Tarea 3, Métodos Numéricos para la Ciencia y la Ingeniería

Jaime Castillo Lara

P1. Buscamos solucionar el problema del oscilador de van der Pool. La EDO que modela el oscilador está dada por:

$$\frac{d^2x}{dt^2} = -kx - \mu(x^2 - a^2)\frac{dx}{dt}$$

Sin embargo, al hacer un cambio de variable, tenemos la EDO:

$$\frac{d^2y}{ds^2} = -y - \mu^*(y^2 - 1)\frac{dy}{ds}$$

Se pide utilizar el método Runge-Kutta de orden 3. Este método, para cada paso que se da, funciona de la forma:

$$x_{i+1} = x_i$$

$$k_1 = hf(x_i, y_i)$$

$$k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{1}{2}k_1\right)$$

$$k_3 = hf(x_i, y_i - k_1 - 2k_2)$$

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)$$

Donde h representa el avance en x para cada paso del método, y $k_1, k_2, k_3$ constantes definidas para cada paso.

Para implementar este método en la EDO simplificada para el problema del oscilador, nos damos $\mu^* = 1.243$ (A partir de los últimos dígitos del RUT:18.138.243-0). Para esto, escribimos:

u_= 1.243

Una vez determinado esto, definimos la función func() la cual recibe valores para “y” y para dy/ds

Y retorna la ecuación de la EDO simplificada. Escribimos:

```
def func(y,v):
    return v, -y -u_*(y*y-1)*v
```

Ahora, definimos funciones para asignar, para cada iteración, los valores para $k_1, k_2, k_3,$

Para $k_1$, necesitamos que $k_1 = hf(x_i, y_i)$. Para esto escribimos:

```
def getk1(y_n,v_n,h,func):
    f_eval= func(y_n,v_n)
    return h*f_eval[0], h*f_eval[1]
    print "k1"
    print getk1[0],getk1[1]
```

Utilizando la función ya definida func() para evaluar $x_i, y_i$ en la ecuación (en nuestro caso “y” y dy/ds) y getk1() nos retornará los valores de $k_1$ para la iteración correspondiente.

Para $k_2$, necesitamos que $k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{1}{2}k_1\right)$. Para esto escribimos:

```
def getk2(y_n,v_n,h,func):
    k1= getk1(y_n,v_n,h,func)
    f_eval= func(y_n + k1[0]/2, v_n + k1[1]/2)
    return h*f_eval[0], h*f_eval[1]
    print "k2"
    print gwtk2[0],getk2[1]
```

Donde nuevamente utilizamos la función func() para evaluar en la EDO, pero esta vez utilizando los parámetros h y $k_1$, este último utilizando getk1(). Con esto, getk2() nos entregará el valor para $k_2$ para la iteración correspondiente.

Para $k_3$, necesitamos que $k_3 = hf(x_i, y_i - k_1 - 2k_2)$. Para esto escribimos:

```
def getk3(y_n,v_n,h,func):
    k1=getk1(y_n,v_n,h,func)
    k2=getk2(y_n,v_n,h,func)
    f_eval=func(y_n-k1[0]-2*k2[0],v_n-k1[0]-2*k2[1])
    return h*f_eval[0], h*f_eval[1]
    print "k3"
    print getk3[0],getk3[1]
```

Con esto, utilizando func(), getk1() y getk2(), la función getk3() nos entregará el valor correspondiente para $k_3$ en la iteración correspondiente.

Finalmente, necesitamos la definición de "y" para el paso siguiente del método Runge-Kutta (en nuestro caso para la variable "v"), la cual está dada por $y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)$.

Para esto, definimos la función:

```
def paso_rk3(y_n,v_n,h,func):
    k1=getk1(y_n,v_n,h,func)
    k2=getk2(y_n,v_n,h,func)
    k3=getk3(y_n,v_n,h,func)
    y_n1= y_n + (k1[0]+4*k2[0]+k3[0])/6
    v_n1= v_n + (k1[1]+4*k2[1]+k3[1])/6
    return y_n1,v_n1
    print "paso_rk3"
    print paso_rk3[0],paso_rk3[1]
```

Ya que tenemos todos los elementos necesarios para implementar el método Runge-Kutta de tercer orden, debemos integrar la EDO sobre el intervalo de tiempo pedido. Para esto definimos:

N_pasos= 1000 (Número de pasos que el método efectuará)

h= (20*np.pi)/N_pasos (h=el avance de la variable “y” para cada paso)

y=np.zeros(N_pasos)

v=np.zeros(N_pasos)

Inicializamos arreglos para ambas variables, con las dimensiones necesarias para que se vayan llenando con los valores que las variables irán tomando en cada paso.

