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 flucosa se convierte en fructosa-6-fosfato (F6P). Luego la F6P es convertida en frutosa-1,6-bifosfato (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 $v_0 < vc$, 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 `higginsselkov`.

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

Índice de Códigos

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 $\vec{x}$.

$$x_{k+1} = x_k + \frac{f(t_k, x_k) + f(t_k + \Delta t, x_k + f(t_k, x_k)\Delta t)}{2}\Delta t \tag{3}$$

Código 1: Definición de kheun

```python
def kheun(f, x0, t0, tf, h):
    N = int((tf - t0) / h)                          # Número de puntos
    t = np.linspace(t0, tf, N + 1)                  # Creo vector con los valores de tiempo
    n = x0.shape[0]                                 # Dimensión del problema
    x = np.zeros((n, N + 1))                        # Creo vector nulo con las dimensiones que debe
    ↪ tener
    x[:, 0] = x0                                    # Le doy los valores inciales

    for k in range(N):
        f1 = h * f(t[k], x[:, k])                   # Calculo de la primer derivada
        f2 = h * f(t[k] + h, x[:, k] + f1)          # Calculo de la segunda derivada
        x[:, k + 1] = x[:, k] + (f1 + f2) / 2.0     # Se hace promedio de las derivadas

    return t, x                                     # Devuelve la lista con los tiempos tomados y el
    ↪ array con los valores obtenidos
```

2. Ejercicio 2

2.1. higginsselkov

El objetivo de esta función es resolver el sistema de EDOS (2) utilizando el método de heun, que es de la forma (3). 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 busco elegir un paso 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 $7,5x10^-5$. Para este cálculo se usó algoritmo dado en clase:

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

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 higginsselkov

```
def higginsselkov():
    t0 = 0
    tf = 600
    h = 0.01
    x0 = np.array([2, 3])

    t, v = kheun(f, x0, t0, tf, h)          # Recibo la lista de tiempos utilizados y la matriz de datos
                                            # obtenidos utilizando heun

    plotfun(t, v[0, :], "s en función de t", "tiempo", "s")
    plotfun(t, v[1, :], "p en función de t", "tiempo", "p")

    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

```python
def f(t, x):                                    # Definimos la función que vamos a evaluar con kheun
    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

```python
def plotfun(x, y, title = "y en función de x", xlabel = "x", ylabel = "y"):
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.plot(x, y)
    plt.show()
    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

```python
def check_perm(v, t0, tf, h):
    i = 0
    for i in range(2):
        # Obtengo mínimos en dos partes diferentes de la función y los comparo
        mthird = min(v[i][int(((tf - t0) / (h * 2))): int((((tf - t0) * (3 / 4)) / h))])  # Tercer cuarto
        mfourth = min(v[i][int((tf - t0) / (h * (4 / 3))): int((tf - t0) / h)])           # Último cuarto
        dif = abs(mthird - mfourth)
        if dif > 0.01:
            print("False")
            return False
        # Repito con máximos
        Mthird = max(v[i][int((tf - t0) / (h * 2)): int((tf - t0) / (h * (4 / 3)))])
        Mfourth = max(v[i][int((tf - t0) / (h * (4 / 3))): int((tf - t0) / h)])
        dif = abs(Mthird - Mfourth)
        if dif > 0.01:
            print("false")
            return False
    print("true")
    return True
```

2.5. findvc

Realiza `higginsselkov` 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 vc es aproximadamente 0.5214068603515625

Código 6: Definición de findvc

```python
def findvc():
    x0 = np.array([2, 3])
    global v0
    v0 = 0.48                    # Defino variables iniciales y el cambio que hara v0 en cada iteración
    vc = v0
    cambio = 0.02
    for i in range(20):
        t, v = kheun(f, x0, 0, 600, 0.01)            # Resuelvo kheun para el vo actual
        if check_perm(v, 0, 600, 0.01):              # Compruebo si se encuentra en ROP
            print("Sigue siendo oscilación permanente para v0 =", v0, "\n")
            v0 = v0 + cambio                         # En caso positivo aumento v0
            if v0 == vc:                             # Dividimos el cambio en 2 para que no se vuelva
                cambio = cambio / 2                  # al ultimo valor evaluado 1 paso atras
                v0 = v0 - cambio
        else:                                        # En caso negativo decremento v0
            print("No fue oscilación permanente buscamos un valor mas pequeño. Usando v0 =", v0, "\n")
            vc = v0
            cambio = cambio / 2                      # También decremento la magnitud del cambio
            v0 = v0 - cambio

