Ecuaciones diferenciales ordinarias 4

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

2 de mayo de 2013

Contenido

Contenido

Estabilidad del método RK4

Un análisis completo de estabilidad depende de la ecuación particular que se quiera tratar, pero la tendencia general de cada esquema de integración puede sondearse estudiando lo que ocurre en el caso de la sencilla ecuación $y' = \lambda y$

Revisaremos la estabilidad para esta ecuación usando el esquema RK4. Calcularemos los k_i y finalmente usaremos la expresión para y_{n+1} que nos proporciona RK4.

En las expresiones de los k_i se reemplaza f por λ multiplicando al segundo argumento de f que, genéricamente es y, obteniendo

$$k_1 = h\lambda y_n$$

$$k_2 = h\lambda\left(y_n + \frac{1}{2}k_1\right) = h\lambda\left(1 + \frac{h}{2}\lambda\right)y_n$$

$$k_3 = h\lambda\left(y_n + \frac{h}{2}\lambda\left(1 + \frac{h}{2}\lambda\right)y_n\right)$$

$$= h\lambda\left(1 + \frac{h}{2}\lambda\left(1 + \frac{h}{2}\lambda\right)\right)y_n$$

$$k_4 = h\lambda\left(1 + \lambda h\left(1 + \frac{h}{2}\lambda\left(1 + \frac{h}{2}\lambda\right)\right)\right)y_n$$

Al reemplazar estos valores en la expresión para y_{n+1} y escribiendo $y_{n+1} = \overline{y}_{n+1} + \epsilon_{n+1}$ y similarmente $y_n = \overline{y}_n + \epsilon_n$ en (donde los $\overline{y}$ son la solución exacta de la ecuación discreta, se obtiene que

$$\frac{\epsilon_{n+1}}{\epsilon_n} = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \frac{(h\lambda)^4}{24}$$

Para que haya estabilidad, éste cociente debe de tener un valor absoluto menos que 1 y puede comprobarse que para que se cumpla, se necesita que

$$-2.7853 < h\lambda < 0$$

Por ejemplo, si $\lambda = -1$ entonces la estabilidad está garantizada con

$$0 < h < 2.7853$$

que da un amplio margen para tener una ecuación absolutamente estable aun cuando, si h no es pequeño, la solución va a ser posiblemente poco confiable.

Sistema Lotka-Volterra

Las ecuaciones de Lotka-Volterra, también son conocidas como ecuaciones de depredador-presa, descrito por un sistema de 2 ecuaciones diferenciales no lineales de primer orden, que se utiliza frecuentemente para describir la dinámica de sistemas biológicos donde interaccionan dos especies, un depredador y una de sus presas.

El sistema evoluciona de acuerdo al par de ecuaciones:

$$\frac{du}{dy} = au - buv$$
$$\frac{dv}{dt} = -cv + dbuv$$

donde:

- u es el número de presas (ej. Conejos)
- v es el número de depredadores (ej. Zorros)
- a es la tasa natural de crecimiiento de conejos, sin que haya zorros.
- b es la tasa natural de la muerte de conejos, debido a la depredación.
- c es la tasa natural de la muerte del zorro, cuando no hay conejos.
- d es el factor que describe el número de conejos capturados.

Vamos a utilizar $X = [u, v]$ para describir el estado de las poblaciones.

Definiendo las ecuaciones

```
1  from numpy import *
2  import pylab as p
3
4  a = 1.
5  b = 0.1
6  c = 1.5
7  d = 0.75
8
9  def dXdt(X, t=0):
10     return array([ a*X[0] -   b*X[0]*X[1] , -
           c*X[1] + d*b*X[0]*X[1] ])
```

Población en equilibrio

Antes de usar Scipy para integrar el sistema, veremos de cerca la posición de equilibrio. El equilibrio ocurre cuando la tasa de crecimiento es igual a 0, lo que nos da dos puntos fijos:

```
1 Xf0 = array([      0.  ,   0.])
2 Xf1 = array([ c/(d*b), a/b])
3 all(dXdt(X_f0) == zeros(2) ) and all(dXdt(X_f1
    ) == zeros(2))
```

Para usar la función `odeint`, hay que definir el parámetro de tiempo t, así como las condiciones inciales de la población: 10 conejos y 5 zorros.