Finalmente, hacemos la iteración para que vaya haciendo el método Runge-Kutta sobre la EDO, en el intervalo pedido, y con los pasos pedidos:

```
for i in range(1, N_pasos):
    y[i],v[i]= paso_rk3(y[i-1],v[i-1],h,func)
```

Para hacer avanzar el tiempo con las iteraciones, escribimos:

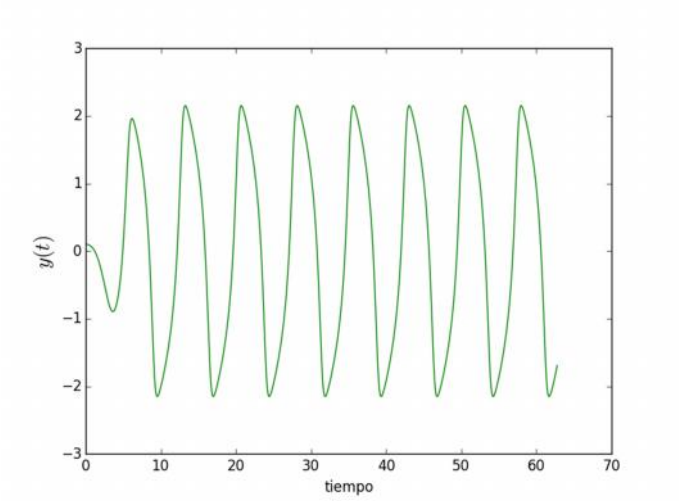
```
t_rk = [h * i for i in range(N_pasos)]
```

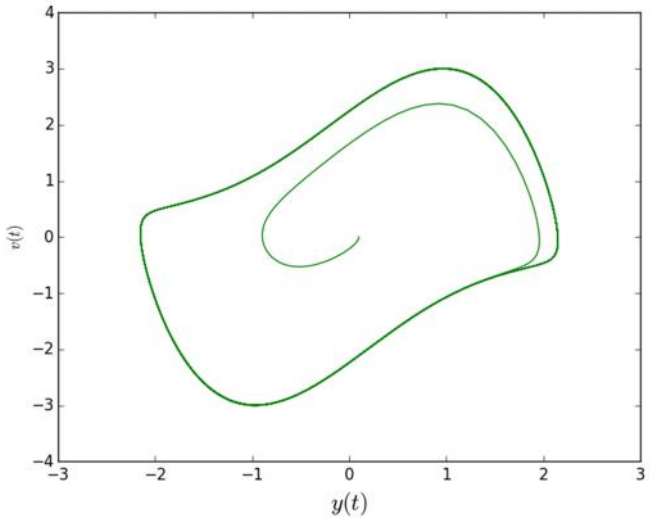
Para las condiciones iniciales $\frac{dy}{ds} = 0 \;\; y = 0.1$, inicializamos con

y[0]=0.1

v[0]=0

y obtenemos:



