El ejemplo de ecuación diferencial que usamos en la clase pasada es muy sencillo, pero lo empleamos porque necesitábamos un ejemplo donde conociéramos la solución exacta para calcular el error real del cálculo numérico. Esta clase está dividida en dos partes:

1. En la primera parte, veremos un método que es mucho más eficiente que Euler y que además es uno de los más usados en la vida real.
2. Luego en la segunda parte, resolveremos el sistema de Lorenz que vimos en clases pasadas el cual representa un modelo de la convección. También discutiremos porque la convección es un claro ejemplo de caos a la luz de una definición apropiada que daremos en esta sección.

Métodos predictor-corrector

El método de Euler es la forma más sencilla de resolver numéricamente una ecuación diferencial de la forma:

$$y' = f(t, y)$$

Donde recordemos que se ejecuta con la fórmula iterativa:

$$y_{n+1} = y_n + \Delta t \times f(t_n, y_n)$$

A partir de este método surgen varias mejoras, incluyendo los métodos predictor-corrector que ejecutan un paso inicial “predictor” y posteriormente ejecutan pasos adicionales “corrector” o correctivos. En este curso no profundizaremos en la justificación detallada detrás de estos métodos, solamente haremos una comparación con uno de los métodos más eficientes que es el Runge-Kutta de orden 4, cuya fórmula iterativa está dada en cuatro pasos así:

$$k_1 = \Delta t \times f(t_n,\ y_n)$$
$$k_2 = \Delta t \times f(t_n + 0.5\Delta t,\ y_n + 0.5k_1)$$
$$k_3 = \Delta t \times f(t_n + 0.5\Delta t,\ y_n + 0.5k_2)$$
$$k_4 = \Delta t \times f(t_n + \Delta t,\ y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6} \times (k_1 + 2k_2 + 2k_3 + k_4)$$

Haremos una comparación entre los métodos Runge-Kutta de orden 4 o RK4 y el método de Euler para ver cómo mejora la precisión numérica de un método a otro. Para ello implementaremos funciones que ejecuten paso a paso ambos algoritmos de una forma diferente a lo que hemos hecho antes (esto con el fin de ser más transparentes a la hora de comparar con las fórmulas matemáticas):

$$y' = f(t, y)$$

$$k_1 = \Delta t \times f(t_n,\ y_n)$$
$$k_2 = \Delta t \times f(t_n + 0.5\Delta t,\ y_n + 0.5k_1)$$
$$k_3 = \Delta t \times f(t_n + 0.5\Delta t,\ y_n + 0.5k_2)$$
$$k_4 = \Delta t \times f(t_n + \Delta t,\ y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6} \times (k_1 + 2k_2 + 2k_3 + k_4)$$

$$y_{n+1} = y_n + \Delta t \times f(t_n,\ y_n)$$

```python
import numpy as np
import matplotlib.pyplot as plt

def f(t, y): return y

def rk4(t0, y0, dt, f):
    k1 = f(t0, y0)
    k2 = f(t0 + dt/2.0, y0 + dt * k1 / 2.0)
    k3 = f(t0 + dt/2.0, y0 + dt * k2 / 2.0)
    k4 = f(t0 + dt, y0 + dt * k3)
    y = y0 + (dt/6.0)*(k1 + 2.0*k2 + 2.0*k3 + k4)
    return y

def euler(t0, y0, dt, f):
    f0 = f(t0, y0)
    y = y0 + dt*f0
    return y
```

Observa que en el código:

1. Dejé por fuera de todos los pasos “predictor-corrector” el Δt, y lo puse al final en la fórmula iterativa que calcula $\mathbf{y_{n+1}}$. Matemáticamente son cosas equivalentes en virtud de la propiedad de factorización algebraica, así que el resultado es el mismo.
2. Lo que en las fórmulas se llama $\mathbf{y_n}$, en el código es $\mathbf{y_0}$. Así mismo $\mathbf{y_n}$ se corresponde con **y**.
3. Cada función ejecuta un solo paso de cada algoritmo, lo que quiere decir que debemos usarlas dentro de una estructura tipo for() o while() para ejecutar varios pasos sucesivamente.

Ahora, para comparar ambos métodos tomaremos como ejemplo el caso de la clase pasada donde ***f (t, y) = y*** cuya solución con el valor inicial ***y(0) = 1*** es:

$$y(t) = e^t$$

Vamos a calcular la solución numérica por ambos métodos, para varios intervalos de tiempo y compararemos con la solución exacta, así:

$y(t) = e^t$

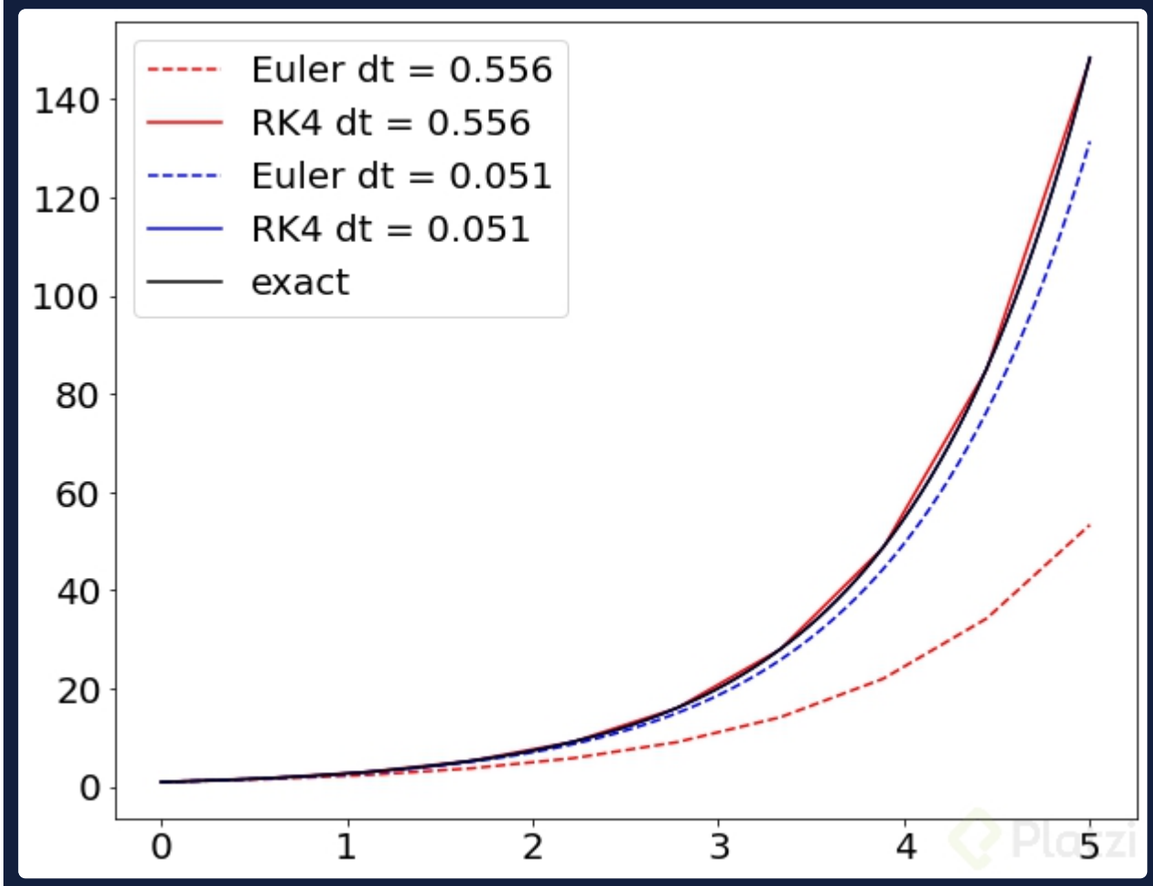
```python
def exact_sol(t): return np.exp(t)

tmax = 5
plt.figure(figsize=(10, 8))
for n, c in zip([10,100], ['red', 'blue']):
    ys_rk4 = [1]
    ys_euler = [1]
    ts = np.linspace(0,tmax,n)
    dt = ts[1]-ts[0]
    for i in range(n-1):
        u_rk4 = rk4(ts[i], ys_rk4[i], dt, f)
        u_euler = euler(ts[i], ys_euler[i], dt, f)
        ys_rk4.append(u_rk4)
        ys_euler.append(u_euler)
    plt.plot(ts, ys_euler, '--', color = c, label = 'Euler dt = {}'.format(round(dt,3)))
    plt.plot(ts, ys_rk4, color=c, label = 'RK4 dt = {}'.format(round(dt, 3)))
plot_text_size = 20
plt.plot(ts, exact_sol(ts), '-k', label = 'exact')
plt.xticks(fontsize=plot_text_size)
plt.yticks(fontsize=plot_text_size)
plt.legend(fontsize = plot_text_size)
plt.show()
```