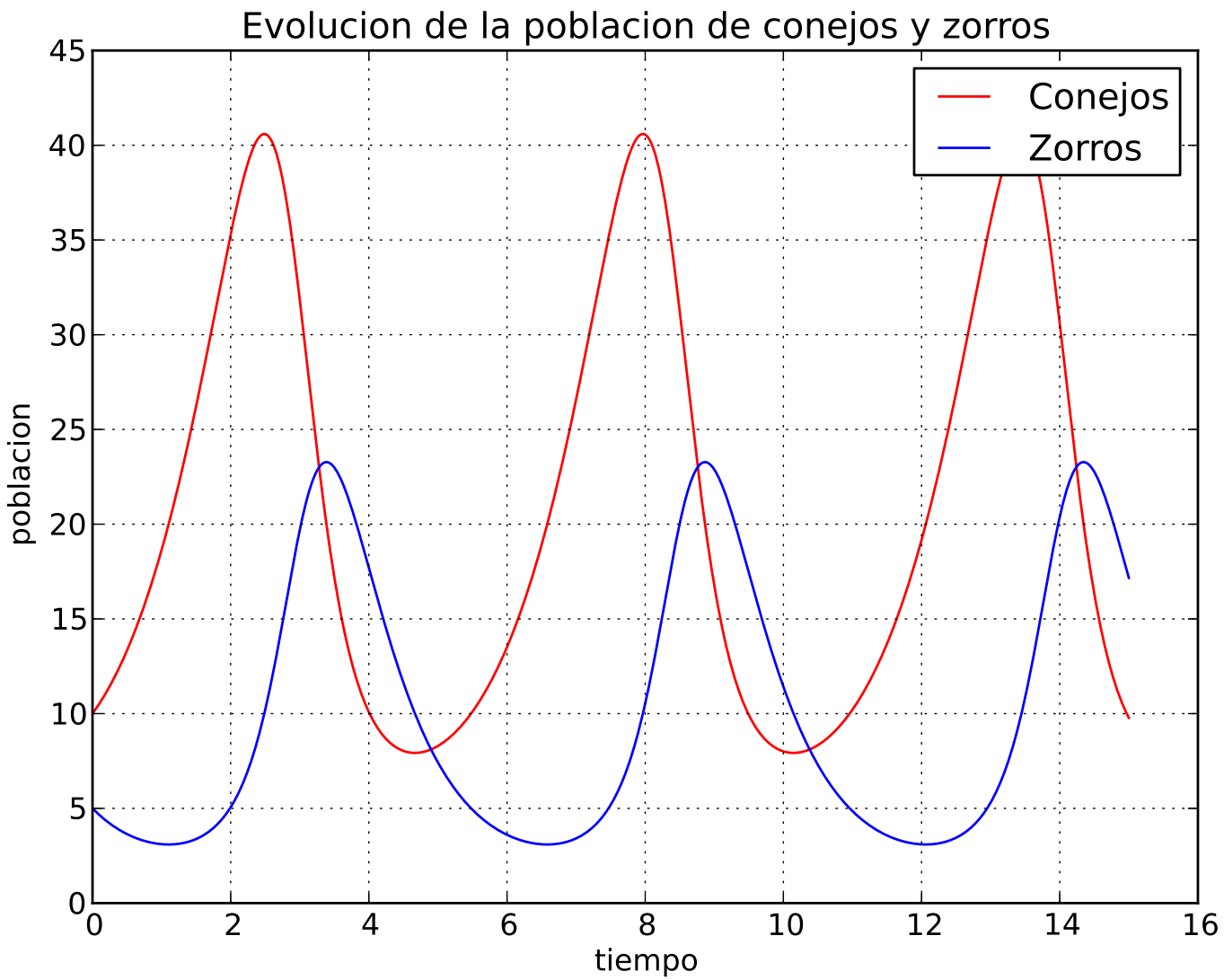
```
1 t = linspace(0, 15, 1000)
2
3 X0 = array([10, 5])
4
5 X = integrate.odeint(dXdt, X0, t)
```

La función `odeint` requiere de tres argumentos: la función (o arreglo de funciones), la condición inicial (o arreglo de condiciones iniciales) y el tiempo.

Una vez obtenido el código para la solución del problema, ahora nos corresponde graficar el conjunto de datos obtenido, para ello, usamos la siguiente rutina en Python:

```
1  conejos, zorros = X.T
2
3  f1 = p.figure()
4  p.plot(t, conejos, 'r-', label='Conejos')
5  p.plot(t, zorros  , 'b-', label='Zorros')
6  p.grid()
7  p.legend(loc='best')
8  p.xlabel('tiempo')
9  p.ylabel('poblacion')
10 p.title('Evolucion de la poblacion de conejos
       y zorros')
11 p.show()
```

Resultado gráfico



La gráfica anterior nos da la información sobre el número tanto de conejos como de zorros durante el intervalo de tiempo estudiado, es decir, tenemos una especie de "censo.

Para ver la dinámica de las poblaciones propiamente, ahora representamos el espacio fase del sistema, por lo que tenemos que hacer algunos ajustes en el código que usamos anteriormente.

Consideremos las condiciones de equilibrio, es decir, donde la tasa de crecimiento es cero:

```
1 X_f0 = array([ 0. , 0.])
2 X_f1 = array([ c/(d*b), a/b])
```

Dibujaremos el espacio fase con algunos elementos visuales con el fin de decoración nada más.

```
values  = linspace(0.3, 0.9, 5)

vcolors = p.cm.autumn_r(linspace(0.3, 1., len(
    values)))

f2 = p.figure()
```

El módulo `cm` proporciona un gran conjunto de mapas de colores, así como las funciones para crear nuevos mapas de color; existen varios mapas ya definidos: autumn, bone, cool, copper, flag, gray, hot, hsv, jet, pink, prism, spring, summer, winter, spectral.

Se van a dibujar ahora las trayectorias para diferentes condiciones iniciales (número de conejos y zorros)

```
1 for v, col in zip(values, vcolors):
2     X0 = v * X_f1
3     X = integrate.odeint( dX_dt, X0, t)
4     p.plot( X[:,0], X[:,1], lw=3.5*v, color=
          col, label='X0=(%.f, %.f)' %( X0[0],
          X0[1]) )
```

La función `zip` sirve para reorganizar las listas en Python. Como parámetros admite un conjunto de listas. Lo que realmente hace es tomar el elemento i-ésimo elemento de cada lista y los une en una tupla, después une todas las tuplas en una lista.

En cada gráfica se modifica el grosor de la línea y el color que se le asocia.

Se define una malla y se calcula la dirección

```
1 ymax = p.ylim(ymin=0)[1]
2 xmax = p.xlim(xmin=0)[1]
3 nb_points    = 20
4
5 x = linspace(0, xmax, nb_points)
6 y = linspace(0, ymax, nb_points)
```