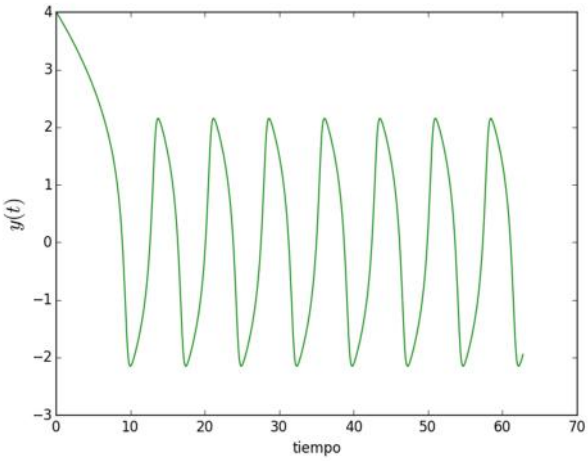



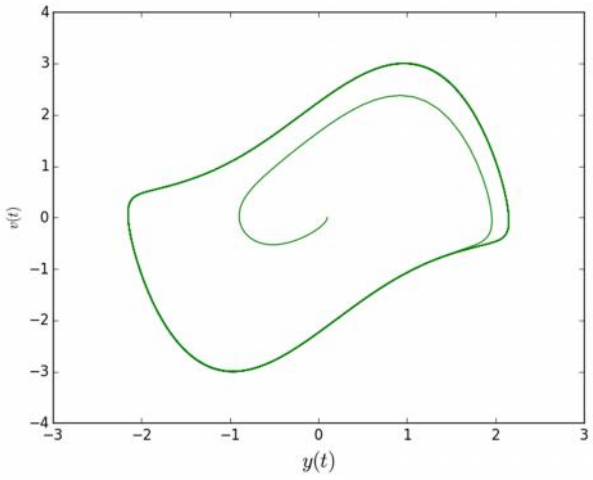
Para las condiciones iniciales $\frac{dy}{ds} = 0 \;\; y = 4$, inicializamos con

y[0]=04

v[0]=0

y obtenemos:




Notamos la similitud entre ambos pares de gráficos. Podemos atribuir esto a la poca diferencia entre las condiciones iniciales escogidas.

P2. Buscamos ahora soluciones el sistema de ecuaciones diferenciales ordinarias llamado sistema de Lorenz, el cual está dado por:

$$\frac{dx}{ds} = \sigma(y - x)$$
$$\frac{dy}{ds} = x(\rho - z) - y$$
$$\frac{dz}{ds} = xy - \beta z$$

Utilizando esta vez del método de Runge Kutta de orden 4. Para esto nos serviremos de la función ode(f)set.integrator('vode',method='adams') el cual corresponde al módulo scipy.integrate.

Para implementarlo, partimos definiendo una función para el sistema de Lorenz. Para esto escribimos:

```
def Lorenz(t,v,cons):

 sigma=cons[0]

 beta=cons[1]

 rho=cons[2]

 dv=np.zeros([3])

 dv[0]=sigma*(v[2]-v[1])

 dv[1]=v[0]*(rho-v[2])-v[1] #dy/ds

 dv[2]=v[0]*v[1]-beta*v[2] #dz/ds

 return dv
```

donde la función recibe valores para $\sigma$ (sigma), $\beta$ (beta), $\rho$ (rho), x (v[0]),y (v[1]), z (v[2]) y retorna valores para $\frac{dx}{ds}$ (dv[0]), $\frac{dy}{ds}$ (dv[1]) y $\frac{dz}{ds}$ (dv[2]) de acuerdo a los parámetros del sistema de Lorenz.

Para asignarle valores a las constantes, determinar condiciones iniciales, y asignar tiempo inicial, final, y delta de tiempo, escribimos:

```
cons=[10 , 2.666 , 28]

ti=0 #tiempo inicial

tf=200 #tiempo final

dt=0.001 #paso de tiempo

vi=[100,2000,400] #condiciones iniciales

s=[]

t=[]
```

Inicializamos además dos arreglos, sobre los que se irán guardando los valores del tiempo, y de las soluciones entregadas por el método de Runge-Kutta.

Asignamos al módulo las funciones y parámetros correspondientes:

```
solucion = ode(Lorenz).set_integrator('vode',method='adams')

solucion.set_f_params(cons).set_initial_value(vi,ti)
```

Iniciamos ahora la iteración en la que el método de Runge-Kutta de orden 4 se irá implementando en el intervalo de tiempo deseado.

```
while solucion.successful() and solucion.t+dt < tf:

  solucion.integrate(solucion.t+dt)

  s.append(solucion.y)

  t.append(solucion.t)
```

Donde se van guardando los valores en los arreglos que inicializamos para ello.

Finalmente, basándonos en el ejemplo entregado, ploteamos en 3 dimensiones:

```
fig = plt.figure(3)

fig.clf()

ax = fig.add_subplot(111, projection='3d')

ax.set_aspect('auto')

ax.set_title(r'$Grafico \ de \ las \ soluciones \ de \ las \ ecuasiones \ de \ Lorenz$')

ax.plot(s[:,0], s[:,1],s[:,2],'r-') #plot(x,y,z)

ax.set_xlabel(r'$Eje \ X$')

ax.set_ylabel(r'$Eje \ Y$')

ax.set_zlabel(r'$Eje \ Z$')

fig.savefig('3d')

plt.show()
```

Lo que nos entrega: