

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

def euler(f, f0, n, hf, t_o = 0):
    t = t_o
    h = ((hf)-t_o)/n
    s0 = f0
    s, T = [],[]
    while t <= hf:
        ssq = s0 + h*f(t, s0)
        s.append(ssq)
        T.append(t)
        s0 = ssq
        t += h
    return s, T

def euler2(f, f0,g0, n, hf, t_o = 0):
    h = ((hf)-t_o)/n
    s, v, T = [f0], [g0], [t_o]
    for i in range(n):
        snext = s[i] + h*v[i]
        vnext = v[i] + h*f(T[i],s[i])
        s.append(snext)
        v.append(vnext)
        T.append(T[i]+h)
    return v,s, T

def rkuta2(f, f0, n, hf, t_o = 0):
    t = t_o
    h = ((hf)-t_o)/n
    s0 = f0
    s,T = [],[]
    while t<=hf:
        k1n = h*f(t, s0)
        k2n = h*f(t, s0 + 0.5*k1n)
        s.append(s0 + k2n)
        T.append(t)
```

```

        .. \
        s0 += k2n
        t += h
    return s, T

```

```

def rk2_2(f, f0,g0, n, hf, t_o = 0):
    s = [f0]
    v = [g0]
    h = ((hf)-t_o)/n
    T = [t_o]
    for i in range(n):
        k1s = h*s[i]
        k1v = h*f(T[i],s[i])
        k2s = h * (v[i] + 0.5 * k1v)
        k2v = h * f(T[i] + 0.5 * h, s[i] + 0.5 * k1s)
        sp = s[i] + k2s
        vp = v[i] + k2v
        s.append(sp)
        v.append(vp)
        T.append(T[i] + h)
    return v,s, T

```

```

def rkuta4(f, f0, n, hf, t_o = 0):
    t = 0
    h = ((hf)-t_o)/n
    s0 = f0
    s,T = [],[]

    while t<=hf:
        k1n = h*f(t, s0)
        k2n = h*f(t, s0 + 0.5*k1n)
        k3n = h*f(t, s0 + 0.5*k2n)
        k4n = h*f(t, s0 + k3n)
        si = s0 + k1n/6 + k2n/3 + k3n/3 + k4n/6
        s.append(si)
        T.append(t)
        s0 = si

        t += h

```

```

    ..
    return s, T
def rk4_2(f, f0,g0, n, hf, t_o = 0):
    s = [f0]
    v = [g0]
    T = [t_o]
    h = ((hf)-t_o)/n
    for i in range(n):
        k1s = h*s[i]
        k1v = h*f(T[i],s[i])
        k2s = h * (v[i] + 0.5 * k1v)
        k2v = h * f(T[i] + 0.5 * h, s[i] + 0.5 * k1s)
        k3s = h * (v[i] + 0.5 * k2v)
        k3v = h * f(T[i] + 0.5 * h, s[i] + 0.5 * k2s)
        k4s = h * (v[i] + k3v)
        k4v = h * f(T[i] + 0.5 * h, s[i] + k3s)
        sp = s[i] + (k1s + 2 * k2s + 2 * k3s + k4s) / 6
        vp = v[i] + (k1v + 2 * k2v + 2 * k3v + k4v) / 6
        s.append(sp)
        v.append(vp)
        T.append(T[i] + h)
    return v,s, T

```

```

def StormerV(f, f0,g0, n, h, t_o = 0):
    p, q = np.zeros(n), np.zeros(n)
    T = [t_o, t_o + h]
    p[0], q[0] = f0, g0
    p[1] = p[0] + q[0]*h + h**2*f(T[0],p[0])*0.5
    for i in range(1,n-1):
        p[i+1] = p[i]+q[i]*h +f(T[i],p[i])*h**2
        q[i+1] = q[i] + f(T[i],p[i])*h
        T.append(T[i] + h)

    return list(p), list(q), T

```

```

def SEuler(f,g, f0,g0, n, h, t_o = 0):
    p, q= np.zeros(n+1), np.zeros(n+1)
    T = [t_o, t_o + h]

```

```

    p[0], q[0] = f0, g0
    for i in range(n):
        p[i+1] = p[i] + f(T[i],q[i]) * h
        q[i+1] = q[i] + g(T[i],p[i+1]) * h
        T.append(T[i] + h)

    return list(p), list(q), T

def dynamics_solve(f, f0 = 1, n = 100, hf = 100, D = 1, t_o = 0, method = "Euler"):
    method_dict = {
        "Euler": euler,
        "RK2": rkuta2,
        "RK4": rkuta4
    }
    if method not in method_dict:
        return None

    selected_method = method_dict[method]
    vsol = [selected_method(f[0], f0, n, hf, t_o)[-1]]
    for i in range(D):
        vsol.append(selected_method(f[i], f0, n, hf, t_o)[0])

    return vsol

def hamiltonian_solve(f,g, f0 = 1,g0=1, n = 100, h = 0.1,hf = 100, D = 1, t_o = 0, method = "Euler"):
    if method == "Euler":
        if D==1:
            vsolp, vsolq, T = euler2(f[0], f0, g0, n, hf, t_o)
            return (vsolp), (vsolq), (T)
        else:
            vsolp, vsolq, T = euler2(f[0], f0, g0, n, hf, t_o)
            for i in range(D):
                eul = euler2(f[i], f0, g0, n, hf, t_o)
                vsolp.append(eul[0])
                vsolq.append(eul[1])

            return vsolp, vsolq, T
    elif method == "RK2":
        if D==1:

```

```

11 D==1:
    vsolp, vsolq, T = rk2_2(f[0], f0, g0, n, hf, t_o)
    return vsolp, vsolq, T
else:
    vsolp, vsolq, T = rk2_2(f[0], f0, g0, n, hf, t_o)
    for i in range(D):
        eul = rk2_2(f[i], f0, g0, n, hf, t_o)
        vsolp.append(eul[0])
        vsolq.append(eul[1])

    return vsolp, vsolq, T
elif method == "RK4":
    if D==1:
        vsolp, vsolq, T = rk4_2(f[0], f0, g0, n, hf, t_o)
        return vsolp, vsolq, T
    else:
        vsolp, vsolq, T = rk4_2(f[0], f0, g0, n, hf, t_o)
        for i in range(D):
            eul = rk4_2(f[i], f0, g0, n, hf, t_o)
            vsolp.append(eul[0])
            vsolq.append(eul[1])

        return vsolp, vsolq, T
elif method == "SV":
    solp, solq, T = StormerV(f[0], f0, g0, n, h, t_o)
    if D == 1:
        return solp, solq, T
    else:
        for i in range(D):
            sverle = StormerV(f[i], f0, g0, n, h, t_o)
            solp.append(sverle[0])
            solq.append(sverle[1])
        return solp, solq, T
elif method == "SE":
    solp, solq, T = SEuler(f[0], g[0], f0, g0, n, h, t_o)
    if D == 1:
        return solp, solq, T
    else:

```

```

    for i in range(v):
        seu = SEuler(f[i], g[i], f0, g0, n, h, t_o)
        solp.append(seu[0])
        solq.append(seu[1])

    return solp, solq, T
else:
    return "None"

```

Comienza a programar o generar con IA.

```

for i in range(2,5):
    print(i)

2
3
4

```

## ▼ Project 1

### 1. Code tests for Euler, RK2, and RK4 using simple population model

$$\frac{dP}{dt}(t) = (B - D)P(t), \quad P(0) \equiv P_o$$

For this population model,  $B_1 = 0.57$ ,  $D_1 = 0.88$  &  $B_2 = 0.99$ ,  $D_2 = 0.45$ .

- The following results were obtained for various  $h$  (step sizes), and it's shown how given  $h$  sufficiently small, one can obtain an arbitrarily good agreement with the function  $P(t) = A \exp((B - D)t)$ , the exact solutions of the previous function, by graphing a series of plots obtained by the different methods (Euler, RK2, RK4), in contrast with the exact solutions, first using  $B_1, D_1$

✓ The blue line is the approximation and the orange line is the analytical solution.

```
b1 = 0.57
```

```
b2 = 0.99
```

```
d1 = 0.88
```

```
d2 = 0.45
```

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 142)
```

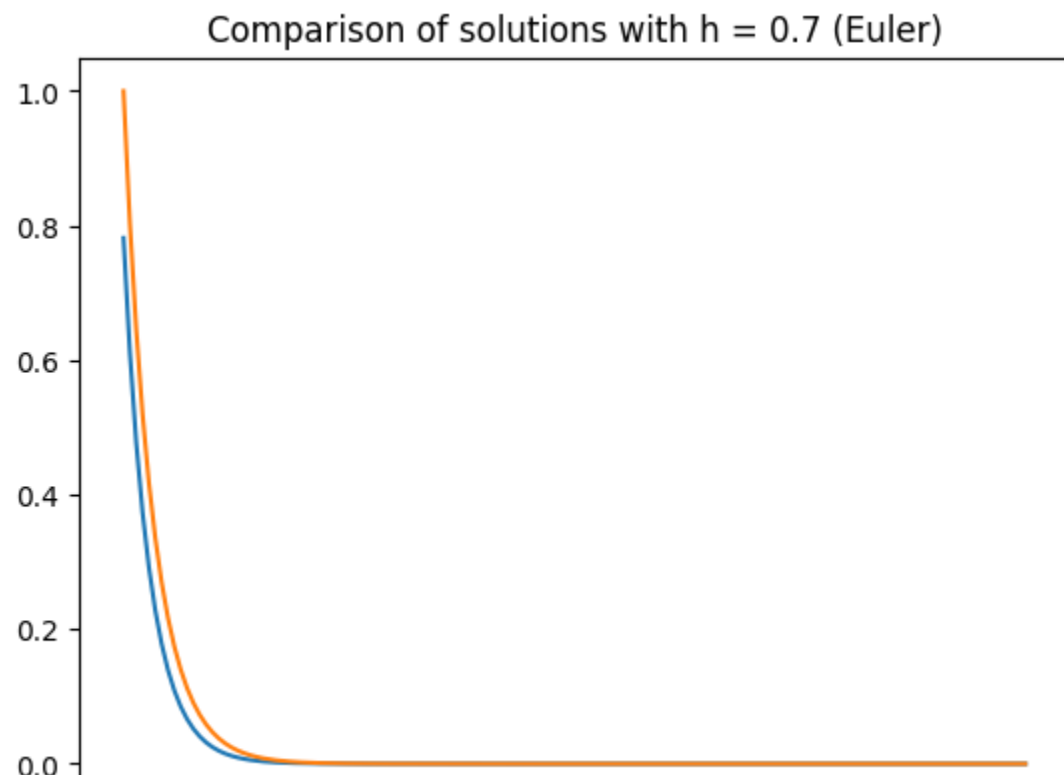
```
plt.title("Comparison of solutions with h = 0.7 (Euler)")
```

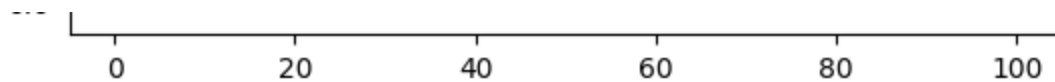
```
y = [np.exp((b1-d1)*t) for t in sol[0]]
```

```
plt.plot(sol[0],sol[1])
```

```
plt.plot(sol[0],y)
```

```
[<matplotlib.lines.Line2D at 0x7e0d08f7c730>]
```





Comienza a programar o [generar](#) con IA.

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n=345)
```

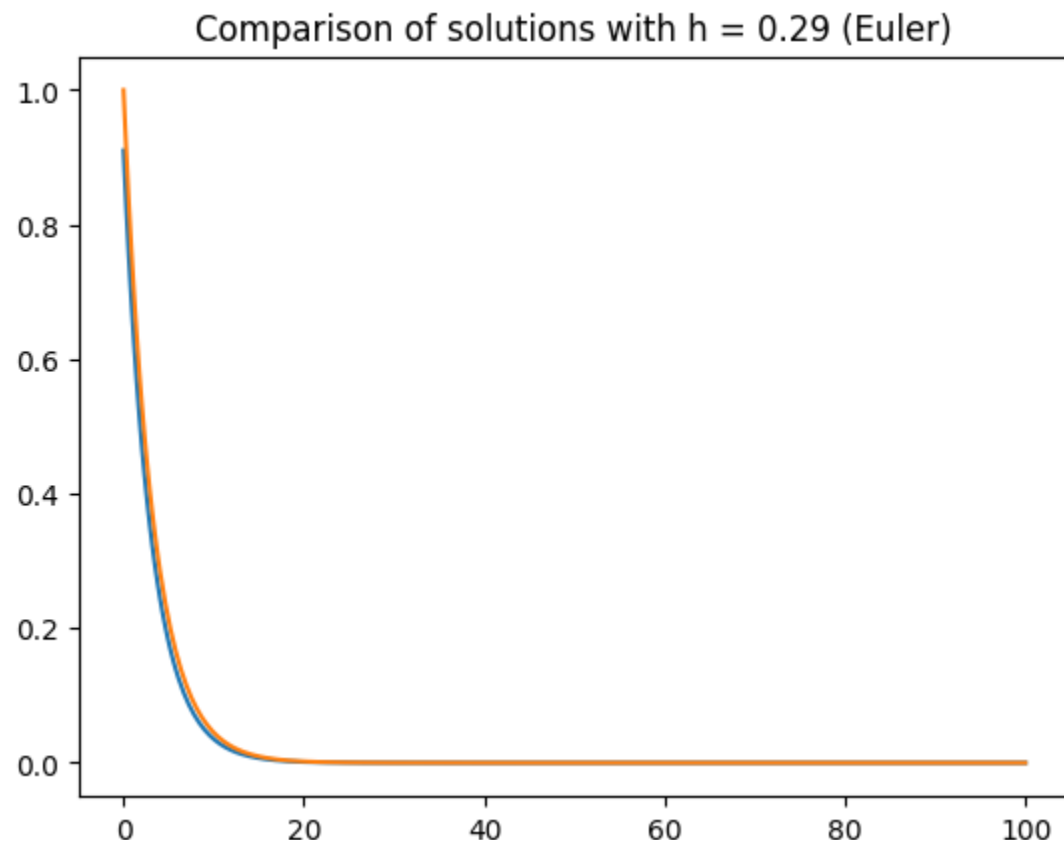
```
plt.title("Comparison of solutions with h = 0.29 (Euler)")
```

```
y=[np.exp((b1-d1)*t) for t in sol[0]]
```

```
plt.plot(sol[0],sol[1])
```

```
plt.plot(sol[0],y)
```

```
[<matplotlib.lines.Line2D at 0x7e0d08e102e0>]
```



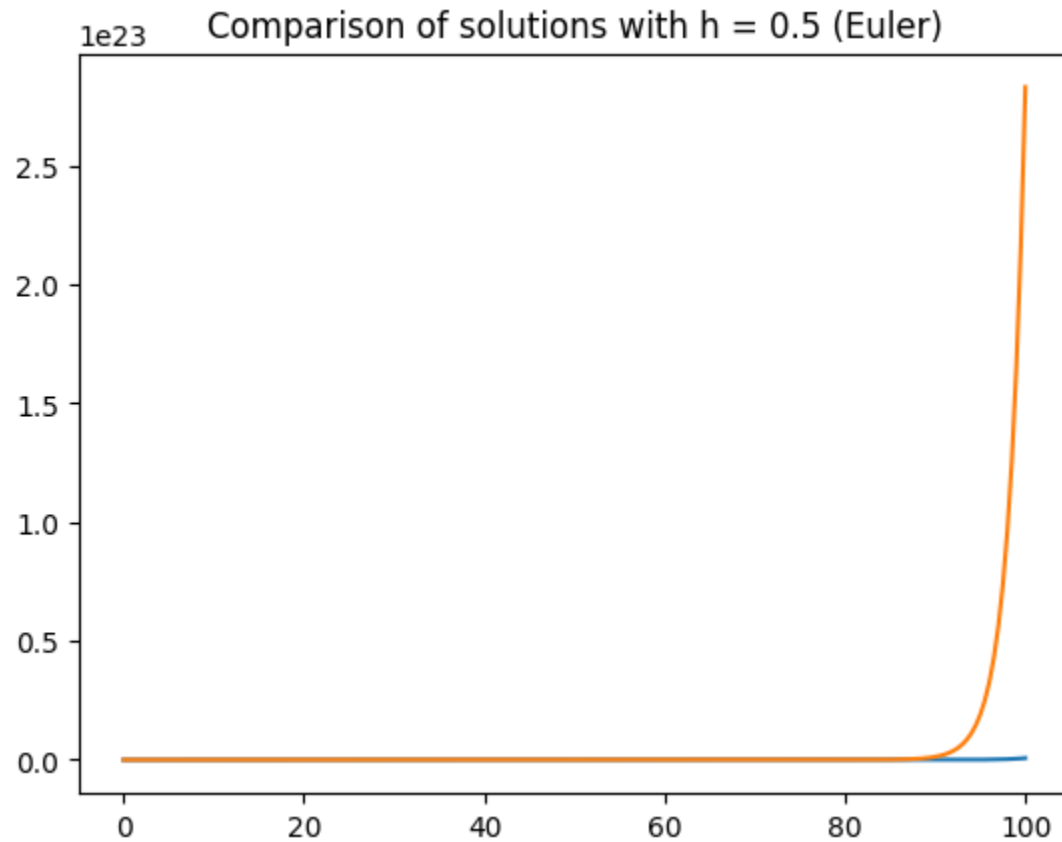


Then using  $B_2, D_2$ .

```
sol = dynamics_solve([lambda t,x: ((b2-d2)*x)], n = 200)

plt.title("Comparison of solutions with h = 0.5 (Euler)")
y=[np.exp((b2-d2)*t) for t in sol[0]]
plt.plot(sol[0],sol[1])
plt.plot(sol[0],y)
```

[<matplotlib.lines.Line2D at 0x7e0d08e5f8b0>]



```
sol = dynamics_solve([lambda t,x: ((b2-d2)*x)], n = 1000)
```

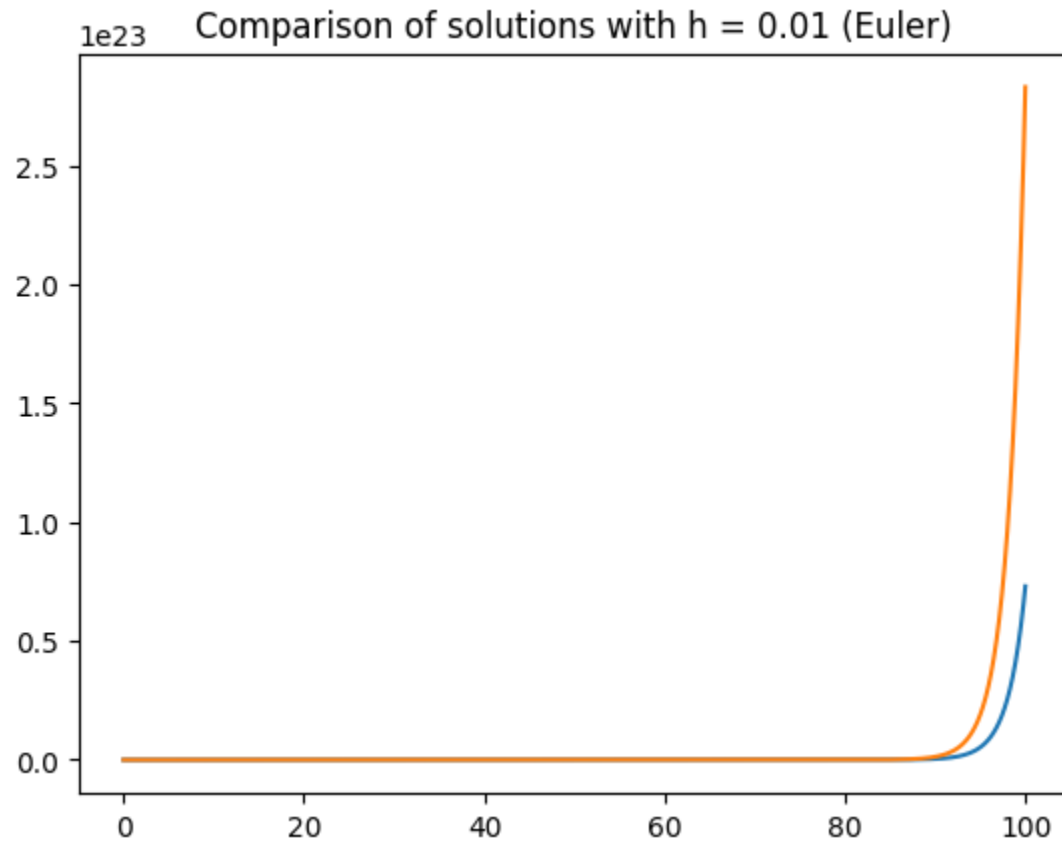
```
plt.title("Comparison of solutions with h = 0.01 (Euler)")
```

```
y = [np.exp((b2-d2)*t) for t in sol[0]]
```

```
plt.plot(sol[0],sol[1])
```

```
plt.plot(sol[0],y)
```

```
[<matplotlib.lines.Line2D at 0x7e0d08ccf220>]
```

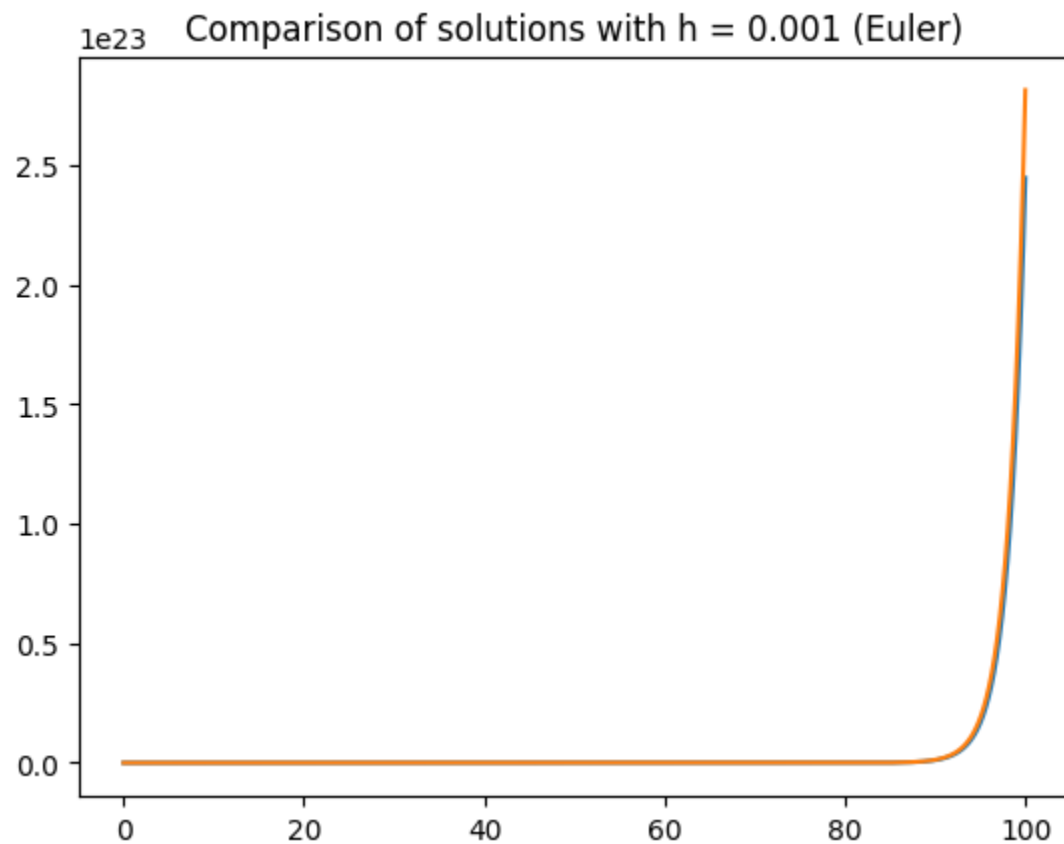


```
sol = dynamics_solve([lambda t,x: ((b2-d2)*x)], n = 10000)
```

```
plt.title("Comparison of solutions with h = 0.001 (Euler)")
```

```
plt.title('Comparison of solutions with h = 0.001 (Euler)')
y = [np.exp((b2-d2)*t) for t in sol[0]]
plt.plot(sol[0], sol[1])
plt.plot(sol[0], y)
```

[<matplotlib.lines.Line2D at 0x7e0d08bfe2c0>]



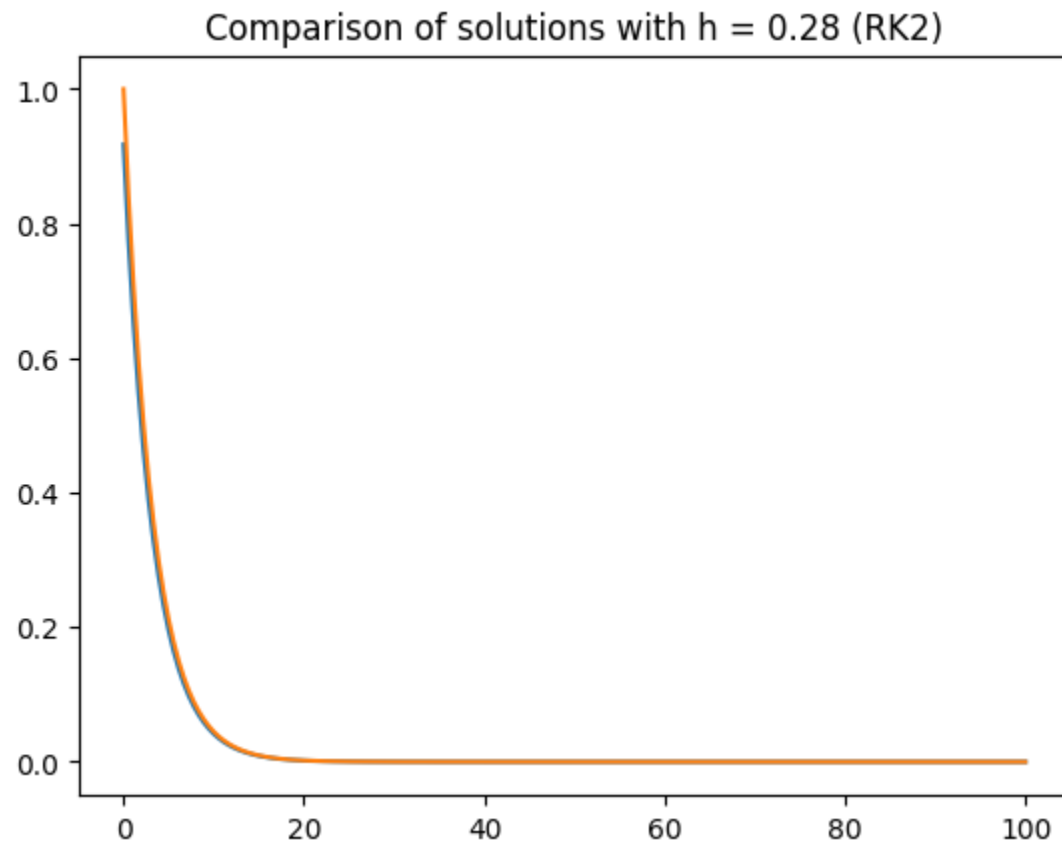
It can be shown that for values of  $h$  sufficiently close to 0, there will be an increasingly better approximation using Euler. Particularly,  $h \ll |B - D|$  for convergence to start appearing.

This process holds true for RK2.

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 360, method = "RK2")
```

```
plt.title("Comparison of solutions with h = 0.28 (RK2)")  
y = [np.exp((b1-d1)*t) for t in sol[0]]  
plt.plot(sol[0],sol[1])  
plt.plot(sol[0],y)
```

```
[<matplotlib.lines.Line2D at 0x7e0d08c75c30>]
```



```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 500, method = "RK2")
```

```
plt.title("Comparison of solutions with h = 0.2 (RK2)")
```

```
y=[np.exp((b1-d1)*t) for t in sol[0]]  
plt.plot(sol[0],sol[1])  
plt.plot(sol[0],y)
```

```
sol = dynamics_solve([lambda t,x: ((b2-d2)*x)], n = 1000, method = "RK2")  
  
plt.title("Comparison of solutions with h = 0.1 (RK2)")  
y=[np.exp((b2-d2)*t) for t in sol[0]]  
plt.plot(sol[0],sol[1])  
plt.plot(sol[0],y)
```

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 200, method = "RK2")

plt.title("Comparison of solutions with h = 0.5 (RK2)")
y=[np.exp((b1-d1)*t) for t in sol[0]]
plt.plot(sol[0],sol[1])
plt.plot(sol[0],y)
```

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 5000, method = "RK2")

plt.title("Comparison of solutions with h = 0.007 (RK2)")

y=[np.exp((b1-d1)*t) for t in sol[0]]
plt.plot(sol[0],sol[1])
plt.plot(sol[0],y)
```

And RK4.

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 357, method = "RK4")
```

```
plt.title("Comparison of solutions with h = 0.28 (RK4)")  
y=[np.exp((b1-d1)*t) for t in sol[0]]  
plt.plot(sol[0],sol[1])  
plt.plot(sol[0],y)
```



```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 2000, method = "RK4")

plt.title("Comparison of solutions with h = 0.05 (RK4)")
y =[np.exp((b1-d1)*t) for t in sol[0]]
plt.plot(sol[0],sol[1])
plt.plot(sol[0],y)
```

```
sol = dynamics_solve([lambda t,x: ((b1-d1)*x)], n = 3000, method = "RK4")

plt.title("Comparison of solutions with h = 0.03 (RK4)")

y =[np.exp((b1-d1)*t) for t in sol[0]]
plt.plot(sol[0],sol[1])
plt.plot(sol[0],y)
```

For RK4, there were cases (where  $h < 0.28$ ) in which the values of the analytic solution and the approximation were indistinguishable.

## ✓ 2. Code tests for Euler, RK2, Symplectic Euler, Stormer Verlet and RK4 for the system:

$$\begin{aligned}\frac{dp}{dt}(t) + kx &= 0 \\ \frac{dx}{dt}(t) - \frac{p}{m} &= 0\end{aligned}$$

Comienza a programar o [generar](#) con IA.

## Tests for fixed N of oscillation periods

We shall compare the graphs of the solution using Symplectic Euler and the analytical version

The blue line is the approximation and the orange line is the analytical solution

## ✓ Symplectic Euler

```
- .  
  
sola = hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 0,g0=1, n= 100, h= 0.5, D = 1, t_o = 0, method = "SE")  
cosx = [np.cos(i) for i in sola[2]]  
sinx = [3*np.sin(i) for i in sola[2]]  
plt.title("solution for h=0.5")  
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```

```
sola = hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100, h = 0.1,D = 1, t_o = 0, method = "SE")  
cosx = [np.cos(i) for i in sola[2]]
```

```
sinx = [3*np.sin(i) for i in sola[2]]  
plt.title("solution for h=0.1")  
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```

```
sola = hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, h = 0.01, n = 1000, D = 1, t_o = 0, method = '  
cosx = [np.cos(i) for i in sola[2]]  
sinx = [3*np.sin(i) for i in sola[2]]  
plt.title("solution for h=0.01")  
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```

We can observe convergence in  $h=0.1$  already.

## ▼ Stormer Verlet

```
sola =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, h= 0.1, n = 100, D = 1, t_o = 0, method = "SV")
plt.title("Solution for h=0.1")
cosx = [1.75*np.cos(i) for i in sola[2]]
sinx = [np.sin(i) for i in sola[2]]
```

```
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```

```
sola =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, h= 0.05, n = 1000, D = 1, t_o = 0, method = "  
plt.title("Solution for h=0.05")  
cosx = [1.75*np.cos(i) for i in sola[2]]  
sinx = [np.sin(i) for i in sola[2]]  
  
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```

```
plt.plot(cosx,sinx)
```

```
sola =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, h= 0.001, n = 10000, D = 1, t_o = 0, method =  
plt.title("Solution for h=0.001")  
cosx = [1.75*np.cos(i) for i in sola[2]]  
sinx = [np.sin(i) for i in sola[2]]  
plt.plot(sola[1],sola[0])  
plt.plot(cosx,sinx)
```



```
sola =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, h= 0.0005, n = 10000, D = 1, t_o = 0, method

plt.title("Solution for h=0.0005")
cosx = [1.75*np.cos(i) for i in sola[2]]
sinx = [np.sin(i) for i in sola[2]]
plt.plot(sola[1],sola[0])
plt.plot(cosx,sinx)
```

## ▼ Euler

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 10, D = 1, t_o = 0, method = "Eu]

cosx = [1*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for h = 0.1")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 1000,hf = 10, D = 1, t_o = 0, method = "Eu  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for h = 0.01")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 10000,hf = 10, D = 1, t_o = 0, method =  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for h = 0.001")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

## ✓ Runge-Kutta 2

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 10, D = 1, t_o = 0, method = "R

cosx = [1*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for h = 0.1")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 1000,hf = 10, D = 1, t_o = 0, method = "  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for h = 0.01")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 10000,hf = 10, D = 1, t_o = 0, method =  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for h = 0.01")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

## ✓ Runge-Kutta 4

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 10, D = 1, t_o = 0, method = "RK4")

cosx = [1*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for h = 0.1")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```



```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 1000,hf = 10, D = 1, t_o = 0, method = "Rk4")

cosx = [1*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for h = 0.01")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 10000,hf = 10, D = 1, t_o = 0, method =  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for h = 0.001")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

## ✎ Tests for increasing N, given $h=0.1$

### ✎ Symplectic Euler

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 100, D = 1, t_o = 0, method = "  
  
plt.title("Solution for n=100")  
plt.plot(sola[1],sola[0])  
cosx = [2*np.cos(i) for i in sola[2]]  
sinx = [np.sin(i) for i in sola[2]]  
plt.plot(cosx, sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 500,hf = 100, D = 1, t_o = 0, method = "  
  
plt.title("Solution for n=500")  
plt.plot(sola[1],sola[0])  
cosx = [2*np.cos(i) for i in sola[2]]  
sinx = [np.sin(i) for i in sola[2]]  
plt.plot(cosx, sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 2000,hf = 100, D = 1, t_o = 0, method =  
  
plt.title("Solution for n=2000")  
plt.plot(sola[1],sola[0])  
cosx = [2*np.cos(i) for i in sola[2]]  
sinx = [np.sin(i) for i in sola[2]]  
plt.plot(cosx, sinx)
```

## ▼ Stormer Verlet

```
solver1 =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 100, D = 1, t_o = 0, method =  
  
cosx = [1.75*np.cos(i) for i in solver1[2]]  
sinx = [np.sin(i) for i in solver1[2]]  
plt.title("solution for n=100")  
plt.plot(solver1[1],solver1[0])  
plt.plot(cosx,sinx)
```

```
solver1=hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 500,hf = 100, D = 1, t_o = 0, method =  
  
cosx = [1.75*np.cos(i) for i in solver1[2]]  
sinx = [np.sin(i) for i in solver1[2]]  
plt.title("solution for n=500")  
plt.plot(solver1[1],solver1[0])  
plt.plot(cosx,sinx)
```

```
solver1=hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 1000,hf = 100, D = 1, t_o = 0, method =  
cosx = [1.75*np.cos(i) for i in solver1[2]]  
sinx = [np.sin(i) for i in solver1[2]]  
plt.title("solution for n=1000")  
plt.plot(solver1[1],solver1[0])  
plt.plot(cosx,sinx)
```

```
plt.title("solution for n=1000")  
plt.plot(solver1[1],solver1[0])  
plt.plot(cosx,sinx)
```

```
solver1=hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 5000,hf = 100, D = 1, t_o = 0, method =  
cosx = [1.75*np.cos(i) for i in solver1[2]]  
sinx = [np.sin(i) for i in solver1[2]]  
plt.title("solution for n=5000")  
plt.plot(solver1[1],solver1[0])  
plt.plot(cosx,sinx)
```



## ▼ Euler

```
solap = hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 10, D = 1, t_o = 0, method = "  
  
cosx = [1.5*np.cos(i) for i in solap[2]]  
sinx = [np.sin(i) for i in solap[2]]  
plt.title("solution for n = 100")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 20, D = 1, t_o = 0, method = "E
cosx = [1.5*np.cos(i) for i in solap[2]]
sinx = [np.sin(i) for i in solap[2]]
plt.title("solution for n = 200")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

## ▼ Runge-kutta 2

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 10, D = 1, t_o = 0, method = "R
cosx = [1.5*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for n = 100")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 20, D = 1, t_o = 0, method = "R
cosx = [1.5*np.cos(i) for i in solap[2]]
sinx = [1*np.sin(i) for i in solap[2]]
plt.title("solution for n = 200")
plt.plot(solap[0],solap[1])
plt.plot(cosx,sinx)
```

## ✓ Runge-Kutta 4

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 100, D = 1, t_o = 0, method = "  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for n = 100")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

```
solap =hamiltonian_solve([lambda t,q: -q*3],[lambda t,p: p/3], f0 = 1,g0=1, n = 100,hf = 20, D = 1, t_o = 0, method = "R  
  
cosx = [1*np.cos(i) for i in solap[2]]  
sinx = [1*np.sin(i) for i in solap[2]]  
plt.title("solution for n = 200")  
plt.plot(solap[0],solap[1])  
plt.plot(cosx,sinx)
```

We start to see convergence for  $n > 100$ , fixed  $h$  for Stormer-Verlet and Symplectic Euler, but for Euler it seems to gain energy for fairly low instances of  $n$ . So far the worst (most deviating) one for this case is Euler. It starts to diverge with fairly low values

However, for increasingly lower values for  $h$ , there seems to be closed orbits for either Stormer/Symplectic Euler or Euler/RK2/RK4. This makes sense since the value of  $q_{i+1}, p_{i+1}$  depends linearly wrt  $h, n$  for Euler, so increasing  $n$  "gives" energy, but reducing  $h$  keeps the results fairly stable.

Out of the non symplectic methods the least bad one was Runge-Kutta 4, mainly because it takes more time correcting losses of energy.

## Energy manipulation of different methods

### ✦ Symplectic Euler

```
sola = SEuler(lambda t,p: p/3, lambda t,q: -3*q,f0=0,g0=1, h=0.5, n=50)
eK = [(sola[0][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
```

## ▼ Stormer Verlet

```
sola =StormerV(lambda t,p: -3*p,f0=1,g0=0, h=0.5, n=100)
eK = [(sola[0][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
```



## ▼ Euler

```
sola =euler2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
eK = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
```

## ▼ Runge Kutta 2

```
sola =rk2_2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
eK = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
```

## ✓ Runge Kutta 4

```
sola =rk4_2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
eK = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
```

## ▼ Comparing Euler, RK2, RK4

```
sola =euler2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
sola2 =rk2_2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
sola3 =rk4_2(lambda t,p: -3*p,f0=1,g0=0, hf=100, n=100)
eK = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK)
eK2 = [(sola2[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK2)
eK3 = [(sola3[1][i]**2)/6 for i in range(len(sola[0]))]
plt.plot(sola[2],eK3)
```

## ▼ Removing Euler

```
eK2 = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK2)  
eK3 = [(sola3[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK3)
```

## ▼ Removing RK2

```
eK2 = [(sola2[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK2)  
eK3 = [(sola3[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK3)
```

## ▼ Removing RK4

```
eK2 = [(sola2[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK2)  
eK3 = [(sola[1][i]**2)/6 for i in range(len(sola[0]))]  
plt.plot(sola[2],eK3)
```

We see that for Symplectic Euler and Stormer Verlet we get the expected result of kinetic energy going up and down with respect to time. There seems to be better overall stability for Euler but for Verlet there are more defined changes in energy. The other methods blow up wrt time in different levels. The worst one is Euler, followed by RK2 and RK4 seems to emulate well the presence of changing energies, but still, the graph makes it seem as if the system is getting energy out of nowhere.