

---

# **npsolve**

***Release 0.1.1***

**Mar 08, 2021**



---

## Contents:

---

<b>1</b>	<b>Tutorials</b>	<b>3</b>
1.1	Tutorial 1 - Basics . . . . .	3
1.2	Tutorial 2 - Handling discontinuities . . . . .	6
1.3	Tutorial 3 - Timeseries input . . . . .	10
1.4	Tutorial 4 - Sharing values between objects . . . . .	13
1.5	Tutorial 5 - Using polymorphism . . . . .	23
1.6	Tutorial 6 - Logging variables and stopping . . . . .	28
<b>2</b>	<b>Package documentation</b>	<b>33</b>
2.1	npsolve package . . . . .	33
2.2	npsolve.core module . . . . .	33
2.3	npsolve.cache module . . . . .	36
2.4	npsolve.soft_functions module . . . . .	37
2.5	npsolve.solvers module . . . . .	41
2.6	npsolve.utils module . . . . .	42
<b>3</b>	<b>Related packages</b>	<b>45</b>
<b>4</b>	<b>Indices and tables</b>	<b>47</b>
	<b>Python Module Index</b>	<b>49</b>
	<b>Index</b>	<b>51</b>



The *npsolve* package lets you use object-oriented classes and methods for calculations with numerical solvers.

Instead of defining equations, you can write methods with any combination of logic and equations. It decouples the calculations from the machinery to numerically iterate with them, giving you all the benefits you'd expect from objects.

Many numerical solvers (like those in *scipy*) provide candidate solutions as a numpy ndarray. They often also require a numpy ndarray as a return value (e.g. an array of derivatives) during the solution. These requirements can make it difficult to use an object oriented approach to performing the calculations. Usually, we end up with script-like code that loses many of the benefits of object-oriented programming.

The npsolve framework links a solver with multiple classes that handle the calculations for each step in the algorithm. It allows different parts of the calculations to be encapsulated and polymorphic, and makes the code much easier to modify and maintain.



The fastest way to learn *npsolve* is to work through some simple examples. These short tutorials walk through some simple examples, to help learn how to use *npsolve*.

### 1.1 Tutorial 1 - Basics

Let's use *npsolve* to do some integration through time, like you would to solve an ODE. Instead of equations, though, we're using class methods.

First, setup some classes that you want to do calculations with. We do this by using the `add_var()` method to setup variables and their initial values.

```
import numpy as np
import npsolve

class Component1(npsolve.Partial):
    def __init__(self):
        super().__init__() # Don't forget to call this!
        self.add_var('position', init=0.1)
        self.add_var('velocity', init=0.3)

class Component2(npsolve.Partial):
    def __init__(self):
        super().__init__() # Don't forget to call this!
        self.add_var('force', init=-0.1)
```

All the variables are made available to all *Partial* instances automatically through their *state* attribute. It's a dictionary. The *add\_var* method sets initial values into the instance's state dictionary. Later, the *Solver* will ultimately replace the *state* attribute with a new dictionary that contains all variables from all the *Partial* classes.

Next, we'll tell these classes how to do some calculations during each time step. The *step* method is called automatically and expects a dictionary of return values (e.g. derivatives). We'll use that one here. The state dictionary is given again as the first argument, but we're going to use the internal *state* attribute instead. So, we'll add some more methods:

```
class Component1(npsolve.Partial):
    def __init__(self):
        super().__init__() # Don't forget to call this!
        self.add_var('position', init=0.1)
        self.add_var('velocity', init=0.3)

    def step(self, state_dct, t, *args):
        """ Called by the solver at each time step

        Calculate acceleration based on the net force.
        """
        acceleration = 1.0 * self.state['force']
        derivatives = {'position': self.state['velocity'],
                      'velocity': acceleration}
        return derivatives

class Component2(npsolve.Partial):
    def __init__(self):
        super().__init__() # Don't forget to call this!
        self.add_var('force', init=-0.1)

    def calculate(self, t):
        ''' Some arbitrary calculations based on current time t
        and the position at that time calculated in Component1.
        This returns a derivative for variable 'c'
        '''
        dc = 1.0 * np.cos(2*t) * self.state['position']
        derivatives = {'force': dc}
        return derivatives

    def step(self, state_dct, t, *args):
        ''' Called by the solver at each time step '''
        return self.calculate(t)
```

Now, we'll set up the solver. For this example, we'll use the `odeint` solver from Scipy. Here's what it looks like:

```
class Solver(npsolve.Solver):
    def solve(self, t_end=10):
        self.npsolve_init() # Initialise
        self.t_vec = np.linspace(0, t_end, 1001)
        result = odeint(self.step, self.npsolve_initial_values, self.t_vec)
        return result
```

Let's look at what's going on in the `solve` method. By default, Solvers have a `step` method that's ready to use. (They also have a `one_way_step` method that doesn't expect return values from the `Partials`, and a `tstep` method that expects a time value as the first argument.) After initialisation, the initial values set by the `Partial` classes are captured in the `npsolve_initial_values` attribute. By default, the Solver's `step` method returns a vector of all the return values, the same size as the Solver's `npsolve_initial_values` array. So most of the work is done for us here already.

Note here that we don't need to know anything about the model or the elements in the model. This allows us to decouple the model and `Partials` from the solver. We can pass in different models, or pass models to different solvers. We can make models with different components. It's flexible and easy to maintain!

To run, we just have to instantiate the Solver and `Partial` instances, then pass a list or dictionary of the `Partial` instances to the `connect()` method of the Solver. They'll link up automatically through *fastwire*.



```
def run():
    solver = Solver()
    partials = [Component1(), Component2()]
    solver.connect(partials)
    res = solver.solve()
    return res, solver
```

Let's set up a plot to see the results. Use the `npsolve_slices` attribute of the Solver to get us the right columns.

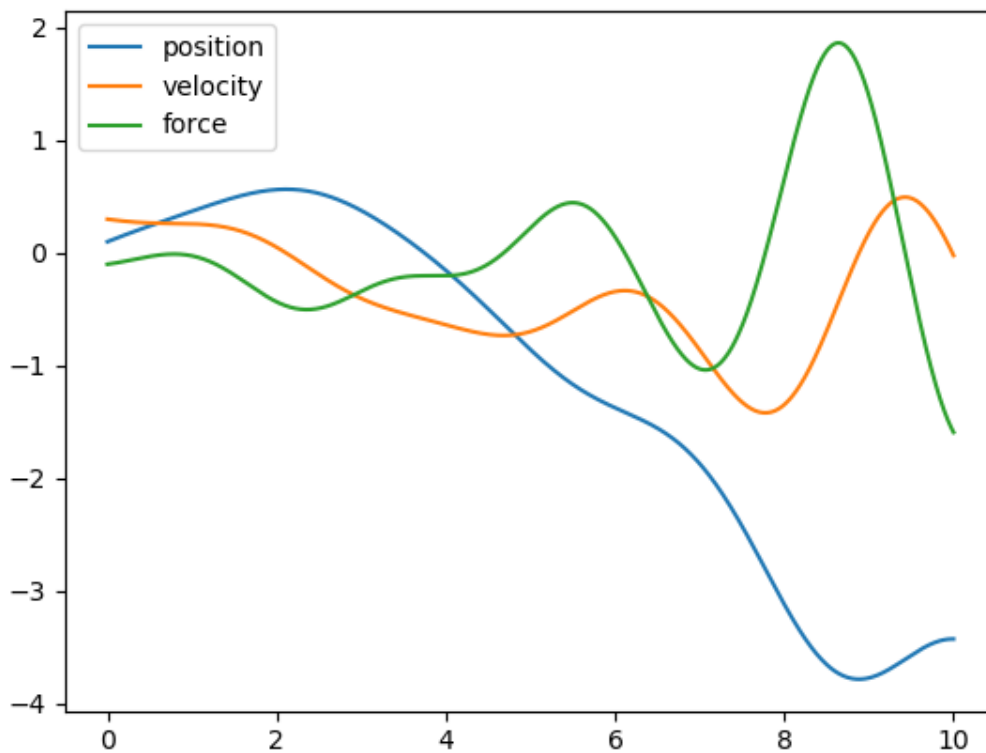
```
import matplotlib.pyplot as plt

def plot(res, s):
    slices = s.npsolve_slices

    plt.plot(s.t_vec, res[:,slices['position']], label='position')
    plt.plot(s.t_vec, res[:,slices['velocity']], label='velocity')
    plt.plot(s.t_vec, res[:,slices['force']], label='force')
    plt.legend()
```

Now let's run it!

```
res, s = run()
plot(res, s)
```



## 1.2 Tutorial 2 - Handling discontinuities

Often, real-world integration problems in engineering have discontinuities. That means that the physics cannot be described adequately by a single set of equations.

Fortunately, *npsolve* provides a *soft\_functions* module to make it easy to handle discontinuities, by preventing them entirely. These functions work by providing a differential that changes smoothly over a very, very small time, distance, or whatever value it's applied to. Variable time step solvers, such many in in *scipy.integrate*, handle these very small but smooth transitions easily. The approximation of these functions inevitably introduces small errors, but for many real-world problems these errors are negligible. It's far more important to get a 99.999% accurate result than none at all.

We'll illustrate the use of two *soft\_functions* to model a bouncing ball. It's going to start rolling along a ledge, before falling off onto a surface that it bounces on. Let's start with some setup:

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

import npsolve
from npsolve.soft_functions import negdiff, below

G = -9.80665
Y_SURFACE = 1.5
X_LEDGE = 2.0
Y_LEDGE = 5.0
```

Now we'll start writing our *Ball* class. We'll give it some mass, an initial x-velocity, and some parameters to control how it bounces. We'll create the bouncing by a combination of spring-like behaviour and velocity-dependent damping. Here's the constructor:

```
class Ball(npsolve.Partial):
    def __init__(self, mass=1.0,
                  k_bounce=1e7,
                  c_bounce=3e3,
                  initial_vel=5.0):
        super().__init__() # Don't forget to call this!
        self.mass = mass
        self.k_bounce = k_bounce
        self.c_bounce = c_bounce
        self.add_var('position', init=np.array([0.0, Y_LEDGE]))
        self.add_var('velocity', init=np.array([initial_vel, 0.0]))
```

Notice here that the *position* and *velocity* variables are 1D numpy ndarrays of size 2, to reflect x and y axes. *npsolve* handles variables of 1D arrays!

Let's create a method in the *Ball* class to calculate the force of gravity on the ball.

```
class Ball():
    # ...

    def F_gravity(self):
        """ Force of gravity """
        return np.array([0, self.mass * G])
```

Pretty simple so far. Now let's make our ledge react the force of gravity until the ball reaches the edge of the ledge with another method:

```
class Ball():
    # ...

    def F_ledge(self):
        x_pos = self.state['position'][0]
        return -self.F_gravity() * below(x_pos, limit=X_LEDGE)
```

Here, we're using the *below* soft function. It's 1 below a limit (here X\_LEDGE) and 0 above it, with a very small but smooth transition around the limit. So, this force will only apply until the ball reaches the edge of the ledge. We do this so the differentials are continuously smooth, so the scipy integrator won't have any trouble with discontinuities.

Now, let's make the method to generate a bouncing force when it hits the surface. This one is a bit more complex:

```
class Ball():
    # ...

    def F_bounce(self):
        """ Force bouncing on the surface """
        y_pos = self.state['position'][1]
        y_vel = self.state['velocity'][1]
        F_spring = -self.k_bounce * negdiff(y_pos, limit=Y_SURFACE)
        c_damping = -self.c_bounce * below(y_pos, Y_SURFACE)
        F_damping = c_damping * negdiff(y_vel, limit=0)
        return np.array([0, F_spring + F_damping])
```

We're using the *negdiff* soft function here. It gives the difference between a value and some limit when the value is below the limit and 0 when it's above. Again, it's a smooth function with a very small transition region around the limit. We apply that to the spring force so it only provides that force when the ball is (slightly) below the surface.

For the damping, we've used the *below* function to set up a damping coefficient, *c\_damping*, that will only apply when the ball is (slightly) below the surface. We also want the damping force to only push the ball - we don't want it to 'pull' the ball like a sticky surface might. So, we've used the *negdiff* function on the velocity. When the velocity is positive, the *negdiff* function goes to 0 so the damping force will be 0 too. We add the two forces to return a bounce force.

Now we can set up the *step* method that gets called by the integrator.

```
class Ball():
    # ...

    def step(self, state_dct, t, *args):
        """ Called by the solver at each time step """
        F_net = self.F_gravity() + self.F_ledge() + self.F_bounce()
        acceleration = F_net / self.mass
        derivatives = {'position': self.state['velocity'],
                      'velocity': acceleration}
        return derivatives
```

This just sums the forces, calculates the acceleration by applying elementary physics, then returns the acceleration.

Now let's set up the Solver:

```
class Solver(npsolve.Solver):
    def solve(self, t_end=3.0, n=100001):
        self.npsolve_init() # Initialise
        t_vec = np.linspace(0, t_end, n)
        solution = odeint(self.step, self.npsolve_initial_values, t_vec)
        dct = self.as_dct(solution)
```

(continues on next page)

(continued from previous page)

```
dct['time'] = t_vec
return dct
```

This is very much like the solver in the quickstart example. Here though, we're using the *as\_dct* method to convert the outputs to a dictionary, in which each key is variable name, and each value is an array of the values through time.

Let's set up a function to run it and plot the results.

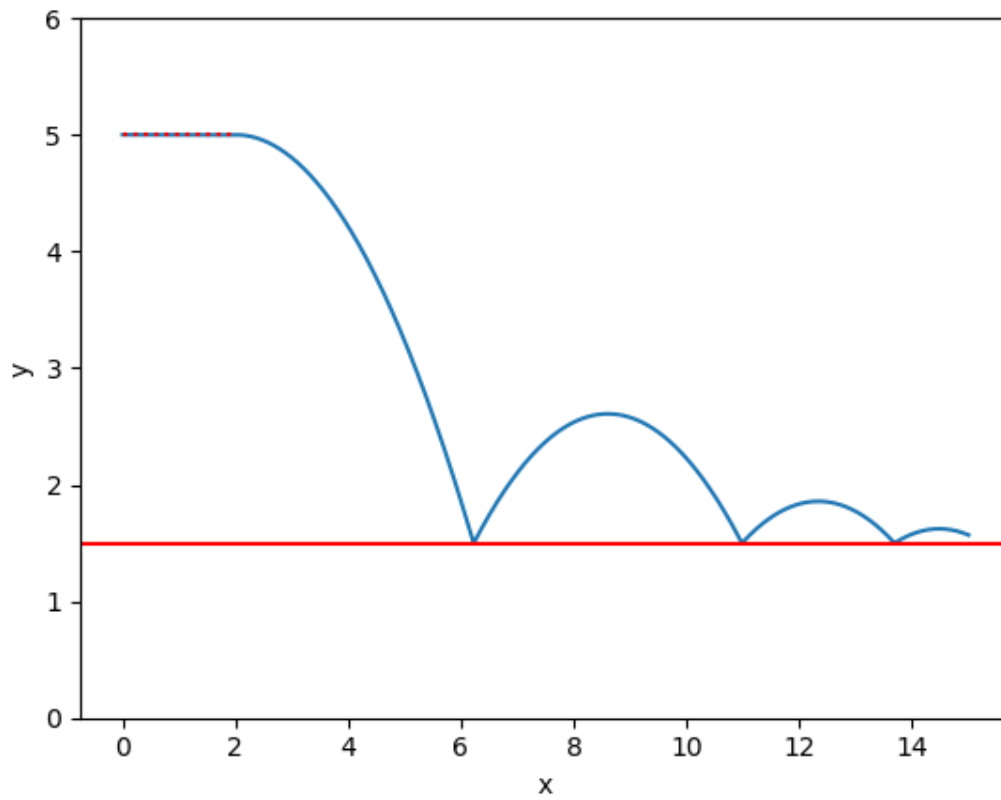
```
def run(partials, t_end=3.0, n=100001):
    solver = Solver()
    solver.connect(partials)
    return solver.solve(t_end=t_end, n=n)

def plot(dct):
    plt.plot(dct['position'][:,0], dct['position'][:,1], label='position')
    plt.axhline(Y_SURFACE, c='r')
    plt.plot([0, X_LEDGE], [Y_LEDGE, Y_LEDGE], 'r:')
    plt.ylim(0, 6)
    plt.xlabel('x')
    plt.ylabel('y')
```

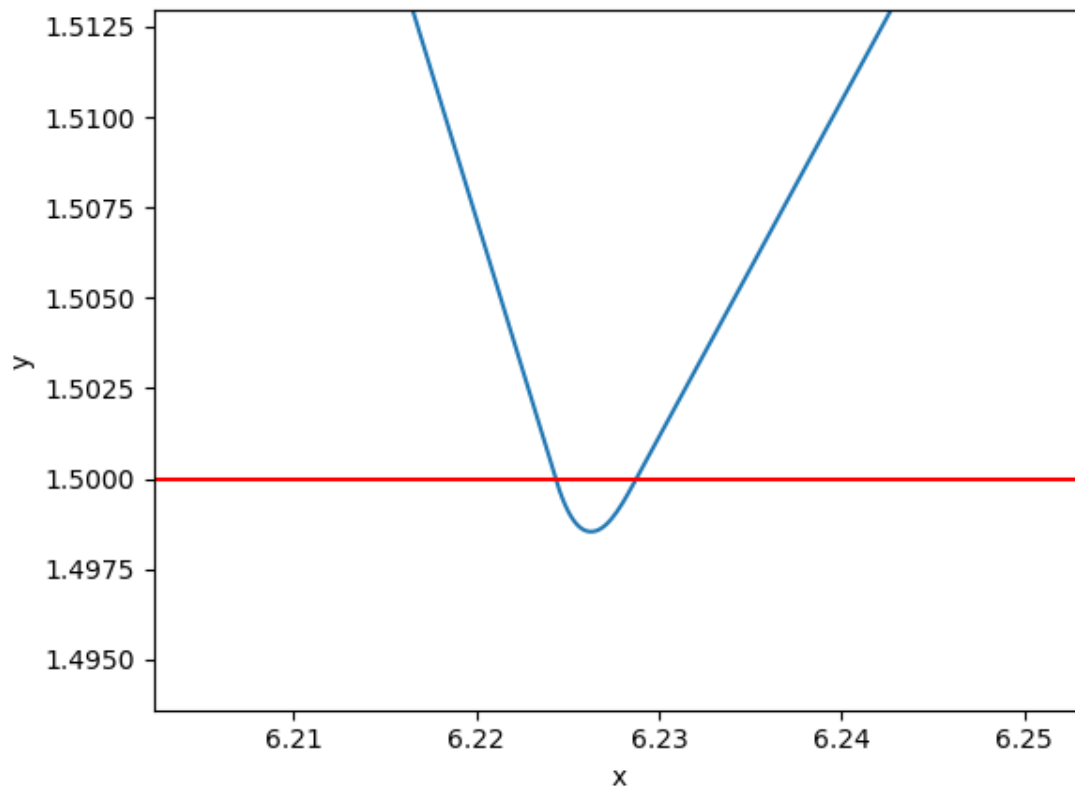
Here, we're making the run method a bit more generic. It's going to take a list of Partial instances, connect them to the solver, call the solve method, then return the result and the partials. Now we can run it!

```
ball = Ball()
dct = run(ball)
plot(dct)
```

We've made a bouncing ball!



If we zoom way in on the bounce, you can see it's actually smooth.



## 1.3 Tutorial 3 - Timeseries input

We might have inputs we want to use that aren't functions. They might be measured timeseries data, for example, such as position over time. Or, they could be hypothetical data. For this tutorial, we're just use some random numbers to make a particle move about.

First, the setup:

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

import npsolve
from npsolve.utils import Timeseries

from tutorial_2 import run
```

Notice, we're going to reuse the Solver and run function we set up in Tutorial 2, but for a completely different model.

Now let's start to write a Particle class:

```
class Particle(npsolve.Partial):
    def __init__(self):
        super().__init__() # Don't forget to call this!
```

(continues on next page)

(continued from previous page)

```

self.time_points = np.linspace(0, 1, 51)
np.random.seed(0)
self.positions = np.random.rand(51, 2) * 10
self.xts = Timeseries(self.time_points, self.positions[:,0])
self.yts = Timeseries(self.time_points, self.positions[:,1])
self.add_var('position', init=self.positions[0,:])

```

We're creating some timeseries data in the *time\_points* and *positions* attributes. Then, for each x or y axis, we're creating a Timeseries object. We pass in the x values (the time points) and the values at those times (positions). The Timeseries class takes care of the rest, and will smoothly interpolate between those points using a spline. Notice also that we're setting the initial values of the *position* variable to the first pair of (x,y) values in the *positions* array.

Now, let's write the step method.

```

class Particle(npsolve.Partial):
    # ...

    def step(self, state_dct, t, *args):
        ''' Called by the solver at each time step
        Calculate acceleration based on the
        '''
        velocity = np.array([self.xts(t, der=1), self.yts(t, der=1)])
        derivatives = {'position': velocity}
        return derivatives

```

We're getting the velocity in each axis by simply calling the Timeseries instances we set up in the constructor. We pass them in the current x value (in this case the time), and the derivative we want (der). In this case, we want the first derivative of the position, which is time.

We can reuse our run method and Solver from Tutorial 2. We'll set up some plot functions though:

```

def plot(dct, particle):
    plt.plot(dct['position'][:,0], dct['position'][:,1], linewidth=0.5)
    plt.scatter(particle.positions[:,0], particle.positions[:,1], c='r',
                marker='.')
    plt.xlabel('x')
    plt.ylabel('y')

def plot_vs_time(dct, particle):
    fig, axes = plt.subplots(2, 1, sharex=True)
    axes[0].plot(dct['time'], dct['position'][:,0], linewidth=0.5)
    axes[0].scatter(particle.time_points, particle.positions[:,0], c='r',
                    marker='.')
    axes[0].set_xlabel('time')
    axes[0].set_ylabel('x')
    axes[1].plot(dct['time'], dct['position'][:,1], linewidth=0.5)
    axes[1].scatter(particle.time_points, particle.positions[:,1], c='r',
                    marker='.')
    axes[1].set_xlabel('time')
    axes[1].set_ylabel('y')

```

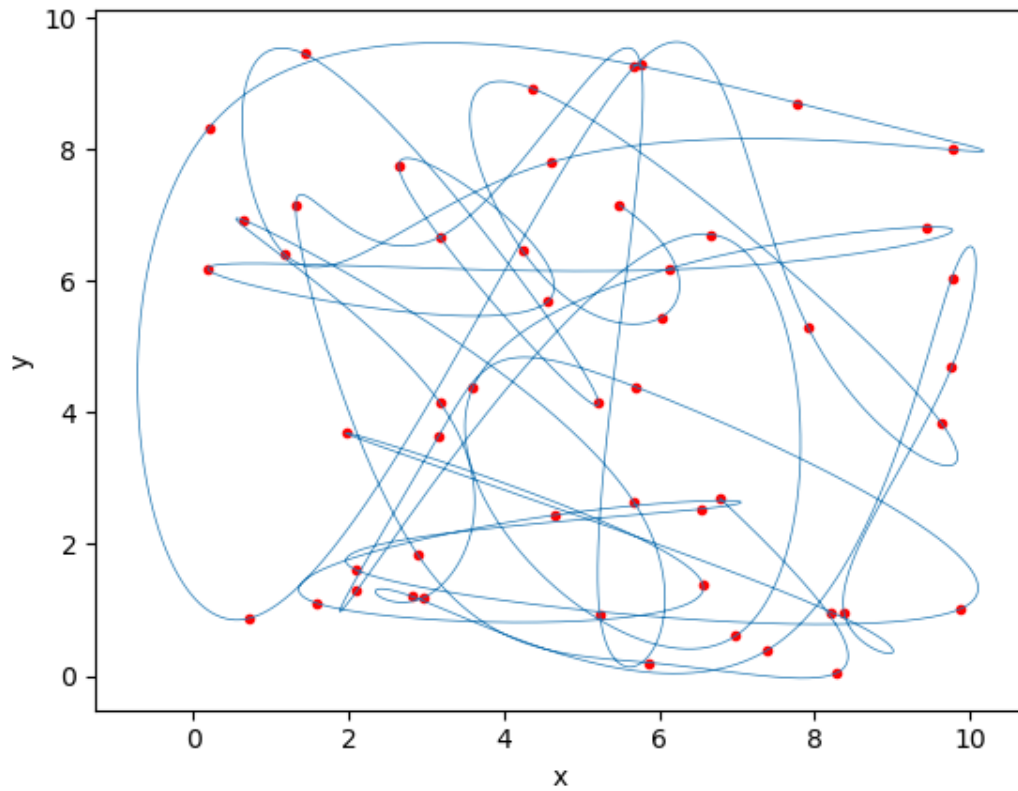
Now everything's set. Let's run it!

```

particle = Particle()
dct = run([particle], t_end=1.0)
plot(dct, particle)
plot_vs_time(dct, particle)

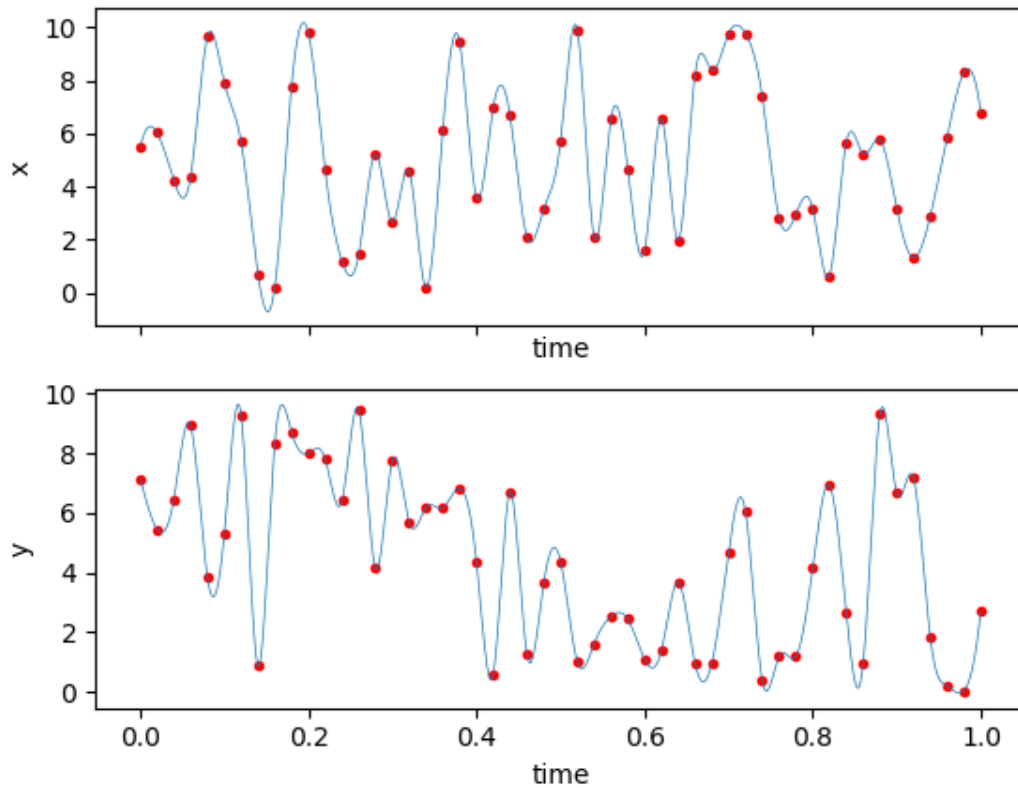
```

Here's how our particle has moved...



And we can see how the Timeseries instances have controlled the velocity, and hence position, over time.





In real-world models, you'll probably want to use the Timeseries classes together with other model equations and logic.

**Tip:** You can make a subclass of a Partial instance and overwrite the step method, so that the derivatives are set by Timeseries classes. Then, you can easily switch between the original Partial instance and one in which one or more derivatives are set by timeseries data!

## 1.4 Tutorial 4 - Sharing values between objects

In more complex models, values often need to be shared between different Partial instances. Sometimes, those values are *state* variables that are declared with the *add\_var* method. In that case, any Partial connected to the Solver will get access to them through their *state* dictionary (i.e. *self.state[<variable name>]*). But some shared values are not state variables. How do we handle that?

We use the *fastwire* package. It provides a convenient, event-like way to share variables. This tutorial will give an example. We're going to simulate the dynamics of a frictionless slider moving in the x axis with a pendulum attached that is free to move in the x and y axes. We'll add a sinusoidal force to the slider to excite the system dynamics.

First, a little setup:

```
import numpy as np
import matplotlib.pyplot as plt
import npsolve
from tutorial_2 import run
```

(continues on next page)

(continued from previous page)

```
G = np.array([0, -9.80665])
```

Here, we've made  $G$  a 2D vector to represent gravity.

Now we'll get a *wire box*. A wire box is a collection of wires that we'll use to pass values. Here's how we do that:

```
import fastwire as fw
wire_box = fw.get_wire_box('demo')
```

You can use `wire_box = fw.get_wire_box('demo')` in any code module and it'll get the same wire box, so you don't have to import that object into other modules.

Now let's start making the Slider.

```
class Slider(npsolve.Partial, fw.Wired):
    def __init__(self, freq=1.0, mass=1.0):
        super().__init__() # Don't forget to call this!
        self.freq = freq
        self.mass = mass
        self.add_var('s_pos', init=np.zeros(2))
        self.add_var('s_vel', init=np.zeros(2))
```

Importantly, we're inheriting the `fw.Wired` class. That lets us use *fastwire* decorators. We're also making the Slider fully 2D, even though at this stage we only want it to move in  $x$ .

We're doing to connect the Pendulum to the Slider, and the Pendulum will need to know where the Slider is so it can pivot about the right point. Here's how we make the pivot location and velocity available to the Pendulum:

```
class Slider(npsolve.Partial, fw.Wired):
    # ...

    @wire_box.supply('pivot')
    def pivot(self, t):
        """ The location of the pivot that connects to the pendulum """
        return self.state['s_pos'], self.state['s_vel']
```

We decorate the method with `@wire_box.supply('pivot')` because we've called our wire box `wire_box`. This tells *fastwire* that this method supplies the values referred to by the wire called 'pivot'. We'll pass in the current time,  $t$ , although we don't need it yet.

Let's set up a method to create the excitation force:

```
class Slider(npsolve.Partial, fw.Wired):
    # ...

    def F_sinusoid(self, t):
        """ The force to make the system do something """
        return 10 * np.cos(2 * np.pi * (self.freq * t))
```

Now we can write our *step* method to return the state derivatives by doing some basic physics.

```
class Slider(npsolve.Partial, fw.Wired):
    # ...

    def step(self, state_dct, t, *args):
        """ Called by the solver at each time step """
```

(continues on next page)

(continued from previous page)

```

F_pivot = -wire_box['F_pivot'].fetch(t)
F_pivot_x = F_pivot[0]
F_sinusoid_x = self.F_sinusoid(t)
F_net_x = F_pivot_x + F_sinusoid_x
acc = np.array([F_net_x / self.mass, 0])
derivatives = {'s_pos': state_dct['s_vel'],
               's_vel': acc}

return derivatives

```

Notice here we're going to pull in a force,  $F_{pivot}$ , which is going to be calculated by the Pendulum class. We just have to use the *fetch* method on the right wire, which here we've called  $F_{pivot}$ . For this example, we'll also pass in the current time 't' to the method that will supply that force (we haven't written that method yet). We're flipping the sign because the slider will see the reaction force.

Now, let's make the Pendulum class.

```

class Pendulum(npsolve.Partial, fw.Wired):
    def __init__(self, mass=1.0, k=1e6, c=1e4, l=1.0):
        super().__init__() # Don't forget to call this!
        self.mass = mass
        self.k = k
        self.c = c
        self.l = l
        self.add_var('p_pos', init=np.array([0, -self.l]))
        self.add_var('p_vel', init=np.array([0, 0]))

```

Again, we're inheriting *fw.Wired*. This class has some stiffness ( $k$ ) and damping  $c$  parameters, along with mass ( $mass$ ) and length ( $l$ ). It needs to calculate the force that arises because of its connection to the Slider. We're going to model a very stiff, damped connection between the pivot on the Slider and the position of the Pendulum.

```

class Pendulum(npsolve.Partial, fw.Wired):
    # ...

    @wire_box.supply('F_pivot')
    @npsolve.mono_cached()
    def F_pivot(self, t):
        """ Work out the force on the pendulum mass """
        pivot_pos, pivot_vel = wire_box['pivot'].fetch(t)
        rel_pos = pivot_pos - self.state['p_pos']
        rel_vel = pivot_vel - self.state['p_vel']
        dist = np.linalg.norm(rel_pos)
        unit_vec = rel_pos / dist
        F_spring = self.k * (dist - self.l) * unit_vec
        rel_vel_in_line = np.dot(rel_vel, unit_vec)
        F_damping = self.c * rel_vel_in_line * unit_vec
        return F_spring + F_damping

```

We're again using the *@wire\_box* decorator so that this method will supply the  $F_{pivot}$  wire. The return value, the force at the pivot, will be used by both the Slider (via the  $F_{pivot}$  wire) and the Pendulum (directly). We can't assume which object will call the  $F_{pivot}$  method first, but we don't want to have it calculate the result twice. (This is a simple example, but in computationally intensive calculations, reducing calculations can be important.) So, we use the *@npsolve.mono\_cached()* decorator here as well. This caches the result for the current timestep. Subsequent calls simply return that value. The *mono\_cached()* doesn't care about the value of arguments. If they might change for the same timestep, you can use the *multi\_cached()* decorator instead.

Let's add the force of gravity now:

```
class Pendulum(npsolve.Partial, fw.Wired):
    # ...

    def F_gravity(self):
        return self.mass * G
```

Finally, we'll make the *step* method, doing some basic physics to calculate acceleration.

```
class Pendulum(npsolve.Partial, fw.Wired):
    # ...

    def step(self, state_dct, t, *args):
        ''' Called by the solver at each time step
        Calculate acceleration based on the
        '''
        F_net = self.F_pivot(t) + self.F_gravity()
        acceleration = F_net / self.mass
        derivatives = {'p_pos': state_dct['p_vel'],
                      'p_vel': acceleration}
        return derivatives
```

Before we run, let's make some plot functions...

```
def plot_xs(dct):
    plt.plot(dct['time'], dct['s_pos'][:,0], label='slider')
    plt.plot(dct['time'], dct['p_pos'][:,0], label='pendulum')
    plt.xlabel('time')
    plt.ylabel('x')
    plt.legend(loc=3)

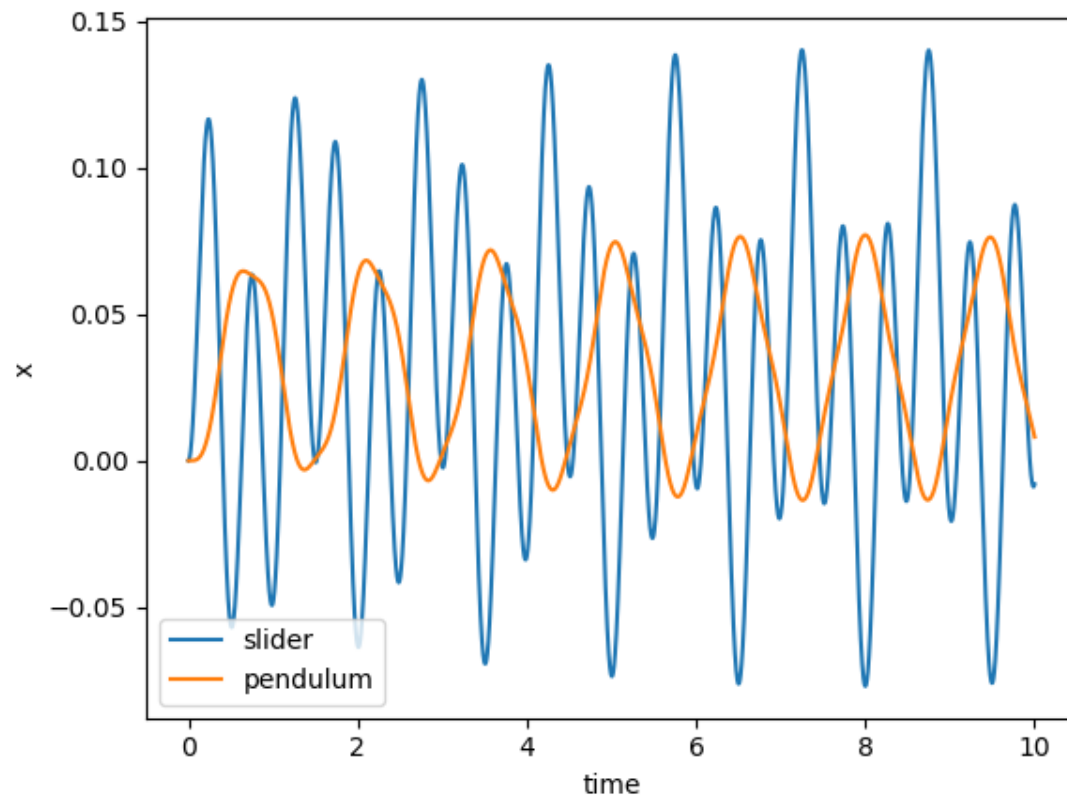
def plot_trajectories(dct):
    plt.plot(dct['s_pos'][:,0], dct['s_pos'][:,1], label='slider')
    plt.plot(dct['p_pos'][:,0], dct['p_pos'][:,1], label='pendulum')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.xlim(-1.5, 1.5)
    plt.ylim(-1.2, 1.2)
    plt.gca().set_aspect('equal')
    plt.legend(loc=2)
```

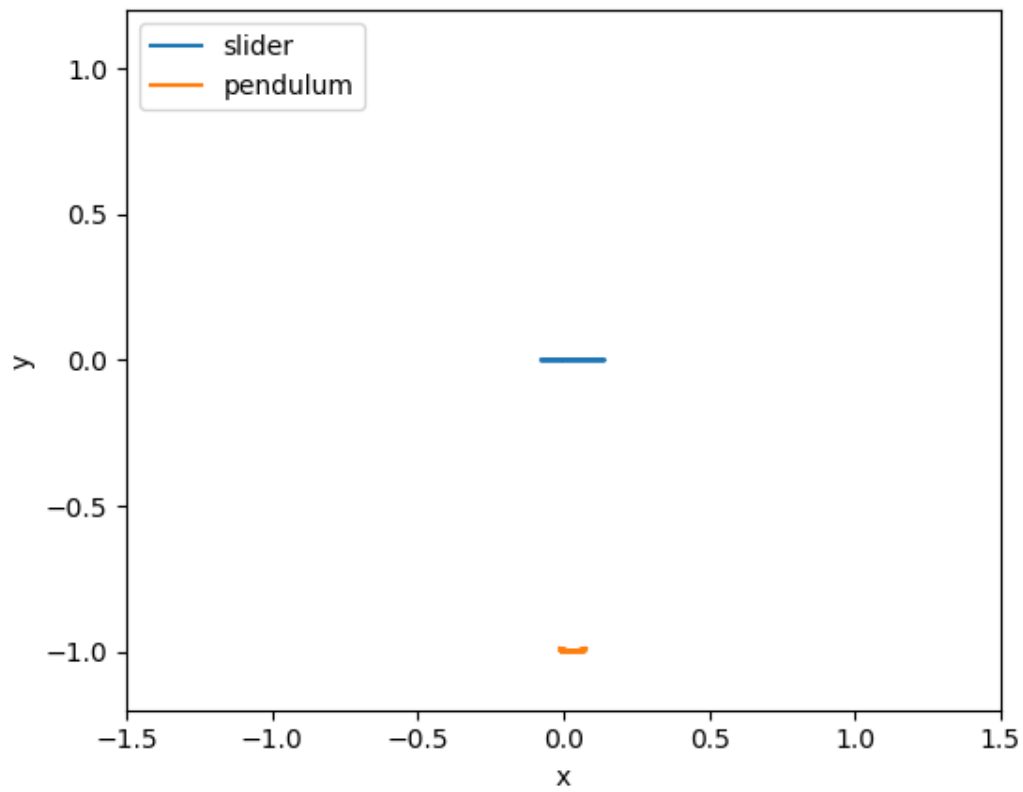
Finally, we'll make a little function to run the model and plot the results.

```
def execute(freq):
    partials = [Slider(freq=freq), Pendulum()]
    dct = run(partial, t_end=10.0, n=10001)
    plot_xs(dct)
    plot_trajectories(dct)
```

Let's see what happens at 2 Hz:

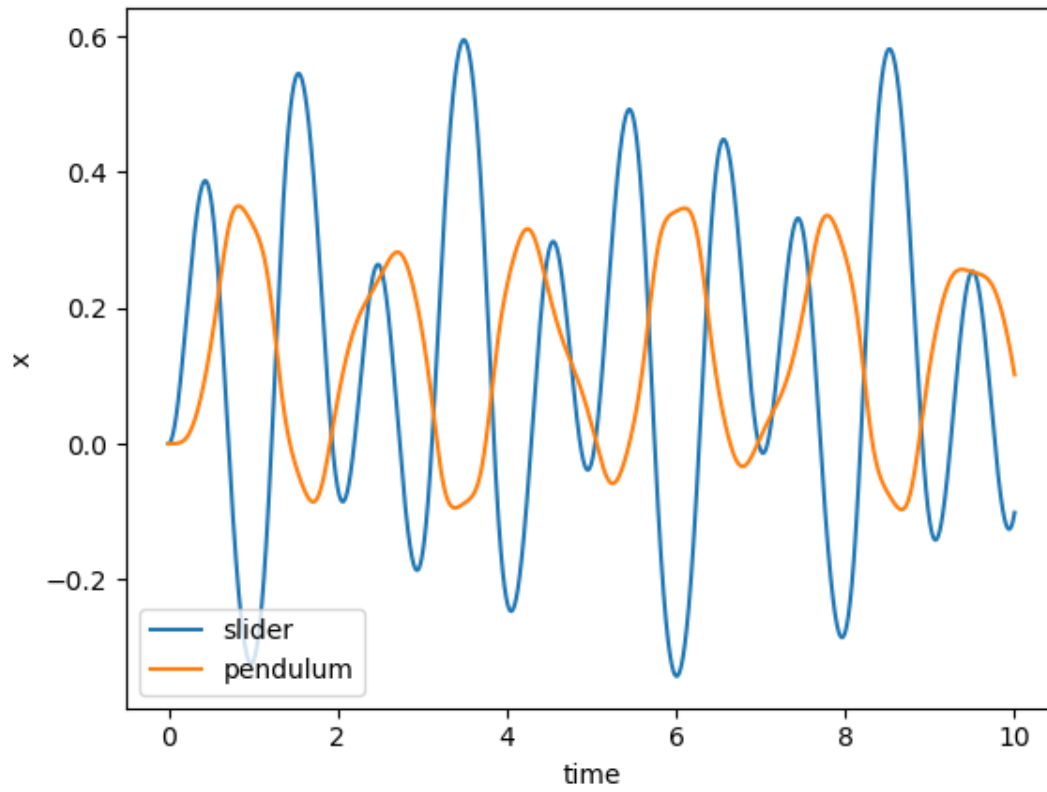
```
execute(2.0)
```

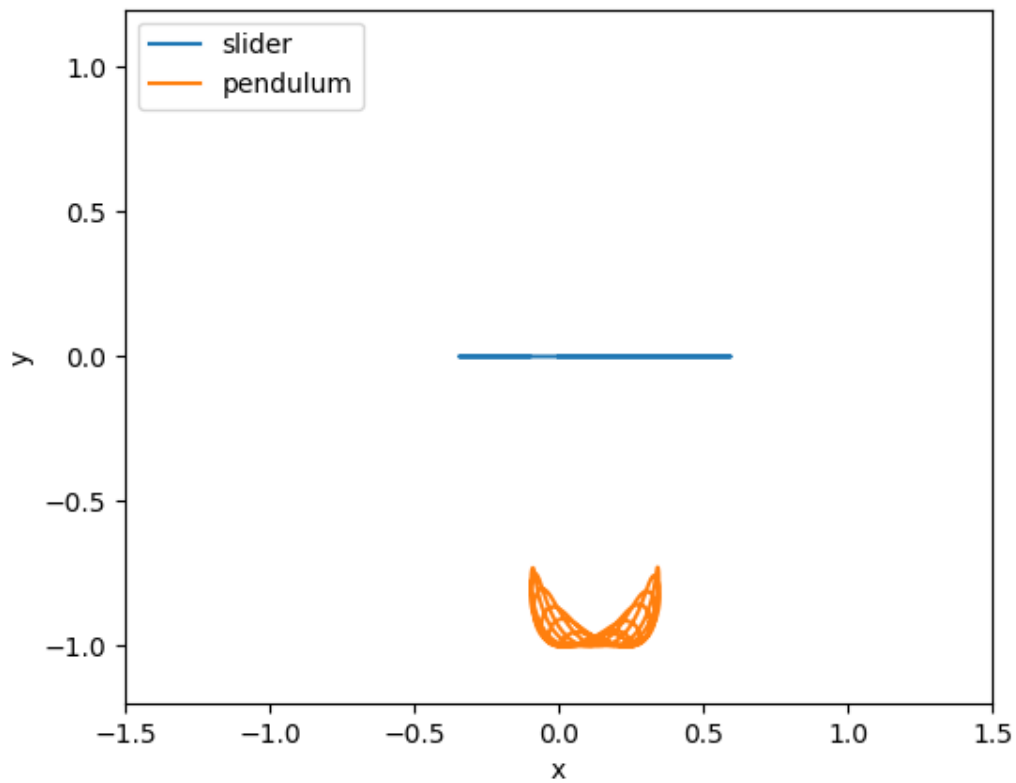




Nothing very interesting. Both objects just oscillate, as you might expect. Now let's try at 1 Hz:

```
execute(1.0)
```

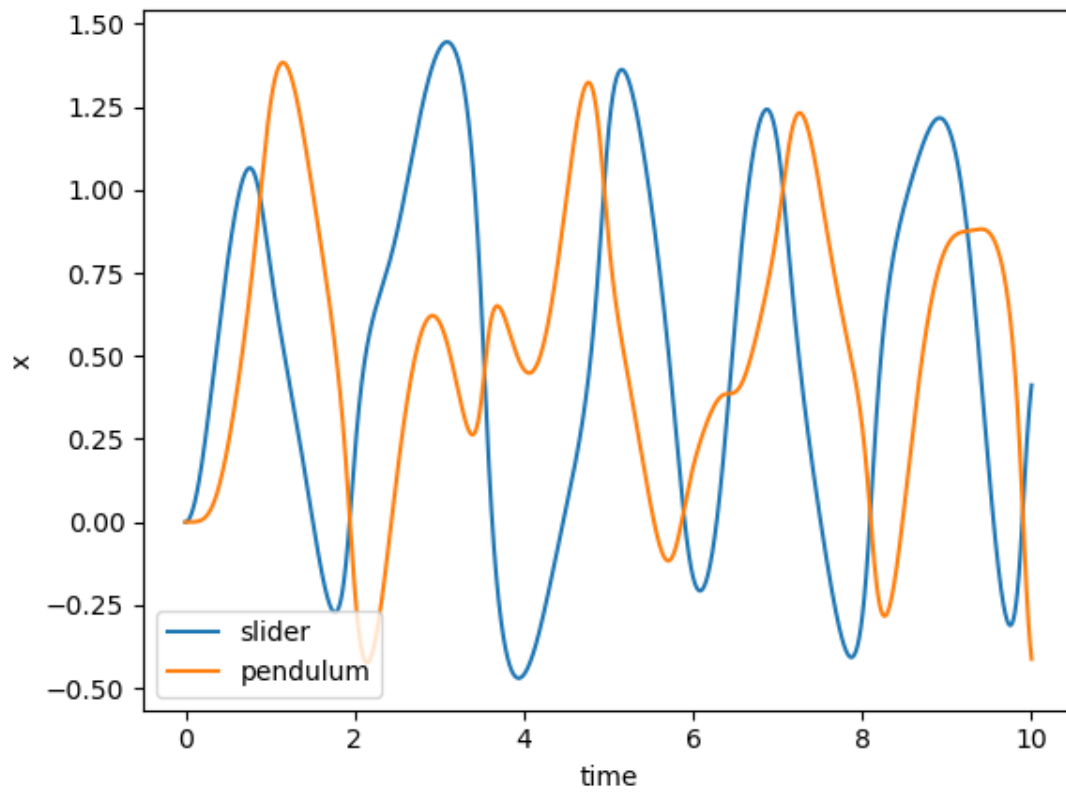




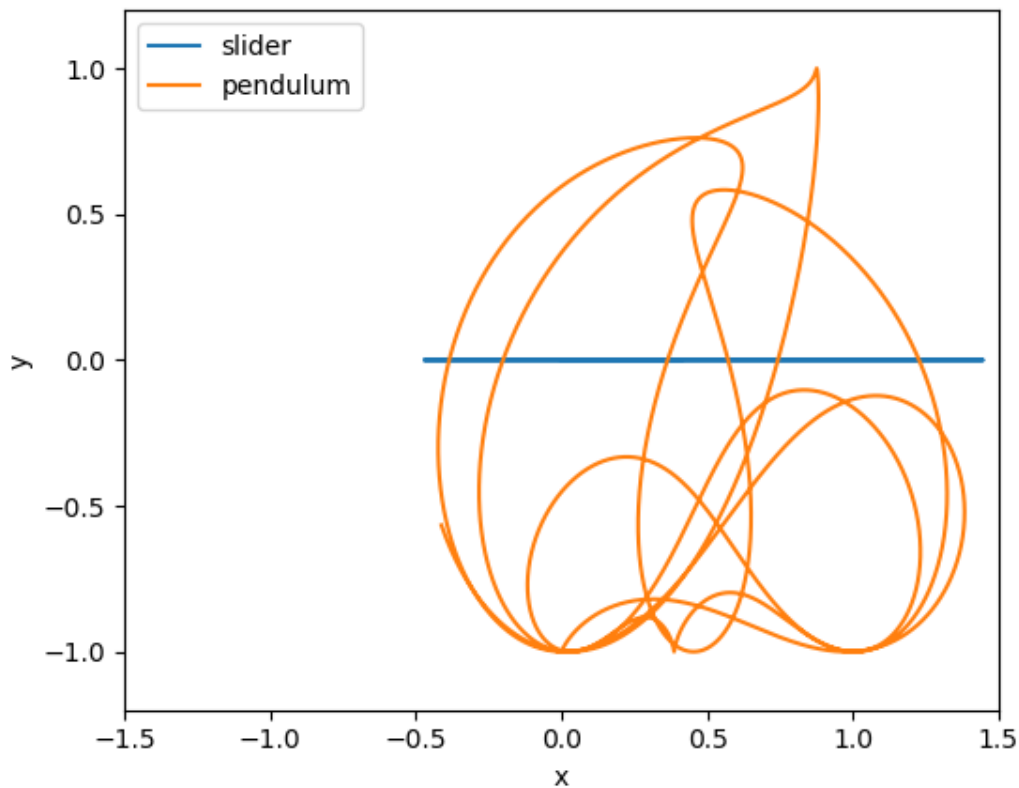
The Pendulum is wobbling around a bit more now. Let's try at 0.5 Hz:

```
execute(1.0)
```



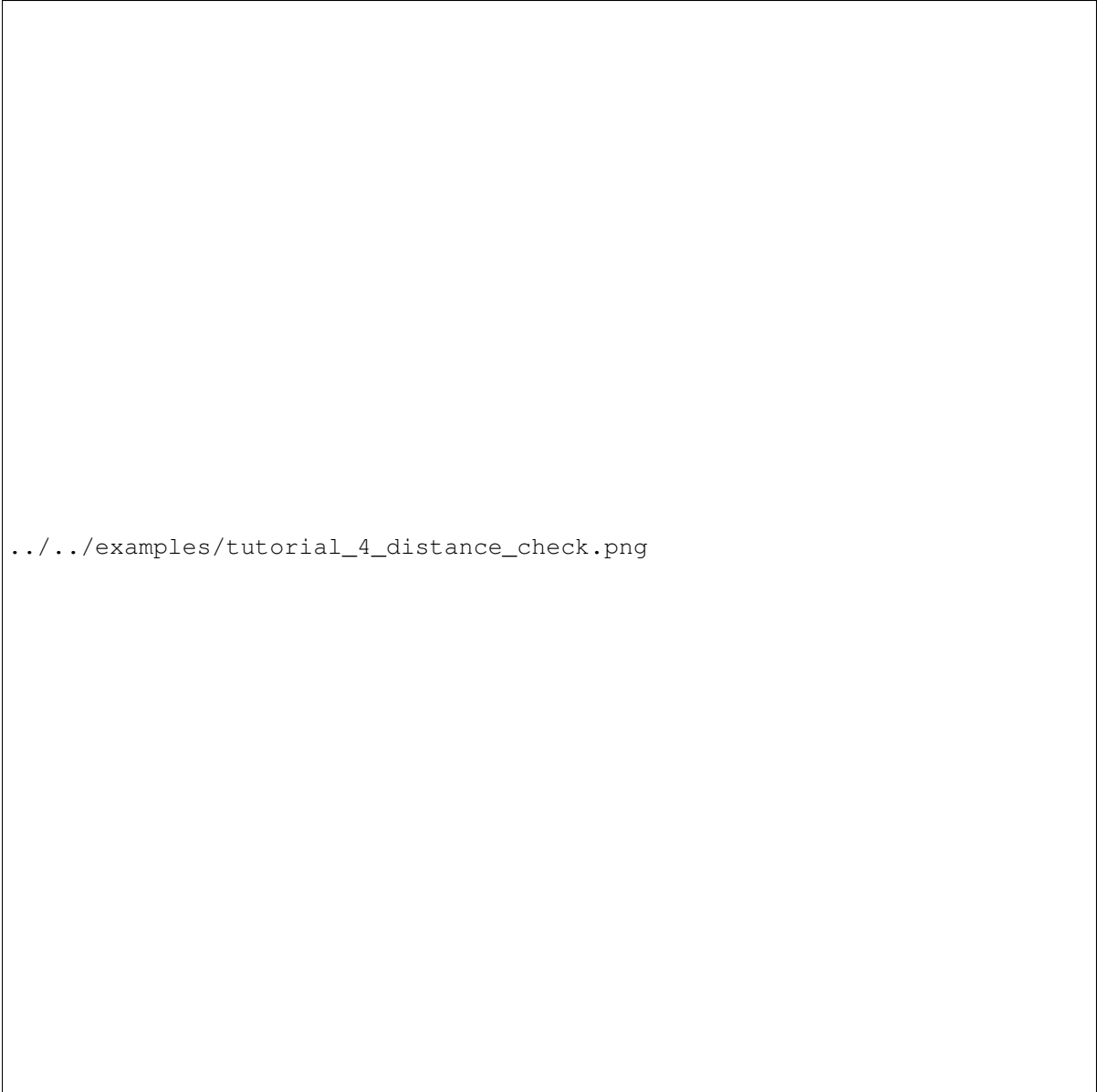


When we look at the trajectories, we see what's really happening...



Remember that our pendulum isn't quite a rigid body - we've approximated it as a very stiff, highly damped spring. We should check that the approximation is good by checking that the distance between the pivot and pendulum is very close to 1.0. Let's plot the distance:

```
def plot_distance_check(dct):  
    diff = dct['p_pos'] - dct['s_pos']  
    dist = np.linalg.norm(diff, axis=1)  
    plt.plot(dct['time'], dist)  
    plt.xlabel('time')  
    plt.ylabel('length')  
  
plot_distance_check(dct)
```



```
../../examples/tutorial_4_distance_check.png
```

Our maximum length error is only 0.0001, compared to our pendulum length of 1.0, so we know the errors due to that approximation will be small.

## 1.5 Tutorial 5 - Using polymorphism

In Tutorial 4, we made a Pendulum move under a Slider. What if that pendulum moved under the Particle we made in Tutorial 3 instead? Let's find out, to demonstrate how *npsolve* makes it easy to do so.

We'll start by importing what we need:

```
import numpy as np
import matplotlib.pyplot as plt
from tutorial_2 import run as t2_run
from tutorial_3 import Particle
```

(continues on next page)

(continued from previous page)

```
from tutorial_4 import Pendulum
import fastwire as fw
wire_box = fw.get_wire_box('demo')
```

We need to add a *pivot* method to the Particle, so we'll subclass it like this.

```
class Particle2(Particle, fw.Wired):

    @wire_box.supply('pivot')
    def pivot(self, t):
        velocity = np.array([self.xts(t, der=1), self.yts(t, der=1)])
        return self.state['position'], velocity
```

That's it - now this class will substitute for the old Slider class! We're returning the position and velocity in the same format that the Pendulum expected from the Slider, and we've wired it to the same 'pivot' wire.

There's one small glitch - initial conditions. In Tutorial 4, we conveniently set up the initial conditions right. How do we do that now? Here's how:

```
def set_init_condition(particle, pendulum):
    init_particle_pos = particle.npsolve_vars['position']['init']
    init_pendulum_pos = init_particle_pos - np.array([0.0, 1.0])
    pendulum.set_init('p_pos', init_pendulum_pos)
```

When a class calls *add\_var*, that information gets added to the *npsolve\_vars* attribute. We're taking that, subtracting our pendulum length from the height, and then calling the *set\_init* method to set the initial position of the pendulum to that position. Easy.

Now we'll make a function to run the new model.

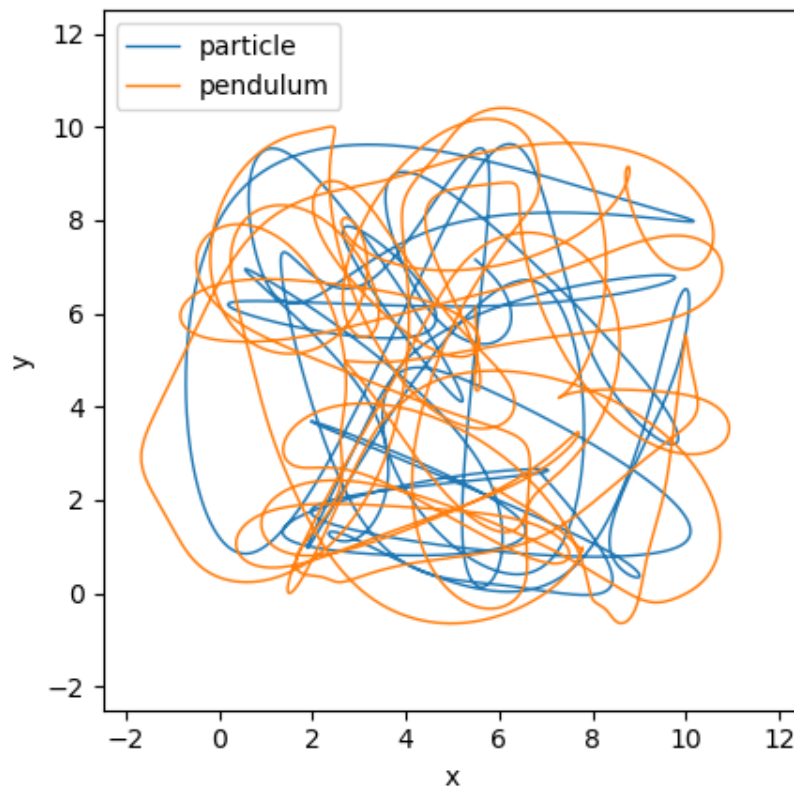
```
def run(k=1e6, c=1e4):
    particle = Particle2()
    pendulum = Pendulum(k=k, c=c)
    set_init_condition(particle, pendulum)
    partials = [particle, pendulum]
    dct = t2_run(partials, t_end=1.0, n=10001)
    return dct
```

And a new plot function to see the results.

```
def plot_trajectories(dct):
    plt.plot(dct['position'][:,0], dct['position'][:,1], linewidth=1.0,
             label='particle')
    plt.plot(dct['p_pos'][:,0], dct['p_pos'][:,1], linewidth=1.0,
             label='pendulum')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.xlim(-2.5, 12.5)
    plt.ylim(-2.5, 12.5)
    plt.gca().set_aspect('equal')
    plt.legend(loc=2)
```

Let's run it!

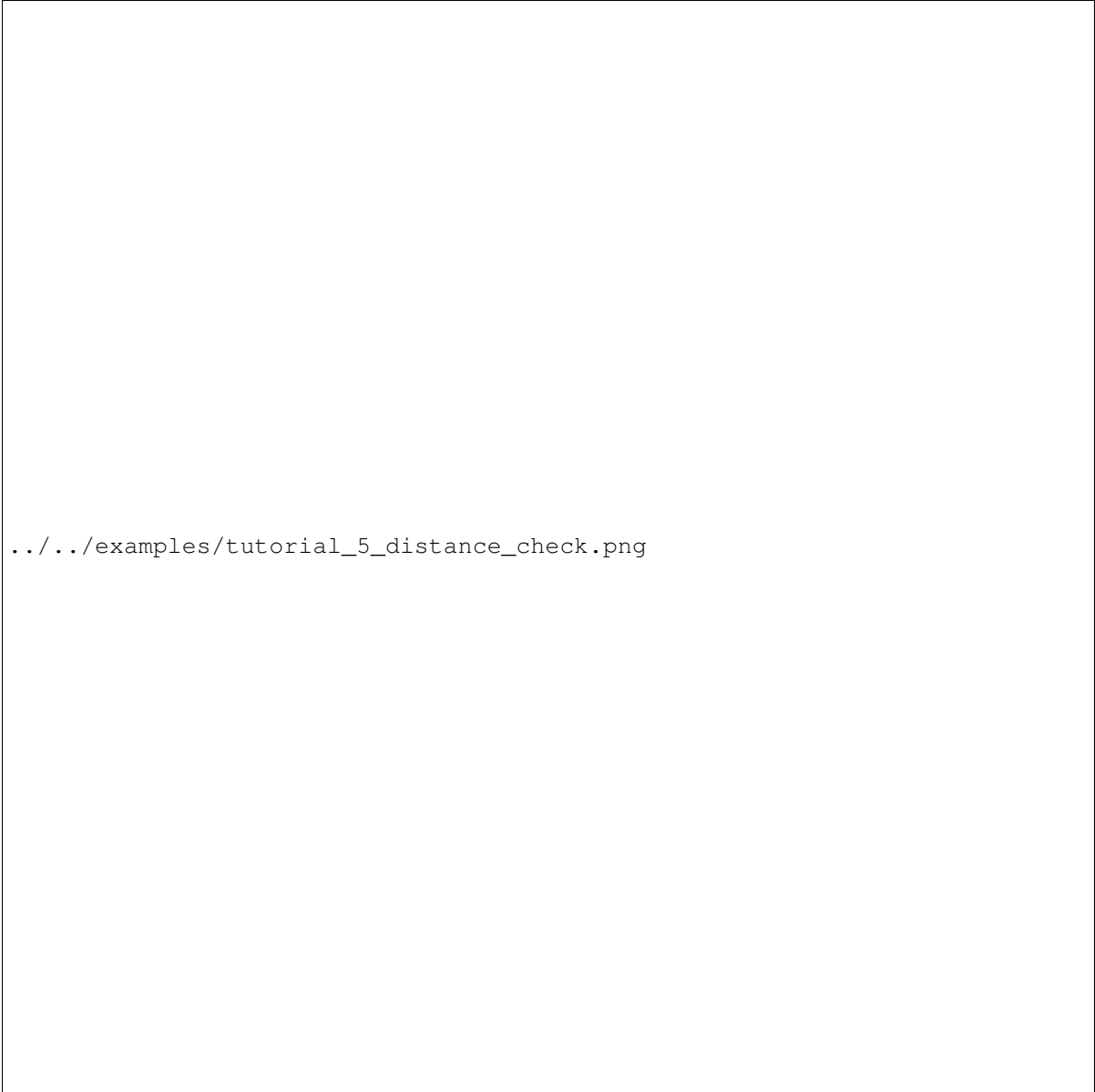
```
dct = run()
plot_trajectories(dct)
```



Our pendulum is now hurtling around with a particle!

Let's check the pendulum length again to ensure it's behaving as expected.

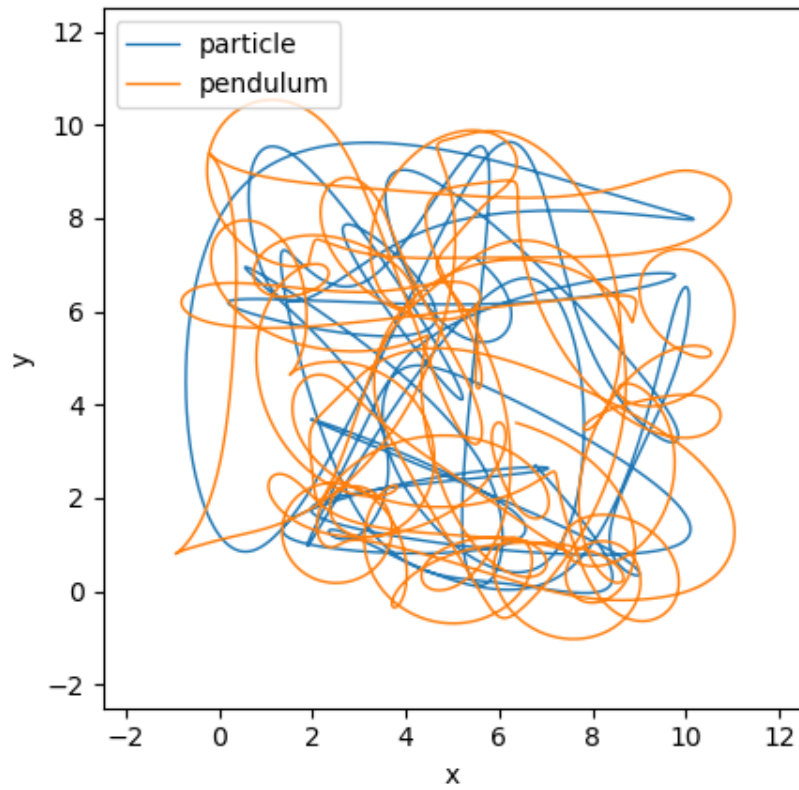
```
def plot_distance_check(dct):  
    diff = dct['p_pos'] - dct['position']  
    dist = np.linalg.norm(diff, axis=1)  
    plt.plot(dct['time'], dist)  
    plt.xlabel('time')  
    plt.ylabel('length')  
  
plot_distance_check(dct)
```



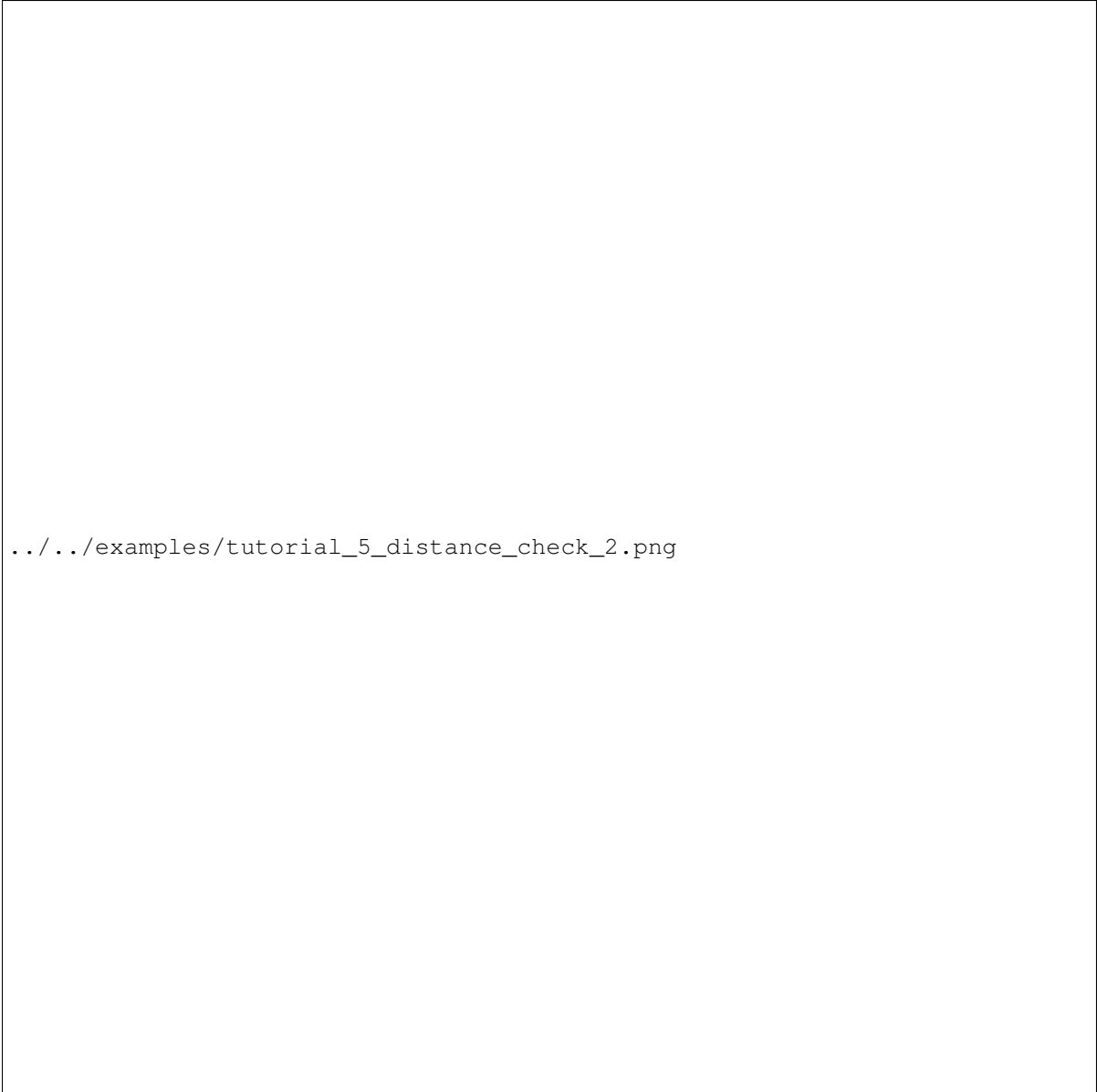
```
../../examples/tutorial_5_distance_check.png
```

Here, our stiff spring and firm damping aren't quite enough to handle the fast accelerations due to the particle motion. So, we'll tweak our parameters and run again:

```
dct = run(k=1e9, c=1e7)
plot_trajectories(dct)
plot_distance_check(dct)
```



Our Pendulum trajectory is different.



```
../../examples/tutorial_5_distance_check_2.png
```

Now, our distance check looks ok, so we can be more confident with this result - as crazy as it is!

Think about what this lets us do. We might write classes for a given situation. Then, say if we run an experiment and get some measured data, we can swap the relevant Partial for one that uses that Timeseries data. Or, perhaps we have a new idea to test - we can easily swap out that part of the model and compare it back to back with the first.

We can validate our classes against unittests, theory, and experimental data. Then, we can run new models that use them without changing anything within those classes. This can provide confidence that we haven't made any mistakes within those classes in the new model.

## 1.6 Tutorial 6 - Logging variables and stopping

We often want to output non-state values during calculation. We may want to end the calculation when a condition is met. Fortunately, these are both easy to do both with *npsolve*.



*npsolve* provides a special Integrator class to help you do these. By default, it uses the LSODA integrator in *scipy* (if *scipy* is present). However, it's built to use any set of integrators that work like *scipy*'s *ode*. It works by breaking up the time domain into short frames. It uses the integrator to integrate between each frame as normal. *scipy*'s integrators are stateful, so they can continue with the next frame with minimal overhead.

At each frame, the Integrator class sets a special *status* flag, which can be anywhere else in the code. When this flag (`npsolve.solvers.Final`) is set, it means the state values for that time are the 'final' (not guesses by the variable-time-step solver). Partial classes can then add values to be logged. The solver also listens to a flag to stop the integration at that point that can be set from anywhere else in the code.

Here's an example. Let's change the step method of the Pendulum class in Tutorial 4 to add some logging and raise a stop flag. We'll first do some imports:

```
import matplotlib.pyplot as plt
import npsolve
from tutorial_4 import Slider, Pendulum
```

Now let's set up our status dictionary and our logger.

```
from npsolve.solvers import FINAL, STOP
status = npsolve.get_status('demos_status')
logger = npsolve.get_logger('demos_logger')
```

We'll use the FINAL and STOP flags with the status dictionary.

Now let's modify the stop method of the Pendulum class so it looks like this:

```
class Pendulum2(Pendulum):
    def step(self, state_dct, t, *args):
        ''' Called by the solver at each time step
        Calculate acceleration based on the
        '''
        F_pivot = self.F_pivot(t)
        F_gravity = self.F_gravity()
        F_net = F_pivot + F_gravity
        acceleration = F_net / self.mass
        if status[FINAL]:
            logger['F_pivot'].append(F_pivot)
            logger['acceleration'].append(acceleration)
        if F_pivot[1] > 90.0:
            status[STOP] = True
        derivatives = {'p_pos': state_dct['p_vel'],
                      'p_vel': acceleration}
        return derivatives
```

We've added some logging under *if status[FINAL]*: and raised a stop flag under *if F\_pivot[1] > 90.0*:

We'll use the Integrator class to solve this, since it knows how to use those flags. We'll write a new run method to use it.

```
def run(partials, t_end=20.0, n=100001):
    solver = npsolve.solvers.Integrator(status=status,
                                         logger=logger,
                                         framerate=n//t_end)

    solver.connect(partials)
    return solver.run(t_end)
```

Now some plotting functions:

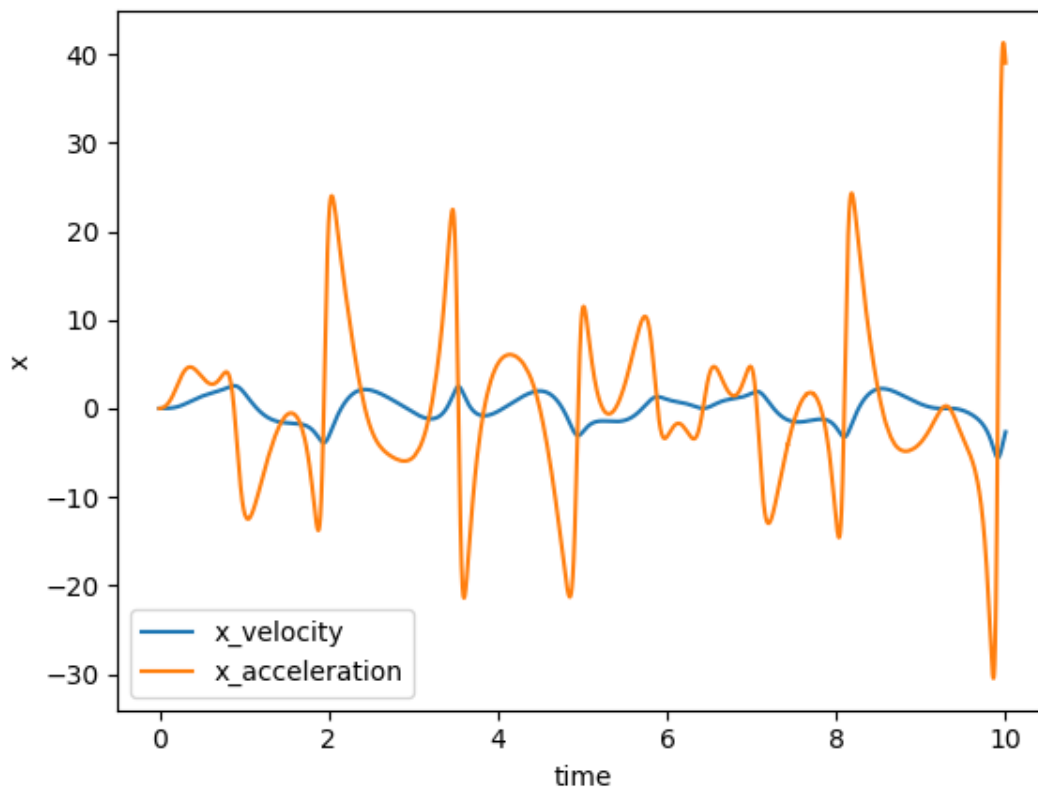
```
def plot_F_pivot(dct):
    plt.figure()
    plt.plot(dct['time'], dct['F_pivot'][:,0], label='F_pivot_x')
    plt.plot(dct['time'], dct['F_pivot'][:,1], label='F_pivot_y')
    plt.xlabel('time')
    plt.ylabel('x')
    plt.legend(loc=3)

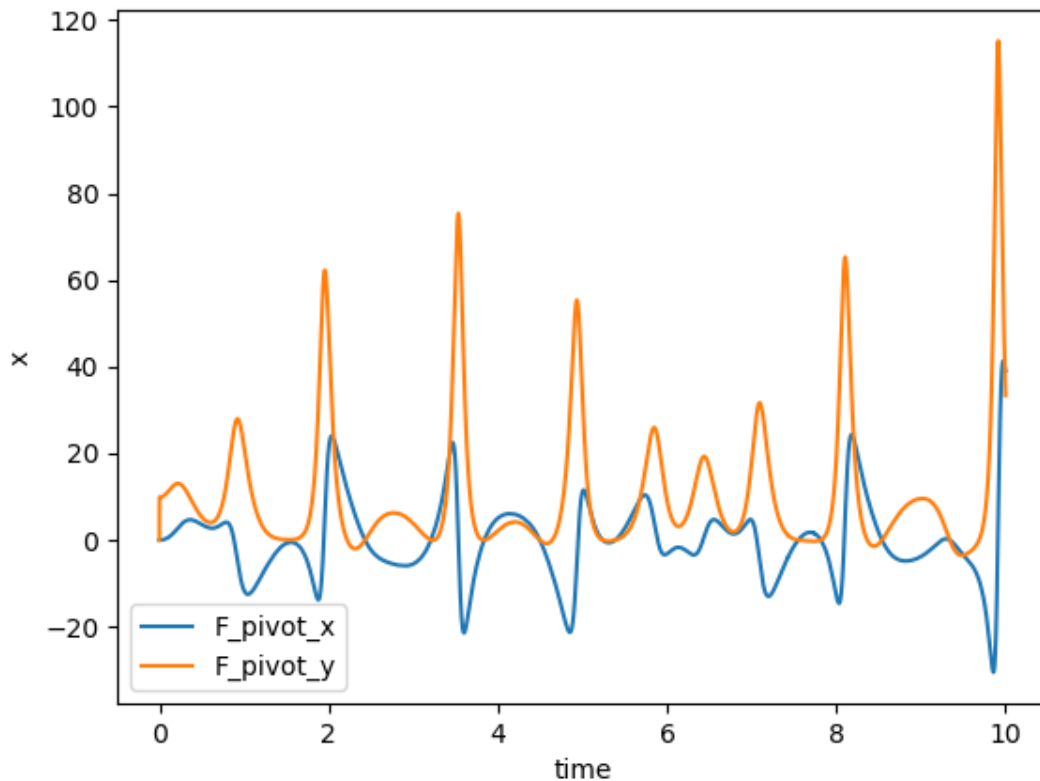
def plot_acc(dct):
    plt.figure()
    plt.plot(dct['time'], dct['p_vel'][:,0], label='x_velocity')
    plt.plot(dct['time'], dct['acceleration'][:,0], label='x_acceleration')
    plt.xlabel('time')
    plt.ylabel('x')
    plt.legend(loc=3)
```

Finally, we'll execute the new code:

```
def execute(freq):
    partials = [Slider(freq=freq), Pendulum2()]
    dct = run(partials, t_end=20.0, n=10001)
    plot_F_pivot(dct)
    plot_acc(dct)

execute(freq=0.5)
```





It's as easy as that. Notice first that the integrator has stopped early because the `Pendulum2` instance raised a `status[STOP] = True` flag.

Our logged outputs are now in the output dictionary along with our state variables, which makes it easy to work with them. The logging is controlled by the `Partial` instances, and we don't have to change anything else in our code. As a bonus, the `Pendulum2` class will still work as normal with the original solver in Tutorial 2 - the logging and stopping just won't work with it because that solver doesn't use them.

**Note:** Be sure that logged variables are logged only *once* per time step, since otherwise the outputs won't match up right. In tricky situations, you can use the `@npsolve.mono_cached()` decorator to do that, since it will only execute the code inside the function once per time step.



## 2.1 npsolve package

### 2.1.1 Module contents

Created on Mon Aug 5 20:46:26 2019

@author: Reuben

## 2.2 npsolve.core module

Created on Mon Aug 5 14:34:54 2019

@author: Reuben

Npsolve has a simple, small core built on fastwire. It's designed to give good flexibility without compromising on performance.

**class** `npsolve.core.Partial`

Bases: `object`

A base class responsible for a set of variables

---

**Note:** `__init__` method must be called.

---

**add\_var** (*name*, *init*, *safe=True*, *live=True*, *\*\*kwargs*)

Add a new variable

#### Parameters

- **name** (*str*) – The variable name
- **init** (*array-like*) – The initial value(s). Can be a scalar or 1D ndarray.

- **\*\*kwargs** – Optional key word attributes for the variable.

**add\_vars** (*dct*)

Add multiple variables

**Parameters** *dct* (*dict*) – A dictionary in which keys are variable names and values are dictionaries with name, initial value, etc.

**cache\_clear** ()

Clear the cache for all cached methods

**clear\_vars** ()

**connect** (*cid=None*)

Connect this instance to the Solver instance

**Parameters**

- *cid* (*int*) – The container id provided the `setup_signals` method
- **the Solver instance.** (*of*) –

**enable\_caching** ()

Enable or

**get\_init** (*name*)

Get the initial value for a variable

**Parameters** *name* (*str*) – The variable name

**set\_init** (*name, init*)

Set the initial value for a variable

**Parameters**

- *name* (*str*) – The variable name
- *init* (*array-like*) – The initial value(s). Can be a scalar or 1D ndarray.

**set\_meta** (*name, \*\*kwargs*)

Set meta information for a variable

**Parameters** **\*\*kwargs** – Key word attributes for the variable.

**set\_vectors** (*state\_dct, ret\_dct*)

Override to set up views of the state vector

**Parameters**

- *state\_dct* (*dict*) – A dictionary of numpy array views for the state
- **all variables. Provided by the Solver.** (*of*) –
- *ret\_dct* (*dict*) – A similar dictionary of return values. Not
- *used.* (*usually*) –

**step** (*state\_dct, \*args*)

**class** `npsolve.core.Solver`

Bases: `object`

The solver that pulls together the partials and allows solving

**as\_dct** (*sol*)

Split out solution array into dictionary of values

**Parameters** *sol* (*ndarray*) – A 1D or 2D array where columns correspond to state values

This convenience method splits out a 2D array into a dictionary of vectors or arrays, with variables as keys.

**close\_signals()**  
Deactive the signal container

---

**Note:** If autoconnecting is enabled, other Partials may connect to the Solver if the container is active.

---

**connect** (*partials*)  
Connect a dict or list of partials to the Solver instance

**Parameters** **partials** (*list*, *dict*, *Partial*) – A list or dictionary of Partial instances.

**fetch\_partials()**  
Fetch a dictionary of all connected Partial instances

**freeze()**  
Give static copies of vectors to connected Partial instances

**Warning:** This will prevent the ‘step’ methods from being able to update the values.

**get\_state\_dct** (*squeeze=True*, *unitise=True*)

**npsolve\_finish()**  
Tidy up after a round of solving

**npsolve\_init** (*pinned=None*)  
Initialise the Partials and be ready to solve

**Parameters** **pinned** (*dict*) – A dictionary of variable-value pairs to hold constant during stepping.

**one\_way\_step** (*vec*, *\*args*, *\*\*kwargs*)  
The method to be called every iteration with no return val

**Parameters**

- **vec** (*ndarray*) – The state vector (passed in by the solver)
- **args** – Optional arguments passed to step method in each Partial.
- **kwargs** – Optional keyword arguments for each step method call.

**Returns** None

**Note:** This method relies on other methods being used to inform the solver during its iteration.

**pinned** (*dct*)  
A context manager that unpins all variables on exit

**remove\_signals()**  
Remove the signal container

---

**Note:** This helps to keep the signal box tidy.

---

**setup\_signals()**  
Setup the fastwire signals that Partial instances will require

**Returns** The container id for the signals.

**Return type** int

**step** (*vec*, \**args*, \*\**kwargs*)

The method to be called every iteration by the numerical solver

**Parameters**

- **vec** (*ndarray*) – The state vector (passed in by the solver)
- **args** – Optional arguments passed to step method in each Partial.
- **kwargs** – Optional keyword arguments for each step method call.

**Returns**

**A dictionary containing keys for each variable. The values** must match the shape of the state. These will often contain derivatives for integration problems and error or cost values for optimisation problems.

**Return type** dict

**tstep** (*t*, *vec*, \**args*, \*\**kwargs*)

The method to be called every iteration by the numerical solver

**Parameters**

- **vec** (*ndarray*) – The state vector (passed in by the solver)
- **args** – Optional arguments passed to step method in each Partial.
- **kwargs** – Optional keyword arguments for each step method call.

**Returns**

**A dictionary containing keys for each variable. The values** must match the shape of the state. These will often contain derivatives for integration problems and error or cost values for optimisation problems.

**Return type** dict

---

**Note:** This method is similar to the `step()` method, but is used where a time value is passed as the first argument.

---

**unfreeze** (*state=None*)

Give 'live' vectors to connected Partial instances

**Parameters** **state** (*ndarray*) – An optional vector to initialise the state.

## 2.3 npsolve.cache module

Created on Wed Aug 7 07:06:54 2019

@author: Reuben

Simple caching inspired by `functools.lru_cache`.

Sometimes, Partial instances may need to call each other (via a fastwire fetch is a good method). Caching allows a way to reuse the computations for each step if needed, to avoid having to double-up in those cases.



`npsolve.cache.mono_cached()`

A cache method that only considers the 'self' argument

This works very similar to multi-cache but doesn't use the `make_key` function from `functools` to save a little bit of time.

`npsolve.cache.multi_cached()`

A cache method that considers arguments

## 2.4 npsolve.soft\_functions module

Created on Fri May 15 16:17:04 2020

@author: Reuben

These functions can be used to prevent discontinuities, which can cause trouble for numerical methods.

`npsolve.soft_functions.above(x, limit=0.0, scale=0.0001)`

A smooth step from 0 below a limit to 1 above it

### Parameters

- **x** (*int, float, ndarray*) – The value(s)
- **limit** (*float*) – [OPTIONAL] The value to step at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

**See also:**

`soft_step`

`npsolve.soft_functions.below(x, limit=0.0, scale=0.0001)`

A smooth step from 1 below a limit to 0 above it

### Parameters

- **x** (*int, float, ndarray*) – The value(s)
- **limit** (*float*) – [OPTIONAL] The value to step at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

**See also:**

`soft_step`

`npsolve.soft_functions.ceil(x, limit=0.0, scale=0.0001)`

Limit value to a maximum softly to prevent discontinuous gradient

### Parameters

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **limit** (*float*) – [OPTIONAL] The value to limit at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

**See also:**

`soft_limit`

`npsolve.soft_functions.clip(x, lower, upper, scale=0.0001)`

Limit value to a range softly to prevent discontinuous gradient

**Parameters**

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **lower** (*float*) – The lower threshold
- **upper** (*float*) – The upper threshold
- **scale** – A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

**See also:**

`soft_limit`

`npsolve.soft_functions.floor(x, limit=0.0, scale=0.0001)`

Limit value to a minimum softly to prevent discontinuous gradient

**Parameters**

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **limit** (*float*) – [OPTIONAL] The value to limit at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

**See also:**

`soft_limit`

`npsolve.soft_functions.gaussian(x, center=0.0, scale=0.0001)`

A gaussian function, with a peak of 1.0

**Parameters**

- **x** (*int, float, ndarray*) – The value(s)
- **center** (*float*) – [OPTIONAL] The x-position of the peak center
- **scale** (*float*) – [OPTIONAL] A scale factor for the curve.

`npsolve.soft_functions.lim(x, limit=0.0, side=1, scale=0.0001)`

Limit the value softly to prevent discontinuous gradient

**Parameters**

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **limit** (*float*) – [OPTIONAL] The value to limit at. Defaults to 0.
- **side** (*int*) – [OPTIONAL] 1 for min, -1 for max. Defaults to 1.

- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

---

**Note:** This function uses a softplus function to perform smoothing. See [https://en.wikipedia.org/wiki/Activation\\_function](https://en.wikipedia.org/wiki/Activation_function). Values for the calculation are clipped to 700 avoid overflow errors, as the max value for a float is exp(709.782).

---

`npsolve.soft_functions.negdiff(x, limit=0.0, scale=0.0001)`

Negative-only difference (difference below limit to 0 above limit)

**Parameters**

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **limit** (*float*) – [OPTIONAL] The value to limit at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

**See also:**

`soft_limit`

`npsolve.soft_functions.outside(x, lower, upper, scale=0.0001)`

Steps smoothly from 1 outside a range to 0 inside it

**Parameters**

- **x** (*int, float, ndarray*) – The value(s)
- **lower** (*float*) – The lower threshold
- **upper** (*float*) – The upper threshold
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

**See also:**

`soft_step`

`npsolve.soft_functions.posdiff(x, limit=0.0, scale=0.0001)`

Positive-only difference (0 below limit to difference above limit)

**Parameters**

- **x** (*int, float, ndarray*) – The value(s) to soft limit
- **limit** (*float*) – [OPTIONAL] The value to limit at. Defaults to 0.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** The limited value(s)

**Return type** float, ndarray

See also:

`soft_limit`

`npsolve.soft_functions.sign(x, scale=0.0001)`

A smooth step from -1 below 0 to +1 above it

**Parameters**

- **x** (*int*, *float*, *ndarray*) – The value(s)
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

---

**Note:** This function uses a sigmoid function to perform smoothing. See [https://en.wikipedia.org/wiki/Sigmoid\\_function](https://en.wikipedia.org/wiki/Sigmoid_function). Values for the calculation are clipped to avoid overflow errors.

---

`npsolve.soft_functions.step(x, limit=0.0, side=1, scale=0.0001)`

A smooth step to prevent discontinuous gradient

**Parameters**

- **x** (*int*, *float*, *ndarray*) – The value(s)
- **limit** (*float*) – [OPTIONAL] The value to step at. Defaults to 0.
- **side** (*int*) – [OPTIONAL] 1 for min, -1 for max. Defaults to 1.
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

---

**Note:** This function uses a sigmoid function to perform smoothing. See [https://en.wikipedia.org/wiki/Sigmoid\\_function](https://en.wikipedia.org/wiki/Sigmoid_function). Values for the calculation are clipped to avoid overflow errors.

---

`npsolve.soft_functions.within(x, lower, upper, scale=0.0001)`

Steps smoothly from 0 outside a range to 1 inside it

**Parameters**

- **x** (*int*, *float*, *ndarray*) – The value(s)
- **lower** (*float*) – The lower threshold
- **upper** (*float*) – The upper threshold
- **scale** (*float*) – [OPTIONAL] A scale factor for the softening

**Returns** Value(s) between 0 and 1

**Return type** float, ndarray

See also:

`soft_step`

## 2.5 npsolve.solvers module

Created on Fri May 22 10:36:19 2020

@author: Reuben

This module contains more specialised solvers based on scipy.

```
class npsolve.solvers.Integrator (status, logger, framerate=60.0, interface_cls=None, integrator_name='lsoda', squeeze=True, x_name='time', update_inits=False, **kwargs)
```

Bases: `npsolve.core.Solver`

A versatile integrator, with extra logging and stop flag

This integrator allows variables to be logged during the integration, which are then included in the output. In addition, Partial instances can set a flag to stop the integration at any point.

### Parameters

- **status** (*defaultdict*) – A dictionary that contains status flags. Key flags are `npsolve.solvers.FINAL` and `npsolve.solvers.STOP` (which are strings). The default should be a function that returns None. Obtain one by calling `npsolve.get_status(<name>)`.
- **logger** (*defaultdict*) – A dictionary in which the default values are lists. Obtain one by calling `npsolve.get_solver(<name>)`.
- **framerate** (*float*) – [OPTIONAL] The number of return values per unit x (which is often time). Defaults to 60.0.
- **interface\_cls** (*class*) – [OPTIONAL] The class of interface to use for integrator algorithms. Defaults to `scipy.integrate.ode` if scipy is found.
- **integrator\_name** (*str*) – [OPTIONAL] The name of the integrator to use. Defaults to 'lsoda'.
- **squeeze** (*bool*) – [OPTIONAL] Ensure output arrays are squeezed. Defaults to True.
- **x\_name** (*str*) – [OPTIONAL]: The name for the x value, which is logged in the outputs. Defaults to 'time'.
- **update\_inits** (*bool*) – [OPTIONAL] Update the initial values of the Partial instances with the solution at the end. Defaults to False.
- **keyword arguments** (*Other*) – [OPTIONAL] Are passed to the integrator by the call `interface_cls.set_integrator(integrator_name, **kwargs)`.

**Returns** A dictionary of integrated values. The values are ndarrays, which are at the framerate specified by the 'framerate' argument.

**Return type** dict

**Adding logged variables:** Only log variables when the solver has finalised the current frame. Integrators like scipy's ode 'lsoda' use variable time steps, and take numerous guesses at the state as they jump around in time. Once it has reached an accurate state for the x value at the end of the frame, `status[npsolve.solvers.FINAL]` is set to True. Only log values when this flag is True. An example:

```
if status[FINAL]:
    logger['variable_name_1'] = current_value
```

**Stopping the integration:** Stop the integration by setting `status[npsolve.solvers.STOP]` to True.

**run** (*end*, *\*\*kwargs*)

Run the solver

**Parameters** *end* (*float*) – The end point for the integration. Integration starts from 0 and will end at this value. Often this is a time.

**Returns** A dictionary where keys are the variable names and other logged names, and the values are ndarrays of the values through time.

**Return type** dict

## 2.6 npsolve.utils module

Created on Sat Sep 14 08:31:52 2019

@author: Reuben

This module contains helper functions and methods that help you get the most from npsolve.

**class** npsolve.utils.Dict\_Container

Bases: dict

**class** npsolve.utils.List\_Container

Bases: dict

**class** npsolve.utils.List\_Container\_Container

Bases: dict

**class** npsolve.utils.Set\_Container

Bases: dict

**class** npsolve.utils.Timeseries (*xs*, *ys*)

Bases: object

A utility class to specify values from time-series data

### Parameters

- **xs** (*ndarray*) – A 1D array of x values. Must be monotonically increasing.
- **ys** (*ndarray*) – A 1D array of y values

**Usage:** The Timeseries class is callable. It interpolates values smoothly between the inputs using splines. It offers the 1st integral to the 3rd derivative of values.

```
timeseries = Timeseries(xs, ys) # Create
timeseries(5) # Value at x=5
timeseries(5, -1) # 1st integral (antiderivative) at x=5
timeseries(5, 0) # Value at x=5
timeseries(5, 1) # 1st derivative at x=5
timeseries(5, 2) # 2nd derivative at x=5
timeseries(5, 3) # 3rd derivative at x=5
```

---

**Note:** Use *Timeseries.from\_csv* to generate values from a csv file.

---

**classmethod** **from\_csv** (*fname*, *x\_col=0*, *y\_col=1*, *skip\_header=0*, *delimiter=''*, *\*\*kwargs*)

Create a Timeseries from csv data

### Parameters

- **fname** (*str*) – The filename
- **x\_col** (*int*) – The column index of the x data (0 is the first column)
- **y\_col** (*int*) – The column index of the y data
- **skip\_header** (*int*) – [Optional] Number of header rows to skip. Defaults to 0.
- **delimiter** (*str*) – [Optional] Delimiter. Defaults to ‘,’.
- **\*\*kwargs** – [Optional] Other keyword arguments passed to `numpy.genfromtxt`.

**get** (*x*, *der=0*, *ext=3*)

Get an interpolated value

#### Parameters

- **x** (*float*, *ndarray*) – The x value(s).
- **der** (*int*) – [Optional] The derivative number. Defaults to 0.
- **ext** (*int*) – [Optional] What to do outside the range of xs. See `scipy.interpolate.splev` for details. Defaults to 3, which means to return the boundary values.

**Returns** The y value(s)

**Return type** `ndarray`

`npsolve.utils.get_dict` (*name*)

`npsolve.utils.get_list` (*name*)

`npsolve.utils.get_list_container` (*name*)

`npsolve.utils.get_logger` (*name*)

`npsolve.utils.get_set` (*name*)

`npsolve.utils.get_status` (*name*)

`npsolve.utils.none` ()





---

### Related packages

---

Here's a list of some related packages. None provide the same functionality as *npsolve*, but they might be useful for you.

- Scipy (Just use the solvers directly)
- Odespy (Class-based models possible, but it looks bit less flexible.)
- Pymunk (2D rigid body motion)
- PyDy (Based on symbolic equations)
- Hamilton (not much documentation?)
- Arboris (robotics)
- SymPy mechanics (a great package, and somewhat object-oriented, but since it is symbolic, it will struggle with discontinuities.)
- PyODE (possibly dead?)
- Odelab (Class-based to some degree, but it looks less flexible.)
- Pyndamics (ODE wrapper)
- ARS (Robotics)



## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### n

- `npsolve`, [33](#)
- `npsolve.cache`, [36](#)
- `npsolve.core`, [33](#)
- `npsolve.soft_functions`, [37](#)
- `npsolve.solvers`, [41](#)
- `npsolve.utils`, [42](#)



## A

above() (in module *npsolve.soft\_functions*), 37  
 add\_var() (*npsolve.core.Partial method*), 33  
 add\_vars() (*npsolve.core.Partial method*), 34  
 as\_dct() (*npsolve.core.Solver method*), 34

## B

below() (in module *npsolve.soft\_functions*), 37

## C

cache\_clear() (*npsolve.core.Partial method*), 34  
 ceil() (in module *npsolve.soft\_functions*), 37  
 clear\_vars() (*npsolve.core.Partial method*), 34  
 clip() (in module *npsolve.soft\_functions*), 38  
 close\_signals() (*npsolve.core.Solver method*), 35  
 connect() (*npsolve.core.Partial method*), 34  
 connect() (*npsolve.core.Solver method*), 35

## D

Dict\_Container (class in *npsolve.utils*), 42

## E

enable\_caching() (*npsolve.core.Partial method*), 34

## F

fetchpartials() (*npsolve.core.Solver method*), 35  
 floor() (in module *npsolve.soft\_functions*), 38  
 freeze() (*npsolve.core.Solver method*), 35  
 from\_csv() (*npsolve.utils.Timeseries class method*), 42

## G

gaussian() (in module *npsolve.soft\_functions*), 38  
 get() (*npsolve.utils.Timeseries method*), 43  
 get\_dict() (in module *npsolve.utils*), 43  
 get\_init() (*npsolve.core.Partial method*), 34  
 get\_list() (in module *npsolve.utils*), 43

get\_list\_container() (in module *npsolve.utils*), 43

get\_logger() (in module *npsolve.utils*), 43

get\_set() (in module *npsolve.utils*), 43

get\_state\_dct() (*npsolve.core.Solver method*), 35

get\_status() (in module *npsolve.utils*), 43

## I

Integrator (class in *npsolve.solvers*), 41

## L

lim() (in module *npsolve.soft\_functions*), 38

List\_Container (class in *npsolve.utils*), 42

List\_Container\_Container (class in *npsolve.utils*), 42

## M

mono\_cached() (in module *npsolve.cache*), 36

multi\_cached() (in module *npsolve.cache*), 37

## N

negdiff() (in module *npsolve.soft\_functions*), 39

none() (in module *npsolve.utils*), 43

npsolve (module), 33

npsolve.cache (module), 36

npsolve.core (module), 33

npsolve.soft\_functions (module), 37

npsolve.solvers (module), 41

npsolve.utils (module), 42

npsolve\_finish() (*npsolve.core.Solver method*), 35

npsolve\_init() (*npsolve.core.Solver method*), 35

## O

one\_way\_step() (*npsolve.core.Solver method*), 35

outside() (in module *npsolve.soft\_functions*), 39

## P

Partial (class in *npsolve.core*), 33

pinned() (*npsolve.core.Solver method*), 35

`posdiff()` (*in module npsolve.soft\_functions*), 39

## R

`remove_signals()` (*npsolve.core.Solver method*), 35

`run()` (*npsolve.solvers.Integrator method*), 41

## S

`Set_Container` (*class in npsolve.utils*), 42

`set_init()` (*npsolve.core.Partial method*), 34

`set_meta()` (*npsolve.core.Partial method*), 34

`set_vectors()` (*npsolve.core.Partial method*), 34

`setup_signals()` (*npsolve.core.Solver method*), 35

`sign()` (*in module npsolve.soft\_functions*), 40

`Solver` (*class in npsolve.core*), 34

`step()` (*in module npsolve.soft\_functions*), 40

`step()` (*npsolve.core.Partial method*), 34

`step()` (*npsolve.core.Solver method*), 36

## T

`Timeseries` (*class in npsolve.utils*), 42

`tstep()` (*npsolve.core.Solver method*), 36

## U

`unfreeze()` (*npsolve.core.Solver method*), 36

## W

`within()` (*in module npsolve.soft\_functions*), 40