Solución numérica de ecuaciones diferenciales ordinarias

¡Bienvenidos al tercer laboratorio! Hoy vamos a resolver ecuaciones diferenciales ordinarias (EDO) numéricamente utilizando métodos que probablemente han escuchado antes, como Euler o Runge-Kutta.

Hagamos algo de memoria. Una EDO es una ecuación donde hay una derivada de una función que depende de una sola variable. Por ejemplo, en la ecuación

$$\frac{dy}{dt} = f(t, y(t))$$

la función y depende solamente de la variable t. La derivada en este caso es de primer orden, pero podría ser de orden superior.

En este laboratorio vamos a discretizar una ecuación de este tipo, para resolverla numéricamente.

Trabajo preliminar

Antes resolver una ecuación diferencial ordinaria (EDO) en el contexto de un problema físico, es aconsejable simplificar las ecuaciones y definir un nuevo sistema de unidades adaptados al problema. En efecto, para minimizar los errores de redondeo, es bueno que los valores numericos utilizados sean del orden de la unidad.

Por ejemplo, para tratar el problema de movimiento de 2 astros ligados por la gravitación, es preferable adoptar unidades astronomicas (tiempo en año, masas en masas solares, distancia en distancia tierra-sol) que en el sistema internacional (segundo, kilogramo, metro).

Cambiar las unidades, finalmente, es definir nuevas magnitud adimensionales. Se buscará siempre a reescribir la EDO usando magnitudes sin dimensión.

Un ejemplo

Tomamos por ejemplo la caida de una esfera en un fluido. Se supone que la resitencia que recibe la esfera proporcional a su velocidad: $F_{arrastre} \propto v$. La relación fundamental de la dinámica permite escribir la ecuación diferencial ordinaria siguiente:

$$m\frac{dv}{dt} = -\alpha v + mg \quad \text{con} \quad v(0) = 0$$

donde m es la masa, α es el coeficiente de fricción (unidades de masa dividido por tiempo) y g es el campo de gravedad. Se puede definir un tiempo caracteristico (tiempo de relaxación): $\tau = \frac{m}{\alpha}$; una velocidad caracteristica (velocidad limite): $v_\infty = g\tau$. Así, podemos adoptar un nuevo sistema de unidad:

- El tiempo se medido en unidad de τ, donde la nueva variable temporal es $t^* = \frac{t}{\tau}$.
- La velocidad es medida en unidad de v_∞, donde la nueva variable es $v^* = \frac{v}{v_\infty}$.
- La distancia es el producto de una velocidad por un tiempo, es medida en unidad de h con $h = v_\infty\tau$.

Con este nuevo sistema de unidad y de variables adimensionales, la ecuación diferencial inicial toma la forma más simple:

$$\frac{dv^*}{dt^*} = 1 - v^* \quad \text{con} \quad v^*(0) = 0$$

Esta ecuación diferencial puede se obtener si hacemos $\tau = 1$ y $v_\infty = 1$. Finalmente, cuando fijamos unitario los parametros fisicos, definimos un nuevo conjunto de unidad.

Método de Euler explicito

El metodo de Euler explicito es el metodo el más simple a programar y a entender. Consideramos la ecuación diferencial siguiente:

$$\begin{aligned} \frac{dy}{dt} = \dot{y} &= f(t, y(t)) \qquad 0 \leq t \leq T \\ y(0) &= 0 \end{aligned}$$

Se puede integrar la ecuación:

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))\, dt$$

El metodo de Euler consiste a aproximar la integral por el metodo de los rectangulos (a la izquierza):

$$\int_{t_n}^{t_{n+1}} f(t, y(t))\, dt \approx dt \times f(t_n, y(t_n))$$

Así, el esquema iterativo es el siguiente:

$$\begin{aligned} y_{n+1} &= y_n + dt f(t_n, y(t_n)) \qquad n = 0, \dots, N \\ y(0) &= 0 \end{aligned}$$

donde y_n es la aproximación numerica de $y(t_n)$

La programación es muy simple. El algorithmo de Euler es:

1. Definir el paso dt, de la duración T.
2. Inicialización de las condiciones iniciales: $t = 0$ y $y = y(0)$.
3. Mientras que $t \leq T$:
   A. Calculo de $k_1 = f(t, y)$
   B. $y = y + dt$; k_1; $t_{new} = t + dt$
   C. Guardar los datos

Ejemplo

Aplicamos el metodo del problema de caida libre visto anteriormente. La ecuación diferencial ordinaria es:

$$\frac{dv^*}{dt^*} = 1 - v^* \qquad \text{con} \qquad v^*(0) = 0$$

y el esquema iterativo se escribe:

$$\begin{aligned} v^*_{n+1} &= v^*_n + dt(1 - v^*_n) \\ v^*_0 &= 0 \end{aligned}$$

Se puede ver que el metodo de Euler es una aproximación de primer orden en dt. De hecho, si calculamos el primer termino v^*_1 y lo comparamos con la solución exacta $v^*(dt)$ (que conocemos: $v^*(t^*) = 1 - e^{-t^*}$), tenemos:

$$\begin{aligned} v^*_1 &= dt \\ v^*(dt) &= 1 - e^{-dt} \end{aligned}$$

La expansión de e^{-dt} hasta el término de primer orden es:

$$e^{-dt} = 1 - dt + O(dt^2)$$

y reordenando queda

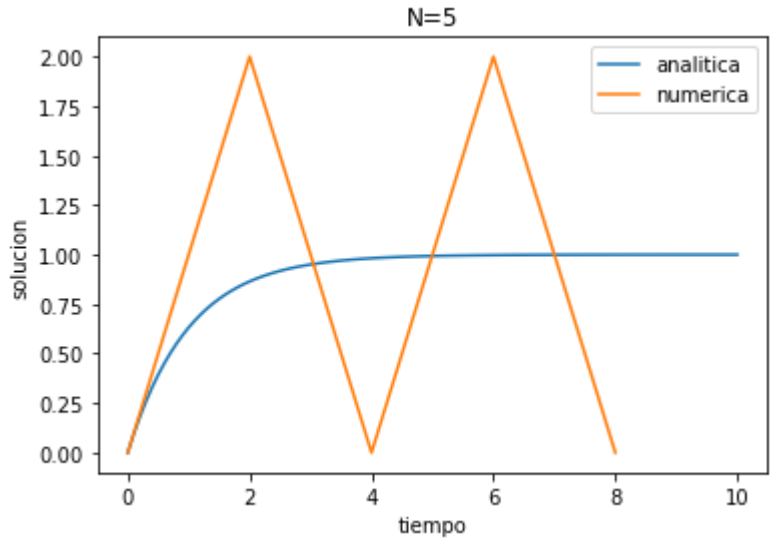
$$dt = 1 - e^{-dt} + O(dt^2)$$

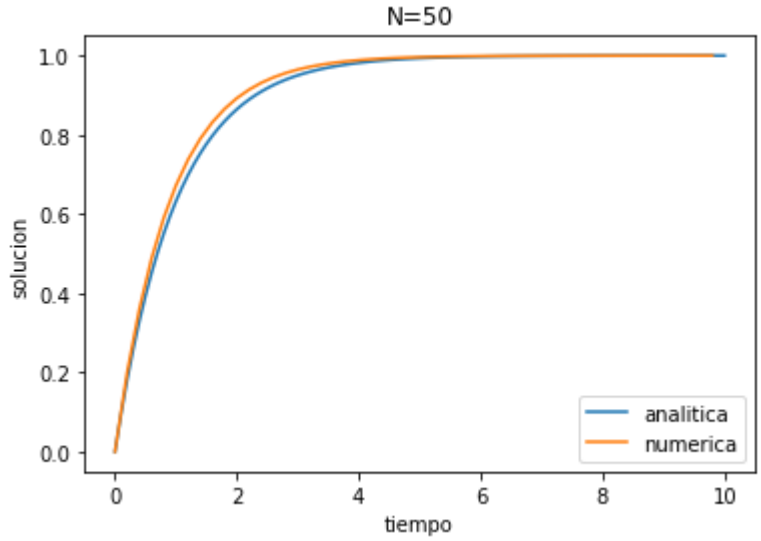
haciendo evidente que dt es una aproximación de primer orden.

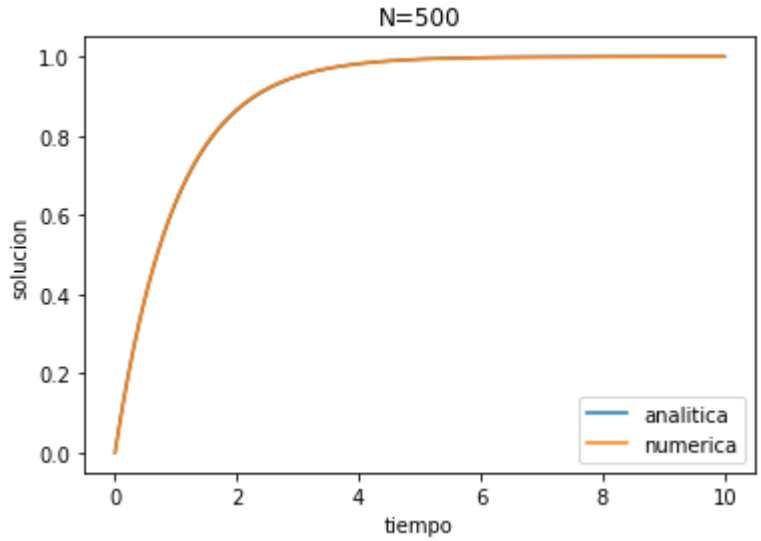
Las figuras abajo muestran el comportamiento de la solución numerica para diferentes pasos de tiempo dt. Prueben con `nt=5` $(dt = 2)$, `nt=50` $(dt = 0,2)$, y `nt=500` $(dt = 0,02)$ ¿Qué pasa en cada caso?

In [10]:
```
import numpy
def Euler(f,t0,tf,y0,n):
    """
    Ecuación y'=f(y(t),t), condición inicial (t0,y0),
    valor terminal tf, n es el número de etapa
    """
    h=(tf-t0)/float(n)
    t = numpy.arange(t0,tf,h) # h es el paso de tiempo dt
    y = numpy.zeros(n)
    y[0] = y0
    for i in range(n-1):
        t[i+1] = t[i] + h
        y[i+1] = y[i] + h*f(t[i],y[i])
    return t,y

# Ejemplo la función 1-v
def f(t,y):
    return 1-y

# Grafico ensayar por 3 valores de h es decir tomando  n=50 ; n=500 y n=5
import matplotlib.pyplot as plt
%matplotlib inline

t_final   = 10
t_inicial = 0
y_inicial = 0
nt        = 5

t,y = Euler(f, t_inicial, t_final, y_inicial, nt)
tsol = numpy.linspace(0,10,101) # dibuja la solucion analitica

plt.figure()
plt.plot(tsol,1-numpy.exp(-tsol), label='analitica')
plt.plot(t,y, label='numerica') # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=5");


nt        = 50
t,y = Euler(f, t_inicial, t_final, y_inicial, nt)

plt.figure()
plt.plot(tsol,1-numpy.exp(-tsol), label='analitica')
plt.plot(t,y, label='numerica') # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=50");

nt        = 500
t,y = Euler(f, t_inicial, t_final, y_inicial, nt)

plt.figure()
plt.plot(tsol,1-numpy.exp(-tsol), label='analitica')
```

```
plt.plot(t,y, label='numerica') # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=500");
```







Claramente se observa que para $nt = 5$, lo que corresponde a un $dt = 2$, el método explícito se vuelve inestable y no converge a la solución analítica.

Si h=0,2 se observa un error $\varepsilon = |y_n - y(t_n)|$ de algunos %, y si $h \ll 1$ el error no es significativo.

En el caso $h = 2$ el paso de tiempo es más grande que el tiempo caracteristico $\tau = 1$, y aparece un fenomeno de inestabilidad numérica, que es un inconvemiente del metodo de Euler.

Ventajas e inconvenientes del método

El método de Euler tiene la ventaja de ser simple de programar y produce en varios casos una solución aceptable como primera aproximación al fenomeno estudiado.

Por otra parte, tiene el inconveniente de propagar fácilmente los errores y de amplificarlos. Además, es inestable en ciertas circunstancias.

Metodo de Runge Kutta de orden 4

El metodo de Runge-Kutta de orden 4 (RK4) es muy utilizado porque es más preciso y estable que el metodo de Euler. En efecto, este método tiene una estimación más precisa de la integral de la EDO.

El esquema numérico de RK4 es

$$\begin{cases} k_1^n = f(t_n, y_n) \\ k_2^n = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1^n}{2}\right) \\ k_3^n = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2^n}{2}\right) \\ k_4^n = f\left(t_{n+1}, y_n + hk_3^n\right) \\ y_0 = y(0) \\ \\ y_{n+1} = y_n + \frac{h}{6}\left(k_1^n + 2k_2^n + 2k_3^n + k_4^n\right) \qquad n = 0, 1, \dots, N \end{cases}$$

Donde h es el paso de tiempo dt. Se ve inmediatamente un defecto respecto al metodo de Euler: el algorithmo exige 4 veces más cálculos en cada paso y entonces es más lento. Además, los errores de redondeos que se acumulan más rapidamente. Sin embargo, estos defectos son compensados por una mejor precisión.

Algorithmo Runge-Kutta orden 4

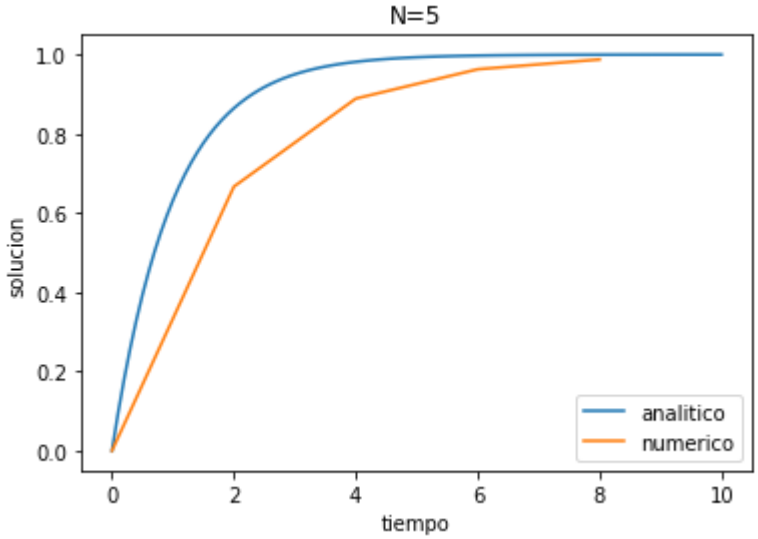
1. Inicialización del paso h y la duración T.
2. Inicialización de las condiciones iniciales: $t = 0$ y $y = y(0)$.
3. Mientras que $t \leq T$:
   A. Calculo de $k_1 = f(t, y)$
   B. Calculo de $k_2 = f\left(t + \frac{h}{2}, y + h\frac{k_1}{2}\right)$
   C. Calculo de $k_3 = f\left(t + \frac{h}{2}, y + h\frac{k_2}{2}\right)$
   D. Calculo de $k_4 = f\left(t_{n+1}, y + hk_3\right)$
   E. $y = y + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$; $t = t + h$
   F. Guardar los datos

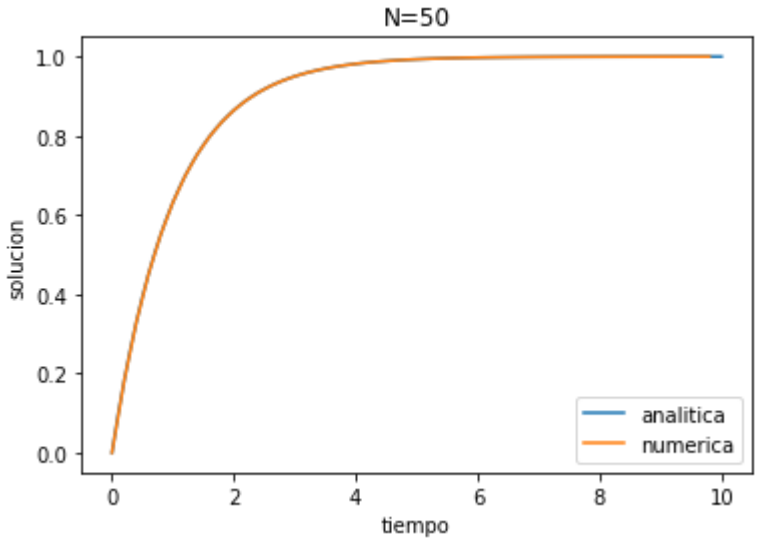
Ejemplo

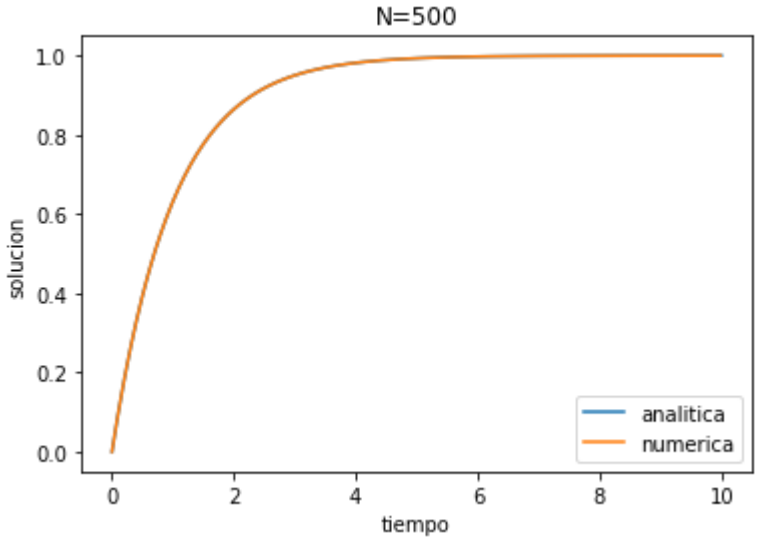
Comparemos el método de RK4 con el método de Euler para el ejemplo de la esfera en caída libre. Usen los mismos valores de `t_inicial`, `t_final` y `y_inicial` que en el caso anterior, y prueben con `nt = 5, 50, 500`.

In [13]:
```
#### ALUMNO

def RungeKutta4(f,t0,tf,y0,n):

    h=(tf-t0)/float(n)
    t = numpy.arange(t0,tf,h) # h es el paso de tiempo dt
    y = numpy.zeros(n)
    k1 = numpy.zeros(n)
    k2 = numpy.zeros(n)
    k3 = numpy.zeros(n)
    k4 = numpy.zeros(n)
    y[0] = y0
    for i in range(n-1):
        t[i+1] = t[i] + h

        k1[i] = f(t[i],y[i])
        k2[i] = f(t[i]+h/2,y[i]+h*k1[i]/2.0)
        k3[i] = f(t[i]+h/2,y[i]+h*k2[i]/2.0)
        k4[i] = f(t[i+1],y[i]+h*k3[i])

        y[i+1] = y[i] + (h/6.0)*(k1[i]+2*k2[i]+2*k3[i]+k4[i])
    return t,y


def f(t,y):
    return 1-y

tsol = numpy.linspace(0,10,101) # dibuja la solucion analitica

t_final   = 10
t_inicial = 0
y_inicial = 0
nt        = 5

t,y = RungeKutta4(f,t_inicial, t_final, y_inicial, nt)

# Grafico ensayar por 3 valores de h es decir tomando  n=50 ; n=500 y n=5
plt.plot(tsol,1-numpy.exp(-tsol), label="analitico")
plt.plot(t,y, label="numerico") # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=5");


nt        = 50

t,y = RungeKutta4(f, t_inicial, t_final, y_inicial, nt)

plt.figure()
plt.plot(tsol,1-numpy.exp(-tsol), label='analitica')
plt.plot(t,y, label='numerica') # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=50");
```

```
nt        = 500

t,y = RungeKutta4(f, t_inicial, t_final, y_inicial, nt)

plt.figure()
plt.plot(tsol,1-numpy.exp(-tsol), label='analitica')
plt.plot(t,y, label='numerica') # dibuja la solucion numerica
plt.ylabel("solucion")
plt.xlabel("tiempo")
plt.legend()
plt.title("N=500");
```


N=5
solucion
tiempo
analitico
numerico


N=50
solucion
tiempo
analitica
numerica


N=500
solucion
tiempo
analitica
numerica

Pueden ver que para $h = 2$ ya no tenemos las instabilidades numericas observadas con el metodo de Euler. Sin embargo la solución obtenida con RK4 está alejada de la solución analitica. Para $h = 0, 2$, la solución numerica es ya aceptable, mostrando la superioridad del metodo RK4 sobre el metodo de Euler.

El cálculo del primer término v_1^* nos da la razón de este comportamiento:

$$v_1^* = h - \frac{1}{2}h^2 + \frac{1}{6}h^3 - \frac{1}{24}h^4$$

coincide con la expansión de orden 4 de la solución analítica de $v(h)$.

Ventaja del método

El metodo RK4 permite ganar en estabilidad y en precisión con un número de pasos pequeño, lo que lo hace un algorithmo bastante usado.

Los errores numericos

Importante: nunca usar un algoritmo a ojos cerrados, sin hacer un análisis critico de los resultados. Si Ud. busca precisión, es importante tener una idea del error producido por el algoritmo. Hay dos tipos de error: error de redondeo y error de truncamiento.

Error de redondeo

Algunas reglas para limitar los errores de redondeo:

- Usar un sistema de unidades tel que los valores usadas sean del orden de la unidad
- Evitar sustracción de numeros comparables
- Minimizar las sustracciones
- Buscar a sumar por paquete las cantidades de misma orden de magnitud.

Error de truncamiento

El truncamiento consiste a reemplazar la expansión en una serie infinita por una expansión con orden limitado. Se obtiene entonces un resultado que contiene un error de aproximación, conocido como error de truncatura.

En análisis numerico existe varias manera de medir este error: ε_T.

- Por ejemplo, se puede calcular la diferencia entre la solución exacta $y(T)$ y la solución aproximada y_N
$$\varepsilon_T = |y(T) - y_N|$$
- Se dice que el metodo es de orden p cuando $\varepsilon_T \propto \frac{1}{N^p}$.
- El metodo de Euler produce un error del orden h^2 en cada paso de tiempo, eso da finalmente un error total:
$$\varepsilon_T \approx Nh^2 = \frac{T^2}{N}$$
sea el metodo de Euler es de orden 1.
- El metodo de Runge-Kutta 4 es, como su nombre lo indica, de orden 4, entonces tenemos en cada paso de tiempo un error de truncamiento de h^5.

Comentarios

Es imposible minimizar al mismo tiempo el error de redondeo y el error de truncación. Se debe encontrar un balance aceptable para minimizar el error total y el tiempo de cálculo.

En caída libre

En realidad, la resistencia del fluido sobre un cuerpo que cae es proporcional a su velocidad al cuadrado: $F_{arrastre} \propto v^2$. La segunda ley de Newton nos permite escribir la siguiente ecuación diferencial ordinaria:

$$m\frac{dv}{dt} = -\alpha v^2 + mg$$

y además sabemos que la velocidad es la derivada temporal de la altura

$$\frac{dh}{dt} = v$$

Tomemos el ejemplo de Luke Aikins, que recientemente se tiró desde un avión a 8 mil metros de altura, sin paracaídas, sobre una malla (https://www.youtube.com/watch?v=6qF_fzEI4wU) (no lo hagan en casa...). El coeficiente de arrastre de una persona es más o menos $C_D = 0.84$, lo cual nos entrega un $\alpha = 0.27$. Considerando que Luke Aikins pesa cerca de 80kg, y que se lanza desde el avión con velocidad nula en el eje y, $v = 0$, grafiquen la velocidad y altura con respecto al tiempo, y responda:

- Llega a la velocidad terminal? A qué altura y cuantos segundos después de lanzarse?
- Cuánto demora en llegar a la malla?
- A qué velocidad llega a la malla?

No es estrictamente necesario que respondan con un valor exacto, pero una estimación gráfica es suficiente. En este caso, no se preocupen por adimensionalizar la ecuación, usen un $\Delta t = 0.2$, e integren usando Euler.

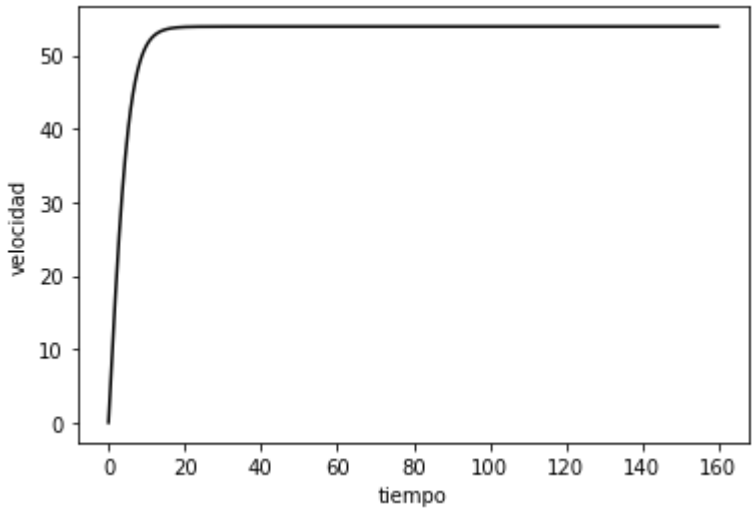
Se escribe el sistema de ecuaciones como:

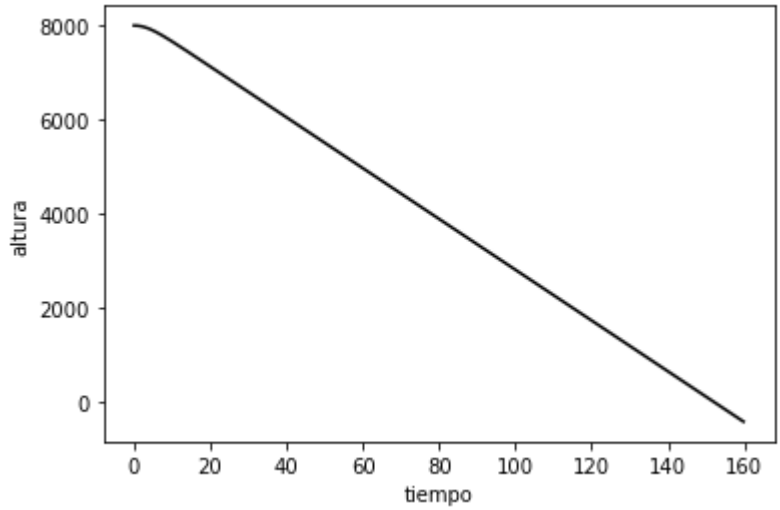
$$\frac{dv}{dt} = -\frac{\alpha v^2}{m} + g$$
$$\frac{dh}{dt} = -v$$

Se define $f1$ y $f2$

$$f1(t, v) = -\frac{\alpha v^2}{m} + g$$

In [78]:
```
### ALUMNO
import numpy as np

def caida_libre(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] = v[i] + dt*(-alpha*(v[i]**2)/m + g)
        y[i+1] = y[i] + -dt*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(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 [74]:
```
for i in range(len(v)):
    if v[i+1]-v[i]<0.0001:
        j = i
        break

print(f'Velocidad terminal: {v[j]}')
print(f'Altura cuando llega a la velocidad terminal {h[j]}')
print(f'Tiempo que demora en llegar a la vel. terminal {time[j]}')
```

```
Velocidad terminal: 53.912151782753696
Altura cuando llega a la velocidad terminal 6590.090786349529
Tiempo que demora en llegar a la vel. terminal 29.999999999999925
```

Los resultados para la velocidad terminar coinciden con los gráficos. La velociodad aproximadamente en 53 [m/s] en la altura y tiempo calculado anteriormente.

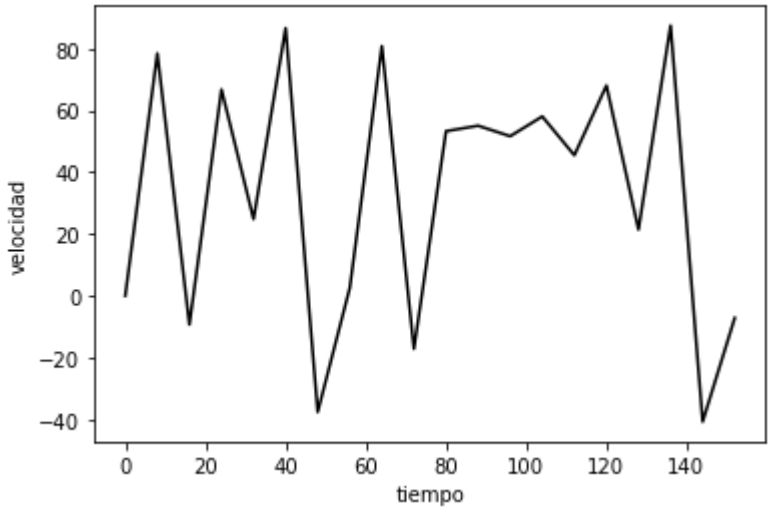
In [75]:
```
for i in range(len(h)):
    if h[i]<0.1:
        k = i
        break

print(f'Velocidad llega al suelo: {v[k]}')
print(f'Tiempo que demora en llegar al suelo {time[k]}')
```

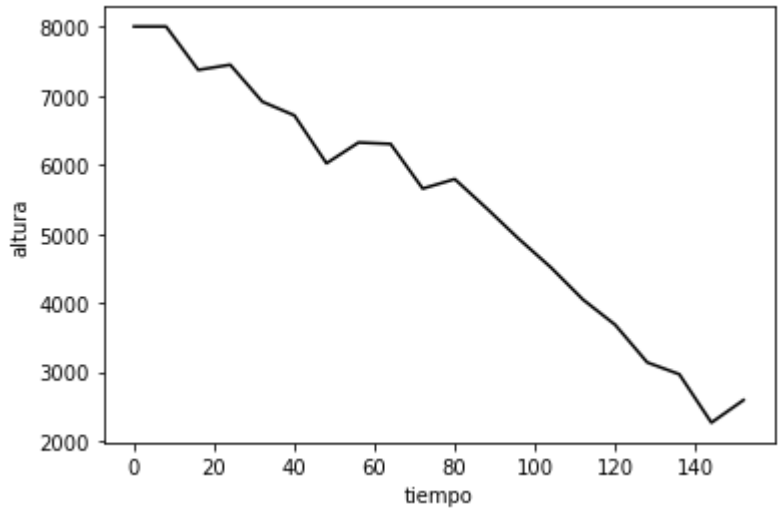
```
Velocidad llega al suelo: 53.913510984415225
Tiempo que demora en llegar al suelo 152.3999999999999
```

Se nota que llega a la velocidad terminal, lo cual tienen sentido. Además, se demora 152 segundos, lo que coincide con el gráfico

Ahora hagan los mismo gráficos con $\Delta t = 8$ ¿Pueden confiar en estos resultados? ¿Por qué se dan así?

In [76]:
```
###ALUMNO
dt = 8

v,h,time = caida_libre(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()
```





Claramente los resultados se vuelven inestables para este paso temporal dado que los errores de truncación son significativos. Es por esto que no se puede confiar en estos resultados.

In [ ]: