

Contenido bajo Creative Commons Attribution license CC-BY 4.0. (c) 2015 O. Skurtys, C. Cooper.

Metodo de Euler implicito

Ya vimos en el notebook anterior que existen límites en el valor de Δt para cumplir con la estabilidad. En este caso, el valor de α era bajo, por lo que se cumplía con la estabilidad para valores relativamente alto de Δt , sin embargo, si α hubiese sido más grande, podría haber ocurrido que el valor de Δt tendría que haber sido muy pequeño para cumplir con la estabilidad, y necesitaríamos muchos pasos de tiempo para llegar al resultado final.

Por otra parte, sabemos que los métodos implícitos no tienen esta limitación, por lo que podemos llegar al resultado final en menos pasos de tiempo. Revisemos el método de Euler implícito en detalle.

Cuando decimos que el método de Euler es *explícito*, nos referimos a que la función a la derecha de la ecuación es evaluada en el tiempo t_n , por lo que es trivial despejar el valor de la función en t_{n+1} . El caso *implícito* es lo contrario: la función es evaluada en el tiempo t_{n+1} . De esta forma, al discretizar en el tiempo, quedamos con la siguiente ecuación

$$\begin{cases} y_0 & \text{dado} \\ y_{n+1} = y_n + \Delta t f(y_{n+1}, t_{n+1}) & \forall n \geq 0 \end{cases}$$

¡Vemos que para calcular y_{n+1} debemos conocer y_{n+1} ! Si la función $f(y, t)$ es no lineal, despejar y_{n+1} deja de ser trivial, y quedamos con una ecuación implícita.

Euler implícito en caída libre

Veamos el caso no lineal del notebook anterior (la caída libre de Luke Aikins). Discretizando la ecuación para la velocidad, quedamos con

$$v_{n+1} = v_n + \Delta t \left(-\frac{\alpha}{m} v_{n+1}^2 + g \right),$$

y despejando lo desconocido a la izquierda y lo conocido a la derecha, quedamos con

$$v_{n+1} + \frac{\Delta t \alpha}{m} v_{n+1}^2 = v_n + \Delta t g$$

Esta es una ecuación de segundo orden, que todavía debiésemos ser capaces de resolver explícitamente, sin embargo, en el caso general, puede que no seamos capaces de despejar v_{n+1} . Afortunadamente, existen algoritmos para calcular las raíces de una ecuación numéricamente disponibles en Python (algo así como cuando escriben `solve` en la TI o en [Wolfram Alpha](https://www.wolframalpha.com/) (<https://www.wolframalpha.com/>)), y así obtener el valor de v_{n+1} . Por lo tanto, tendrán que hacer un `solve` en cada paso de tiempo.

Un ejemplo de esto es la función `fsolve` (<http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.fsolve.html>), que se encuentra en la librería `scipy`:

```
In [1]: from scipy.optimize import fsolve
```

La función `fsolve` encuentra las raíces de una ecuación de tipo $f(x) = 0$. Usemos un ejemplo cuadrático (igual al que necesitarán para el caso de Luke Aikins); para definir $f(x)$, podemos hacer

```
In [2]: def f(x, a, b, c):
        return a*x**2+b*x+c
```

Resolvamos la ecuación $2x^2 + 3x - 4 = 0$:

```
In [4]: x0 = 0
        a=2
        b=3
        c=-4
        x_root = fsolve(f,x0,args=(a,b,c))
        print(x_root)

[0.85078106]
```

Donde `x0` es una estimación del resultado final (el valor con que el método iterativo comienza a buscar la raíz). Si estamos bien, $f(x_{root}) = 0$; probemos:

```
In [6]: print(f(x_root,a,b,c))

[2.04281037e-14]
```

A resolver!

Resuelvan el mismo problema del notebook anterior, usando un método de Euler implícito. Ayúdense de la función `scipy.optimize.fsolve` para resolver la ecuación no lineal en cada paso de tiempo. Es más, pueden usar la función $f(x,a,b,c)$ definida arriba ¿pueden ustedes la ecuación discretizada con Euler implícito de la forma $ax^2 + bx + c$?

Ojo que la discretización de la ecuación para la altura también debe ser implícita!

Grafiquen la velocidad y altura respecto del tiempo, y prueben valores altos de Δt (incluso $\Delta t > 8$!) ¿Ven resultados similares que en el caso explícito? ¿Por qué? ¿Qué opinen de la exactitud de los resultados usando Δt alto?

Para pensar

- Si usáramos el modelo lineal descrito al principio del notebook anterior, ¿necesitaríamos `fsolve`? ¿Por qué?
- Vemos que todo se reduce a resolver una ecuación cuadrática en cada paso de tiempo ¿Podríamos despejar v_{n+1} de alguna manera? No es necesario que lo implemente.

Se nota que la ecuación queda:

$$\frac{\Delta t \alpha}{m} v_{n+1}^2 + v_{n+1} - v_n - \Delta t g = 0$$

De donde en cada paso temporal hay que resolver para v_{n+1}

```
In [17]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

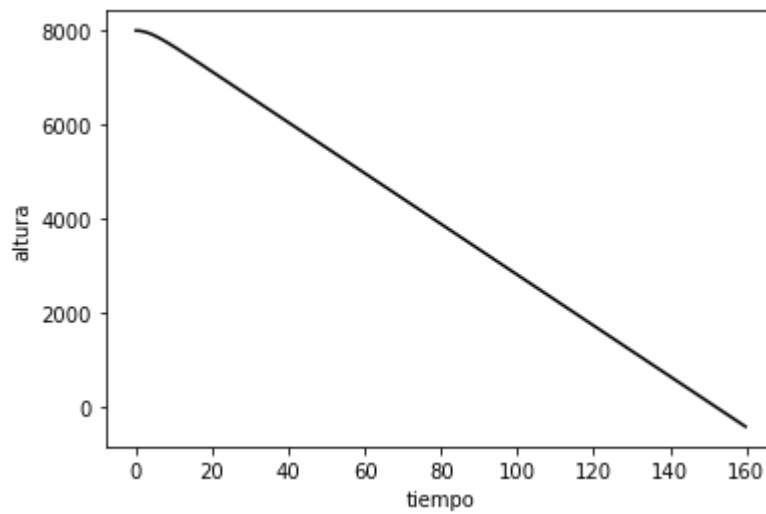
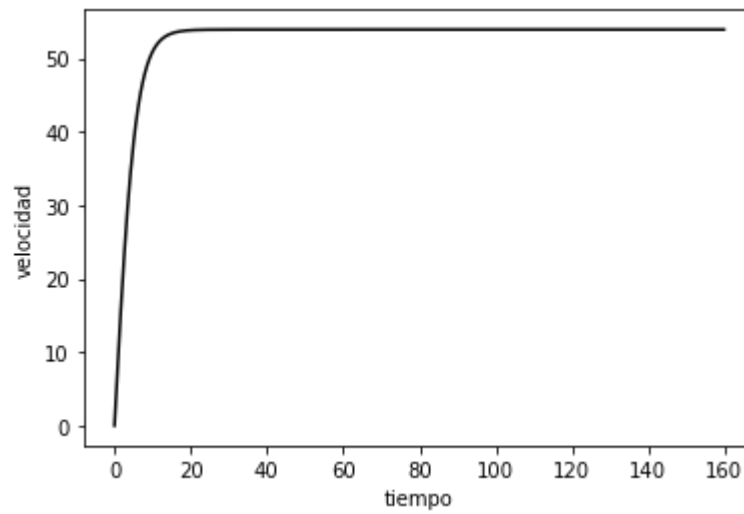
### ALUMNO
import numpy as np

def caida_libre_implicito(v0, h0, dt, tf, alpha, m, g):
    t0 = 0
    n = int((tf-t0)/dt)
    tf = n*dt
    t = np.arange(t0,tf,dt)
    v = np.zeros(n)
    v[0] = v0
    y = np.zeros(n)
    y[0] = h0

    for i in range(n-1):
        t[i+1] = t[i] + dt
        v[i+1] = fsolve(f,v[i],args=(dt*alpha/m,1,-v[i]-dt*g))
        y[i+1] = y[i] + -dt*v[i+1]
    return v,y,t

v0 = 0.
h0 = 8000.
tf = 160.
dt = 0.2
alpha = 0.27
m = 80
g = 9.81
v,h,time = caida_libre_implicito(v0, h0, dt, tf, alpha, m,g)

plt.plot(time, v, c='k')
plt.ylabel("velocidad")
plt.xlabel("tiempo")
plt.show()
plt.clf()
plt.plot(time, h, c='k')
plt.ylabel("altura")
plt.xlabel("tiempo")
plt.show()
```

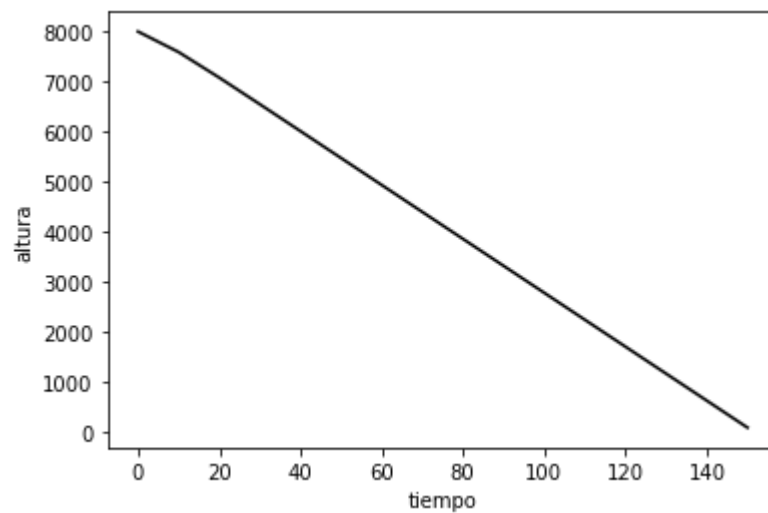
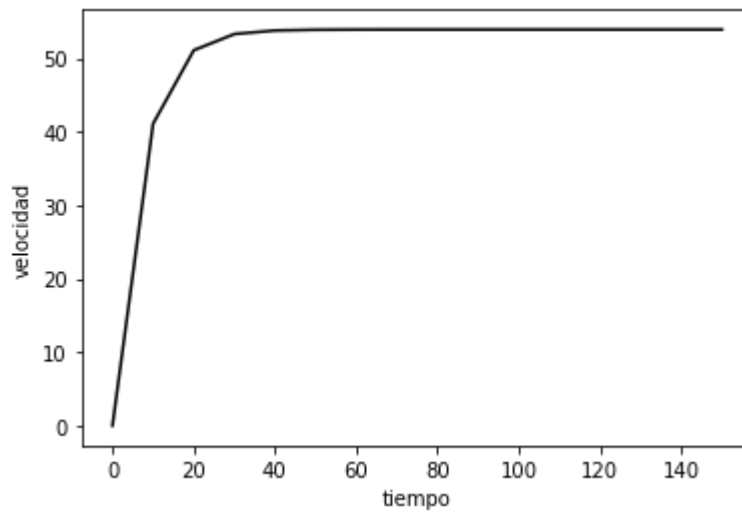


Se nota claramente que los resultados coinciden con los del método explícito.

```
In [22]: dt = 10

v,h,time = caida_libre_implicito(v0, h0, dt, tf, alpha, m,g)

plt.plot(time, v, c='k')
plt.ylabel("velocidad")
plt.xlabel("tiempo")
plt.show()
plt.clf()
plt.plot(time, h, c='k')
plt.ylabel("altura")
plt.xlabel("tiempo")
plt.show()
```



Se nota claramente que los resultados mejoran a la vista con el método implícito. Además, como se dijo antes, con pasos temporales altos el algoritmo no se vuelve inestable. Esto puede traer la ventaja de probar códigos antes de implementarlos con mallas más finas para mayor precisión.

Si se hubiera usado el modelo lineal del notebook anterior, no hubiera sido necesario usar fsolve dado que se podría implementar el código con un simple despeje analítico. De todas formas, este código (arraste proporcional a la velocidad al cuadrado) también se puede implementar sin fsolve dado que existen soluciones analíticas a la ecuación cuadrática.

Metodo del trapecio

El metodo del trapecio es un metodo de orden 2 y está definido por la formula siguiente:

$$\begin{cases} y_0 & \text{dado} \\ y_{n+1} = y_n + dt \frac{f(y_{n+1}) + f(y_n)}{2} & \forall n \geq 0 \end{cases}$$

Resuelva el mismo problema usando el método del trapecio. Siga los siguientes pasos:

- Escriba la ecuación discretizada con el método del trapecio
- Reescriba esta ecuación en forma $ax^2 + bx + c$
- Resuélvala de forma similar a como resolvió el caso de Euler implícito
- Grafique los resultados

Para pensar

- ¿Qué ventaja tiene que este método sea de segundo orden?
- Computacionalmente es casi igual de pesado que Euler implícito. Si tuviesen que elegir, ¿preferirían Euler implícito o el trapecio?

Las ecuaciones discretizadas quedan:

$$\begin{aligned} v_{n+1} &= v_n + \frac{\Delta t}{2} \left(\left(-\frac{\alpha}{m} v_{n+1}^2 + g \right) + \left(-\frac{\alpha}{m} v_n^2 + g \right) \right), \\ h_{n+1} &= h_n + \frac{\Delta t}{2} (v_{n+1} + v_n), \end{aligned}$$

La ecuación cuadrática a resolver en cada paso temporal termina siendo:

$$\frac{\alpha \Delta t}{2m} v_{n+1}^2 + v_{n+1} - v_n + \frac{\alpha \Delta t}{2m} v_n^2 - \Delta t g = 0$$

```

In [27]: ### ALUMNO

def caida_libre_trapecio(v0, h0, dt, tf, alpha, m, g):
    t0 = 0
    n = int((tf-t0)/dt)
    tf = n*dt
    t = np.arange(t0,tf,dt)
    v = np.zeros(n)
    v[0] = v0
    y = np.zeros(n)
    y[0] = h0

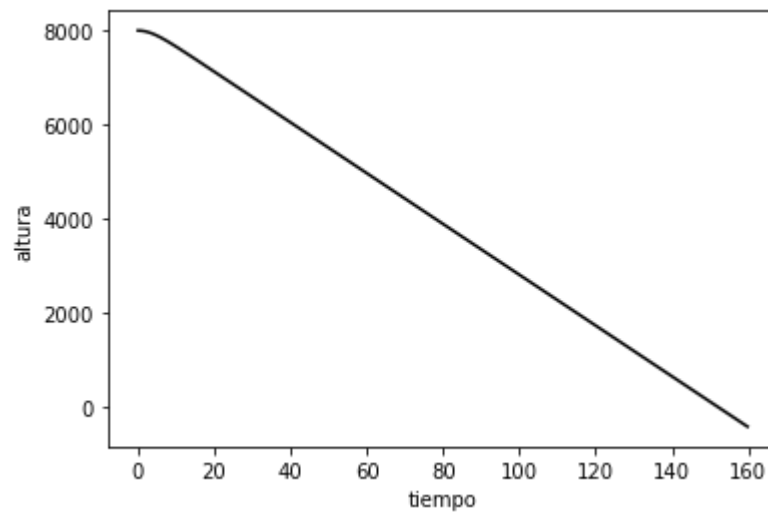
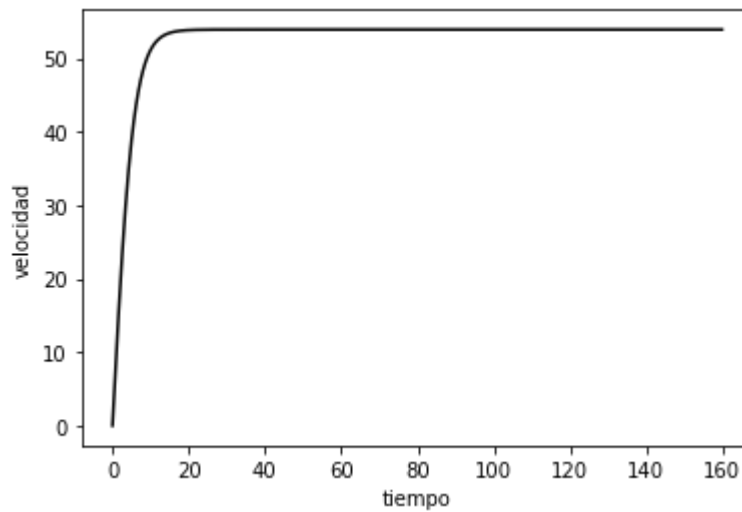
    for i in range(n-1):
        t[i+1] = t[i] + dt
        v[i+1] = fsolve(f,v[i],args=(dt*alpha/(2*m),1,-v[i]+((alpha*dt)/(2*m))
*v[i]**2-dt*g))
        y[i+1] = y[i] + -(dt/2)*(v[i+1]+v[i])
    return v,y,t

v0 = 0.
h0 = 8000.
tf = 160.
dt = 0.2
alpha = 0.27
m = 80
g = 9.81

v,h,time = caida_libre_trapecio(v0, h0, dt, tf, alpha, m,g)

plt.plot(time, v, c='k')
plt.ylabel("velocidad")
plt.xlabel("tiempo")
plt.show()
plt.clf()
plt.plot(time, h, c='k')
plt.ylabel("altura")
plt.xlabel("tiempo")
plt.show()