      #Gráfico de s y p
        plotfun(t, v[0, :], "s en función de t para el valor de v0 actual", "tiempo", "s")
        plotfun(t, v[1, :], "p en función de t para el valor de v0 actual", "tiempo", "p")

    return vc
```

3. Ejercicio 3

3.1. test

Función de Prueba. Primero prueba `kheun`:

- `testf1a` resuelve:

$$\dot{x}(t) = x(t), \qquad y(0) = 1 \tag{5}$$

  con `solf1`.

- `testf1b` resuelve el sistema:

$$\begin{aligned} \dot{x}(t) &= 2x(t) - 3y(t), & x(0) = 1 \\ \dot{y}(t) &= -x(t) + 4y(t), & y(0) = 2 \end{aligned} \tag{6}$$

  con `scipy.integrate.solve_ivp`

Luego prueba `higginsselkov` comparando con la función `scipy.integrate.solve_ivp` utilizando el método RK23 (De los métodos brindados por la función fue el mejor observado).

Código 7: Definición de test

```
def test():
# test 1a: edo simple y'(t) = y(t)
    t, v = kheun(testf1a, np.array([1]), 0, 9, 0.01)                # Kheun
    plotfun(t,v[0,:],"y en funcion de t usando kheun","Tiempo", "y(t)")

    vsol = solf1(t)                                                 # Solución exacta y(t)=e^t
    plotfun(t,vsol,"Solucion de y'=y", "tiempo","y(t)")

    err = np.zeros(len(t))                                          # Error
    for j in range(len(t)):
        err[j] = abs(v[0][j] - vsol[j])
    plt.title("error")
    plt.xlabel("tiempo")
    plt.ylabel("y")
    plt.plot(t, err)
    plt.show()

# test 1b: Sistema de EDOs {x'=2x - 3y, y'= -x + 4y}
    t, v = kheun(testf1b, np.array([1,2]), 0, 10, 0.01)             # Kheun
    plotfun(t,v[0,:],"x(t) usando kheun", "tiempo","x(t)")
    plotfun(t,v[1,:],"y(t) usando kheun", "tiempo","y(t)")

    sol = sci.solve_ivp(testf1b, (0, 10), [1, 2], "RK23")           # Resolución de scipy
    plotfun(sol.t, sol.y[0, :], "x(t) usando scipy", "tiempo", "x(t)")
    plotfun(sol.t, sol.y[1, :], "y(t) usando scipy", "tiempo", "y(t)")

# test 2: higginsselkov vs scipy
    t0 = 0.0
    tf = 600.0
```

```
    higginsselkov()                                    # Higginsselkov
    sol = sci.solve_ivp(f, (t0, tf), [2, 3], "RK23")       # resolucion de scipy
    plotfun(sol.t, sol.y[0, :], "s usando scipy", "tiempo", "s")
    plotfun(sol.t, sol.y[1, :], "p usando scipy", "tiempo", "p")

    print("fin de prueba")
    return
```

3.2. testf1a

Fórmula de la EDO (5), escrita de tal forma que `kheun` pueda recibirla.

Código 8: Definición de testf1a

```
def testf1a(t, x):
    return np.array([x[0]])
```

3.3. testf1b

Sistema de EDOs (6), escrita de tal forma que `kheun` pueda recibirla.

Código 9: Definición de testf1b

```
def testf1b(t, x):
    return np.array([2*x[0]-3*x[1], -x[0] + 4*x[1]])
```

3.4. solf1

Solución conocida de (5) que es:

$$x(t) = e^t \tag{7}$$

Código 10: Definición de solf1

```
def solf1(t):
    x = np.zeros(len(t))
    for i in range(len(t)):
        x[i] = np.e**t[i]
    return x
```