Programming

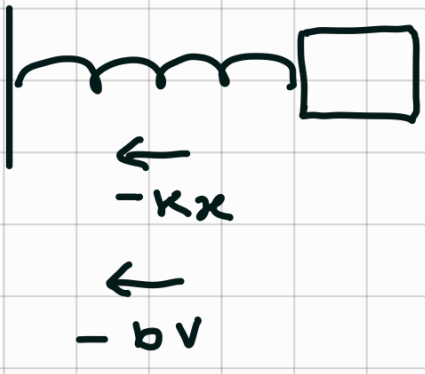
Oscillations

Solve - IVP:

To solve second-order differential IVP's, we can consider the vector fields they produce in state-space.

To do a concrete example, let us consider the damped harmonic oscillator.

We can form a second order ODE by considering the forces on our mass.



$$F_{res} = ma$$

$$-kx - bv = ma$$

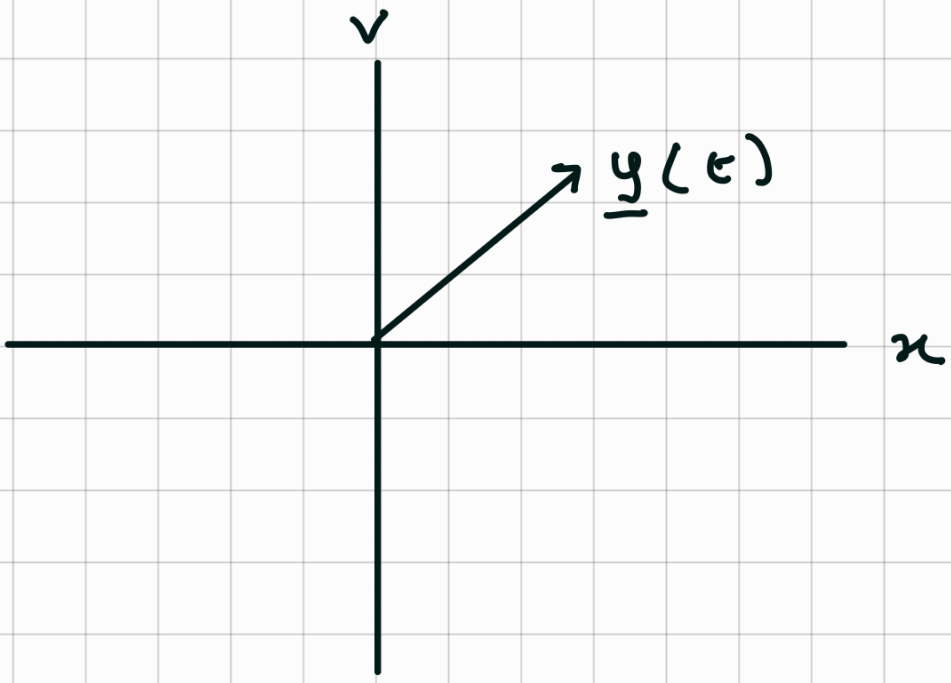
$$\therefore ma + bv + kx = 0$$

$$\frac{d^2x}{dt^2} + \frac{b}{m}v + \frac{k}{m}x = 0$$

Every possible state our pendulum is can be represented by a combination of x and v, which we can package into a vector.

$$\underline{y}(t) = \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

This then forms a space, known as the state space, or phase space:

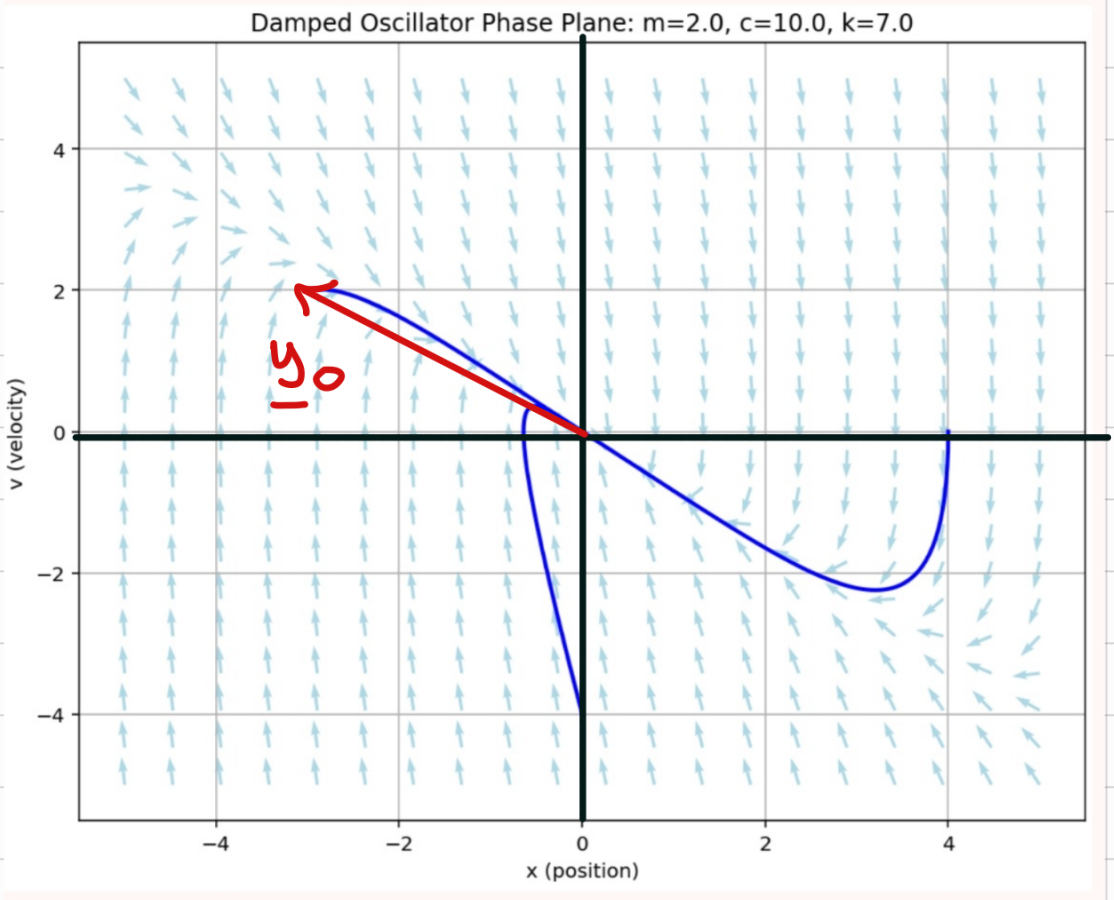


we can then find $\frac{d\underline{y}(t)}{dt}$, to form a vector field.

$$\frac{d\underline{y}(t)}{dt} = \begin{bmatrix} \frac{d\underline{x}(t)}{dt} \\ \frac{d\underline{v}(t)}{dt} \end{bmatrix}$$

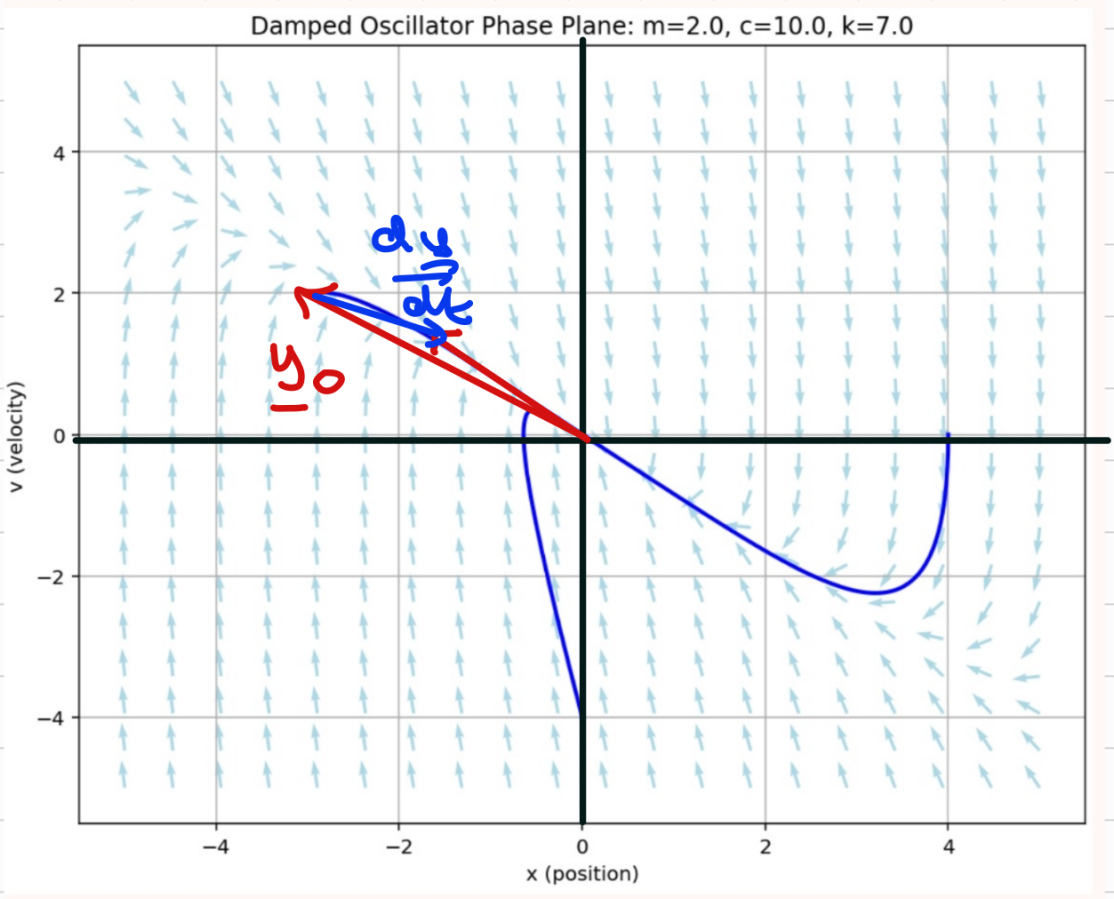
$$= \begin{bmatrix} v \\ a \end{bmatrix} = \begin{bmatrix} v \\ -\frac{b}{m}v - \frac{k}{m}x \end{bmatrix}$$

This vector field tells us where a given state vector wants to move.



Hence, by choosing a certain $\underline{y}_0$, we can find the analytical solution by finding an expression for how $\underline{y}(t)$ evolves.

This also means we can numerically determine the evolution of $\underline{y}(t)$, so find how $x(t)$ and $v(t)$ evolve.



To find an estimate for $\underline{y}(t + \Delta t)$

$$\underline{y}(t + \Delta t) \approx \underline{y}_0 + \frac{d\underline{y}}{dt}\Delta t$$

Δt: some time step

Hence:

$$\underline{y}(t) \approx \sum_{\text{start}}^{\text{end}} \underline{y}_0 + \frac{d\underline{y}}{dt}\Delta t$$

Δt: interval. (small is more accurate).

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

fig, axes = plt.subplots(3,1, figsize = [4,4], layout = 'constrained')

# input
alpha = 3
k = 3
m = 2
x0 = 1
v0 = 0

y0 = [x0, v0]

def damped_harmonic_oscillator(t, y):
    dxdt = y[1]
    dvdt = -(alpha/m)*y[1] - (k/m)*y[0]
    return dxdt, dvdt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

sol = solve_ivp(damped_harmonic_oscillator, t_range, y0, t_eval = t_solve)

for i in range(0, 3, 1):
    axes[i].grid("true")

axes[0].plot(sol.t, sol.y[0], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes[0].set_xlabel('Time [s]')
axes[0].set_ylabel('Displacement [m]')
axes[0].set_title('Distance/Time Graph');

axes[1].plot(sol.t, sol.y[1], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes[1].set_xlabel('Time [s]')
axes[1].set_ylabel('Velocity [ms^-1]')
axes[1].set_title('Velocity/Time Graph');

axes[2].plot(sol.y[0], sol.y[1], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes[2].set_xlabel('Displacement [m]')
axes[2].set_ylabel('Velocity [ms^-1]')
axes[2].set_title('Phase-Space Graph');
```

integrate as we are doing a sum.

define parameters

define y_0

define a function which returns $\frac{dy}{dt}$ for a given $y(t)$.

range you do it

determines size of interval

evaluates the sum at your times.

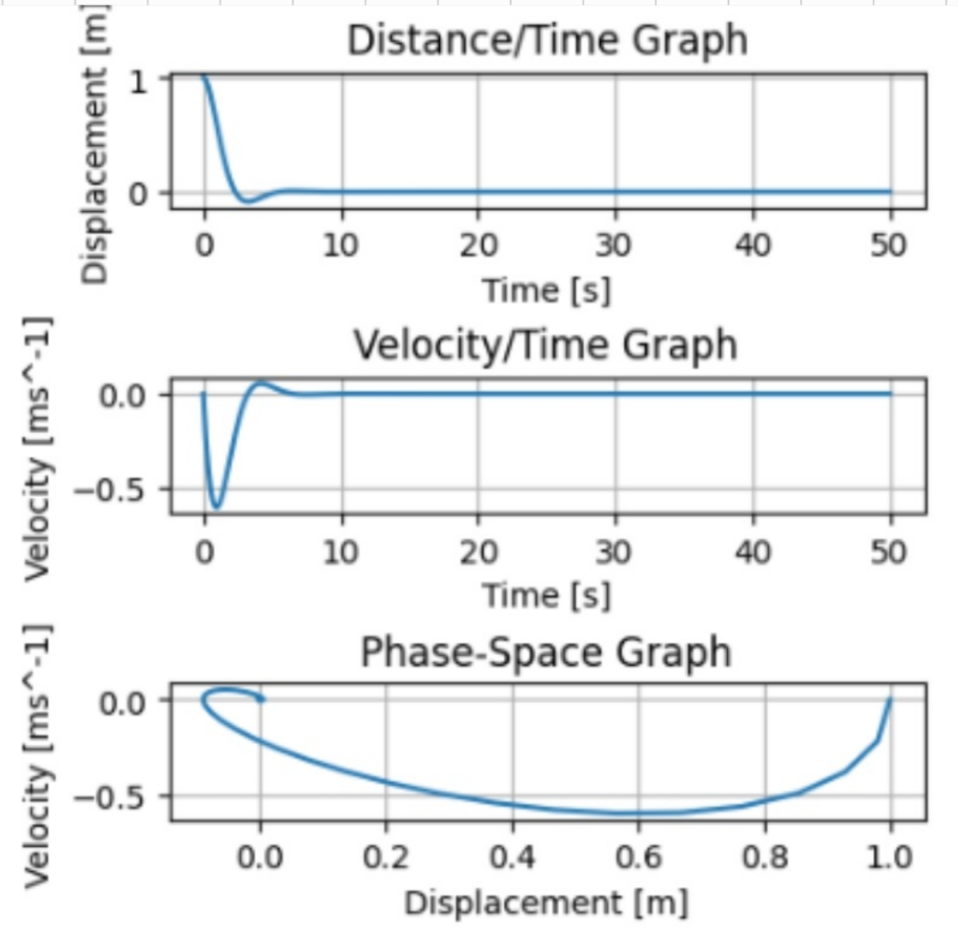
t-values solved at.

state space path.

sol.y is the vector which contains the 'path' you solved for. sol.y[0] returns the first row, which is $x(t)$, and sol.y[1] would give $v(t)$, stored in the second row.

$$sol.y = \begin{bmatrix} x(t_0) & x(t_1) & x(t_2) & \dots \\ v(t_0) & v(t_1) & v(t_2) & \dots \end{bmatrix}$$

$$sol.t = \begin{bmatrix} t_0 & t_1 & t_2 & \dots \end{bmatrix}$$



These are the graphs produced.

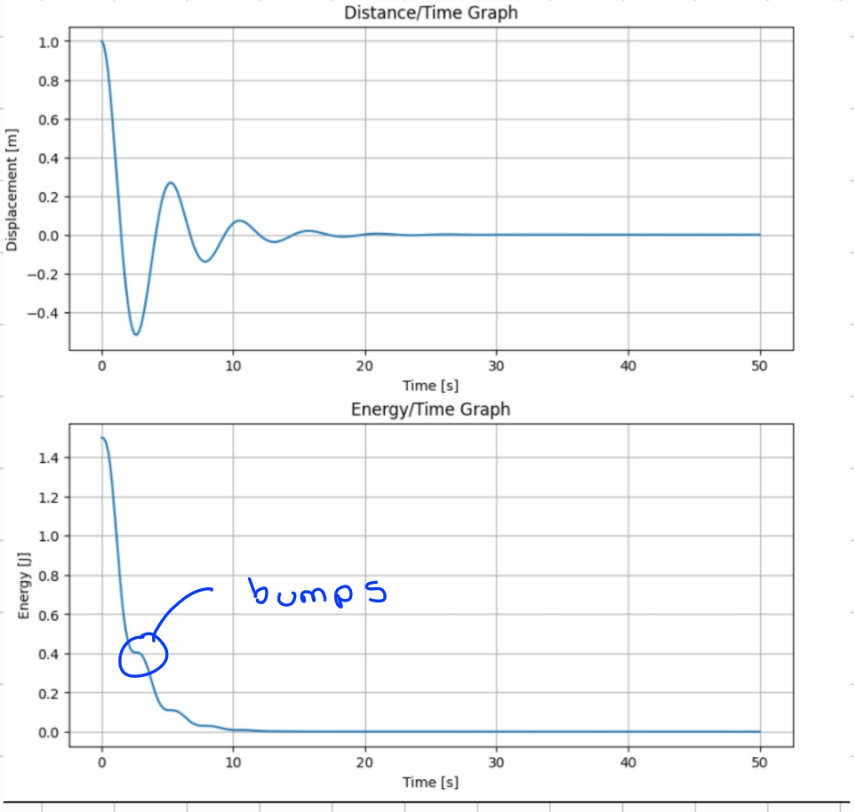
Energy of DHO:

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

fig, axes = plt.subplots(2,1, figsize = [8,8], layout = 'constrained')

# input
alpha = 1
k = 3
m = 2
x0 = 1
v0 = 0

y0 = [x0, v0]

def damped_harmonic_oscillator(t, y):
    dxdt = y[1]
    dvdt = -(alpha/m)*y[1] - (k/m)*y[0]
    return dxdt, dvdt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

sol = solve_ivp(damped_harmonic_oscillator, t_range, y0, t_eval = t_solve)

for i in range(0, 2, 1):
     axes[i].grid("true")

axes[0].plot(sol.t, sol.y[0], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes[0].set_xlabel('Time [s]')
axes[0].set_ylabel('Displacement [m]')
axes[0].set_title('Distance/Time Graph');

def E(x, v):
    return 1/2*m*v**2 + 1/2*k*x**2

axes[1].plot(sol.t, E(sol.y[0], sol.y[1]), label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes[1].set_xlabel('Time [s]')
axes[1].set_ylabel('Energy [J]')
axes[1].set_title('Energy/Time Graph');

```

← define E(x,v) with x and v inputs.

put in x(t) and v(t).

← plot graph.



Note how graph looses energy due to damping.

Bumps → due their being no damping when v = 0.

Event Detection

Events are ways to track when certain things happen in your system.

The way it works is you define a function includes your state, e.g

```
def x_crossing(t, y)
    return y[0].
```

when puting:

```
events = x_crossing
```

As an arguement into solve_ivp, the computer passes sol.y into the y - parameter of our function.

When sol.y[0] (i.e x(t)) changes sign, the computer will mark this as an event.

Hence, the above snippet makes the computer store x-crossing times in a list called:

sol.t_events.

i.e

sol.t_events[0] gives times for the first defined event (which may be the only one, if you only have one.

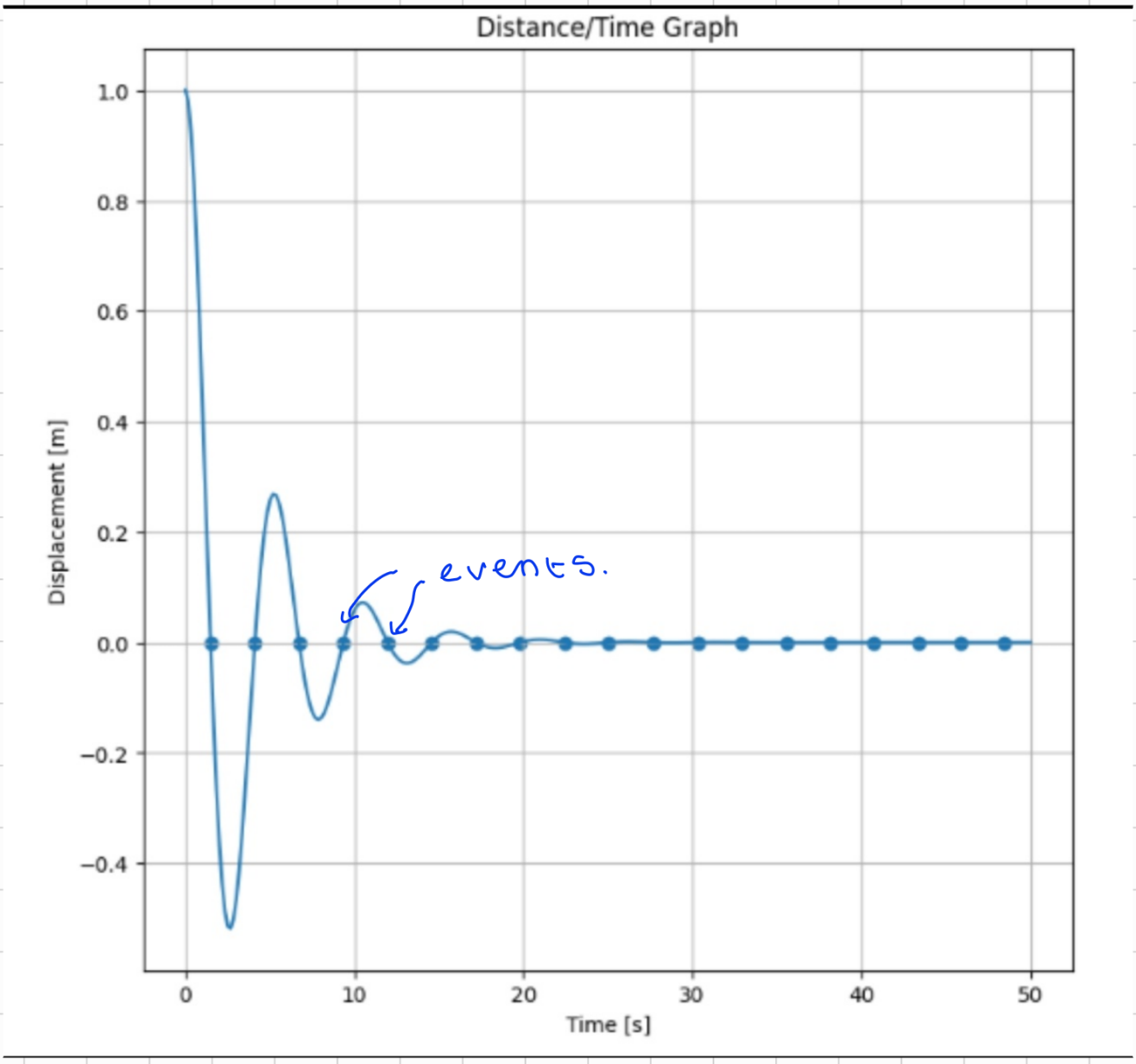
sol.t_events[1] second event and so on...

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

fig, axes = plt.subplots(figsize = [8,8])
axes.grid("true")
axes.set_xlabel("Times [s]")
axes.set_ylabel("Displacement [m]")

# input
alpha = 1
k = 3
m = 2
x0 = 1
v0 = 0

y0 = [x0, v0]


def damped_harmonic_oscillator(t, y):
    dxdt = y[1]
    dvdt = -(alpha/m)*y[1] - (k/m)*y[0]
    return dxdt, dvdt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

def x_crossing(t, y):
    return y[0]

sol = solve_ivp(damped_harmonic_oscillator, t_range, y0, t_eval = t_solve, events = x_crossing)

axes.plot(sol.t, sol.y[0], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes.set_xlabel('Time [s]')
axes.set_ylabel('Displacement [m]')
axes.set_title('Distance/Time Graph');
plt.scatter(sol.t_events[0], np.zeros_like(sol.t_events[0]));


```

event.

event.

plot events as scatter.



We can make events more specific.

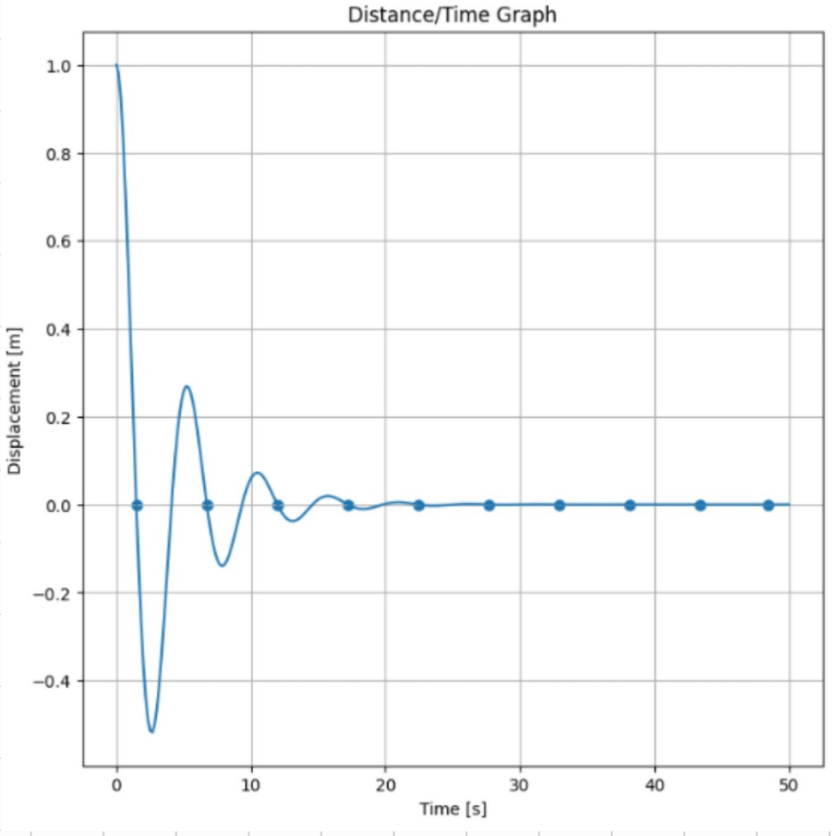
One thing we can do is put rules on that sign change.

'x_crossing.direction = -1'

Makes the computer only count positive→negative crossings.

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

fig, axes = plt.subplots(figsize = [8,8])
axes.grid("true")
axes.set_xlabel("Times [s]")
axes.set_ylabel("Displacement [m]")

# input
alpha = 1
k = 3
m = 2
x0 = 1
v0 = 0

y0 = [x0, v0]


def damped_harmonic_oscillator(t, y):
    dxdt = y[1]
    dvdt = -(alpha/m)*y[1] - (k/m)*y[0]
    return dxdt, dvdt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

def x_crossing(t, y):
    return y[0]

x_crossing.direction = -1

sol = solve_ivp(damped_harmonic_oscillator, t_range, y0, t_eval = t_solve, events = x_crossing)

axes.plot(sol.t, sol.y[0], label = f"m = {m}, k = {k}, alpha = {alpha}, x0 = {x0}, v0 = {v0}")
axes.set_xlabel('Time [s]')
axes.set_ylabel('Displacement [m]')
axes.set_title('Distance/Time Graph');
plt.scatter(sol.t_events[0], np.zeros_like(sol.t_events[0]));
```

note, must be outside function body.



wonderful!

You can also make the first event detection stop / terminate the integration.

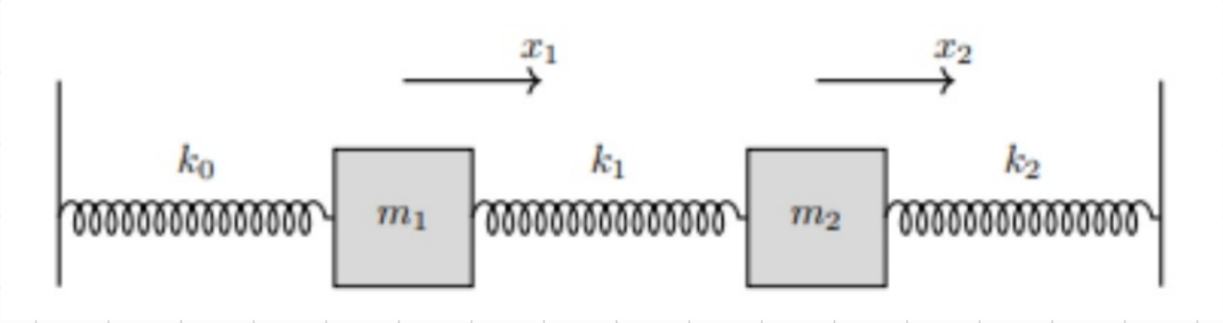
(could be useful for, say, stopping a ball going through the floor when modelling SUVAT).

'x_crossing.terminal = True'

Coupled - Oscillators:

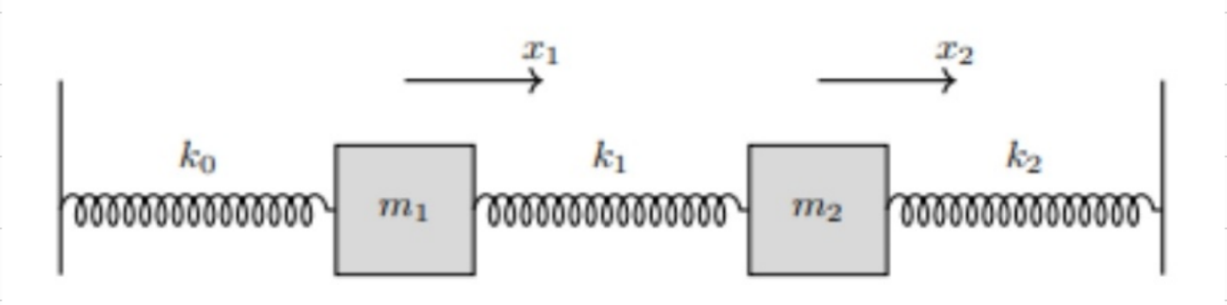
The use of computing programs is more clear when dealing with systems with more complecated systems.

For example, take 2 coupled oscillators.



we have two methods on how to handle this. One which is easier, and the other which is more generalisable to n-masses, rather than 2.

Method 1:



Since we have two masses, our state vector, which must represent all possible states, must have $x_1(t)$ $v_1(t)$, and $x_2(t)$ and $v_2(t)$.

We can package this information in the vector in any order we want, we will package it this way:

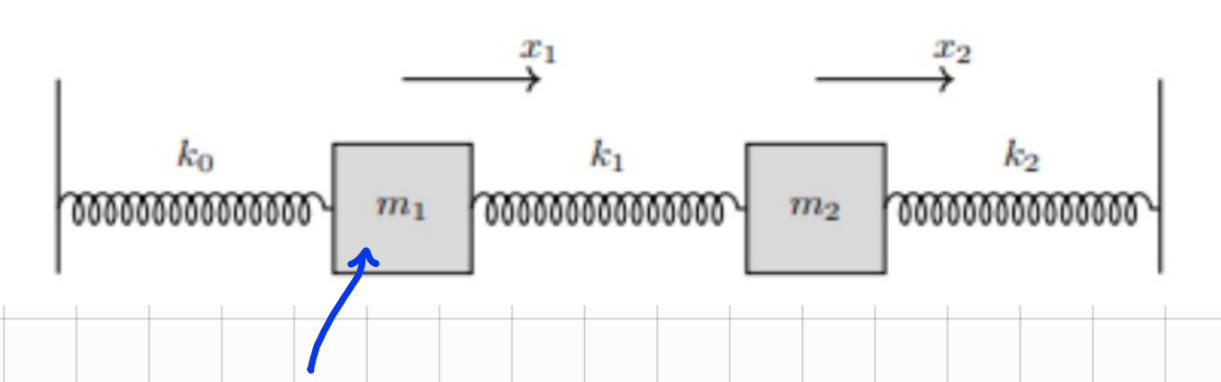
$$\underline{y} = \begin{bmatrix} x_1(t) \\ x_2(t) \\ v_1(t) \\ v_2(t) \end{bmatrix}$$

To do our solve.ivp process, we still need:

$$\frac{d\vec{y}}{dt} = \begin{bmatrix} v_1(t) \\ v_2(t) \\ a_1(t) \\ a_2(t) \end{bmatrix}$$

to get a_1 and a_2, we just need to look at the equations of motion for our system.

This system is not damped, so:



$$F_{res} = ma$$

$$-k_0 x_1 + k_1(x_2 - x_1) = m_1 a_1$$

Likewise

$$-k_2 x_2 - k_1(x_2 - x_1) = m_2 a_2$$

Hence

$$a_1 = \frac{1}{m_1}\left(-k_0 x_1 + k_1(x_2 - x_1)\right)$$

$$a_2 = \frac{1}{m_2}\left(-k_2 x_2 - k_1(x_2 - x_1)\right)$$

This is all we need, as well as an intial $\underline{y_0}$.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
fig, ax = plt.subplots(2,1, figsize = (8,8), layout = 'constrained')

# input values
m1 = 2
m2 = 2
k0 = 2
k1 = 3
k2 = 4
x0_1 = 1
x0_2 = 2
v0_1 = 3
v0_2 = 4
y0 = [x0_1, x0_2, v0_1, v0_2]

def coupled_oscillator(t, y):
    v1 = y[2]
    v2 = y[3]
    x1 = y[0]
    x2 = y[1]

    dx1dt = v1
    dx2dt = v2
    dv1dt = (1/m1)*(-k0 * x1 + k1 * (x2 - x1))
    dv2dt = (1/m2)*(-k2 * x2 - k1 * (x2 - x1))
    return dx1dt, dx2dt, dv1dt, dv2dt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

sol = solve_ivp(coupled_oscillator, t_range, y0, t_eval = t_solve)

ax[0].plot(sol.t, sol.y[0], label = 'Mass 1');
ax[0].plot(sol.t, sol.y[1], label = 'Mass 2');
ax[0].set_xlabel("t [s]")
ax[0].set_ylabel("displacement [m]");
ax[0].set_title("Displacement/Time Graph");
ax[0].legend()
ax[0].grid("true")

ax[1].plot(sol.t, sol.y[2], label = 'Mass 1');
ax[1].plot(sol.t, sol.y[3], label = 'Mass 2');
ax[1].set_xlabel("t [s]")
ax[1].set_ylabel("velocity [ms^-1]");
ax[1].set_title("Velocity/Time Graph");
ax[1].legend()
ax[1].grid("true")
```

← $\underline{y_0}$

← define $\frac{d\underline{y}}{dt}$

Just calling elements of $\underline{y}$ something to make easier.

solves as usual.

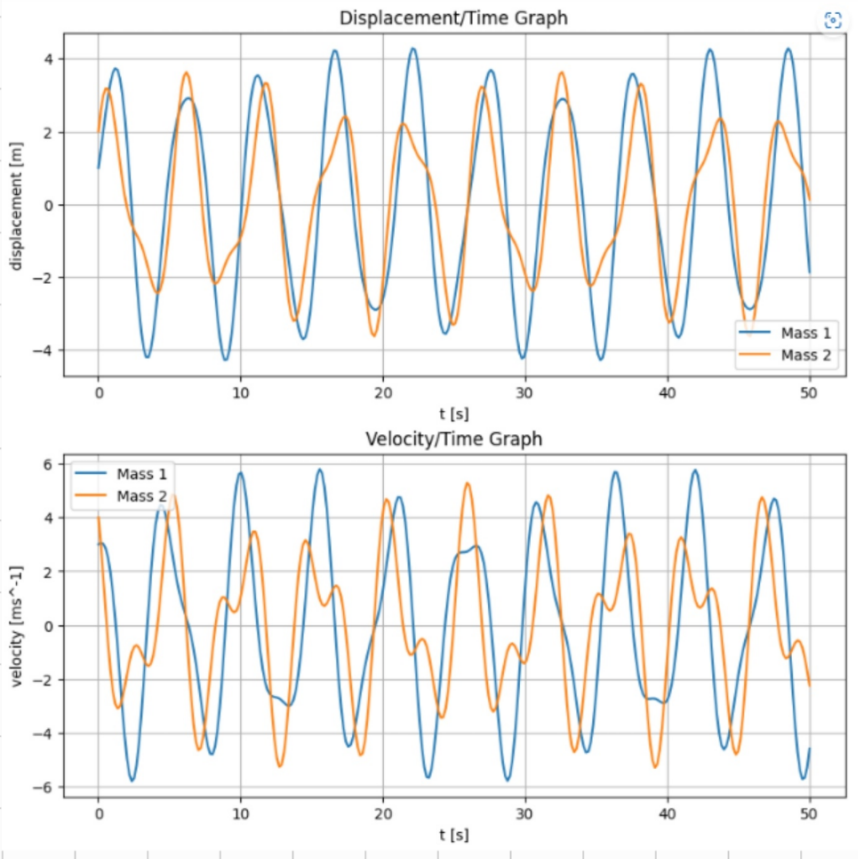
As expected, sol.y will have shape:

$$sol.y = \begin{bmatrix} x_1(t_0) & x_1(t_1) & x_1(t_2) & \dots \\ x_2(t_0) & x_2(t_1) & x_2(t_2) & \dots \\ v_1(t_0) & v_1(t_1) & v_1(t_2) & \dots \\ v_2(t_0) & v_2(t_1) & v_2(t_2) & \dots \end{bmatrix}$$

and

$$sol.t = \begin{bmatrix} t_0 & t_1 & t_2 & \dots \end{bmatrix}$$

Hence, plotting sol.y[0] and sol.y[1] against sol.t, and sol.y[2] and sol.y[3] against sol.t, produces:



If we want total energy, we can follow a similar scheme as before, which is defining a function for energy in terms of x and v, and making the function take in sol.y elements as inputs:

$$E(t) = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 + \frac{1}{2} k_0 x_1^2 + \frac{1}{2} k_1 (x_2 - x_1)^2 + \frac{1}{2} k_2 x_2^2$$

A little unwieldy, but it should work.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
fig, ax = plt.subplots(3,1, figsize = (8,8), layout = 'constrained')

# input values
m1 = 2
m2 = 2
k0 = 2
k1 = 3
k2 = 4
x0_1 = 1
x0_2 = 2
v0_1 = 3
v0_2 = 4
y0 = [x0_1, x0_2, v0_1, v0_2]


def coupled_oscillator(t, y):
    v1 = y[2]
    v2 = y[3]
    x1 = y[0]
    x2 = y[1]

    dx1dt = v1
    dx2dt = v2
    dv1dt = (1/m1)*(-k0 * x1 + k1 * (x2 - x1))
    dv2dt = (1/m2)*(-k2 * x2 - k1 * (x2 - x1))
    return dx1dt, dx2dt, dv1dt, dv2dt

t_range = [0, 50]
t_solve = np.linspace(t_range[0], t_range[1], 300)

sol = solve_ivp(coupled_oscillator, t_range, y0, t_eval = t_solve)

def E(x1, x2, v1, v2):
    return 0.5*m1*v1**2 + 0.5*m2*v2**2 + 0.5*k0*x1**2 + 0.5*k1*(x2-x1)**2 + 0.5*k2*x2**2

ax[0].plot(sol.t, sol.y[0], label = 'Mass 1');
ax[0].plot(sol.t, sol.y[1], label = 'Mass 2');
ax[0].set_xlabel("t [s]")
ax[0].set_ylabel("displacement [m]");
ax[0].set_title("Displacement/Time Graph");
ax[0].legend()
ax[0].grid("true")

ax[1].plot(sol.t, sol.y[2], label = 'Mass 1');
ax[1].plot(sol.t, sol.y[3], label = 'Mass 2');
ax[1].set_xlabel("t [s]")
ax[1].set_ylabel("velocity [ms^-1]");
ax[1].set_title("Velocity/Time Graph");
ax[1].legend()
ax[1].grid("true")

ax[2].plot(sol.t, E(sol.y[0], sol.y[1], sol.y[2], sol.y[3]));
ax[2].set_xlabel("t [s]")
ax[2].set_ylabel("Energy [J]");
ax[2].set_title("Energy/Time Graph");
ax[2].grid("true")
ax[2].set_ylim(-40, 40);
```

← energy.

↙ input into energy.