```

Se nota que los resultados a simple vista se ven idénticos a los del Euler implícito. La gran ventaja que tiene este método en comparación al Euler implícito es que es de orden 2, en comparación al anterior que es de orden 1. Esto implica que el error de truncación cae más rápido al refinar la malla.

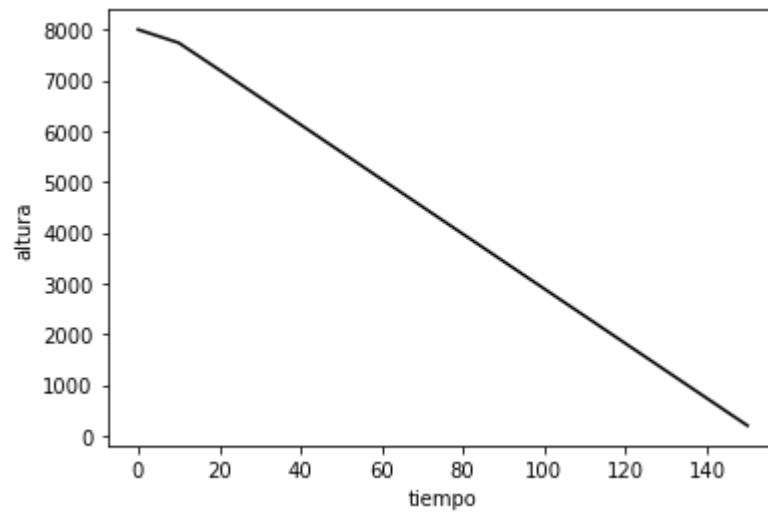
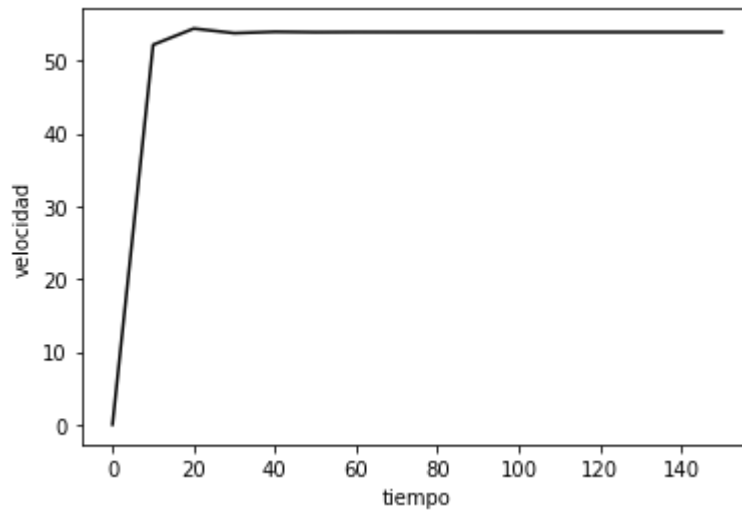
Si ambos usan la misma capacidad computacional o similar, yo me quedaría con el método trapezoidal ya que es un método de mayor orden. De todas formas, esta respuesta es asumiendo que los pasos temporales son pequeños para evitar inestabilidades. En los siguientes gráficos se ilustra el comportamiento al realizar pasos muy grandes.

En conclusión, para una prueba rápida a pasos grandes, "cálculo de servilleta", usaría el implícito. Teniendo ya un comportamiento aproximado o probable, probaría con una malla más fina con un método de mayor orden como lo es el del trapecio o RK4.

```
In [29]: dt = 10

v,h,time = caida_libre_trapecio(v0, h0, dt, tf, alpha, m,g)

plt.plot(time, v, c='k')
plt.ylabel("velocidad")
plt.xlabel("tiempo")
plt.show()
plt.clf()
plt.plot(time, h, c='k')
plt.ylabel("altura")
plt.xlabel("tiempo")
plt.show()
```



In []: