Análisis de Solvers de EDO en Diferentes Herramientas

- Esneider Fabian Sierra Alba
- Juan José Martínez Castiblanco
- Juan Carlos Sánchez Orjuela

Selección de Herramientas

Solver 1: Python, SciPy.

SciPy es una librería gratis y de codigo abierto(Open Source) de Python para computación científica que incluye el módulo integrate, el cual contiene funciones para resolver EDOs. Esta libreria fue lanzada desarrollada inicialmente en 2001 por Travis Oliphant, Pearu Peterson y Eric Jones.

Documentación: SciPy Integrate

Repositorio: SciPy GitHub

Fun fact: Primer commit

Solver 2: Matlab, ode23s.

Matlab es un lenguaje de programación de alto nivel, que, como Python, tiene gran presencia en el mundillo de la computación científica. Matlab surge en los años 70, a manos de Cleve Barry Moler, como un paquete de computación numérica con un fuerte enfoque en operaciones matriciales.

Documentación: Matlab ode23s

Análisis de Solvers

Solver 1: Python, SciPy

El análisis del código se hizo usando la versión 1.15.2.

SciPy cuenta con los siguientes métodos para resolver problemas de valor inicial en EDO:

- `solve_ivp`
- `RK23`
- `RK45`
- `DOP853`
- `Radau`
- `BDF`
- `LSODA`
- `OdeSolver`
- `DenseOutput`
- `OdeSolution`

En este caso analizaremos `solve_ivp`.

`solve_ivp`

Este método puede resolver un problema de EDO con valor inicial utilizando cualquiera de los métodos descritos anteriormente. Por defecto, éste usa `RK45` explícito, es decir, el método de Runge–Kutta **explícito** de orden 5. El 4 en el nombre se debe a que se utiliza un método de Runge–Kutta de orden 4 para comparar la precisión, es decir, los errores se estiman comparando las soluciones de orden 4 y 5.

Esta función recibe los siguientes parámetros:

- **fun:** La función que define la EDO en la forma $\frac{dy}{dt} = f(t, y)$. Esta función debe poder llamarse como `fun(t, y)`, donde y debe tener la misma longitud que y_0.
- **t_span:** Intervalo de evaluación (t_0, t_f).
- **y0:** Valor inicial en forma de arreglo.
- **dense_output** (opcional, `False`): Booleano para especificar si se desea calcular una solución continua.
- **events** (opcional, *callables*): Lista de funciones (*callables*) que se ejecutarán cada vez que se encuentre un cero en la función.
- **args** (opcional, tupla): Argumentos adicionales que se pasarán a la función. Una función que necesite argumentos se envolverá en un *wrapper* para ser llamada por el solver de la siguiente manera:

```
# scipy/integrate/_ivp/ivp.py
def fun(t, x, fun=fun):
    return fun(t, x, *args)
```

- **options** (opcional): Opciones que se pasarán al solver, como se ve a continuación:

```
# scipy/integrate/_ivp/ivp.py
solver = method(fun, t0, y0, tf, vectorized=vectorized, **options)
```

Algunas opciones relevantes son:

- **first_step** (opcional, flotante): Tamaño del paso inicial. Si no se proporciona, el algoritmo elegirá uno.
- **max_step** (opcional, flotante): Tamaño máximo del paso. Si no se proporciona, el algoritmo elegirá uno.
- **atol, rtol** (opcional, flotante o `array_like`): Tolerancia absoluta y relativa, respectivamente. El solver garantiza que el error en cada paso satisface $\text{error} \leq atol + rtol\,|y|$

Este solver funciona de la siguiente manera:

Una vez llamada la función, se procede a verificar que el método asignado esté implementado y sea del tipo `OdeSolver` (el solver soporta métodos personalizados). Luego, se comprueba que los puntos de los intervalos sean válidos para, posteriormente, instanciar una clase del solver, como se muestra a continuación:

```
# scipy/integrate/_ivp/ivp.py
solver = method(fun, t0, y0, tf, vectorized=vectorized, **options)
```

En el caso por defecto, se instancia la clase `RK45`, que hereda de la clase `RungeKutta`. En esta inicialización se declaran las siguientes variables relevantes:

```
# scipy/integrate/_ivp/rk.py
C = np.array([0, 1/5, 3/10, 4/5, 8/9, 1])
A = np.array([
    [0, 0, 0, 0, 0],
    [1/5, 0, 0, 0, 0],
    [3/40, 9/40, 0, 0, 0],
```

```
    [44/45, -56/15, 32/9, 0, 0],
    [19372/6561, -25360/2187, 64448/6561, -212/729, 0],
    [9017/3168, -355/33, 46732/5247, 49/176, -5103/18656]
])
```

