In [1]:

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

In [2]:

```
# El código siguiente recarga (reloads) las rutinas externas cada vez que el código cambia (es útil para "debuggear" código externo)

%load_ext autoreload
%autoreload 2
```

Método de Euler

Si nos permitimos un poco de *sloppiness* , podemos hacer lo siguiente:

$$\begin{aligned} a &\equiv \frac{dv}{dt} \\ dv &= adt \end{aligned}$$

Y reconociendo que tenemos números flotantes con precisión finita:

$$\Delta v = a\Delta t$$

Conociendo la posición inicial x_i y el cambio δx podemos estimar la nueva velocidad:

$$v = v_i + a\Delta t$$

Entonces, haciendo recursivos los *pasos*

$$v_{i+1} = v_i + \frac{dv}{dt}\Delta t$$

Y podemos hacer lo mismo para la posición:

$$x_{i+1} = x_i + \frac{dx}{dt}\Delta t$$

Una representación en imagen del método, se muestra a continuación:

Imagen de Wikipedia

Como se puede apreciar en la figura, la aproximación con el ***método de Euler*** va empeorando conforme aumentamos los pasos. Para combatir este error, se pueden disminuir el tamaño del paso, pero como veremos más adelante, esto tiene sus limitaciones.

Ejemplo: Caída libre

La ecuación de movimiento en caída libre es:

$$\ddot{x} = -g$$

donde g es la constante de aceleración de la gravedad.

El método de Euler sólo funciona con ecuaciones diferenciales de primer orden, pero podemos hacer el siguiente truco:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -g \end{aligned}$$

Por lo que ahora nuestro sistema está descrito por dos **ecuaciones lineales acopladas de primer orden**.

El **método de Euler** nos dice que la solución de estas ecuaciones es:

$$\begin{aligned} x_{i+1} &= x_i + \dot{x}\Delta t \\ v_{i+1} &= v_i + \dot{v}\Delta t \end{aligned}$$

El cual se puede escribir como

$$y_{i+1} = y_i + \dot{y}\Delta t$$

donde

$$y = \begin{bmatrix} x \\ v \end{bmatrix}$$

y

$$\dot{y} = \begin{bmatrix} v \\ -g \end{bmatrix}$$

En python definimos una función para representar este sistema

In [4]:

```
#En esta ecuacion sencilla no utilizamos sistema
def caida_libre(estado, sistema):
    g0 = estado[1]
    g1 = -9.8

    return np.array([g0, g1])

#g0 y g1 son la nueva derivada g0=v y g1=-g
#Estado son las condiciones iniciales i.e. y punto
```

In [9]:

```
#Ecuacion diferencial
#Caida libre son las derivadas
def euler(y, t, dt, derivadas):
    #y_next es y(+1)
    y_next = y + derivadas(y, t)*dt
    return y_next
```

In [10]:

```
N = 1000 # número de pasos

x0 = 0.0  # posición inicial

v0 = 0.0  # velocidad inicial

g = -9.8 # aceleración de la gravedad en la tierra

tau = 3.0 # tiempo de la simulación

dt = tau/float(N-1)  # tamaño del paso
```

In [11]:

```
time = np.linspace(0, tau, N)
```

In [12]:

```
y = np.zeros([N,2])

#Posicion y velocidad inicial
y[0,0] = x0
y[0,1] = v0
```

In [13]:

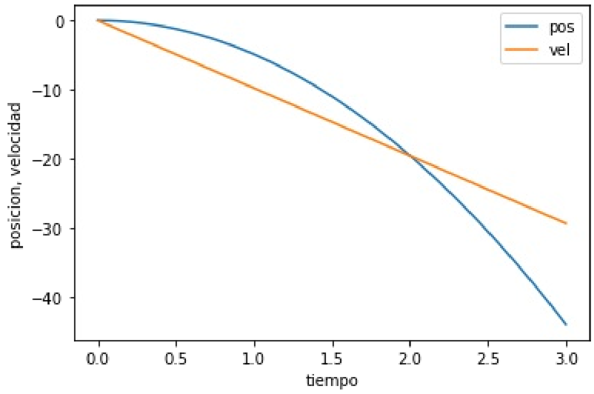
```
for j in range(N-1):
    y[j+1] = euler(y[j], time[j], dt, caida_libre)
```

In [14]:

```
xdata = [y[j,0] for j in range(N)]
vdata = [y[j,1] for j in range(N)]

plt.plot(time, xdata, label="pos")
plt.plot(time, vdata, label="vel")
plt.xlabel("tiempo")
plt.ylabel("posicion, velocidad")
plt.legend(loc="best")
```

Out[14]:

```
<matplotlib.legend.Legend at 0x7f99210e2af0>
```



En el caso de la caída libre, es posible obtener una solución exacta:

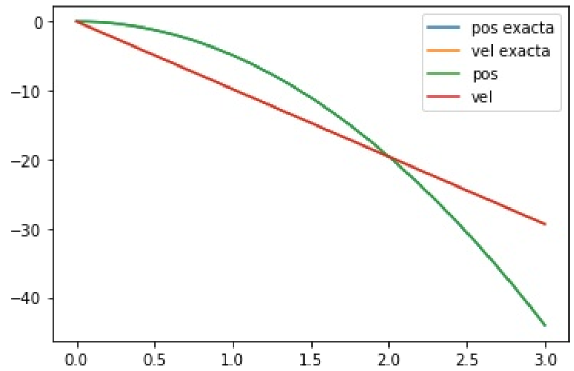
$$x(t) = x_i + v_i t + \frac{1}{2} g t^2$$
$$v(t) = v_i + gt$$

In [15]:

```
xt = lambda x_i, v_i, a, t: x_i + v_i*a + 0.5*a*t**2
vt = lambda v_i, a, t: v_i + a*t

plt.plot(time, xt(x0, v0, g, time), label="pos exacta")
plt.plot(time, vt(v0, g, time), label="vel exacta")
plt.plot(time, xdata, label="pos")
plt.plot(time, vdata, label="vel")
plt.legend(loc="best")
```

Out[15]:

```
<matplotlib.legend.Legend at 0x7f992100a0a0>
```



Al parecer en este caso no hay error (**perceptible**) en la diferenciación.

Ejemplo: Péndulo simple

La ecuación de movimiento del péndulo es la siguiente:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta)$$

donde θ es el ángulo medido desde la vertical, g es la aceleración debida a la gravedad y l es la longitud del péndulo.

Esta ecuación es **no lineal** y la revisitaremos cuando veamos caos.

Es posible **linearizarla** para el caso cuando θ es pequeño, en este caso $\sin(\theta) \approx \theta$

$$\frac{d^2\theta}{dt^2} \approx -\frac{g}{l}\theta$$

$$\ddot{\theta} \approx -\frac{g}{l}\theta$$

Esta ecuación de segundo orden se puede transformar en un sistema de ecuaciones de primer orden haciendo:

$$\dot{\theta} = \omega$$
$$\dot{\omega} = -\frac{g}{l}\theta$$

Una cantidad importante es la frecuencia

$$\Omega = \sqrt{\frac{g}{l}}$$

y su inverso el periodo

$$T = \frac{2\pi}{\Omega}$$

La **energía total exacta** E **del péndulo** es:

$$E = \frac{1}{2}ml^2\omega^2 + mgl(1 - \cos(\theta))$$

y en **nuestra aproximación** $\cos(\theta) \approx 1 - \theta^2/2$, entonces

$$E \approx \frac{1}{2}ml^2\left(\omega^2 + \frac{g}{l}\theta^2\right)$$

Estamos definiendo la energía para poder evaluar el error de nuestro método de resolución de ecuaciones diferenciales.

La ecuación del péndulo, en nuestra aproximación, tiene una **solución analítica**:

$$\theta(t) = \theta_i\cos(\Omega t) + \frac{\omega_i}{\Omega}\sin(\Omega t)$$

In [19]:

```
masa = 1.0 # En kilogramos
longitud = 1.0 # En metros
gravedad = 9.8 # m/s^2

#Omega es omega mayuscula
Omega = np.sqrt(g/longitud)
periodo = 2*np.pi/Omega
```

In [21]:

```
#Es nuestra aproximación de la energía exacta
def energia_pendulo(theta, omega, m = masa, g = gravedad, l = longitud):
    return 0.5*m*l**2 * (omega**2 + (g/l)*theta**2)
```

Ejercicio: Define una función `pendulo_analitico` que calcule en función del tiempo la posición del péndulo

Ejercicio: Grafica la solución analítica, con condiciones $\theta_i = 0.2$ y $\omega_i = 0$.

In [22]:

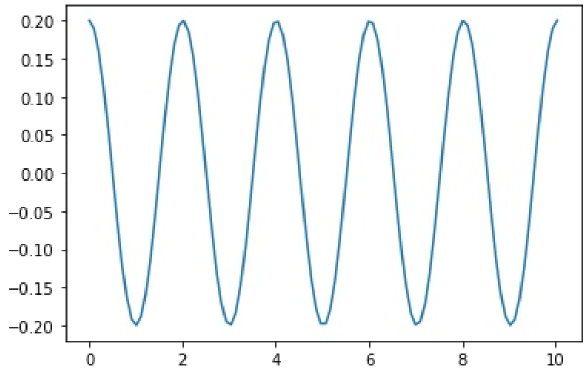
```
#t es lo unico que va a variar
#Es la solucion analitica
def pendulo_analitico(t, theta, omega, Omega):
    return theta*np.cos(Omega*t) + (omega/Omega)*np.sin(Omega*t)
```

In [23]:

```
#Son las ecuaciones diferenciales con sus condiciones iniciales
def pendulo_lineal(estado, tiempo, g=g, l=longitud):
    g0 = estado[1]
    g1 = -g/l*estado[0]
    return np.array([g0, g1])
```

In [24]:

```
tau = 5*periodo

N = 100

dt = tau/(float)(N-1)

tiempo = np.linspace(0, tau, num=N)

y = np.zeros([N,2])

plt.plot(tiempo, pendulo_analitico(tiempo, .2, 0, Omega));
```



In [25]:

```
y[0,0] = 0.2
y[0,1] = 0.0
```

In [26]:

```
def pendulo_lineal_euler(y, tiempo, dt):

    for j in range(N-1):
        y[j+1] = euler(y[j], tiempo[j], dt, pendulo_lineal)


    theta = np.array([y[j,0] for j in range(N)])
    omega = np.array([y[j,1] for j in range(N)])

    return theta, omega
```

In [27]:

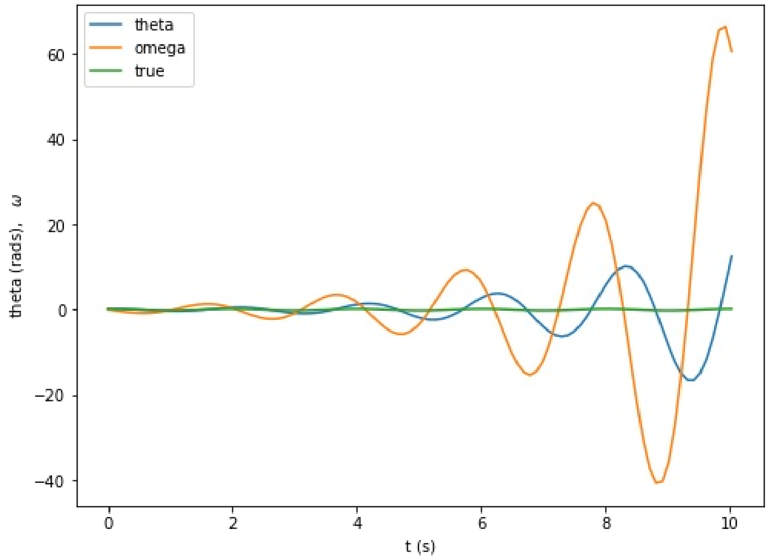
```
theta_euler, omega_euler = pendulo_lineal_euler(y, tiempo, dt)
```

In [28]:

```
plt.figure(1, figsize=(8,6))
plt.plot(tiempo, theta_euler, label="theta")
plt.plot(tiempo, omega_euler, label="omega")
plt.plot(tiempo, pendulo_analitico(tiempo, .2, 0, Omega), label="true");
plt.xlabel(r"t (s)")
plt.ylabel(r"theta (rads),$\quad\omega$")
plt.legend(loc="best")
```

Out[28]:

```
<matplotlib.legend.Legend at 0x7f9920f32e20>
```



Ejercicio: Agrega en esta gráfica la solución analítica.

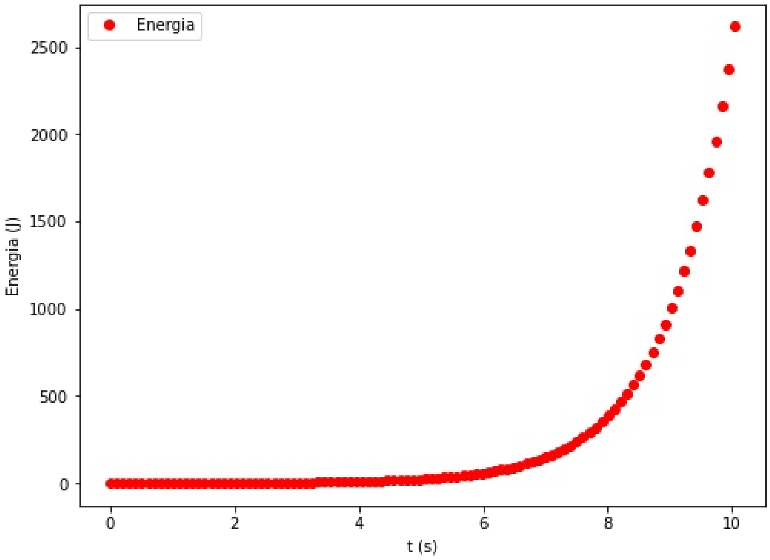
Mmmm... Creo que no se ve bien esto ¿Cómo se ve la energía?

In [29]:

```
plt.figure(1, figsize=(8,6))
plt.plot(tiempo, energia_pendulo(theta_euler, omega_euler), marker='o', linestyle='None', color='red', label="Ene
rgia")
plt.xlabel(r"t (s)")
plt.ylabel(r"Energia (J)")
plt.legend(loc="best")
```

Out[29]:

```
<matplotlib.legend.Legend at 0x7f9920eaafd0>
```



Ejercicio: ¿Por qué está mal la energía?

Ejercicio: Agrega un _widget_ para ver el número de paso N (entre $1,000$ y $30,000$) ¿Qué sucede con la energia?

Mejorando el código

Podemos mejorar el codigo si definimos los siguientes métodos:

In [30]:

```
class Pendulo:

    def __init__(self, masa, longitud, gravedad):
        self.masa = masa
        self.longitud = longitud
        self.gravedad = gravedad
        self.Omega = np.sqrt(g/longitud)
        self.period = 2*np.pi/Omega


    def theta(self):
        return self.trajectory[:,0]

    def omega(self):
        return self.trajectory[:,1]

    def plot(self):
        fig, ax = plt.subplots(3,1, figsize=(10,8), sharex = True)

        ax[0].plot(self.tau, self.theta(), label="theta", color="blue")
        ax[1].plot(self.tau, self.omega(), label="omega", color="green")
        ax[2].plot(self.tau, self.energy(),  marker='o', linestyle='None', color='red', label="Energia")

        ax[0].set_ylabel("Theta (rads)")
        ax[0].set_xlabel("tiempo (s)")

        ax[1].set_ylabel("Omega (rads/s)")
        ax[1].set_xlabel("tiempo (s)")

        ax[2].set_ylabel("Energia (J)")
        ax[2].set_xlabel("tiempo (s)")

    def initial_conditions(self, theta_i, omega_i):
        self.theta_i = theta_i
        self.omega_i = omega_i

    def dynamics(self, state, t):
        g0 = state[1]
        g1 = -self.gravedad/self.longitud*state[0]
        return np.array([g0, g1])

    def energy(self):
        return 0.5*self.masa*self.longitud**2 * (self.omega()**2 + (self.gravedad/self.longitud)*self.theta()**2)

    def integrate(self, num_steps, t_i, t_f, method):

        self.tau, self.dt = np.linspace(t_i, t_f, num=num_steps, retstep=True)
        self.trajectory = np.zeros([num_steps, 2])
        self.trajectory[0,0] = self.theta_i
        self.trajectory[0,1] = self.omega_i

        for j in range(N-1):
            self.trajectory[j+1] = method(self.trajectory[j], self.tau[j], self.dt, self.dynamics)
```

Ejemplo de uso

In [31]:

```
p = Pendulo(masa = 1.0, longitud = 1.0, gravedad = 9.8)
p.initial_conditions(theta_i=0.2, omega_i=0.0)
```

In [32]:

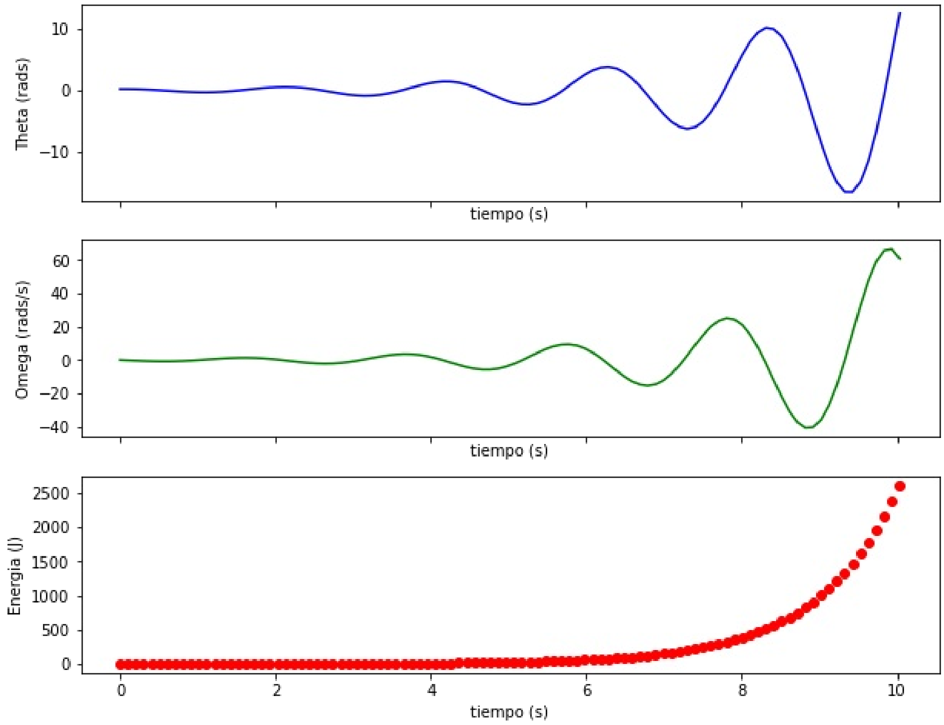
```
N = 100
tiempo_inicial = 0.0
tiempo_final = 5*p.period
```

In [33]:

```
p.integrate(N, tiempo_inicial, tiempo_final, euler)
```

In [34]:

```
p.plot()
```



Métodos de Runge-Kutta

In [35]:

```
def RK2(y, t, dt, derivadas):
    k0 = dt*derivadas(y, t)
    k1 = dt*derivadas(y + k0, t + dt)
    y_next = y + 0.5*(k0 + k1)

    return y_next
```

In [36]:

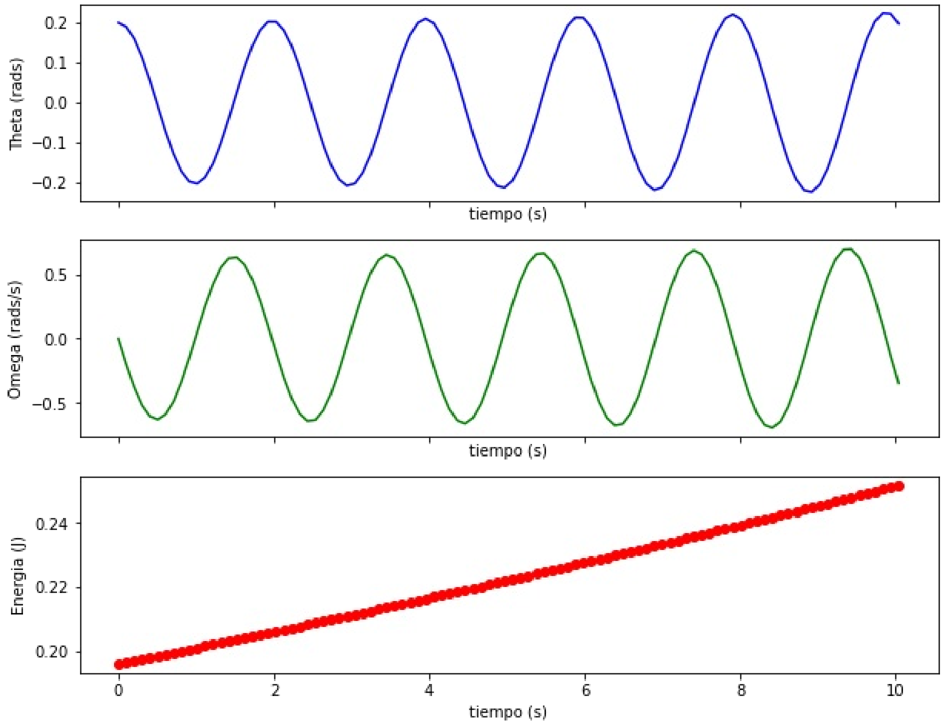
```
p2 = Pendulo(masa = 1.0, longitud = 1.0, gravedad = 9.8)
p2.initial_conditions(theta_i=0.2, omega_i=0.0)
```

In [37]:

```
N = 100
tiempo_inicial = 0.0
tiempo_final = 5*p2.period
```

In [38]:

```
p2.integrate(N, tiempo_inicial, tiempo_final, RK2)
p2.plot()
```



Ejercicio: Crea una imagen donde se muestre la θ calculada con el método de Euler, RK2 y analítica. ¿Qué observas?

In [39]:

```
plt.figure(figsize=(15, 10), dpi=80)


plt.plot(p2.tau, p2.theta(), label="RK2", color="blue")
plt.plot(p.tau, p.theta(), label="Euler", color="green")
plt.plot(tiempo, pendulo_analitico(tiempo, .2, 0, Omega), label="Analitica", color="red")

plt.ylabel("Theta (rads)")
plt.xlabel("tiempo (s)")

plt.legend(loc="best")
```

Out[39]:

```
<matplotlib.legend.Legend at 0x7f99208e7490>
```

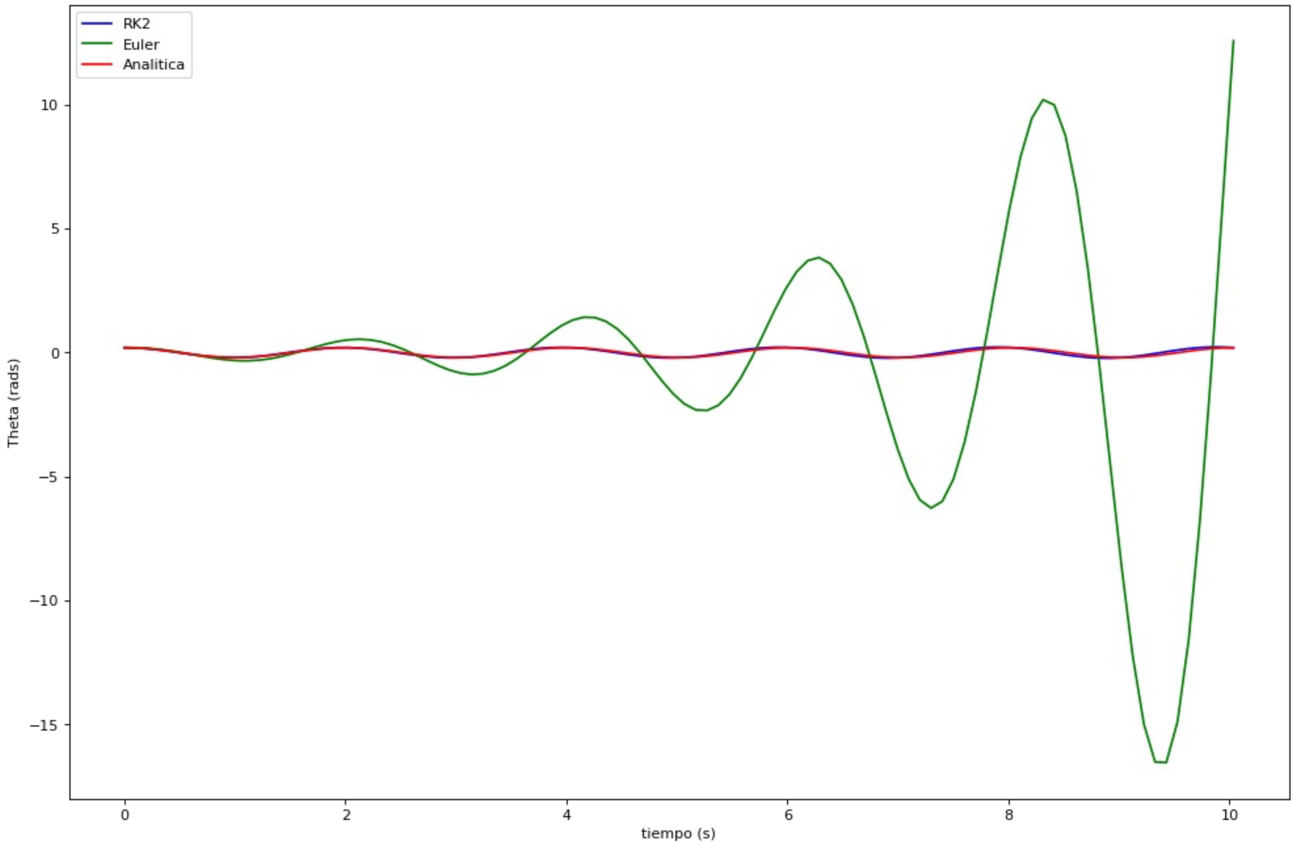


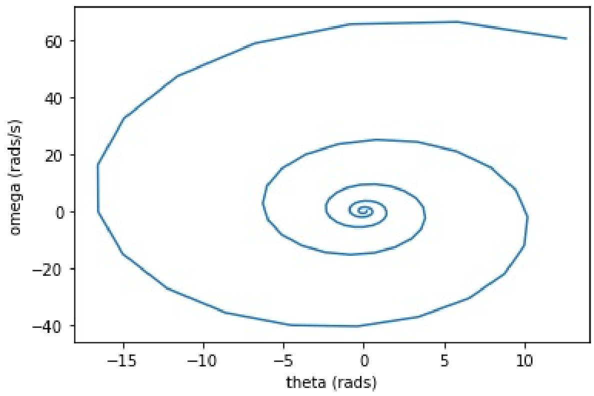
Diagrama de fase

In [40]:

```
#De euler
#p es el pendulo
plt.plot(p.theta(), p.omega())
plt.xlabel("theta (rads)")
plt.ylabel("omega (rads/s)")
```

Out[40]:

```
Text(0, 0.5, 'omega (rads/s)')
```

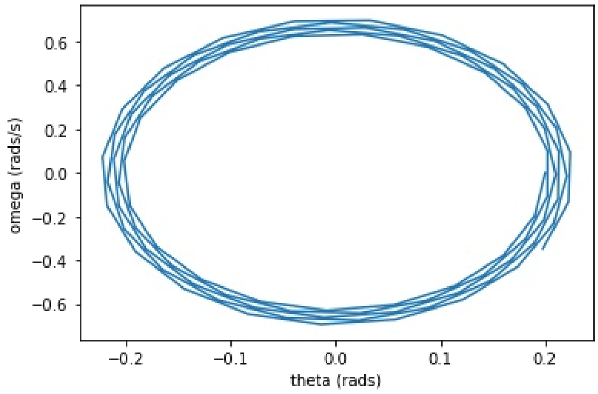


In [41]:

```
#De RK2
plt.plot(p2.theta(), p2.omega())
plt.xlabel("theta (rads)")
plt.ylabel("omega (rads/s)")
```

Out[41]:

```
Text(0, 0.5, 'omega (rads/s)')
```



Ejercicio: Agregue el método para dibujar el diagrama de fase a la clase `Pendulo`

Ejercicio: Cree un archivo `pendulo_linealizado.py` y guarde ahí la clase. Cárguelo a la sesión.

Ejercicio: Modifique el método `dynamics` para el caso en que no está linealizado el péndulo, i.e.

$$\ddot{\theta} = -\frac{g}{l}\sin\theta$$

Grafique los diagramas con el método Runge - Kutta de 2do Orden. Guardelo en el archivo `pendulo_real.py`

Ejercicio: Modifique el método `dynamics` para el caso en que no está linealizado el péndulo, exista un amortiguamiento y ademas haya una fuerza externa, como se muestra en la siguiente ecuación.

$$\ddot{\theta} = -\frac{g}{l}\sin\theta - \beta\dot{\theta} + A\cos(\omega t)$$

Grafique los diagramas con el método Runge - Kutta de 2do Orden. Guárdelo en el archivo `pendulo_actuado.py`

Métodos de Scipy

`Scipy` implementa una rutina que resuelve ecuaciones diferenciales, `odeint()` del paquete `scipy.integrate` .

In [42]:

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

In [43]:

```
odeint?
```

Como puedes observar se puede invocar con la función `dynamics` , un arreglo que represente el estado inicial y un `array` de tiempos (en lugar de un *time step*).

Ejemplo: Péndulo con resorte

una masa m está sujeta a un resorte con constante elástica k, el resorte a su vez está pegado al techo. La longitud del resorte sin deformar es L_0, y el ángulo respecto a la vertical es θ. Usando ecuaciones de [Euler-Lagrange (http://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation)](http://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation), se encuentra que las ecuaciones de movimiento son:

$$\ddot{L} = (L_0 + L)\dot{\theta}^2 - \frac{k}{m}L + g\cos\theta$$
$$\ddot{\theta} = -\frac{1}{L_0 + L}\left[g\sin\theta + 2\dot{L}\dot{\theta}\right]$$

Los escribimos como sistemas de primer grado.

$$\dot{L} = l$$
$$\dot{l} = (L_0 + L)\dot{\theta}^2 - \frac{k}{m}L + g\cos\theta$$
$$\dot{\theta} = \Theta$$
$$\dot{\Theta} = -\frac{1}{L_0 + L}\left[g\sin\theta + 2\dot{L}\dot{\theta}\right]$$

Ejercicio Escribe este par de ecuaciones como un sistema de ecuaciones de primer orden

In [44]:

```
N = 1000

# Constantes
L_0 = 1.0
L = 1.0
v_i = 0.0
theta_i = 0.3
omega_i = 0.0

k = 3.5 # COnstante del resorte en N/m
m = 0.2 # masa en kilogramos
g = 9.8 # Constante de gravedad terrestre, en m/s^2
```

In [45]:

```
# Estado inicial
y = np.zeros([4])
y
```

Out[45]:

```
array([0., 0., 0., 0.])
```

In [46]:

```
# Estado inicial
y[0] = v_i
y[1] = L_0
y[2] = omega_i
y[3] = theta_i
```

In [47]:

```
y
```

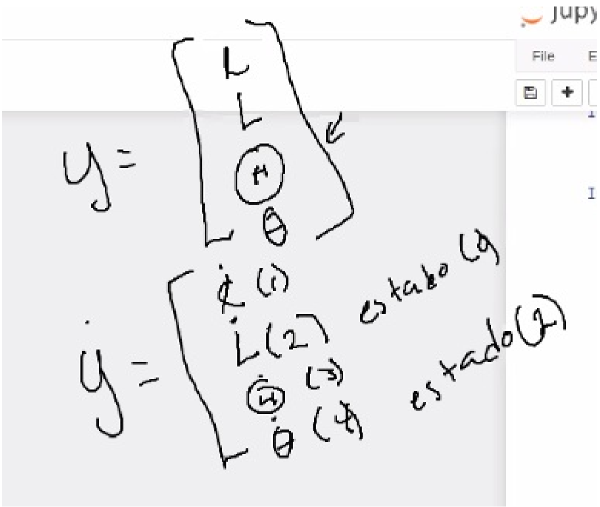
Out[47]:

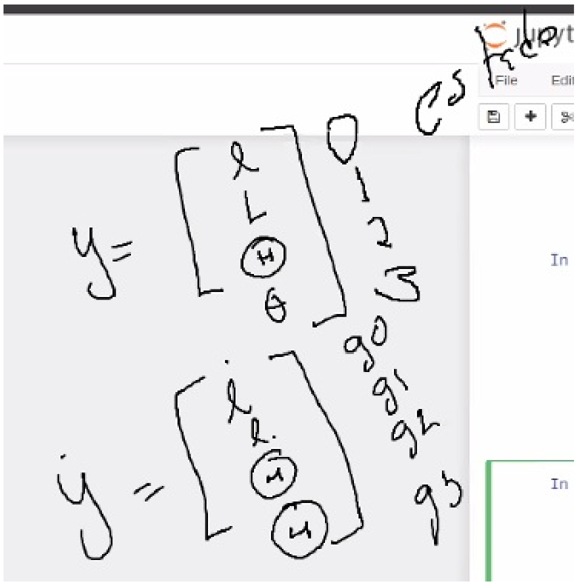
```
array([0. , 1. , 0. , 0.3])
```

In [48]:

```
time = np.linspace(0,25,N)
```

Ejercicio Escribe el sistema de ecuaciones en el método `pendulo_con_resorte(estado, tiempo)`





In [49]:

```
def pendulo_con_resorte(estado, tiempo):
    g0 = (L_0+estado[1])*estado[2]**2-k/m*estado[1] + g*np.cos(estado[3])
    g1 = estado[0]
    g2 = -1/(L_0 + estado[1])*(g*np.sin(estado[3])+2 * estado[0] * estado[2])
    g3 = estado[2]
    return np.array([g0,g1,g2,g3])
```

In [50]:

```
solucion = odeint(func=pendulo_con_resorte, y0 = y, t = time)
solucion
```

Out[50]:

```
array([[ 0.        ,  1.        ,  0.        ,  0.3       ],
       [-0.20324155,  0.99745441, -0.03629672,  0.29954621],
       [-0.40405995,  0.98984797, -0.0729493 ,  0.29818038],
       ...,
       [ 1.8035742 ,  0.4619703 , -0.61265583, -0.25346162],
       [ 1.84047428,  0.50761623, -0.5348056 , -0.26781053],
       [ 1.85363073,  0.553888  , -0.46132699, -0.28026521]])
```

Dibujaremos el movimiento en el espacio físico euclídeo 2D (x, y), para esto, necesitamos convertir a estas coordenadas en lugar usando las fórmulas trigonométricas.

In [51]:

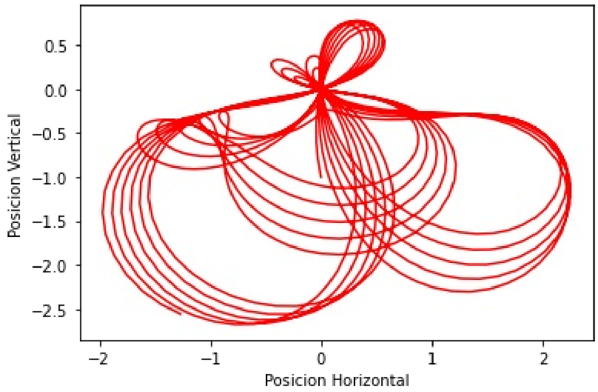
```
xdata = (L_0 + solucion[:,0])*np.sin(solucion[:,2])
ydata = -(L_0 + solucion[:,0])*np.cos(solucion[:,2])
```

In [52]:

```
plt.plot(xdata, ydata, 'r-')
plt.xlabel("Posicion Horizontal")
plt.ylabel("Posicion Vertical")
```

Out[52]:

```
Text(0, 0.5, 'Posicion Vertical')
```



Ejercicio: Dibuja respecto al tiempo el valor de L y de θ.

Ejercicio Resuelve el péndulo amortiguado, grafica en el espacio euclídeo y también la posición respecto al tiempo. La ecuación del péndulo amortiguado es:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta - b\dot{\theta} + \beta\cos(\omega t)$$