Aquí escogemos el número de pasos y de esto se calcula Δt

Dentro del ciclo *for()* ejecutamos paso a paso RK4 y Euler

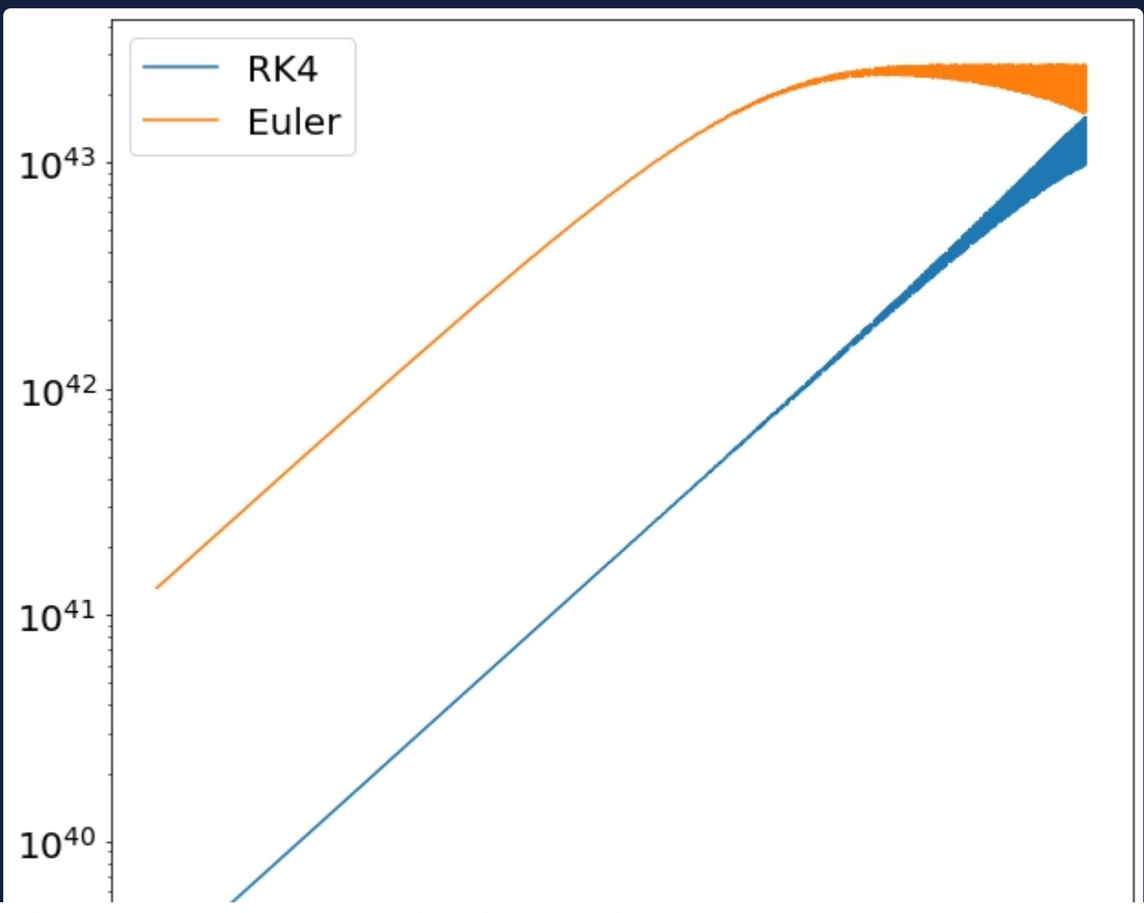
El final del código son líneas que tienen que ver solamente con la visualización de las soluciones numéricas y la exacta, cuyo resultado se ve así:

