Lecture 16: Scheming with ODEs (MOSTLY the same as Tutorial Solutions 06)

In this tutorial, we will experiment with different ODE models using different time integration schemes.

One of the best all-purpose methods for time stepping is the Runge-Kutta 4th-order method.

https://en.wikipedia.org/wiki/List_of_Runge–Kutta_methods (https://en.wikipedia.org/wiki/List_of_Runge–Kutta_methods)

$$\begin{aligned} k_1 &= f(t,\, x(t)) \\ k_2 &= f(t + \tfrac{h}{2},\, x(t) + \tfrac{h}{2} k_1) \\ k_3 &= f(t + \tfrac{h}{2},\, x(t) + \tfrac{h}{2} k_2) \\ k_4 &= f(t + h,\, x(t) + h\, k_3) \end{aligned}$$

$$x(t+h) \;=\; x(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

We will formulate this in a slightly different way than is usually found in books.

$$i_1 = \frac{h}{2} f(x(t)), \quad i_2 = \frac{h}{2} f(x(t) + i_1), \quad i_3 = h f(x(t) + i_2), \quad i_4 = \frac{h}{2} f(x(t) + i_3)$$

$$x(t+h) = x(t) + \frac{1}{3}\,(i_1 + 2\,i_2 + i_3 + i_4)$$

This first way which this is simpler is that there is no explicit time dependence. This might seem like a reduction in the number of systems we can solve, but there is not actually loss of generality.

Suppose we have a system

$$\frac{dx}{dt} = f(t, x)$$

where $x \in \mathbb{R}^d$. We can write this system in a slightly different way

$$\frac{dx}{dt} = f(s, x), \qquad \frac{ds}{dt} = 1$$

If we now define variable and function

$$z = [\, s,\ x \,]^\top$$

and

$$F(z) = [f(z),\ 1\,]^\top.$$

Then

$$z \in \mathbb{R}^{d+1} \quad \text{and} \quad F : \mathbb{R}^{d+1} \to \mathbb{R}^{d+1}$$

$$\frac{dx}{dt} = f(t, x) \qquad \Longrightarrow \qquad \frac{dz}{dt} = F(z)$$

There are convenience reasons we might want to leave time-dependent systems as a special case. But for now it means we can build a time stepping code that doesn't worry about it.

TASK 0

There is a simple relation between k_1, k_2, k_3, k_4 and i_1, i_2, i_3, i_4. Find this rescaling.

If you equate the (second) arguments of f, you see that

$$i_1 = \frac{h}{2} k_1$$
$$i_2 = \frac{h}{2} k_2$$
$$i_3 = h k_3$$

Then, using the last equation for $x(t + h)$, you equate them to get

$$i_4 = \frac{h}{2} k_4$$

Not sure how useful this is. Especially because they aren't all the same rescaling.

RK4 class

Below is a Python `class` that impliments the 4th-order Runge-Kutta scheme (RK4).

This method is called 4th-order accurate in h. If you analyse the error, you'll find the first error term is $\mathcal{O}(h^5)$. But this is for a single time step. But over time, the ***global accumulated*** error will be $\mathcal{O}(h^4)$.

This `class` takes a starting value for x, and pure function $f(x)$ and a possible starting time.

The `__call__()` method takes a single time step with size `dt`. It stores the past data in a list. You can get `numpy arrays` with the data and times with the `state()` method. The `now()` method gives the latest time.

In [1]:

```
import numpy as np

class RK4():

    def __init__(self,start,forcing,time=0):
        self.time = [time]
        self.data = [start]
        self.RHS  = forcing

    def __call__(self,dt):

        self.time += [ self.time[-1] + dt ]

        x = self.data[-1]
        f = self.RHS

        i1 = (dt/2)*f(x)
        i2 = (dt/2)*f(x+i1)
        i3 =   (dt)*f(x+i2)
        i4 = (dt/2)*f(x+i3)

        self.data += [ x + (1/3)*( i1 + 2*i2 + i3 + i4 ) ]

    def state(self):
        return np.array(self.time), np.array(self.data).T

    def now(self):
        return self.time[-1]
```

To see how `RK4` works we need a system to integrate.

The pendulum equation is

$$x''(t) = -\sin x(t)$$

becuase

$$\sin x \;\approx\; x + \mathcal{O}(x^3), \quad \text{as} \quad x \to 0$$

We can also consider the equation for a simple harmonic oscillator

$$x''(t) = -x(t)$$

This is the same as a linearised pendulum

In both the nonlinear and linearised case, we need to put the system into 1st-order form.

$$\frac{d}{dt}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} x_1 \\ -\sin x_0 \end{bmatrix} \qquad \text{or} \qquad \frac{d}{dt}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} x_1 \\ -x_0 \end{bmatrix}$$

Below we see an example of a Python `class` that has both options.

In [2]:

```
class Pendulum():

    def __init__(self,linear=False):
        self.linear = linear

    def __call__(self,x):

        if self.linear:
            return np.array( [ x[1], -x[0] ] )

        return np.array( [ x[1], -np.sin(x[0]) ] )
```

This `class` will give us *objects* that are individual functions that allow us to evaluate the right-hand side of the

In [3]:

```
P = Pendulum()
L = Pendulum(linear=True)
```

`P` and `L` are both functions that we can call. They are also objects in the sense that they are particualr *instances* of the class `Pendulum`.

To see how they work, we need some data. We can even put a bunch of vectors in at the same and see what comes out.

In [4]:

```
x = np.array([[1,0],[0,1],[2,3],[4,5]]).T
print('x =')
print(x)
print('')
print('L(x)=')
print(L(x))
```

```
x =
[[1 0 2 4]
 [0 1 3 5]]

L(x)=
[[ 0  1  3  5]
 [-1  0 -2 -4]]
```

We can see something different comes out with `P`.

In [5]:

```
x = np.array([[1,0],[0,1],[2,3],[4,5],[np.pi/2,9]]).T
print('x =')
print(x)
print('')
print('P(x)=')
print(P(x))
```

```
x =
[[1.         0.         2.         4.         1.57079633]
 [0.         1.         3.         5.         9.        ]]

P(x)=
[[ 0.          1.          3.          5.          9.        ]
 [-0.84147098 -0.         -0.90929743  0.7568025  -1.        ]]
```

We can now make time steppers for each one of these function. Each time stepper will keep track of the data as we go.

In [6]:

```
x0, y0 = .6, 1.9
initial_data = np.array([x0, y0])

linear_RK4    = RK4(initial_data,L)
nonlinear_RK4 = RK4(initial_data,P)
```

We make the initial conditions $[\,0.6,\ 1.9\,]$ in both cases. We can see that the states are the same as the initial conditions.

In [7]:

```
t, x = linear_RK4.state()
print(t)
print(x)

t, x = nonlinear_RK4.state()
print(t)
print(x)
```

```
[0]
[[0.6]
 [1.9]]
[0]
[[0.6]
 [1.9]]
```

Let's take a single timestep with each

In [8]:

```
dt = 0.02
linear_RK4(dt)
nonlinear_RK4(dt)
```

It might look like nothing happened. But the states are updated.

In [9]:

```
t, x = linear_RK4.state()
print(t)
print(x)

t, x = nonlinear_RK4.state()
print(t)
print(x)
```

```
[0.   0.02]
[[0.6        0.63787747]
 [1.9        1.88762081]]
[0.   0.02]
[[0.6       0.637885 ]
 [1.9       1.8883969]]
```

Let's advance these 20 time units:

In [10]:

```
while linear_RK4.now() < 20:
    linear_RK4(dt)
    nonlinear_RK4(dt)
```

Get the data:

In [11]:

```
t, xl = linear_RK4.state()
t, xn = nonlinear_RK4.state()
```

Let's make some plots.

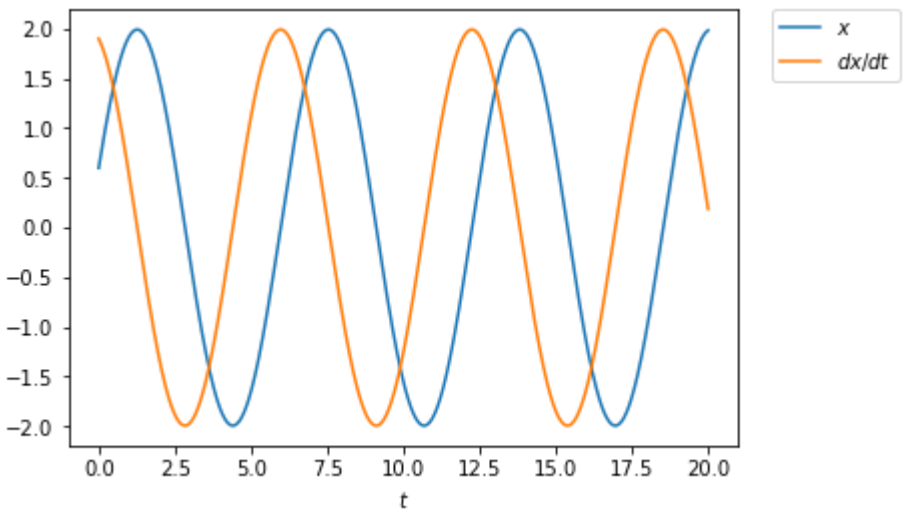
In [19]:

```
import matplotlib.pyplot as plt
%matplotlib inline

def time_plots(t,X,ylabel='',title='',legend=None):
    fig, ax = plt.subplots()
    for x in X: ax.plot(t,x)
    ax.set_title(title)
    ax.set_xlabel('$t$')
    ax.set_ylabel(ylabel)
    if legend: ax.legend(legend, bbox_to_anchor=(1.05,1), loc=2, borderaxespad=0
)
    return fig, ax
```

In [20]:

```
fig, ax = time_plots(t,[xl[0],xl[1]],legend=['$x$','$dx/dt$'])
```



TASK 1

Determine the analytical solution to the linear oscilator problem and analyse the error from the `RK4` integration as a function of time.

Well, the analytical solution is a sinusoid.

$$x = A\sin t + B\cos t$$
$$x' = A\cos t - B\sin t$$

You can then use the initial conditions to conclude that for the above question

$$x = \frac{19}{10}\sin t + \frac{6}{10}\cos t$$
$$x' = \frac{19}{10}\cos t - \frac{6}{10}\sin t$$

In [15]:

```
def linear_pendulum(t, x0=0, y0=0):
    """Solve linear pendulum."""
    return np.array([y0*np.sin(t) + x0*np.cos(t),
                     y0*np.cos(t) - x0*np.sin(t)])
```

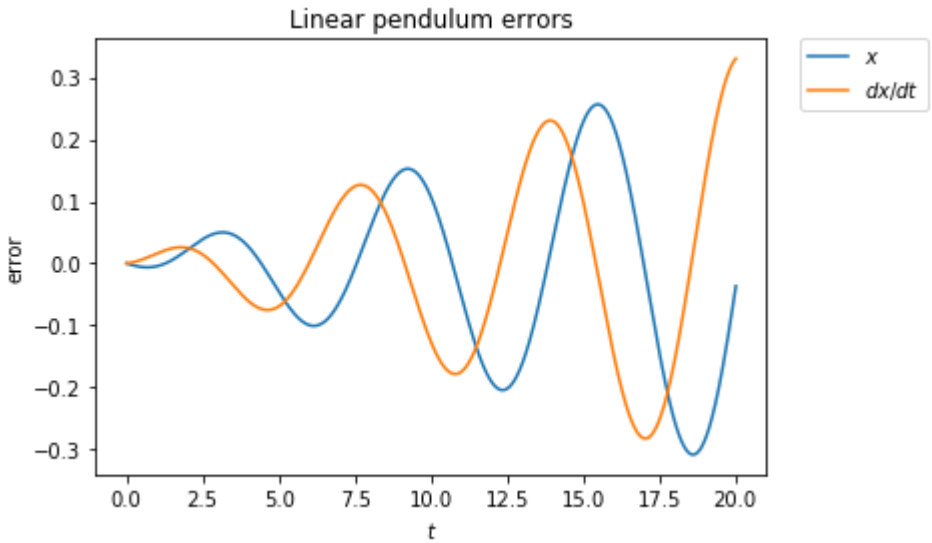
In [22]:

```
error = (xl - linear_pendulum(t,x0=x0,y0=y0))/(dt**4)

time_plots(t,error,ylabel='error',legend=['$x$','$dx/dt$'],title='Linear pendulu
m errors')
```

Out[22]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f876860>)
```



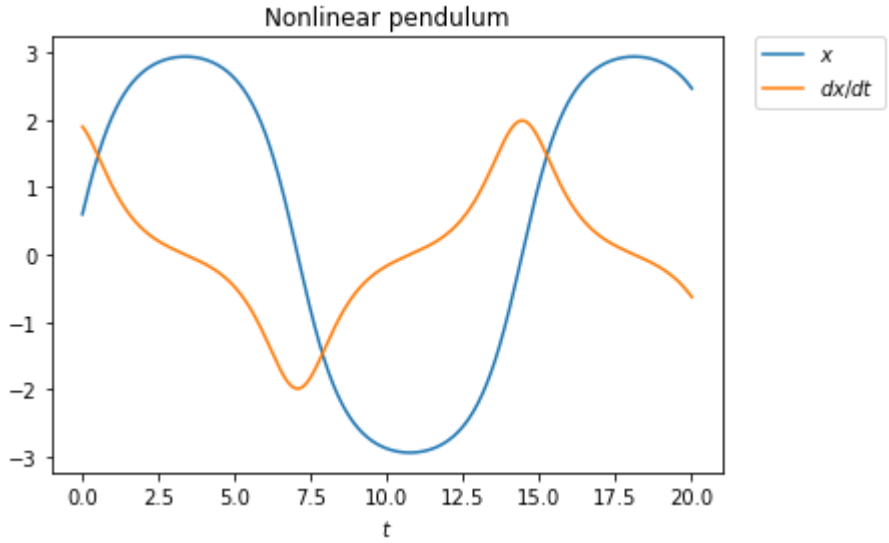
We can make a similar plot for the nonlinear pendulum.

In [23]:

```
time_plots(t,[xn[0],xn[1]],title='Nonlinear pendulum',legend=['$x$','$dx/dt$'])
```

Out[23]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f86d9b0>)
```



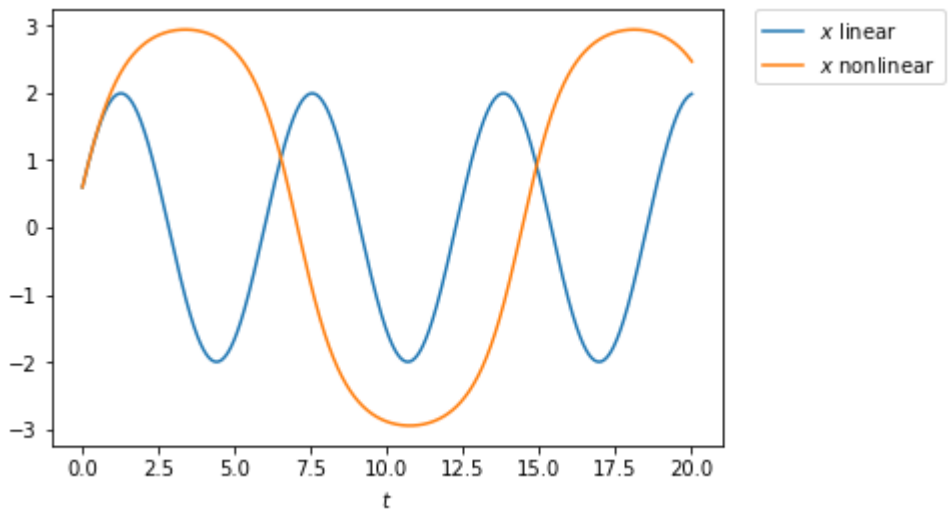
We can also compare the linear and nonlinear cases together,

In [24]:

```
time_plots(t,[xl[0],xn[0]],legend=['$x$ linear','$x$ nonlinear'])
```

Out[24]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f9b6320>)
```



In addition to simple plots of $x(t)$ and $x'(t)$ as a function of time, we can also make a ***Phase Portrait***: https://en.wikipedia.org/wiki/Phase_portrait (https://en.wikipedia.org/wiki/Phase_portrait)

This is one of the most useful tools for analysing dynamical systems. It's a parametric plot of $x'(t)$ ***versus*** $x(t)$**.** Very simple, but very powerful.

With higher-dimensional systems, we can take any two variables. We can also make a 3D picture also. We'll do that later.

In [25]:

```
def phase_portrait_2D(X,legend=None,title=''):
    fig, ax = plt.subplots()
    for x in X: ax.plot(x[0],x[1])
    ax.set_aspect('equal')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$dx/dt$')
    ax.set_title(title)
    if legend: ax.legend(legend, bbox_to_anchor=(1.05,1), loc=2, borderaxespad=0
)
    return fig, ax
```

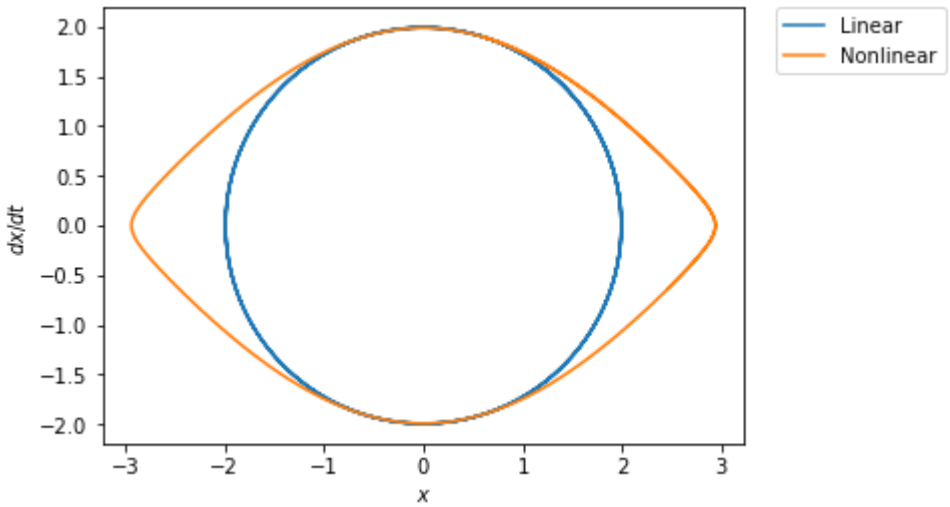
We can plot the linear and nonlinear cases on top of eachother.

In [26]:

```
phase_portrait_2D([xl,xn],legend=['Linear','Nonlinear'])
```

Out[26]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661fb7d5c0>)
```



We can now see the weird behaviour of the pendulum is just a warped circle.

We can do something similar in 3D

In [27]:

```
from mpl_toolkits.mplot3d import Axes3D

def phase_portrait_3D(X,title='',legend=None):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    for x in X: ax.plot(x[0],x[1],x[2], lw=2.5)
    ax.set_xlabel("$x$")
    ax.set_ylabel("$dx/dt$")
    ax.set_zlabel("$t$")
    ax.set_title(title)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])
    if legend: ax.legend(legend, bbox_to_anchor=(1.05,1), loc=2, borderaxespad=0
)
    return fig, ax
```

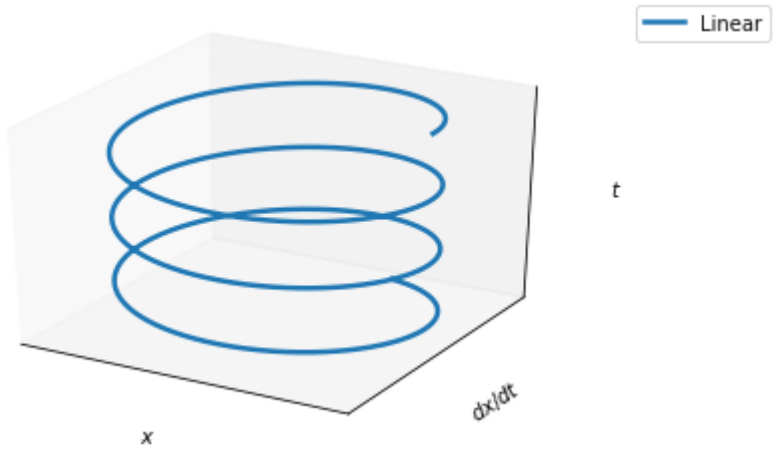
We can see that the linear case is a spiral if we include time as the 3rd dimension.

In [28]:

```
phase_portrait_3D([[ xl[0], xl[1], t ]], legend=['Linear'])
```

Out[28]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.Axes3DSubplot at 0x7f661f87e160>)
```



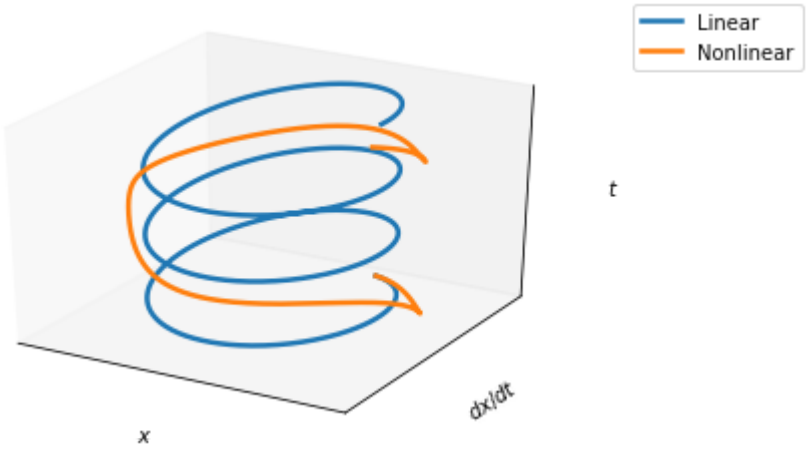
We can put the linear and nonlinear cases together and compare them. We can see that the nonlinear case takes longer to complete its trajectory.

In [29]:

```
phase_portrait_3D([[ xl[0], xl[1], t ] ,
                   [ xn[0], xn[1], t ]],
                  legend=['Linear','Nonlinear'])
```

Out[29]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.Axes3DSubplot at 0x7f661f702278>)
```



Conservation of energy

Both of these problems are ***conservative***. That means they have an ***energy*** function that doesn't change in time.