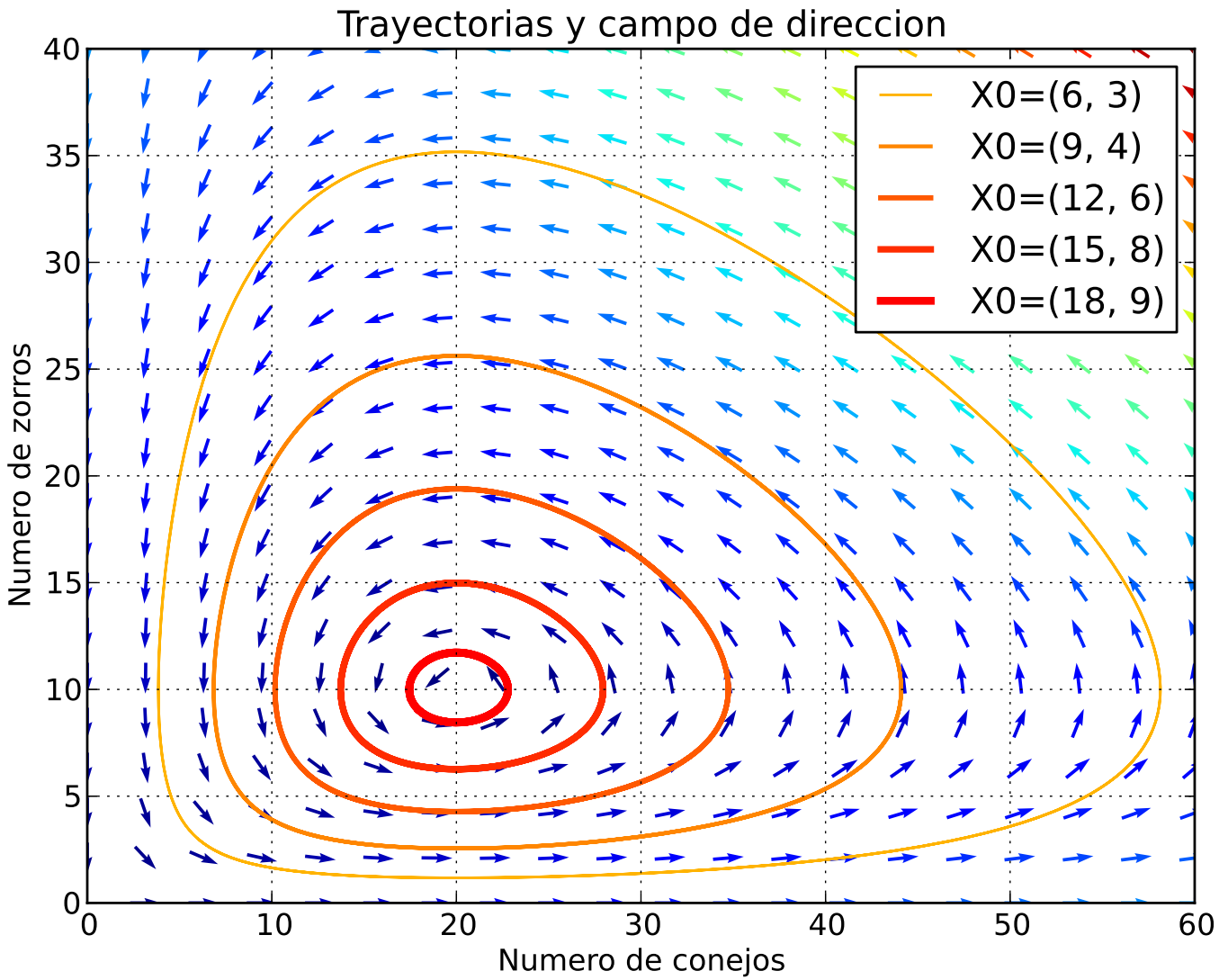
```
X1 , Y1 = meshgrid(x, y)
DX1, DY1 = dX_dt([X1, Y1])
M = (hypot(DX1, DY1))
M[ M == 0] = 1.
DX1 /= M
DY1 /= M
```

Se dibujan las direcciones usando quiver

```
1 p.title('Trayectorias y campo de direccion')
2 Q = p.quiver(X1, Y1, DX1, DY1, M, pivot='mid',
      cmap=p.cm.jet)
3 p.xlabel('Numero de conejos')
4 p.ylabel('Numero de zorros')
5 p.legend()
6 p.grid()
7 p.xlim(0, xmax)
8 p.ylim(0, ymax)
9 p.show()
```

La función `quiver` genera el mapa vectorial, requiere de cinco argumentos: las posiciones X1,Y1 de inicio, el valor de las componentes del vector DX1, DY1 y el color asociado, el argumento `pivot` indica en qué parte de la malla se va a colocar el vector.

Resultado gráfico



Ejercicio para resolver

El modelo de Lorenz se usa para estudiar la formación de torbellinos en la atmósfera, aunque abordó el problema de manera general, estableció las bases para el estudio de sistemas dinámicos, el conjunto de ecuaciones está dado por

$$\frac{dy_1}{dt} = a(y_2 - y_1)$$
$$\frac{dy_2}{dt} = (b - y_3)y_1 - y_2$$
$$\frac{dy_3}{dt} = y_1 y_2 - cy_3$$

en el modelo,a, b y c son parámetros positivos. Resuelve este modelo numéricamente, grafica la solución

Resultado gráfico

Usando $a = 10$, $b = 28$ y $c = 8/3$