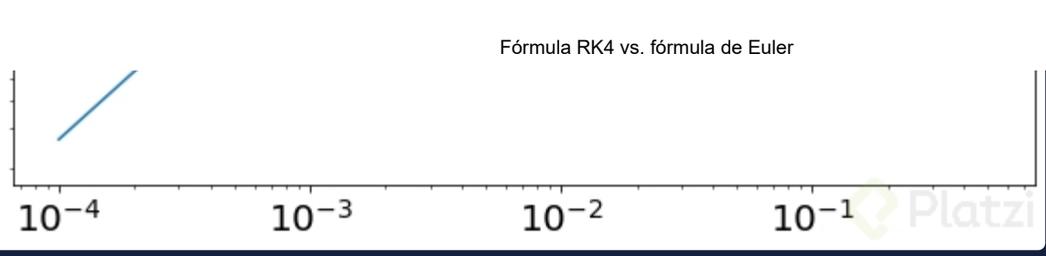


Y de esta visualización notamos inmediatamente que para un mismo Δt el rendimiento de RK4 es superior al método de Euler (compara las curvas en color rojo que representan ambas soluciones para $\Delta t = 0.556$. Así como hicimos en la clase pasada aquí también podemos calcular el error acumulado, ahora para ambos métodos dándonos como resultado lo siguiente:

```
tmax = 100
dt_arr = np.arange(0.0001, 0.5, 0.0001)
error_rk4 = []
error_euler = []
for dt in dt_arr:
    ys_rk4 = [1]
    ys_euler = [1]
    ts = np.arange(0,tmax,dt)
    for i in range(len(ts)):
        u_rk4 = rk4(ts[i], ys_rk4[i], dt, f)
        u_euler = euler(ts[i], ys_euler[i], dt, f)
        ys_rk4.append(u_rk4)
        ys_euler.append(u_euler)
    ys_exact = exact_sol(ts)
    error_rk4.append(np.abs(ys_rk4[-1]-ys_exact[-1]))
    error_euler.append(np.abs(ys_euler[-1]-ys_exact[-1]))

plt.figure(figsize=(10, 8))
plot_text_size = 20
plt.plot(dt_arr, error_rk4, label = 'RK4')
plt.plot(dt_arr, error_euler, label = 'Euler')
plt.plot(dt_arr[:-4900], error_rk4[:-4900], '-*r')
plt.xticks(fontsize=plot_text_size)
plt.yticks(fontsize=plot_text_size)
plt.legend(fontsize = plot_text_size)
plt.yscale('log')
plt.xscale('log')
plt.show()
```





El gráfico resultante nos muestra claramente que el error para RK4, es al menos dos órdenes de magnitud más pequeño que el de Euler, y esto refleja la evidente superioridad de RK4 incluso para intervalos Δt relativamente grandes.

En nuestra próxima clase, veremos cómo usar RK4 para resolver sistemas complejos que presentan comportamiento caótico y el notebook de esta clases lo encuentras en este link.