In the linear case

$$E = \frac{x_1^2 + x_0^2}{2}$$

In the nonlinear case

$$E = \frac{x_1^2}{2} - \cos x_0$$

Conservative systems are useful for testing time-stepping schemes. We can look and see how well they are keeping track of the energy.

In [30]:

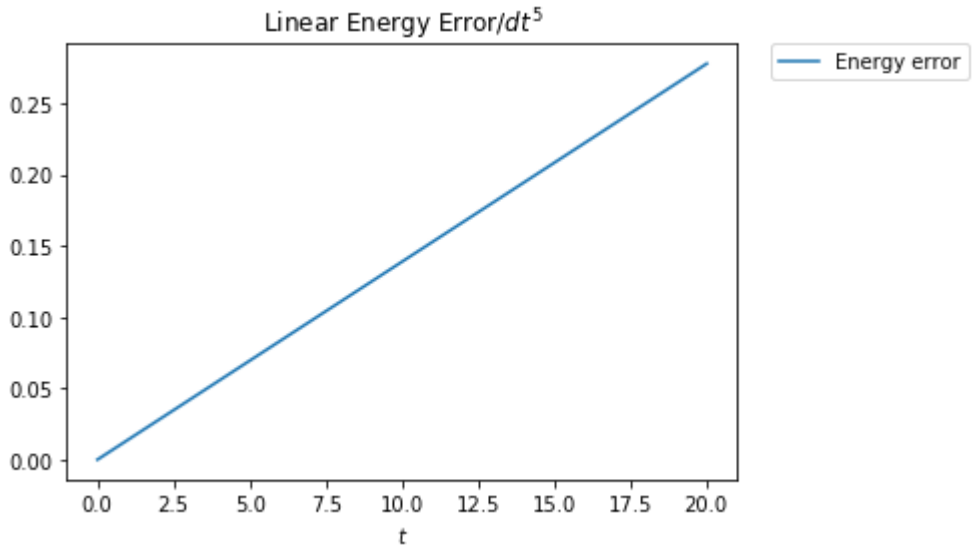
```
el = 0.5*(xl[1]**2 + xl[0]**2)
energy_error = abs((el - el[0])/el[0])
```

In [31]:

```
time_plots(t, [energy_error/(dt**5)], legend=['Energy error'],title='Linear Ener
gy Error/$dt^5$')
```

Out[31]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f6cbfd0>)
```



We can see that the energy is increasing over time, but the error is less than 10^{-9}. This is roughly $\mathcal{O}(h^5)$ for 20 time units of integration. So the energy does a little better than the solution itself. This is something to watch out for. The energy is helpful, but ***just because the energy is conserved, doesn't guarantee the solution is as accurate.***

TASK 2

Make the same analysis for the nonlinear case.

In [32]:

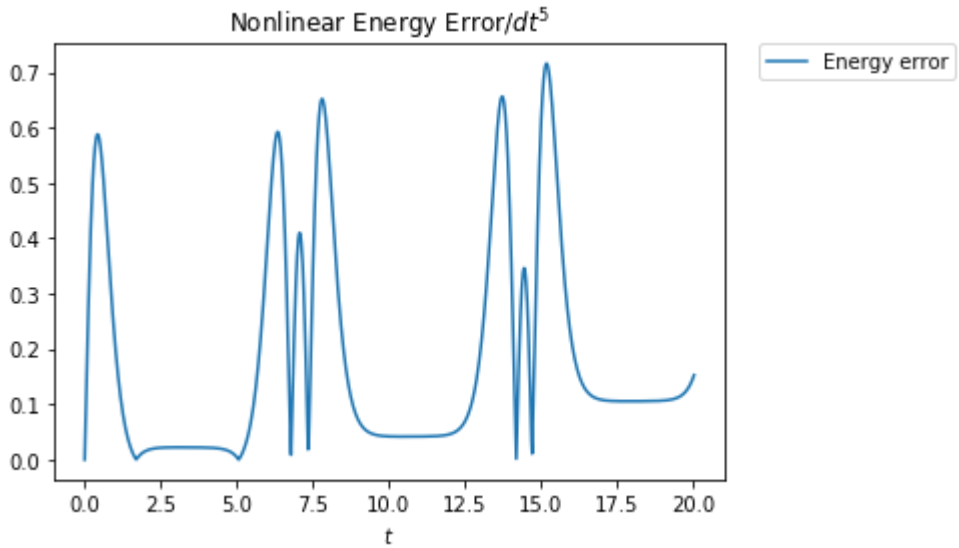
```
en = 0.5*(xn[1]**2) - np.cos(xn[0])
energy_error = abs((en - en[0])/en[0])
```

In [33]:

```
time_plots(t, [energy_error/(dt**5)], legend=['Energy error'],title='Nonlinear E
nergy Error/$dt^5$')
```

Out[33]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f9c7080>)
```



All of this was for the 4th-order Runge-Kutta scheme. The is only one of many similar methods. Another common example is the 2nd-order RK2, and the 1st-order Euler method.

The **Euler method** is the absolute simplest:

$$x(t+h) = x(t) + hf(x(t))$$

The error for this scheme is $\mathcal{O}(h^2)$ after one time step, and $\mathcal{O}(h)$ after the error accumulates.

TASK 3

Adapt the `RK4` class to make an new `Euler` class that computes the Euler method.

In [34]:

```
class Euler():

    def __init__(self,start,forcing,time=0):
        self.time = [time]
        self.data = [start]
        self.RHS  = forcing

    def __call__(self,dt):

        self.time += [ self.time[-1] + dt ]

        x = self.data[-1]
        f = self.RHS

        self.data += [ x + dt*f(x) ]

    def state(self):
        return np.array(self.time), np.array(self.data).T

    def now(self):
        return self.time[-1]
```

It turns out that there are quite a few 2nd-order Runge-Kutta type schemes. A common one also goes by the name ***Heun's method*** : https://en.wikipedia.org/wiki/Heun's_method (https://en.wikipedia.org/wiki/Heun's_method)

In the same style as our RK4:

$$i_1 = hf(x(t)), \quad i_2 = hf(x(t) + i_1)$$

$$x(t+h) = x(t) + \frac{i_1 + i_2}{2}$$

We call time steppers ***integrators*** becuase if the right-hand side function *only* depends on time

$$\frac{dI}{dt} = f(t)$$

Then the result of time stepping will be some approximation to the integral

$$I(t) = \int_0^t f(s)\,\mathrm{d}s$$

TASK 4

Show that `RK4` is equivilent to Simpson's method, and Heun's method is equivilent to the trapazoidal rule for $f(t,x) = f(t)$ only.

Consider Simpson's rule on a function $f(t)$ for a single step from $t = 0$ to $t = h$.

We have

$$\int_t^{t+h} f(t)dt \approx \frac{h}{6}(f(t) + 4f(t + h/2) + f(t + h))$$

Now, consider integrating the ODE

$$x'(t) = f(t)$$

from t to $t + h$. Then,

$$x(t + h) - x(t) = \int_t^{t+h} f(t)dt$$

Using RK4 calculation, where $f(t)$ only, you calculate:

$$\begin{aligned} k_1 &= f(t,\ x(t)) \\ k_2 &= f(t + \frac{h}{2}) \\ k_3 &= f(t + \frac{h}{2}) \\ k_4 &= f(t + h) \\ x(t + h) &= x(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ \implies x(t + h) - x(t) &= \frac{h}{6}\left(f(t) + 4f(t + \frac{h}{2}) + f(t + h)\right) \end{aligned}$$

This is for a single step. Obviously it's true when you do multiple steps after each other.

So RK4 is equivalent to Simpson's method when integrating a function of time!

The analysis for Heun's method is very similar. **For you to complete**.

TASK 5

Show that the `Euler` method is equivilent to a Riemann sum for if $f(t, x) = f(t)$ only.

The Euler method is actually the only example of an `RK1` scheme, but it gets its own name because it's the Granddaddy of the all.

Again, **you can complete this**.

Check what the Riemann sum is, and evaluate what the Euler integration step gives you.

TASK 6

Adapt the `Euler` and/or `RK4` class to make a class that computes the `RK2` scheme (Heun's method)

In [35]:

```
class RK2():

    def __init__(self,start,forcing,time=0):
        self.time = [time]
        self.data = [start]
        self.RHS  = forcing

    def __call__(self,dt):
        self.time += [ self.time[-1] + dt ]
        x = self.data[-1]
        f = self.RHS
        i1 = dt*f(x)
        i2 = dt*f(x+i1)
        self.data += [ x + 0.5*(i1+i2) ]

    def state(self):
        return np.array(self.time), np.array(self.data).T

    def now(self):
        return self.time[-1]
```

TASK 6

Use the `Euler` and `RK2` classes to solve the linear and nonlinear problems and compare the results to the `RK4` benchmark.

This makes the different time steppers with the different right-hand side functions.

In [36]:

```
linear_E1    = Euler(initial_data,L)
nonlinear_E1 = Euler(initial_data,P)

linear_RK2    = RK2(initial_data,L)
nonlinear_RK2 = RK2(initial_data,P)
```

This advances the states to 20 time units.

In [37]:

```
while linear_E1.now() < 20:
    linear_E1(dt)
    nonlinear_E1(dt)
    linear_RK2(dt)
    nonlinear_RK2(dt)
```

This gets all the different states.

In [38]:

```
t, xl_4 = linear_RK4.state()
t, xn_4 = nonlinear_RK4.state()

t, xl_2 = linear_RK2.state()
t, xn_2 = nonlinear_RK2.state()

t, xl_1 = linear_E1.state()
t, xn_1 = nonlinear_E1.state()
```

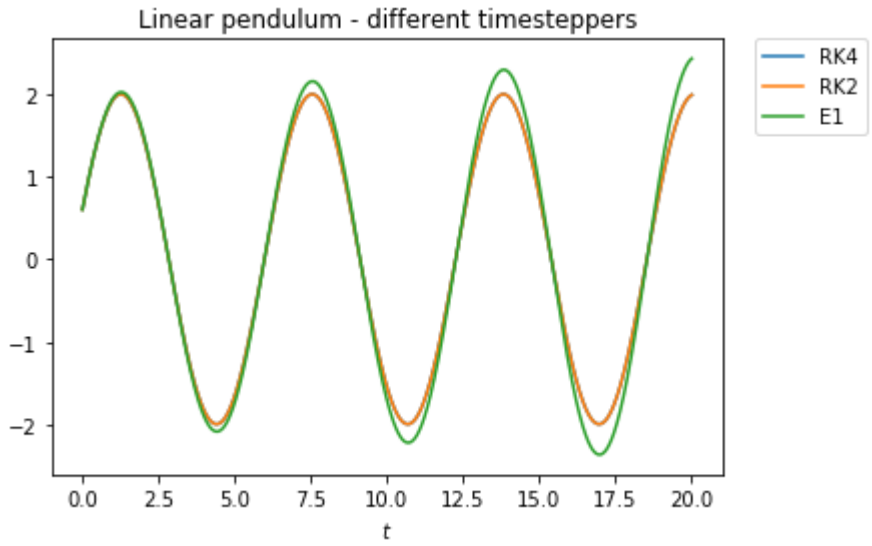
We can look at them on top of eachother:

In [39]:

```
time_plots(t,[xl_4[0], xl_2[0],xl_1[0]],legend=['RK4','RK2','E1'],title='Linear
 pendulum - different timesteppers')
```

Out[39]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f5bc8d0>)
```



We can see that the `Euler` scheme drifts quite a lot over time.

We can look at the differences scaled by `dt**2` and `dt` respectively.

In [40]:

```
xl_ans = linear_pendulum(t, x0=x0,y0=y0)
```

In [41]:

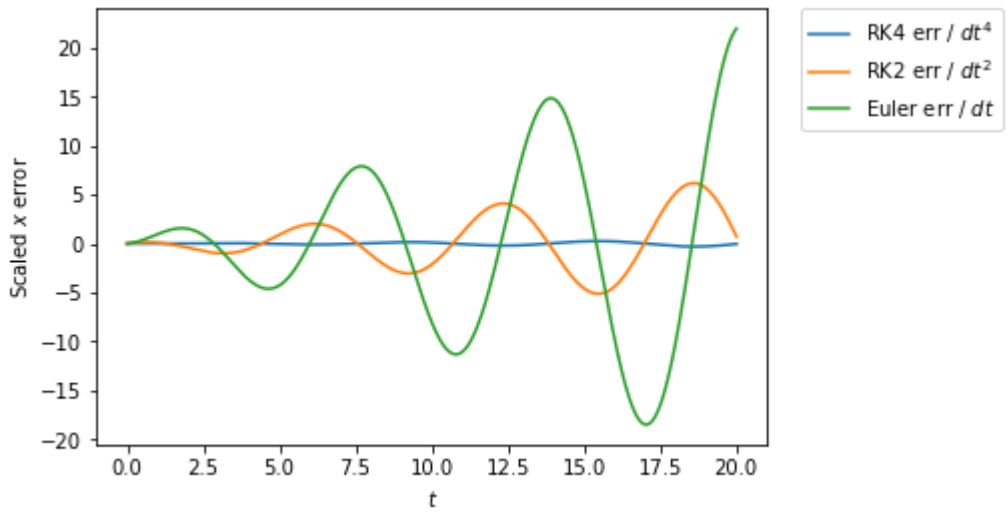
```
el_4 = (xl_4[0] - xl_ans[0])/dt**4
el_2 = (xl_2[0] - xl_ans[0])/dt**2
el_1 = (xl_1[0] - xl_ans[0])/dt
```

In [42]:

```
time_plots(t,[el_4, el_2, el_1],ylabel='Scaled $x$ error',
           legend=['RK4 err / $dt^4$','RK2 err / $dt^2$','Euler err / $dt$'])
```

Out[42]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f61bb00>)
```



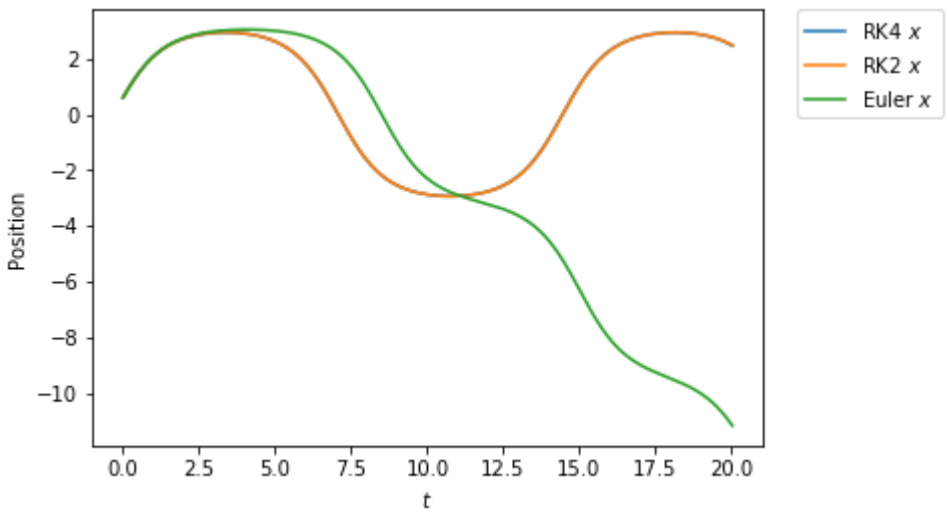
We can see the results are even clearer for the nonlinear case

In [43]:

```
time_plots(t,[xn_4[0], xn_2[0],xn_1[0]],
           ylabel='Position',
           legend=['RK4 $x$','RK2 $x$','Euler $x$'])
```

Out[43]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661fb6dac8>)
```



A more fun example is the **Van der Pol oscillator (https://en.wikipedia.org/wiki/Van_der_Pol_oscillator)**:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0$$

TASK 7: Van der Pol

Adapt the `Pendulum` class and make a new class to accomadate the Van der Pol equation. This should take a parameter μ and give back a pure function that is the first-order formulation right-hand side.

In [44]:

```
class Van_der_Pol():
    def __init__(self,mu):
        self.mu    = mu

    def __call__(self,x):
        mu    = self.mu
        f0 =  x[1]
        f1 = -x[0] + mu*(1-x[0]**2)*x[1]
        return np.array( [ f0, f1 ] )
```

TASK 8

Investigate this system for different values of μ. Using the `RK4` integrator.

In [45]:

```
V01 = Van_der_Pol(1)
V10 = Van_der_Pol(10)
VPO_01_RK4 = RK4( initial_data , V01 )
VPO_10_RK4 = RK4( initial_data , V10 )
```

In [46]:

```
while VPO_01_RK4.now() < 20:
    VPO_01_RK4(dt)
    VPO_10_RK4(dt)
```

In [47]:

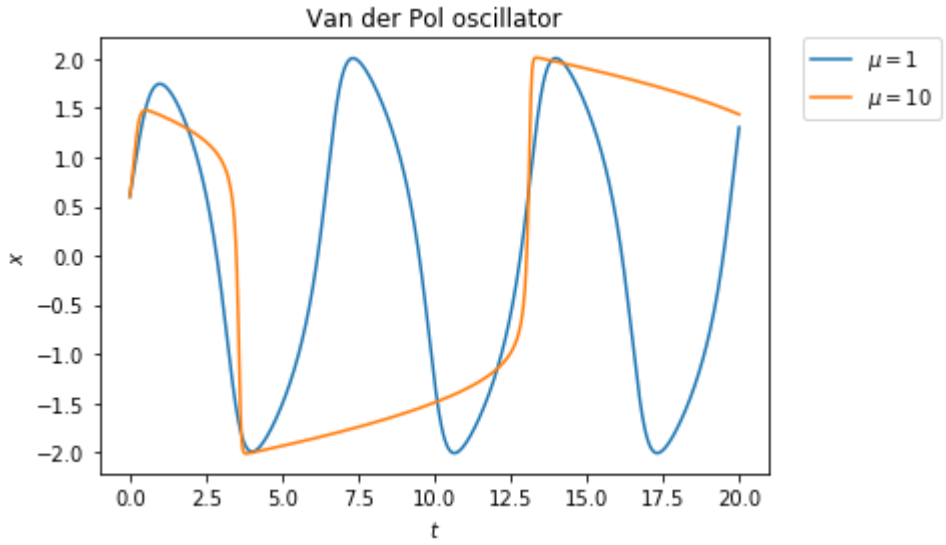
```
t, xv_01 = VPO_01_RK4.state()
t, xv_10 = VPO_10_RK4.state()
```

In [48]:

```
time_plots(t,[xv_01[0],xv_10[0]],
           ylabel='$x$',
           title='Van der Pol oscillator',
           legend=['$\mu = 1$','$\mu = 10$'])
```

Out[48]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f40e7b8>)
```



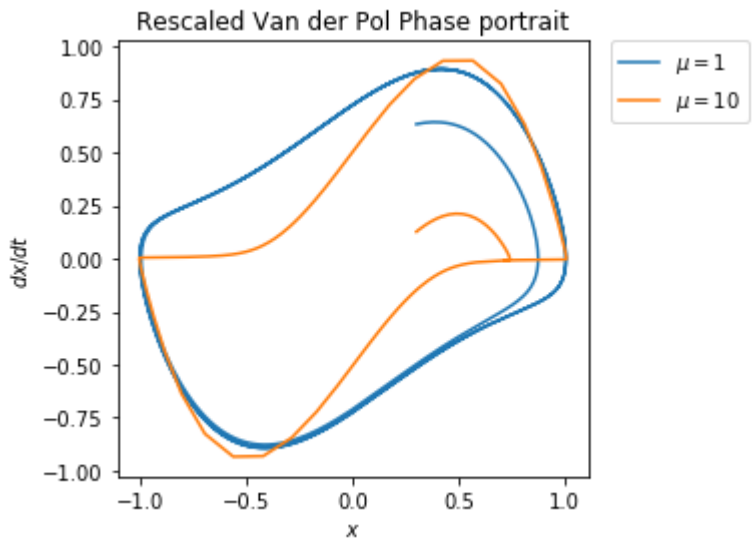
This is a prime example of a "fast-slow" system. As μ becomes larger, the system is able to change very rapidly. But only for shot bursts.

In [49]:

```
phase_portrait_2D([[xv_01[0]/2,xv_01[1]/3],[xv_10[0]/2,xv_10[1]/15]],
                  legend=['$\mu = 1$','$\mu = 10$'],
                  title = 'Rescaled Van der Pol Phase portrait')
```

Out[49]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f661f3721d0>)
```



TASK 9: Lorenz

Do all the same analysis with the 3D **Lorenz system (https://en.wikipedia.org/wiki/Lorenz_system)**

$$\frac{dx}{dt} = s\,(y - x)$$

$$\frac{dy}{dt} = x\,(r - z) - y$$

$$\frac{dz}{dt} = x\,y - b\,z$$

For parameters s, r, and b.

In [50]:

```
class Lorenz():

    def __init__(self,s,r,b):
        self.s    = s
        self.r    = r
        self.b    = b

    def __call__(self,x):

        s    = self.s
        r    = self.r
        b    = self.b
        f0 =  s*(x[1]-x[0])
        f1 =  x[0]*(r-x[2]) - x[1]
        f2 =  x[0]*x[1] - b*x[2]

        return np.array( [ f0, f1, f2 ] )
```

In [51]:

```
LNZ     = Lorenz(10,28,8/3)
LNZ_RK4 = RK4( np.array([0.6,1.9,2.3]) , LNZ )
```

In [52]:

```
while LNZ_RK4.now() < 20:
    LNZ_RK4(0.1*dt)
```

In [53]:

```
t, x = LNZ_RK4.state()
```

In [54]:

```
phase_portrait_3D([x ], title='Lorenz Butterfly')
```

Out[54]:

```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.Axes3DSubplot at 0x7f661f2e5a20>)
```