Eddy Herrera Daza eherrera@javeriana.edu.co

Eddy Herrera Daza -Análisis Numérico

Ecuaciones Diferenciales Ordinarias: Métodos Numéricos

Introducción

Cuando los métodos analíticos para resolver una ecuación diferencial ordinaria no se pueden aplicar, por ejemplo para resolver la ecuación de variables separable:

$$\frac{dy}{dx} = e^{-x^2}$$

Los métodos numéricos son una salida para este tipo de problemas, es decir tendremos no una solución exacta sino una aproximación numérica a una solución de una Ecuación diferencial. De éstos métodos, la salida no es una función analitica $y(x)$ o una familia de funciones $\{y(x) + C\}$, sino un vector de valores numéricos que representan la solución en cada x_0 . Existen muchos enfoques para resolver problemas relacionados con ecuaciones diferenciales ordinarias en forma numérica, y generalmente están diseñados para resolver problemas que están formulados como un sistema de ecuaciones diferenciales de primer orden de la forma estándar:

I Problema con valores Iniciales:

Para una ecuación diferencial ordinaria (EDO) de orden n:

$$\begin{cases} F(x, y, \frac{dy}{dx}, \dots, \frac{d^n y}{dx^n} \\ y(x_0) = y_0 \\ \vdots \\ \frac{d^{n-1} y}{dx^{n-1}}(x_0) = y_{n-1} \end{cases}$$

Donde, $x_0 \in I$ (intervalo donde se encuentra la solución) y $y_0, \dots, y_{n-1}$ son constantes dadas

Python: **SymPy** nos proporciona un solucionador genérico de Ecuaciones diferenciales ordinarias, *sympy.dsolve*, el cual es capaz de encontrar soluciones

analíticas a muchas EDOs elementales. Mientras sympy.dsolve se puede utilizar para resolver muchas EDOs sencillas simbólicamente, como veremos a continuación, debemos tener en cuenta que la mayoría de las EDOs no se pueden resolver analíticamente.

Ejemplo:

Dada la siguiente ecuación diferencial de primer orden encontrar su solución

$$\frac{dy}{dx} = -3x^2y(x) + 6x^2$$

Forma de la EDO: Ecuación lineal porque corresponde a la forma:

$$\frac{dy}{dx} + 3x^2y(x) = 6x^2: \frac{dy}{dx} + P(x)y = Q(x)$$

Solución Analítica "Factor integrante"

$I(x) = e^{\int P(x)dx}$ Luego la solución $y(x) = \frac{1}{I(x)}[\int I(x)Q(x)dx + C]$

$$I(x) = e^{\int 3x^2dx} = e^{x^3} \rightarrow$$

$$y(x) = \frac{1}{e^{x^3}}\left[\int e^{x^3}6x^2dx + C\right] = \frac{1}{e^{x^3}}\left[2e^{x^3} + C\right] = 2 + Ce^{-x^3}$$

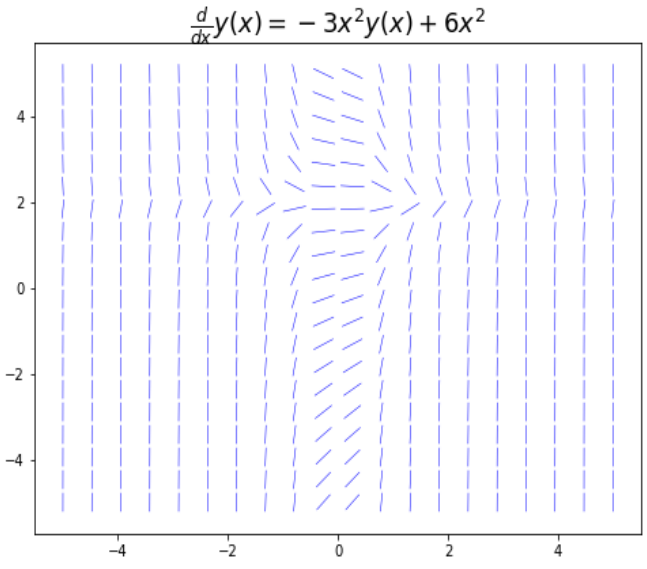
Utilizando Python:

Código

```
# importando módulos necesarios
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import sympy
from scipy import integrate
# imprimir con notación matemática.
sympy.init_printing(use_latex='mathjax')
# imprimir con notación matemática.
sympy.init_printing(use_latex='mathjax')
# defino las incognitas
x = sympy.Symbol('x')
y = sympy.Function('y')
# expreso la ecuacion
f = 6*x**2 - 3*x**2*(y(x))
# Resolviendo la ecuación
sympy.dsolve(y(x).diff(x) - f)
```