Donde A, B, C y E corresponden al Butcher tableau expandido para calcular la solución de orden 4. Estos coeficientes se obtienen al resolver los pesos que se deben asignar a la solución de orden 5 para obtener la de orden 4; nótese que esto funciona debido a que existen muchos valores similares en el Butcher tableau para ambas soluciones. Este método también es conocido como método Dormand–Prince.

Con la clase instanciada se procede a evaluar cada paso del solver, como se ve a continuación:

```
# scipy/integrate/_ivp/ivp.py
while status is None:
    message = solver.step()

    if solver.status == 'finished':
        status = 0
    elif solver.status == 'failed':
        status = -1
        break
```

Al llamar a `solver.step()` se invoca la implementación de Runge–Kutta `_step_impl()`, en la que se procede al cálculo del tamaño del paso:

```
# scipy/integrate/_ivp/ivp.py

min_step = 10 * np.abs(np.nextafter(t, self.direction * np.inf) - t)

if self.h_abs > max_step:
    h_abs = max_step
elif self.h_abs < min_step:
    h_abs = min_step
else:
    h_abs = self.h_abs
```

Nótese que se utiliza `np.nextafter`; esto determina la tolerancia mínima del sistema, por lo que el tamaño del paso puede llegar hasta $eps \times 10$. Una vez definido el paso, se procede a evaluar la función para obtener y_{n+1} y $f(t + h, y_{n+1})$:

```
# scipy/integrate/_ivp/ivp.py
...
y_new, f_new = rk_step(self.fun, t, y, self.f, h, self.A,
                                    self.B, self.C, self.K)
...
self.t = t_new
self.y = y_new
...
def rk_step(fun, t, y, f, h, A, B, C, K):
    K[0] = f
    for s, (a, c) in enumerate(zip(A[1:], C[1:]), start=1):
        dy = np.dot(K[:s].T, a[:s]) * h
        K[s] = fun(t + c * h, y + dy)

    y_new = y + h * np.dot(K[:-1].T, B)
    f_new = fun(t + h, y_new)

    K[-1] = f_new
```

En este código se evalúan los coeficientes (K) y se calcula (y{n+1}) *(almacenado en `y*new`)* según: $k_s = f(t_n + c_s\,h,\ y_n + h\sum_{j=1}^{s-1} a_{sj}\,k_j)$ guardando los respectivos k_s en una matriz. A continuación, se procede a calcular el error y, si éste ajustado a la escala basada en la tolerancia asignada resulta menor que 1, se acepta el paso; de lo contrario, se ajusta el paso hasta que el cálculo cumpla con el error requerido.

```
# scipy/integrate/_ivp/ivp.py
scale = atol + np.maximum(np.abs(y), np.abs(y_new)) * rtol
error_norm = self._estimate_error_norm(self.K, h, scale)
```

Luego, se verifica que el paso se encuentre dentro de los límites de evaluación y, de no ser así, se termina la ejecución del algoritmo de Runge–Kutta:

```
# scipy/integrate/_ivp/base.py
if not success:
    self.status = 'failed'
else:
    self.t_old = t
    if self.direction * (self.t - self.t_bound) >= 0:
        self.status = 'finished'
```

Si aún se está dentro del límite de evaluación, se procede a guardar los nuevos (t) e (y) calculados:

```
# scipy/integrate/_ivp/ivp.py
...
t = solver.t
y = solver.y
...
ts.append(t)
ys.append(y)
...
```

Una vez terminada la ejecución de RK, se crea el objeto solución, que contiene principalmente un arreglo con los tiempos de ejecución del algoritmo y un arreglo con la función $y(t)$ evaluada en cada punto.

Un ejemplo sencillo es la ecuación de decaimiento exponencial $\frac{dy}{dt} = -0.5y, \quad y(0) = 2$, Usando la librería se obtiene:

```
import numpy as np
from scipy.integrate import solve_ivp

def exponential_decay(t, y):
    return -0.5 * y

sol = solve_ivp(exponential_decay, [0, 5], [2])

print(sol.t)
print(sol.y)

# Salida:
# [0.         0.11488132 1.26369452 3.06074656 4.81637262 5.        ]
# [[2.         1.88835583 1.0632438  0.43316531 0.18014905 0.16434549]]
```

Nótese que la solución exacta es $y(t) = 2e^{-0.5t}$.

