chequear que  $t_0$  sea mayor a cero (ya que (2) sólo es válida en el intervalos  $(0, \infty)$ ) porque se define dentro de nuestra función y no tendría sentido definir en el código algo inválido.

Código 2: Definición de higginselkov

```

1 def higginselkov():
2     t0 = 0
3     tf = 600
4     h = 0.01
5     x0 = np.array([2, 3])
6
7     t, v = kheun(f, x0, t0, tf, h)      # Recibo la lista de tiempos utilizados y la matriz de datos
8                                         # obtenidos utilizando heun
9
10    plotfun(t, v[0, :], "s en función de t", "tiempo", "s")
11    plotfun(t, v[1, :], "p en función de t", "tiempo", "p")
12
13    return

```

### 2.2. f

Función a pasar **kheun**, recibe tiempo y arreglo cuyos elementos son  $s$  y  $p$  respectivamente. En nuestro caso, **f** devuelve un arreglo de **numpy** donde el primer elemento la primer función de (2) y el segundo es la otra. Cabe destacar que mientras **f** no utiliza el tiempo, es necesario que esté definido con él de todas formas para que sea aceptada por **kheun**.

Código 3: Definición de f

```

1 def f(t, x):                                # Definimos la función que vamos a evaluar con kheun
2     return np.array([v0 - 0.23 * x[0] * (x[1] ** 2), 0.23 * x[0] * (x[1] ** 2) - 0.4 * x[1]])

```

## 2.3. plotfun

Grafica una función dada, decidimos implementarla para facilitar la creación de gráficos simples. Recibe los valores de las abscisas y las ordenadas junto con el título y los labels que personalizan el gráfico.

Código 4: Definición de plotfun

```

1 def plotfun(x, y, title = "y en función de x", xlabel = "x", ylabel = "y"):
2     plt.title(title)
3     plt.xlabel(xlabel)
4     plt.ylabel(ylabel)
5     plt.plot(x, y)
6     plt.show()
7     return

```

## 2.4. check\_perm

Función utilizada para acompañar el cálculo de  $v_c$ , compara el mínimo de una función en dos intervalos distintos (tercer cuarto y último cuarto) y los compara entre sí, si hay mucha diferencia entre ellos no se trata de un régimen estacionario permanente. Luego se realiza lo mismo para sus máximos. Ambas  $s(t)$  y  $p(t)$  tienen que cumplir esto.

Código 5: Definición de check\_perm

```

1 def check_perm(v, t0, tf, h):
2     i = 0
3     for i in range(2):
4         # Obtengo mínimos en dos partes diferentes de la función y los comparo
5         mthird = min(v[i][int(((tf - t0) / (h * 2))): int(((tf - t0) * (3 / 4)) / h)]) # Tercer cuarto
6         mfourth = min(v[i][int((tf - t0) / (h * (4 / 3))): int((tf - t0) / h)]) # Último cuarto
7         dif = abs(mthird - mfourth)
8         if dif > 0.01:
9             print("False")
10            return False
11        # Repito con máximos
12        Mthird = max(v[i][int((tf - t0) / (h * 2)): int((tf - t0) / (h * (4 / 3)))]])
13        Mfourth = max(v[i][int((tf - t0) / (h * (4 / 3))): int((tf - t0) / h)])
14        dif = abs(Mthird - Mfourth)
15        if dif > 0.01:
16            print("false")
17            return False
18    print("true")
19    return True

```

## 2.5. findvc

Realiza `higginselkov` para distintos valores de  $v_0$  hasta encontrar aproximadamente  $v_c$ . Se comienza con el  $v_0$  más pequeño, en caso de tener un régimen estacionario se incrementará en un cantidad, en caso de no tener un régimen estacionario se decrementa la cantidad en la que varía, y también  $v_0$ .

Nota: consideramos leves disminuciones que al tender a infinito puedan llevar a que la función no esté en régimen estacionario permanente.

Como conclusión, decimos que  $v_c$  es aproximadamente 0.5214068603515625

Código 6: Definición de `findvc`

```

1 def findvc():
2     x0 = np.array([2, 3])
3     global v0
4     v0 = 0.48                # Defino variables iniciales y el cambio que hara v0 en cada iteración
5     vc = v0
6     cambio = 0.02
7     for i in range(20):
8         t, v = kheun(f, x0, 0, 600, 0.01)        # Resuelvo kheun para el vo actual
9         if check_perm(v, 0, 600, 0.01):          # Compruebo si se encuentra en ROP
10             print("Sigue siendo oscilación permanente para v0 =", v0, "\n")
11             v0 = v0 + cambio                      # En caso positivo aumento v0
12             if v0 == vc:                          # Dividimos el cambio en 2 para que no se vuelva
13                 cambio = cambio / 2              # al ultimo valor evaluado 1 paso atras
14                 v0 = v0 - cambio
15             else:                                # En caso negativo decremento v0
16                 print("No fue oscilación permanente buscamos un valor mas pequeño. Usando v0 =", v0, "\n
↪ ")
17                 vc = v0
18                 cambio = cambio / 2              # También decremento la magnitud del cambio
19                 v0 = v0 - cambio
20
21     #Gráfico de s y p
22     plotfun(t, v[0, :], "s en función de t para el valor de v0 actual", "tiempo", "s")
23     plotfun(t, v[1, :], "p en función de t para el valor de v0 actual", "tiempo", "p")
24
25     return vc

```

### 3. Ejercicio 3

#### 3.1. test

Función de Prueba. Primero prueba **kheun**:

- **testfla** resuelve:

$$\dot{x}(t) = x(t) \quad (4)$$

con **solfl**.

- **testflb** resuelve el sistema:

$$\begin{aligned} \dot{x}(t) &= 2x(t) - 3y(t) \\ \dot{y}(t) &= -x(t) + 4y(t) \end{aligned} \quad (5)$$

con **scipy.integrate.solve\_ivp**

Luego prueba **higginselkov** comparando con la función **textttscipy.integrate.solve\_ivp** utilizando el método RK23 (De los métodos brindados por la función fue el mejor observado)

Código 7: Definición de test

```
1 def test():
2     return
3
```

← REGIÓN de MÁS

#### 3.2. testfla

Fórmula de la EDO (4), escrita de tal forma que **kheun** pueda recibirla.

SAN GRÍA

Código 8: Definición de testfla

```
1 def testfla(t, x):
2     return np.array([x[0]])
```

#### 3.3. testflb

Sistema de EDOs (5), escrita de tal forma que **kheun** pueda recibirla.

Código 9: Definición de testflb

```
1 def testflb(t, x):
2     return np.array([2*x[0]-3*x[1], -x[0] + 4*x[1]])
```

#### 3.4. solfl

Solución conocida de (4) que es:

$$x(t) = e^t \quad (6)$$



Código 10: Definición de solf1

```
1 def solf1(ti, tf, t, h):  
2     N = int((tf - ti) / h)  
3     x = np.zeros(N + 1)  
4     for i in range(N + 1):  
5         x[i] = np.e**t[i]  
6     return x
```