Out [2]:

$$y(x) = C_1e^{-x^3} + 2$$



Si además el problema tiene la siguiente condición inicial $y(0) = 0$ entonces,

```
# Condición inicial
ics = {y(0): 0}
f = 6*x**2 - 3*x**2*(y(x))
# Resolviendo la ecuación diferencial
edo_sol = sympy.dsolve(y(x).diff(x) – f)
edo_sol
#Sustitur la condicion
C_eq = sympy.Eq(edo_sol.lhs.subs(x,
0).subs(ics), edo_sol.rhs.subs(x, 0))
C_eq
#valor de la constante
sympy.solve(C_eq)
```

II Método de Euler "Método de la recta tangente"

El método de Euler es un procedimiento que permite construir aproximaciones a las soluciones de un problema con valor inicial con una ecuacion diferencial ordinaria de primer orden. Por ejemplo, dado el siguiente problema de valor inicial

$$\frac{dy}{dx} = f(x, y)\ , y(x_0) = x_0$$

Si tenemos, la pendiente de la curva $(\frac{dy}{dx})$ en cualquier punto. Es decir, el ritmo de cambio de la curva, que no es otra cosa que su derivada, la cual podemos utilizar para iterar sobre soluciones en distintos puntos.

Sea el punto inicial (x_0, y_0) que se encuentra sobre la curva solución (que es $y(x)$) con pendiente $f(x_0, y_0) = \frac{dy}{dx}|(x_0, y_0)$

¿Qué pasaría entonces si quisiéramos encontrar la solución numérica en el punto (x, y) que se encuentra a una corta distancia h? En este caso, podríamos definir a la ecuación de una línea recta como: $y = y_0 + \Delta y$, es decir, y(x) va ser igual al valor de y en el punto inicial, más el cambio que ocurrió en y al movernos a la distancia h.

$y = y_0 + \Delta y = y = y_0 + h\frac{dy}{dx}|(x_0, y_0)$ Luego, $y = y_0 + hf(x_0, y_0)$

En general, el método de Euler comienza por el punto dado por la condición inicial y continuemos en la dirección indicada por el campo de direcciones (o campo de pendientes). Luego nos detengamos, miramos a la pendiente en la nueva ubicación, y procedemos en esas direcciones.

Lo anterior, se puede generalizar:

Dado el problema:

$$\frac{dy}{dx} = f(x,y)\ , y(x_0) = x_0$$

Con solución única $\Phi(x)$ en cierto intervalo de centro x_0. Sea h un número positivo fijo (tamaño del paso) y sean $x_n = x_0 + nh; n = 0,1,2, \ldots$. La construcción de los valores y_n que aproximan los valores de la solución $\Phi(x_n)$ están dados por:

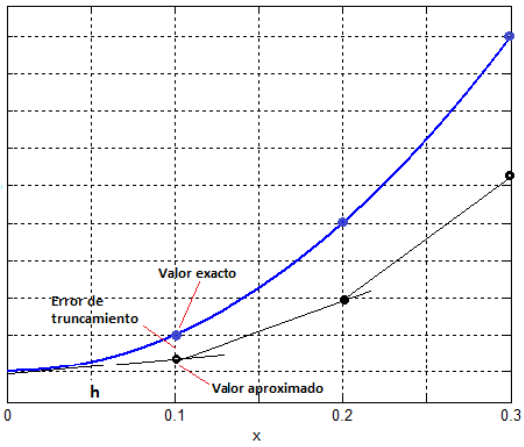
$$x_{n+1} = x_n + h$$

$$\Phi(x_{n+1}) \approx y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2!} f^{"}(\xi), \text{con } x_i \leq \xi \leq x_{i+1}$$

Luego, el error de truncamiento $\frac{h^2}{2!} f^{"}(\xi)$ entonces el orden es $O(h^2)$

Error de Truncamiento y Error de Redondeo

La formula de Euler utiliza la pendiente de la recta en cada punto para predecir y estimar la solución en el siguiente punto, a una distancia elegida h. La diferencia del punto calculado, con respecto al valor exacto es el error de truncamiento, el cual puede crecer al proseguir el cálculo.



Para $E \to 0, h \to 0$ sin embargo, la consecuencia de esto está dado por:

$$E_T = \sum_{i=1}^{i=m} E_i$$

En cuanto el error de redondeo E_R se tiene que cada vez que se toma una medida $f(x_i, y_i)$ se introduce un error de redondeo, que se acumula en cada paso:

$E_R = \sum_{i=1}^{i=m} R_i = m\bar{R}$ con $h = \frac{b-a}{m}$ para algún $\bar{R}$ en cada paso. Luego se tiene que el error total esta dado:

$$ET = E_T + E_R$$

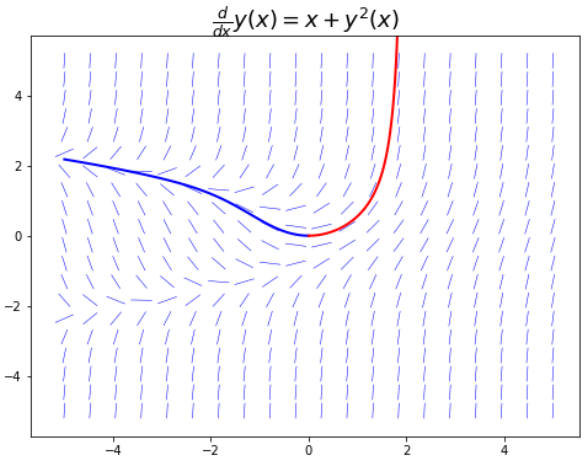
Por lo tanto, si m es muy grande, h será muy pequeño y el error total será pequeño, pero al reducir h, el error de redondeo puede crecer y llegar a ser mayor al error de truncamiento; por lo tanto el resultado perderá precisión en vez de aumentarla. Luego, es mejor utilizar formulas con error de truncamiento de mayor orden, si se quieren valores con mayor precisión

Soluciones Numéricas con Python: Para resolver EDO en forma numérica con Python, podemos utilizar las herramienta de integración que nos ofrece SciPy, particularmente los dos solucionadores:: integrate.odeint y integrate.ode. La principal diferencia entre ambos, es que integrate.ode es más flexible, ya que nos ofrece la posibilidad de elegir entre distintos solucionadores

Ejemplos

```
# Defino la función
f = y(x)**2 + x
f
# la convierto en una función ejecutable
f_np = sympy.lambdify((y(x), x), f)

# Definimos los valores de la condición inicial y el rango
#de x sobre los
# que vamos a iterar para calcular y(x)
y0 = 0
xp = np.linspace(0, 1.9, 100)

# Calculando la solución numerica para los valores de y0
#y xp
yp = integrate.odeint(f_np, y0, xp)

# Aplicamos el mismo procedimiento para valores de x ne
gativos
xn = np.linspace(0, -5, 100)

yn = integrate.odeint(f_np, y0, xn)

xn

yn
```

```
fig, axes = plt.subplots(1, 1, figsize=(8, 6))
plot_direction_field(x, y(x), f, ax=axes)
axes.plot(xn, yn, 'b', lw=2)
axes.plot(xp, yp, 'r', lw=2)
plt.show()
```



Otra forma:

```
def euler(f,x,y,h,m):
    u=[]
    v=[]
    for i in range(m):
        y=y+h*f(x,y)
        x=x+h
        u=u+[x]
        v=v+[y]
    return [u,v]
```

```
from pylab import*
from euler import*
def f(x,y):return y-x**2+x+1
[u,v]=euler(f,0,1,0.1,20)
plot(u,v,'or')
def y(x):return exp(x)+x**2+x
x=arange(0,2.1,0.1)
plot(x,y(x),'ob')
grid(True)
show()
```

El Método de Runge-Kutta

Otro método que podemos utilizar para encontrar soluciones numéricas a Ecuaciones diferenciales, y que incluso es más preciso que el Método de Euler, es el Método de Runge-Kutta. En el cual la relación de recurrencia va a estar dada por un promedio ponderado de términos de la siguiente manera:

$$y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

en dónde:

$$k_1 = f(x_k, y_k)h_k,$$
$$k_2 = f\left(x_k + \frac{h_k}{2}, y_k + \frac{k_1}{2}\right)h_k,$$
$$k_3 = f\left(x_k + \frac{h_k}{2}, y_k + \frac{k_2}{2}\right)h_k,$$
$$k_4 = f\left(x_k + h_k, y_k + k_3\right)h_k.$$

Sistema de Ecuaciones:

En este ejemplo, solucionamos solo una ecuación. Generalmente, la mayoría de los problemas se presentan en la forma de sistemas de ecuaciones diferenciales ordinarias, es decir, que incluyen varias ecuaciones a resolver. Para ver cómo podemos utilizar a integrate.odeint para resolver este tipo de problemas, consideremos el siguiente sistema de ecuaciones diferenciales ordinarias, conocido el atractor de Lorenz:

Estas ecuaciones son conocidas por sus soluciones caóticas, que dependen sensiblemente de los valores de los parámetros. Veamos cómo podemos resolverlas con la ayuda de Python.

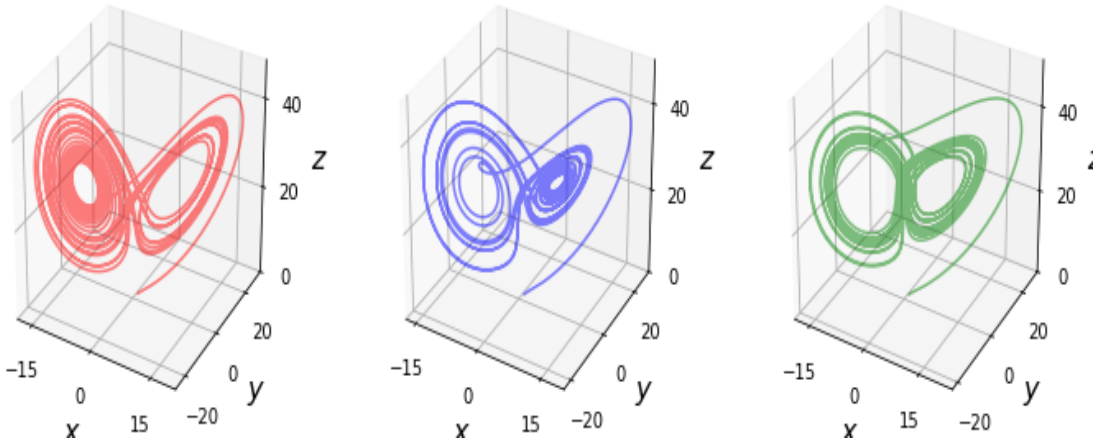
$$x'(t) = \sigma(y - x),$$
$$y'(t) = x(\rho - z) - y,$$
$$z'(t) = xy - \beta z$$

```
# Definimos el sistema de ecuaciones
def f(xyz, t, sigma, rho, beta):
    x, y, z = xyz
    return [sigma * (y - x),
            x * (rho - z) - y,
            x * y - beta * z]

# Asignamos valores a los parámetros
sigma, rho, beta = 8, 28, 8/3.0

# Condición inicial y valores de t sobre los que calcular
xyz0 = [1.0, 1.0, 1.0]
t = np.linspace(0, 25, 10000)

# Resolvemos las ecuaciones
xyz1 = integrate.odeint(f, xyz0, t, args=(sigma, rho, beta))
xyz2 = integrate.odeint(f, xyz0, t, args=(sigma, rho, 0.6*beta))
xyz3 = integrate.odeint(f, xyz0, t, args=(2*sigma, rho, 0.6*beta))
```

Grafica de las soluciones:

```
# Graficamos las soluciones
from mpl_toolkits.mplot3d.axes3d import Axes3D
fig, (ax1,ax2,ax3) = plt.subplots(1, 3, figsize=(12, 4),
                                  subplot_kw={'projection':'3d'})

for ax, xyz, c in [(ax1, xyz1, 'r'), (ax2, xyz2, 'b'), (ax3, xyz3, 'g')]:
    ax.plot(xyz[:,0], xyz[:,1], xyz[:,2], c, alpha=0.5)
    ax.set_xlabel('$x$', fontsize=16)
    ax.set_ylabel('$y$', fontsize=16)
    ax.set_zlabel('$z$', fontsize=16)
    ax.set_xticks([-15, 0, 15])
    ax.set_yticks([-20, 0, 20])
    ax.set_zticks([0, 20, 40])
```

Eddy Herrera Daza eherrera@javeriana.edu.co



Código :Generar Campo de Pendientes

```
def plot_direction_field(x, y_x, f_xy, x_lim=(-5, 5), y_lim=(-5, 5), ax=None):
"""Esta función dibuja el campo de dirección de una EDO"""
    f_np = sympy.lambdify((x, y_x), f_xy, modules='numpy')
    x_vec = np.linspace(x_lim[0], x_lim[1], 20)
    y_vec = np.linspace(y_lim[0], y_lim[1], 20)

    if ax is None:
        _, ax = plt.subplots(figsize=(4, 4))

    dx = x_vec[1] - x_vec[0]
    dy = y_vec[1] - y_vec[0]

    for m, xx in enumerate(x_vec):
        for n, yy in enumerate(y_vec):
            Dy = f_np(xx, yy) * dx
            Dx = 0.8 * dx**2 / np.sqrt(dx**2 + Dy**2)
            Dy = 0.8 * Dy*dy / np.sqrt(dx**2 + Dy**2)
            ax.plot([xx - Dx/2, xx + Dx/2],
                    [yy - Dy/2, yy + Dy/2], 'b', lw=0.5)

    ax.axis('tight')
    ax.set_title(r"$%s$" %
                 (sympy.latex(sympy.Eq(y(x).diff(x), f_xy))),
                 fontsize=18)

    return ax
#Ejemplo
Defino incognitas
x = sympy.symbols('x')
y = sympy.Function('y')

# Defino la función
f = y(x)**2 + x**2 -1
# grafico de campo de dirección
fig, axes = plt.subplots(1, 1, figsize=(8, 6))
campo_dir = plot_direction_field(x, y(x), f, ax=axes)
```