Solver 2: ode23s

El solver `ode23s` se basa en los métodos de Rosenbrock, estos son una generalización de los métodos de runge-Kutta y son un compromiso entre ser implícitos y lineales. `ode23s` esta como tal disenado para resolver EDOs rígidas, usa un método de orden 2 sobre el tamaño del paso, tamaños de paso adaptativos y de un solo paso, es decir, no utiliza la información anterior a algún tiempo t_n para construir la del tiempo t_{n+1}[1][2].

Para el caso de un problema de valor inicial, autónomo (por empezar por una versión más simple): $y' = F(y)$; para convertir el problema en mera álgebra lineal, los métodos de Rosenbrock tienen la forma:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

Donde s nos va a dar el orden del método, y los valores k_i son las soluciones para:

$$Wk_i = F\left(y_n + h\sum_{j=1}^{i-1} a_{ij}k_j\right)\$ + hJ\sum_{j=1}^{i-1} d_{ij}k_j$$

Arriba, h es el tamaño del paso, los valores a_{ij}, d y d_{ij} son constantes especialmente seleccionadas para que el error sea de cierto orden; J es el Jacobiano de F evaluado en (t_n, y_n), y $W = I - hdJ$.

La fórmula de segundo orden (s=2) luce así:

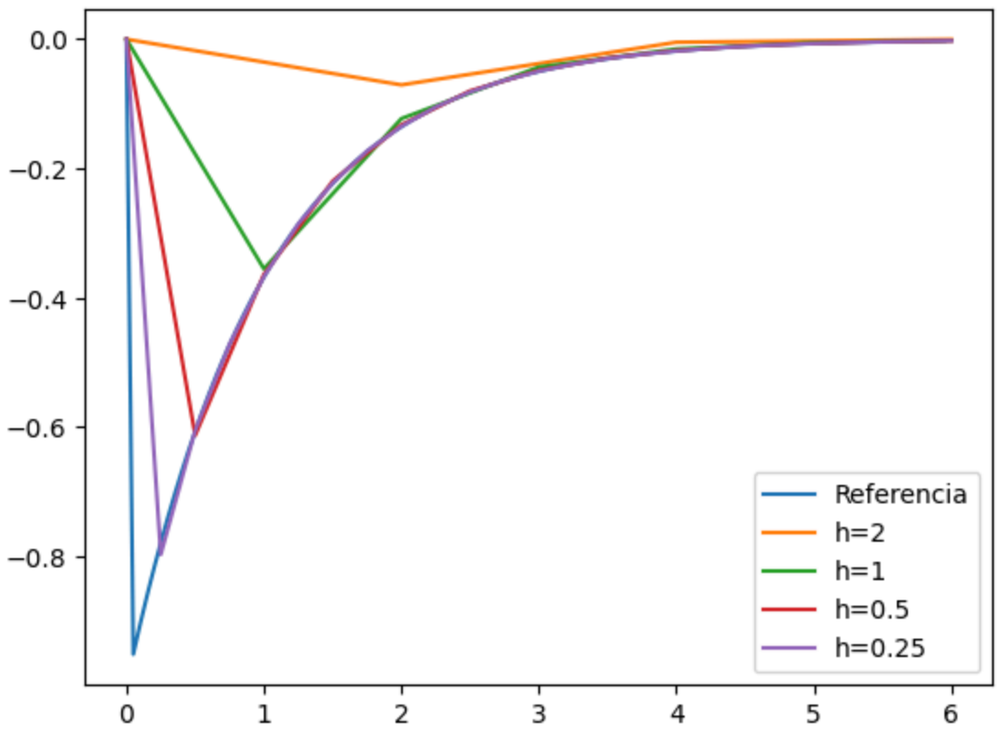
$$\begin{cases} Wk_2 = F(y_n) \\ Wk_2 = f\left(y_n + \frac{2}{3}hk_1\right) - \frac{4}{3}hdJk_1 \\ d = 1\frac{1}{2+\sqrt{2}} \\ y_{n+1} = y_n + \frac{h}{4}(k_1 + 3k_2) \end{cases}$$

http://bicycle.tudelft.nl/schwab/TAM674/SR97.pdf

El siguiente código de Python ilustra superficialmente algunas aproximaciones para distintos tamaños de paso fijos usando esta última fórmula.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate

def Jac(y):
    return np.array([[998, 1998],[-999, -1999]])

def sir(y,x=None):
      S = 998*y[0] + 1998*y[1]
      I = -999*y[0] - 1999*y[1]
      return np.array([S,I])

def approach_n_plot(h):
    y = [1, 0]
    ys = [0]
    xs = [0]

    for i in range(int(6/h)):
        J = Jac(y)
        B1 = sir(y)
        W = np.identity(2) - h*J/(2+2**0.5)

        k1 = np.linalg.solve(W,  B1)

        y2 = y + 2*h*k1/3
        B2 = sir(y2) - 4*h/(3*(2+2**0.5)) * (J.dot(k1))
        k2 = np.linalg.solve(W,  B2)

        # Calcular y_{n+1}
        y = y + h*(k1+3*k2)/4

        ys.append(y[1])
        xs.append(xs[-1]+h)
    lab = "h=" + str(h)
    plt.plot(xs, ys, label=lab)

# Solucion referencia
odxs = [x/20 for x in range(120)]
fitted = integrate.odeint(sir, y0=[1, 0], t=odxs)[:,1]

h = 1
plt.plot(odxs, fitted, label="Referencia")
approach_n_plot(2*h)
approach_n_plot(h)
approach_n_plot(h/2)
approach_n_plot(h/4)
plt.legend()
plt.show()
```

La gráfica resultante se muestra en seguida:



El caso general tiene la forma:

$$\begin{cases} (I - hd_{ii}J)k_{ni} = F\left(t_n + a_i h,\ y_n + h\sum_{j=1}^{i-1} a_{ij}k_{nj}\right) + hJ\sum_{j=1}^{i-1} d_{ij}k_{nj} + hd_i J \\ y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_{ni} \end{cases}$$

Con $a_i = \sum_{j=1}^{i-1} a_{ij}$ y $d_i = \sum_{j=1}^{i-1} d_{ij}$

La implementación de Matlab recibe los parámetros:

- **odefun:** La función que define la EDO en la forma.
- **tspan:** Intervalo de tiempo de evaluación (t_0, t_f).
- **y0:** Condición inicial en forma de arreglo.
- **options** (opcional): creadas con el comando `odeset`. Estas permiten configurar tolerancias: `AbsTol`, `ResTol`, especificar un Jacobiano, activar la visualización de estadísticas del progreso del solver, entre otras opciones.

Este método retorna dos arreglos: **t** y **y**. Cada fila en el segundo corresponde a un valor en el primero. Alternativamente, puede retornar una estructura para evaluar la solución en cualquier punto del intervalo entregado, cosa que liciria de la forma:

```
sol = ode23s(odefun,tspan,y0,options)
```

Opcionalmente, se pueden solicitar información respecto a los ceros de **t**.

Para decidir el tamaño de paso, Matlab utiliza Control de Tamaño de Paso Adaptativo. Para esto estima el error en una iteración calculando una aproximación de orden 3 (recicla los cálculos realizados para k1 y k2 para calcular k3, cabe mencionar que esta aproximación no es estable en el tiempo) que usada junto a la primera aproximación da una idea del error proveniente de la iteración. Dependiendo de las tolerancias definidas al llamar a la función, el paso se ajusta para reducir el error o ahorrar recursos.

Podemos considerar el ejemplo que usamos con solve_ivp: $\frac{dy}{dt} = -0.5y, \quad y(0) = 2$. Usando ode23 se obtiene:

```
>> %Solución numérica usando ode23s
tspan = 0:0.1:5;
opts = odeset('Stats','on');
y0 = 2;
[t,y] = ode23s(@(t,y) -0.5*y, tspan, y0);

%Solución numérica usando solve_ivp
a = [0.         0.11488132 1.26369452 3.06074656 4.81637262 5.        ];
b = [2.         1.88835583 1.0632438  0.43316531 0.18014905 0.16434549];

%Solución analítica
x = 0:0.1:5;
z = 2*exp(-0.5*x);

%Gráficas
tiledlayout(3,1)
nexttile
plot(t,y)
title('Matlab: ode23s')
nexttile
plot(a,b)
title('Python SciPy: solve-ivp')
nexttile
plot(t,y)
title('Solución Analítica')
```

Podemos visualizar las gráficas de cada uno de los solvers y compararlas con la solución analítica

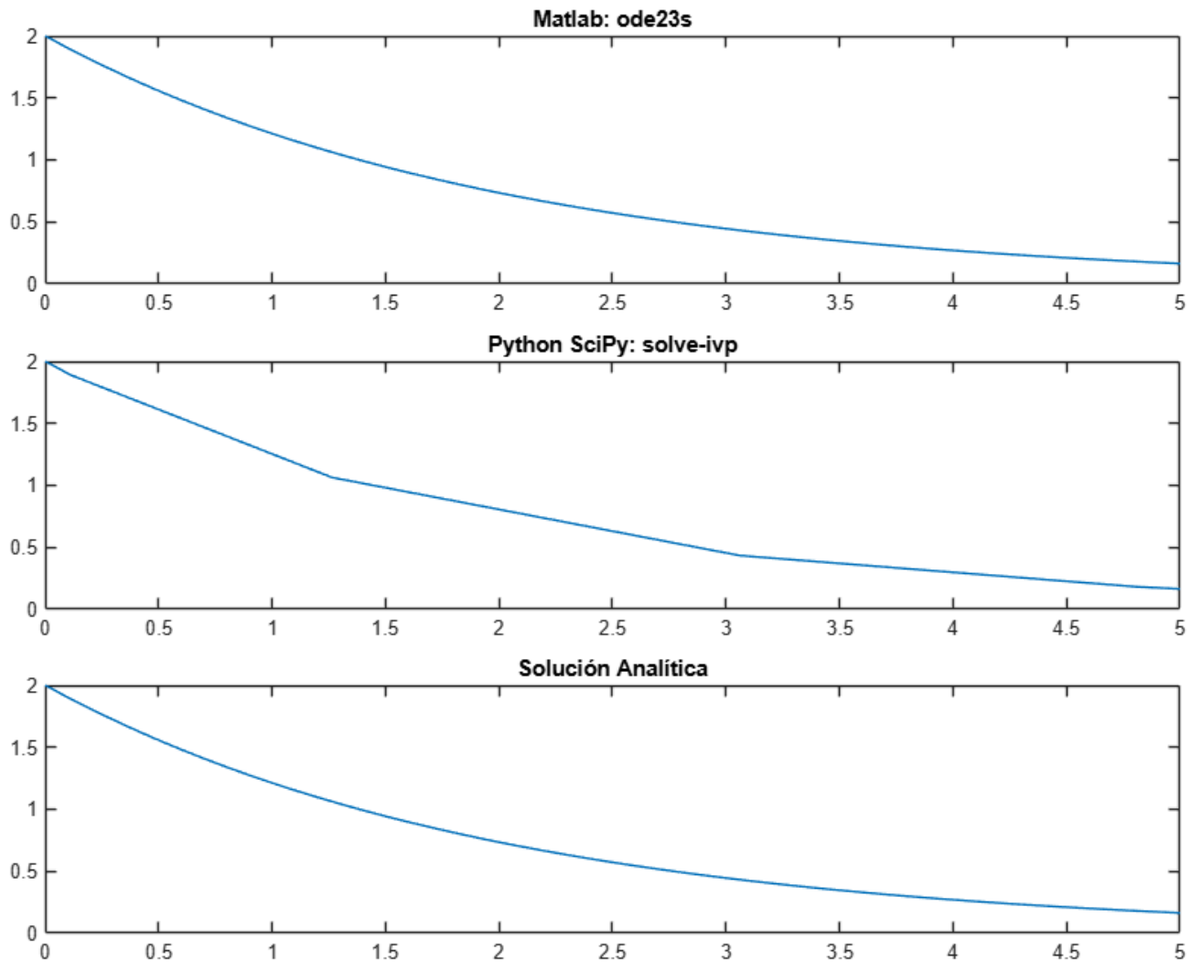


Figure 1. Matlab: ode23s VS Python SciPy: solve-ivp VS Solución Analítica.

Recursos

[1] Lawrence F. Shampine, Mark W. Reichelt, The MATLAB ODE Suite. SIAM J. Sci. Comput. (1997) https://api.semanticscholar.org/CorpusID:14004171

[2] Lang, J., Verwer, J. ROS3P—An Accurate Third-Order Rosenbrock Solver Designed for Parabolic Problems. BIT Numerical Mathematics 41, 731–738 (2001). https://doi.org/10.1023/A:1021900219772