
Dynamics and Control

Release 0.0.1

Carl Sandrock

Dec 29, 2023

CONTENTS

1	Getting Started	1
1.1	Introduction to Sympy and the Jupyter Notebook for engineering calculations	1
1.2	Python stuff not done in MPR	6
1.3	The Jupyter notebook cheat sheet	11
2	Dynamics	19
2.1	Modelling	19
2.2	Time domain simulation	22
2.3	Linear systems	55
2.4	First and second order system Dynamics	70
2.5	Complex system dynamics	87
2.6	Multivariable system representations	106
2.7	System identification	112
2.8	Frequency domain	129
2.9	Sampled systems	154
3	Control	169
3.1	Conventional feedback control	169
3.2	Laplace domain analysis of control systems	181
3.3	PID controller design, tuning and troubleshooting	187
3.4	Frequency domain analysis of control systems	194
3.5	Advanced control methods	197
3.6	Discrete control and analysis	201
3.7	Multivariable control	227
3.8	Control Practice	244
4	Simulation	247
4.1	No delays	247
4.2	Dead time	249
4.3	Nonlinear tank system	251
4.4	PI Control	253
4.5	Classes	254
4.6	Taking off the engine cover	256
4.7	Objects	259
4.8	A discrete controller class	263
4.9	Blocksim	266
4.10	Disturbances	268
4.11	Algebraic equations	270
5	Temperature Control Lab (TCLab)	273

5.1	FOPDT fit	273
5.2	TCLab in the frequency domain	275
6	Search Page	283

GETTING STARTED

1.1 Introduction to Sympy and the Jupyter Notebook for engineering calculations

Sympy is a computer algebra module for Python. You are looking at the convenient [Jupyter Notebook](#) interface. This notebook aims to show some of the useful features of the Sympy system as well as the notebook interface.

This notebook will use Python as the programming language. This means that most of what you learned in MPR can be applied in the notebook. The notebook interface provides “cells” where one can input code. To run the code, click on a cell and press Shift+Enter.

1.1.1 A quick tour

Take a second to go through the tour of the notebook interface by clicking on “Help, User Interface Tour”. Also note that there is help available for a number of other things under that menu.

Now that you are familiar with the nomenclature, let’s run some code!

Evaluate the cell below to print out a message by clicking inside the cell and then pressing Shift + Enter

```
[1]: for word in ['Hello', 'World']:
      print(word)
```

```
Hello
World
```

```
[2]: a = 1
```

1.1.2 Math in text boxes

The text editor supports math in `.math:LaTeX`<>`` notation. You can double-click on a text box to see the codes used to enter it:

$$f(a) = \int_{\infty}^0 \frac{1}{a+2} da$$

Double-click on the formula above to see the code that produced it.

1.1.3 Special symbols in variable names

The notebook supports easy entry of special symbols in variable names. Simply type a backslash with the name of the symbol, then press tab to have it replaced by the symbol. For example:

Enter `\alpha`, then press tab. This will be replaced by α

```
[3]:  $\alpha = 1$ 
```

1.1.4 SymPy

We need to import the `SymPy` module to get symbolic math capabilities.

```
[4]: import sympy
```

We need to start the pretty-printer to get nicely typeset math

Note that this changes somewhat based on the version of sympy

```
[5]: sympy.init_printing()
```

In order to do symbolic calculations, we need to create a symbol

```
[6]: x = sympy.Symbol('x')
```

```
[7]: x
```

```
[7]:  $x$ 
```

Sympy allows us to do many mathematical operations that would be tedious by hand. For instance, we can expand a polynomial:

```
[8]: polynomial = (2*x + 3)**4  
polynomial.expand()
```

```
[8]:  $16x^4 + 96x^3 + 216x^2 + 216x + 81$ 
```

Notice what happened - we defined a new name called “polynomial” and then used the `.expand()` method to expand the polynomial. We can see all the methods associated with an object by typing its name and a dot then pressing “tab”.

Call up the list of methods for the polynomial variable by entering “.” and pressing tab at the end of the line in the cell below:

```
[9]: polynomial
```

```
[9]:  $(2x + 3)^4$ 
```

To get help about any method, we can type its name and append a `?` at the end, then evaluate the cell

Obtain help about the `.expand()` method by evaluating the cell below:

```
[10]: polynomial.expand()
```



```
[10]:
```

$$16x^4 + 96x^3 + 216x^2 + 216x + 81$$

It is also possible to obtain help for a function by placing the cursor between the parentheses and pressing Shift+Tab

Of course, we can also factor polynomials:

```
[11]: (x**2 + 2*x + 1).factor()
```

```
[11]:
```

$$(x + 1)^2$$

1.1.5 Calculus

Sympy knows how to integrate and differentiate

```
[12]: eq = sympy.tan(sympy.log(x**2 + 1))
```

```
[13]: eq.diff(x)
```

```
[13]:
```

$$\frac{2x (\tan^2(\log(x^2 + 1)) + 1)}{x^2 + 1}$$

```
[14]: polynomial.diff(x) # First derivative
```

```
[14]:
```

$$8(2x + 3)^3$$

```
[15]: polynomial.diff(x, 2) # Second derivative
```

```
[15]:
```

$$48(2x + 3)^2$$

```
[16]: polynomial.integrate(x) # indefinite integral - note no constant of integration is_
↳added
```

```
[16]:
```

$$\frac{16x^5}{5} + 24x^4 + 72x^3 + 108x^2 + 81x$$

```
[17]: polynomial.integrate((x, 1, 2)) # Note that integrate takes one argument which is a_
↳tuple for the definite integral
```

```
[17]:
```

$$\frac{6841}{5}$$

1.1.6 Limits

We can evaluate limits using SymPy, even for “interesting” limits where we would need L’Hopital’s rule

```
[18]: badeq = (2*sympy.sin(x) - sympy.sin(2*x))/(x - sympy.sin(x))
```

```
[19]: badeq
```

```
[19]:
```

$$\frac{2 \sin(x) - \sin(2x)}{x - \sin(x)}$$

```
[20]: a = sympy.symbols('a')
```

```
[21]: lim = sympy.limit(badeq, x, a)
```

```
[22]: lim
```

```
[22]:
```

$$-\frac{-2 \sin(a) + \sin(2a)}{a - \sin(a)}$$

```
[23]: lim.subs(sympy.sin(a), 1)
```

```
[23]:
```

$$-\frac{\sin(2a) - 2}{a - 1}$$

1.1.7 Approximation

SymPy has built-in support for Taylor series expansion

```
[24]: nonlinear_expression = sympy.sin(x)
sympy.series(nonlinear_expression, x, 2, 7) # Taylor expansion in terms of the x_
↪variable, around x=2, first order.
```

```
[24]:
```

$$\sin(2) + (x-2)\cos(2) - \frac{(x-2)^2 \sin(2)}{2} - \frac{(x-2)^3 \cos(2)}{6} + \frac{(x-2)^4 \sin(2)}{24} + \frac{(x-2)^5 \cos(2)}{120} - \frac{(x-2)^6 \sin(2)}{720} + O\left((x-2)^7; x \rightarrow 2\right)$$

To remove the order term use `.removeO()`

```
[25]: temp = sympy.series(nonlinear_expression, x, 2, 2)
temp.removeO()
```

```
[25]:
```

$$(x-2)\cos(2) + \sin(2)$$

You will also notice that SymPy’s default behaviour is to retain exact representations of certain numbers:

```
[26]: number = sympy.sqrt(2)*sympy.pi
number
```



```
[26]: 
$$\sqrt{2\pi}$$

```

To convert the exact representations above to an approximate **floating point** representations, use one of these methods. `sympy.N` works with complicated expressions containing variables as well. `float` will return a normal Python float and is useful when interacting with non-sympy programs.

```
[27]: sympy.N(number*x)
```

```
[27]: 4.44288293815837x
```

```
[28]: float(number)
```

```
[28]: 4.442882938158366
```

1.1.8 Solving equations

Sympy can help us solve/manipulate equations using the `solve` function. Like many solving functions, it finds zeros of a function, so we have to rewrite equalities to be equal to zero,

```
nb sphinx-math: begin{align}
2x^2 + 2 &= 4 \ 2x^2 + 2 - 4 &= 0 end{align}
```

```
[29]: sympy.plot(x**2 + 1)
```

```
<Figure size 640x480 with 1 Axes>
```

```
[29]: <sympy.plotting.plot.Plot at 0x10b531438>
```

```
[30]: solutions = sympy.solve(2*x**2 + 2 - 4)
solutions
```

```
[30]: [-1, 1]
```

```
[31]: solutions[0]
```

```
[31]: -1
```

We can also use `sympy.Eq` to construct equations

```
[32]: equation = sympy.Eq(2*x**2 + 2, 4)
equation
```

```
[32]: 
$$2x^2 + 2 = 4$$

```

The `roots` function will give us the multiplicity of the roots as well.


```
[33]: sympy.roots(equation)
```

```
[33]: {-1: 1, 1: 1}
```

The results are given as a dictionary. If this is not familiar to you, have a look in the [Extra Python notebook](#).

We can also solve systems of equations by passing a list of equations to solve and asking for a list of variables to solve for

```
[34]: x, y = sympy.symbols('x, y')
sympy.solve([x + y - 2,
            x - y - 0], [x, y])
```

```
[34]: {x: 1, y: 1}
```

This even works with symbolic variables in the equations

```
[35]: a, b, c = sympy.var('a, b, c')
solution = sympy.solve([a*x + b*y - 2,
                       a*x - b*y - c], [x, y])
solution
```

```
[35]: {x:  $\frac{c+2}{2a}$ , y:  $\frac{-c+2}{2b}$ }
```

```
[ ]:
```

1.2 Python stuff not done in MPR

1.2.1 List comprehensions

This is a common pattern - accumulating into a list:

```
[1]: inputs = [2, 1, 3, 2, 4, 5, 6]
result = [] # Start with an empty list
for i in inputs: # Iterate over an input list
    if i < 4: # if some condition holds
        result.append(i**2) # Append the result of a calculation
result
```

```
[1]: [4, 1, 9, 4]
```

It's common enough that Python includes dedicated syntax for this:

```
[2]: result = [i**2 for i in inputs if i < 4]
result
```

```
[2]: [4, 1, 9, 4]
```


1.2.2 Dictionaries

You should be familiar with lists, which are ordered container types.

```
[1]: lst = [1, 2, 3]
```

We can retrieve elements from the list by *indexing* it

```
[4]: lst[1]
```

```
[4]: 2
```

A dictionary gives us a container like a list, but the indexes can be much more general, not just numbers but strings or sympy variables (and a whole host of other types)

```
[5]: dic = {'a': 100, 2: 45, 100: 45}
     dic['a']
```

```
[5]: 100
```

When we solve an equation in sympy, the result is a dictionary

```
[6]: import sympy
     sympy.init_printing()
```

```
[7]: x, y = sympy.symbols('x, y')
```

```
[8]: solution = sympy.solve([x - y, 2 + 2*x + y], [x, y])
```

```
[9]: solution
```

```
[9]: 
$$\left\{ x: -\frac{2}{3}, \quad y: -\frac{2}{3} \right\}$$

```

```
[10]: type(solution)
```

```
[10]: dict
```

This means we can find the value of one of the answers by indexing.

```
[11]: solution[x]
```

```
[11]: 
$$-\frac{2}{3}$$

```


1.2.3 Tuples

You are familiar with lists:

```
[12]: x = 1, 2, 3

      a, b, c = x

      a, b, c = 1, 2, 3
```

```
[13]: def f(x):
      Ca, Cb, Cc = x
```

```
[14]: l = [1, 2, 3, 4]
```

```
[15]: type(l)
```

```
[15]: list
```

Tuples are like lists, but they are created with commas:

```
[16]: t = 1, 2, 3, 4
```

```
[17]: type(t)
```

```
[17]: tuple
```

In some cases it is useful to use parentheses to group tuples (but note that they are not required syntax:

```
[18]: t2 = (1, 2, 3, 4)
```

```
[19]: type(t2)
```

```
[19]: tuple
```

It is important to understand that the comma, not the parentheses make tuples:

```
[20]: only_one = ((((((1))))))
```

```
[21]: type(only_one)
```

```
[21]: int
```

```
[22]: only_one = 1,
```

```
[23]: type(only_one)
```

```
[23]: tuple
```

```
[24]: len(only_one)
```

```
[24]: 1
```


The only exception to this rule is that an empty tuple is built with `()`:

```
[25]: empty = ()
```

```
[26]: type(empty)
```

```
[26]: tuple
```

```
[27]: len(empty)
```

```
[27]: 0
```

The differences between tuples and lists are that tuples are immutable (they cannot be changed in place)

```
[28]: l.append(1)
```

If we were to run

```
t.append(1)
```

We would see

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-41-c860940312ad> in <module>()
----> 1 t.append(1)

AttributeError: 'tuple' object has no attribute 'append'
```

Tuple expansion

A very useful and general feature of the assignment operator in Python is that tuples will be expanded and assigned in matched patterns:

```
[29]: a, b = 1, 2
```

This is quite sophisticated and can handle nested structures and expanded to lists:

```
[30]: [(a, b), c, d] = [(1, 2), 3, 4]
```

1.2.4 The for loop in Python

This [talk](#) is excellent for understanding the way that Python “wants” to use the for loop

zip

Let's say we're trying to calculate the credit-weighted average of a student's marks using loops:

```
[31]: credits = [8, 16, 8, 8, 16]
      marks = [75, 60, 60, 75, 45]
```

One way is to use indexing:

```
[32]: weightedsum = 0
      creditsum = 0
      for i in range(len(credits)):
          weightedsum += credits[i]*marks[i]
          creditsum += credits[i]

      avg = weightedsum/creditsum
      print(avg)

60.0
```

But Python supplies a method which allows us to iterate directly over the pairs:

```
[33]: weightedsum = 0
      creditsum = 0
      for credit, mark in zip(credits, marks):
          weightedsum += credit*mark
          creditsum += credit

      avg = weightedsum/creditsum
      print(avg)

60.0
```

This `zip` function returns an iterator which groups its inputs into tuples, suitable for expansion in the `for` loop. We can see the effect if we convert to a list:

```
[34]: list(zip(credits, marks))
[34]: [(8, 75), (16, 60), (8, 60), (8, 75), (16, 45)]
```

These pairs are then assigned out to the arguments in the `for` loop above

1.2.5 lambda

Some functions expect functions as arguments. For instance, `scipy.optimize.fsolve` solves equations numerically:

```
[35]: def f(x):
      return x**2 - 3
```

```
[36]: import scipy
```

```
[37]: import scipy.optimize
```



```
[38]: scipy.optimize.fsolve(f, 2)
```

```
[38]: array([1.73205081])
```

For very simple functions, `lambda` allows us to construct functions in a more compact way and not give them a name:

```
[39]: scipy.optimize.fsolve(lambda x: x**2 - 3, 2)
```

```
[39]: array([1.73205081])
```

The function constructed by `lambda` works the same as the one constructed by `def` in most ways. My recommendation is to use `lambda` with caution. It is never *necessary* to use `lambda`. I include this section mostly so that you can understand what this does if you encounter it in documentation.

1.3 The Jupyter notebook cheat sheet

This document will be available to you during tests and exams

1.3.1 Table of Contents

Numeric

Basic plotting functions

Symbolic manipulation

Equation solving

Matrix math

```
[1]: import tbcontrol
      tbcontrol.expectversion('0.1.2')
```

1.3.2 Numeric

```
[2]: import numpy
      import scipy
```

```
[3]: a = numpy.array([1, 2, 3])
```

```
[4]: t = numpy.linspace(0, 10)
```

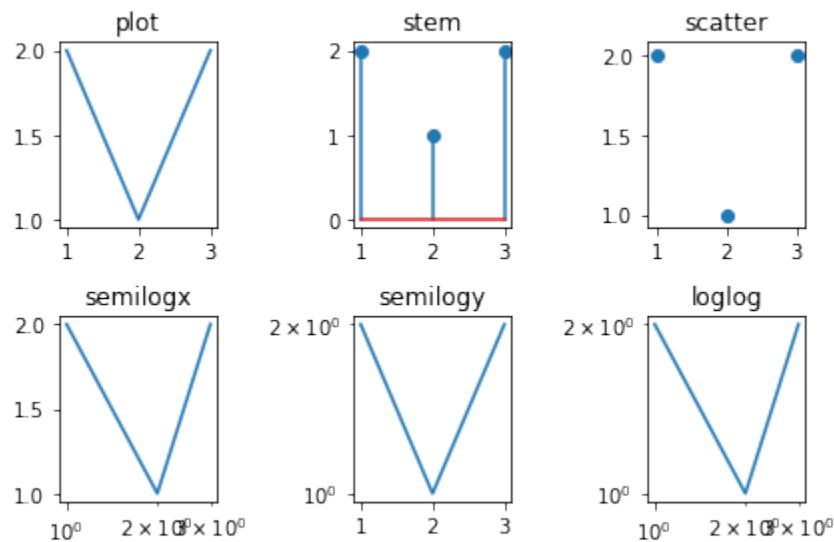
1.3.3 Basic plotting functions

```
[5]: import matplotlib.pyplot as plt
      %matplotlib inline
```



```
[6]: plotfuncs = [plt.plot,
                  plt.stem,
                  plt.scatter,
                  plt.semilogx,
                  plt.semilogy,
                  plt.loglog]

for i, func in enumerate(plotfuncs, 1):
    plt.subplot(2, 3, i)
    func([1, 2, 3], [2, 1, 2])
    plt.title(func.__name__)
plt.tight_layout()
```



1.3.4 Symbolic manipulation

Imports

```
[7]: import sympy
sympy.init_printing()
```

Symbol definitions

```
[8]: s = sympy.Symbol('s') # A single symbol
tau, K_c = sympy.symbols('tau K_c', positive=True) # we can use real=True or
↳ complex=True for other kinds of variables
```

Example controller and system

```
[9]: Gc = K_c * ((tau*s + 1) / (tau*s))
GvGpGm = 5 / ((10*s + 1)**2)
```


Working with rational functions and polynomials

We often want nice rational functions, but sympy doesn't make expressions rational by default

```
[10]: chareq = GvGpGm*Gc + 1
      chareq
```

```
[10]:
```

$$\frac{5K_c(s\tau + 1)}{s\tau(10s + 1)^2} + 1$$

The `cancel` function forces this to be a fraction. `collect` collects terms.

```
[11]: chareq = chareq.cancel().collect(s)
      chareq
```

```
[11]:
```

$$\frac{5K_c + 100s^3\tau + 20s^2\tau + s(5K_c\tau + \tau)}{100s^3\tau + 20s^2\tau + s\tau}$$

In some cases we can factor equations:

```
[12]: chareq.factor(s)
```

```
[12]:
```

$$\frac{5K_c + 100s^3\tau + 20s^2\tau + s(5K_c\tau + \tau)}{s\tau(10s + 1)^2}$$

Obtain the numerator and denominator:

```
[13]: sympy.numer(chareq), sympy.denom(chareq)
```

```
[13]:
```

$$(5K_c + 100s^3\tau + 20s^2\tau + s(5K_c\tau + \tau), \quad 100s^3\tau + 20s^2\tau + s\tau)$$

If you want them both, you can use

```
[14]: chareq.as_numer_denom()
```

```
[14]:
```

$$(5K_c + 100s^3\tau + 20s^2\tau + s(5K_c\tau + \tau), \quad 100s^3\tau + 20s^2\tau + s\tau)$$

Convert to polynomial in s

```
[15]: numer = sympy.poly(sympy.numer(chareq), s)
```

Once we have a polynomial, it is easy to obtain coefficients:

```
[16]: numer.all_coeffs()
```

```
[16]:
```

$$[100\tau, \quad 20\tau, \quad 5K_c\tau + \tau, \quad 5K_c]$$

Calculate the Routh Array


```
[17]: from tbcontrol.symbolic import routh
```

```
[18]: routh(number)
```

```
[18]:
```

$$\begin{bmatrix} 100\tau & 5K_c\tau + \tau \\ 20\tau & 5K_c \\ -25K_c + \tau(5K_c + 1) & 0 \\ 5K_c & 0 \end{bmatrix}$$

To get a function which can be used numerically, use `lambdify`:

```
[19]: f = sympy.lambdify((K_c, tau), K_c + tau)
```

```
[20]: f(1, 2)
```

```
[20]:
```

$$3$$

Functions useful for discrete systems

```
[21]: z, q = sympy.symbols('z, q')
```

```
[22]: Gz = z**(-1)/(1 - z**(-1))  
Gz
```

```
[22]:
```

$$\frac{1}{z\left(1 - \frac{1}{z}\right)}$$

Write in terms of positive powers of z :

```
[23]: Gz.cancel()
```

```
[23]:
```

$$\frac{1}{z - 1}$$

Write in terms of negative powers of z :

```
[24]: Gz.subs({z: q**(-1)}).cancel()
```

```
[24]:
```

$$-\frac{q}{q - 1}$$

Inversion of the z transform

```
[25]: from tbcontrol.symbolic import sampledvalues
```

```
[26]: sampledvalues(Gz, z, 10)
```

```
[26]:
```

$$[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

1.3.5 Equation solving

Symbolic

```
[27]: x, y, z, a = sympy.symbols('x, y, z, a')
      residuals = [x + y - 2, y + z - a, x + y + z]
      unknowns = [x, y, z]
      sympy.solve(residuals, unknowns)
```

```
[27]: {x: -a, y: a + 2, z: -2}
```

Numeric sympy

```
[28]: residuals = [2*x**2 - 2*y**2, sympy.sin(x) + sympy.log(y)]
      unknowns = [x, y]
      sympy.nsolve(residuals, unknowns, [1, 3])
```

```
[28]: [-2.21910714891375]
      [ 2.21910714891375]
```

Numeric

```
[29]: import scipy.optimize
```

```
[30]: def residuals(unknowns):
      x, y = unknowns
      return [2*x**2 - 2*y**2, numpy.sin(x) + numpy.log(y)]
```

```
[31]: starting_point = [1, 3]
```

```
[32]: residuals(starting_point)
```

```
[32]: [-16, 1.9400832734760063]
```

```
[33]: scipy.optimize.fsolve(residuals, starting_point)
```

```
[33]: array([-2.21910715,  2.21910715])
```


1.3.6 Matrix math

Symbolic

```
[34]: G11, G12, G21, G22 = sympy.symbols('G11, G12, G21, G22')
```

Creation

```
[35]: G = sympy.Matrix([[G11, G12], [G21, G22]])
G
```

```
[35]: 
$$\begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix}$$

```

Determinant, inverse, transpose

```
[36]: G.det(), G.inv(), G.T
```

```
[36]: 
$$\left( G_{11}G_{22} - G_{12}G_{21}, \begin{bmatrix} \frac{G_{22}}{G_{11}G_{22} - G_{12}G_{21}} & -\frac{G_{12}}{G_{11}G_{22} - G_{12}G_{21}} \\ -\frac{G_{21}}{G_{11}G_{22} - G_{12}G_{21}} & \frac{G_{11}}{G_{11}G_{22} - G_{12}G_{21}} \end{bmatrix}, \begin{bmatrix} G_{11} & G_{21} \\ G_{12} & G_{22} \end{bmatrix} \right)$$

```

Math operations: Multiplication, addition, elementwise multiplication:

```
[37]: G*G, G+G, G.multiply_elementwise(G)
```

```
[37]: 
$$\left( \begin{bmatrix} G_{11}^2 + G_{12}G_{21} & G_{11}G_{12} + G_{12}G_{22} \\ G_{11}G_{21} + G_{21}G_{22} & G_{12}G_{21} + G_{22}^2 \end{bmatrix}, \begin{bmatrix} 2G_{11} & 2G_{12} \\ 2G_{21} & 2G_{22} \end{bmatrix}, \begin{bmatrix} G_{11}^2 & G_{12}^2 \\ G_{21}^2 & G_{22}^2 \end{bmatrix} \right)$$

```

Numeric

Creation

```
[38]: G = numpy.matrix([[1, 2], [3, 4]])
```

Determinant, inverse, transpose

```
[39]: numpy.linalg.det(G), G.I, G.T
```

```
[39]: (-2.0000000000000004, matrix([[ -2. ,  1. ],
 [ 1.5, -0.5]]), matrix([[1, 3],
 [2, 4]]))
```

Math operations: Multiplication, addition, elementwise multiplication:

```
[40]: G*G, G+G, G.A*G.A
```

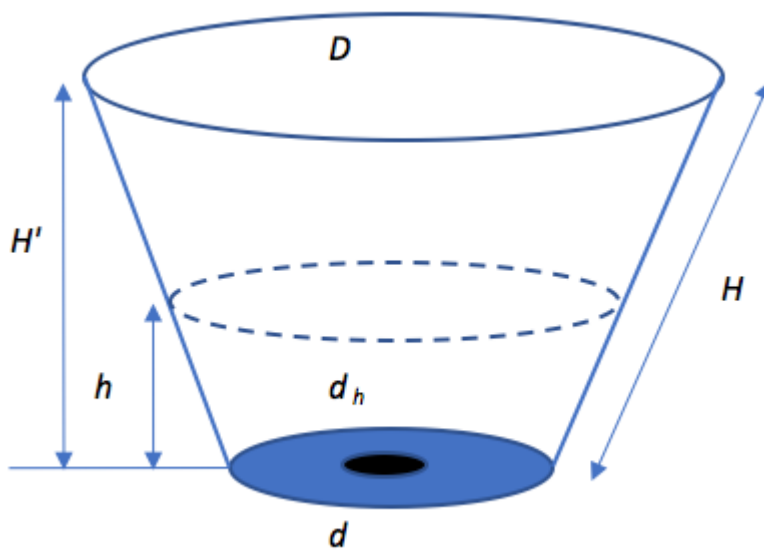
```
[40]: (matrix([[ 7, 10],
 [15, 22]]), matrix([[2, 4],
 [6, 8]]), array([[ 1,  4],
 [ 9, 16]]))
```


[]:

2.1 Modelling

2.1.1 The draining cup problem

In this notebook we will be modelling a draining cup. We assume the cup is shaped like a **conical frustum** or truncated cone:



D and d are the top and bottom diameters of the cup, H is the side length between the diameters, d_h is the hole diameter. We also define H' as the vertical height of the cup and h as the vertical height (or level) of the liquid in the cup

Volume-height relationship

Let's work out the relationship between the volume of water and the level in the cup by integrating the area:

$$V = \int_0^h A(h)dh$$

```
[1]: import sympy
sympy.init_printing()

[2]: D, d, H, h = sympy.symbols('D, d, H, h', real=True)

[3]: R = D/2
r = d/2
Hprime = sympy.sqrt(H**2 - (R - r)**2) # Pythagoras
```

The radius changes linearly from the small one to the large one:

```
[4]: radius = r + h/Hprime*(R - r)
```

Now it is easy to calculate the area:

```
[5]: A = sympy.pi*radius**2
```

And from there, the volume:

```
[6]: V = sympy.integrate(A, (h, 0, h))
```

```
[7]: V
```

```
[7]: 
$$\frac{\pi d^2 h}{4} - \frac{h^3 (\pi D^2 - 2\pi Dd + \pi d^2)}{3D^2 - 6Dd - 12H^2 + 3d^2} - \frac{h^2 (-\pi Dd + \pi d^2)}{2\sqrt{-D^2 + 2Dd + 4H^2 - d^2}}$$

```

```
[8]: print(V)

pi*d**2*h/4 - h**3*(pi*D**2 - 2*pi*D*d + pi*d**2)/(3*D**2 - 6*D*d - 12*H**2 + 3*d**2) -
h**2*(-pi*D*d + pi*d**2)/(2*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))
```

Dynamic model

The basic model of the cup we will be working with looks something like this:

$\frac{dV}{dt} = -F_{\text{out}}$	Mass Balance simplified to volume balance	(2.1)
$F_{\text{out}} = f(h)$	Hydraulics	(2.2)
$h = f(V)$	Geometry	(2.3)
		(2.4)

The above geometric description allows us to find the $V(h)$, but we actually want $h(V)$.


```
[17]: Vsymb = sympy.symbols('V', real=True)
```

```
[20]: hV = sympy.solve(Vsymb - V, h)
```

```
[21]: hV
```

```
[21]:
```

$$\left[\sqrt[3]{\frac{-4 \left(-\frac{3(-3D^2d^2+6Dd^3+12H^2d^2-3d^4)}{4D^2-8Dd+4d^2} + \frac{(3D^2d-6Dd^2-12H^2d+3d^3)^2}{(-2D\sqrt{-D^2+2Dd+4H^2-d^2}+2d\sqrt{-D^2+2Dd+4H^2-d^2})^2} \right)^3 + \left(\frac{27(3D^2V-6DVd-12H^2V+3Vd^2)}{\pi D^2-2\pi Dd+\pi d^2} - \frac{9(3D^2d-6Dd^2-12H^2d+3d^3)}{(-2D\sqrt{-D^2+2Dd+4H^2-d^2}+2d\sqrt{-D^2+2Dd+4H^2-d^2})^2} \right)}{2}} \right]$$

```
[19]: #print(hV)
```

```
-(-3*(-3*D**2*d**2 + 6*D*d**3 + 12*H**2*d**2 - 3*d**4)/(4*D**2 - 8*D*d + 4*d**2) +
↳ (3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)**2/(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 -
↳ d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))**2)/(3*(sqrt(-4*(-3*(-3*D**2*d**2 -
↳ 6*D*d**3 + 12*H**2*d**2 - 3*d**4)/(4*D**2 - 8*D*d + 4*d**2) + (3*D**2*d -
↳ 6*D*d**2 - 12*H**2*d + 3*d**3)**2/(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) +
↳ 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))**2)**3 + (27*(3*D**2*V - 6*D*V*d -
↳ 12*H**2*V + 3*V*d**2)/(pi*D**2 - 2*pi*D*d + pi*d**2) - 9*(3*D**2*d - 6*D*d**2 -
↳ 12*H**2*d + 3*d**3)*(-3*D**2*d**2 + 6*D*d**3 + 12*H**2*d**2 - 3*d**4)/((-2*D*sqrt(-
↳ D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))*(4*D**2 -
↳ 8*D*d + 4*d**2)) + 2*(3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)**3/(-2*D*sqrt(-D**2 -
↳ 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))**3)**2)/2 +
↳ 27*(3*D**2*V - 6*D*V*d - 12*H**2*V + 3*V*d**2)/(2*(pi*D**2 - 2*pi*D*d + pi*d**2)) -
↳ 9*(3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)*(-3*D**2*d**2 + 6*D*d**3 +
↳ 12*H**2*d**2 - 3*d**4)/(2*(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-
↳ D**2 + 2*D*d + 4*H**2 - d**2))*(4*D**2 - 8*D*d + 4*d**2)) + (3*D**2*d - 6*D*d**2 -
↳ 12*H**2*d + 3*d**3)**3/(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 +
↳ 2*D*d + 4*H**2 - d**2))**3)**(1/3)) - (sqrt(-4*(-3*(-3*D**2*d**2 + 6*D*d**3 +
↳ 12*H**2*d**2 - 3*d**4)/(4*D**2 - 8*D*d + 4*d**2) + (3*D**2*d - 6*D*d**2 - 12*H**2*d -
↳ 3*d**3)**2/(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d +
↳ 4*H**2 - d**2))**2)**3 + (27*(3*D**2*V - 6*D*V*d - 12*H**2*V + 3*V*d**2)/(pi*D**2 -
↳ 2*pi*D*d + pi*d**2) - 9*(3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)*(-3*D**2*d**2 +
↳ 6*D*d**3 + 12*H**2*d**2 - 3*d**4)/((-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) +
↳ 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))*(4*D**2 - 8*D*d + 4*d**2)) + 2*(3*D**2*d -
↳ 6*D*d**2 - 12*H**2*d + 3*d**3)**3/(-2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) +
↳ 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2))**3)**2)/2 + 27*(3*D**2*V - 6*D*V*d -
↳ 12*H**2*V + 3*V*d**2)/(2*(pi*D**2 - 2*pi*D*d + pi*d**2)) - 9*(3*D**2*d - 6*D*d**2 -
↳ 12*H**2*d + 3*d**3)*(-3*D**2*d**2 + 6*D*d**3 + 12*H**2*d**2 - 3*d**4)/(2*(-
↳ 2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 -
↳ d**2))*(4*D**2 - 8*D*d + 4*d**2)) + (3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)**3/(-
↳ 2*D*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 -
↳ d**2))**3)**(1/3)/3 - (3*D**2*d - 6*D*d**2 - 12*H**2*d + 3*d**3)/(3*(-2*D*sqrt(-
↳ D**2 + 2*D*d + 4*H**2 - d**2) + 2*d*sqrt(-D**2 + 2*D*d + 4*H**2 - d**2)))
```

```
[ ]:
```


2.2 Time domain simulation

2.2.1 Equation solving tools

We distinguish between root finding or solving algebraic equations and solving differential equations.

It is also useful to distinguish between approximate solution using numeric methods and exact solution.

Exact solution using sympy

We can solve systems of equations exactly using sympy's `solve` function. This is usually done using what is known as the residual form. The residual is simply the difference between the LHS and RHS of an equation, or put another way, we rewrite our equations to be equal to zero:

$$\begin{aligned}x + y &= z & (2.5) \\ \therefore x + y - z &= 0 & (2.6)\end{aligned}$$

```
[1]: import sympy
sympy.init_printing()
%matplotlib inline
```

```
[2]: x, y, z = sympy.symbols('x, y, z')
```

```
[3]: sympy.solve(x + y - z, z)
```

```
[3]:  $[x + y]$ 
```

We can solve systems of equations using `solve` as well, by passing a list of equations

```
[4]: equations = [x + y - z,
                  2*x + y + z + 2,
                  x - y - z + 2]
unknowns = [x, y, z]
```

```
[5]: solution = sympy.solve(equations, unknowns)
solution
```

```
[5]:  $\left\{x: -\frac{4}{3}, \quad y: 1, \quad z: -\frac{1}{3}\right\}$ 
```

```
[6]: %%timeit
sympy.solve(equations, unknowns)

13.5 ms ± 4.97 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Notice that the result is a [dictionary](#). We can get the individual answers by indexing (using `[]`)

```
[7]: solution[x]
```



```
[7]:
```

$$-\frac{4}{3}$$

We often need the numeric value rather than the exact value. We can convert to a floating point number using `.n()`

```
[8]: solution[x].n()
```

```
[8]:
```

$$-1.33333333333333$$

Special case: linear systems

For linear systems like the one above, we can solve very efficiently using matrix algebra. The system of equations can be rewritten in matrix form:

$$A\mathbf{x} = \mathbf{b}$$

```
[9]: equations
```

```
[9]:
```

$$[x + y - z, \quad 2x + y + z + 2, \quad x - y - z + 2]$$

```
[10]: A = sympy.Matrix([[1, 1, -1],
                        [2, 1, 1],
                        [1, -1, -1]])
      b = sympy.Matrix([0, -2, -2]).T
```

```
[11]: A.solve(b)
```

```
[11]:
```

$$\begin{bmatrix} -\frac{4}{3} \\ 1 \\ -\frac{1}{3} \end{bmatrix}$$

```
[12]: %%time
      A.solve(b)
```

```
CPU times: user 2.49 ms, sys: 882 µs, total: 3.37 ms
Wall time: 10 ms
```

```
[12]:
```

$$\begin{bmatrix} -\frac{4}{3} \\ 1 \\ -\frac{1}{3} \end{bmatrix}$$

We can repeat the solution using `numpy`. This is considerably faster than using `sympy` for large matrices.

```
[13]: import numpy
```



```
[14]: A = numpy.matrix([[1, 1, -1],
                        [2, 1, 1],
                        [1, -1, -1]])
      b = numpy.matrix([[0, -2, -2]]).T
```

```
[15]: numpy.linalg.solve(A, b)
```

```
[15]: matrix([[ -1.33333333],
             [ 1.          ],
             [-0.33333333]])
```

```
[16]: %%time
      numpy.linalg.solve(A, b)

CPU times: user 113 µs, sys: 24 µs, total: 137 µs
Wall time: 129 µs
```

```
[16]: matrix([[ -1.33333333],
             [ 1.          ],
             [-0.33333333]])
```

The numpy version is much faster, even for these small matrices. Let's try that again for a bigger matrix:

```
[17]: N = 30
      bigA = numpy.random.random((N, N))
```

```
[18]: bigb = numpy.random.random((N,))
```

```
[19]: %%timeit
      numpy.linalg.solve(bigA, bigb)

40.6 µs ± 1.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
[20]: bigsymbolicA = sympy.Matrix(bigA)
```

```
[21]: bigsymbolicb = sympy.Matrix(bigb)
```

```
[22]: %%timeit
      bigsymbolicA.solve(bigsymbolicb)

1.18 s ± 69 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Wow! That takes about a million times longer.

Nonlinear equations

In some cases, sympy can solve nonlinear equations exactly:

```
[23]: x, y = sympy.symbols('x, y')
```

```
[24]: sympy.solve([x + sympy.log(y), y**2 - 1], [x, y])
```

```
[24]: [(0, 1), (-iπ, -1)]
```


We can also specify the kinds of solutions we are interested in by increasing our specifications on the symbols. The answer above contained a complex answer. In engineering we often want only real solutions

```
[25]: x, y = sympy.symbols('x, y', real=True)

[26]: sympy.solve([x + sympy.log(y), y**2 - 1], [x, y])

[26]: [(0, 1)]
```

But sometimes nonlinear equations don't admit a closed-form solution:

```
[27]: unsolvable = x + sympy.cos(x) + sympy.log(x)

sympy.solve(unsolvable, x)

-----
NotImplementedError                                Traceback (most recent call last)
<ipython-input-27-8845e2a074b6> in <module>()
      1 unsolvable = x + sympy.cos(x) + sympy.log(x)
----> 2 sympy.solve(unsolvable, x)

~/anaconda3/lib/python3.6/site-packages/sympy/solvers/solvers.py in solve(f, *symbols,
-> **flags)
    1063     #####
-> #
    1064     if bare_f:
-> 1065         solution = _solve(f[0], *symbols, **flags)
    1066     else:
    1067         solution = _solve_system(f, symbols, **flags)

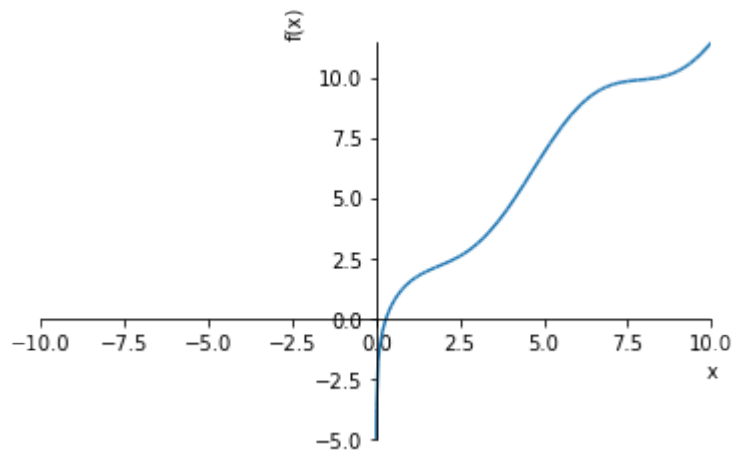
~/anaconda3/lib/python3.6/site-packages/sympy/solvers/solvers.py in _solve(f, *
-> *symbols, **flags)
    1632
    1633     if result is False:
-> 1634         raise NotImplementedError('\n'.join([msg, not_impl_msg % f]))
    1635
    1636     if flags.get('simplify', True):

NotImplementedError: multiple generators [x, cos(x), log(x)]
No algorithms are implemented to solve equation x + log(x) + cos(x)
```

Numeric root finding

In such cases we need to use approximate (numeric) solutions. When finding roots numerically it is a good idea to produce a plot if possible:

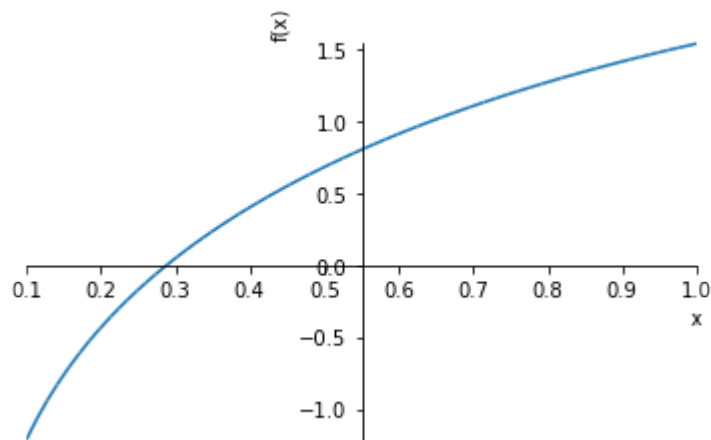
```
[28]: sympy.plot(unsolvable)
```

```
[28]: <sympy.plotting.plot.Plot at 0x11ee4c198>
```

We see the root is between 0 and 1 and there appears to be an asymptote at 0. Let's zoom in a bit

```
[29]: sympy.plot(unsolvable, (x, 0.1, 1))
```



```
[29]: <sympy.plotting.plot.Plot at 0x11f2dd908>
```

Sympy.nsolve will attempt to find a root starting near a starting point. 0.3 looks like a good first guess.

```
[30]: sympy.nsolve(unsolvable, x, 0.3)
```

```
[30]: 0.287518275445415
```

If we're going to be using numeric methods anyway, we can also use the routines in `scipy.optimize` to solve equations:

```
[31]: import scipy.optimize
```

The function `sympy.lambdify` can be used to build a function which evaluates `sympy` expressions numerically:

```
[32]: plus_two = lambda x: x+2
```



```
[33]: plus_two(2)
```

```
[33]: 4
```

```
[34]: def plus_two(x):
      return x + 2
```

```
[35]: unsolvable_numeric = sympy.lambdify(x, unsolvable)
```

```
[36]: unsolvable_numeric(0.3)
```

```
[36]: 0.051363684799669906
```

This is the kind of thing we can pass to `scipy.optimize.fsolve`

```
[37]: scipy.optimize.fsolve(unsolvable_numeric, 0.1)
```

```
[37]: array([0.28751828])
```

`fsolve` works for multiple equations as well, just return a list:

```
[38]: def multiple_equations(unknowns):
      x, y = unknowns
      return [x + y - 1,
              x - y]
```

```
[39]: multiple_equations([1, 2])
```

```
[39]: [2, -1]
```

```
[40]: first_guess = [1, 1]
      scipy.optimize.fsolve(multiple_equations, first_guess)
```

```
[40]: array([0.5, 0.5])
```

Downsides of numerical solution

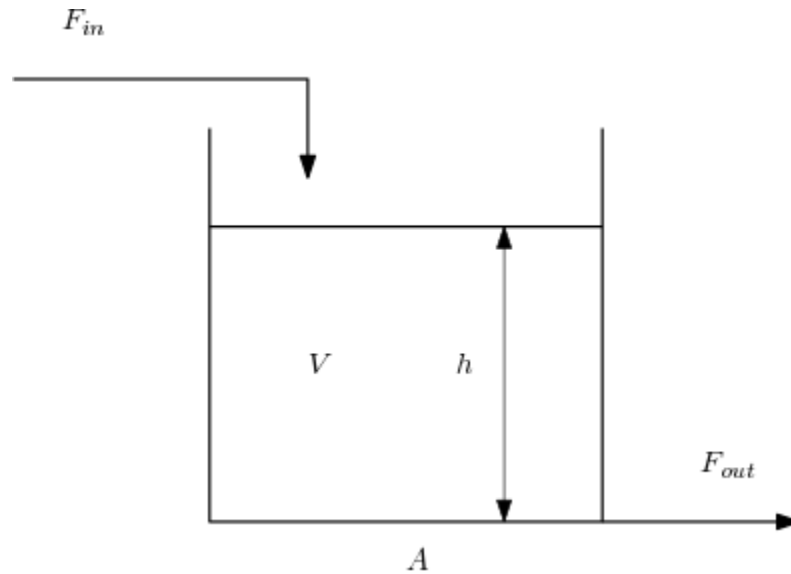
Remember the downsides of numerical solution:

1. Approximate rather than exact
2. Requires an initial guess
3. Slower to solve the equation every time rather than solving it once and then substituting values.
4. Typically only finds one solution, even if there are many.

Differential equations

Now for differential equations.

We'll solve the “classic” tank problem:



$$F_{out} = kh \quad (2.7)$$

$$\frac{dh}{dt} = \frac{1}{A} (F_{in} - F_{out}) \quad (2.8)$$

$$(2.9)$$

Analytic solution

Sympy can solve some differential equations analytically:

```
[41]: h = sympy.Function('h') # This is how to specify an unknown function in sympy
      t = sympy.Symbol('t', positive=True)
```

```
[42]: Fin = 2
      K = 1
      A = 1
```

```
[43]: Fout = K*h(t)
      Fout
```

```
[43]: h(t)
```

We use `.diff()` to take the derivative of a function.

```
[44]: de = h(t).diff(t) - 1/A*(Fin - Fout)
      de
```


[44]:

$$1.0h(t) + \frac{d}{dt}h(t) - 2.0$$

Here we calculate the general solution. Notice this equation just satisfies the original differential equation when we plug it in, we don't have specific values at points in time until we specify boundary conditions.

[45]: `solution = sympy.dsolve(de)`
`solution`

[45]:

$$h(t) = C_1 e^{-1.0t} + 2.0$$

We need a name for the constant of integration which Sympy created. Expressions are arranged as trees with the arguments as elements. We can navigate this tree to get the C1 element:

[46]: `C1 = solution.rhs.args[1].args[0]`

We can find the value of the constant by using an initial value:

[47]: `h0 = 1`

[48]: `constants = sympy.solve(solution.rhs.subs({t: 0}) - h0, C1)`
`constants`

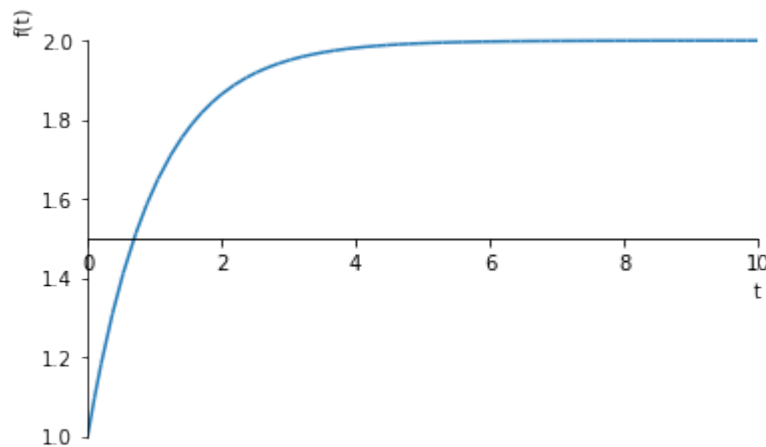
[48]:

$$[-1.0]$$

Let's see what that looks like

[49]: `import matplotlib.pyplot as plt`
`%matplotlib inline`

[50]: `sympy.plot(solution.rhs.subs({C1: constants[0]}), (t, 0, 10))`



[50]: `<sympy.plotting.plot.Plot at 0x10209a5d68>`

Numeric solution

When the boundary conditions of differential equations are specified at $t = 0$, this is known as an **Initial Value Problem** or IVP. We can solve such problems numerically using `scipy.integrate.solve_ivp`.

```
[51]: import scipy.integrate
```

```
[52]: Fin = 2
```

```
[53]: def dhdt(t, h):
    """Function returning derivative of h - note it takes t and h as arguments"""
    Fout = K*h
    return 1/A*(Fin - Fout)
```

`solve_ivp` will automatically determine the time steps to use, integrating between the two points in `tspan`:

```
[54]: tspan = (0, 10)
```

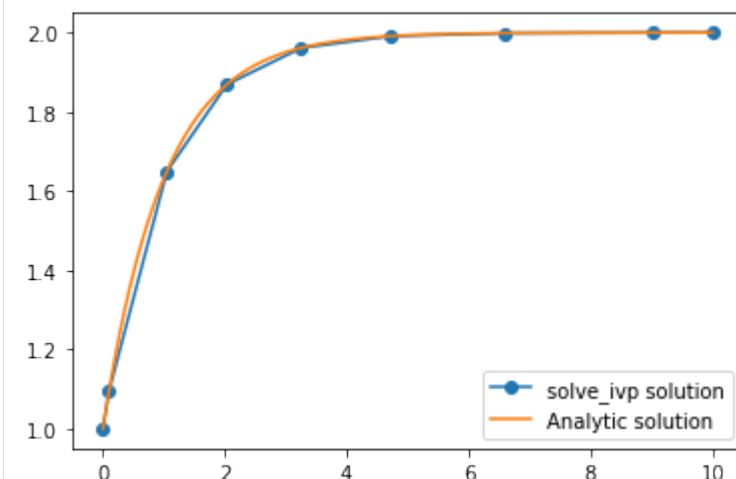
```
[55]: sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0])
```

We'll need a smooth set of time points to evaluate the analytic solution

```
[56]: tsmooth = numpy.linspace(0, 10, 1000)
    hanalytic = 2 - numpy.exp(-tsmooth)
```

```
[57]: plt.plot(sol.t, sol.y.T, 'o-', label='solve_ivp solution')
    plt.plot(tsmooth, hanalytic, label='Analytic solution')
    plt.legend()
```

```
[57]: <matplotlib.legend.Legend at 0x1520b9e6a0>
```



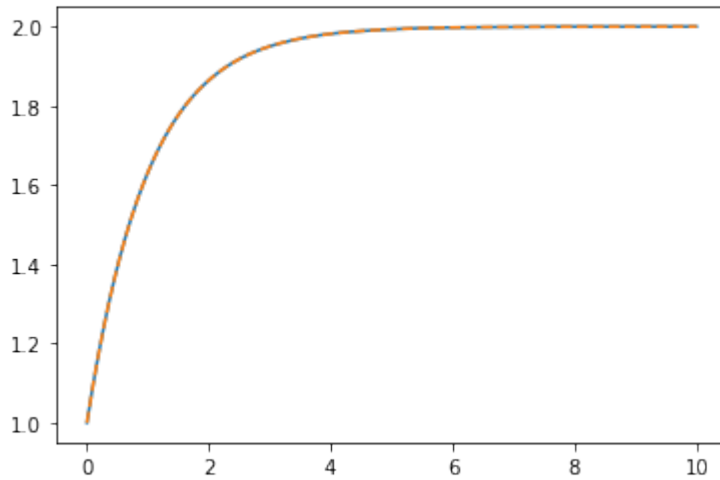
Notice that `solve_ivp` is taking really large steps but is still getting a really accurate solution. Of course, because we are taking such big steps to solve the differential equation, we now have a problem of interpolating between those points. The linear interpolation is clearly not very good, so `solve_ivp` supplies an extra argument which allows us to specify points we want the solution at. Note that this does not change the step size. The same steps are used internally and are then interpolated using a smooth function which is known to approximate the solution to the differential equation well.

```
[58]: sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], t_eval=tsmooth)
```



```
[59]: plt.plot(tsmooth, sol.y.T)
plt.plot(tsmooth, hanalytic, '--')

[59]: [<matplotlib.lines.Line2D at 0x1520beb908>]
```



We can see that this interpolation is very close to the correct solution.

There is a problem with taking big steps if inputs change discontinuously. This example illustrates the problem:

```
[60]: import scipy.integrate

[61]: def Fin(t):
    """ A step which starts at t=2 """
    if t < 2:
        return 1
    else:
        return 2

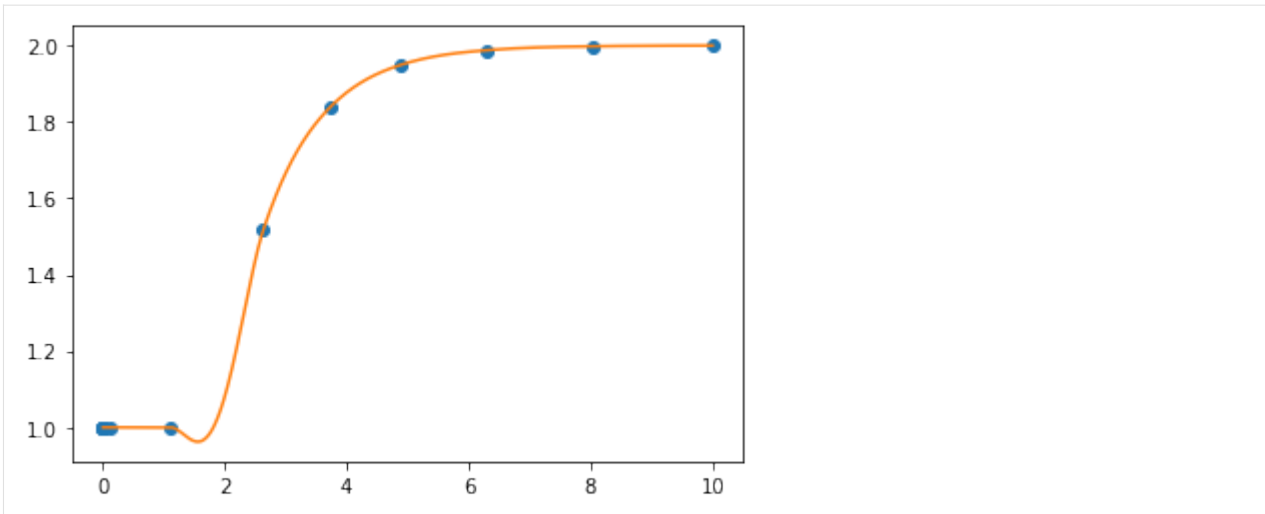
[62]: def dhdt(t, h):
    Fout = K*h
    return 1/A*(Fin(t) - Fout)

[63]: tspan = (0, 10)

[64]: sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0])
smoothsol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], t_eval=tsmooth)

[65]: plt.plot(sol.t, sol.y.T, 'o')
plt.plot(smoothsol.t, smoothsol.y.T)

[65]: [<matplotlib.lines.Line2D at 0x1520cab358>]
```

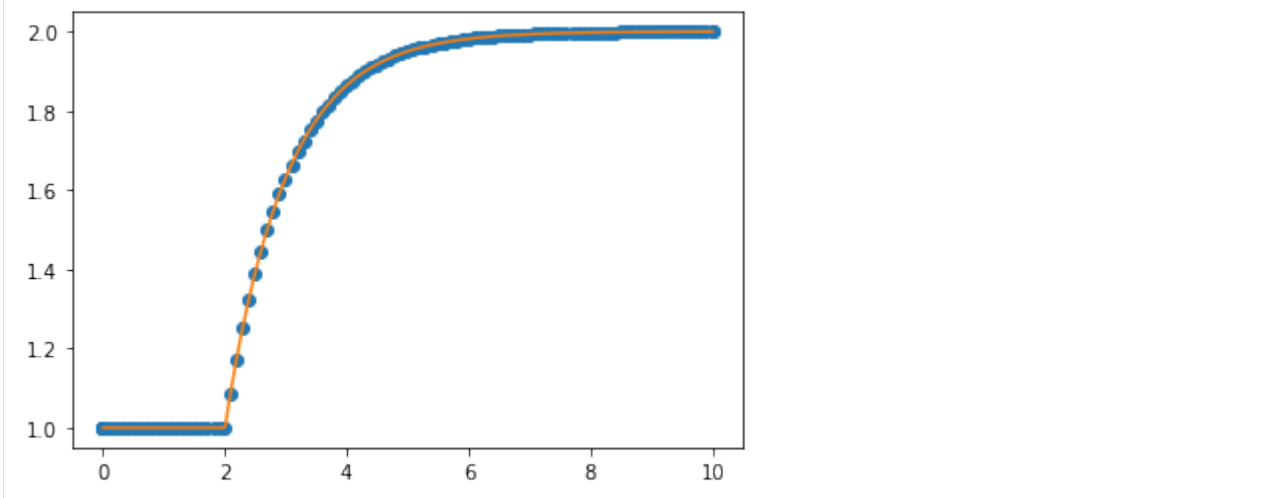



That downward bump in the level is a numerical anomaly due to the places where the samples are taking during integration. To make sure that we don't miss the moment when the step occurs, we can limit the step size that `solve_ivp` uses.

```
[66]: sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], max_step=0.1)
smoothsol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], t_eval=tsmooth, max_step=0.1)
```

```
[67]: plt.plot(sol.t, sol.y.T, 'o')
plt.plot(smoothsol.t, smoothsol.y.T)
```

```
[67]: [<matplotlib.lines.Line2D at 0x1520d8ee48>]
```



This works, but we pay for this with computer time:

```
[68]: %%timeit
sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0])

3.25 ms ± 207 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[69]: %%timeit
sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], max_step=0.1)

19.5 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```


A note about odeint

The default solver for ODEs in scipy used to be `odeint`, but this is now officially deprecated. You may still encounter it in older codes, so take note of the differences:

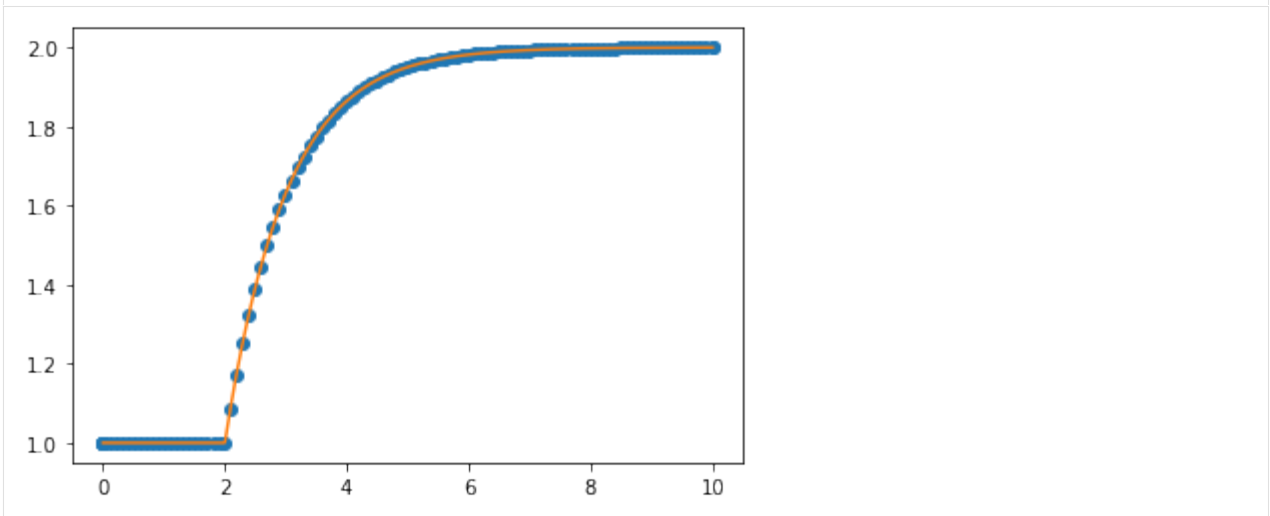
```
[70]: def odeintdhdh(h, t):
      """Odeint expects a function with the arguments reversed from solve_ivp"""
      return dhdt(t, h)
```

The order in which the initial values and the input times are specified is different. Also, unlike `solve_ivp`, you always give `odeint` the times you want the results at.

```
[71]: odeint_h = scipy.integrate.odeint(odeintdhdh, h0, tsmooth)
```

```
[72]: plt.plot(sol.t, sol.y.T, 'o')
      plt.plot(tsmooth, odeint_h)
```

```
[72]: [<matplotlib.lines.Line2D at 0x11eca3518>]
```



2.2.2 The problem with simple math on computers

Have you ever considered how computers store numbers? Can you explain why this happens?

```
[1]: a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
      a
```

```
[1]: 0.9999999999999999
```

```
[2]: a == 1.
```

```
[2]: False
```

```
[3]: b = 0.125 + 0.125 + 0.125 + 0.125 + 0.125 + 0.125 + 0.125 + 0.125
      b
```

```
[3]: 1.0
```



```
[4]: b == 1.
[4]: True
```

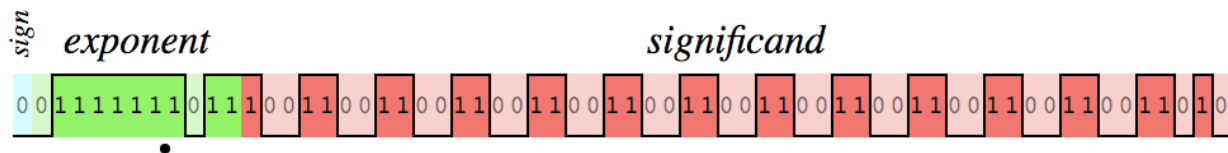
Computers use base 2 instead of base 10

You've heard that computers are all about ones and zeros, right? What does this mean?

When I write a “normal” number like 123, what I mean is $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$. This idea is called base 10 or decimal representation. Computers use binary or base 2 representation. This means you would write $101_2 = 5_{10}$, with the subscript representing the base. The math would work out as $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, just like in the 123 example.

This representation is exact for integers, but we run into problems when we use fractions. For instance, we all know that $1/3$ doesn't have a finite representation in decimals, since $1/3 = 0.\overline{33} = 3 \times 10^{-1} + 3 \times 10^{-2} + \dots$ forever. Notice that in base 3, $1/3$ works out fine as 0.1_3 since $1/3 = 1 \times 3^{-1}$ exactly. So here's the problem with writing 0.1 in binary:

This visualisation shows how IEEE floats are represented and indicates the repeating structure of the representation of 0.1.



We can see that the binary representation is not finite, so the computer treats $1/10$ more like a number like $1/7$ (which we all know has an infinite decimal representation).

There is a great deal more information on this issue at these pages:

- [The Floating-Point Guide](#) - this is an easy-to-read page with lots of examples
- [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#) - a more in-depth analysis of floating-point with lots of math

Solutions

Built-in to Python

The solution that Python supplies in the standard library is the decimal module:

```
[5]: import decimal

[6]: a = decimal.Decimal('0.1')

[7]: a = 1/decimal.Decimal(10)

[8]: sum(a for i in range(10))
[8]: Decimal('1.0')

[9]: sum(0.1 for i in range(10))
[9]: 0.9999999999999999
```


Sympy

Sympy also has a solution in the form of a Rational object

```
[10]: import sympy
sympy.init_printing()
```

```
[11]: b = sympy.Rational('0.1')
```

```
[12]: b
```

```
[12]: 
$$\frac{1}{10}$$

```

We can also use `sympy.nsimplify`.

```
[13]: b = 1
c = 10
```

```
[14]: a = b/c
```

```
[15]: type(a)
```

```
[15]: float
```

```
[16]: sympy.nsimplify(a)
```

```
[16]: 
$$\frac{1}{10}$$

```

Why isn't math always done in base 10?

The extra precision comes at a cost.

```
[17]: %%timeit
a = 0.1
s = 0
for i in range(100000):
    s += a

5 ms ± 321 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[18]: %%timeit
a = decimal.Decimal('0.1')
s = decimal.Decimal(0)
for i in range(100000):
    s += a

10.8 ms ± 246 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[19]: %%timeit
a = sympy.Rational(1, 10)
```

(continues on next page)

(continued from previous page)

```
s = 0
for i in range(100000):
    s += a
```

2.03 s \pm 166 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

Using sympy rationals is about a thousand times slower than using built-in Python floats.

Forcing rounding of exact representations

If an equation results in an “Exact” answer which isn’t “useful”, like $\sqrt{3}x$, we can approximate that using `sympy.N`

```
[20]: x = sympy.Symbol('x')
```

```
[21]: expr = sympy.sqrt(3)*x
      expr
```

```
[21]: 
$$\sqrt{3}x$$

```

```
[22]: sympy.simplify(expr**2)
```

```
[22]: 
$$3x^2$$

```

```
[23]: sympy.simplify(sympy.N(expr, 3)**2)
```

```
[23]: 
$$3.0x^2$$

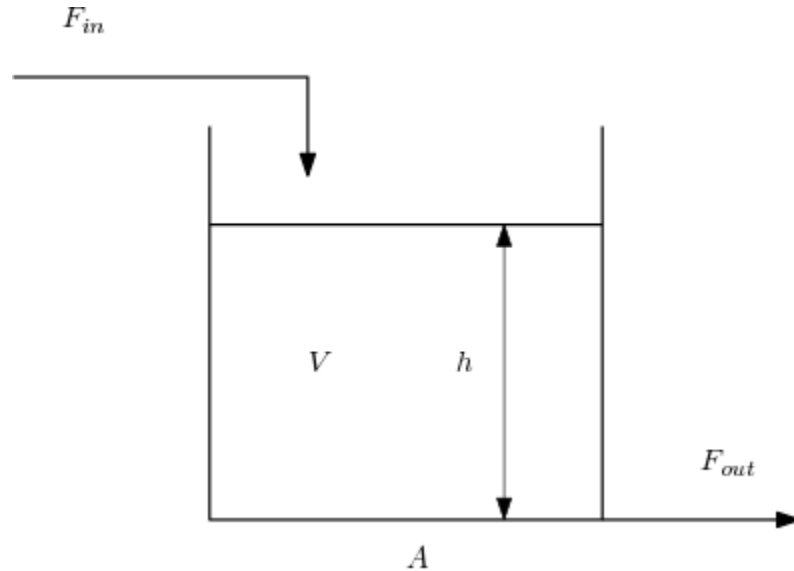
```

```
[ ]:
```

```
[1]: import numpy
      import matplotlib.pyplot as plt
      %matplotlib inline
```


2.2.3 Read simulation input from a file

It is often useful to read simulation inputs from a file. Let's do this for our standard tank system.



$$F_{out} = kh \quad (2.10)$$

$$\frac{dh}{dt} = \frac{1}{A} (F_{in} - F_{out}) \quad (2.11)$$

$$(2.12)$$

First we define the parameters of the system

```
[2]: K = 1
      A = 1
```

Then, we'll read the values of F_{in} from an Excel file using `pandas.read_excel`.

```
[3]: import pandas
```

```
[4]: df = pandas.read_excel('../assets/tankdata.xlsx')
      df
```

```
[4]:   Time  Fin
0      0  1.0
1      5  2.0
2     10  2.0
3     15  1.5
4     20  1.0
5     25  2.0
6     30  2.0
```

We'll set this function up to interpolate on the above table for the value of F_{in} given a known time.

```
[5]: def Fin(t):
      return numpy.interp(t, df.Time, df.Fin)
```

We can test for one value at a time


```
[6]: Fin(1)
```

```
[6]: 1.2
```

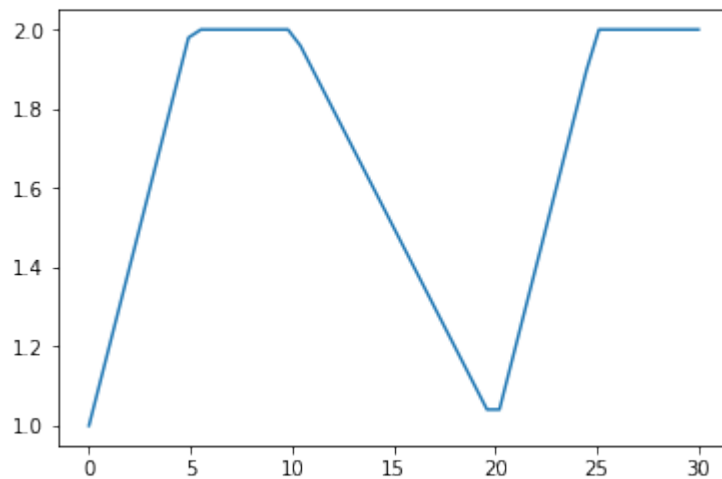
interp also accepts vector inputs:

```
[7]: tspan = (0, 30)
```

```
t = numpy.linspace(*tspan)
```

```
[8]: plt.plot(t, Fin(t))
```

```
[8]: [<matplotlib.lines.Line2D at 0x112b4c208>]
```



Now we're ready to define our differential equation function:

```
[9]: def dhdt(t, h):  
    Fout = K*h  
    return 1/A*(Fin(t) - Fout)
```

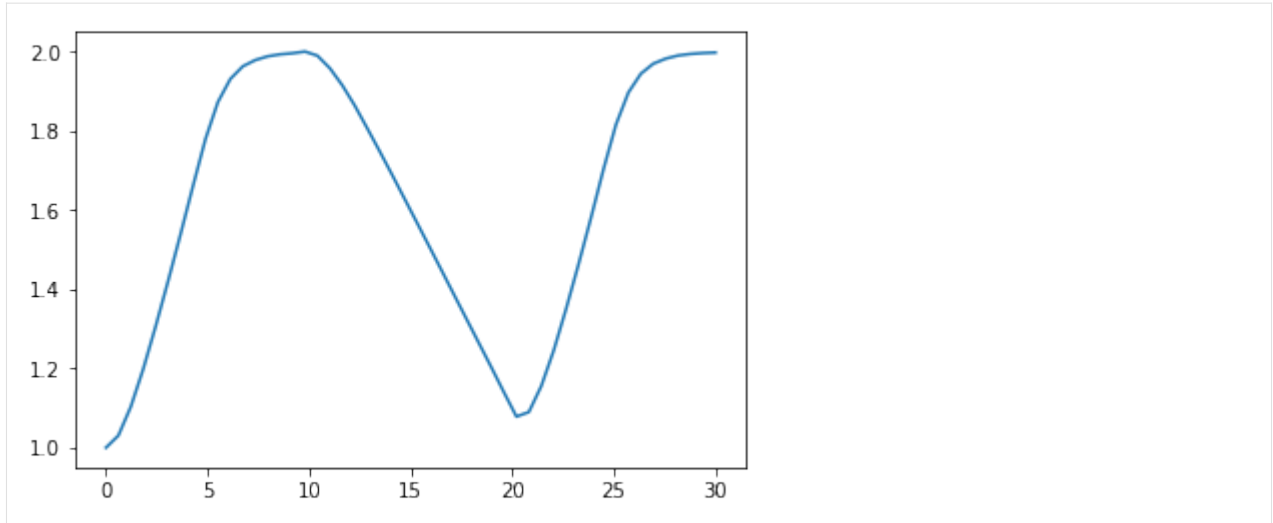
```
[10]: h0 = 1
```

```
[11]: import scipy.integrate
```

```
[12]: sol = scipy.integrate.solve_ivp(dhdt, tspan, [h0], t_eval=t)
```

```
[13]: plt.plot(sol.t, sol.y.T)
```

```
[13]: [<matplotlib.lines.Line2D at 0x1513ee6b38>]
```

[]:

2.2.4 Fed Batch Bioreactor

This model represents the fed batch bioreactor in section 2.4.9 of Seborg et al.

Nr	Symbol	Name	Units
1	μ_{\max}	Maximum specific growth rate	1/hr
2	K_s	Monod constant	g/L
3	$Y_{X/S}$	Cell yield coefficient	1
4	$Y_{P/S}$	Product yield coefficient	1
5	X	Cell mass concentration	g/L
6	S	Substrate mass concentration	g/L
7	V	Volume	L
8	P	Substrate mass concentration	g/L
9	F	Feed flow rate	L/hr
10	S_f	Substrate mass concentration	g/L
11	μ	Specific growth rate	1/hr
12	r_g	Cell growth rate	g / (L hr)
13	r_p	Product growth rate	g / (L hr)

Model equations:

Nr	Equation	Inputs	Outputs	Parameters
1	$r_g = \mu X$	•	r, μ, X	•
2	$\mu = \mu_{max} \frac{S}{K_S + S}$	•	S	μ_{max}, K_s
3	$r_p = Y_{P/X} r_G$	•	r_p	$Y_{P/X}$
4	$\frac{d(XV)}{dt} = Vr_g$	•	V	•
5	$\frac{d(PV)}{dt} = Vr_p$	•	P	•
6	$\frac{d(SV)}{dt} = FS_f - Vr$	F, S_f	•	$Y_{X/S}$
7	$\frac{dV}{dt} = F$	•	•	•
Number	7	2	7	4

Parameters

```
[1]: mu_max = 0.2 # Maximum growth rate
     K_s = 1.0 # Monod constant
     Y_xs = 0.5 # Cell yield coefficient
     Y_px = 0.2 # Product yield coefficient
```

States

```
[2]: X = 0.05 # Concentration of the cells
     S = 10 # Concentration of the substrate
     P = 0 # Concentration of the product
     V = 1 # Reactor volume

     x0 = [X, S, P, V] # State vector
```


Inputs

```
[3]: F = 0.05 # Feed rate
     S_f = 10 # Concentration of substrate in feed
```

```
[4]: def dxdt(t, x):
     [X, S, P, V] = x

     mu = mu_max * S / (K_s + S)
     rg = mu * X
     rp = Y_px * rg
     dVdt = F
     dXdtdt = 1/V*(V * rg - dVdt*X)
     dPdtdt = 1/V*(V * rp - dVdt*P)
     dSdtdt = 1/V*(F * S_f - 1 / Y_xs * V * rg - dVdt*S)

     return [dXdtdt, dSdtdt, dPdtdt, dVdtdt]
```

```
[5]: import scipy.integrate
     import numpy
```

```
[6]: import matplotlib.pyplot as plt
     %matplotlib inline
```

```
[7]: tspan = [0, 30]
```

```
[8]: tsmooth = numpy.linspace(0, 30)
```

```
[9]: Fs = [0.05, 0.02]
```

```
[10]: results = []
```

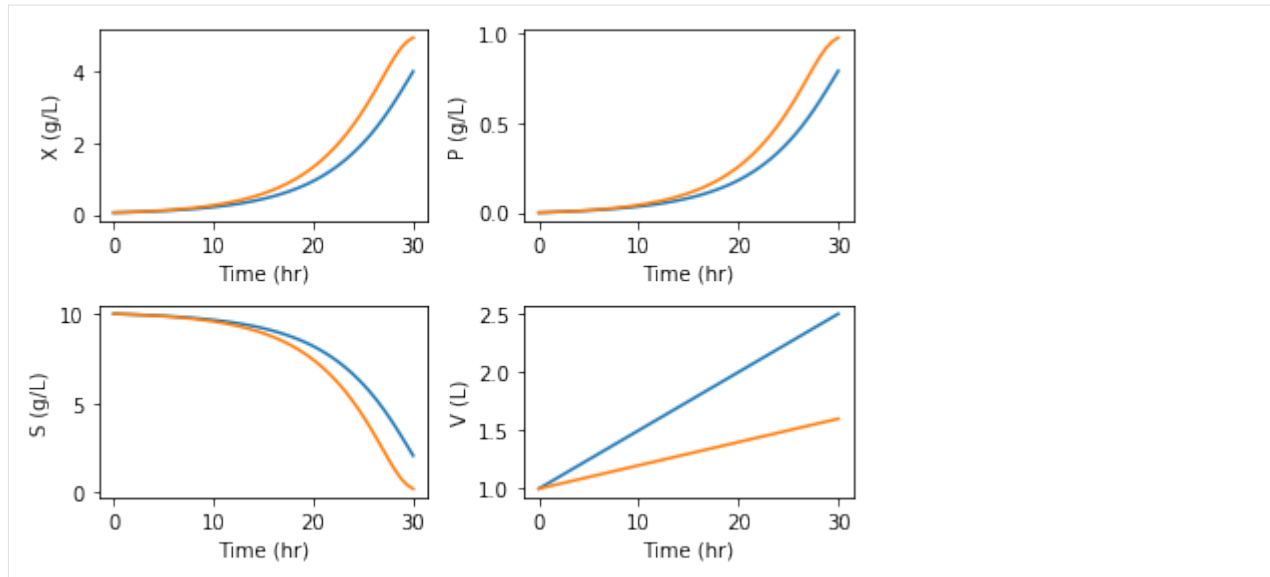
```
[11]: for F in Fs:
     out = scipy.integrate.solve_ivp(dxdt, tspan, x0, t_eval=tsmooth)
     results.append(out)
```

```
[12]: names = ['X', 'S', 'P', 'V']
     units = {'X': 'g/L', 'S': 'g/L', 'P': 'g/L', 'V': 'L'}
```

```
[13]: ax = {}

     fig, [[ax['X'], ax['P']], [ax['S'], ax['V']]] = plt.subplots(2, 2)

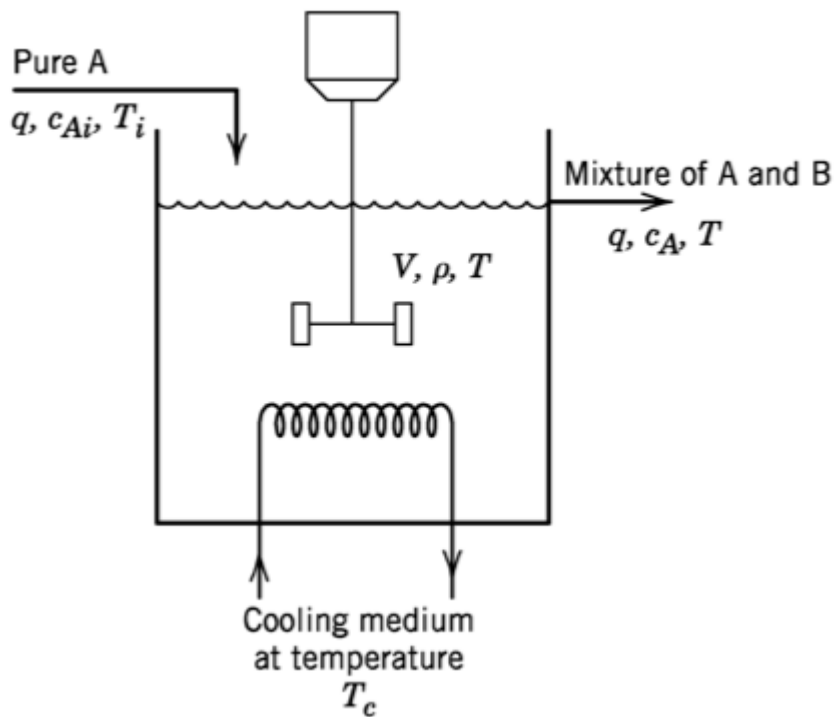
     for F, out in zip(Fs, results):
         var = {name: y for name, y in zip(names, out.y)}
         for name in names:
             ax[name].plot(out.t, var[name])
             ax[name].set_ylabel(f'{name} ({units[name]})')
             ax[name].set_xlabel('Time (hr)')
     plt.tight_layout()
```

[]:

2.2.5 CSTR system

This notebook explores solutions to the system discussed in Seborg, Edgar, Melichamp & Doyle “Process Dynamics and Control” (3rd Ed).



Model

$$k = k_0 \exp\left(\frac{-E}{RT}\right) \quad (2.13)$$

$$w = q\rho \quad (2.14)$$

$$V \frac{dc_A}{dt} = q(c_{Ai} - c_A) - Vkc_A \quad (2.15)$$

$$V\rho C \frac{dT}{dt} = wC(T_i - T) + (-\Delta H_R)Vkc_A + UA(T_c - T) \quad (2.16)$$

EB:

$$\frac{dE}{dt} = E_{in} - E_{out}$$

MB:

$$\frac{dm}{dt} = \dot{m}_{in} - \dot{m}_{out}$$

CB:

$$\frac{dN_A}{dt} = N_{A,in} - N_{A,out} + N_{A,gen} - N_{A,consumed}$$

```
[1]: import numpy
import scipy
import scipy.optimize
import scipy.integrate
```

Parameter values from Table 2.3

```
[2]: q = 100 # L/min
cA_i = 1 # mol/L
T_i = 350 # K
V = 100 # L
rho = 1000 # g/L
C = 0.239 # J/(g K)
Hr = -5e4 # J/(g K)
E_over_R = 8750 # K
k0 = 7.2e10 # 1/min
UA = 5e4 # J/(min K)
```

This is the initial value of the input T_c

```
[3]: Tc = Tc0 = 300 # K
```

These are the initial values of the states given in the question. Notice that these are not 100 % accurate. When we simulate using these values, we don't get a perfect straight line (derivatives equal to zero) as we should when using the steady state values.

```
[4]: cA0 = 0.5 # mol/L
T0 = 350 # K
```

We define the function to calculate the derivatives here.


```
[5]: def intsys(t, x):
    cA, T = x
    k = k0*numpy.exp(-E_over_R/T)
    w = q*rho
    dcAdt = q*(cA_i - cA)/V - k*cA
    dTdt = 1/(V*rho*C)*(w*C*(T_i - T) - Hr*V*k*cA + UA*(Tc - T))
    return dcAdt, dTdt
```

```
[6]: x0 = [cA0, T0]
```

Let's see what the derivatives look like at this "steady state"

```
[7]: intsys(0, x0)
```

```
[7]: (3.40208612952253e-05, -0.007117334999003795)
```

That doesn't seem very close to zero...

Now, let's simulate

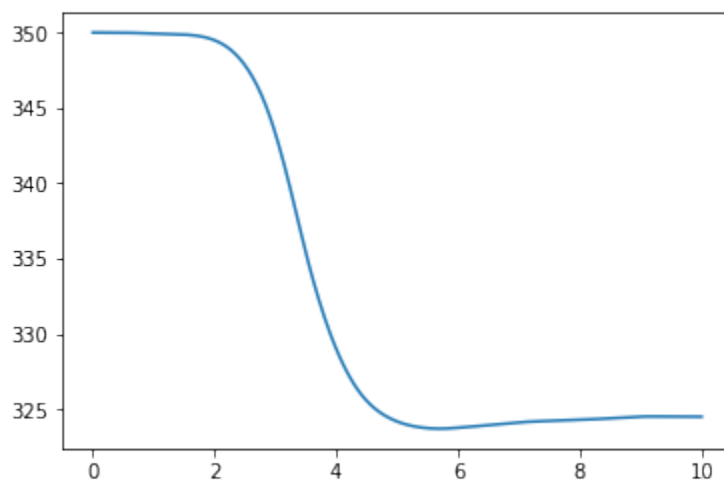
```
[8]: import matplotlib.pyplot as plt
    %matplotlib inline
```

```
[9]: tspan = (0, 10)
    t = numpy.linspace(*tspan, 1000)
```

```
[10]: def simulate():
    r = scipy.integrate.solve_ivp(intsys, tspan, x0, t_eval=t)
    return r.y
```

```
[11]: cA, T = simulate()
```

```
[12]: plt.plot(t, T)
    plt.show()
```



We see a significant deviation from the straight line we were expecting!

Solve for steady state

Now, let's solve for a better initial value by setting the derivatives equal to zero

```
[13]: def ss(x):
      """ This wrapper function simply calls intsys with a zero time"""
      return intsys(0, x)
```

We use fsolve to solve for a new steady state

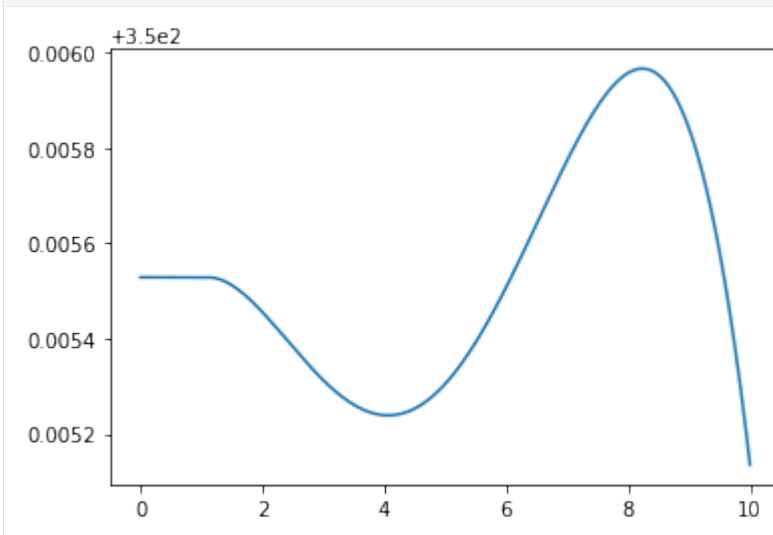
```
[14]: x0 = scipy.optimize.fsolve(ss, x0)
```

Let's check that:

```
[15]: ss(x0)
[15]: (3.7614356074300304e-13, -7.879222786077396e-11)
```

Much better, let's simulate:

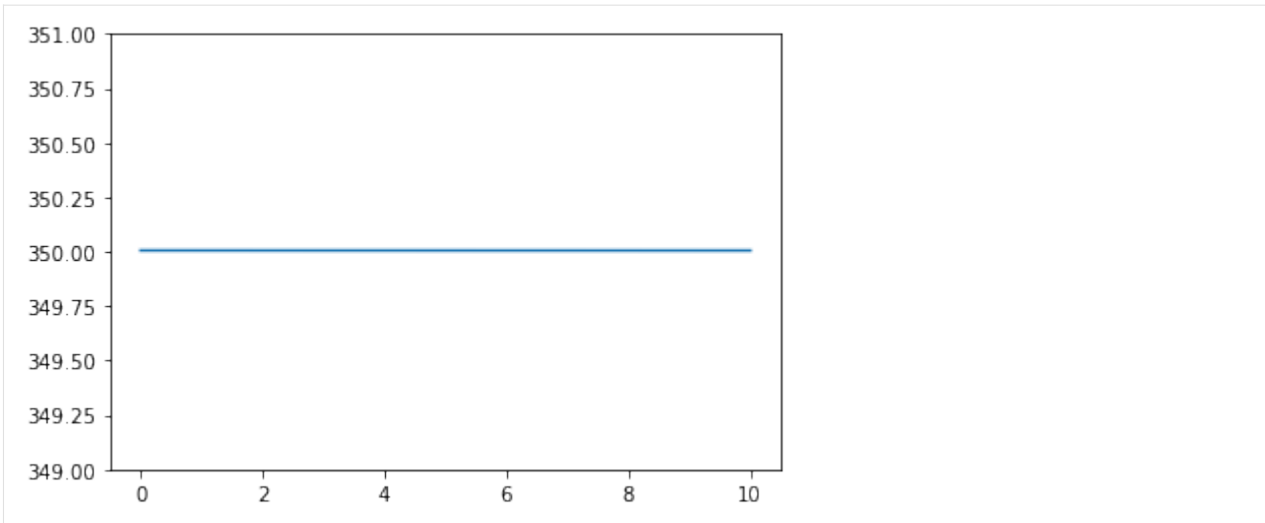
```
[16]: cA, T = simulate()
      plt.plot(t, T)
      plt.show()
```



What happened there? It seems as though these equations are quite hard to balance perfectly at the steady state, since starting quite close to zero derivatives still gave some deviation toward the end. Nonlinear differential equations often exhibit this high sensitivity to the initial value.

But look closely and you can see that the axis is strangely indicated. When we zoom out just a little the solution is more clear:

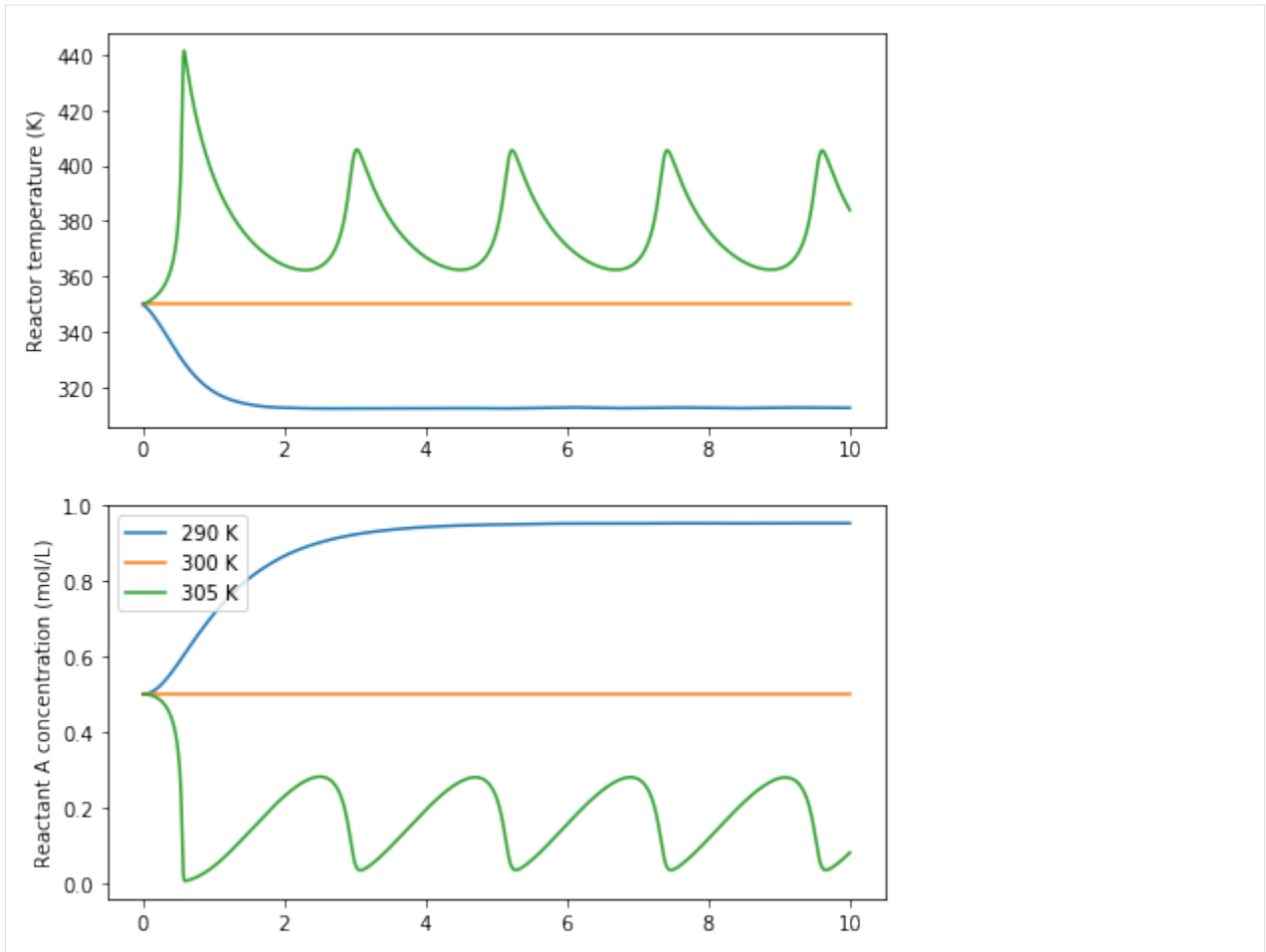
```
[17]: cA, T = simulate()
      plt.plot(t, T)
      plt.ylim(349, 351)
      plt.show()
```

I'd say that's good enough.

Now we are ready to reproduce the figure

```
[18]: fig, (axT, axcA) = plt.subplots(2, 1, figsize=(7, 8))
      for Tc in [290, 300, 305]:
          cA, T = simulate()
          axT.plot(t, T, label='{} K'.format(Tc))
          axT.set_ylabel('Reactor temperature (K)')
          axcA.plot(t, cA, label='{} K'.format(Tc))
          axcA.set_ylabel('Reactant A concentration (mol/L)')
      axcA.legend()
      plt.show()
```

Nonlinear behaviour

Nonlinear differential equations like this can exhibit very strange behaviour. We may expect that increasing the cooling water temperature will always increase the reactor temperature, measured after a certain amount of time, but the oscillatory behaviour we see in the graphs above give us a clue that everything may not be as simple as it appears.

```
[19]: sol = scipy.integrate.solve_ivp(intsys, tspan, x0)
```

```
[20]: Tends = []
Tcs = numpy.linspace(300, 310, 1000)
for Tc in Tcs:
    sol = scipy.integrate.solve_ivp(intsys, tspan, x0)
    T = sol.y[-1]
    Tends.append(T[-1])
```

```
/Users/alchemyst/anaconda3/lib/python3.6/site-packages/scipy/integrate/_ivp/rk.py:141:
↳ RuntimeWarning: invalid value encountered in true_divide
    error_norm = norm(error / scale)
/Users/alchemyst/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:6:
↳ RuntimeWarning: overflow encountered in double_scalars

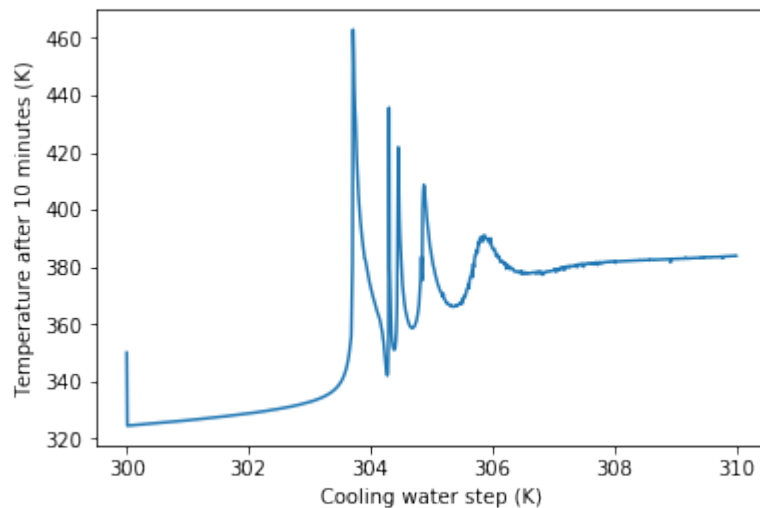
/Users/alchemyst/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:5:
↳
```

(continues on next page)

(continued from previous page)

```
↳RuntimeWarning: overflow encountered in double_scalars
"""
```

```
[21]: plt.plot(Tcs, Tends)
plt.ylabel('Temperature after 10 minutes (K)')
plt.xlabel('Cooling water step (K)')
plt.show()
```



We can see that there is often no easy explanation to system behaviour like “Making x bigger will make y bigger”. For nonlinear systems the answer to questions the direction of an effect is often very much “it depends”

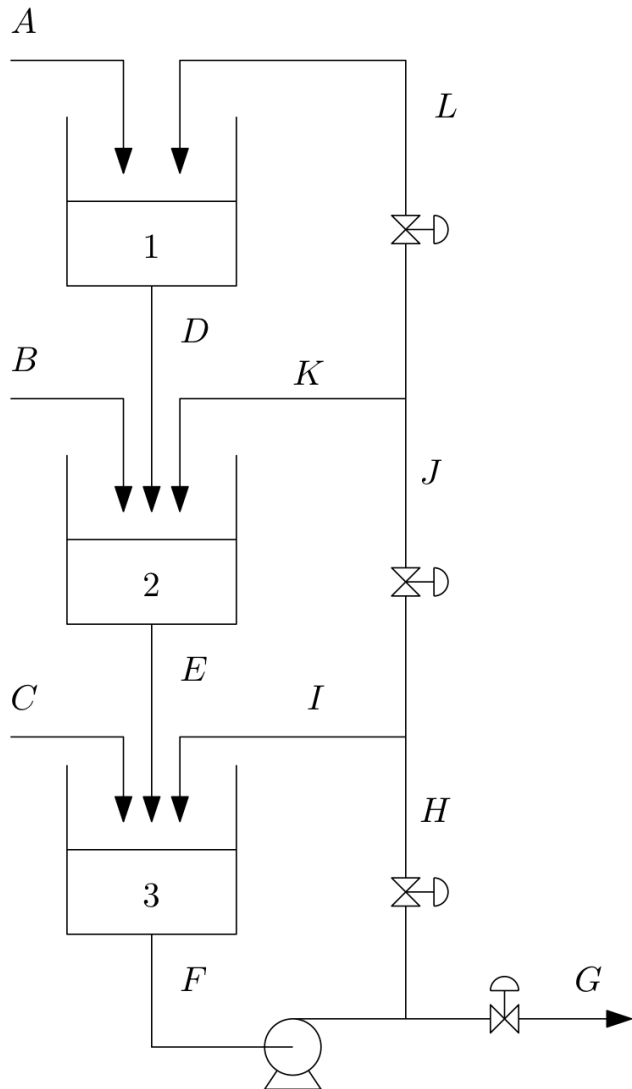
2.2.6 Mixing system

Problem statement: The figure below shows a set of well-mixed mixing tanks. All the streams contain a binary mixture of substance X and substance Y. Steams A, B and C are fed into the system from an upstream process.

Tanks 1 and 2 are drained by the force of gravity (assume flow is proportional to level), while the pump attached to the tank 3 output is sized such that the level in tank 3 does not affect the flowrate through the pump.

You may assume that the valves in lines G, H, J and L can manipulate those flows directly.

The density of substance X is $\rho_X = 1000 \text{ kg/m}^3$ and the density of substance Y is $\rho_Y = 800 \text{ kg/m}^3$.



2.2.7 Steady state calculation

Find the steady state flow rates and compositions of all the streams given that 3 * Stream A is 1m³/h of substance X * Streams B and C are both 1m³/h of substance Y. * H=G,H=2J,J=2L.

Flow rates

```
[1]: ρx = 1000 # kg/m3
      ρy = 800 # kg/m3
```

```
A = 1*ρx
B = 1*ρy
C = 1*ρy
```

```
[2]: G = A + B + C
```



```
[3]: H = G
      J = H/2
      L = J/2
```

```
[4]: F = G + H
```

```
[5]: D = A + L
```

```
[6]: K = J - L
      I = H - J
```

```
[7]: E = B + D + K
```

```
[8]: A, B, C, D, E, F, G, H, I, J, K, L
```

```
[8]: (1000,
      800,
      800,
      1650.0,
      3100.0,
      5200,
      2600,
      2600,
      1300.0,
      1300.0,
      650.0,
      650.0)
```

Compositions

```
[9]: xA = 1
      xB = 0
      xC = 0
```

```
[10]: xG = (xA*A + xB*B + xC*C) / G
```

```
[11]: x3 = xF = xH = xI = xJ = xK = xL = xG
```

```
[12]: x1 = xD = (xA*A + xL*L) / D
```

```
[13]: x2 = xE = (xB*B + xD*D + xK*K) / E
```


2.2.8 Design

Assuming all three tanks are of constant cross-sectional area of 3m^2 , find out what the proportionality constants should be for tank 1 and 2 so that the steady state levels will be 1 m.

```
[14]: A1 = A2 = A3 = 3
```

```
[15]: h1 = h2 = h3 = 1
```

```
[16]: k1 = D/h1
```

```
[17]: k2 = E/h2
```

```
[18]: k1, k2
```

```
[18]: (1650.0, 3100.0)
```

2.2.9 Dynamic simulation

Now that you have all the parameters in your system, simulate the response of the system to a sudden increase in flow rate of A from $1\text{ m}^3/\text{h}$ to $1.5\text{ m}^3/\text{h}$ at time 0. You should start your simulation at steady state.

Assume that the level in tank 3 is also 1 m at the initial conditions. Note that the steady state relationships between H, G, J and L will not hold over the whole simulation. Simply set them to their steady state values.

```
[19]: import scipy.integrate
```

Our states will be the total masses and mass of X in each tank. Let's find the initial values at steady state first:

Find volumes via tank geometry

```
[20]: V1 = A1*h1
      V2 = A2*h2
      V3 = A3*h3
```

Masses from volumes - assume ideal mixing

```
[21]: M1 = V1/(x1/ρx + (1 - x1)/ρy)
      M2 = V2/(x2/ρx + (1 - x2)/ρy)
      M3 = V3/(x3/ρx + (1 - x3)/ρy)
```

```
[22]: y0 = [M1, M2, M3, M1*x1, M2*x2, M3*x3]
```

```
[23]: t = 0
```

```
[24]: def dMdt(t, y):
      M1, M2, M3, M1x1, M2x2, M3x3 = y

      if t <= 0:
          A = 1*ρx
      else:
          A = 1.5*ρx

      xD = x1 = M1x1/M1
```

(continues on next page)

(continued from previous page)

```

xE = x2 = M2x2/M2
xF = x3 = M3x3/M3

V1 = M1*(x1/ρx + (1 - x1)/ρy)
V2 = M2*(x2/ρx + (1 - x2)/ρy)
V3 = M3*(x3/ρx + (1 - x3)/ρy)

h1 = V1/A1
h2 = V2/A2
h3 = V3/A3

xH = xI = xJ = xK = xL = xG = x3

D = k1*h1
E = k2*h2

dM1dt = A + L - D
dM2dt = B + D + K - E
dM3dt = C + E + I - F

dM1x1dt = xA*A + xL*L - xD*D
dM2x2dt = xB*B + xD*D + xK*K - xE*E
dM3x3dt = xC*C + xE*E + xI*I - xF*F

return dM1dt, dM2dt, dM3dt, dM1x1dt, dM2x2dt, dM3x3dt

```

We expect $t=0$ to give zero derivatives

```

[25]: dMdt(0, y0)
[25]: (0.0,
      4.547473508864641e-13,
      0.0,
      0.0,
      4.547473508864641e-13,
      -4.547473508864641e-13)

```

And for other times to give non-zero derivatives

```

[26]: dMdt(1, y0)
[26]: (500.0,
      4.547473508864641e-13,
      0.0,
      500.0,
      4.547473508864641e-13,
      -4.547473508864641e-13)

```

```

[27]: sol = scipy.integrate.solve_ivp(dMdt, (0, 10), y0)

```

```

[28]: sol
[28]: message: 'The solver successfully reached the end of the integration interval.'
      nfev: 68
      njev: 0
      nlu: 0
      sol: None

```

(continues on next page)

(continued from previous page)

```

status: 0
success: True
t: array([0.00000000e+00, 1.00000000e-04, 9.32930762e-04, 9.26223838e-03,
        9.25553145e-02, 9.25486076e-01, 1.86268269e+00, 3.13373601e+00,
        4.83441945e+00, 7.23515873e+00, 9.99326358e+00, 1.00000000e+01])
t_events: None
y: array([[2828.57142857, 2828.61687016, 2829.03321934, 2833.18623756,
        2873.68652849, 3191.18414035, 3411.4077485 , 3576.81492267,
        3679.20042145, 3732.50482985, 3751.73742159, 3751.76206113],
        [2657.14285714, 2657.14285827, 2657.14297442, 2657.1545728 ,
        2658.2647728 , 2730.90792703, 2850.15605228, 2978.77726282,
        3075.5278406 , 3130.61308859, 3150.4898384 , 3150.51308613],
        [2600.          , 2600.          , 2600.00000004, 2600.00003725,
        2600.0360844 , 2626.36069917, 2755.48727041, 3096.98554741,
        3748.1911919 , 4840.17117488, 6180.11425868, 6183.43458116],
        [2142.85714286, 2142.90258442, 2143.31893191, 2147.4717797 ,
        2187.95586832, 2505.19417937, 2729.14975288, 2908.4244449 ,
        3034.24936155, 3113.23792305, 3148.959073 , 3149.01344238],
        [1285.71428571, 1285.71428686, 1285.71440471, 1285.72617038,
        1286.84986914, 1359.78761856, 1480.91665298, 1618.72961343,
        1733.52304584, 1810.12997441, 1844.51567419, 1844.56643364],
        [1000.          , 1000.          , 1000.00000004, 1000.00004033,
        1000.03821936, 1022.64243623, 1116.4204006 , 1327.1016063 ,
        1689.48149947, 2263.4913374 , 2943.38602418, 2945.04374286]])

```

Plot the composition of stream G as well as the compositions and levels in all three tanks.

```
[29]: import matplotlib.pyplot as plt
      %matplotlib inline
```

```
[30]: M1, M2, M3, M1x1, M2x2, M3x3 = sol.y
```

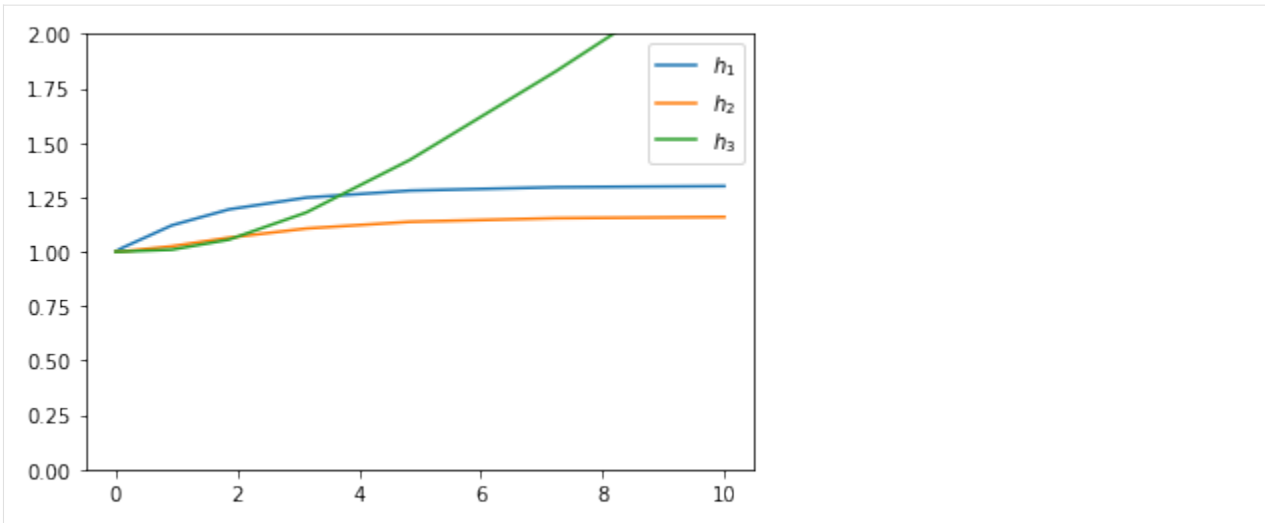
```
[31]: x1 = M1x1/M1
      x2 = M2x2/M2
      x3 = M3x3/M3

      V1 = M1*(x1/ρx + (1 - x1)/ρy)
      V2 = M2*(x2/ρx + (1 - x2)/ρy)
      V3 = M3*(x3/ρx + (1 - x3)/ρy)

      h1 = V1/A1
      h2 = V2/A2
      h3 = V3/A3
```

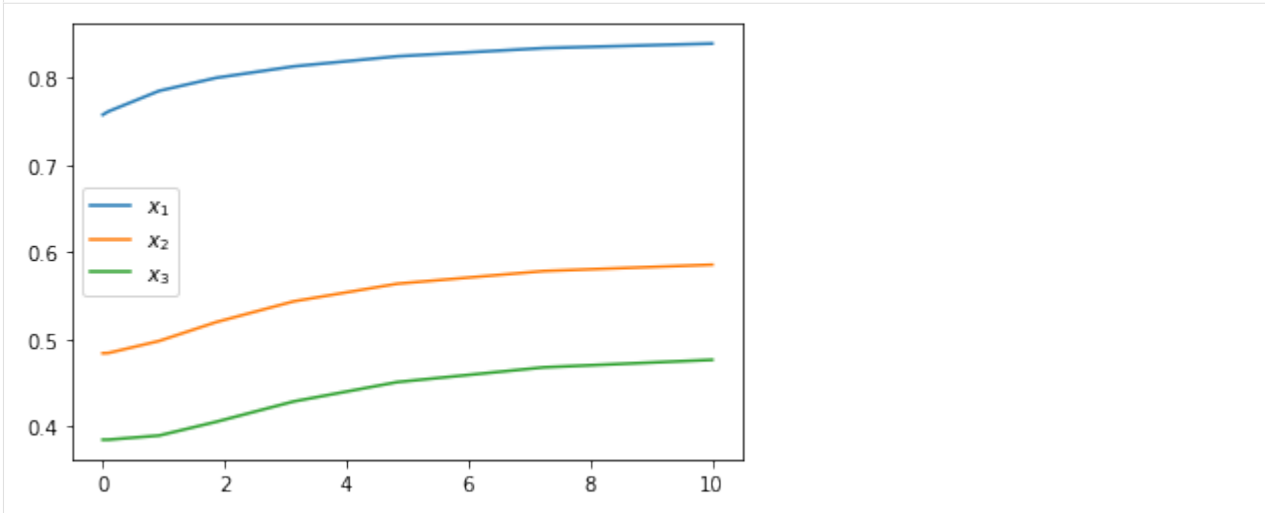
```
[32]: plt.plot(sol.t, h1,
               sol.t, h2,
               sol.t, h3)
      plt.ylim(0, 2)
      plt.legend(['$h_1$', '$h_2$', '$h_3$'])
```

```
[32]: <matplotlib.legend.Legend at 0x1519b93c50>
```

```
[33]: plt.plot(sol.t, x1,
              sol.t, x2,
              sol.t, x3)
plt.legend(['$x_1$', '$x_2$', '$x_3$'])
```

```
[33]: <matplotlib.legend.Legend at 0x1519bf0e10>
```



```
[ ]:
```


2.3 Linear systems

```
[1]: import sympy
sympy.init_printing()
```

2.3.1 Valve equation

Let's linearise the nasty nonlinear term in the equation percentage valve relationship in T4 Problem 4 (or T2 problem 4)

$$F = \underbrace{C_v \alpha^{x-1}}_{\text{nonlinear}}$$

First we introduce the requisite symbols. Notice that we specify constraints on these variables, this will make simplifications better later on.

```
[2]: C_v, alpha, x = sympy.symbols('C_v, alpha, x', positive=True)
```

```
[3]: term = C_v*alpha**(x - 1)
```

We also introduce a barred versions of the variable. Sympy automatically constructs these to typesetting nicely.

```
[4]: xbar = sympy.symbols('xbar', positive=True)
```

For single variable expressions, we can use `sympy.series` to linearise for us. Note that the help for `sympy.series` references the help for `sympy.Expr.series`, which has a lot more detail about the operation of this function

```
[5]: sympy.series?
```

```
[6]: sympy.Expr.series?
```

Calling `series` by itself will result in an error term (the one with an \mathcal{O}). This is useful to estimate the error of the approximation.

```
[7]: sympy.series(term, x, xbar, 2)
```

```
[7]:
```

$$\frac{C_v e^{\bar{x} \log(\alpha)}}{\alpha} + \frac{C_v (x - \bar{x}) e^{\bar{x} \log(\alpha)} \log(\alpha)}{\alpha} + O\left((x - \bar{x})^2; x \rightarrow \bar{x}\right)$$

But mostly we will be interested in the expression rather than the error, so we will remove that term with the `removeO` method:

```
[8]: lineq = sympy.series(term, x, xbar, 2).removeO()
lineq
```

```
[8]:
```

$$\frac{C_v (x - \bar{x}) e^{\bar{x} \log(\alpha)} \log(\alpha)}{\alpha} + \frac{C_v e^{\bar{x} \log(\alpha)}}{\alpha}$$

Rewriting in terms of deviation variables

While we are here, we can also rewrite in terms of deviation variables:

```
[9]: xprime = sympy.symbols("x'", positive=True)
```

```
[10]: lineq_deviation = lineq.subs({x: xprime + xbar})
lineq_deviation
```

```
[10]:
```

$$\frac{C_v x' e^{\bar{x} \log(\alpha)} \log(\alpha)}{\alpha} + \frac{C_v e^{\bar{x} \log(\alpha)}}{\alpha}$$

2.3.2 A note about simplification

You will note that we specified `positive=True` for all our symbols when we created them. This is because the default assumptions about variables in SymPy are that they are complex. And for complex numbers, `log` is not a 1-to-1 function. See if you understand the following:

```
[11]: xbar, alpha = sympy.symbols('xbar, alpha')
sympy.exp(xbar*sympy.log(alpha)).simplify()
```

```
[11]:
```

$$e^{\bar{x} \log(\alpha)}$$

```
[12]: xbar, alpha = sympy.symbols('xbar, alpha', positive=True)
sympy.exp(xbar*sympy.log(alpha)).simplify()
```

```
[12]:
```

$$\alpha^{\bar{x}}$$

Multiple variables

Unfortunately, SymPy doesn't have a built-in function for multivariate Taylor series, and consecutive application of the `series` function doesn't do exactly what we want.

```
[13]: variables = x, y, z = sympy.symbols('x, y, z')
bars = xbar, ybar, zbar = sympy.symbols('xbar, ybar, zbar')
```

```
[14]: term = x*y*z
```

Note that the other variables are assumed to be constant here, so we don't recover the answer we are looking for.

```
[15]: term.series(x, xbar, 2).removeO().series(y, ybar, 2).removeO()
```

```
[15]:
```

$$x\bar{y}z + xz(y - \bar{y})$$

The function `tbcontrol.symbolic.linearise` calculates a multivariable linearisation using the textbook formula. Note that it does not handle expressions which include derivatives or equalities, so don't try to pass a full equation, just use it for the nonlinear terms.


```
[16]: import tbcontrol.symbolic
```

```
[17]: bars, linearexpression = tbcontrol.symbolic.linearise(term, variables)
linearexpression
```

```
[17]: 
$$\bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}(z - \bar{z}) + \bar{x}\bar{z}(y - \bar{y}) + \bar{y}\bar{z}(x - \bar{x})$$

```

2.3.3 Laplace transforms in SymPy

The Laplace transform is

$$\mathcal{L}\{f(t)\} = \int_0^{\infty} f(t)e^{-st} ds$$

```
[1]: import sympy
sympy.init_printing()
```

```
[2]: import matplotlib.pyplot as plt
%matplotlib inline
```

Let's define some symbols to work with.

```
[3]: t, s = sympy.symbols('t, s')
a = sympy.symbols('a', real=True, positive=True)
```

Direct evaluation

We start with a simple function

```
[4]: f = sympy.exp(-a*t)
f
```

```
[4]:  $e^{-at}$ 
```

We can evaluate the integral directly using integrate:

```
[5]: sympy.integrate(f*sympy.exp(-s*t), (t, 0, sympy.oo))
```

```
[5]: 
$$\begin{cases} \frac{1}{s(\frac{a}{s}+1)} & \text{for } |\arg(s)| \leq \frac{\pi}{2} \\ \int_0^{\infty} e^{-at} e^{-st} dt & \text{otherwise} \end{cases}$$

```


Library function

This works, but it is a bit cumbersome to have all the extra stuff in there.

Sympy provides a function called `laplace_transform` which does this more efficiently. By default it will return conditions of convergence as well (recall this is an improper integral, with an infinite bound, so it will not always converge).

```
[6]: sympy.laplace_transform(f, t, s)
```

```
[6]:  $\left(\frac{1}{a+s}, -a, \text{True}\right)$ 
```

If we want just the function, we can specify `noconds=True`.

```
[7]: F = sympy.laplace_transform(f, t, s, noconds=True)
F
```

```
[7]:  $\frac{1}{a+s}$ 
```

We will find it useful to define a quicker version of this:

```
[8]: def L(f):
    return sympy.laplace_transform(f, t, s, noconds=True)
```

Inverses are simple as well,

```
[9]: def invL(F):
    return sympy.inverse_laplace_transform(F, s, t)
```

```
[10]: invL(F)
```

```
[10]:  $e^{-at}\theta(t)$ 
```

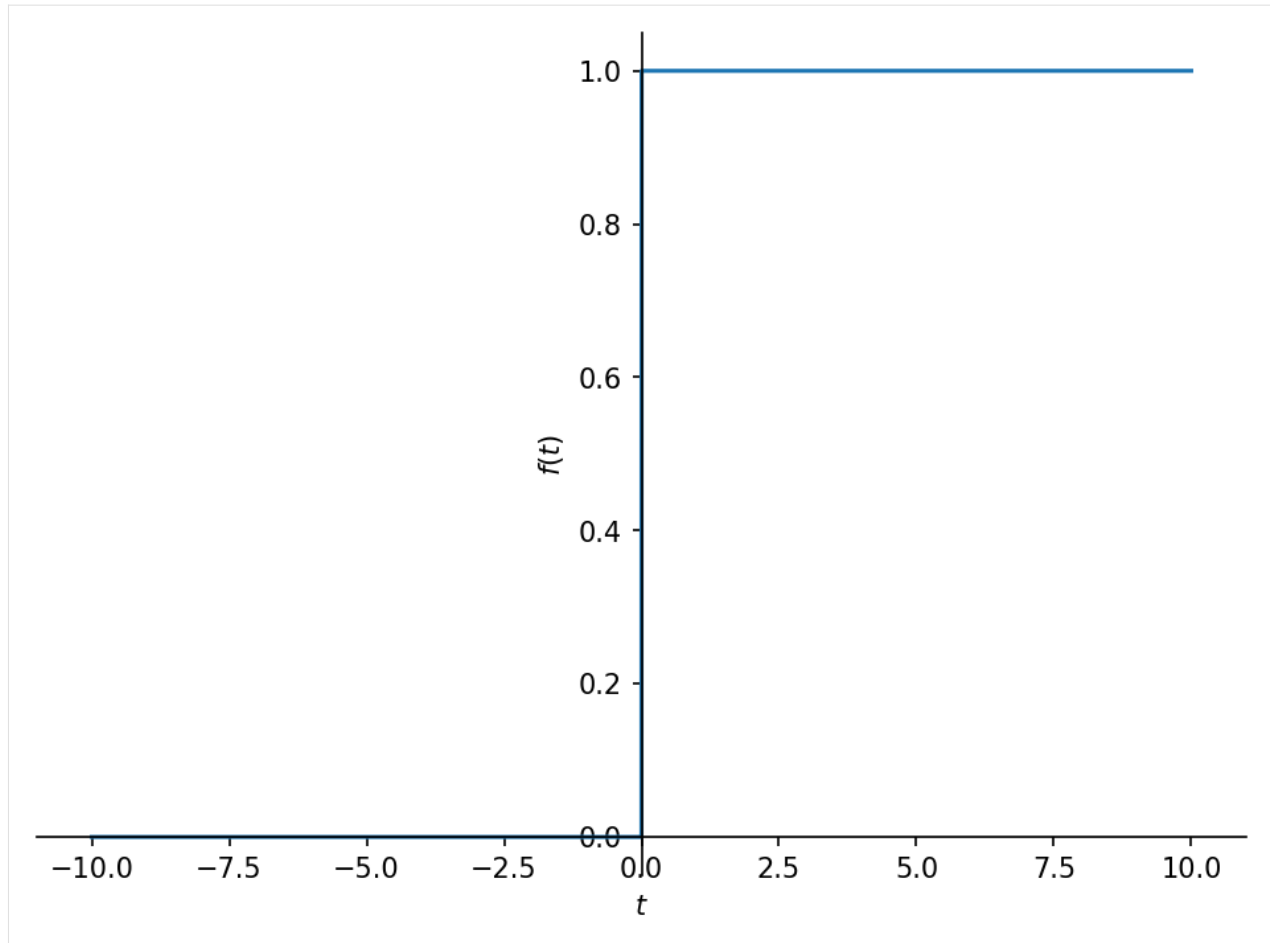
What is that θ ?

The unit step function is also known as the Heaviside step function. We will see this function often in inverse laplace transforms. It is typeset as $\theta(t)$ by sympy.

```
[11]: sympy.Heaviside(t)
```

```
[11]:  $\theta(t)$ 
```

```
[12]: sympy.plot(sympy.Heaviside(t));
```

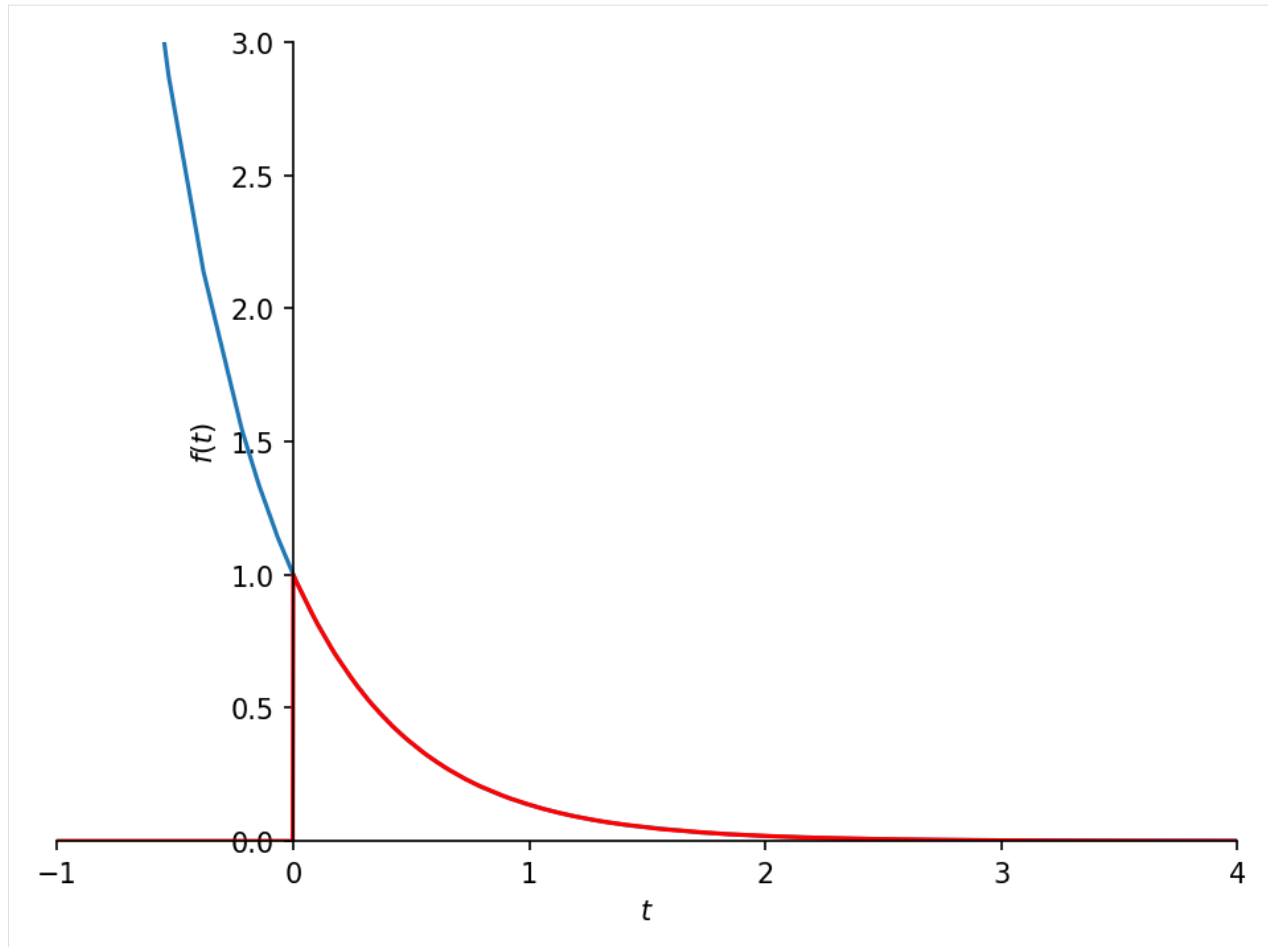



Look at the difference between f and the inverse laplace transform we obtained, which contains the unit step to force it to zero before $t = 0$.

```
[13]: invL(F).subs({a: 2})
```

```
[13]:  $e^{-2t}\theta(t)$ 
```

```
[14]: p = sympy.plot(f.subs({a: 2}), invL(F).subs({a: 2}),
                    xlim=(-1, 4), ylim=(0, 3), show=False)
p[1].line_color = 'red'
p.show()
```

Reproducing standard transform table

Let's see if we can match the functions in the table

```
[15]: omega = sympy.Symbol('omega', real=True)
exp = sympy.exp
sin = sympy.sin
cos = sympy.cos
functions = [1,
             t,
             exp(-a*t),
             t*exp(-a*t),
             t**2*exp(-a*t),
             sin(omega*t),
             cos(omega*t),
             1 - exp(-a*t),
             exp(-a*t)*sin(omega*t),
             exp(-a*t)*cos(omega*t),
             ]
functions
```

```
[15]: [1, t, e-at, te-at, t2e-at, sin(ωt), cos(ωt), 1 - e-at, e-at sin(ωt), e-at cos(ωt)]
```



```
[16]: Fs = [L(f) for f in functions]
      Fs
```

```
[16]: 
$$\left[ \frac{1}{s}, \frac{1}{s^2}, \frac{1}{a+s}, \frac{1}{(a+s)^2}, \frac{2}{(a+s)^3}, \frac{\omega}{\omega^2+s^2}, \frac{s}{\omega^2+s^2}, -\frac{1}{a+s} + \frac{1}{s}, \frac{\omega}{\omega^2+(a+s)^2}, \frac{a+s}{\omega^2+(a+s)^2} \right]$$

```

We can make a pretty good approximation of the table with a little help from pandas

```
[17]: from pandas import DataFrame
```

```
[18]: def makelatemx(args):
      return ["${{}}$.format(sympy.latex(a)) for a in args]
```

```
[19]: DataFrame(list(zip(makelatemx(functions), makelatemx(Fs))))
```

```
[19]:
```

	0	\
0	$\frac{1}{s}$	
1	$\frac{1}{s^2}$	
2	e^{-at}	
3	$t e^{-at}$	
4	$t^2 e^{-at}$	
5	$\sin(\omega t)$	
6	$\cos(\omega t)$	
7	$1 - e^{-at}$	
8	$e^{-at} \sin(\omega t)$	
9	$e^{-at} \cos(\omega t)$	
		1
0		$\frac{1}{s}$
1		$\frac{1}{s^2}$
2		$\frac{1}{a+s}$
3		$\frac{1}{(a+s)^2}$
4		$\frac{2}{(a+s)^3}$
5		$\frac{\omega}{\omega^2+s^2}$
6		$\frac{s}{\omega^2+s^2}$
7		$-\frac{1}{a+s} + \frac{1}{s}$
8		$\frac{\omega}{\omega^2+(a+s)^2}$
9		$\frac{a+s}{\omega^2+(a+s)^2}$

More complicated inverses

Why doesn't the table feature more complicated functions? Because higher-order rational functions can be written as sums of simpler ones through application of partial fractions expansion.

```
[20]: F = ((s + 1)*(s + 2)*(s + 3))/((s + 4)*(s + 5)*(s + 6))
```

```
[21]: F
```

```
[21]: 
$$\frac{(s+1)(s+2)(s+3)}{(s+4)(s+5)(s+6)}$$

```

```
[22]: F.apart(s)
```

```
[22]: 
$$1 - \frac{30}{s+6} + \frac{24}{s+5} - \frac{3}{s+4}$$

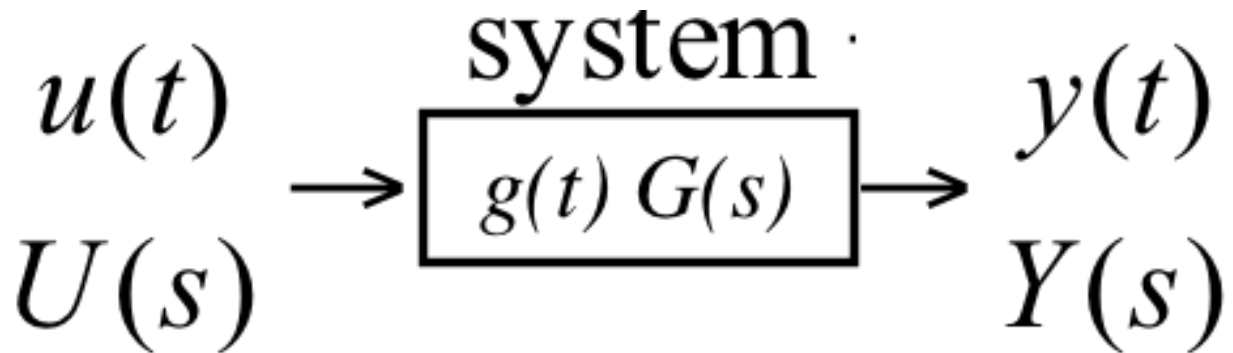
```


In most cases, sympy will be able to figure out how to do the right thing with most of the functions we use, even the hard ones. Notice the relationship between this inverse Laplace and the expression we obtained above:

```
[23]: invL(F)
[23]:  $\delta(t) - 3e^{-4t}\theta(t) + 24e^{-5t}\theta(t) - 30e^{-6t}\theta(t)$ 
```

2.3.4 Convolution and transfer functions

So far, we have calculated the response of systems by finding the Laplace transforms of the input and the system (transfer function), multiplying them and then finding the inverse Laplace transform of the result.



We have been using the idea that, with the nomenclature of the diagram shown above,

$$Y(s) = G(s)U(s). \quad (1)$$

But, if we follow the nomenclature, then $\mathcal{L}\{u(t)\} = U(s)$ and $\mathcal{L}\{y(t)\} = Y(s)$ and similarly there is some $g(t)$ such that $\mathcal{L}\{g(t)\} = G(s)$. So far we have not really used $g(t)$ explicitly, but we know that the time-domain version of equation 1 is given by

$$y(t) = (g * u)(t)$$

where $*$ denotes [convolution](#), which is defined by the following integral:

$$(g * u)(t) = \int_{-\infty}^{\infty} g(\tau)u(t - \tau)d\tau = \int_{-\infty}^{\infty} u(\tau)g(t - \tau)d\tau$$

Since we are primarily concerned with functions where both $g(t) = 0$ and $u(t) = 0$ for $t < 0$, the integral bounds can be written as

$$(g * u)(t) = \int_0^t g(\tau)u(t - \tau)d\tau = \int_0^t u(\tau)g(t - \tau)d\tau$$

This gives us a whole new way to think about the response of a system to an input. Let's revisit the first order step response by thinking about convolution.

```
[1]: import sympy
sympy.init_printing()
%matplotlib inline

[2]: s = sympy.Symbol('s')
t = sympy.Symbol('t', real=True)
tau = sympy.Symbol('tau', real=True, positive=True)
```


We start by considering the first order transfer function:

```
[3]: G = 1/(tau*s + 1)
```

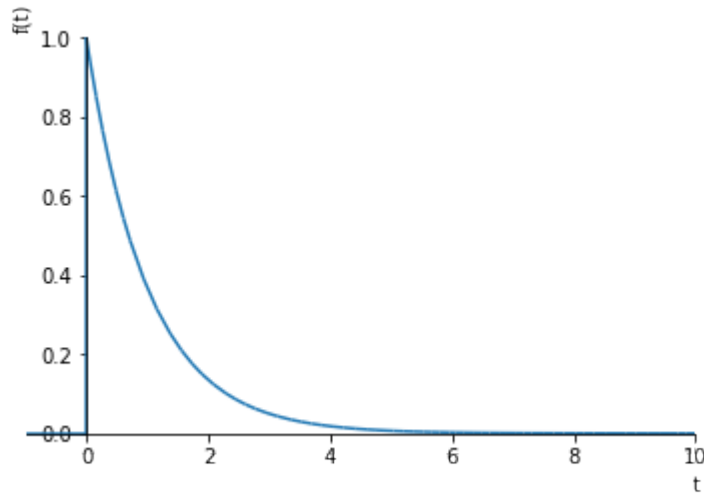
We can interpret $g(t)$ as the *impulse response* of the system, since $\mathcal{L}\{\delta(t)\} = 1$ (in words, the Laplace transform of the Dirac delta is one). So, if $u(t) = \delta(t)$, then $U(s) = 1$, so $Y(s) = G(s)$ and therefore $y(t) = g(t)$.

This is what the impulse response of the first order system looks like with $\tau = 1$:

```
[4]: g = sympy.inverse_laplace_transform(G.subs({tau: 1}), s, t)
g
```

```
[4]: e-tθ(t)
```

```
[5]: sympy.plot(g, (t, -1, 10))
```



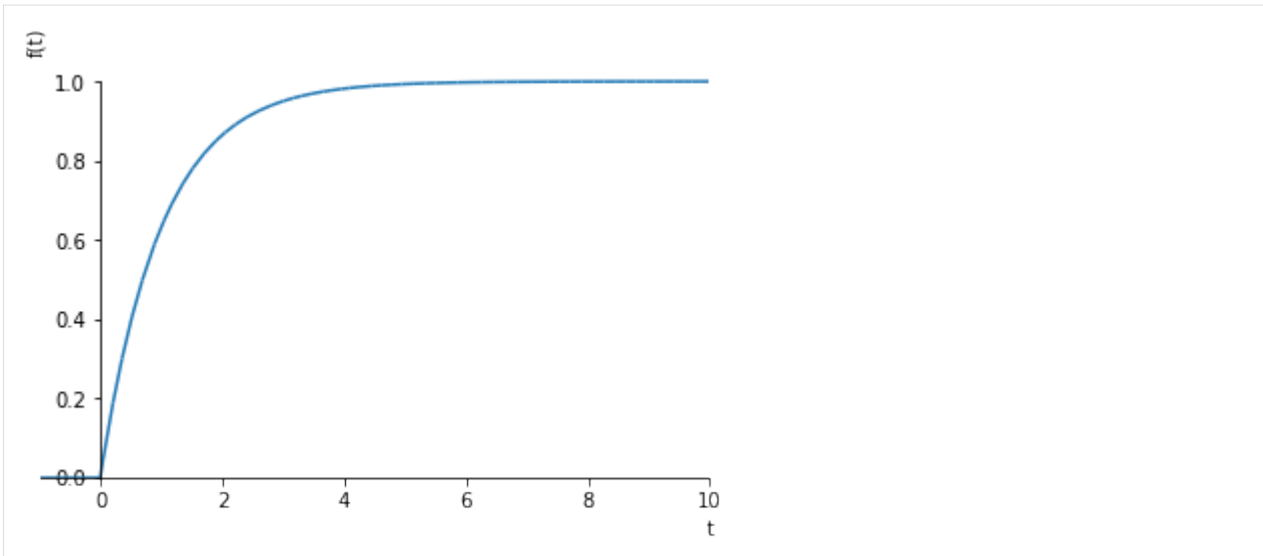
```
[5]: <sympy.plotting.plot.Plot at 0x12024ce10>
```

Earlier, we calculated the step response of the system by calculating $U(s) = \frac{1}{s}$, therefore $Y(s) = \frac{G(s)}{s}$ and then calculating the inverse Laplace:

```
[6]: stepresponse = sympy.inverse_laplace_transform(G.subs({tau: 1})/s, s, t)
stepresponse
```

```
[6]: θ(t) − e-tθ(t)
```

```
[7]: sympy.plot(stepresponse, (t, -1, 10), ylim=(0, 1.1))
```

```
[7]: <sympy.plotting.plot.Plot at 0x1209f5748>
```

We can get the same result by convolving the unit step function with the impulse response. Sympy doesn't handle the improper integral correctly here (as of the below version number):

```
[8]: sympy.__version__
```

```
[8]: '1.4'
```

```
[9]: u = sympy.Heaviside(t)
```

```
[10]: product = g.subs({t: tau})*u.subs({t: t - tau})
```

```
[11]: sympy.integrate(product, (tau, -sympy.oo, sympy.oo))
```

```
[11]: ∞
```

But it does appear to work for the rewritten integral bounds:

```
[12]: sympy.integrate(product, (tau, 0, t))
```

```
[12]:  $-(1 + e^{-t})\theta(t)$ 
```

Numeric convolution

sympy cannot evaluate the convolution integral for all impulse response functions, so it is often useful to do the convolution numerically.

The `numpy.convolve` function calculates the discrete convolution, that is

$$(g * u)[n] = \sum_{m=-\infty}^{\infty} g[m]u[n - m]$$

Let's compare that to the continuous convolution integral:

$$(g * u)(t) = \int_{-\infty}^{\infty} g(\tau)u(t - \tau)d\tau$$

If we discretize this integral to a [Riemann sum](#) with a discrete time step Δt , we obtain

```
:nbsphinx-math: `begin{align} (g*u)(n\Delta t) &\approx \sum_{m=-\infty}^{\infty} g(m\Delta t) u(n\Delta t - m\Delta t) \Delta t \backslash
```

$$\&= \sum_{m=-\infty}^{\infty} g[m] u[n - m] \Delta t \backslash \&= (g*u)[n] \Delta t \backslash$$

```
end{align}`
```

```
[13]: import numpy
import matplotlib.pyplot as plt
```

```
[14]: ts = numpy.linspace(0, 10, 200)
      \Delta t = ts[1] # space between timesteps
```

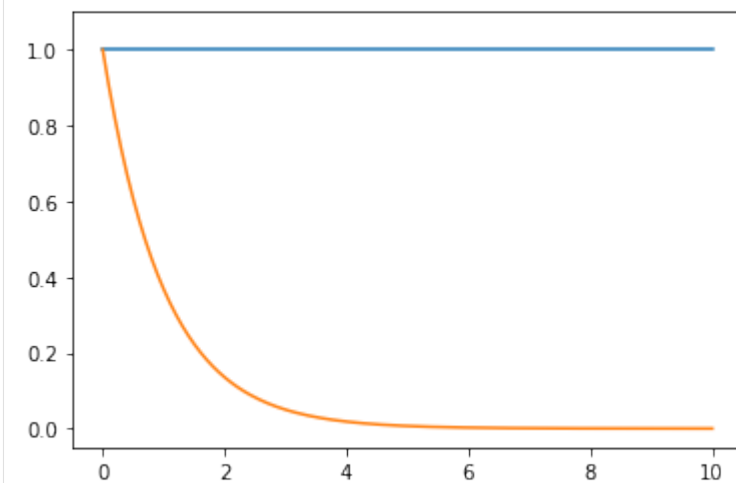
We evaluate the impulse response:

```
[15]: gt = numpy.exp(-ts)
```

```
[16]: ut = numpy.ones_like(ts)
```

```
[17]: plt.plot(ts, ut, ts, gt)
      \plt.ylim(ymax=1.1)
```

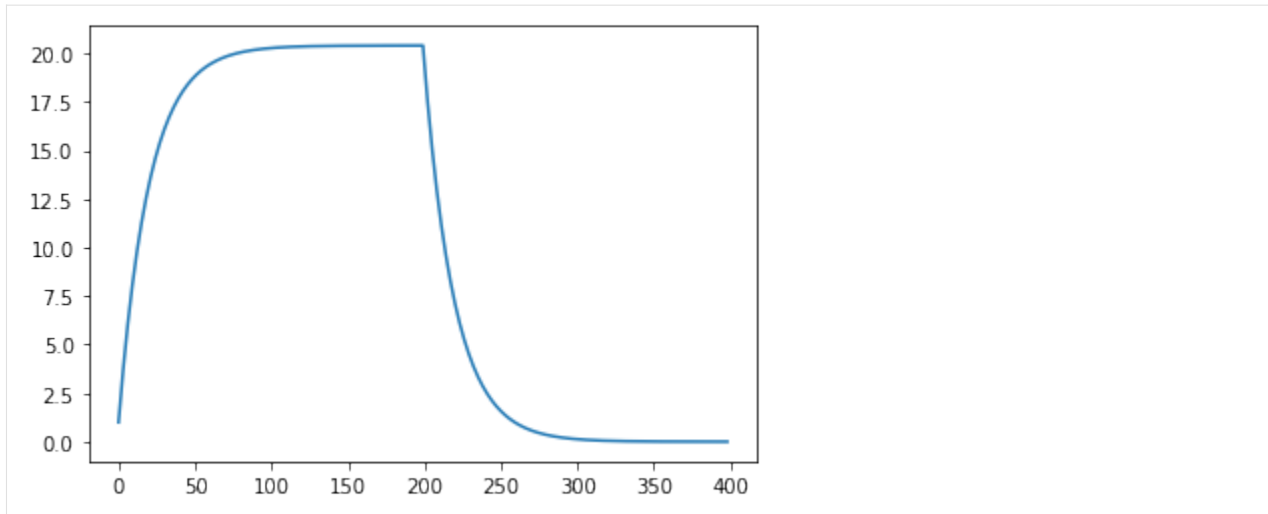
```
[17]: (-0.04995233007374939, 1.1)
```



Also notice that the default behaviour is for the convolution to be calculated over a larger time than originally, so this contains the step response up and down

```
[18]: full_convolution = numpy.convolve(gt, ut)
      \plt.plot(full_convolution)
```

```
[18]: [<matplotlib.lines.Line2D at 0x120ed6eb8>]
```

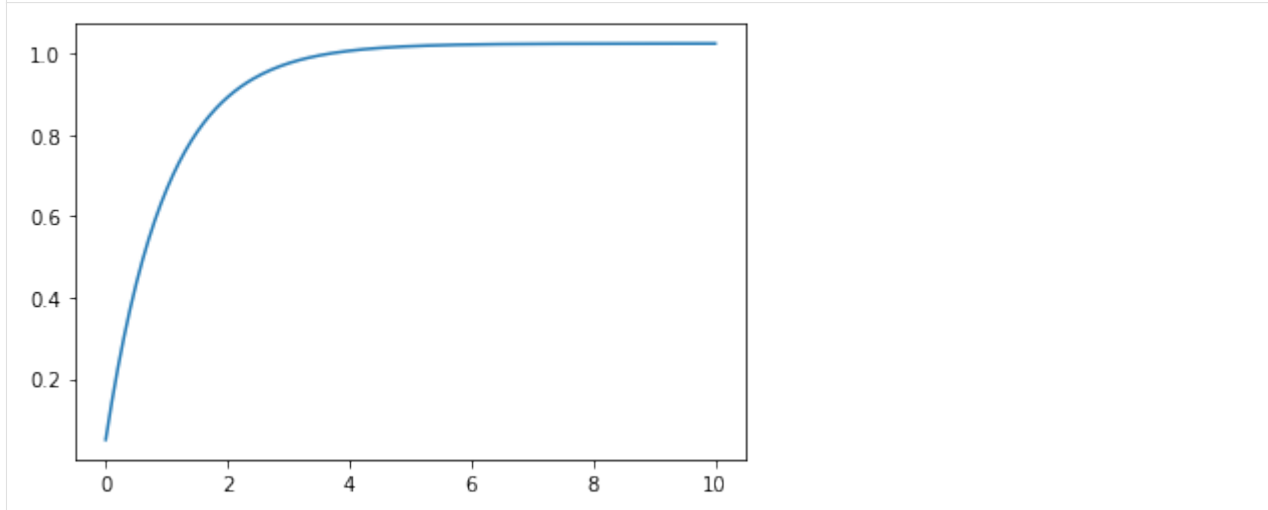



To get the correct integral and just the first part, we can do this:

```
[19]: yt = full_convolution[:len(ts)]*Δt
```

```
[20]: plt.plot(ts, yt)
```

```
[20]: [<matplotlib.lines.Line2D at 0x120faa160>]
```



Notice that this allows us to calculate the response of a system to an arbitrarily complex input numerically. It also gives us a whole new way to think about how a system will behave by thinking about what its impulse response looks like.

2.3.5 Visualising complex functions

One-dimensional functions

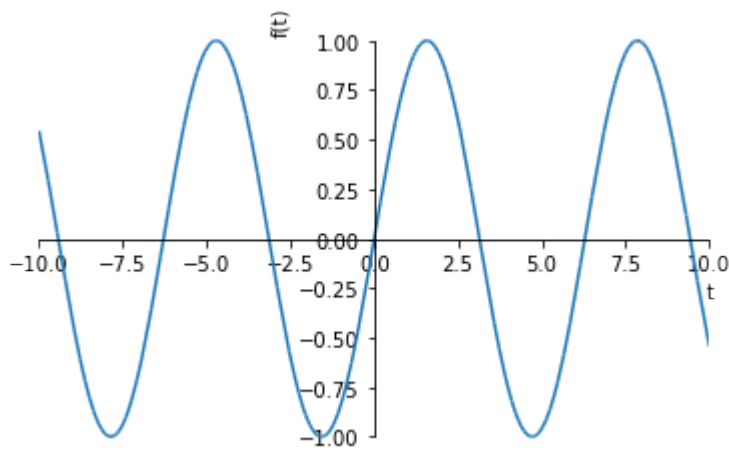
Consider a “normal” plot of $\sin(t)$:

```
[1]: import sympy
      sympy.init_printing()

      %matplotlib inline
```

```
[2]: t = sympy.Symbol('t')
```

```
[3]: sympy.plot(sympy.sin(t))
```



```
[3]: <sympy.plotting.plot.Plot at 0x10f372470>
```

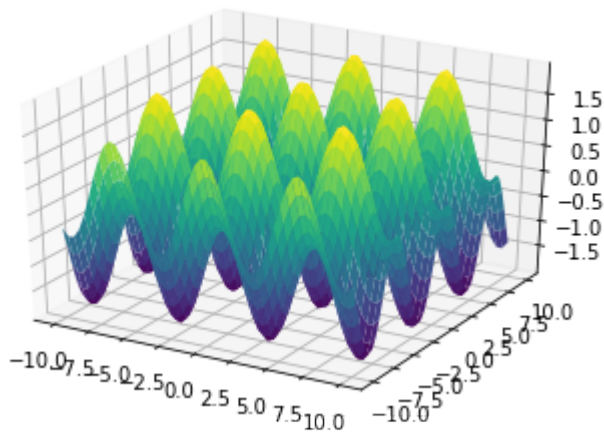
This allows us to see exactly what the value of $\sin(t)$ is for every value of t . To do this, we need two dimensions: one for the value of t and one for $\sin(t)$.

If we want to plot a function of two variables we start needing three dimensions. The following plot shows a “fake” 3d plot. It’s fake because in fact there are only two dimensions available on your computer screen. So we’re already in a bit of trouble when trying to visualise the relationships represented by higher dimensional functions.

```
[4]: x, y = sympy.symbols('x, y')
```

```
[5]: f2 = sympy.sin(x) + sympy.cos(y)
```

```
[6]: sympy.plotting.plot3d(f2)
```

```
[6]: <sympy.plotting.plot.Plot at 0x1113c2320>
```

We can make up for some of this lack by using colors to represent another axis.

```
[7]: import numpy
import matplotlib.pyplot as plt
```

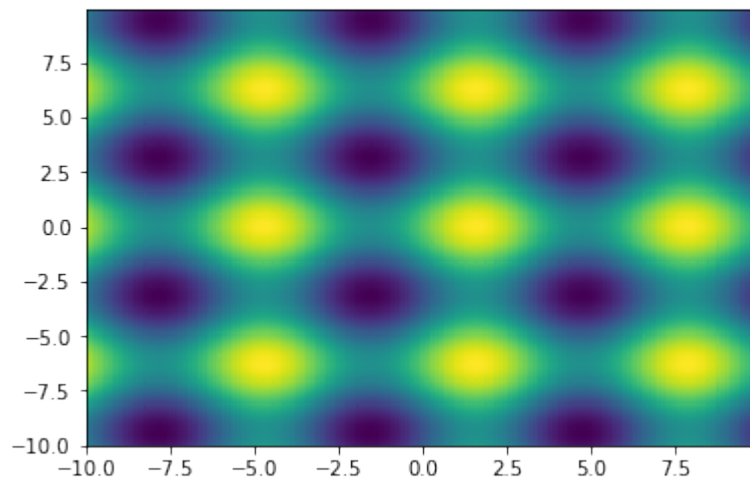
```
[8]: f2numeric = sympy.lambdify((x, y), f2, 'numpy')
```

```
[9]: xx, yy = numpy.mgrid[-10:10:0.1, -10:10:0.1]
```

```
[10]: zz = f2numeric(xx, yy)
```

```
[11]: plt.pcolor(xx, yy, zz)
```

```
[11]: <matplotlib.collections.PolyCollection at 0x111922550>
```

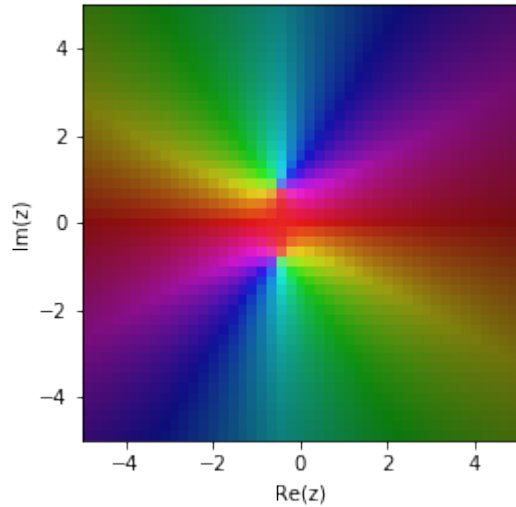


For transfer functions, we have an even bigger challenge, because we have two input variables (the real and imaginary part of s) and two output variables (the real and imaginary part of $G(s)$). One solution to the problem is to use colors for $\angle G(s)$ and brightness for $|G(s)|$. This is known as **domain colouring** and is supplied by `mpmath.cplot`.


```
[12]: try:
      import mpmath
    except:
      from sympy import mpmath
```

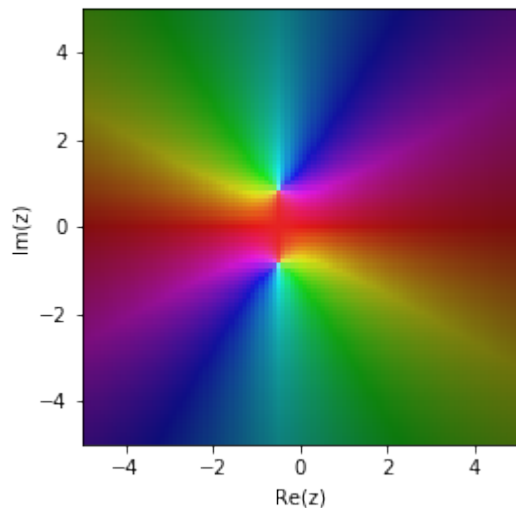
```
[13]: s = sympy.Symbol('s')

      G = 1/(s**2 + s + 1)
      Gnumeric = sympy.lambdify(s, G)
      mpmath.cplot(Gnumeric)
```



We can get a smoother result by plotting with more points

```
[14]: mpmath.cplot(Gnumeric, points=10000)
```

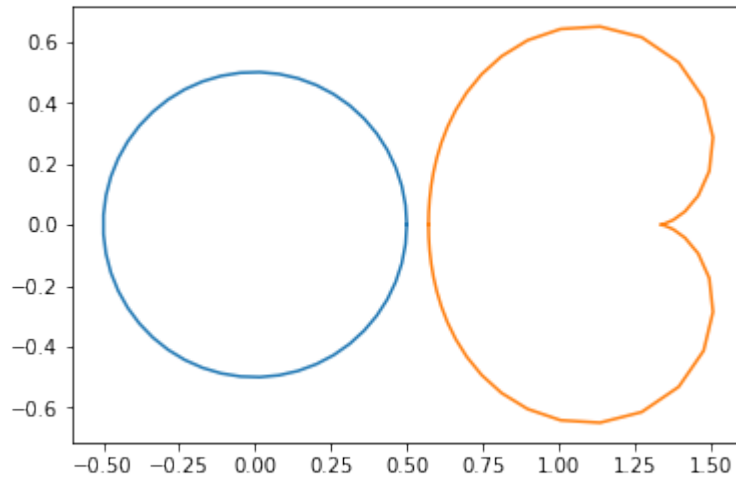


The roots of the denominator appear as two bright regions.

Another solution is to plot the image of $G(s)$ as s goes through a particular path. We will speak more of this when we cover the frequency domain, but have a look at this example. We generate values of s in a circle of diameter 1 (radius 0.5) around the origin, then plot $G(s)$ for all these s values. This is known as the image of $G(s)$ for these values of s .


```
[15]: theta = numpy.linspace(0, 2*numpy.pi) # angles around the circle
      s = 0.5*(numpy.cos(theta) + numpy.sin(theta)*1j) # The circle with radius 0.5
      Gs = Gnumeric(s)
```

```
[16]: plt.plot(s.real, s.imag)
      plt.plot(Gs.real, Gs.imag)
      plt.axis('equal');
```



```
[ ]:
```

2.4 First and second order system Dynamics

2.4.1 Standard process inputs

```
[1]: import sympy
      import matplotlib.pyplot as plt
      sympy.init_printing()
      %matplotlib inline
```

Step

A step input of magnitude M can be written as

$$u_S(t) = \begin{cases} 0 & t < 0, \\ M & t \geq 0 \end{cases}$$

Sympy supplies a unit step function called `Heaviside`, which is typeset as $\theta(t)$

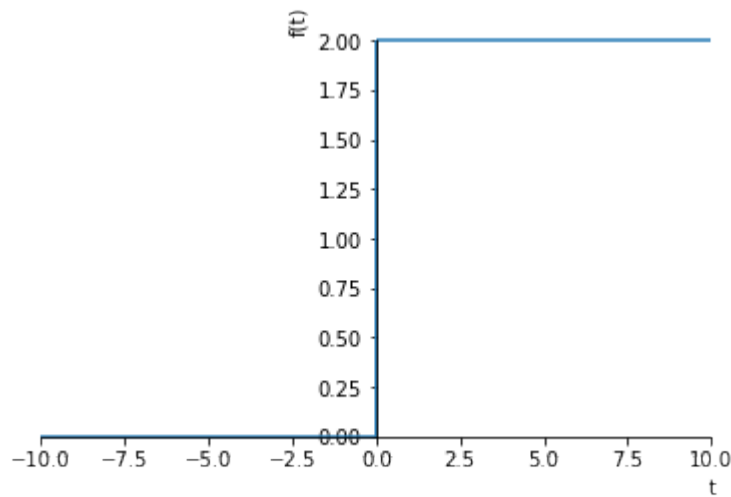
```
[2]: t = sympy.symbols('t')
```

```
[3]: S = sympy.Heaviside
```



```
[4]: M = 2
```

```
[5]: sympy.plot(M*S(t))
```



```
[5]: <sympy.plotting.plot.Plot at 0x1178f46a0>
```

Laplace transform

Sympy can calculate laplace transforms of the step easily:

```
[6]: M, s = sympy.symbols('M, s')
```

```
[7]: def L(f):
      return sympy.laplace_transform(f, t, s, noconds=True)
      def invL(F):
          return sympy.inverse_laplace_transform(F, s, t)
```

```
[8]: L(M*S(t))
```

```
[8]:
```

$$\frac{M}{s}$$

Scaling and translation

We can scale and translate the step function in the normal way. Notice that how the time translation is handled.

```
[9]: from ipywidgets import interact
```

```
[10]: def translated_step(scale, y_translation, t_translation):
      f = scale*S(t - t_translation) + y_translation
      print("f =", f, " \u2112(f) =", L(f))
      sympy.plot(f, (t, -10, 10), ylim=(-2, 4))
```



```
[11]: interact(translated_step,
               scale=(0.5, 3.),
               y_translation=(-1., 1.),
               t_translation=(0., 5.));

interactive(children=(FloatSlider(value=1.75, description='scale', max=3.0, min=0.5),
                      FloatSlider(value=0.0, d...
```

Rectangular pulse

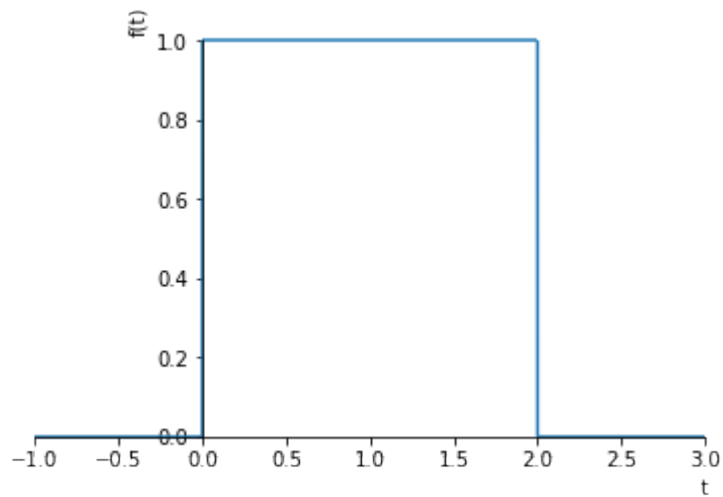
It is now easy to see how we can construct a rectangular pulse, of height h and width t_w ,

$$u_{RP}(t) = \begin{cases} 0 & t < 0, \\ h & 0 \leq t < t_w \\ 0 & t \geq t_w \end{cases}$$

by using shifted versions of the step, so that

```
[12]: h, t_w = sympy.symbols('h, t_w')
u_RP = h*(S(t - 0) - S(t - t_w))

[13]: sympy.plot(u_RP.subs({h: 1, t_w: 2}), (t, -1, 3));
```



```
[14]: L(u_RP)
```

```
[14]: 
$$\frac{h}{s} - \frac{h}{s} e^{-st_w}$$

```


Arbitrary piecewise constant functions

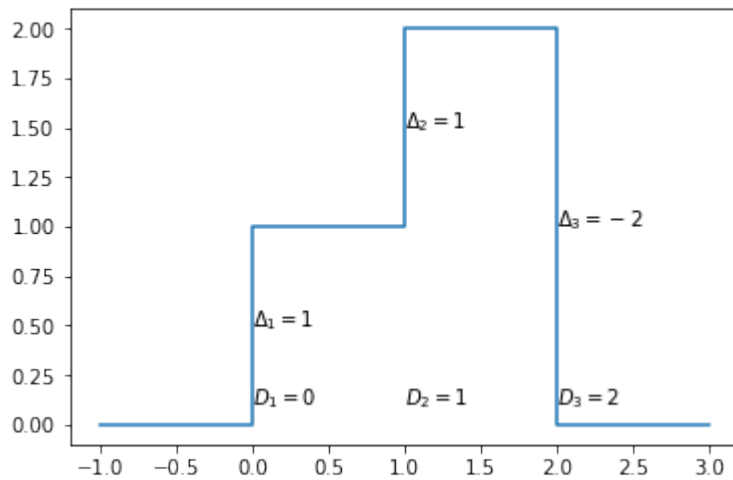
We can construct any piecewise constant function by adding together step functions shifted in time. As an example, we can take the function represented below:

```
[15]: x = [-1, 0, 0, 1, 1, 2, 2, 3]
      y = [0, 0, 1, 1, 2, 2, 0, 0]
```

```
[16]: plt.plot(x, y)
      plt.text(0, 0.5, r'\Delta_1=1$')
      plt.text(1, 1.5, r'\Delta_2=1$')
      plt.text(2, 1, r'\Delta_3=-2$')

      plt.text(0, 0.1, r'$D_1=0$')
      plt.text(1, 0.1, r'$D_2=1$')
      plt.text(2, 0.1, r'$D_3=2$')
```

```
[16]: Text(2,0.1,'$D_3=2$')
```



In general piecewise constant functions like the one above can be written as

$$f_c(t) = \sum_{i=1}^{N_d} \Delta_i S(t - D_i)$$

where S is the unit step. We calculate Δ_i as the difference between the values at the discontinuities, positive if the function is rising and negative if it is falling. D_i is the time at which the value changes and N_d is the number of discontinuities.

We can apply this directly for the example function.

```
[17]: f = 1*S(t) + 1*S(t-1) - 2*S(t-2)
      f
```

```
[17]: \theta(t) - 2\theta(t-2) + \theta(t-1)
```

Or a little more generally using some code:

```
[18]: Delta = [1, 1, -2]
      D = [0, 1, 2]
      Nd = len(Delta)
```



```
[19]: f = sum(Delta[i]*S(t - D[i]) for i in range(Nd))
```

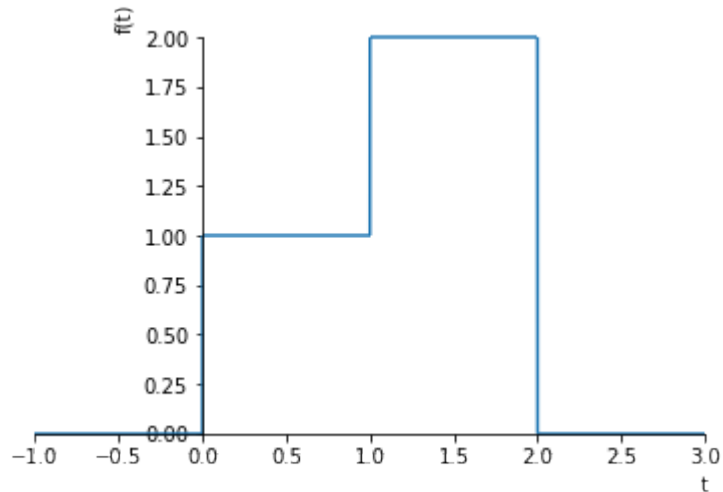
```
[20]: f
```

```
[20]: 
$$\theta(t) - 2\theta(t-2) + \theta(t-1)$$

```

Let's verify it works properly:

```
[21]: sympy.plot(f, (t, -1, 3));
```



```
[22]: sympy.expand(L(f))
```

```
[22]: 
$$\frac{1}{s} + \frac{e^{-s}}{s} - \frac{2}{s}e^{-2s}$$

```

Ramp

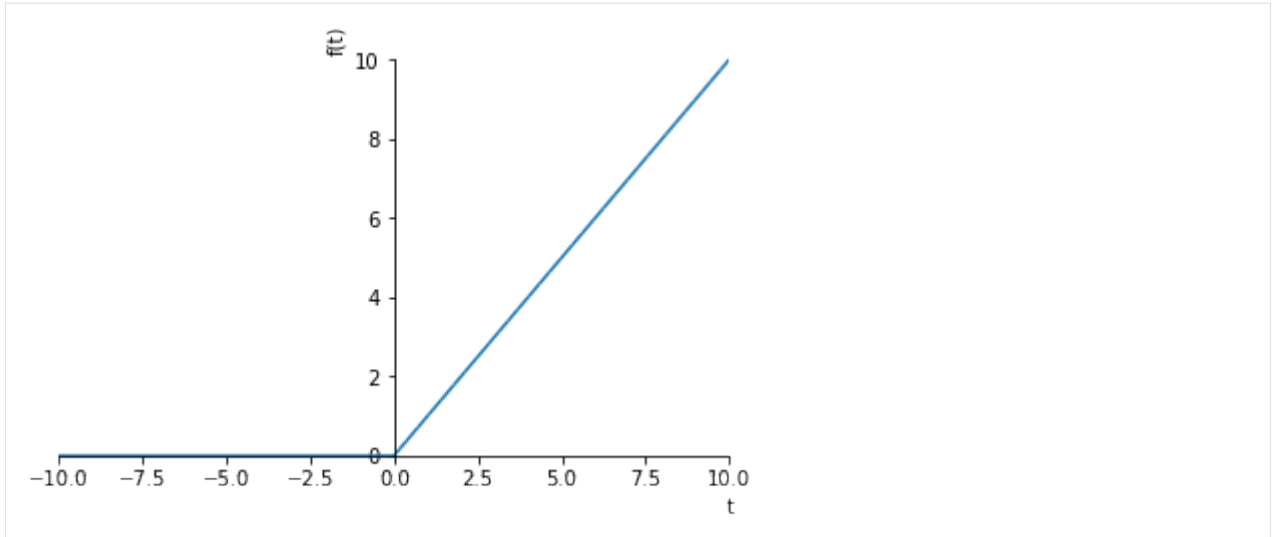
A ramp with slope a can be written as

$$u_R(t) = \begin{cases} 0 & t < 0, \\ a & t \geq 0 \end{cases}$$

We can construct a unit ($a = 1$) ramp by simply multiplying the unit step by t

```
[23]: def R(t):
      return t*S(t)
```

```
[24]: sympy.plot(R(t))
```

[24]: <sympy.plotting.plot.Plot at 0x11a539198>

[25]: $L(R(t))$

[25]: $\frac{1}{s^2}$

Continuous piecewise linear functions

We can build any continuous piecewise linear function by using shifted and scaled ramps.

$$f(t) = \sum_{i=1}^{N_s} \Delta m_i R(t - D_{s,i})$$

This time Δm_i represents changes in slopes ($m = \frac{\Delta y}{\Delta x}$). $D_{s,i}$ are the times at which the slopes change and N_s is the number of slope changes.

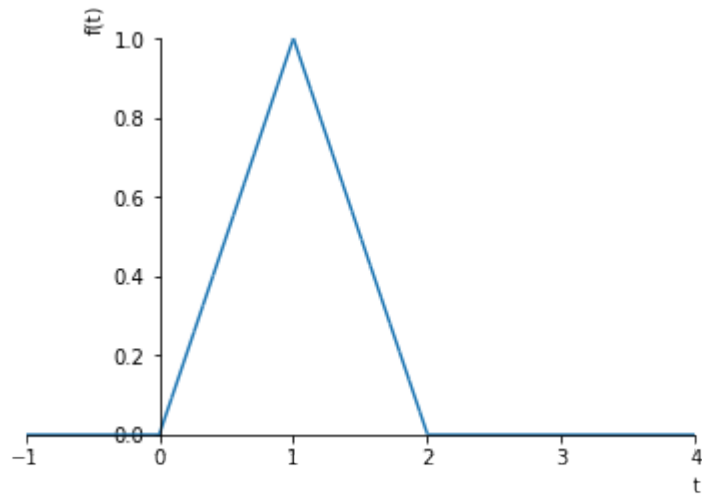
For instance, we can construct a triangular pulse by adding three ramps together.

```
[26]: r1 = t*S(t - 0)
      r2 = -2*(t - t_w/2)*S(t - t_w/2)
      r3 = (t - t_w)*S(t - t_w)
      u_TP = 2/t_w*(r1 + r2 + r3)
```

[27]: u_TP

[27]: $\frac{2}{t_w} \left(t\theta(t) + (-2t + t_w)\theta\left(t - \frac{t_w}{2}\right) + (t - t_w)\theta(t - t_w) \right)$

```
[28]: sympy.plot(u_TP.subs({t_w: 2}), (t, -1, 4))
```

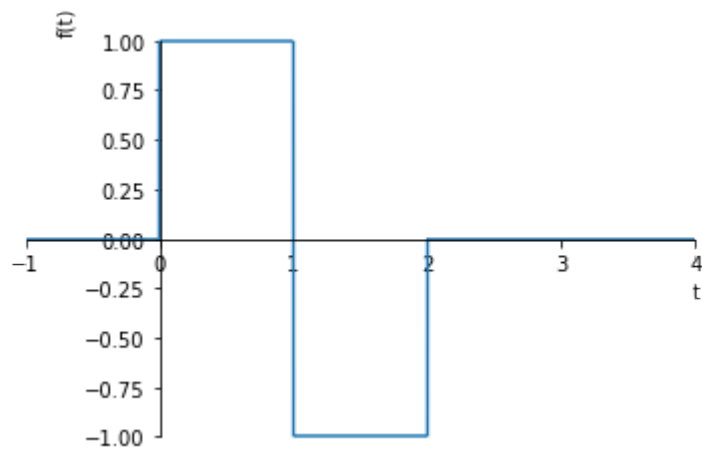
[28]: <sympy.plotting.plot.Plot at 0x11a6de550>

[29]: `L(u_TP.subs({t_w: 2})).expand()`

[29]:
$$\frac{1}{s^2} - \frac{2}{s^2}e^{-s} + \frac{1}{s^2}e^{-2s}$$

Notice that there are three ramps here (one may have expected only two). It becomes more clear when we think about the derivative of this function:

[30]: `sympy.plot(u_TP.diff(t).subs({t_w: 2}), (t, -1, 4), ylim=(-1.1, 1.1))`



[30]: <sympy.plotting.plot.Plot at 0x11c5a5710>

We now see that the derivative of a piecewise linear continuous function is a piecewise constant function. We can apply our rule for piecewise constant functions and integrate the steps to ramps:

[31]: `derivative = 1*S(t-0) - 2*S(t-1) + 1*S(t-2)`

[32]: `final = 1*R(t-0) - 2*R(t-1) + 1*R(t-2)`

Arbitrary piecewise linear functions

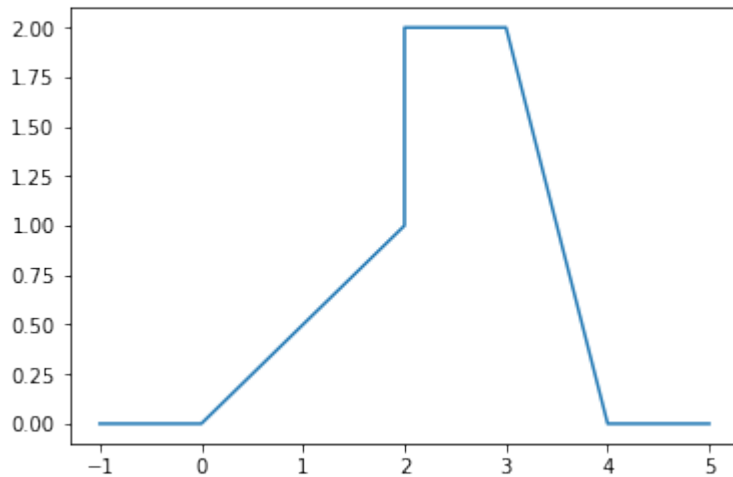
We can construct any piecewise linear function by adding together ramp functions and steps shifted in time. The general rule now becomes

$$f(t) = \underbrace{\sum_{i=1}^{N_s} \Delta m_i R(t - D_{s,i})}_{\text{slope changes}} + \underbrace{\sum_{i=1}^{N_d} \Delta_i S(t - D_i)}_{\text{discontinuities}}$$

```
[33]: x = [-1, 0, 2, 2, 3, 4, 5]
      y = [0, 0, 1, 2, 2, 0, 0]
```

```
[34]: plt.plot(x, y)
```

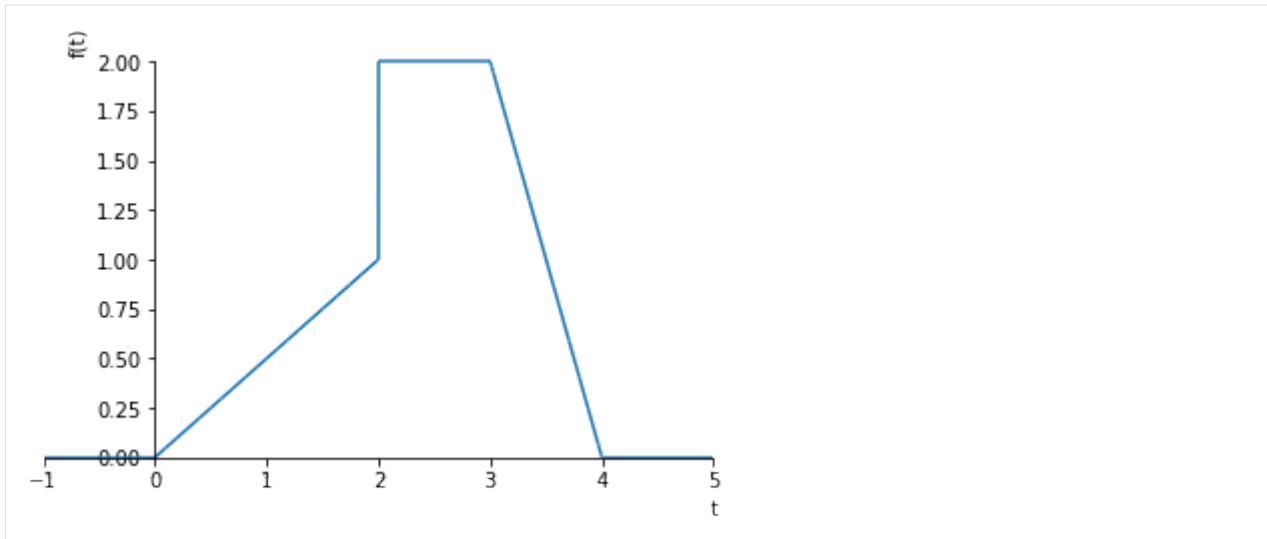
```
[34]: [<matplotlib.lines.Line2D at 0x11ac04d30>]
```



We see that there are 4 slope changes (at $t=0$, $t=2$, $t=3$ and $t=4$) and 1 discontinuity. Applying our formula yields:

```
[35]: g = 0.5*R(t) - 0.5*R(t-2) - 2*R(t-3) + 2*R(t - 4) + S(t - 2)
```

```
[36]: sympy.plot(g, (t, -1, 5));
```

```
[37]: sympy.expand(L(g))
```

```
[37]: 
$$\frac{1.0}{s}e^{-2s} + \frac{0.5}{s^2} - \frac{0.5}{s^2}e^{-2s} - \frac{2.0}{s^2}e^{-3s} + \frac{2.0}{s^2}e^{-4s}$$

```

2.4.2 First order systems

```
[1]: import sympy
import matplotlib.pyplot as plt
import numpy
sympy.init_printing()
%matplotlib inline
```

```
[2]: t, K, tau = sympy.symbols('t, K, tau', real=True, positive=True)
s = sympy.Symbol('s')
```

```
[3]: u = sympy.Heaviside(t)
```

```
[4]: def L(f):
    return sympy.laplace_transform(f, t, s, noconds=True)
def invL(F):
    return sympy.inverse_laplace_transform(F, s, t)
```

```
[5]: U = L(u)
U
```

```
[5]: 
$$\frac{1}{s}$$

```

All first order linear differential equations with constant coefficients can be rewritten in the following form:

$$\frac{dy}{dt} = ay(t) + bu(t)$$

Where y is the output and u is the input or forcing function.

If we Laplace transform this, we end up with

$$\mathcal{L}\left\{\frac{dy}{dt}\right\} = \mathcal{L}\{ay(t) + bu(t)\} \quad (2.17)$$

$$sy(s) = ay(s) + bu(s) \quad (2.18)$$

$$y(s) = \frac{b}{s-a}u(s) \quad (2.19)$$

$$(2.20)$$

By convention, we usually rewrite the above in the following form, for reasons which will become apparent soon:

```
[6]: G = K/(tau*s + 1)
      G
```

```
[6]:
```

$$\frac{K}{s\tau + 1}$$

The inverse laplace of a transfer function is its impulse response

```
[7]: impulseresponse = invL(G)
      impulseresponse
```

```
[7]:
```

$$\frac{K}{\tau}e^{-\frac{t}{\tau}}$$

If $u(t)$ is the unit step function, $U(s) = \frac{1}{s}$ and we can obtain the step response as follows:

```
[8]: u = 1/s
      stepresponse = invL(G*u)
```

```
[9]: stepresponse
```

```
[9]:
```

$$K - Ke^{-\frac{t}{\tau}}$$

Ramp response:

```
[10]: u = 1/s**2
        rampresponse = invL(G*u)
        rampresponse
```

```
[10]:
```

$$Kt - K\tau + K\tau e^{-\frac{t}{\tau}}$$

```
[11]: from ipywidgets import interact
```

```
[12]: evalfimpulse = sympy.lambdify((K, tau, t), impulseresponse, 'numpy')
        evalfstep = sympy.lambdify((K, tau, t), stepresponse, 'numpy')
        evalframp = sympy.lambdify((K, tau, t), rampresponse, 'numpy')
```



```
[13]: ts = numpy.linspace(0, 10)

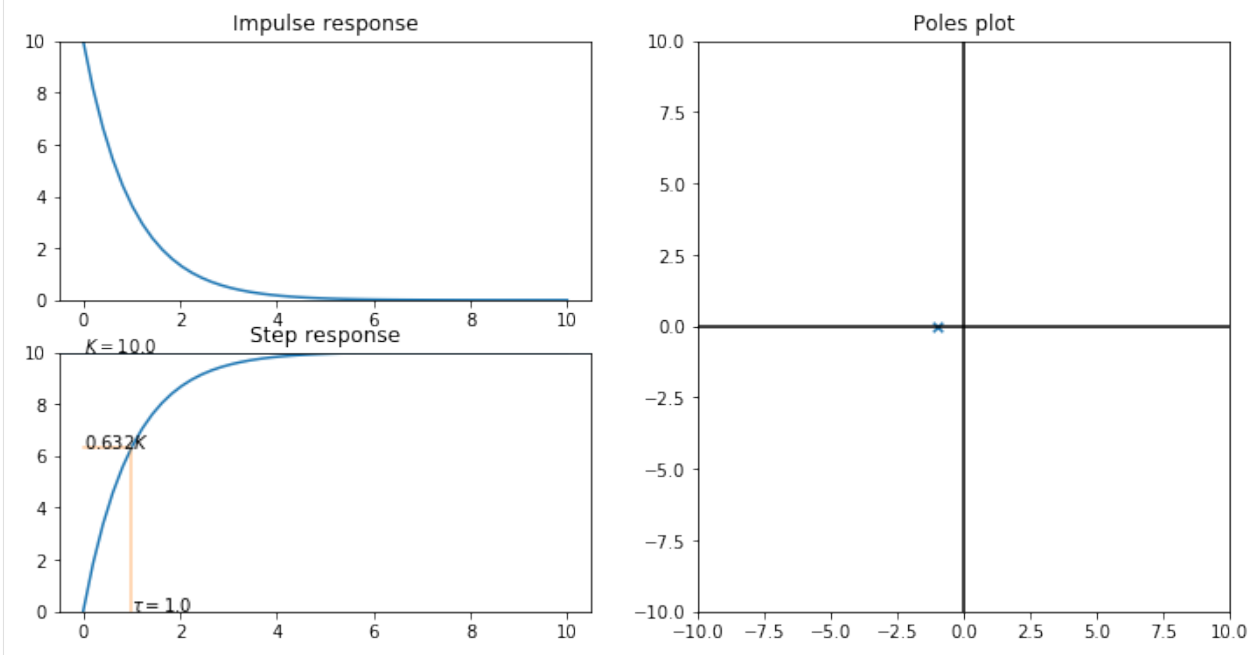
def firstorder(tau_in, K_in):
    plt.figure(figsize=(12, 6))
    ax_impulse = plt.subplot2grid((2, 2), (0, 0))
    ax_step = plt.subplot2grid((2, 2), (1, 0))
    ax_complex = plt.subplot2grid((2, 2), (0, 1), rowspan=2)

    ax_impulse.plot(ts, evalfimpulse(K_in, tau_in, ts))
    ax_impulse.set_title('Impulse response')
    ax_impulse.set_ylim(0, 10)

    tau_height = 1 - numpy.exp(-1)
    ax_step.set_title('Step response')
    ax_step.plot(ts, evalfstep(K_in, tau_in, ts))
    ax_step.axhline(K_in)
    ax_step.plot([0, tau_in, tau_in], [K_in*tau_height]*2 + [0], alpha=0.4)
    ax_step.text(0, K_in, '$K=${}'.format(K_in))
    ax_step.text(0, K_in*tau_height, '{:.3}$K$'.format(tau_height))
    ax_step.text(tau_in, 0, r'$\tau={:.3}$'.format(tau_in))
    ax_step.set_ylim(0, 10)

    ax_complex.set_title('Poles plot')
    ax_complex.scatter(-1/tau_in, 0, marker='x', s=30)
    ax_complex.axhline(0, color='black')
    ax_complex.axvline(0, color='black')
    ax_complex.axis([-10, 10, -10, 10])
```

```
[14]: firstorder(1., 10.)
```



```
[15]: interact(firstorder, tau_in=(0.1, 10.), K_in=(0.1, 10.));

interactive(children=(FloatSlider(value=5.05, description='tau_in', max=10.0, min=0.
↪1), FloatSlider(value=5.05...
```


Exploration of the above interaction allows us to see the following:

- K scales the response in the y direction
- τ scales the response in the t direction
- The response of the system is always $0.63K$ when $t = \tau$

We get the “magic number” 0.63 by substituting $t = \tau$ into the response:

```
[16]: sympy.N((stepresponse.subs(t, tau)/K).simplify())
```

```
[16]: 0.632120558828558
```

```
[1]: import sympy
import matplotlib.pyplot as plt
import numpy
sympy.init_printing()
%matplotlib inline
```

```
[2]: tau, zeta, t, w, K = sympy.symbols('tau, zeta, t, w, K', real=True, positive=True)
s = sympy.Symbol('s')
```

```
[3]: def L(f):
    return sympy.laplace_transform(f, t, s, noconds=True)
def invL(F):
    return sympy.inverse_laplace_transform(F, s, t)
```

This is the standard form for the second order system transfer function

```
[4]: G = K/(tau**2*s**2 + 2*tau*zeta*s + 1)
G
```

```
[4]: 
$$\frac{K}{s^2\tau^2 + 2s\tau\zeta + 1}$$

```

In recent versions of SymPy, we can solve for the step response directly.

```
[5]: sympy.__version__
```

```
[5]: '1.1.1'
```

```
[6]: invL(G/s)
```

```
[6]: 
$$\frac{Ke^{-\frac{t\zeta}{\tau}}}{2(-\zeta + \sqrt{\zeta - 1}\sqrt{\zeta + 1})^2(\zeta + \sqrt{\zeta - 1}\sqrt{\zeta + 1})^2\sqrt{\zeta^2 - 1}} \left( -i\zeta e^{\frac{t}{\tau}\sqrt{\zeta + 1}} \cos\left(\frac{1}{2}\operatorname{atan}_2(0, \zeta - 1)\right)\sqrt{|\zeta - 1|} \sin\left(\frac{t}{\tau}\sqrt{\zeta + 1}\right) \sin\left(\frac{1}{2}\operatorname{atan}_2(0, \zeta - 1)\right) \right.$$

```

That’s a really hairy expression, so let’s try to simplify.

The characteristic equation is the denominator of the transfer function

```
[7]: ce = sympy.Eq(sympy.denom(G), 0)
ce
```


[7]:
$$s^2\tau^2 + 2s\tau\zeta + 1 = 0$$

[8]: `roots = sympy.roots(ce, s)`
`roots`

[8]:
$$\left\{ \frac{1}{\tau} \left(-\zeta - \sqrt{\zeta - 1} \sqrt{\zeta + 1} \right) : 1, \quad \frac{1}{\tau} \left(-\zeta + \sqrt{\zeta - 1} \sqrt{\zeta + 1} \right) : 1 \right\}$$

The shape of the inverse Laplace depends on the nature of the roots, which depends on the value of ζ

Overdamped: $\zeta > 1$. Two distinct real roots

[9]: `invL(G.subs({zeta: 7})).simplify().expand()`

[9]:
$$\frac{\sqrt{3}K}{24\tau} e^{-\frac{7t}{\tau}} e^{\frac{4t}{\tau}\sqrt{3}} - \frac{\sqrt{3}K}{24\tau} e^{-\frac{7t}{\tau}} e^{-\frac{4t}{\tau}\sqrt{3}}$$

Critically damped: $\zeta = 1$. Repeated roots.

[10]: `invL(G.subs({zeta: 1}))`

[10]:
$$\frac{Kt}{\tau^2} e^{-\frac{t}{\tau}}$$

Underdamped: $0 < \zeta < 1$: a complex conjugate pair of roots

[11]: `r = invL(G.subs({zeta: 0.7}))`
`r`

[11]:
$$\frac{1.40028008402801K}{\tau} e^{-\frac{0.7t}{\tau}} \sin\left(\frac{0.714142842854285t}{\tau}\right)$$

We can get prettier results if we use Rationals instead of floats

[12]: `r = invL(G.subs({zeta: sympy.Rational(1, 2)}))`
`r`

[12]:
$$\frac{2\sqrt{3}K}{3\tau} e^{-\frac{t}{2\tau}} \sin\left(\frac{\sqrt{3}t}{2\tau}\right)$$

[13]: `from ipywidgets import interact`

[14]: `def secondorder(K_in, tau_in, zeta_in, tmax):`
 `values = {K: sympy.nsimplify(K_in), tau: sympy.nsimplify(tau_in), zeta: sympy.`
 `↪ nsimplify(zeta_in)}`
 `stepresponse = sympy.re(invL(G.subs(values)/s))`
 `sympy.plot(stepresponse, (t, 0, tmax), ylim = [0, 10])`


```
[15]: interact(secondorder, K_in=(0, 10.), tau_in=(0., 10.), zeta_in=(0., 2), tmax=(30., ↵
↵100));

interactive(children=(FloatSlider(value=5.0, description='K_in', max=10.0), ↵
↵FloatSlider(value=5.0, description=...
```

```
[ ]:
```

```
[ ]:
```

```
[1]: import sympy
sympy.init_printing()
%matplotlib inline
```

2.4.3 Sinusoidal response

In this notebook we will look at the response of first and second order systems to sinusoidal inputs. Recall that we always assume inputs were zero for times less than zero, so in fact the input looks like this:

$$u_{sin}(t) = \begin{cases} 0 & t < 0 \\ A \sin(\omega t) & t \geq 0 \end{cases}$$

We will define our symbols in such a way that the positive t is assumed and Sympy will do the math correctly.

```
[24]: A, t, omega = sympy.symbols('A, t, omega', positive=True)
s = sympy.Symbol('s')
```

So we can define our input like this and get the same Laplace transform as in our table of standard transforms:

```
[25]: usin = A*sympy.sin(t*omega)
```

```
[26]: usin_s = sympy.laplace_transform(usin, t, s, noconds=True)
usin_s
```

```
[26]: 
$$\frac{A\omega}{\omega^2 + s^2}$$

```

First order

Let's explore how first order systems respond to this kind of input:

```
[27]: K, tau = sympy.symbols('K, tau', positive=True)
```

```
[28]: G = K/(tau*s + 1)
```

```
[29]: y = G*usin_s
```

```
[35]: yt = sympy.inverse_laplace_transform(y, s, t)
yt
```


[35]:

$$\frac{AKe^{-\frac{t}{\tau}}}{(i\omega\tau - 1)^2 (i\omega\tau + 1)^2} (\omega^2\tau^2 + 1) \left(-\omega\tau e^{\frac{t}{\tau}} \cos(\omega t) + \omega\tau + e^{\frac{t}{\tau}} \sin(\omega t) \right)$$

[37]: `yt = yt.simplify().expand()`
`yt`

[37]:

$$-\frac{AK\omega\tau e^{\frac{t}{\tau}} \cos(\omega t)}{\omega^2\tau^2 e^{\frac{t}{\tau}} + e^{\frac{t}{\tau}}} + \frac{AK\omega\tau}{\omega^2\tau^2 e^{\frac{t}{\tau}} + e^{\frac{t}{\tau}}} + \frac{AKe^{\frac{t}{\tau}} \sin(\omega t)}{\omega^2\tau^2 e^{\frac{t}{\tau}} + e^{\frac{t}{\tau}}}$$

[38]: `def response(omega_, A_, tau_, K_):`
 `sympy.plot(usin.subs({omega: omega_, A: A_})),`
 `yt.subs({A: A_, tau: tau_, K: K_, omega: omega_})),`
 `(t, 0, 30), ylim=(-2, 2))`

[39]: `from ipywidgets import interact`

[40]: `interact(response,`
 `omega_=(0.1, 10.),`
 `A_=(0.1, 2.),`
 `tau_=(0.1, 10.),`
 `K_=(-0.1, 2.))`
`interactive(children=(FloatSlider(value=5.05, description='omega_', max=10.0, min=0.`
`↪1), FloatSlider(value=1.05...`

[40]: `<function __main__.response(omega_, A_, tau_, K_)>`

We see the response is eventually sinusoidal.

Second order sinusoidal response

[41]: `zeta = sympy.Symbol('zeta', positive=True)`

[42]: `G2 = K/(tau**2*s**2 + 2*tau*zeta*s + 1)`
`G2`

[42]:

$$\frac{K}{s^2\tau^2 + 2s\tau\zeta + 1}$$

Sympy can calculate this response analytically.

Warning: This next cell takes quite a long time.

[43]: `yt = sympy.inverse_laplace_transform(G2*usin_s, s, t)`

[44]: `def response(omega_, A_, K_, tau_, zeta_):`
 `sympy.plot(usin.subs({omega: omega_, A: A_})),`
 `yt.subs({A: A_, tau: tau_, K: K_, zeta: zeta_, omega: omega_})),`
 `(t, 0, 20), ylim=(-2, 2))`

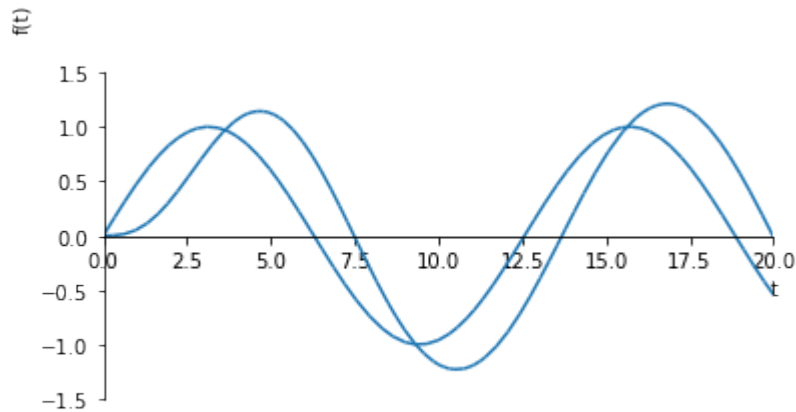

```
[45]: interact(response,
               omega_=(0.1, 10.),
               A_=(0.1, 2.),
               tau_=(0.1, 10.),
               K_=(-0.1, 2.),
               zeta_=(0.1, 1.2))

interactive(children=(FloatSlider(value=5.05, description='omega_', max=10.0, min=0.
↪1), FloatSlider(value=1.05...
```

```
[45]: <function __main__.response(omega_, A_, K_, tau_, zeta_)>
```

Again we see that the response is eventually sinusoidal, with a longer transient. Unlike the first order system, there are frequencies where the output is larger than the input when the system is underdamped. This is known as a harmonic response:

```
[18]: response(0.5, 1, 1, 1.1, 0.4)
```



Amplitude over frequency

It can be shown (unfortunately SymPy is not quite up to the task) that for first order processes, the eventual magnitude of the sinusoidal responses are as follows:

First order processes

$$\frac{KA}{\sqrt{\omega^2\tau^2 + 1}}$$

Second order processes

$$\frac{KA}{\sqrt{(1 - \omega^2\tau^2)^2 + (2\zeta\omega\tau)^2}}$$

It is useful to plot the normalised amplitude ratio (the above amplitudes divided by KA) of the two systems as a function of frequency:


```
[19]: firstorder = 1/sympy.sqrt(omega**2*tau**2 + 1)
      secondorder = 1/sympy.sqrt((1 - omega**2*tau**2)**2 + (2*zeta*omega*tau)**2)
```

```
[20]: def frequencyplot(tau_, zeta_):
      sympy.plot(firstorder.subs({tau: tau_}),
                  secondorder.subs({tau: tau_, zeta: zeta_}),
                  (omega, 0.1, 10),
                  xscale='log',
                  )
```

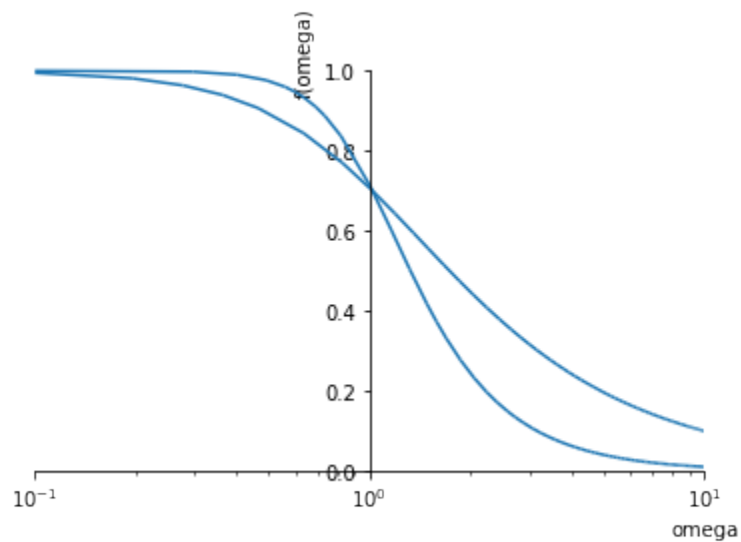
```
[21]: interact(frequencyplot, tau_=(0.01, 4), zeta_=(0.1, 1.2))

interactive(children=(FloatSlider(value=2.005, description='tau_', max=4.0, min=0.01),
  ↪ FloatSlider(value=0.649...
```

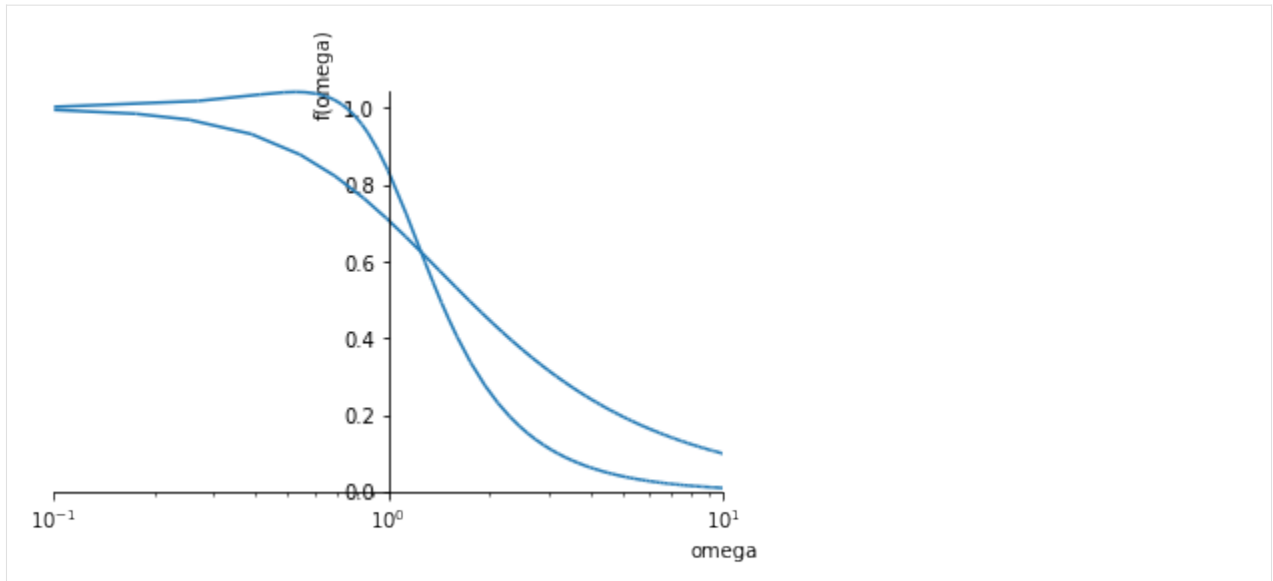
```
[21]: <function __main__.frequencyplot(tau_, zeta_)>
```

It is clear that the value of τ determines the frequency of the peak in the second order plot. This peak is known as the “harmonic nose” and is only larger than 1 when $0 < \zeta < 0.7$.

```
[22]: frequencyplot(1, 0.7)
```



```
[23]: frequencyplot(1, 0.6)
```

[]:

[]:

2.5 Complex system dynamics

2.5.1 Random response generator

This sheet will generate random systems and show their step responses. See if you can predict the responses from the transfer functions and the poles and zeros.

This notebook assumes version 0.8.0 of the control library or better

```
[1]: import control
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[2]: def viz(order):
    # Not all random transfer functions work, so we generate one
    # and try to calculate the step response, only continuing when it works
    valid = False
    while not valid:
        coeffs = (numpy.random.random(order)*3).tolist() + [1]
        G = control.tf(1, coeffs)

        try:
            t, y = control.step_response(G)
            valid = True
        except ValueError:
            continue

    fig, [ax_complex, ax_time] = plt.subplots(1, 2, figsize=(10, 5))
```

(continues on next page)

(continued from previous page)

```
plt.sca(ax_complex)
control.pzmap(G)
ax_complex.axis('equal')
ax_complex.axis([-5, 5, -5, 5])
ax_complex.grid()
ax_time.plot(t, y)
ax_time.axhline(1)
```

```
[3]: from ipywidgets import interact
```

```
[4]: interact(viz, order=(1, 5))
```

```
interactive(children=(IntSlider(value=3, description='order', max=5, min=1),
↳Output()), _dom_classes=('widget-...
```

```
[4]: <function __main__.viz(order)>
```

2.5.2 Simulation of arbitrary transfer functions

In some cases we can calculate the response of a system to an input completely analytically using SymPy as discussed in other notebooks. Sometimes these methods are not sufficient as they fail to calculate the inverse or because we have different inputs. This notebook covers several methods of simulating systems of arbitrary complexity.

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

Convert to ODE and integrate manually

We are very familiar with the Euler integration method by now. Let's recap a quick simulation of a first-order system

$$G = \frac{y(s)}{u(s)} = \frac{K}{\tau s + 1}$$

We can rewrite this in the time domain as

$$y(s)(\tau s + 1) = K u(s)$$

$$\tau \frac{dy}{dt} + y(t) = K u(t)$$

$$\frac{dy}{dt} = -\frac{1}{\tau} y(t) + \frac{K}{\tau} u(t)$$

```
[2]: K = 1
tau = 5
```

This is our input function. Note that it could be anything (not just a step)

```
[3]: def u(t):
    if t < 1:
        return 0
    else:
        return 1
```

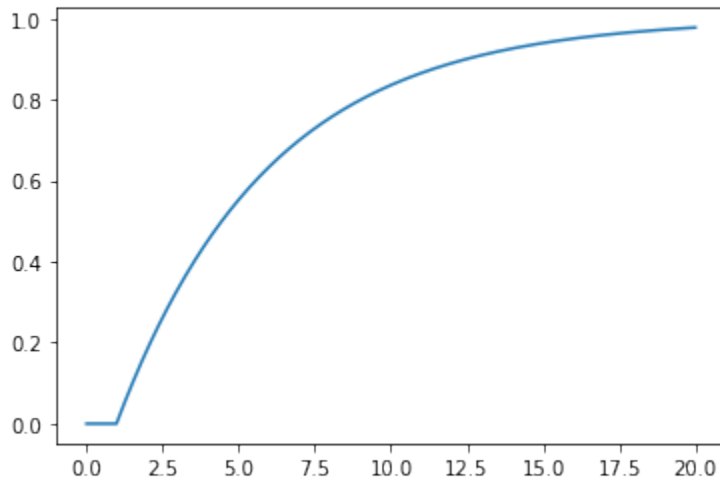


```
[4]: ts = numpy.linspace(0, 20, 1000)
      dt = ts[1]
      y = 0
      ys = []
      for t in ts:
          dydt = -1/tau*y + 1/tau*u(t)

          y += dydt*dt
          ys.append(y)
```

```
[5]: plt.plot(ts, ys)
```

```
[5]: [<matplotlib.lines.Line2D at 0x231e91d1eb0>]
```



LTI support in `scipy.signal`

Notice in the previous code that all the differential equations were linear and that none of the coefficients of the variables change over time. Such a system is known as a Linear, Time Invariant (LTI) system. The `scipy.signal` module supplies many functions for dealing with LTI systems

```
[6]: import scipy.signal
```

We define an LTI system by passing the coefficients of the numerator and denominator to the `lti` constructor

```
[7]: numerator = K
      denominator = [tau, 1]
      G = scipy.signal.lti(numerator, denominator)
```

```
[8]: G
```

```
[8]: TransferFunctionContinuous(
      array([0.2]),
      array([1. , 0.2]),
      dt: None
      )
```

```
[9]: type(G)
```



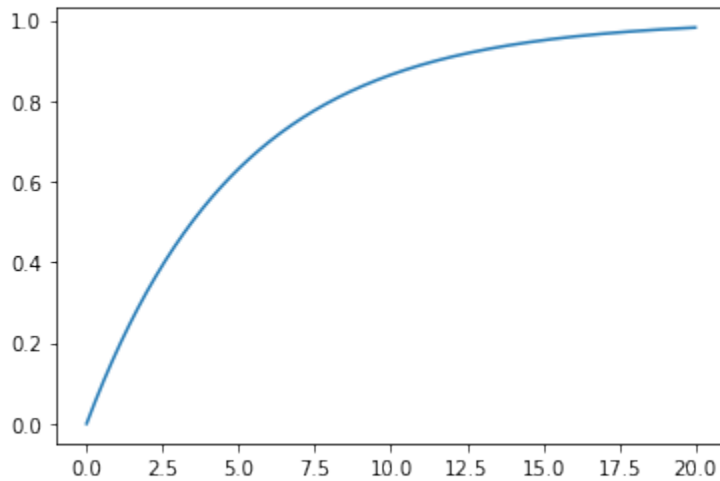
```
[9]: scipy.signal.lti.sys.TransferFunctionContinuous
```

Step responses

We can obtain the step response of the system by using the `step` method of the object

```
[10]: def plotstep(G):  
      _, ys_step = G.step(T=ts)  
      plt.plot(ts, ys_step);
```

```
[11]: plotstep(G)
```

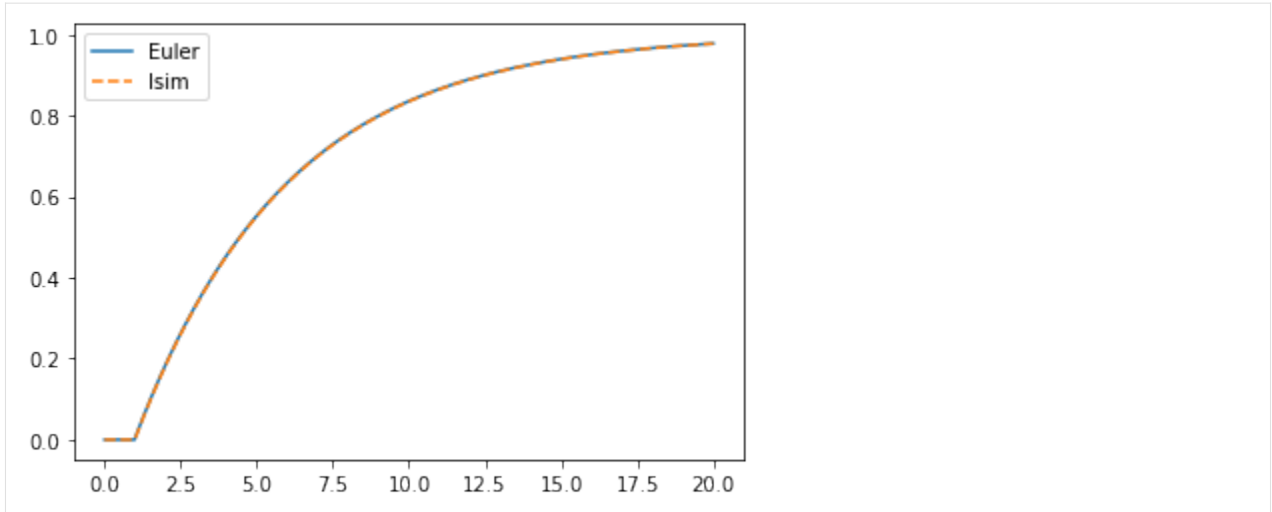


Responses to arbitrary inputs

We can also find the response of the system to an arbitrary input signal by using `scipy.signal.lsim()`. This is useful when simulating a complicated response or a response to an input read from a file.

```
[12]: us = [u(t) for t in ts] # evaluate the input function at all the times  
      _, ys_lsim, xs = scipy.signal.lsim(G, U=us, T=ts)  
      plt.plot(ts, ys)  
      plt.plot(ts, ys_lsim, '--');  
      plt.legend(['Euler', 'lsim'])
```

```
[12]: <matplotlib.legend.Legend at 0x231ea2517c0>
```

Manual integration using state space form

We can also use our Euler loop to simulate arbitrary systems using the state space representation

:nbsphinx-math: begin{align}

$$\dot{x} = Ax + Bu \quad y = Cx + Du$$

end{align}

This is a useful technique when simulating Hybrid systems (where some parts are LTI and some are nonlinear systems).

Luckily the `lti` object we created earlier can be converted to a state space representation.

```
[13]: Gss = G.to_ss()
      Gss

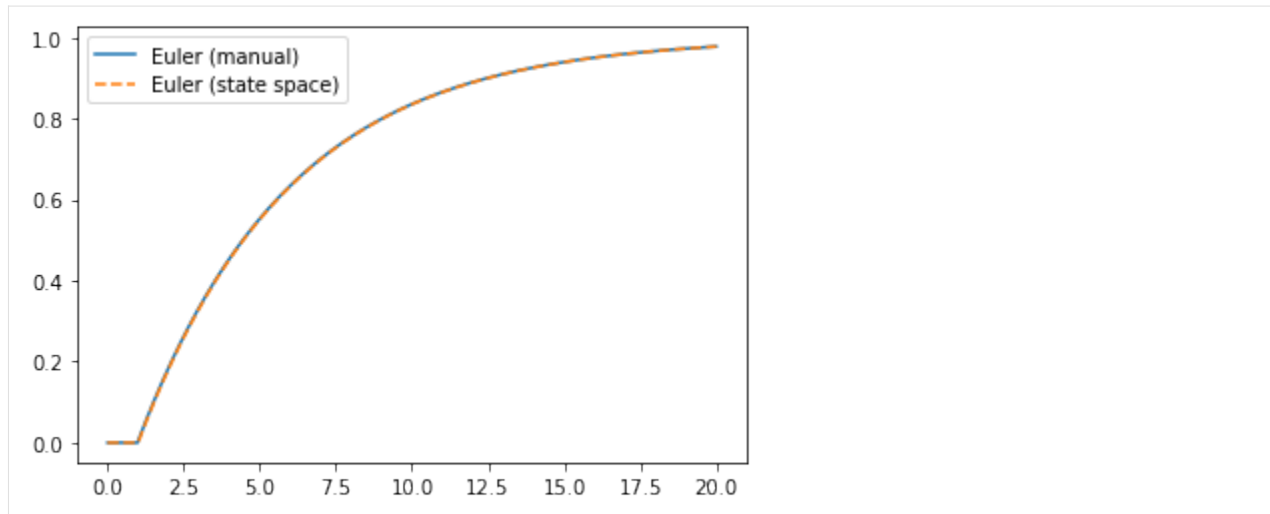
[13]: StateSpaceContinuous(
      array([[ -0.2]]),
      array([[ 1.]]),
      array([[ 0.2]]),
      array([[ 0.]]),
      dt: None
      )

[14]: x = numpy.zeros((Gss.A.shape[0], 1))
      ys_statespace = []
      for t in ts:
          xdot = Gss.A.dot(x) + Gss.B.dot(u(t))
          y = Gss.C.dot(x) + Gss.D.dot(u(t))

          x += xdot*dt
          ys_statespace.append(y[0,0])

[15]: plt.plot(ts, ys)
      plt.plot(ts, ys_statespace, '--')
      plt.legend(['Euler (manual)', 'Euler (state space)'])

[15]: <matplotlib.legend.Legend at 0x231ea2dfdf0>
```

Demonstration for higher order functions

As mentioned before, SymPy cannot always be used to obtain inverse Laplace transforms. The `scipy.signal` functions continue to work for higher order functions, too. Let's find the step response of the following transfer function:

$$G_2 = \frac{1}{s^3 + 2s^2 + s + 1}$$

Feel free to uncomment and run the block below: I gave up waiting for the inverse to be calculated.

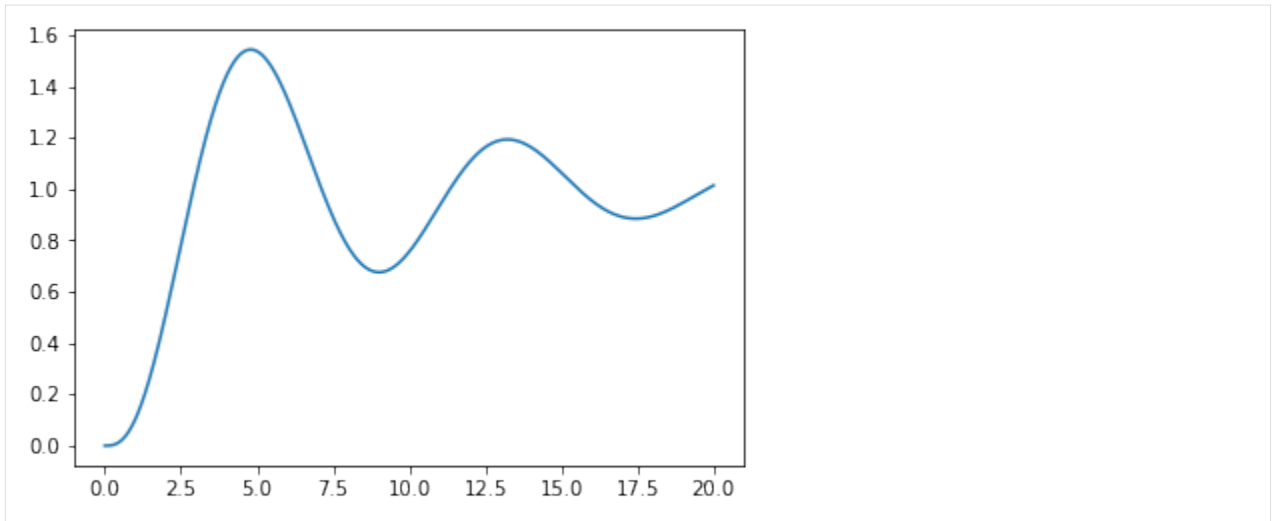
```
[16]: # import sympy

# s, t = sympy.symbols('s, t')
# G2 = 1/(s**3 + 2*s**2 + s + 1)
# r = sympy.inverse_laplace_transform(G2/s, s, t)
```

However, the step response is calculated quickly by the LTI object.

```
[17]: numerator = 1
denominator = [1, 2, 1, 1]
G2 = scipy.signal.lti(numerator, denominator)
```

```
[18]: plotstep(G2)
```

State space for higher order functions

```
[19]: Gss = G2.to_ss()
```

```
[20]: Gss
```

```
[20]: StateSpaceContinuous(
  array([[ -2.,  -1.,  -1.],
         [  1.,   0.,   0.],
         [  0.,   1.,   0.])),
  array([[1.],
         [0.],
         [0.])),
  array([[0., 0., 1.]]),
  array([[0.]]),
  dt: None
)
```

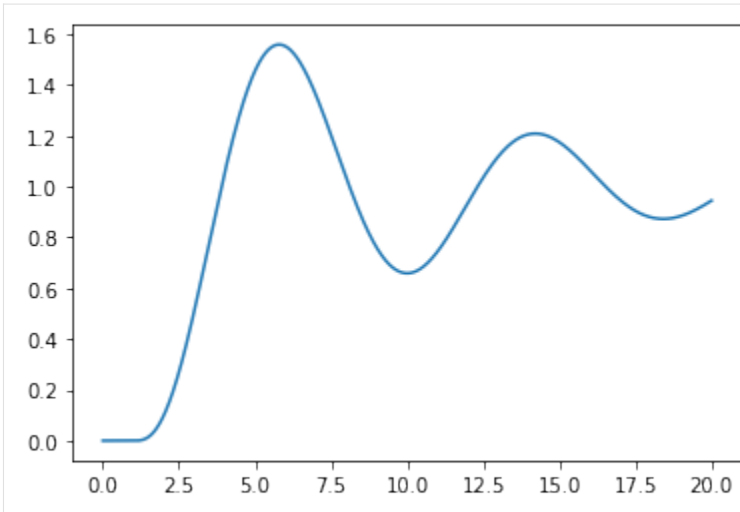
We can use the same code as before. Now, I'm going to store all the states as well. Notice that we have three states for the third order system.

```
[21]: x = numpy.zeros((Gss.A.shape[0], 1))
      ys_statespace = []
      xs = []
      for t in ts:
          xdot = Gss.A.dot(x) + Gss.B.dot(u(t))
          y = Gss.C.dot(x) + Gss.D.dot(u(t))

          x += xdot*dt
          ys_statespace.append(y[0, 0])
          # We need to copy otherwise the x update will overwrite all the values
          xs.append(x.copy())
```

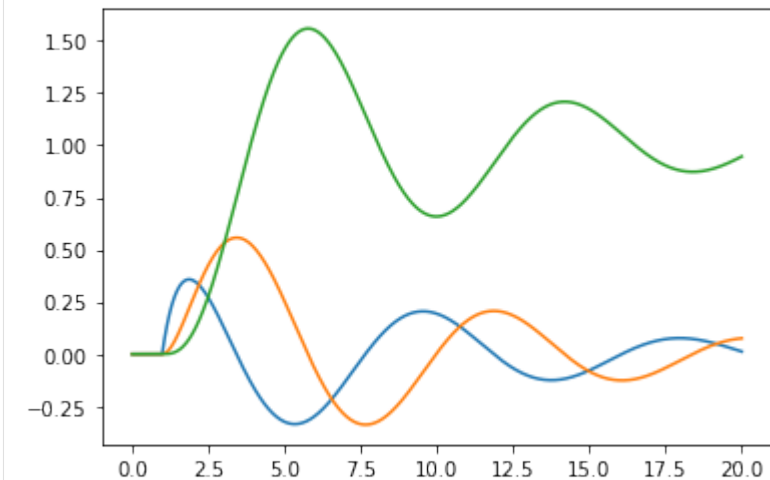
```
[22]: plt.plot(ts, ys_statespace)
```

```
[22]: [<matplotlib.lines.Line2D at 0x231eb3bad90>]
```

```
[23]: plt.plot(ts, numpy.concatenate(xs, axis=1).T)
```

```
[23]: [<matplotlib.lines.Line2D at 0x231ea3750a0>,  
<matplotlib.lines.Line2D at 0x231ea3750d0>,  
<matplotlib.lines.Line2D at 0x231ea375460>]
```



Systems in series

What if we wanted to see the response of $G_3(s) = G(s)G_2(s)$? You may expect that we would be able to find the product of two transfer functions, but the `scipy.signal` functions don't allow this.

```
[24]: G
```

```
[24]: TransferFunctionContinuous(  
array([0.2]),  
array([1. , 0.2]),  
dt: None  
)
```



```
[25]: # G*G2

# TypeError                                Traceback (most recent call last)
# <ipython-input-25-2585f2f9cba1> in <module>()
# ----> 1 G*G2

# TypeError: unsupported operand type(s) for *: 'TransferFunctionContinuous' and
# ↪ 'TransferFunctionContinuous'
```

Instead, we could use the convolution of the numerators and the denominators. This is equivalent to polynomial multiplication. For instance, let's work out $(s+1)^3$

```
[26]: numpy.convolve(numpy.convolve([1, 1], [1, 1]), [1, 1])
```

```
[26]: array([1, 3, 3, 1])
```

```
[27]: G.num
```

```
[27]: array([0.2])
```

```
[28]: numerator = numpy.convolve(G.num, G2.num)
      numerator
```

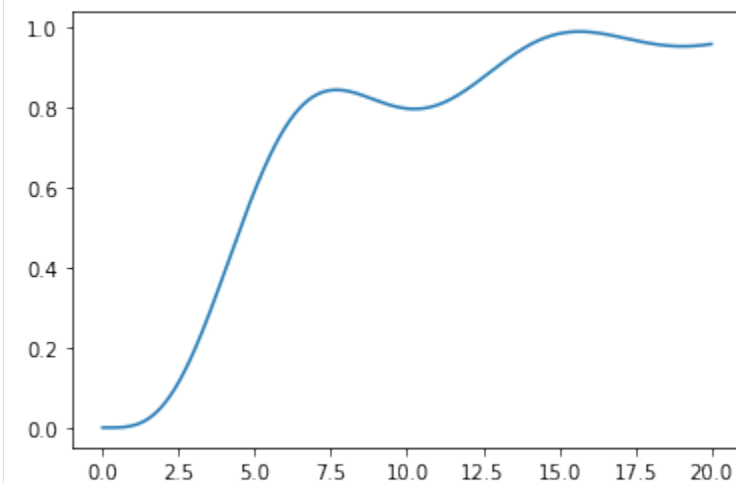
```
[28]: array([0.2])
```

```
[29]: denominator = numpy.convolve(G.den, G2.den)
      denominator
```

```
[29]: array([1. , 2.2, 1.4, 1.2, 0.2])
```

```
[30]: G3 = scipy.signal.lti(numerator, denominator)
```

```
[31]: plotstep(G3)
```



3. Control module

Another option for handling LTI systems is to use the [Python Control Systems Library](#). Unfortunately, this is not included in anaconda, so you will have to install it before use by uncommenting the line below and running it:

```
[32]: #!/pip install control
```

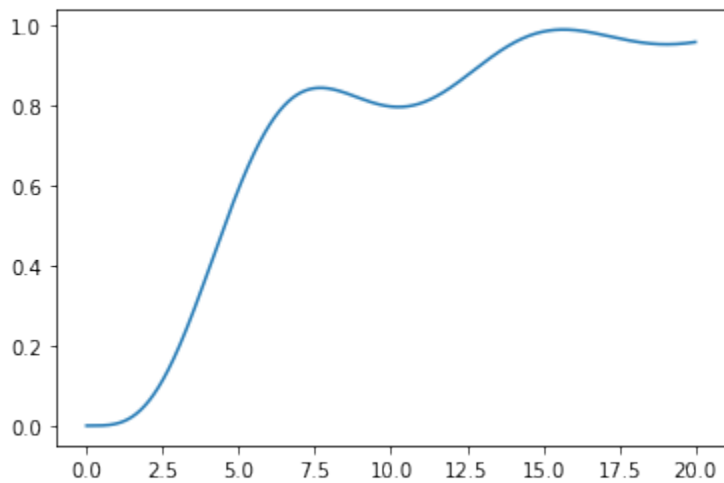
```
[33]: import control
```

A big benefit of this module is that its transfer function objects support arithmetic operations:

```
[34]: G = control.tf(K, [tau, 1])  
      G2 = control.tf(1, [1, 2, 1, 1])  
      G3 = G*G2
```

```
[35]: _, y = control.step_response(G3, ts)  
      plt.plot(ts, y)
```

```
[35]: [matplotlib.lines.Line2D at 0x231eb5af490>]
```

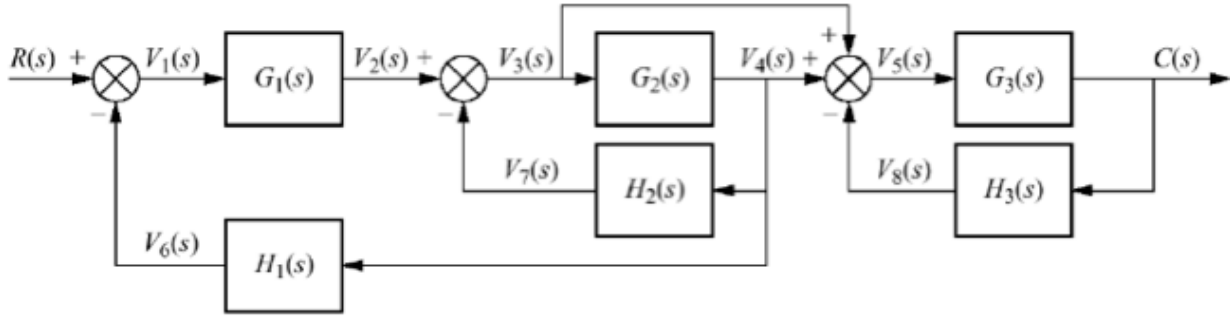


```
[ ]:
```


2.5.3 Simplifying block diagrams

These notes detail some techniques for reducing block diagrams graphically. In this notebook, I will solve the problem using SymPy.

Let's reduce this big block diagram to one input-output relationship (Example 4 in the notes linked to above):



```
[1]: import sympy
sympy.init_printing()
```

```
[2]: (R, V1, V2, V3, V4, V5, V6, V7, V8, C,
      G1, G2, G3, H1, H2, H3) = sympy.symbols('R, V1, V2, V3, V4, V5, V6, V7, V8, C, '
                                              'G1, G2, G3, H1, H2, H3')
unknowns = V1, V2, V3, V4, V5, V6, V7, V8, C
```

```
[3]: eqs = [# Blocks
            V2 - G1*V1,
            V4 - G2*V3,
            C - G3*V5,
            V6 - H1*V4,
            V7 - H2*V4,
            V8 - H3*C,
            # Sums
            V1 - (R - V6),
            V3 - (V2 - V7),
            V5 - (V4 + V3 - V8),
            ]
```

```
[4]: sol = sympy.solve(eqs, unknowns)
sol
```

```
[4]:
```

$$\left\{ C : \frac{G_1 G_3 R (G_2 + 1)}{G_1 G_2 G_3 H_1 H_3 + G_1 G_2 H_1 + G_2 G_3 H_2 H_3 + G_2 H_2 + G_3 H_3 + 1}, \quad V_1 : \frac{R (G_2 H_2 + 1)}{G_1 G_2 H_1 + G_2 H_2 + 1}, \quad V_2 : \frac{G_1 R (G_2 H_2 + 1)}{G_1 G_2 H_1 + G_2 H_2 + 1} \right.$$

The solution in the notes is factored:

```
[5]: sol[C].factor()
```

```
[5]:
```

$$\frac{G_1 G_3 R (G_2 + 1)}{(G_3 H_3 + 1) (G_1 G_2 H_1 + G_2 H_2 + 1)}$$

2.5.4 Approximation

There are many cases where we end up with very high order models or models with dead time which we would like to approximate with lower order models or models without dead time. This notebook illustrates some of the approaches.

```
[1]: import sympy
sympy.init_printing()
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[2]: import tbcontrol
```

```
[3]: tbcontrol.expectversion('0.1.7')
```

Taylor approximation

We have encountered Taylor approximants before. They are polynomial approximations and can easily be calculated using the `sympy.series` function.

```
[4]: x = sympy.symbols('x')
```

```
[5]: f = sympy.sin(x)
```

Note that Sympy uses the number of terms instead of the order of the polynomial, so this is a second order polynomial about the point $x=2$

```
[6]: f.series(x, 2, 3).removeO()
```

```
[6]: 
$$-\frac{(x-2)^2 \sin(2)}{2} + (x-2) \cos(2) + \sin(2)$$

```

Let's plot a couple of approximations of $\sin(x)$:

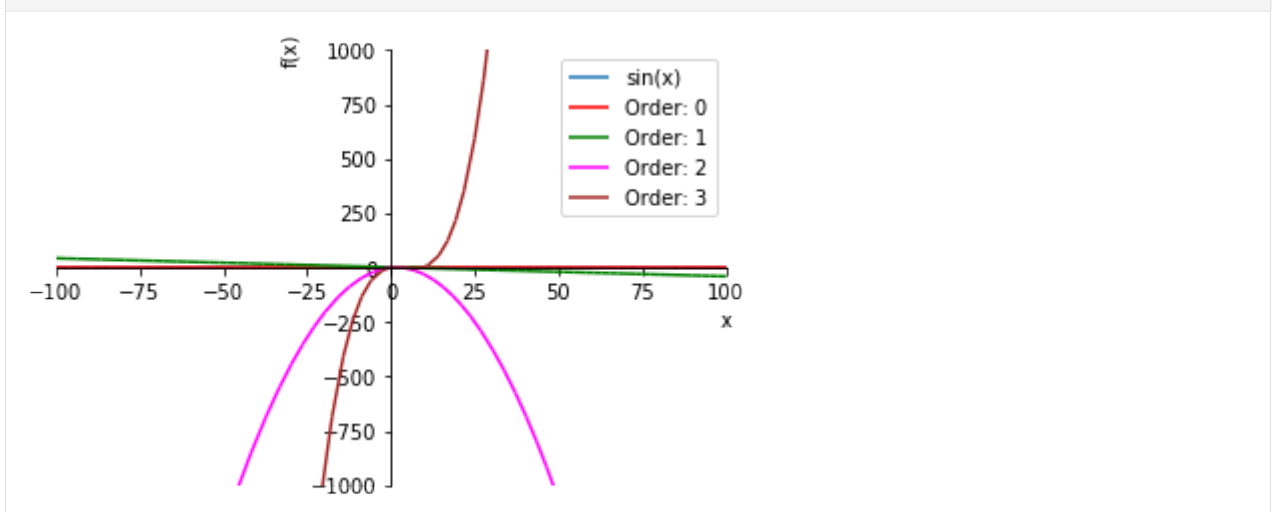
```
[7]: def taylor(xlim, ylim):
    p = sympy.plot(f, (x, *xlim), show=False)
    colors = ['red', 'green', 'magenta', 'brown']
    for n, color in enumerate(colors, 1):
        approx = f.series(x, 2, n).removeO()
        p.extend(sympy.plot(approx, (x, *xlim),
                             line_color=color, show=False))
        p[n].label = f'Order: {n-1}'
    p.ylim = ylim
    p.xlim = xlim
    p.legend = True
    p.show()
```

```
[8]: taylor((-10, 10), (-4, 4))
```




An important characteristic of all polynomial approximations is that the function always grows large “far enough” away from the origin and therefore cannot model asymptotes very well. Let’s zoom out on that graph a bit:

```
[9]: taylor((-100, 100), (-1000, 1000))
```



Padé approximation

Padé approximation is an extension of the concept of Taylor approximation with rational functions rather than polynomials. The basic premise is that the approximation is made by matching derivatives at a particular point. Padé approximants are often associated with dead time approximation, but can be used for arbitrary transfer functions as well.

One of the big benefits of Padé approximants is that rational functions can become constant for large magnitudes of x .

We will approximate a Laplace dead time

```
[10]: s = sympy.symbols('s')
```

```
[11]: G = sympy.exp(-2*s)
G
```



```
[11]:  $e^{-2s}$ 
```

by a 1/1 Padé approximation. This means first order above the line and first order below. In order to force uniqueness of the solution, we force the constant term in the denominator to be unity.

```
[12]: import tbcontrol.symbolic
```

```
[13]: s0 = 0
```

```
[14]: G_pade = tbcontrol.symbolic.pade(G, s, 1, 1, s0)
      G_pade
```

```
[14]:  $\frac{1-s}{s+1}$ 
```

Compare this with a Taylor approximation with same number of coefficients (matching the same number of derivatives)

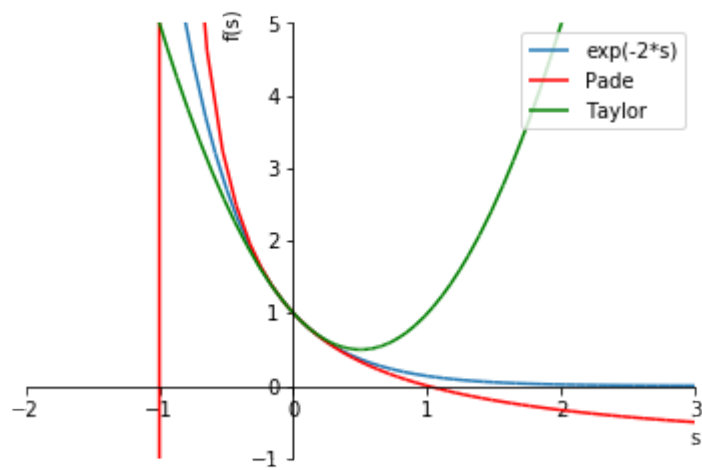
```
[15]: G_taylor = G.series(s, s0, 3).removeO()
      G_taylor
```

```
[15]:  $2s^2 - 2s + 1$ 
```

So how much do the approximations resemble the original function?

First, let's check just the real part

```
[16]: plotrange = (s, -2, 3)
      def plot_approx(G, G_pade, G_taylor):
          p = sympy.plot(G, plotrange, show=False)
          pade_approx = sympy.plot(G_pade, plotrange, show=False, line_color='red')
          taylor_approx = sympy.plot(G_taylor, plotrange, show=False, line_color='green')
          p.extend(pade_approx)
          p.extend(taylor_approx)
          p[1].label = 'Pade'
          p[2].label = 'Taylor'
          p.ylim = (-1, 5)
          p.legend = True
          p.show()
      plot_approx(G, G_pade, G_taylor)
```



Note the singularity in the Padé approximation, as well as the fact that the Taylor polynomial has an unbounded error to the right, while the Padé approximation is better behaved.

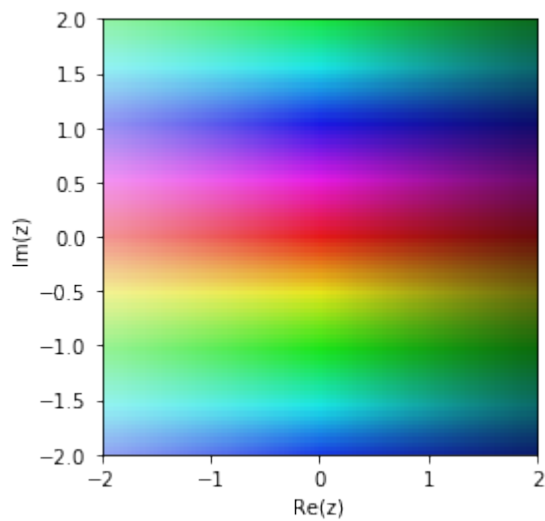
Now, let's see what this looks like for the whole complex plane

```
[17]: try:
      import mpmath
    except ImportError:
      from sympy import mpmath
```

```
[18]: def cplot(G):
      f = sympy.lambdify(s, G, ['mpmath', 'numpy'])
      mpmath.cplot(f, [-2, 2], [-2, 2], points=10000)
```

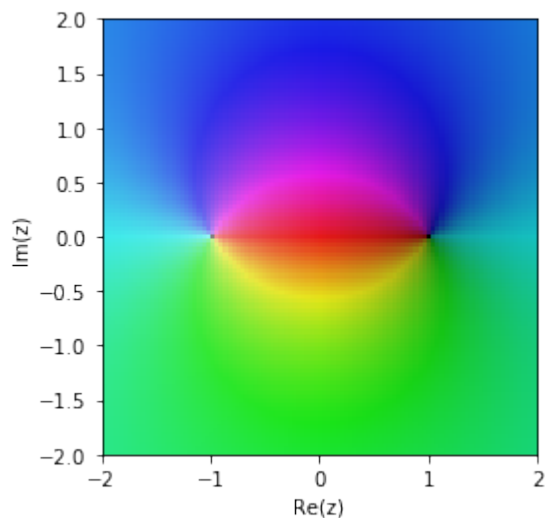
The original function

```
[19]: cplot(G)
```



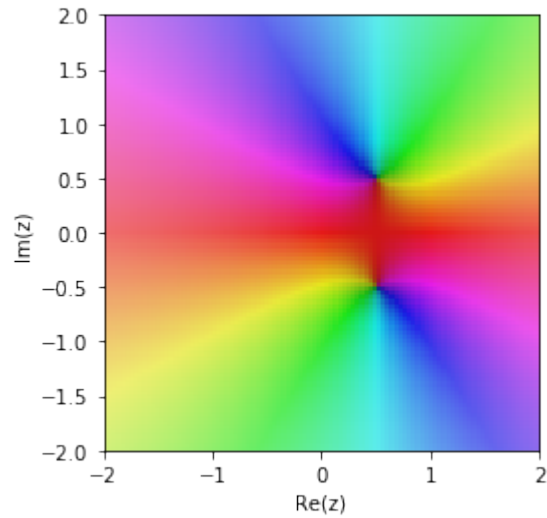
Padé approximation

```
[20]: cplot(G_pade)
```



Taylor approximation

```
[21]: cplot(G_taylor)
```



The Padé approximation is much better in the region around 0.

Further exploration

Padé approximations with order 0 below the line are effectively Taylor polynomials

```
[22]: tbcontrol.symbolic.pade(G, s, 1, 0, 0)
```

```
[22]: 1 - 2s
```

This form is often used the other way around to approximate lags with dead time

```
[23]: tbcontrol.symbolic.pade(G, s, 0, 1, 0)
```

```
[23]: 1
      2s + 1
```

```
[24]: def approx_comparison(G, M, N):
      P = tbcontrol.symbolic.pade(G, s, M, N, 0)
      T = sympy.series(G, s, 0, N+M+1).removeO()
      plot_approx(G, P, T)
```

```
[25]: from ipywidgets import interact
```

```
[26]: deadtime = sympy.exp(-2*s)
      high_order = 1/(s + 1)**10
```

```
[27]: plorange=(s, -5, 5)
```

```
[28]: interact(approx_comparison, G=[deadtime, high_order], N=(0, 3), M=(1, 3))
      interactive(children=(Dropdown(description='G', options=(exp(-2*s), (s + 1)**(-10)),
      ↪value=exp(-2*s)), IntSlid...
```



```
[28]: <function __main__.approx_comparison(G, M, N)>
```

Approximations based on response matching

The approximations we discussed above are based on matching the values in the Laplace domain. However, we often want to find an approximation which has the property of matching the time domain responses.

A common-sense rule is that larger time constants are more important to retain than smaller ones. My personal rule is that any time constant which is less than 10 times smaller than the next largest one can usually be ignored, in other words, for our purposes

$$\frac{1}{(10s + 1)(s + 1)} \approx \frac{1}{10s + 1}$$

Note 1 It is conventional to arrange the terms in descending orders of time constants.

Note 2 This is a rule of thumb and should not be applied during intermediate calculations. You should always be aware of the point where you are applying approximation and make a note that you have done this.

In this section I'll be using the [Python Control Systems Library](#). It doesn't support dead time in its transfer function object, but I'll fake it in the responses by shifting them with a certain dead time. We assume version 0.8.0.

```
[29]: import control
```

```
[30]: control.__version__
```

```
[30]: '0.8.2'
```

I like defining s like this to make formulae easier to type later on. Note that this overwrites our earlier symbolic s , so after this definition we can no longer use s in sympy.

```
[31]: s = control.tf([1, 0], 1)
```

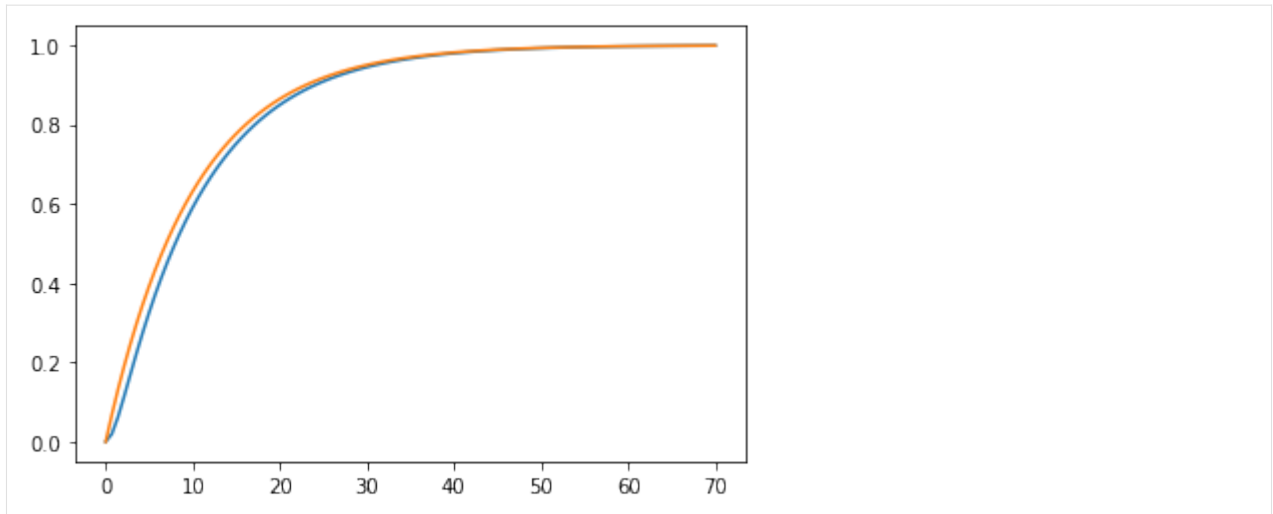
We'll be plotting lots of step responses for delayed transfer functions. This function will “fake” this by calculating the undelayed response and plotting it shifted up by the delay.

```
[ ]:
```

```
[32]: def plotstep(G, D=0, T=None):
      t, y = control.step_response(G, T=T)
      new_t = numpy.concatenate([[0], t + D])
      new_y = numpy.concatenate([[0], y])
      plt.plot(new_t, new_y)
```

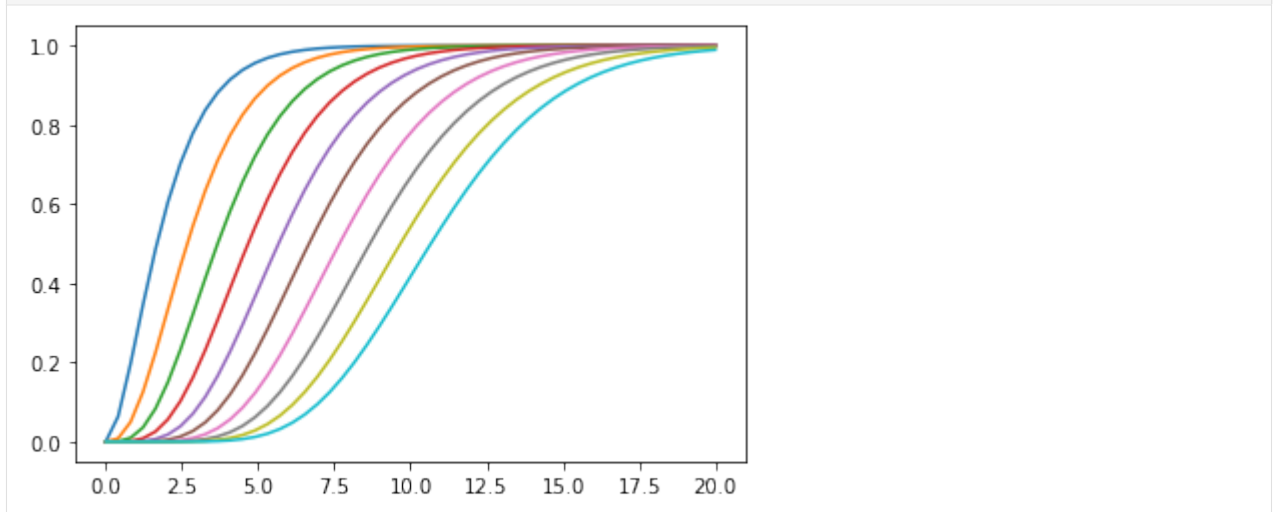
```
[33]: G1 = 1/((s + 1)*(10*s + 1))
      G2 = 1/((10*s + 1))
```

```
[34]: plotstep(G1)
      plotstep(G2)
```

First order systems in series often have step responses which resemble those of lower order systems with increasing dead time:

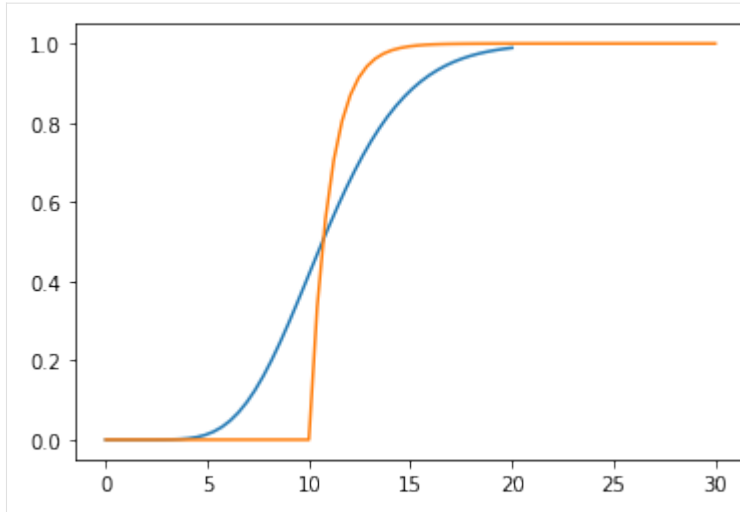
```
[35]: ts = numpy.linspace(0, 20)
G = 1/(s + 1)
for i in range(10):
    G *= 1/(s + 1)
    plotstep(G, T=ts)
```



If we use the 0, 1 Padé approximation of dead time:

$$e^{-\theta s} \approx \frac{1}{1 + \theta s}$$

```
[36]: plotstep(G, T=ts)
plotstep(1/(s + 1), D=10, T=ts)
```

We see that we get the same kind of behaviour but the dynamics start too fast and end too slow.

We can “eyeball” a lower-order response which matches the last 10th order response pretty well. Play with these sliders and see how easy it is to match the lines together.

```
[37]: def approx(tau, D):
      plotstep(G, T=numpy.linspace(0, 30))
      plotstep(1/(tau*s + 1), D=D, T=numpy.linspace(0, 30 - D))
      plt.show()
```

```
[38]: interact(approx, tau=(1., 10.), D=(1., 10.))

interactive(children=(FloatSlider(value=5.5, description='tau', max=10.0, min=1.0),
FloatSlider(value=5.5, des...
```

```
[38]: <function __main__.approx(tau, D)>
```

Skogestad’s “Half Rule”

Skogestad’s “half rule” is specifically designed to approximate complicated transfer functions as first order plus dead time (FOPDT) or second order plus dead time (SOPDT) models:

$$\text{FOPDT: } \frac{K e^{-\theta s}}{\tau s + 1} \quad \text{SOPDT: } \frac{K e^{-\theta s}}{(\tau_1 s + 1)(\tau_2 s + 1)}$$

The method does not work for systems with complex roots or unstable systems.

The function `tbcontrol.numeric.skogestad_half` implements this method.

For instance, let’s take the transfer function from Example 5.4:

$$G(s) = \frac{K(-0.1s + 1)}{(5s + 1)(3s + 1)(0.5s + 1)}$$

The gains are always matched, so we can safely use $K = 1$

```
[39]: K = 1
```



```
[40]: G = K*(-0.1*s + 1)/((5*s + 1)*(3*s + 1)*(0.5*s + 1))

[41]: from tbcontrol.numeric import skogestad_half

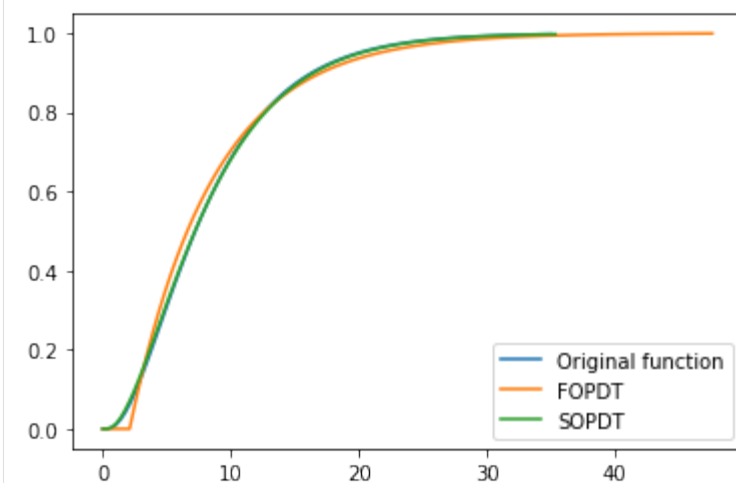
[42]: theta1, [tau] = skogestad_half([-0.1], [5, 3, 0.5], delay=0, order=1)
      Gapprox1 = K/(tau*s + 1)

[43]: theta2, [tau1, tau2] = skogestad_half([-0.1], [5, 3, 0.5], delay=0, order=2)
      Gapprox2 = K/((tau1*s + 1)*(tau2*s + 1))
```

Let's see what our final approximations look like:

```
[44]: plotstep(G)
      plotstep(Gapprox1, D=theta1)
      plotstep(Gapprox2, D=theta2)
      plt.legend([
          'Original function',
          'FOPDT',
          'SOPDT'])

[44]: <matplotlib.legend.Legend at 0x1228c6dd8>
```



Not bad!

2.6 Multivariable system representations

2.6.1 Transfer function matrices

Let's say we have two inputs and two outputs. We can write the linearised effect as follows:

$$y_1 = G_{11}u_1 + G_{12}u_2 \quad (2.21)$$

$$y_2 = G_{21}u_1 + G_{22}u_2 \quad (2.22)$$

Which is equivalent to a matrix expression

$$\mathbf{y} = \mathbf{G}\mathbf{u}$$

with

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad G = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

I find it useful to picture the input going into the top of the matrix and the output coming out of the side

Representing matrices in SymPy

```
[1]: import sympy
      sympy.init_printing()
```

```
[2]: s = sympy.Symbol('s')
```

```
[3]: G11 = (s + 1)/(s + 2)
      G12 = 1/(2*s + 1)
      G21 = 1/(3*s + 1)
      G22 = 1/(4*s + 1)

      row1 = [G11, G12]
      row2 = [G21, G22]
      list_of_lists = [row1,
                        row2]
```

```
[4]: G = sympy.Matrix(list_of_lists)
```

```
[5]: G[0, 0]
```

```
[5]: 
$$\frac{s + 1}{s + 2}$$

```

```
[6]: sympy.simplify(G.inv())
```

```
[6]: 
$$\begin{bmatrix} \frac{6s^3+17s^2+11s+2}{6s^3+7s^2-3s-1} & -\frac{12s^3+31s^2+15s+2}{6s^3+7s^2-3s-1} \\ -\frac{8s^3+22s^2+13s+2}{6s^3+7s^2-3s-1} & \frac{24s^4+50s^3+35s^2+10s+1}{6s^3+7s^2-3s-1} \end{bmatrix}$$

```

```
[7]: G + G
```

```
[7]: 
$$\begin{bmatrix} \frac{2(s+1)}{s+2} & \frac{2}{2s+1} \\ \frac{2}{3s+1} & \frac{2}{4s+1} \end{bmatrix}$$

```

Representing matrices using the control library

```
[8]: import control
```

```
[9]: control.tf([0, 1], [1, 2])
```

```
[9]: 
$$\frac{1}{s + 2}$$

```



```
[10]: s = control.tf([1, 0], 1)
```

```
[11]: s
```

```
[11]: 
$$\frac{s}{1}$$

```

```
[12]: G = 1/(s + 1)
```

```
[13]: num11 = [1, 2]
      num12 = [2]
      num21 = [3]
      num22 = [4]

      row1 = [num11, num12]
      row2 = [num21, num22]
      numerator = [row1,
                   row2]

      denominator = [[[1, 1], [2, 1]],
                     [[3, 1], [4, 1]]]
      Gmatrix = control.tf(numerator,
                           denominator)
```

```
[14]: Gmatrix
```

```
[14]: 
$$\begin{bmatrix} \frac{s+2}{s+1} & \frac{2}{2s+1} \\ \frac{3}{3s+1} & \frac{4}{4s+1} \end{bmatrix}$$

```

2.6.2 Conversion to state space

See the [state space notebook](#) for more information about conversion between state space and transfer function form. The examples in that notebook are for SISO transfer functions.

There are no tools in `scipy.signal` to deal with multivariable transfer functions. However, the control library can do the conversion from a transfer function matrix to a state space form if you have the `'slycot <>'` library installed.

You can try to install slycot using this command:

```
[15]: #!conda install -c conda-forge control slycot
```

```
[16]: control.ss(Gmatrix)
```

```
[16]: 
$$\begin{pmatrix} -1.33 & -1.11 \cdot 10^{-16} & -1.67 \cdot 10^{-16} & -0.333 & 1 & -1.11 \cdot 10^{-16} \\ -3.33 \cdot 10^{-16} & -0.75 & -0.125 & 6.94 \cdot 10^{-17} & 0 & 1 \\ 1.11 \cdot 10^{-16} & 1 & -2.1 \cdot 10^{-16} & -3.41 \cdot 10^{-16} & 0 & 0 \\ 1 & 0 & 1.31 \cdot 10^{-16} & -9.52 \cdot 10^{-17} & 0 & 0 \\ \hline 1 & 1 & 0.25 & 0.333 & 1 & 0 \\ 1 & 1 & 0.5 & 1 & 0 & 0 \end{pmatrix}$$

```

What are the true values of those small (10^{-16}) values?

2.6.3 State space representation

The “standard” or most commonly used state space representation is

$$\dot{x} = Ax + Bu \quad (2.23)$$

$$y = Cx + Du \quad (2.24)$$

Take note that Seborg uses a slightly different version:

$$\dot{x} = Ax + Bu + Ed \quad (2.25)$$

$$y = Cx \quad (2.26)$$

This second version can not represent pure gain systems as it effectively assumes $D = 0$. It is also possible to stack u and d from the bottom form into one input vector, so the E matrix really doesn’t add much. As you may infer, I prefer the top version and it is also the version used by most libraries.

```
[1]: import numpy
import numpy.linalg
```

Converting between state space and transfer function forms

There is good support in various libraries for converting systems with numeric coefficients between transfer function and state space representation.

Scipy.signal

The `scipy.signal` library handles conversion between transfer function coefficients and state space matrices easily. Note that `scipy.signal` only handles SISO transfer functions.

```
[2]: import scipy.signal
```

```
[3]: G = scipy.signal.lti(1, [1, 1])
```

```
[4]: G
```

```
[4]: TransferFunctionContinuous(
array([1.]),
array([1., 1.]),
dt: None
)
```

This object allows us to access the numerator and denominator

```
[5]: G.num, G.den
```

```
[5]: (array([1.]), array([1., 1.]))
```

To convert to state space, we can use the `.to_ss()` method


```
[6]: Gss = G.to_ss()
```

```
[7]: Gss.A, Gss.B, Gss.C, Gss.D
```

```
[7]: (array([[ -1.]]), array([[ 1.]]), array([[ 1.]]), array([[ 0.]])
```

We can build another object using the state space matrices instead of the Laplace form

```
[8]: G2ss = scipy.signal.lti(Gss.A, Gss.B, Gss.C, Gss.D)
G2ss
```

```
[8]: StateSpaceContinuous(
array([[ -1.]]),
array([[ 1.]]),
array([[ 1.]]),
array([[ 0.]])
dt: None
)
```

We can convert to transfer function form using `.to_tf()` (there is a small warning about bad coefficients, but the answer is reliable).

```
[9]: G2 = G2ss.to_tf()
```

```
C:\Users\Admin\anaconda3\lib\site-packages\scipy\signal\filter_design.py:1630:
↳BadCoefficients: Badly conditioned filter coefficients (numerator): the results may
↳be meaningless
warnings.warn("Badly conditioned filter coefficients (numerator): the "
```

We can now access the numerator and denominator again:

```
[10]: G2.num, G2.den
```

```
[10]: (array([1.]), array([1., 1.]))
```

Instead of building objects we can also use the functions in `scipy.signal.lti_conversion`:

```
[11]: scipy.signal.lti_conversion.tf2ss(1, [1, 1])
```

```
[11]: (array([[ -1.]]), array([[ 1.]]), array([[ 1.]]), array([[ 0.]])
```

```
[12]: scipy.signal.lti_conversion.ss2tf(-1, 1, 1, 0)
```

```
[12]: (array([[ 0., 1.]]), array([1., 1.]))
```

Control library

The control library (at least from version 0.8.0) does a good job with these conversions as well.

```
[13]: import control
```

```
[14]: Gtf = control.tf([1], [1, 1])
Gtf
```

```
[14]: 
$$\frac{1}{s + 1}$$

```


In the control library we convert the system using `ss` (short for state space) to get a State Space representation:

```
[15]: Gss = control.ss(Gtf)
      Gss
```

```
[15]:
```

$$\left(\begin{array}{c|c} -1 & 1 \\ \hline 1 & 0 \end{array} \right)$$

```
[16]: Gss.A
```

```
[16]: array([[ -1.]])
```

Symbolic conversion

It is easy to convert state space models to transfer functions since the Laplace transform is a linear operator:

$$\begin{aligned} \dot{x} &= Ax + Bu \quad \therefore \quad sX(s) = AX(s) + BU(s) \quad X(s) = (sI - A)^{-1}BU(s) \\ y &= Cx + Du \quad \therefore \quad Y(s) = CX(s) + DU(s) \quad Y(s) = \underbrace{(C(sI - A)^{-1}B + D)}_{G(s)}U(s) \end{aligned}$$

This conversion is handled for symbolic matrices by `tbcontrol.symbolic.ss2tf`

```
[17]: import sympy
```

```
[18]: import tbcontrol
      tbcontrol.expectversion('0.1.8')
      import tbcontrol.symbolic
```

```
[19]: s = sympy.symbols('s')
```

```
[20]: A, B, C, D = [sympy.Matrix(m) for m in [G2ss.A, G2ss.B, G2ss.C, G2ss.D]]
```

```
[21]: A, B, C, D
```

```
[21]: (Matrix([[ -1.0]]), Matrix([[1.0]]), Matrix([[1.0]]), Matrix([[0.0]]))
```

```
[22]: G = tbcontrol.symbolic.ss2tf(A, B, C, D, s)
      G
```

```
[22]:  $\begin{bmatrix} \frac{1.0}{s+1.0} \end{bmatrix}$ 
```

Note that `ss2tf` returns a sympy Matrix. To get the SISO result, we need to index into the matrix:

```
[23]: G[0, 0]
```

```
[23]:  $\frac{1.0}{s+1.0}$ 
```


Analysis

Notice that the roots of the characteristic function correspond with the eigenvalues of the A matrix. The numerator and denominator of control transfer functions are stored as lists of lists to accomodate MIMO systems.

```
[24]: Gtf.pole()
[24]: array([-1.])

[25]: numpy.roots(Gtf.den[0][0])
[25]: array([-1.])

[26]: numpy.linalg.eig(Gss.A)
[26]: (array([-1.]), array([[1.]])
```

2.7 System identification

```
[1]: import numpy
    from matplotlib import pyplot as plt
    %matplotlib inline
```

2.7.1 Linear regression

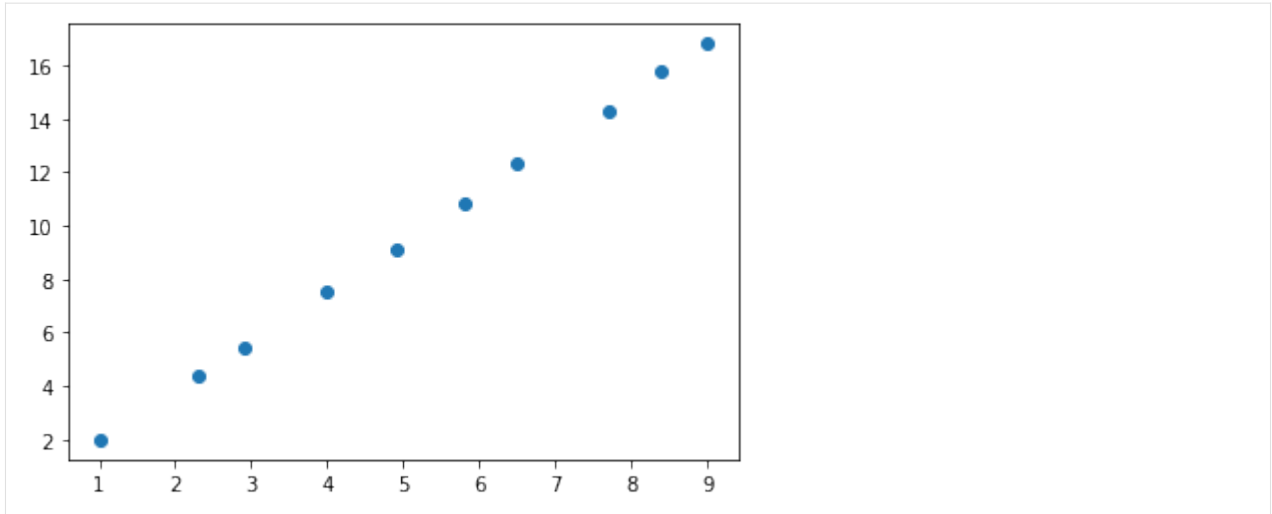
Data from Example 6.1 of Seborg, Edgar, Melichamp and Doyle (3rd edition)

```
[2]: import pandas

[3]: df = pandas.read_excel('../assets/example_6_1.xlsx')

[4]: x = df['Fuel Flow Rate']
    y = df['Power Generated']

[5]: plt.scatter(x, y)
[5]: <matplotlib.collections.PathCollection at 0x11b005588>
```

That resembles a straight line! Let's say the line is described by

$$y = ax + b$$

In regression, we are trying to find a and b given lots of values for y and x , so we have something like

$$y_0 = ax_0 + b \quad (2.27)$$

$$y_1 = ax_1 + b \quad (2.28)$$

$$y_2 = ax_2 + b \quad (2.29)$$

$$\vdots = \vdots \quad (2.30)$$

$$(2.31)$$

Which we factor as

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \end{bmatrix}}_Y = \underbrace{\begin{bmatrix} x_0 & 1 \\ x_1 & 1 \\ \vdots & \vdots \end{bmatrix}}_X \underbrace{\begin{bmatrix} a \\ b \end{bmatrix}}_{\beta}$$

In this case, when there are many points, X is taller than it is wide and as such we know that there is no solution to the exact equations. Instead, we may try to get “close” to the solution. We can write the residuals as

$$\epsilon = Y - X\beta$$

Here, ϵ is a vector of errors, one for each data point. The (euclidian) length of this vector is given by

$$\|\epsilon\| = \sqrt{\sum_i \epsilon_i^2}$$

It is common to focus on minimising the part without the square root to make calculations simpler. This leads to the popular way of determining the error of a fit called the “sum of square errors”, sometimes expressed as $\|\epsilon\|^2$.

This minimisation is written in mathematical notation as

$$\min_{\beta} \sum_i \epsilon_i^2$$

which is read in words as as “minimise (*with respect to or by changing* β) the sum of the square error”.

2.7.2 Create the design matrices

The matrices Y and X are sometimes called the design matrices. We can build them using basic numpy functions as follows:

```
[6]: Y = numpy.asmatrix(y).T
     X = numpy.bmat([numpy.c_[x], numpy.ones_like(Y)])
```

Note `numpy.c_` produces a two dimensional array from a one dimensional one in a column.

In the case of polynomials, X is a special matrix called a **Vandermonde matrix**, and numpy has a function which generates them more easily.

```
[7]: X = numpy.asmatrix(numpy.vander(x, 2))
```

There is also a library called **patsy** which supplies a simplified syntax to construct these matrices:

```
[8]: import patsy
```

```
[9]: Y, X = patsy.dmatrices("Q('Power Generated') ~ Q('Fuel Flow Rate')", df)
```

```
[10]: Y, X = map(numpy.asmatrix, (Y, X))
```

Pseudoinverse solution

First, let's apply the pseudoinverse method directly (note **you should never do this for production code**, as calculating inverses is computationally expensive)

The solution minimising the sum of the squares of the residual $\|\epsilon\|^2$ can be shown to be

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

```
[11]: #Excel: =MMULT((MINVERSE(MMULT(TRANSPOSE(_X); _X))); MMULT(TRANSPOSE(_X); _Y))
     betahat = (X.T * X).I * X.T * Y
     betahat

[11]: matrix([[0.07854918],
             [1.85932397]])
```

Note: The code above is as close as possible to the equation above, as I have made X and Y matrices. **The numpy developers advise against using the `numpy.matrix` class.**

Matrix properties: `*`: Transpose `.I`: Inverse `.A`: Array form of matrix

Normal `numpy.array`s don't have an `.I` property and don't multiply matrix-fashion but rather element-wise. Here is how we would have to write the code if we used arrays:

```
[12]: Y = Y.A
     X = X.A

[13]: numpy.linalg.inv(X.T @ X) @ X.T @ Y

[13]: array([[0.07854918],
            [1.85932397]])
```

The `@` operator always does matrix multiplication and expects two-dimensional arrays on both sides.

Dedicated solvers

Calculating the inverse is not a numerically well behaved operation, so you should rather use a dedicated routine if you are solving this kind of problem:

```
[14]: beta, residuals, rank, s = numpy.linalg.lstsq(X, Y, rcond=None)
      beta
[14]: array([[0.07854918],
            [1.85932397]])
```

However, polynomial fits are such a common operation that there are nicer routines to do this fit. There are a whole range of functions in numpy which start with poly. Press tab to see them.

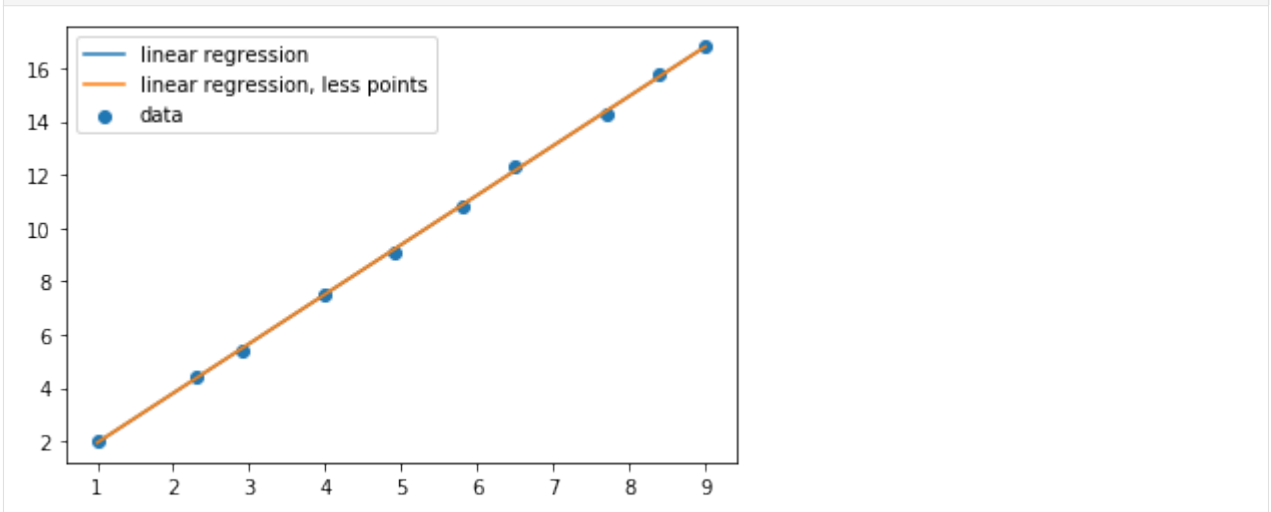
```
[15]: numpy.poly
[15]: <function numpy.poly(seq_of_zeros)>
```

```
[16]: poly = numpy.polyfit(x, y, 1)
      poly
[16]: array([1.85932397, 0.07854918])
```

Notice that we could just pass the data directly, and the routine handled building the various matrices and the fitting itself. It is useful to plot the regression with the data points, but we should sample on a finer grid.

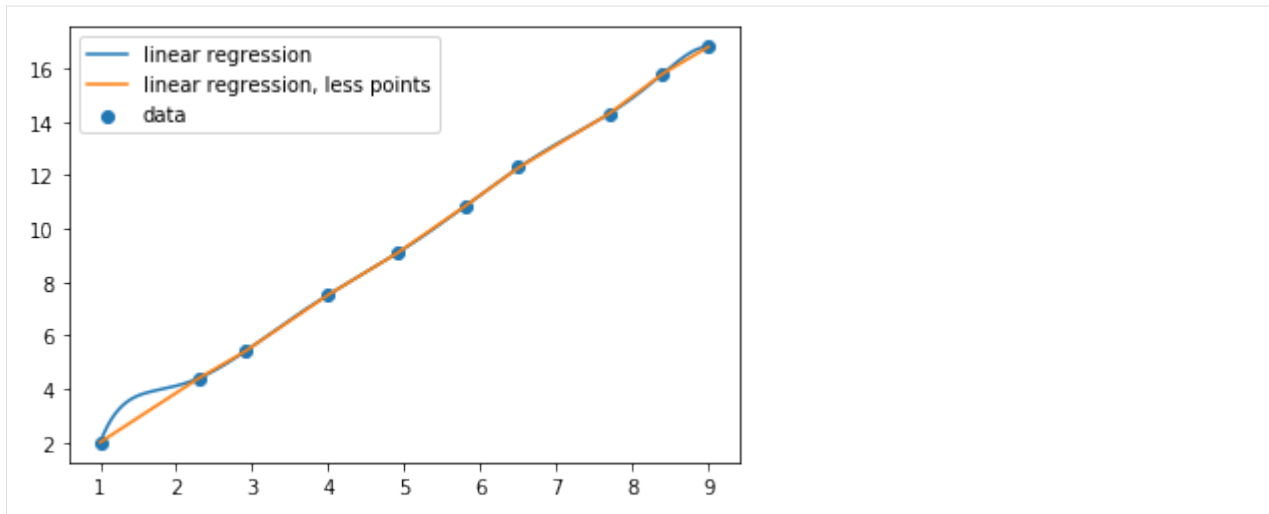
```
[17]: smoothx = numpy.linspace(min(x), max(x), 1000)

[18]: def regplot(poly):
      smoothy = numpy.polyval(poly, smoothx)
      plt.scatter(x, y, label='data')
      plt.plot(smoothx, smoothy, label='linear regression')
      plt.plot(x, numpy.polyval(poly, x), label='linear regression, less points')
      plt.legend(loc='best')
      regplot(poly)
```



There's obviously no difference between the two for a linear fit, but what about higher orders?

```
[19]: poly9 = numpy.polyfit(x, y, 8)
      regplot(poly9)
```

If we had just plotted the connecting lines, we would have missed the bit sticking up on the left!

2.7.3 Nonlinear regression

We can apply the same principles to fit nonlinear functions as well. The `scipy.optimize.curve_fit` function can be used to fit an arbitrary function to data

```
[20]: import scipy.optimize
```

Let's start by reproducing the results from the linear fit

```
[21]: def f(x, a, b):
      """fitting function linear in coefficient"""
      return a*x + b
```

```
[22]: beta, _ = scipy.optimize.curve_fit(f, x, y, [1, 0])
      beta
```

```
[22]: array([1.85932396, 0.07854919])
```

Now let's build some data which is obviously nonlinear

```
[23]: x = numpy.arange(1, 10)
      y = 2*numpy.sin(3*x) + x + 1
```

```
[24]: def f2(x, a, b, c, d):
      """ A nonlinear fitting function """
      return a*numpy.sin(b*x) + c*x + d
```

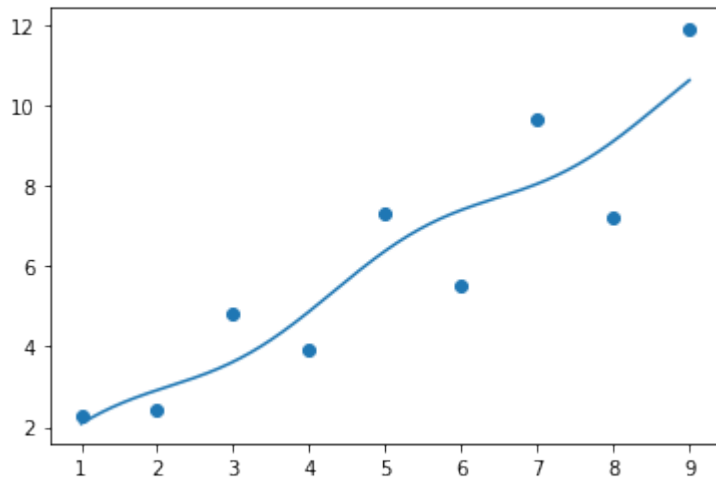
```
[25]: def fit_and_plot(beta0):
      beta, _ = scipy.optimize.curve_fit(f2, x, y, beta0)

      plt.scatter(x, y)
      plt.plot(smoothx, f2(smoothx, *beta))
      return beta
```



```
[26]: fit_and_plot([1, 1, 1, 1])
```

```
[26]: array([0.32658273, 1.45786791, 1.0854963 , 0.6763682 ])
```



What went wrong?

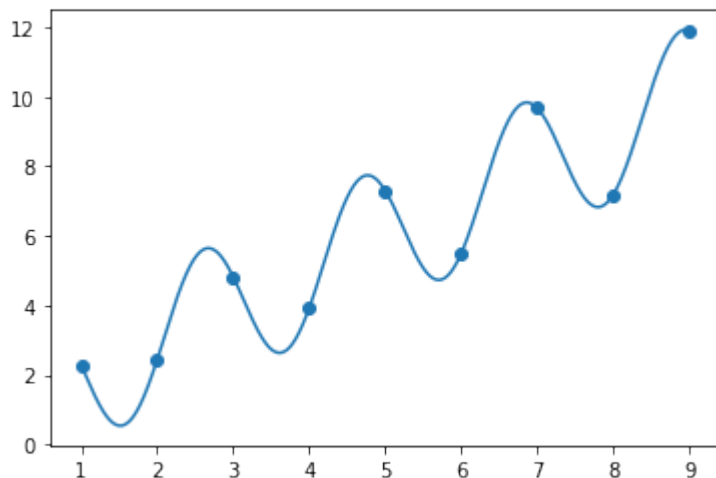
The initial values we chose were not sufficiently close to the “correct” values. This shows the first main problem with nonlinear regression: there may be multiple solutions which are returned based on the initial guess.

Linear regression	Nonlinear regression
single correct solution	multiple solutions possible depending on initial guess
requires no initial guess	requires initial guess
Never returns the “wrong” local minimum solution	Sometimes claims success with “wrong” answer
less flexible in functional form	more flexible functional form

Let’s try a different starting point. Remember the initial data were generated with $\beta = [2, 3, 1, 1]$

```
[27]: beta2 = fit_and_plot([2, 2.8, 1, 1])
beta2
```

```
[27]: array([2., 3., 1., 1.])
```



OK, now we know we're right, right? The error of this fit is essentially zero.

```
[28]: def fiterror(beta):
      return sum((y - f2(x, *beta))**2)
```

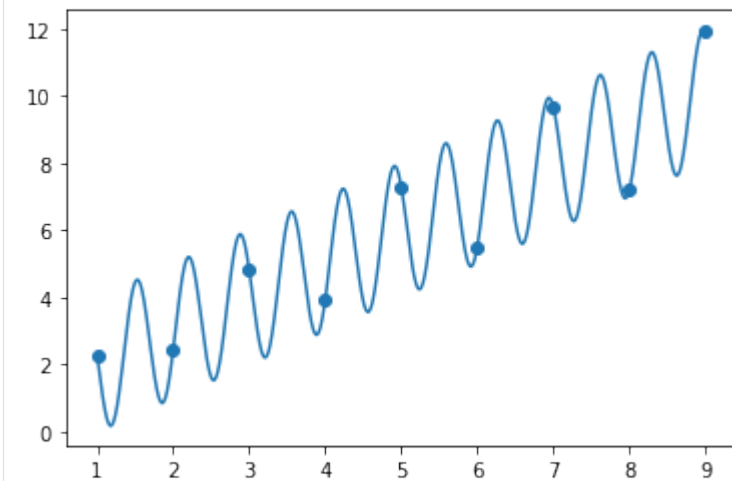
```
[29]: fiterror(beta2)
```

```
[29]: 1.8617117363215879e-28
```

But wait, what about this:

```
[30]: beta3 = fit_and_plot([2, 9.2, 1, 1])
      beta3
```

```
[30]: array([2.          , 9.28318531, 1.          , 1.          ])
```



```
[31]: fiterror(beta3)
```

```
[31]: 2.6505726415425997e-28
```

It is clear that there are multiple solutions to this problem which are equally good! This is a property of nonlinear regression. It is often impossible to recover the “right” values of the parameters. You should therefore be careful of interpreting a good fit as evidence of correctness of your model.

2.7.4 Fitting step responses

It is often prohibitively expensive to develop first principle models of processes and therefore it is very common to estimate low order transfer functions directly from plant data. This is simple to do if we have access to step test results.

```
[29]: import control
      import numpy
      import matplotlib.pyplot as plt
      import tbcontrol
      tbcontrol.expectversion("0.1.10")
      %matplotlib inline
```

Let's start with a higher order process to generate our “real data”

```
[30]: Greal = control.tf([1, 2], [2, 3, 4, 1])
```



```
[31]: ts, ys = control.step_response(Greal)
```

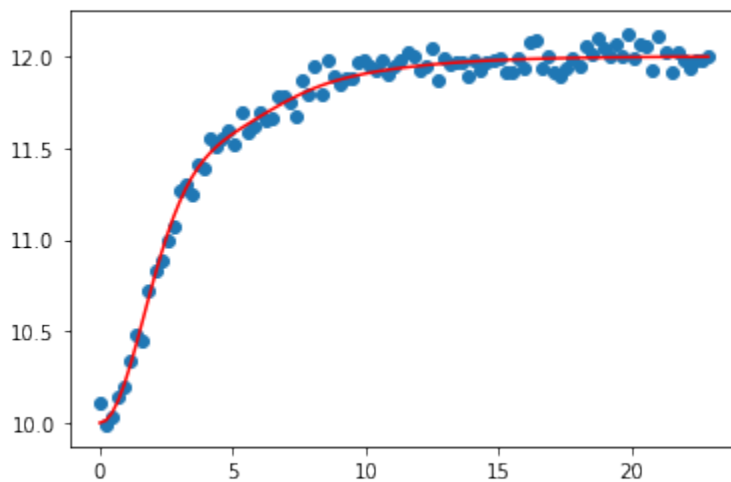
Remember that the real data will not necessarily start at zero, so we'll add in some value for the initial output. We will also add some normally distributed noise to represent measurement error.

```
[32]: yinitial = 10
      measurement_noise = numpy.random.randn(len(ys))*0.05
```

```
[33]: ym = ys + yinitial + measurement_noise
```

```
[34]: plt.scatter(ts, ym)
      plt.plot(ts, ys + yinitial, color='red')
```

```
[34]: [<matplotlib.lines.Line2D at 0x1c1f8f10b8>]
```



```
[35]: import scipy.optimize
```

We'll fit a first order plus dead time and second order plus dead time model. The `tbcontrol.responses` library contains the analytic formulae for these step responses.

```
[36]: from tbcontrol.responses import fopdt, sopdt
```

It is very important to have a good idea of the initial parameter values. Interaction makes it easy to figure out.

```
[37]: from ipywidgets import interact
```

```
[38]: def resultplot(K, tau, theta, y0):
      plt.scatter(ts, ym)
      plt.plot(ts, fopdt(ts, K, tau, theta, y0), color='red')
      plt.show()
```

```
[39]: interact(resultplot,
              K=(1., 10.),
              tau=(0., 10.),
              theta=(0., 10.),
              y0=(0., 20.));

interactive(children=(FloatSlider(value=5.5, description='K', max=10.0, min=1.0),
  ↳FloatSlider(value=5.0, descr...
```


We can use the `scipy.optimize.curve_fit` tool to do this fit just like when we did regression without time.

```
[26]: [K, tau, theta, y0], _ = scipy.optimize.curve_fit(fopdt, ts, ym, [2, 4, 1, 10])
      [K, tau, theta, y0]
```

```
[26]: [1.954094509563327, 2.8181973943514884, 0.6113717655974688, 10.031145239708668]
```

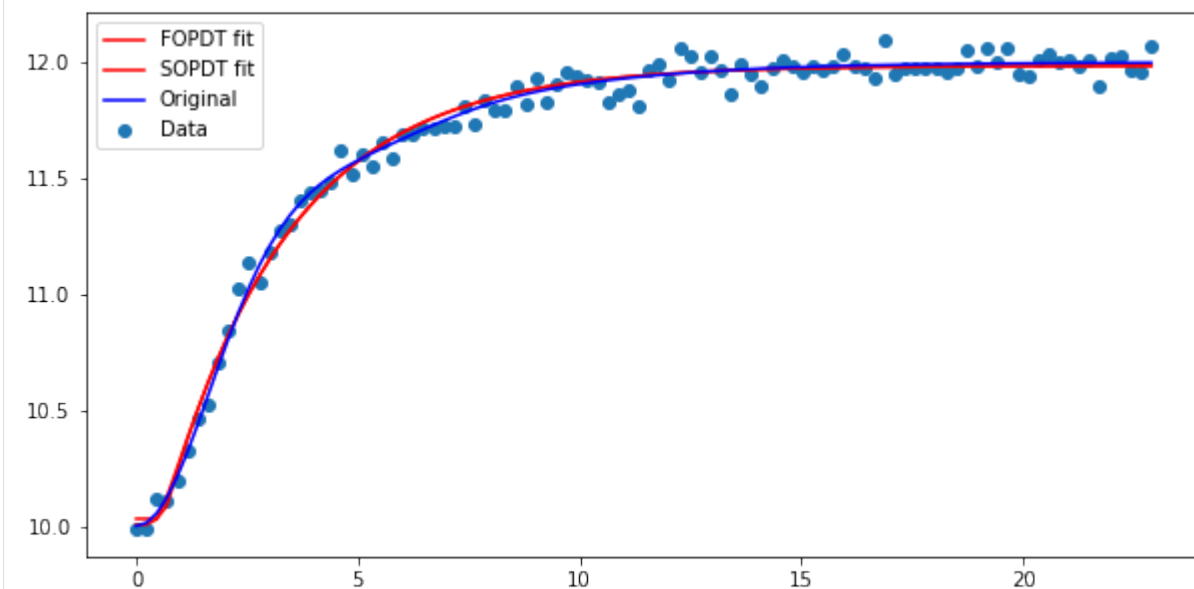
The parameters for the second order model should be similar, with a smaller time constant and overdamped nature.

```
[27]: [K_2, tau_2, zeta_2, theta_2, y0_2], _ = scipy.optimize.curve_fit(sopdt, ts, ym, [2, 1.5, 1, 10])
      [K_2, tau_2, zeta_2, theta_2, y0_2]
```

```
[27]: [1.9798056401724866,
      0.8305183954384022,
      1.8452576915795178,
      0.3199080792764559,
      10.005597510469169]
```

```
[28]: plt.figure(figsize=(10, 5))
      plt.scatter(ts, ym, label='Data')
      plt.plot(ts, fopdt(ts, K, tau, theta, y0), color='red', label='FOPDT fit')
      plt.plot(ts, sopdt(ts, K_2, tau_2, zeta_2, theta_2, y0_2), color='red', label='SOPDT fit')
      plt.plot(ts, ys + 10, color='blue', label='Original')
      plt.legend(loc='best')
```

```
[28]: <matplotlib.legend.Legend at 0x1c1fd1c710>
```



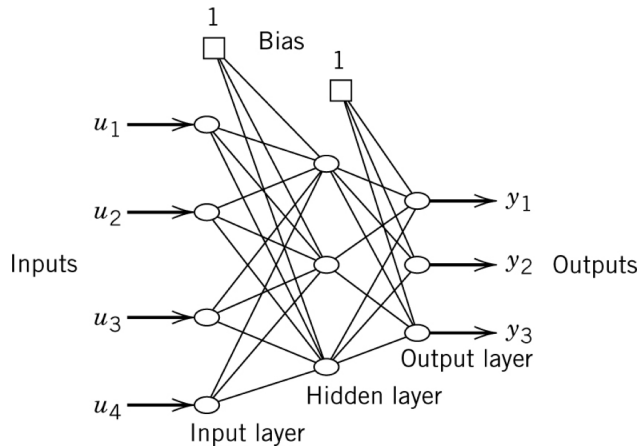
```
[ ]:
```

```
[ ]:
```

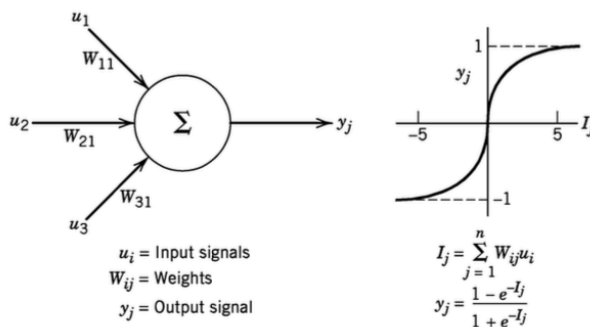

2.7.5 Neural network regression

Neural networks have become very popular recently due to the advent of high performance GPU algorithms for their application. Modern applications of neural networks often use very large networks, but in this sample we will demonstrate the possibilities using a network with a single hidden layer.

The general idea of a neural network is shown in the picture below:



Each circle represents a neuron, and the output of the neuron is calculated as shown below. In simple terms, the output of a neuron is the weighted average of its inputs, passed through what is known as an activation function. The function shown in the picture is known as a logistic or sigmoid function.



```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

We will build our own network in this example to regress a one dimensional function. This means we will only have one input (u), and one output (y), but we can choose the number of hidden neurons. The more hidden neurons we have, the more curves we can handle in the fit function. 3 appears to be enough to do the general fits we do here without the training taking very long. There is no general rule for choosing the number of hidden neurons - use as few as possible to capture your behaviour as each new hidden neuron adds lots of weights which all have to be found.

```
[55]: Nhidden = 3
```

We need to create weights between each input neuron and each hidden neuron as well as each hidden neuron and each output neuron.

```
[56]: w_in_hidden = numpy.random.rand(Nhidden)
w_hidden_out = numpy.random.rand(Nhidden)
```


We also need a bias for the hidden layer and the output layer

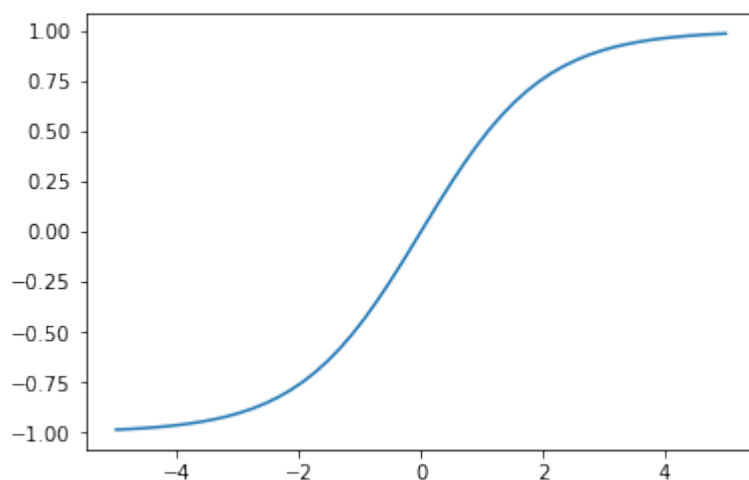
```
[57]: bias_hidden = numpy.random.rand()
      bias_output = numpy.random.rand()
```

We will use a sigmoidal activation function:

```
[58]: def sigmoid(i):
      expi = numpy.exp(-i)
      return ((1 - expi)/(1 + expi))
```

```
[59]: x = numpy.linspace(-5, 5)
      plt.plot(x, sigmoid(x))
```

```
[59]: [<matplotlib.lines.Line2D at 0x1a1ca307b8>]
```



To calculate the output of a neuron, we take the weighted sum of its inputs and apply the activation function. We can do this all simultaneously with numpy arrays:

```
[60]: def network_output(u, w_in_hidden, w_hidden_out, bias_hidden, bias_output):
      h = sigmoid(w_in_hidden*u + bias_hidden)
      y = sigmoid((w_hidden_out*h + bias_output).sum())

      return y
```

```
[61]: network_output(0.1, w_in_hidden, w_hidden_out, bias_hidden, bias_output)
```

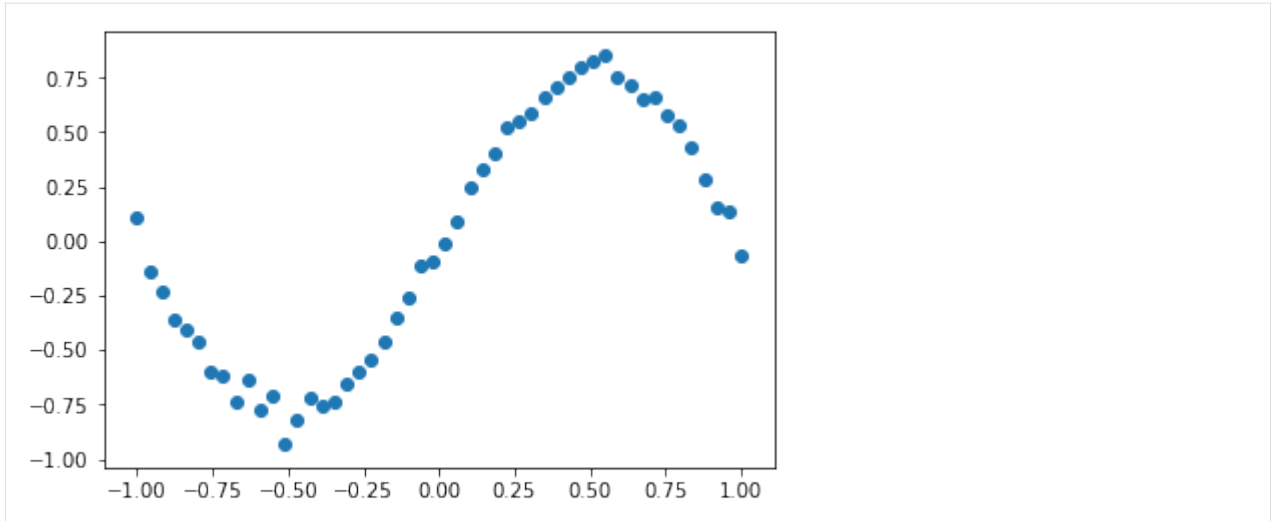
```
[61]: 0.4723938343512528
```

Let's find the weights and bias to regress a function. Due to later decisions about the final activation function which is limited to be between -1 and 1, we will limit our function to be in that range.

```
[62]: known_u = numpy.linspace(-1, 1)
      known_y = numpy.sin(known_u*numpy.pi)*0.8 + numpy.random.randn(len(known_u))*0.05
```

```
[63]: plt.scatter(known_u, known_y)
```

```
[63]: <matplotlib.collections.PathCollection at 0x1a1cb4c898>
```

```
[64]: import scipy.optimize
```

Since we're going to use optimisation functions which take an array, we need to be able to our parameters into a single array and unpack them again.

```
[65]: def pack(w_in_hidden, w_hidden_out, bias_hidden, bias_output):
        return numpy.concatenate([w_in_hidden,
                                   w_hidden_out,
                                   numpy.array([bias_hidden]),
                                   numpy.array([bias_output])])

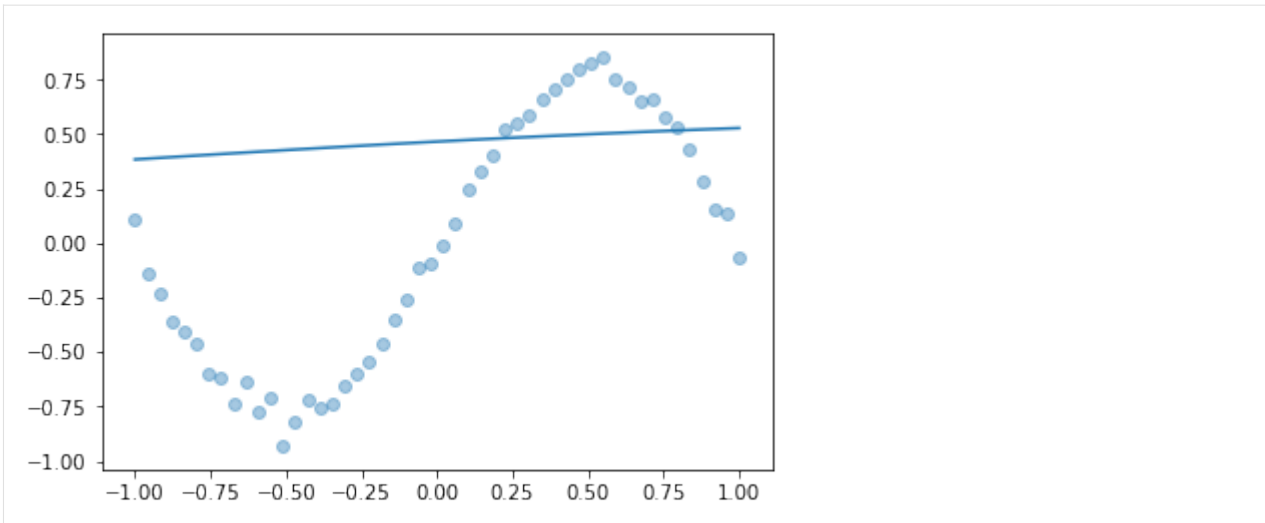
    def unpack(parameters):
        parts = numpy.split(parameters, [Nhidden, 2*Nhidden, 2*Nhidden + 1])
        return parts
```

```
[66]: p0 = pack(w_in_hidden, w_hidden_out, bias_hidden, bias_output)
```

```
[67]: def predict(parameters, us):
        w_in_hidden, w_hidden_out, bias_hidden, bias_output = unpack(parameters)
        return numpy.array([network_output(u, w_in_hidden, w_hidden_out, bias_hidden,
        ↪ bias_output) for u in us])
```

```
[68]: def plotfit(predictions):
        plt.scatter(known_u, known_y, alpha=0.4)
        plt.plot(known_u, predictions)
```

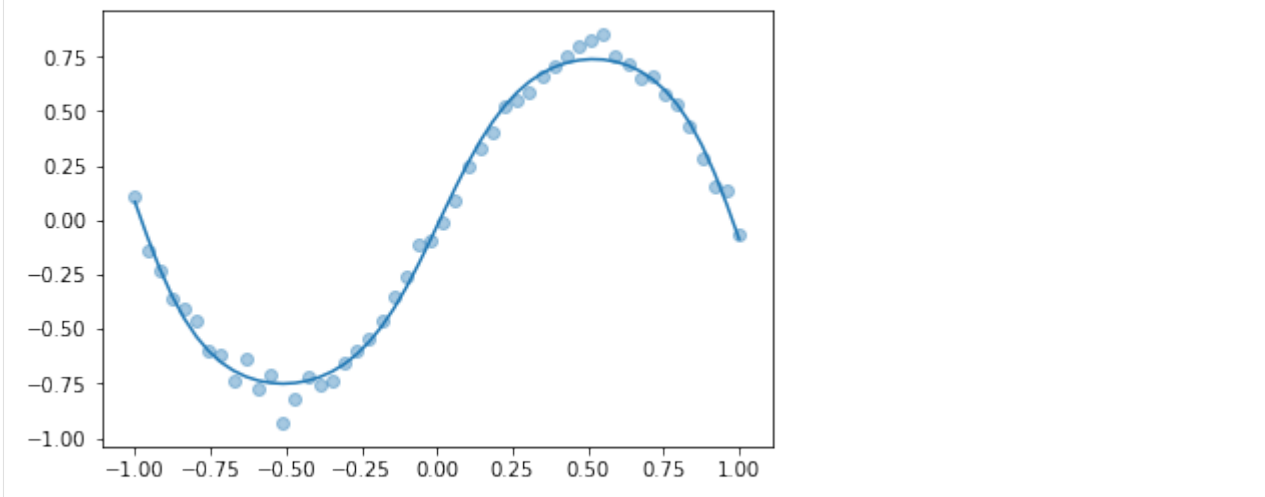
```
[69]: plotfit(predict(p0, known_u))
```

```
[70]: def errorfunction(parameters):  
       return known_y - predict(parameters, known_u)
```

```
[71]: result = scipy.optimize.least_squares(errorfunction, p0)
```

```
[72]: plotfit(predict(result.x, known_u))
```



Scikit-learn

As I've mentioned before, you're probably better off using a library for things like this. The Scikit-Learn library has neural network regression built in. It is part of the standard Anaconda install.

```
[44]: import sklearn  
import sklearn.neural_network
```

```
[45]: net = sklearn.neural_network.MLPRegressor(hidden_layer_sizes=Nhidden,  
                                                activation='tanh',  
                                                solver='lbfgs', max_iter=1000, learning_  
↳ rate_init=0.001)
```

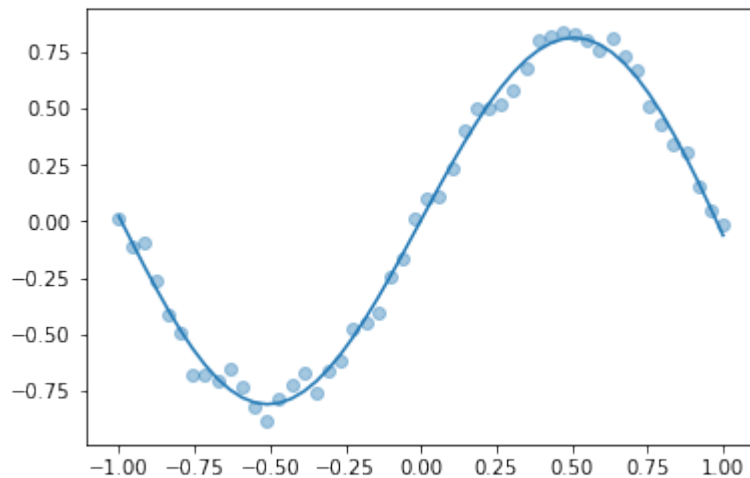


```
[46]: observations = numpy.atleast_2d(known_u).T
```

```
[47]: net.fit(observations, known_y)
```

```
[47]: MLPRegressor(activation='tanh', alpha=0.0001, batch_size='auto', beta_1=0.9,
    beta_2=0.999, early_stopping=False, epsilon=1e-08,
    hidden_layer_sizes=4, learning_rate='constant',
    learning_rate_init=0.001, max_iter=1000, momentum=0.9,
    nesterovs_momentum=True, power_t=0.5, random_state=None,
    shuffle=True, solver='lbfgs', tol=0.0001, validation_fraction=0.1,
    verbose=False, warm_start=False)
```

```
[48]: plotfit(net.predict(observations))
```



Keras

The Keras library offers additional flexibility, but is not installed by default in Anaconda.

```
[49]: import keras
```

```
/Users/alchemyst/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36:
↳FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↳floating` is deprecated. In future, it will be treated as `np.float64 == np.
↳dtype(float).type`.
    from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

```
[50]: model = keras.models.Sequential()
```

```
[51]: model.add(keras.layers.Dense(Nhidden, input_shape=(1,), activation='tanh'))
    model.add(keras.layers.Dense(1, activation='tanh'))
```

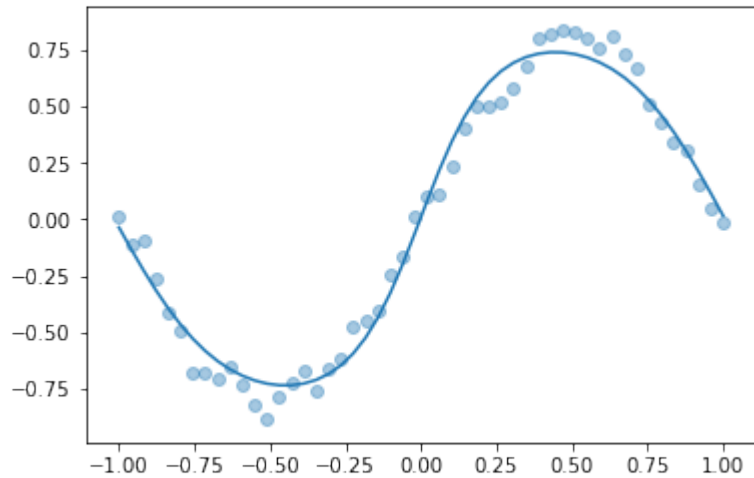
```
[52]: model.compile(optimizer='rmsprop', loss='mse')
```

```
[53]: model.fit(known_u, known_y, epochs=10000, verbose=False)
```



```
[53]: <keras.callbacks.History at 0x1a1c6d5048>
```

```
[54]: plotfit(model.predict(known_u))
```



```
[28]: import pandas
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

We will fit an ARX model of this form to a step response.

$$y(k) = a_1y(k-1) + a_2y(k-2) + b_1u(k-1) + b_2u(k-2)$$

```
[29]: data = pandas.read_csv('../assets/data.csv', index_col='k')
data['uk'] = 1
data.loc[0] = [0, 1] # input changes at t=0
data.loc[-1] = [0, 0] # everything was steady at t < 0
data = data.sort_index()
data
```

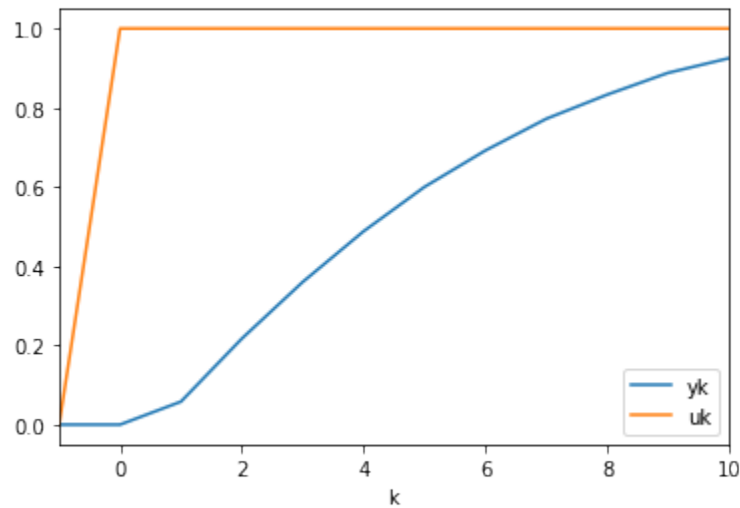
```
[29]:
```

	yk	uk
k		
-1	0.000	0
0	0.000	1
1	0.058	1
2	0.217	1
3	0.360	1
4	0.488	1
5	0.600	1
6	0.692	1
7	0.772	1
8	0.833	1
9	0.888	1
10	0.925	1

```
[30]: y = data.yk
u = data.uk
```



```
[31]: data.plot()
[31]: <matplotlib.axes._subplots.AxesSubplot at 0x6215bf828>
```



We effectively have the following equations (I repeat the equation here for convenience):

$$y(k) = a_1 y(k-1) + a_2 y(k-2) + b_1 u(k-1) + b_2 u(k-2)$$

```
[32]: for k in range(1, 11):
        print(f'{y[k]:.2} = a1×{y[k-1]:.2} + a2×{y[k-2]:.2} + b1×{u[k-1]} + b2×{u[k-2]}')
        ↪
```

```
0.058 = a1×0.0 + a2×0.0 + b1×1 + b2×0
0.22 = a1×0.058 + a2×0.0 + b1×1 + b2×1
0.36 = a1×0.22 + a2×0.058 + b1×1 + b2×1
0.49 = a1×0.36 + a2×0.22 + b1×1 + b2×1
0.6 = a1×0.49 + a2×0.36 + b1×1 + b2×1
0.69 = a1×0.6 + a2×0.49 + b1×1 + b2×1
0.77 = a1×0.69 + a2×0.6 + b1×1 + b2×1
0.83 = a1×0.77 + a2×0.69 + b1×1 + b2×1
0.89 = a1×0.83 + a2×0.77 + b1×1 + b2×1
0.93 = a1×0.89 + a2×0.83 + b1×1 + b2×1
```

```
[33]: pandas.DataFrame([(k,
                           y[k],
                           y[k-1],
                           y[k-2],
                           u[k-1],
                           u[k-2])
                           for k in range(1, 11)], columns=['k', 'y[k]', 'y[k-1]',
                           ↪ 'y[k-2]', 'u[k-1]', 'u[k-2]'])
```

```
[33]:
```

	k	y[k]	y[k-1]	y[k-2]	u[k-1]	u[k-2]
0	1	0.058	0.000	0.000	1	0
1	2	0.217	0.058	0.000	1	1
2	3	0.360	0.217	0.058	1	1
3	4	0.488	0.360	0.217	1	1
4	5	0.600	0.488	0.360	1	1
5	6	0.692	0.600	0.488	1	1

(continues on next page)

(continued from previous page)

6	7	0.772	0.692	0.600	1	1
7	8	0.833	0.772	0.692	1	1
8	9	0.888	0.833	0.772	1	1
9	10	0.925	0.888	0.833	1	1

We notice that these equations are linear in the coefficients. We can define $\beta = [a_1, a_2, b_1, b_2]^T$. Now, to write the above equations in matrix form $Y = X\beta$, we define

```
[34]: Y = numpy.atleast_2d(y.loc[1:]).T
      Y
```

```
[34]: array([[0.058],
            [0.217],
            [0.36 ],
            [0.488],
            [0.6   ],
            [0.692],
            [0.772],
            [0.833],
            [0.888],
            [0.925]])
```

To build the coefficient matrix we observe that there are two blocks of constant diagonals (the part with the y s and the part with the u s). Matrices with constant diagonals are called Toeplitz matrices and can be constructed with the `scipy.linalg toeplitz` function.

```
[35]: import scipy
      import scipy.linalg
```

```
[36]: X1 = scipy.linalg.toeplitz(y.loc[0:9], [0, 0])
      X1
```

```
[36]: array([[0.   , 0.   ],
            [0.058, 0.   ],
            [0.217, 0.058],
            [0.36 , 0.217],
            [0.488, 0.36 ],
            [0.6  , 0.488],
            [0.692, 0.6  ],
            [0.772, 0.692],
            [0.833, 0.772],
            [0.888, 0.833]])
```

```
[37]: X2 = scipy.linalg.toeplitz(u.loc[0:9], [0, 0])
      X2
```

```
[37]: array([[1, 0],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1],
            [1, 1]])
```



```
[24]: X = numpy.hstack([X1, X2])
```

```
[25]: X
```

```
[25]: array([[0.    , 0.    , 1.    , 0.    ],
          [0.058, 0.    , 1.    , 1.    ],
          [0.217, 0.058, 1.    , 1.    ],
          [0.36 , 0.217, 1.    , 1.    ],
          [0.488, 0.36 , 1.    , 1.    ],
          [0.6  , 0.488, 1.    , 1.    ],
          [0.692, 0.6  , 1.    , 1.    ],
          [0.772, 0.692, 1.    , 1.    ],
          [0.833, 0.772, 1.    , 1.    ],
          [0.888, 0.833, 1.    , 1.    ]])
```

Another option is to use the loop from before to construct the matrices. This is a little more legible but slower:

```
[26]: Y = []
      X = []
      for k in range(1, 11):
          Y.append([y[k]])
          X.append([y[k - 1], y[k - 2], u[k - 1], u[k - 2]])
      Y = numpy.array(Y)
      X = numpy.array(X)
```

We solve for β as we did for linear regression:

```
[27]: beta, _, _, _ = numpy.linalg.lstsq(X, Y, rcond=None)
      beta
[27]: array([[ 0.98464753],
          [-0.12211256],
          [ 0.058      ],
          [ 0.10124916]])
```

```
[ ]:
```

2.8 Frequency domain

```
[1]: import sympy
      sympy.init_printing()
      %matplotlib inline
```

2.8.1 Fourier series

We can approximate a periodic function of period P to arbitrary accuracy by adding sine and cosine terms (disguised via the Euler formula in the complex exponential):

$$S_N(t) = \sum_{n=-N}^N c_n \exp\left(\frac{i2\pi nt}{P}\right)$$

with

$$c_n = \frac{1}{P} \int_{t_0}^{t_0+P} f(t) \exp\left(\frac{-i2\pi nt}{P}\right) dt$$

Compare this last equation to the Fourier transform and the Laplace transform:

$$\mathcal{F}\{\cdot\} = \int_{-\infty}^{\infty} f(t) \exp(-i\omega t) dt \quad \mathcal{L}\{\cdot\} = \int_0^{\infty} f(t) \exp(-st) dt$$

The following two functions attempt to match the notation above as closely as possible using sympy functions

```
[2]: import sympy

[3]: i2pi = sympy.I*2*sympy.pi
    exp = sympy.exp

[4]: def S(N):
    return sum(c(n)*exp(i2pi*n*t/P) for n in range(-N, N+1)).expand(complex=True).
    ↪simplify()

[5]: def c(n):
    return (sympy.integrate(
        f(t)*exp((-i2pi * n * t)/P),
        (t, t0, t0 + P))/P)
```

These functions work quite well for a periodic sawtooth function:

```
[6]: a = sympy.Symbol('a', positive=True)

[7]: def f(t):
    return t

[8]: P = 20
    t0 = -10

[9]: t = sympy.Symbol('t', real=True)

[10]: N = 6

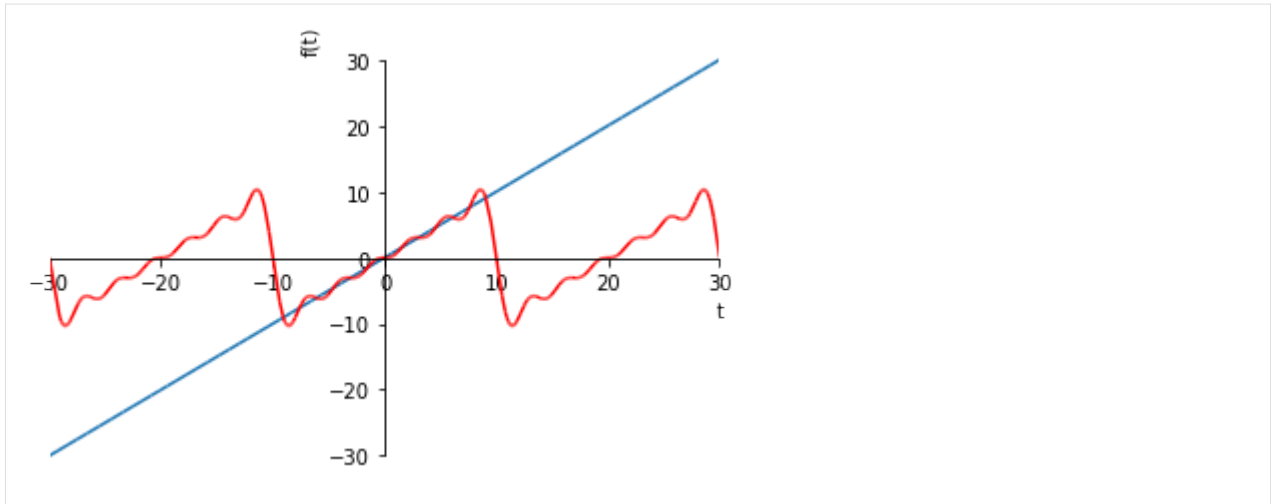
[11]: analytic_approx = S(N).expand()
    analytic_approx

[11]: 
$$\frac{20}{\pi} \sin\left(\frac{\pi t}{10}\right) - \frac{10}{\pi} \sin\left(\frac{\pi t}{5}\right) + \frac{20}{3\pi} \sin\left(\frac{3\pi t}{10}\right) - \frac{5}{\pi} \sin\left(\frac{2\pi t}{5}\right) + \frac{4}{\pi} \sin\left(\frac{\pi t}{2}\right) - \frac{10}{3\pi} \sin\left(\frac{3\pi t}{5}\right)$$

```

Notice that the function we defined was just $y = t$, but Fourier series always approximates periodic signals. We can see the bit we approximated repeating if we plot over a larger interval

```
[12]: interval = (t, t0-P, t0+2*P)
    p1 = sympy.plot(f(t), interval, show=False)
    p2 = sympy.plot(analytic_approx, interval, show=False)
    p2[0].line_color = 'red'
    p1.extend(p2)
    p1.show()
```

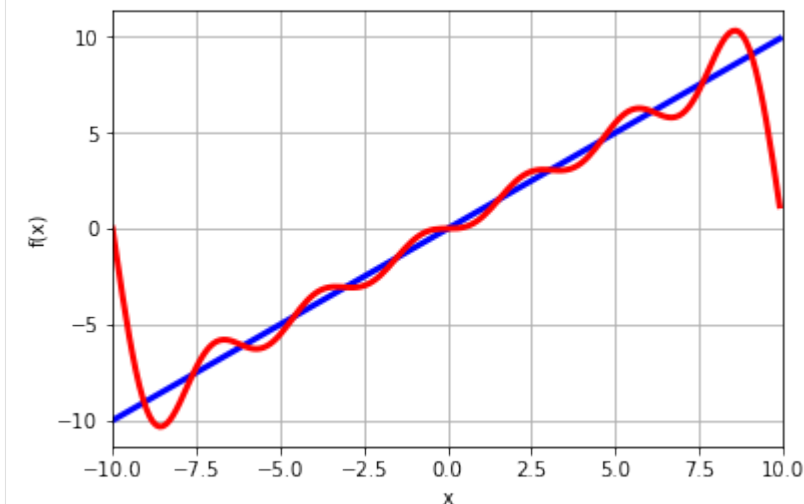
Unfortunately, this notationally elegant approach does not appear to work for `f = sympy.Heaviside`, and is also quite slow. The `mpmath` library supplies a numeric equivalent.

```
[13]: try:
      import mpmath
    except ImportError:
      from sympy import mpmath
```

```
[14]: cs = mpmath.fourier(f, [t0, t0+P], N)

def numeric_approx(t):
    return mpmath.fourierval(cs, [t0, t0+P], t)

mpmath.plot([f, numeric_approx], [t0, t0+P])
```



The coefficients returned by the `mpmath.fourier` functions are for the cosine and sine terms in this alternate representation of S_N

$$s_N(t) = \sum_{n=0}^N \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right) \right)$$

We can see the similarity clearly by showing the expression we obtained before multiplied out and numerically evaluated

```
[15]: sympy.N(analytic_approx)
```

```
[15]: 6.36619772367581 sin( $\frac{\pi t}{10}$ ) - 3.18309886183791 sin( $\frac{\pi t}{5}$ ) + 2.12206590789194 sin( $\frac{3\pi}{10}t$ ) - 1.59154943091895 sin( $\frac{2\pi}{5}t$ ) + 1.2732395447351628 sin( $\frac{3\pi}{5}t$ ) - 1.0610329539459689 sin( $\frac{4\pi}{5}t$ )
```

```
[16]: cs
```

```
[16]: ([mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0')],  
      [mpf('0.0'),  
      mpf('6.366197723675814'),  
      mpf('-3.183098861837907'),  
      mpf('2.1220659078919377'),  
      mpf('-1.5915494309189535'),  
      mpf('1.2732395447351628'),  
      mpf('-1.0610329539459689')])
```

Notice that all the cosine coefficients are zero. This is true in general for odd functions.

Step function

Let's now switch to the Heaviside step and draw the various sinusoids in the approximation more explicitly.

```
[17]: import matplotlib.pyplot as plt  
import numpy  
%matplotlib inline
```

```
[18]: N = 20
```

```
[19]: @numpy.vectorize  
def f(t):  
    if t < 0:  
        return 0  
    else:  
        return 1
```

```
[20]: cs = mpmath.fourier(f, [t0, t0+P], N)  
cs
```

```
[20]: ([mpf('0.5'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0')],  
      [mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0'),  
      mpf('0.0')])
```

(continues on next page)

(continued from previous page)

```

mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0'),
mpf('0.0')],
[mpf('0.0'),
mpf('0.63675856242795037'),
mpf('0.0002778345608862911'),
mpf('0.21262398013873579'),
mpf('0.00055771310594110598'),
mpf('0.12802302327076248'),
mpf('0.00084172518315335683'),
mpf('0.091931640154520794'),
mpf('0.0011320535572161288'),
mpf('0.072015836887865267'),
mpf('0.0014310262052054015'),
mpf('0.059459060405769593'),
mpf('0.0017411752581774926'),
mpf('0.050872056499090879'),
mpf('0.0020653060695558258'),
mpf('0.044674904291434447'),
mpf('0.0024065804723679999'),
mpf('0.040032978763113902'),
mpf('0.002768619592788056'),
mpf('0.036465018126357898'),
mpf('0.0031556335077388429')])

```

We see that all the cosine terms but the first are zero, so the first cosine coefficient represents a constant being added to the sine series.

```

[21]: constant = cs[0][0]
      sinecoefficients = cs[1]

```

I'm constructing two dimensional arrays here which will allow my formulae to work nicely when broadcasting. I'll use the convention that the time dimension is in the columns and each sine response is in the rows.

```

[22]: tt = numpy.linspace(t0, t0 + P, 1000)
      t2d = tt[numpy.newaxis] # two dimensional time array, time in the columns

```

```

[23]: # two dimensional arrays for the sine coefficients and n - they vary in the rows
      an = numpy.array(sinecoefficients)[:, numpy.newaxis]
      n = numpy.arange(N+1)[:, numpy.newaxis]

```

This way I can build a whole array of sine responses automatically.

```

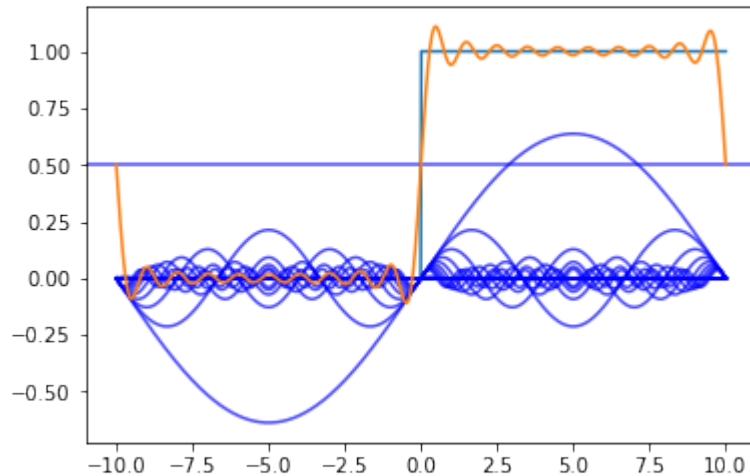
[24]: sines = an*numpy.sin(2*numpy.pi*n*t2d/P)

```



```
[25]: plt.plot(tt, f(tt))
plt.plot(tt, sines.T, color='blue', alpha=0.7)
plt.axhline(constant, color='blue', alpha=0.7)
plt.plot(tt, sum(sines) + constant)
```

```
[25]: [<matplotlib.lines.Line2D at 0x112f5ccf8>]
```



We can see that the sum of the sines approximates the step function.

Step response via Frequency response

Let's compare the result of a "traditional" calculation of a step response of a second order system to a new way which uses the frequency response of the transfer function and the Fourier series of the step input.

First I am going to find the solution using purely analytic methods.

```
[26]: s = sympy.Symbol('s')
```

```
[27]: tau = 1
zeta = sympy.Rational(1, 2)
```

We redefine t here to be positive, which will allow us a simple evaluation of the inverse laplace later.

```
[28]: t = sympy.Symbol('t', positive=True)
```

```
[29]: G = 1/(tau**2*s**2 + 2*tau*zeta*s + 1)
g = sympy.inverse_laplace_transform(G/s, s, t)
```

Here I build a function which can evaluate the response numerically. I use the extra argument to make the function able to work with numpy arrays

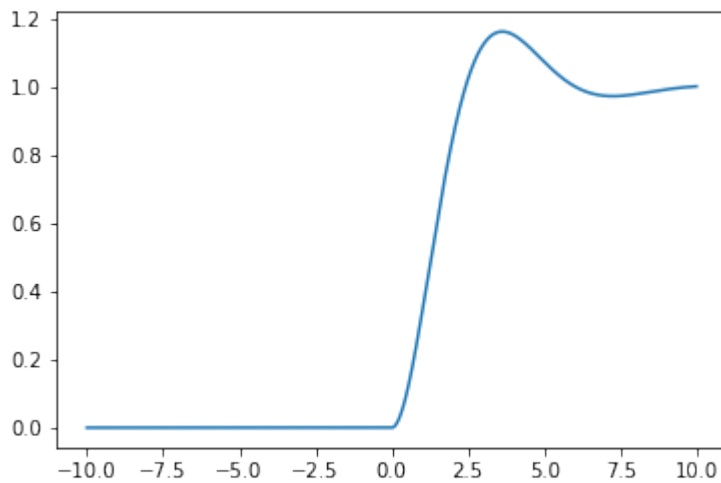
```
[30]: geval = sympy.lambdify(t, sympy.N(g.simplify()), 'numpy')
```

```
[31]: gtvalues = geval(tt)
gtvalues[tt<0] = 0
```

We can see the familiar second order step response:


```
[32]: plt.plot(tt, gtvalues)
```

```
[32]: [<matplotlib.lines.Line2D at 0x1130fd160>]
```



Now, let's try to get the same answer by exploiting the frequency response of the transfer function. First, we need a function which can evaluate the transfer function at particular values and return a numeric result. As before I use `lambdify` and the extra argument to make the function able to work with numpy arrays

```
[33]: Geval = sympy.lambdify(s, G, 'numpy')
```

Now, we need to build an array for the frequencies of the Fourier series. Remember we had terms of the form $\sin\left(\frac{2\pi nt}{P}\right) = \sin(\omega t)$ in the Fourier series.

```
[34]: omega = 2*n*numpy.pi/P
```

We evaluate the frequency response of the transfer function at the Fourier frequencies by using the substitution $s = \omega i$. The complex number is `j` in Python.

```
[35]: freqresp = Geval(omega*1j)
```

```
[36]: K = gain = numpy.abs(freqresp)
```

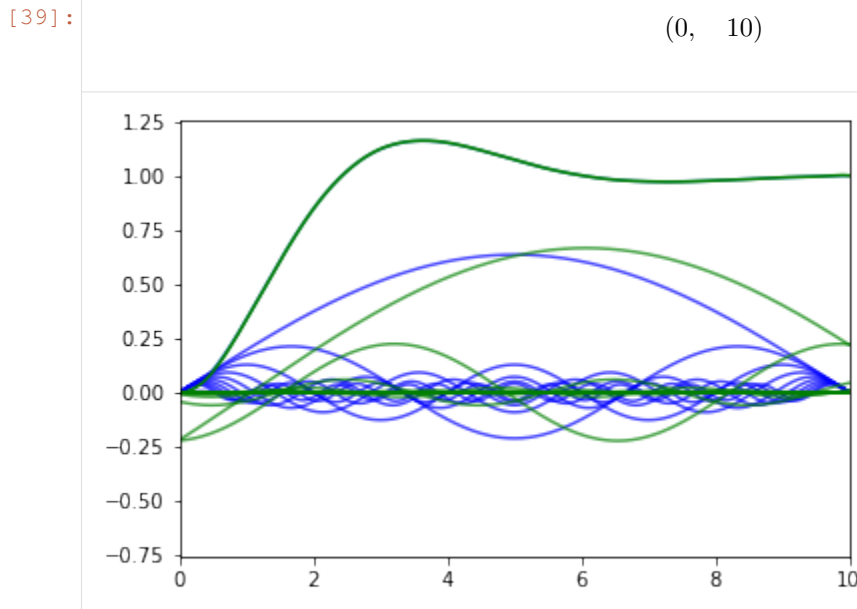
```
[37]: phi = phase = numpy.angle(freqresp)
```

Now, we build the approximation of the response by simply multiplying by the gain K and shifting by the phase ϕ :

$$r(t) = K a_n \sin(2\pi nt/P + \phi)$$

```
[38]: shiftedsines = K*a*numpy.sin(2*numpy.pi*n*t2d/P + phi)
```

```
[39]: plt.plot(tt, gtvalues)
plt.plot(tt, sines.T, color='blue', alpha=0.7)
plt.plot(tt, shiftedsines.T, color='green', alpha=0.7)
plt.plot(tt, sum(shiftedsines) + constant, color='green')
plt.xlim(0, t0 + P)
```

[]:

```
[1]: from IPython.display import Audio
```

```
[2]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

2.8.2 What does a sinusoid sound like?

We all have advanced frequency detectors in our ears. This notebook will show how listening to signals gives you a whole different way to understand the frequency domain.

We start by constructing a musically relevant note as a pure sine wave

```
[48]: fs = 44100 # sampling frequency, Hz This is the rate at which CDs are sampled.
T = 0.5 # note length, seconds
A = 440 # Note frequency, Hz
twopi = 2*numpy.pi

t = numpy.linspace(0, T, int(T*fs), endpoint=False) # time variable
dt = t[1] # Sampling time
def note(frequency):
    return numpy.sin(twopi*frequency*t) # pure sine wave at 440 Hz

# load a NumPy array
Audio(note(A), rate=fs)
```

```
[48]: <IPython.lib.display.Audio object>
```

Now we construct the Chromatic 12 note scale:

Chromatic scale


```
[49]: scalet = numpy.linspace(0, T*13, int(T*fs*13), endpoint=False)
      scale = numpy.concatenate([note(A*2**(i/12.)) for i in range(13)])
```

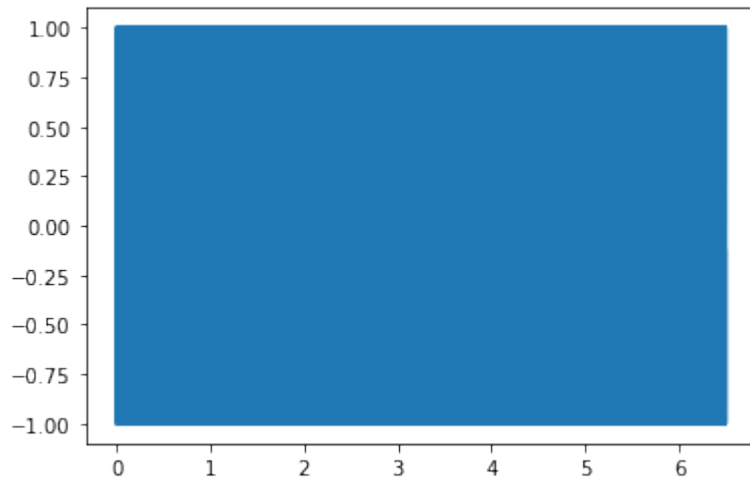
```
[50]: Audio(scale, rate=fs)
```

```
[50]: <IPython.lib.display.Audio object>
```

Of course, this is quite a large number of points, you can't really see what's going on

```
[55]: plt.plot(scalet, scale)
```

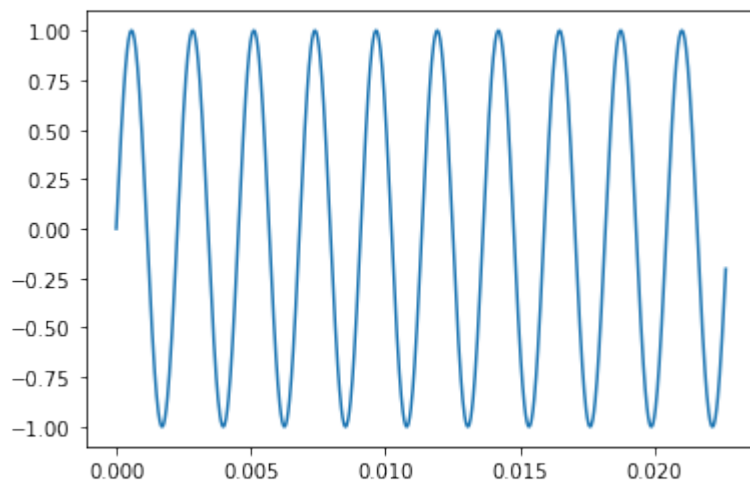
```
[55]: [<matplotlib.lines.Line2D at 0x124602b70>]
```



You can see the sinusoids better if you zoom in

```
[56]: Nzoom = 1000
      plt.plot(scalet[:Nzoom], scale[:Nzoom])
```

```
[56]: [<matplotlib.lines.Line2D at 0x11b71b208>]
```



Let's listen to the effect of running this through a first order filter.

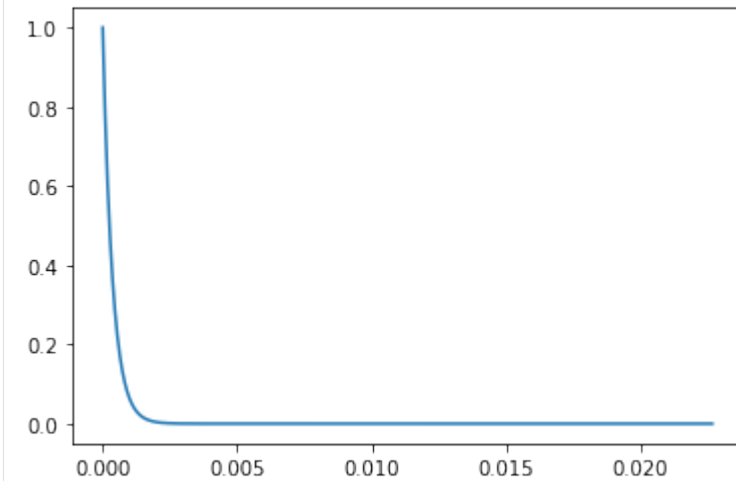
$$G_f = \frac{1}{\tau s + 1}$$

We'll use the convolution, so we first obtain the impulse response of the filter

```
[57]: omega = 440*twopi # (440 cycles/second)*(2 pi radians / cycle)
      tau = 1/omega
      first_order_impulse = numpy.exp(-t/tau)
```

```
[58]: plt.plot(t[:Nzoom], first_order_impulse[:Nzoom])
```

```
[58]: [<matplotlib.lines.Line2D at 0x11cc9cef0>]
```



Then we calculate the output signal via convolution. Recall that when we wrote $y(s) = G_f(s)u(s)$, this was equivalent to calculating $y(t) = g_f(t) * u(t)$ where the $*$ represents convolution. In this case the filtered signal is $y(t)$ and the original scale is $u(t)$.

```
[59]: def filtersignal(signal):
      return numpy.convolve(signal, first_order_impulse, 'same')/len(signal)*len(first_
      ↪order_impulse)
```

```
[60]: filtered = filtersignal(scale)
```

As a side note: This is not how filtering is done in practice, you can see this process takes a while.

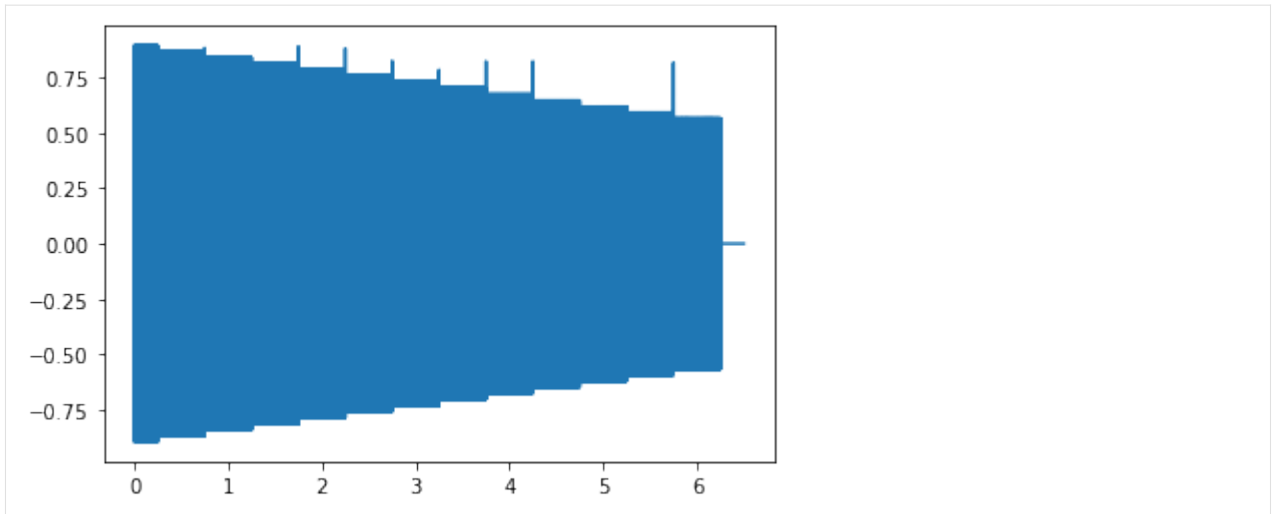
```
[61]: Audio(filtered, rate=fs)
```

```
[61]: <IPython.lib.display.Audio object>
```

Can you hear how the sound gets softer as the frequency goes up? This is the effect of the filter. If we plot the whole waveform we can clearly see the amplitude going down in steps.

```
[13]: plt.plot(scalet, filtered)
```

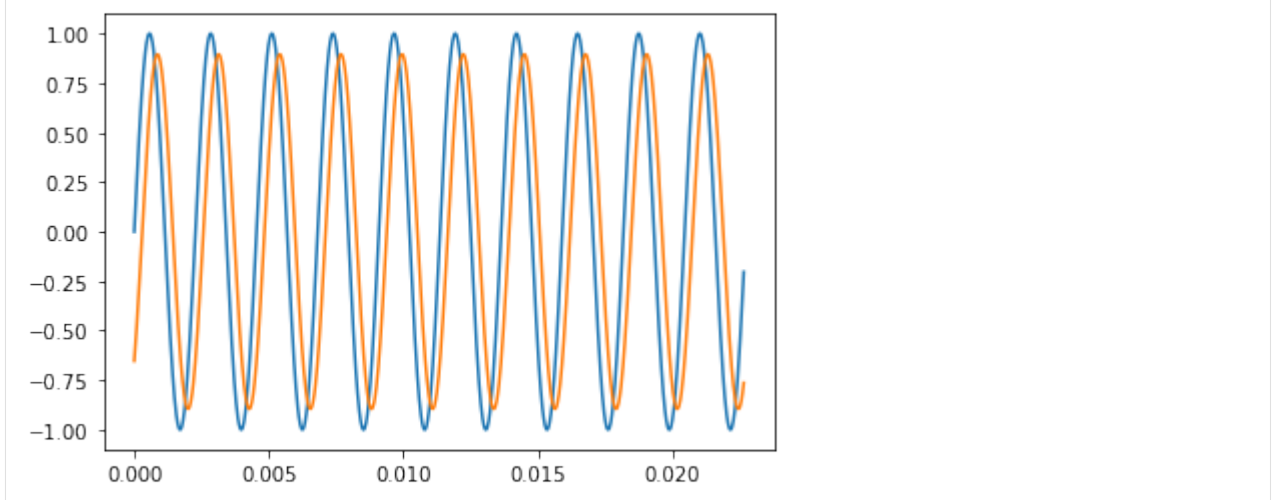
```
[13]: [<matplotlib.lines.Line2D at 0x118c3fa20>]
```

When we zoom in so that we can see the wave forms, we can see that the low frequencies are attenuated less than the high frequencies.

```
[62]: plt.plot(scale[:Nzoom], scale[:Nzoom],
               scale[:Nzoom], filtered[:Nzoom])
```

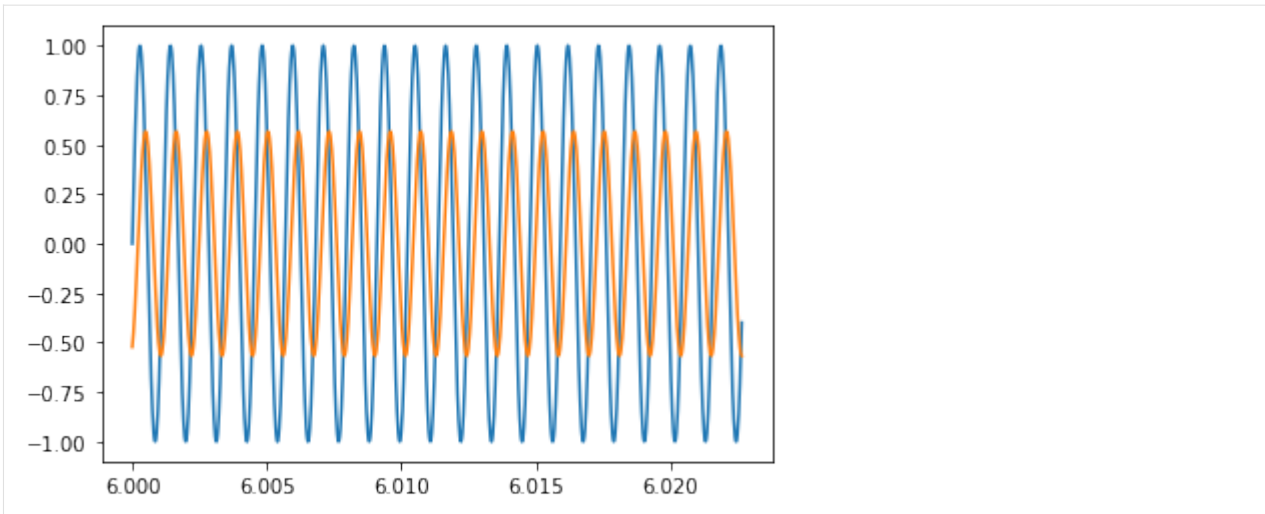
```
[62]: [<matplotlib.lines.Line2D at 0x129127e10>,
       <matplotlib.lines.Line2D at 0x129127f60>]
```



```
[63]: start = int(fs/T)*3
```

```
[64]: plt.plot(scale[start:start+Nzoom], scale[start:start+Nzoom],
               scale[start:start+Nzoom], filtered[start:start+Nzoom])
```

```
[64]: [<matplotlib.lines.Line2D at 0x12565d1d0>,
       <matplotlib.lines.Line2D at 0x12565d320>]
```

But signals aren't pure sinusoids

Why have I chosen pure sinusoids to play with here? Because they can be combined to form all the other signals via Fourier series. Just to show that everything still works even when we don't have pure sinusoids, let's form a chord, which will be three sinusoids playing together. I'll build a [major chord](#) in just intonation.

I'll also use the highest note an octave higher so that we can hear the filtering effect more clearly. I'm using `tile` here to repeat the sample a couple of times to make the sound longer.

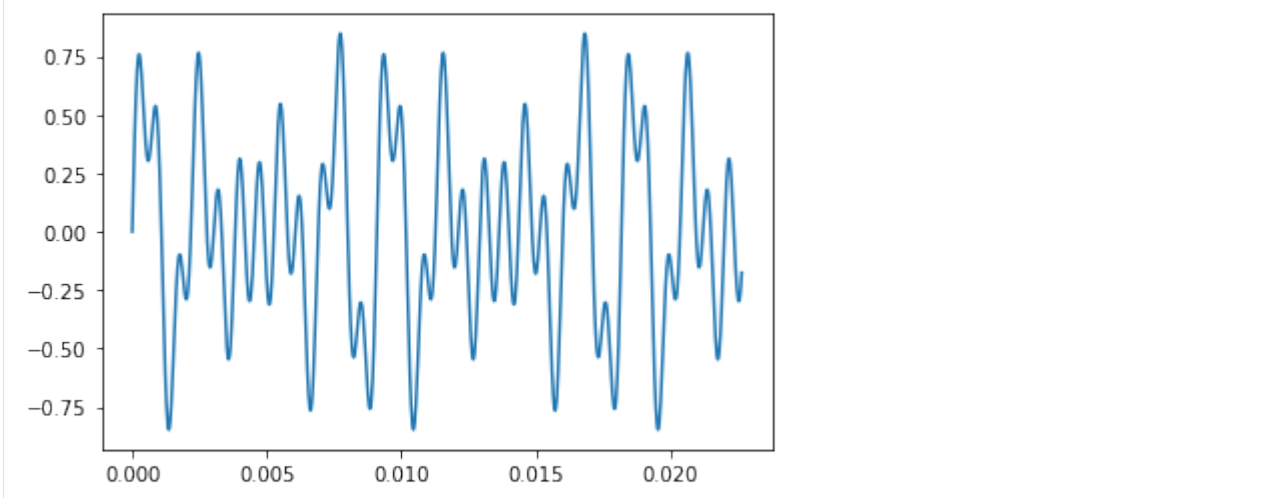
```
[65]: notes = [A, A*5/4, A*3/2*2]
```

```
[66]: chord = numpy.tile(1/len(notes)*sum(note(n) for n in notes), 3)
```

We can see that this is no longer a simple sinusoid

```
[67]: plt.plot(scalet[:Nzoom], chord[:Nzoom])
```

```
[67]: [<matplotlib.lines.Line2D at 0x12a146d68>]
```




```
[68]: Audio(chord, rate=fs)
[68]: <IPython.lib.display.Audio object>

[69]: filtered = filtersignal(chord)

[70]: Audio(filtered, rate=fs)
[70]: <IPython.lib.display.Audio object>
```

Numeric Fourier Transform

We can obtain an approximation of a Fourier transform of a signal by using the Fast Fourier Transform (fft). Functions relating to the fft are found in `numpy.fft`.

First we calculate the fft using `rfft`. The `r` stands for “real”, because we have a signal containing only real values. If we use `numpy.fft.fft` it assumes that the signal could contain complex values and returns twice as many values. `numpy.fft.tfftfreq` is used to calculate the frequencies for which `rfft` has calculated the values.

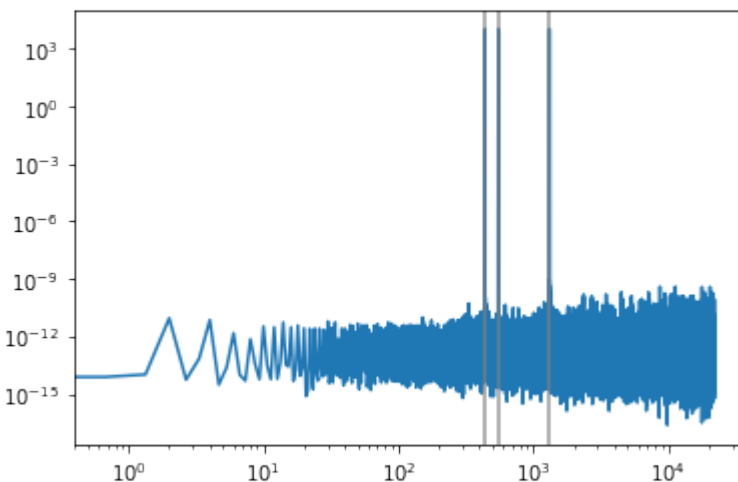
```
[74]: fft = numpy.fft.rfft(chord)
      fftfreq = numpy.fft.rfftfreq(len(chord), dt)
```

When we plot the gain part of this signal, we can clearly see the peaks at the frequencies of the sinusoids from before.

```
[75]: def bodegain(fft):
      plt.loglog(fftfreq, numpy.abs(fft))

      def shownotes():
          for n in notes:
              plt.axvline(n, color='grey', alpha=0.8)
```

```
[76]: bodegain(fft)
      shownotes()
```

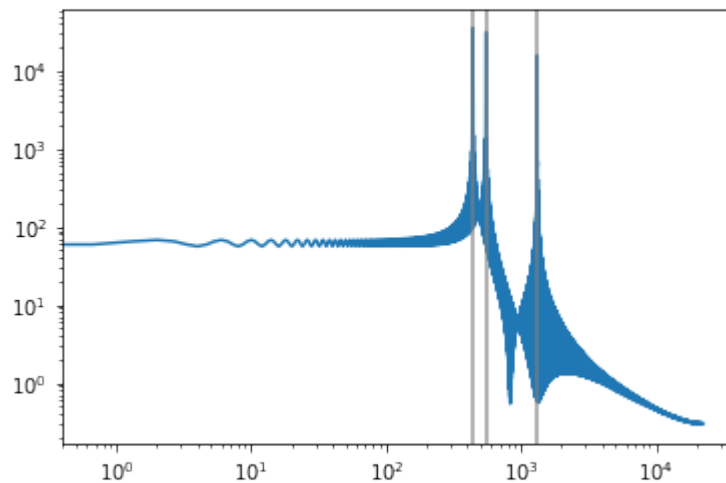


Now, let's look at the filtered version

```
[77]: filtered_fft = numpy.fft.rfft(filtered)
```



```
[78]: bodegain(filtered_fft)
      shownotes()
```



It's clear how the lower frequencies are attenuated (made smaller) by the filter.

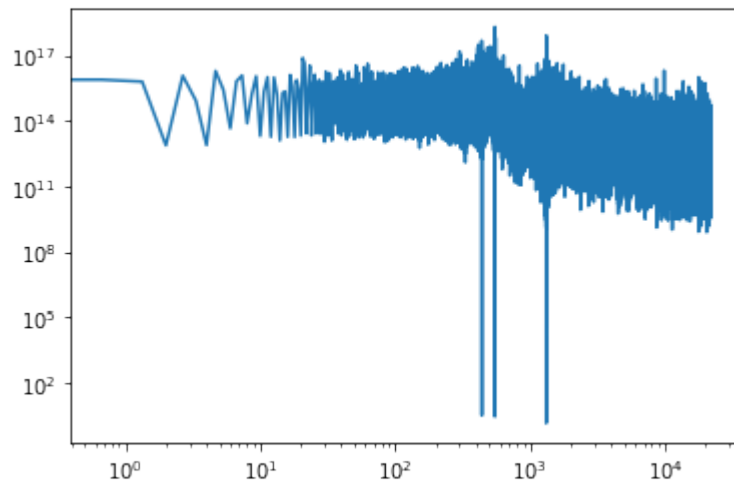
We can also work backwards from the measured signals to obtain the frequency response of the system. Consider that

$$G(s) = \frac{y(s)}{u(s)} \quad \text{so} \quad G(i\omega) = \frac{y(i\omega)}{u(i\omega)}$$

```
[79]: Gfreq = filtered_fft/fft
```

```
[80]: plt.loglog(fftfreq, numpy.abs(Gfreq))
```

```
[80]: [<matplotlib.lines.Line2D at 0x12ad5a630>]
```



The gain is a bit strange (very large) here because we used a pure sinusoidal signal to start with, with no low frequency content, but we can still see the characteristic shape of a first order transfer function.

But that sounds terrible

Pure sinusoids are not in fact all that musical, let's rather use a proper song.

You can convert from MP3 to wav using one of [these techniques](#)

```
[81]: import scipy.io.wavfile
```

I've used "Wedding Day" by SAINT JHN here. I think 7 seconds falls within fair use. You might want to use your own favourite song to hear the differences clearly.

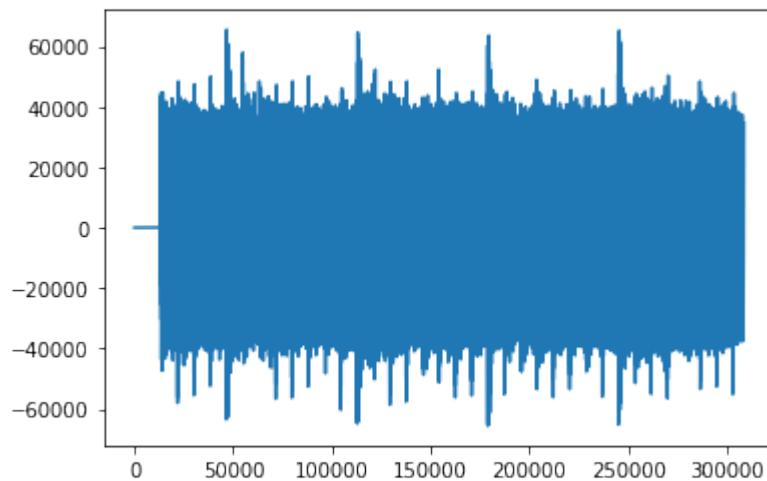
```
[88]: samplingrate, song = scipy.io.wavfile.read('../assets/weddingday.wav')
```

```
[89]: dt = 1/samplingrate
```

```
[90]: samplelength = 7*samplingrate
      songsample = song.sum(axis=1)[:samplelength]
```

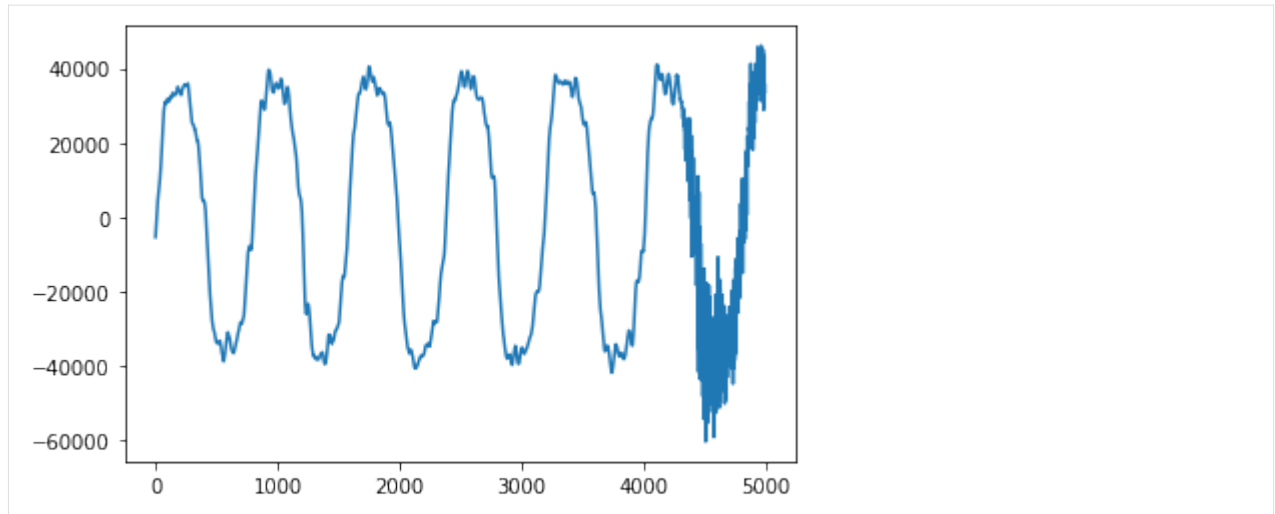
```
[91]: plt.plot(songsample)
```

```
[91]: [<matplotlib.lines.Line2D at 0x1257aa240>]
```



```
[92]: plt.plot(songsample[100000:105000])
```

```
[92]: [<matplotlib.lines.Line2D at 0x12ad78f60>]
```

Let's first hear what it sounds like unfiltered.

```
[99]: songsample[100000:105000]
[99]: array([-5346, -4772, -4268, ..., 35405, 34417, 33684])
```

```
[93]: Audio(songsample, rate=samplingrate)
[93]: <IPython.lib.display.Audio object>
```

Now with one pass through the filter

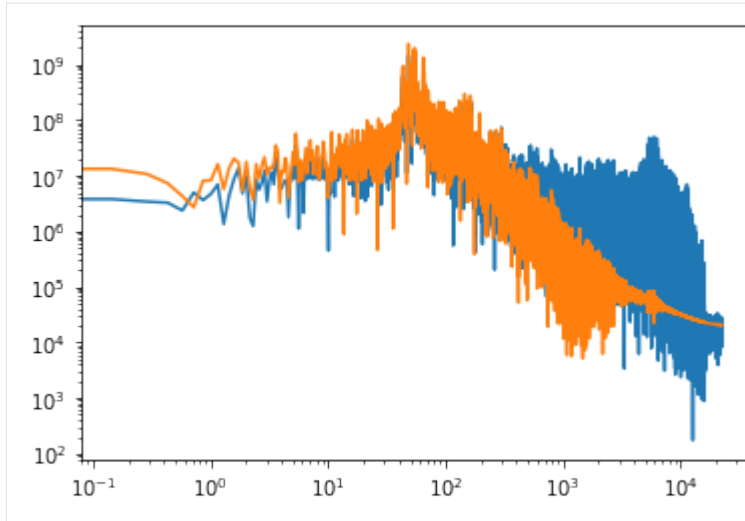
```
[94]: filtered = filtersignal(songsample)
      Audio(filtered, rate=samplingrate)
[94]: <IPython.lib.display.Audio object>
```

We can filter the signal a second time to really hear the high notes fall away.

```
[96]: filtered = filtersignal(filtered)
      Audio(filtered, rate=samplingrate)
[96]: <IPython.lib.display.Audio object>

[97]: songfft = numpy.fft.rfft(songsample)
      filteredfft = numpy.fft.rfft(filtered)
      fftfreq = numpy.fft.rfftfreq(len(songsample), dt)

[98]: bodegain(songfft)
      bodegain(filteredfft)
```

Let's see what the effect of the filter looks like as a function of frequency.

```
[41]: Gfft = filteredfft/songfft
```

```
[42]: omega = numpy.logspace(0, 5, 1000)
```

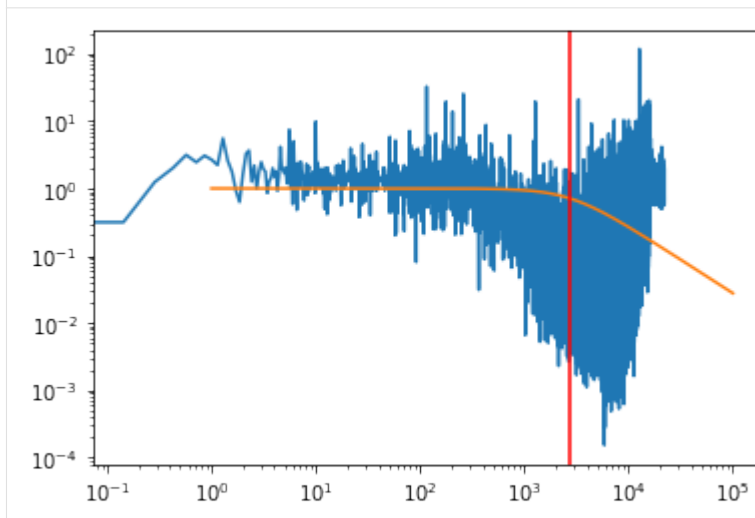
```
[43]: s = 1j*omega
```

```
[44]: Gw = 1/(tau*s + 1)
```

```
[45]: gain = numpy.abs(Gw)
```

```
[46]: plt.loglog(fftfreq, numpy.abs(Gfft))
plt.loglog(omega, gain)
plt.axvline(1/tau, color='red')
```

```
[46]: <matplotlib.lines.Line2D at 0x11a7d5978>
```



2.8.3 Frequency response plots

Frequency responses are very easy to calculate numerically if we remember that the frequency domain is basically the part of the Laplace domain on the imaginary axis, or mathematically $s = i\omega$

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

Frequency responses often make use of logarithmic scales, so we'll generate logarithmically spaced points. This will be linearly spaced on a logarithmic scale:

```
[2]: omega = numpy.logspace(-2, 2, 1000)
```

```
[3]: s = omega*1j
```

It is important to realise that `omega` and `s` are just collections of numbers:

```
[4]: omega[:5]
```

```
[4]: array([0.01          , 0.01009262, 0.0101861 , 0.01028045, 0.01037567])
```

```
[5]: s[:5]
```

```
[5]: array([0.+0.01j          , 0.+0.01009262j, 0.+0.0101861j , 0.+0.01028045j,
          0.+0.01037567j])
```

As an example, we can use a first order transfer function

```
[6]: tau1 = 2
```

```
[7]: G1 = 1/(tau1*s + 1)
```

and a second order with complex poles

```
[8]: tau = 1
```

```
[9]: zeta = 0.5
```

```
[10]: G2 = 1/(tau**2*s**2 + 2*tau*zeta*s + 1)
```

Similarly to `omega` and `s`, `G1` and `G2` are just arrays.

```
[11]: G1[:5]
```

```
[11]: array([0.99960016-0.019992j , 0.99959272-0.02017702j,
          0.99958515-0.02036375j, 0.99957743-0.02055221j,
          0.99956957-0.0207424j  ])
```

```
[12]: G2[:5]
```

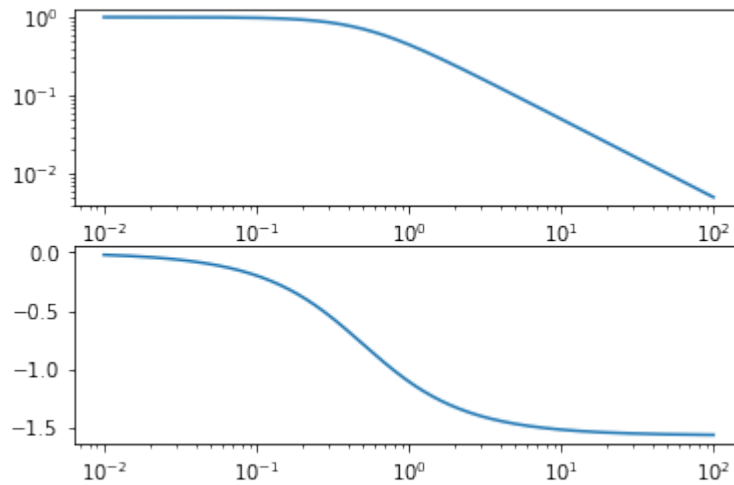
```
[12]: array([0.99999999-0.010001j , 0.99999999-0.01009365j,
          0.99999999-0.01018716j, 0.99999999-0.01028153j,
          0.99999999-0.01037678j])
```


2.8.4 Bode

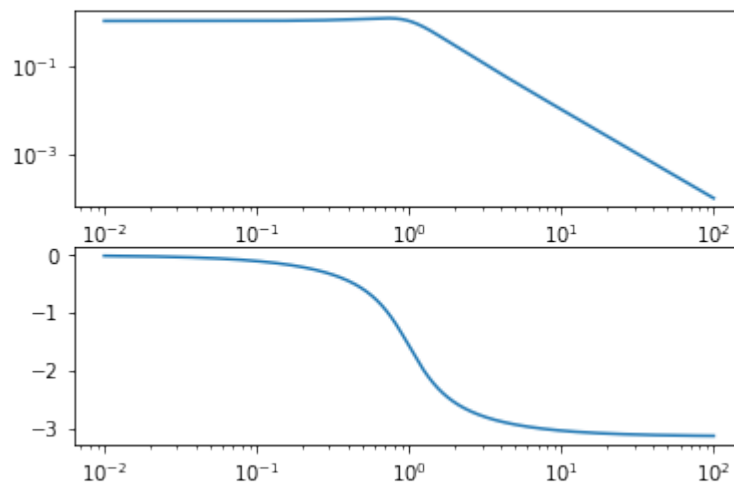
Bode diagrams are the most common plots. The magnitude and angle of the frequency response is shown as a function of frequency. This is such a common representation that when most control engineers say something like “Show me the Frequency response” they will mean “Show me a Bode diagram”.

```
[13]: def bode(G):
    fig, [ax_mag, ax_phase] = plt.subplots(2, 1)
    ax_mag.loglog(omega, numpy.abs(G))
    ax_phase.semilogx(omega, numpy.angle(G))
```

```
[14]: bode(G1)
```

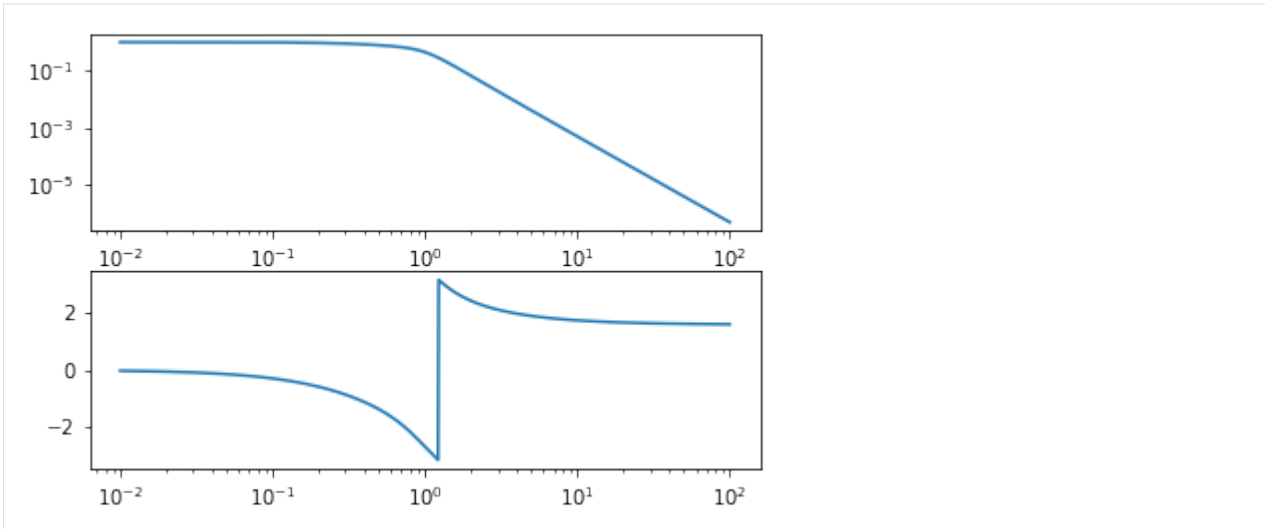


```
[15]: bode(G2)
```



It is easy to predict how the product of the two functions would look since both plots add together when the transfer functions are multiplied.

```
[16]: bode(G1*G2)
```

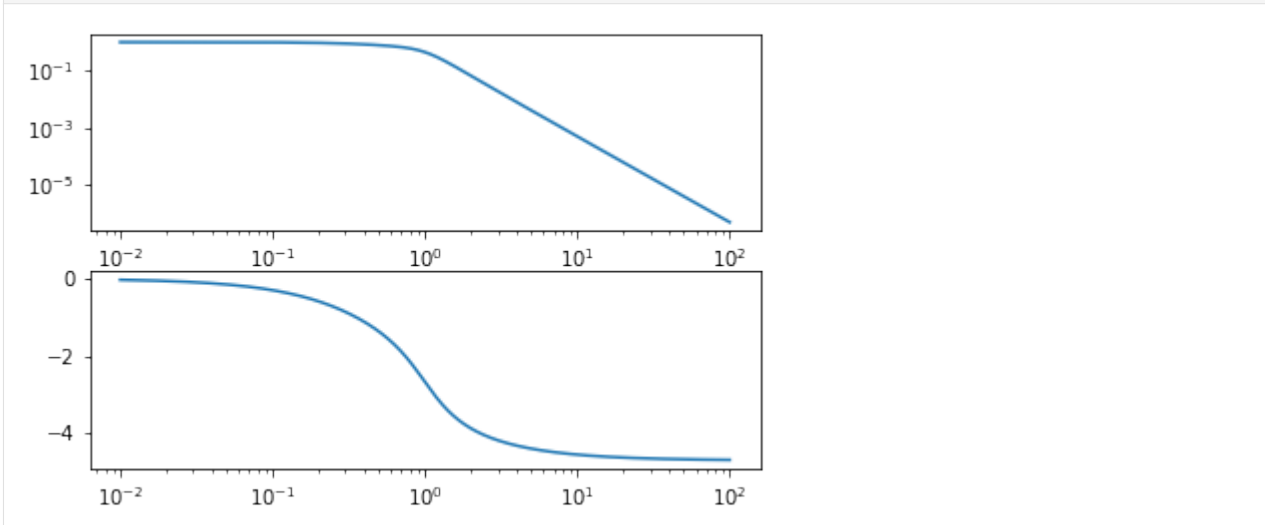



2.8.5 Phase unwrapping

Wait... What happened in that plot? That discontinuity happens when the angle reaches $-\pi$ and starts counting positive angles, jumping to π . To get rid of this, we need to redefine our Bode function to use the function `numpy.unwrap` which removes discontinuities by subtracting an appropriate multiple of 2π .

```
[17]: def bode(G):
      fig, [ax_mag, ax_phase] = plt.subplots(2, 1)
      ax_mag.loglog(omega, numpy.abs(G))
      ax_phase.semilogx(omega, numpy.unwrap(numpy.angle(G)))
```

```
[18]: bode(G1*G2)
```



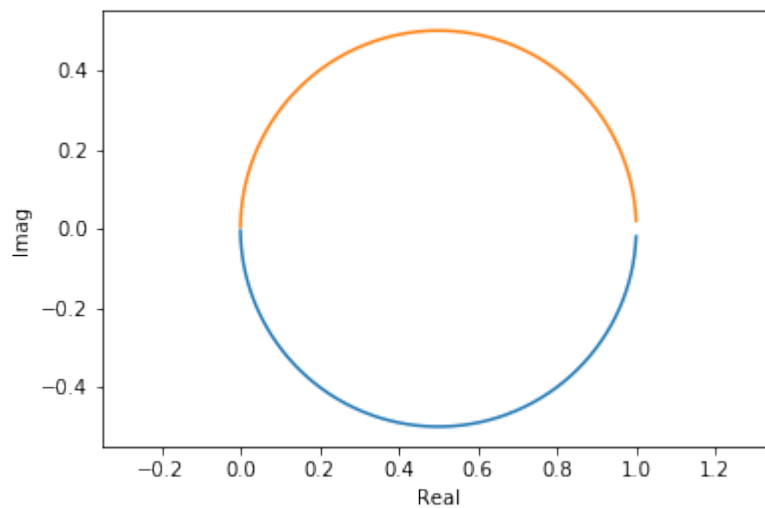
Much better.

2.8.6 Nyquist

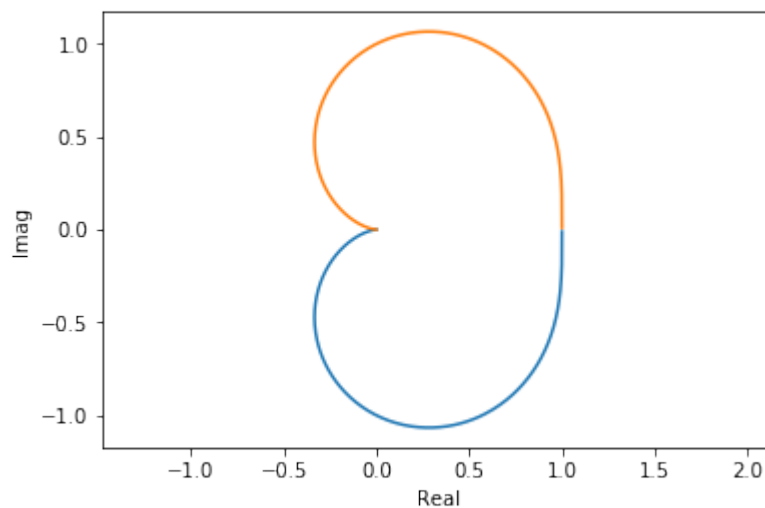
Nyquist diagrams are simply the real and imaginary components of the frequency response plotted on a plane. By convention not only the positive frequencies are plotted but the negative as well. This is the image of $G(s)$ as s traverses the entire imaginary line.

```
[19]: def nyquist(G):
      plt.plot(G.real, G.imag,
               G.real, -G.imag)
      plt.xlabel('Real')
      plt.ylabel('Imag')
      plt.axis('equal')
```

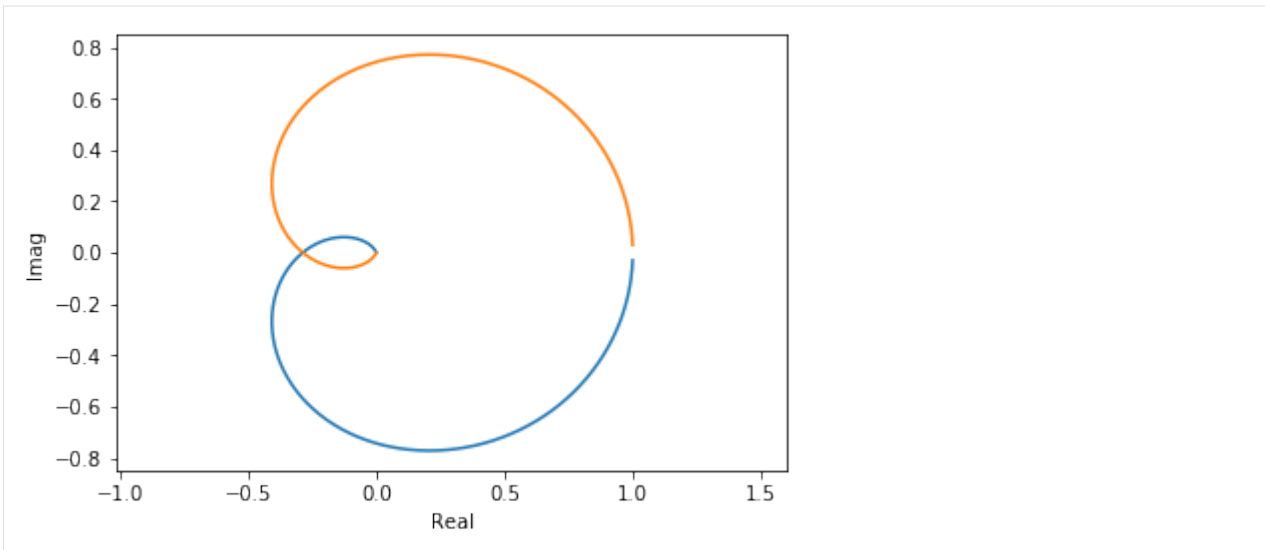
```
[20]: nyquist(G1)
```



```
[21]: nyquist(G2)
```



```
[22]: nyquist(G1*G2)
```

You can see an extra 90° twist for every order below the line.

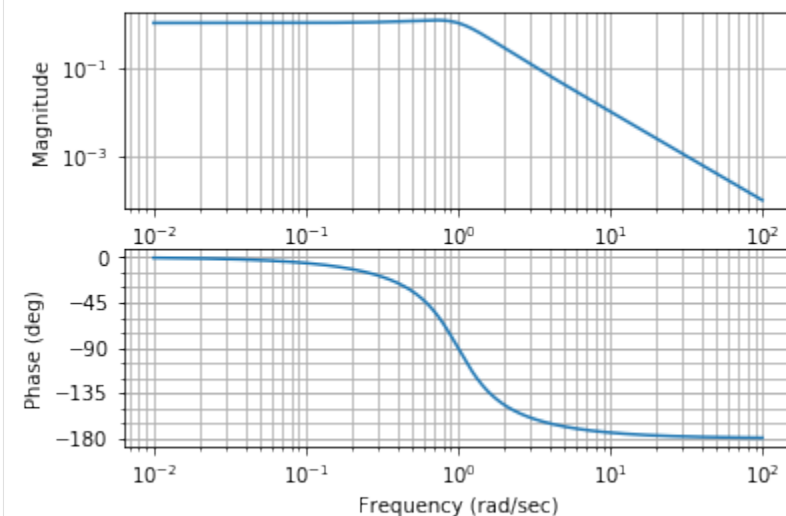
2.8.7 With the control library

The control library saves us some typing to create these diagrams:

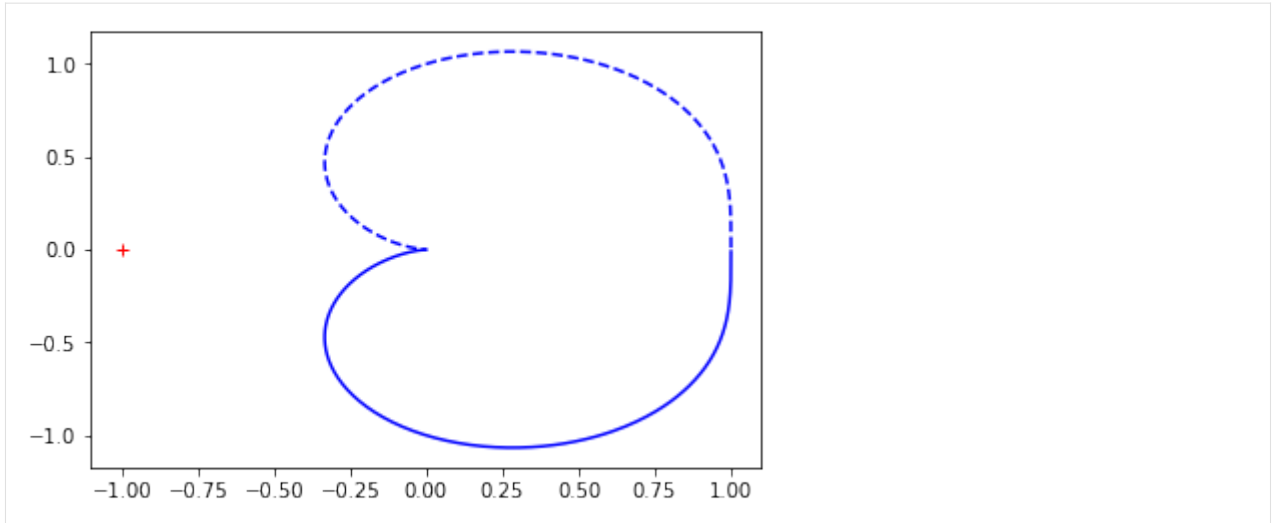
```
[23]: import control
```

```
[24]: G = control.tf(1, [tau**2, 2*tau*zeta, 1])
```

```
[25]: control.bode(G, omega);
```



```
[26]: control.nyquist_plot(G, omega);
```

```
[ ]:
```

2.8.8 Asymptotic Bode diagrams

One of the big advantages of Bode diagrams is that they are very easy to sketch out by hand (or, equivalently, to visualise mentally).

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[2]: omega = numpy.logspace(-2, 2, 1000)
s = 1j*omega
```

2.8.9 Systems with real poles

Let's study the bode diagrams of systems of the form

$$\frac{K}{(\tau s + 1)^n}$$

```
[3]: def annotated_bode(ax_gain, ax_phase, G, K, tau, order):
    high_freq_asymptote = K/(tau*omega)**order

    # Gain part
    ax_gain.loglog(omega, numpy.abs(G))
    ax_gain.axhline(K, color='grey') # Rule 1
    ax_gain.loglog(omega, high_freq_asymptote, color='grey') # Rule 2
    ax_gain.axvline(1/tau, color='grey') # Rule 2
    ax_gain.set_ylim([1e-2, 1e+1])
    ax_gain.set_ylabel('|G|')

    # Phase part
```

(continues on next page)

(continued from previous page)

```

ax_phase.axhline(0, color='grey') # Rule 3
ax_phase.semilogx(omega, numpy.unwrap(numpy.angle(G)))
ax_phase.axhline(-numpy.pi/2*order, color='grey') # Rule 4
ax_phase.axvline(1/tau, color='grey') # Rule 5
ax_phase.set_ylim([-3*numpy.pi/2, 2*numpy.pi/2])
ax_phase.set_ylabel(r'$\angle G$')

```

```

[4]: def plotresponse(order=1, tau=1, K=1):
    G = K/(tau*s + 1)**order

    fig, [ax_gain, ax_phase] = plt.subplots(2, 1)
    annotated_bode(ax_gain, ax_phase, G, K, tau, order)

```

```

[5]: from ipywidgets import interact

```

```

[6]: interact(plotresponse, order=(-2, 3), tau=(0.1, 10), K=(-1., 2));

interactive(children=(IntSlider(value=1, description='order', max=3, min=-2),
↪FloatSlider(value=1.0, descripti...

```

We see that we can construct a reasonable approximation by knowing a couple of things

1. The gain (K) of the system defines the low frequency asymptote of the gain graph
2. The high frequency asymptote of the gain is $\frac{K}{(\omega\tau)^n}$. Effectively, on a loglog scale, this means we have $-n/\text{decade}$ slope above frequencies of around $1/\tau$
3. The low frequency phase asymptote is 0
4. The high frequency phase asymptote is $-n\pi/2$
5. The phase curve has an inflection at $1/\tau$

2.8.10 Systems with complex poles

Systems with complex poles show unique frequency response behaviour. We will focus on the second order system shown below:

$$G = \frac{K}{\tau^2 s^2 + 2\tau\zeta s + 1}$$

```

[7]: def plotresponse(K=1, tau=1, zeta=1):
    plt.figure(figsize=(15, 5))
    order = 2
    G = K/(tau**2*s**2 + 2*tau*zeta*s + 1)

    ax_gain = plt.subplot2grid((2, 2), (0, 0))
    ax_phase = plt.subplot2grid((2, 2), (1, 0))
    ax_complex = plt.subplot2grid((2, 2), (0, 1), rowspan=2)

    annotated_bode(ax_gain, ax_phase, G, K, tau, order)

    # poles
    poles = numpy.roots([tau**2, 2*tau*zeta, 1])
    ax_complex.scatter(poles.real, poles.imag)

```

(continues on next page)

(continued from previous page)

```
ax_complex.axhline(0)
ax_complex.axvline(0)
ax_complex.axis([-2, 2, -2, 2])
```

```
[8]: interact(plotresponse, K=(0.1, 2), tau=(0.1, 2), zeta=(0., 1.1))

interactive(children=(FloatSlider(value=1.0, description='K', max=2.0, min=0.1),
↵FloatSlider(value=1.0, descri...

[8]: <function __main__.plotresponse(K=1, tau=1, zeta=1)>
```

We see that the rules from before still hold, except that we start seeing the so-called “harmonic nose” emerge when $\zeta < \sqrt{2}/2 \approx 0.7$. The maximum of the nose occurs at the resonant frequency of

$$\omega_r = \frac{\sqrt{1 - 2\zeta^2}}{\tau}$$

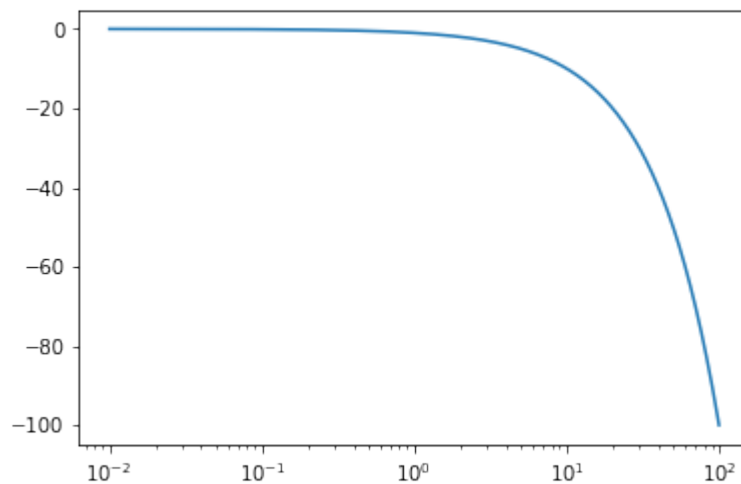
2.8.11 Dead time

The effect of dead time is to increase the phase lag indefinitely as a function of frequency. Delay has no effect on the gain of a system.

```
[9]: D = 1
G = numpy.exp(-D*s)
```

```
[10]: plt.semilogx(omega, numpy.unwrap(numpy.angle(G)))
```

```
[10]: [<matplotlib.lines.Line2D at 0x117a944e0>]
```



2.9 Sampled systems

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[5]: from ipywidgets import interact, Checkbox
```

```
[6]: f = numpy.sin
```

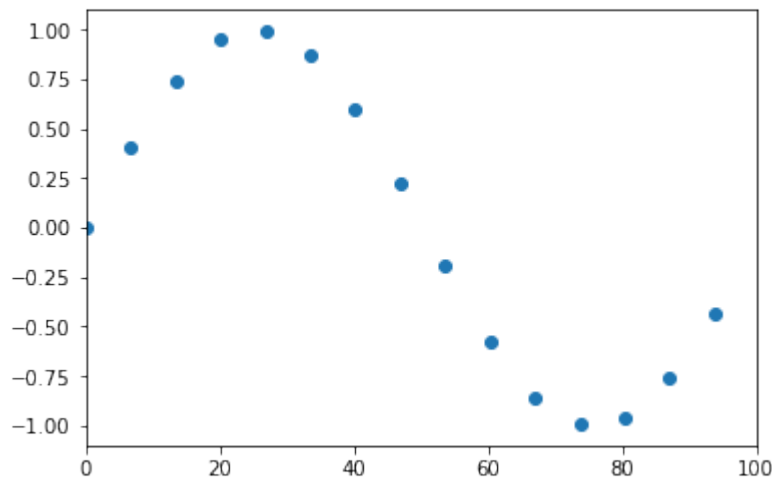
Let's generate a signal and sample it

```
[7]: maxt = 100
t = numpy.linspace(0, maxt, 1000)
y = f(t)
```

```
[8]: def show_sampled(T=6.7, show_f=True):
    t_sampled = numpy.arange(0, maxt, T)
    y_sampled = f(t_sampled)

    if show_f:
        plt.plot(t, y)
    plt.scatter(t_sampled, y_sampled)
    plt.axis([0, maxt, -1.1, 1.1])
```

```
[9]: interact(show_sampled, T=(0.1, 10), show_f=Checkbox());
```



The default sampling rate in the demo above illustrates the idea of *aliasing*, where a higher frequency sinusoid can masquerade as a lower frequency one. We can avoid this problem by ensuring that we sample at least twice per cycle for the highest frequency in the signal we are sampling. See the Wikipedia page on the [Nyquist-Shannon sampling theorem](#) for more information.

```
[1]: import matplotlib.pyplot as plt
%matplotlib inline
```

```
[2]: import numpy
```



```
[3]: from tbcontrol.responses import sopdt
```

2.9.1 Strategies for filtering out noise from a sampled signal

In some cases our measurements have been altered by some kind of noise. Commonly this is “white noise”, which is normally distributed with zero mean.

```
[4]: N = 100
```

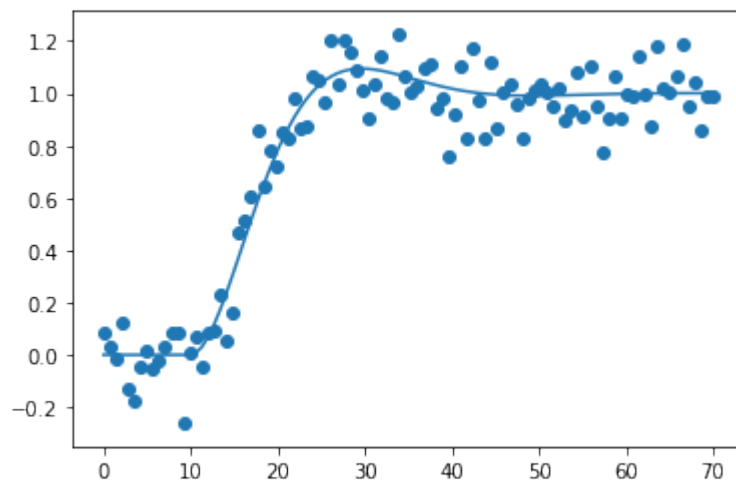
```
[5]: t = numpy.linspace(0, 70, N)
```

```
[6]: y = sopdt(t, K=1, tau=5, zeta=0.6, theta=10)
```

```
[7]: ym = y + numpy.random.randn(N)*0.1
```

```
[8]: plt.scatter(t, ym)
plt.plot(t, y)
```

```
[8]: [<matplotlib.lines.Line2D at 0x11a9b65d0>]
```



Pandas

Pandas includes many common filtering strategies in an easy-to-use package. Let’s get the data into a DataFrame.

```
[9]: import pandas
```

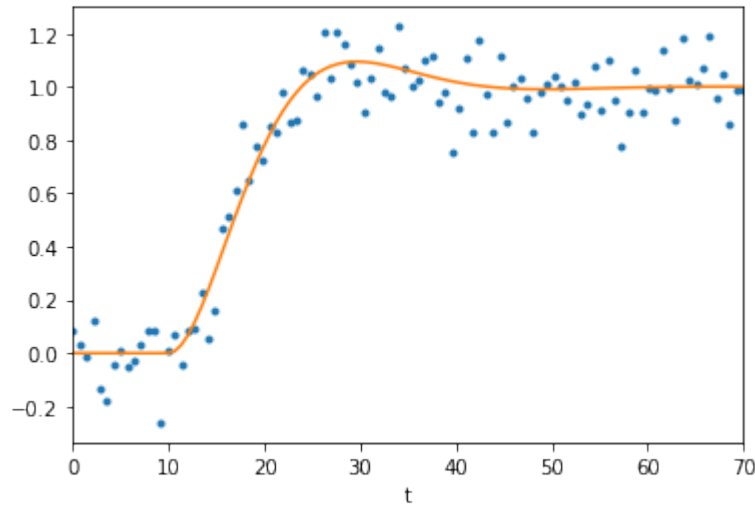
```
[10]: df = pandas.DataFrame({'t': t, 'y': y, 'ym': ym}).set_index('t')
```

```
[11]: def noisy_and_original():
    df['ym'].plot(style='.')
    df['y'].plot()
```

```
[12]: measured = df['ym']
```



```
[13]: noisy_and_original()
```

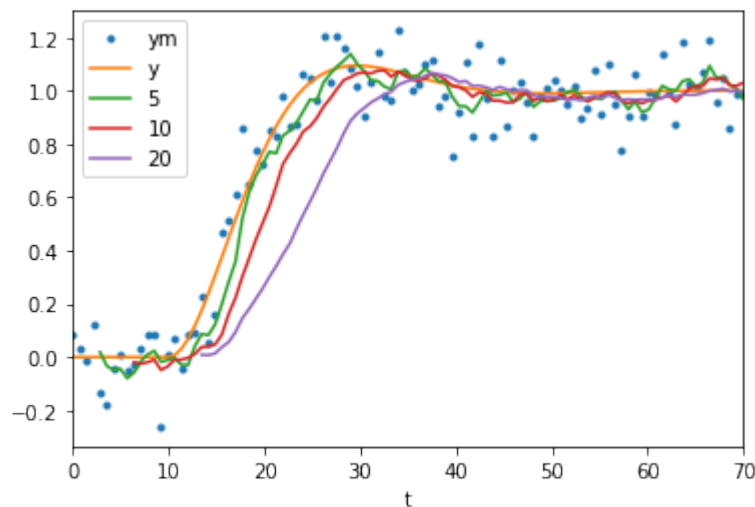


2.9.2 Moving averages

Moving averages are a very common way to filter out noise. The idea is to average together a certain number of samples to get the value of a sample. This operation is common enough that it can be selected as a dropdown option in Excel.

```
[14]: def moving(center=False):
        noisy_and_original()
        for window in [5, 10, 20]:
            measured.rolling(window, center=center).mean().plot(label=window)
        plt.legend()
```

```
moving()
```

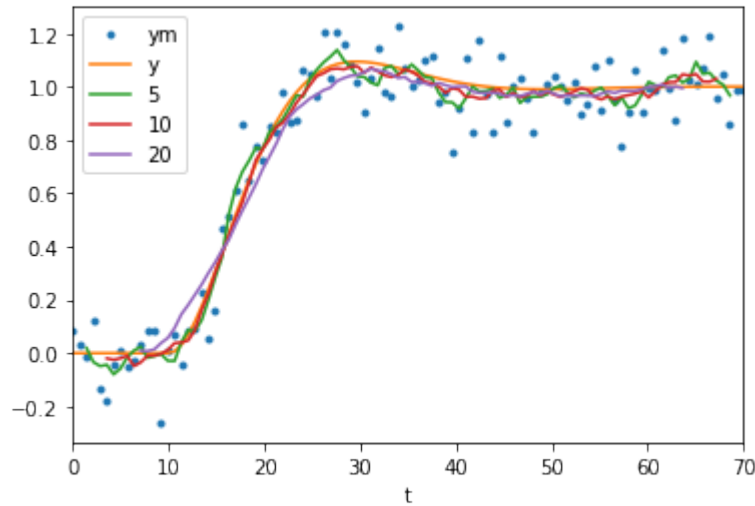


As with all causal filters (filters which only use information from *before* the point at which they calculate a value) we see that the filter introduces a delay between the original signal and the filtered signal.

In Pandas it is easy to get a less delayed result by using a centered moving average (where points before *and* after the

reported time are used).

```
[15]: moving(center=True)
```



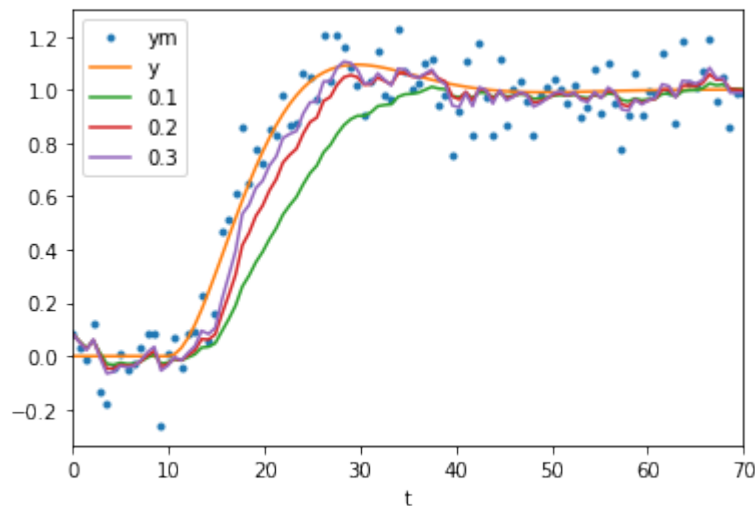
Note that these signals are much closer to the original data. As a general rule, non-causal filters outperform their causal counterparts at the cost of having to be done offline. However, also notice that the wider windows are making the response look less sharp at the start and suppressing the overshoot.

Exponentially weighted moving average

Pandas also includes an easy way to produce exponentially weighted moving averages. These are the digital equivalent of first order analog filters.

```
[16]: noisy_and_original()
      for alpha in [0.1, 0.2, 0.3]:
          measured.ewm(alpha=alpha).mean().plot(label=alpha)
      plt.legend()
```

```
[16]: <matplotlib.legend.Legend at 0x11cc8a2d0>
```



2.9.3 The z -transform

This notebook shows some techniques for dealing with discrete systems analytically using the z transform

```
[1]: import sympy
sympy.init_printing()

[2]: import tbcontrol
tbcontrol.expectversion('0.1.2')

[3]: s, z = sympy.symbols('s, z')
k = sympy.Symbol('k', integer=True)
Dt = sympy.Symbol('\Delta t', positive=True)
```

Definition

The z transform of a sampled signal ($f^*(t)$) is defined as follows:

$$\mathcal{Z}[f^*(t)] = \sum_{k=0}^{\infty} f(k\Delta t)z^{-k}$$

Note The notation is often abused, so you may also encounter $\mathcal{Z}[f(t)]$, which should be interpreted as having the sampling implied $\mathcal{Z}[F(s)]$, which implies that you should first calculate the inverse Laplace and then sample, so something like $\mathcal{Z}[F(s)] = \mathcal{Z}[\mathcal{L}^{-1}[F(s)]] = \mathcal{Z}[f(t)]$ * Seborg et al use $\mathcal{Z}[F(s)]$ to mean the transfer function of $F(s)$ with a sample and zero order hold in front of it, which in these notebooks will be expressed as $\mathcal{Z}[H(s)F(s)]$.

Direct calculation in SymPy

For a unit step, $f(t) = 1$ and we can obtain the z transform as an infinite series as follows:

```
[4]: unitstep = sympy.Sum(1 * z**(-k), (k, 0, sympy.oo))
unitstep

[4]: 
$$\sum_{k=0}^{\infty} z^{-k}$$

```

Sympy can recognise this infinite series as a geometric series, and under certain conditions for convergence, it can find a finite representation:

```
[5]: shortform = unitstep.doit()
shortform

[5]: 
$$\begin{cases} \frac{1}{1-\frac{1}{z}} & \text{for } \frac{1}{|z|} < 1 \\ \sum_{k=0}^{\infty} z^{-k} & \text{otherwise} \end{cases}$$

```

To extract the first case solution, we use `args`:

```
[6]: uz = shortform.args[0][0]
uz

[6]: 
$$\frac{1}{1 - \frac{1}{z}}$$

```

Notice what has happened here: we have taken the infinite series and written it in a compact form. You should always keep in mind that these two forms are equivalent.

Transfer functions from difference equations

For a first order difference equation (the discrete equivalent of a first order differential equation):

$$y(k) + a_1 y(k-1) = b_1 u(k-1)$$

If we interpret z^{-n} as an n time step delay, can write

$$\mathcal{Z}[y(k-n)] = Y(z)z^{-n}$$

This transforms our difference equation to

$$Y(z) + a_1 z^{-1} Y(z) = b_1 z^{-1} U(z)$$

Leading to a discrete transfer function:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{b_1 z^{-1}}{1 + a_1 z^{-1}}$$

Unfortunately, sympy simplifies this expression using positive powers of z

```
[7]: a1, b1 = sympy.symbols('a1, b1')
```

```
[8]: Gz = b1*z**-1/(1 + a1*z**-1)
Gz.cancel()
```

```
[8]: 
$$\frac{b_1}{a_1 + z}$$

```

Since I have not found an easy way to get sympy to report negative powers of z , I find it convenient to define

$$q = z^{-1}$$

```
[9]: q = sympy.symbols('q')
```

```
[10]: def qsubs(fz):
        return fz.subs({z: q**-1})
```

```
[11]: qsubs(Gz)
```

```
[11]: 
$$\frac{b_1 q}{a_1 q + 1}$$

```

Responses and inversion

To find the response of this system to the unit input, we can multiply the input and the transfer function. Note that this is equivalent to convolution of the polynomial coefficients.

```
[12]: yz = Gz*uz
qsubs(yz)
```

```
[12]: 
$$\frac{b_1 q}{(1-q)(a_1 q + 1)}$$

```

Let's evaluate that with numeric values for the coefficients:


```
[13]: K = 2 # The worked version in the textbook uses 2 even though the text says 20
      tau = 1
```

```
[14]: Dt = 1
```

```
[15]: parameters = {a1: -sympy.exp(-Dt/tau),
                    b1: K*(1 - sympy.exp(-Dt/tau))
                    }
      stepresponse = yz.subs(parameters)
```

Remember that the z transform was defined using the values of the sampled signal at the sampling points.

$$\mathcal{Z}[f^*(t)] = \sum_{k=0}^{\infty} f(k\Delta t)z^{-k} = f(0) + f(\Delta t)z^{-1} + f(2\Delta t)z^{-2} + \dots$$

To obtain the values of the response at the sampling points (also called inverting the z transform), we need to expand the polynomial. We can do this using Taylor series. Sympy has a `Poly` class which can extract all the coefficients of the polynomial easily.

```
[16]: N = 10
```

```
[17]: qpoly = sympy.Poly(qsubs(stepresponse).series(q, 0, N).removeO(), q)
      qpoly
```

```
[17]: Poly(1.99975318039183q9 + 1.99932907474419q8 + 1.99817623606889q7 + 1.99504249564667q6 + 1.98652410600183q5 + 1.96336872222253q4 + 1.90042111111111q3 + 1.80000000000000q2 + 1.66666666666667q + 1.5)
```

```
[18]: qpoly.all_coeffs()
```

```
[18]: [1.99975318039183, 1.99932907474419, 1.99817623606889, 1.99504249564667, 1.98652410600183, 1.96336872222253, 1.90042111111111, 1.80000000000000, 1.66666666666667, 1.5]
```

Notice that the coefficients are returned in decreasing orders of q , but we want them in increasing orders to plot them.

```
[19]: responses = list(reversed(qpoly.all_coeffs()))
```

We'll be using this operation quite a lot so there's a nice function in `tbcontrol.symbolic` that does the same thing:

```
[20]: import tbcontrol.symbolic
```

```
[21]: responses = tbcontrol.symbolic.sampledvalues(stepresponse, z, N)
```

```
[22]: import matplotlib.pyplot as plt
      import numpy
```

```
[23]: %matplotlib inline
```

We'll compare the values we obtained above with the step response of a continuous first order system with the same parameters:

```
[24]: sampled_t = Dt*numpy.arange(N)
```

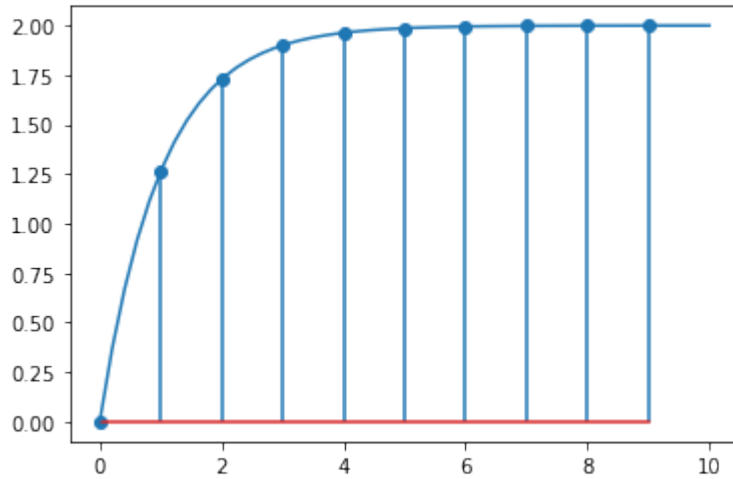
```
[25]: smooth_t = numpy.linspace(0, Dt*N)
```



```
[26]: analytic_firstorder = K*(1 - numpy.exp(-smootht/tau))
```

```
[27]: plt.stem(sampled_t, numpy.array(responses, dtype=float))
plt.plot(smootht, analytic_firstorder)
```

```
[27]: [<matplotlib.lines.Line2D at 0x2a264c57388>]
```



Calculation using scipy

We can get the same values without going through the symbolic steps by using the `scipy.signal` library.

```
[28]: import scipy.signal
```

```
[29]: a1 = -numpy.exp(-Dt/tau)
b1 = K*(1 - numpy.exp(-Dt/tau))
```

Note this uses the transfer function in terms of z (not z^{-1}).

```
[30]: Gz.expand()
```

```
[30]: 
$$\frac{b_1}{a_1 + z}$$

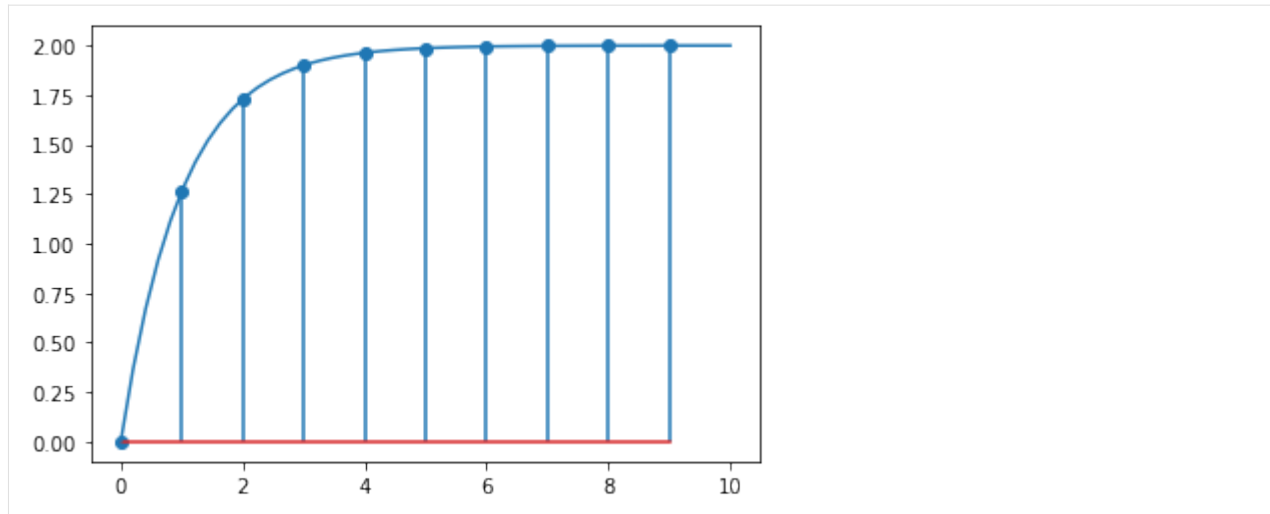
```

```
[31]: Gdiscrete = scipy.signal.dlti(b1, [1, a1], dt=1)
```

```
[32]: _, response = Gdiscrete.step(n=N)
```

```
[33]: plt.stem(sampled_t, numpy.squeeze(response))
plt.plot(smootht, analytic_firstorder)
```

```
[33]: [<matplotlib.lines.Line2D at 0x2a276be5348>]
```

Calculation using the control library

```
[34]: import control
```

Explicitly creating the discrete time transfer function

```
[35]: G = control.tf(b1, [1, a1], 1)
```

```
[36]: G
```

```
[36]:
```

$$\frac{1.264}{z - 0.3679} \quad dt = 1$$

The discrete-time transfer function can also be sampled from the continuous transfer function

```
[54]: G_continuous = control.tf(K, [tau, 1])
```

```
[55]: G_continuous
```

```
[55]:
```

$$\frac{2}{s + 1}$$

```
[56]: G_discrete = G_continuous.sample(1)
```

```
[57]: G_discrete
```

```
[57]:
```

$$\frac{1.264}{z - 0.3679} \quad dt = 1$$

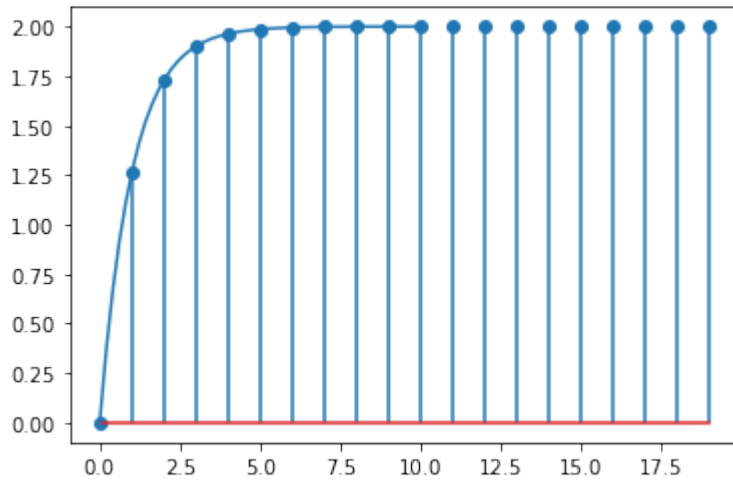
We continue with the explicit transfer function (which is identical to the sampled one)

```
[37]: sampled_t, response = control.step_response(G)
```



```
[38]: plt.stem(sampled_t, numpy.squeeze(response))
      plt.plot(smooth_t, analytic_firstorder)

[38]: [<matplotlib.lines.Line2D at 0x2a276e57188>]
```



I have found it useful to define

```
[39]: z = control.tf([1, 0], 1, 1)
```

```
[40]: z
```

```
[40]: 
$$\frac{z}{1} \quad dt = 1$$

```

This allows us to do calculations in a relatively straightforward way:

```
[41]: step = 1/(1 - z**(-1))
```

```
[42]: step
```

```
[42]: 
$$\frac{z}{z - 1} \quad dt = 1$$

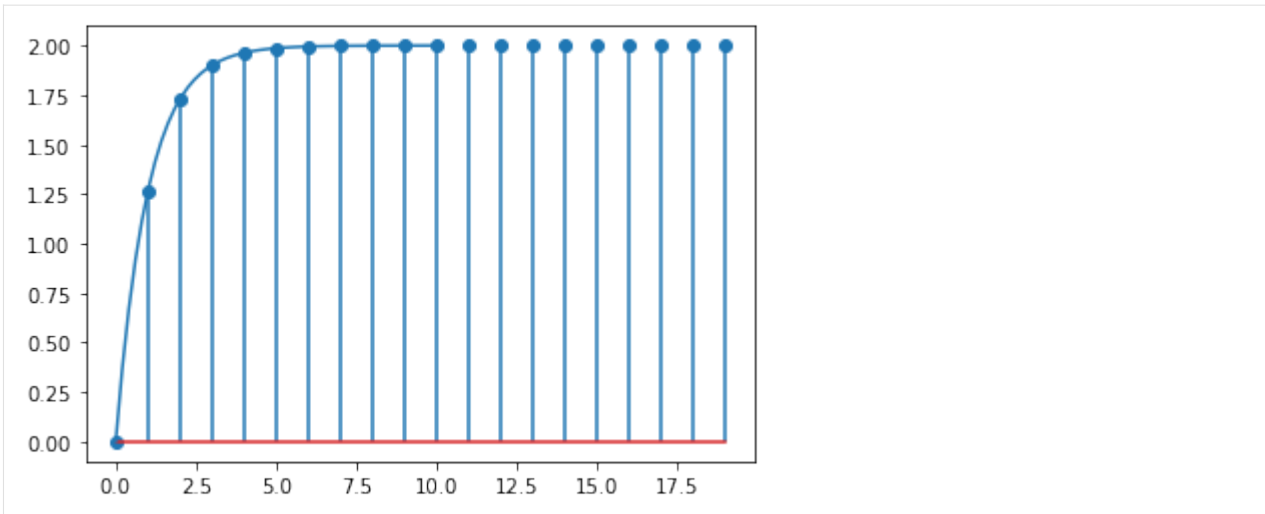
```

If we remember that the inversion of the signal is the same as an impulse response, we can also get the same result as follows:

```
[43]: sampled_t, response = control.impulse_response(G*step)
```

```
[44]: plt.stem(sampled_t, numpy.squeeze(response))
      plt.plot(smooth_t, analytic_firstorder)
```

```
[44]: [<matplotlib.lines.Line2D at 0x2a277119948>]
```

The control library also has the capability of sampling a state space system.

```
[58]: Gss = control.ss(G_continuous)
```

```
[59]: Gss
```

```
[59]: A = [[-1.]]
```

```
B = [[1.]]
```

```
C = [[2.]]
```

```
D = [[0.]]
```

Sampling a continuous state space representation will result in a discrete-time state space representation of the system.

```
[60]: Gss.sample(1)
```

```
[60]: A = [[0.36787944]]
```

```
B = [[0.63212056]]
```

```
C = [[2.]]
```

```
D = [[0.]]
```

```
dt = 1
```

Note that MIMO functionality in the `control` library depends on the `slycot` module. In its current revision MIMO systems cannot be sampled. However you can sample the individual SISO systems from a transfer function matrix, and then build the discrete-time transfer function matrix, which can be converted to a state-space representation.

```
[1]: import sympy
sympy.init_printing()
import matplotlib.pyplot as plt
%matplotlib inline
import tbcontrol
tbcontrol.expectversion('0.1.3')
```

There is a difference between the z transform of an impulse response and the equivalent z transform of a continuous

system with a hold element.

Let's consider the system

$$G(s) = \frac{K}{s + r}$$

```
[2]: s, z, q = sympy.symbols('s, z, q')
      K, r, t = sympy.symbols('K, r, t', real=True)
      Dt = sympy.Symbol(r'\Delta t', positive=True)
```

```
[3]: G = K / (s + r)
```

The *impulse response* of this system is simply the inverse laplace transform:

```
[4]: import tbcontrol.symbolic
```

```
[5]: gt = sympy.inverse_laplace_transform(G, s, t)
      gt
```

```
[5]: 
$$Ke^{-rt}\theta(t)$$

```

The z transform of this function of time, sampled at a sampling rate of Δt can be read off the table as

```
[6]: b = sympy.exp(-r*Dt)
      Gz = K / (1 - b*z**(-1))
```

Let's choose values and plot the response.

```
[7]: parameters = {K: 3, r: 0.25, Dt: 2}
```

```
[8]: import numpy
```

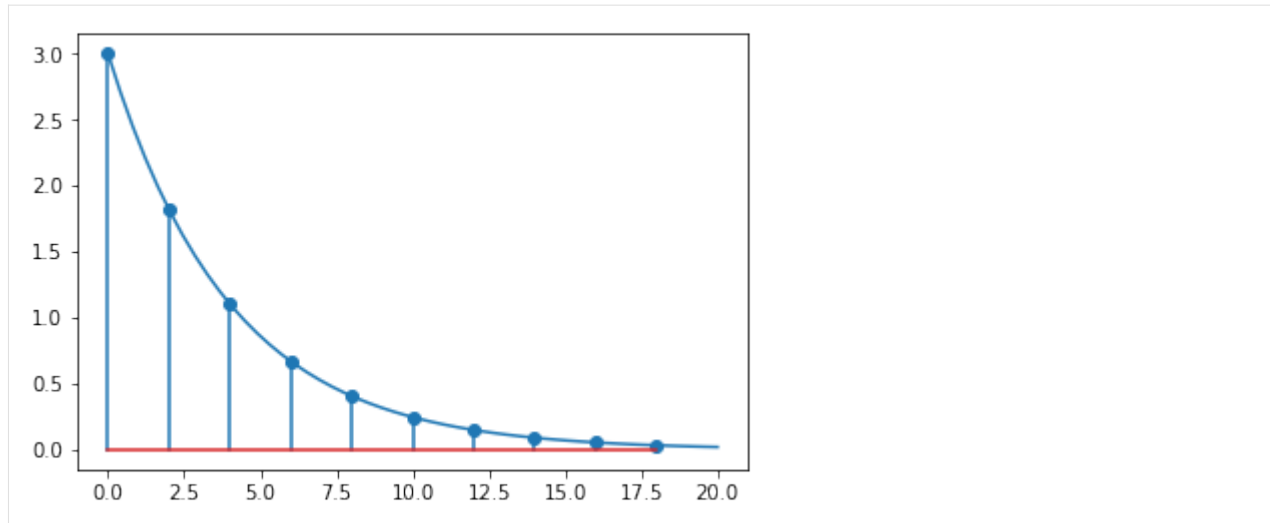
```
[9]: ts = numpy.linspace(0, 20)
```

```
[10]: terms = 10
```

```
[11]: def plot_discrete(Gz, N, Dt):
        ts = [Dt*n for n in range(N)]
        values = tbcontrol.symbolic.sampledvalues(Gz.subs(parameters), z, N)
        plt.stem(ts, values)
```

```
[12]: def values(expression, ts):
        return tbcontrol.symbolic.evaluate_at_times(expression.subs(parameters), t, ts)
```

```
[13]: plt.plot(ts, values(gt, ts))
        plot_discrete(Gz, terms, parameters[Dt])
```

But the value in Table 17.1 is

```
[14]: a1 = -b
      b1 = K/r*(1 - b)
      Gz_seborg = (b1 * z**(-1)) / (1 + a1*z**(-1))
```

```
[15]: Gz_seborg
```

```
[15]:
```

$$\frac{K(1 - e^{-\Delta tr})}{rz \left(1 - \frac{e^{-\Delta tr}}{z}\right)}$$

That's clearly not the same as the discrete transform in the datasheet. What is going on?

The values in the table in seborg are the z transform of the transfer function *with a hold element*!

The z-transform of this combination can be written $\mathcal{Z}\{H(s)G(s)\}$. Remember, $H(s) = \frac{1}{s}(1 - e^{-\Delta ts})$. Now we can show

```

\begin{align}
\mathcal{Z}\{H(s)G(s)\} &= \mathcal{Z}\left\{\frac{1}{s}(1 - e^{-Ts})G(s)\right\} \\
&= \mathcal{Z}\left\{\underbrace{\frac{G(s)}{s}}_{F(s)}(1 - e^{-Ts})\right\} \\
&= \mathcal{Z}\{F(s) - F(s)e^{-Ts}\} \\
&= \mathcal{Z}\{F(s)\} - \mathcal{Z}\{F(s)e^{-Ts}\} = F(z) - F(z)z^{-1} = F(z)(1 - z^{-1})
\end{align}
```

So the z transform we're looking for will be $F(z)(1 - z^{-1})$ with $F(z)$ being the transform on the right of the table of $\frac{1}{s}G(s)$.

To remind ourselves,

```
[16]: G
```

```
[16]:
```

$$\frac{K}{r + s}$$

So we're looking for


```
[17]: (G/s)
```

```
[17]:
```

$$\frac{K}{s(r+s)}$$

So we should see the same response if we plot this:

There is an element in the table for

$$\frac{a}{s(s+a)}$$

which is the same as what we want but multiplied by a . We should be able to use the associated z transform:

```
[18]: a = r
```

```
[19]: table_value = (1 - b)*z**-1/((1 - z**-1)*(1 - b*z**-1))
```

```
[20]: Fz = K * table_value / a
```

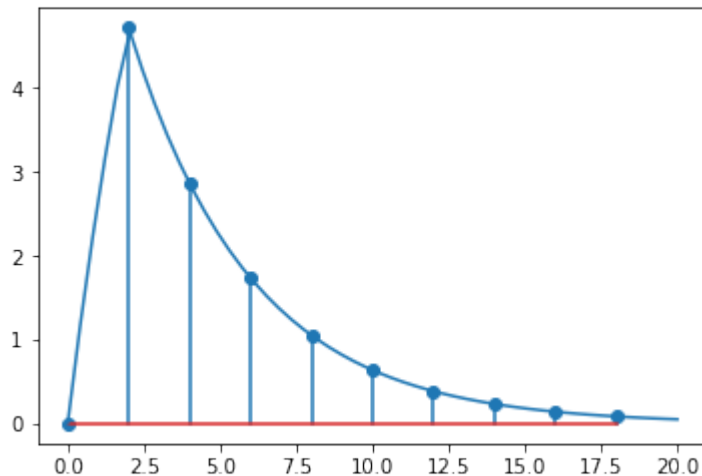
```
[21]: Fz * (1 - z**-1) == Gz_seborg
```

```
[21]: True
```

```
[22]: response = sympy.inverse_laplace_transform(G/s, s, t)
```

```
[23]: plot_discrete(Gz_seborg, terms, parameters[Dt])
      plt.plot(ts, values(response - response.subs(t, t-parameters[Dt])), ts))
```

```
[23]: [<matplotlib.lines.Line2D at 0x1184bacf8>]
```



```
[ ]:
```


3.1 Conventional feedback control

```
[1]: import matplotlib.pyplot as plt
    %matplotlib inline

[2]: from controlgame import ControlGame

[3]: game = ControlGame(runtime=30) # seconds
```

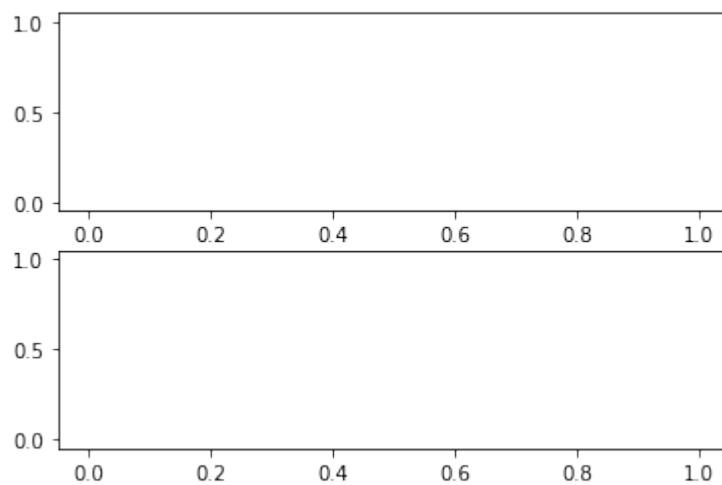
3.1.1 Instructions

Run the cell below and click the “run” button. Then move the “MV” slider in a way which gets the controlled slider close to the setpoint. Your score increases more quickly when Controlled is near Setpoint. See how high your score can get by clicking run a couple of times. To see your performance graphed out, execute the next cell (`game.plot()`)

```
[4]: game.ui()

VBox(children=(HBox(children=(Button(description='Run', style=ButtonStyle()),
↵Text(value='0', description='Sco...
```

```
[5]: game.plot()
```




```
[6]: import scipy.signal

[7]: ts = game.ts

[8]: G = scipy.signal.lti(2, [2, 0])

[9]: import numpy

[10]: LIMIT = 100

[11]: def score(ts, sps, cvs):
    scores = 1 - numpy.minimum(numpy.abs(numpy.array(sps) - numpy.array(cvs)), LIMIT) /
    ↪LIMIT

    score = sum(scores)

    return score

[12]: def sim(ts, mvs):
    _, cvs, _ = scipy.signal.lsim(G, mvs, ts)

    return cvs

[13]: def objective(mvs):
    return -score(game.ts, game.sps, sim(game.ts, mvs))

[15]: objective(game.mvs)

-----
IndexError                                Traceback (most recent call last)
<ipython-input-15-c61c532d40ad> in <module>
----> 1 objective(game.mvs)

<ipython-input-13-96e53751337c> in objective(mvs)
      1 def objective(mvs):
----> 2     return -score(game.ts, game.sps, sim(game.ts, mvs))

<ipython-input-12-d6eb717acf80> in sim(ts, mvs)
      1 def sim(ts, mvs):
----> 2     _, cvs, _ = scipy.signal.lsim(G, mvs, ts)
      3
      4     return cvs

~/anaconda3/lib/python3.7/site-packages/scipy/signal/ltisys.py in lsim(system, U, T, X0, interp)
    1944     xout = zeros((n_steps, n_states), sys.A.dtype)
    1945
-> 1946     if T[0] == 0:
    1947         xout[0] = X0
    1948     elif T[0] > 0:

IndexError: index 0 is out of bounds for axis 0 with size 0

[ ]:
```



```
[ ]: import scipy.optimize

[ ]: guesses = 1

[ ]: bestmvs = game.mvs
    for i in range(guesses):
        sol = scipy.optimize.minimize(objective, bestmvs + 2*(numpy.random.
        ↪rand(len(bestmvs))*2-1), bounds=[(-LIMIT, LIMIT)]*len(game.mvs))
        print('Score:', -sol.fun)
        bestmvs = sol.x
        bestmvs[numpy.abs(bestmvs)<10] = 0

[ ]: bestcvs = sim(ts, bestmvs)

[ ]: fig, (axmv, axcv) = plt.subplots(2, 1)
    axmv.plot(ts, bestmvs)
    axcv.plot(ts, game.sps, ts, bestcvs)
```

3.1.2 PID step responses

Here are some open loop step responses of PID controllers in different configurations.

PI

```
[1]: import control
    import numpy
    import matplotlib.pyplot as plt
    %matplotlib inline

[2]: s = control.tf([1, 0], 1)
    ts = numpy.linspace(0, 5)

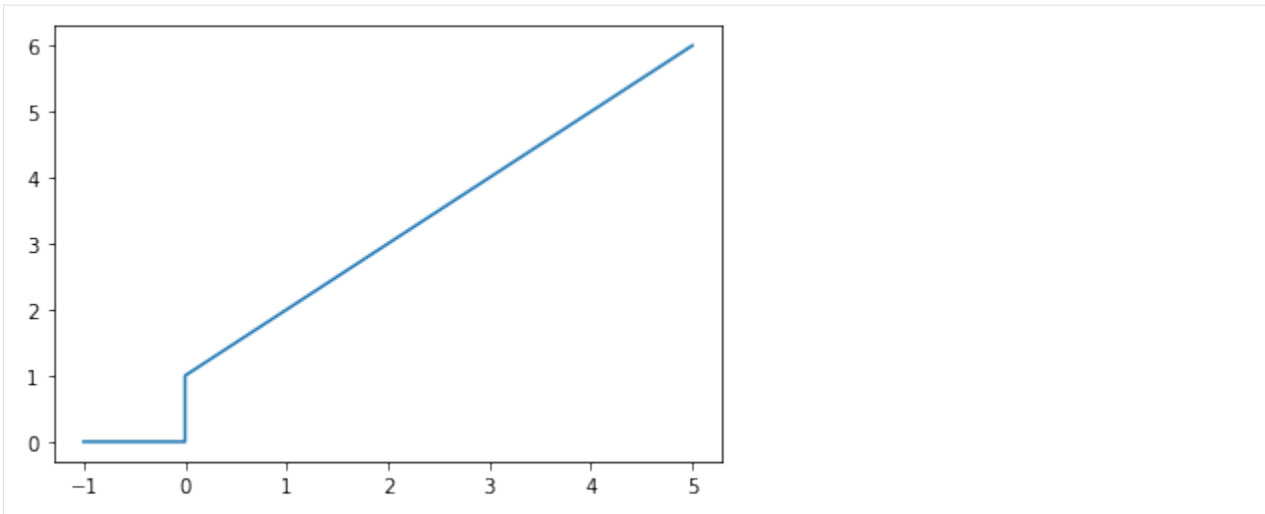
[3]: def plot_step_response(G):
    t, y = control.step_response(G, ts)
    # Add some action before time zero so that the initial step is visible
    t = numpy.concatenate([[-1, 0], t])
    y = numpy.concatenate([[0, 0], y])
    plt.plot(t, y)

[4]: K_C = 1

[5]: tau_I = 1

[6]: Gc = K_C*(1 + 1/(tau_I*s))

[7]: plot_step_response(Gc)
```

PID

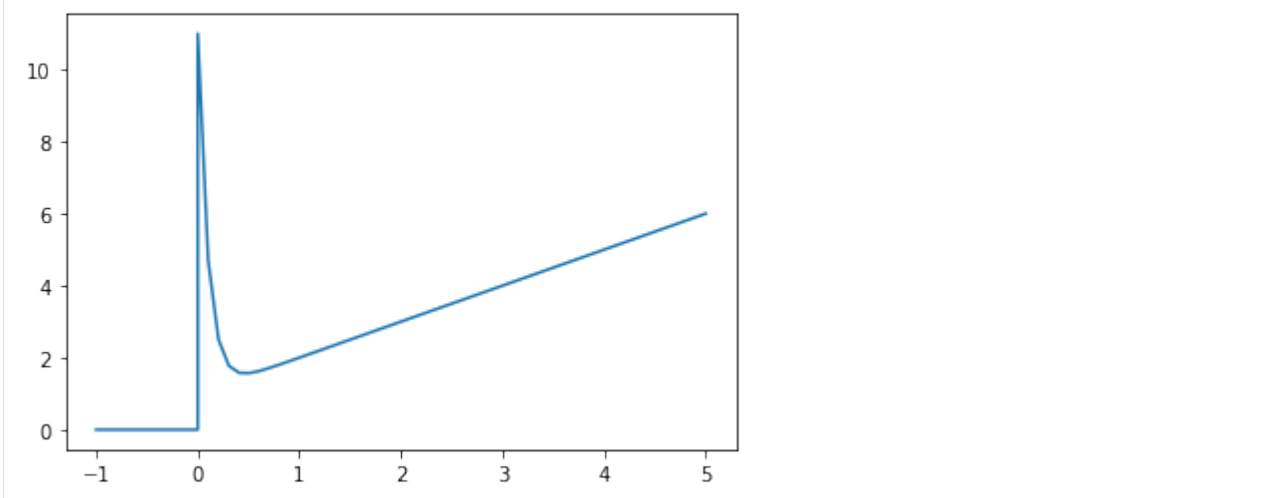
Because the ideal PID is unrealisable, we can't plot the response of the ideal PID, but we can do it for the realisable ISA PID.

$$G_c = K_C \left(1 + \frac{1}{\tau_I s} + \frac{\tau_D s}{\alpha \tau_D s + 1} \right)$$

```
[13]: alpha = 0.1  
      tau_D = 1
```

```
[14]: Gc = K_C*(1 + 1/(tau_I*s) + 1*s/(alpha*tau_D*s + 1))
```

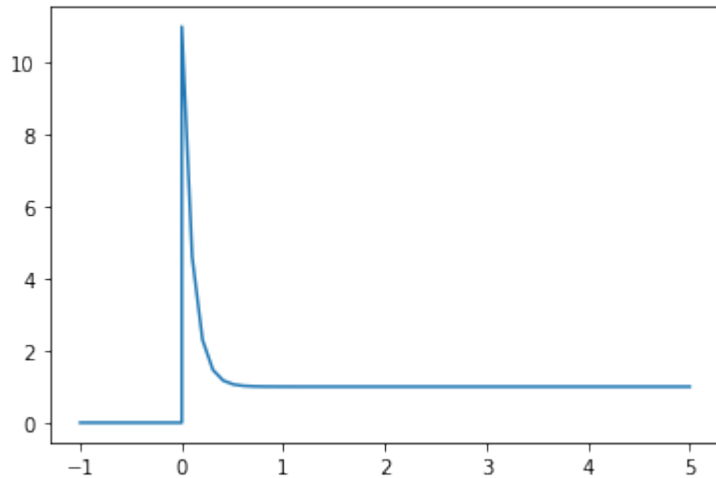
```
[15]: plot_step_response(Gc)
```



PD

```
[16]: Gc = K_C*(1 + 1*s/(alpha*tau_D*s + 1))
```

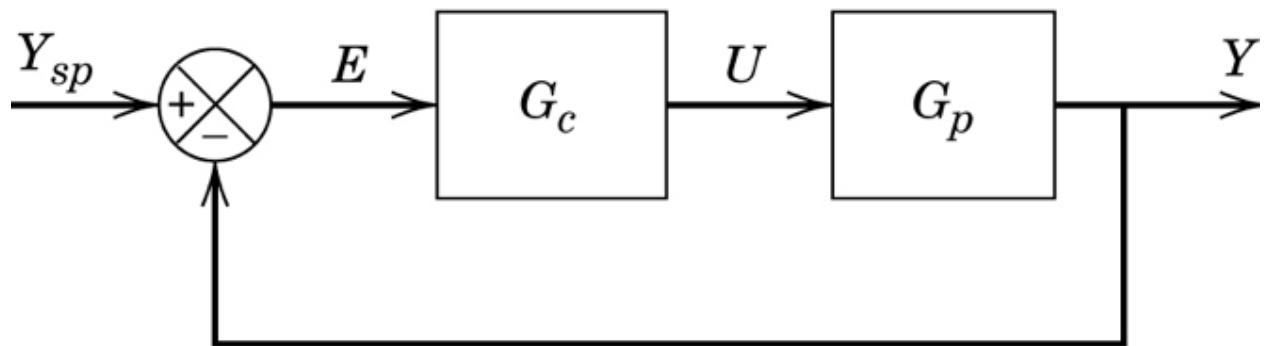
```
[17]: plot_step_response(Gc)
```



```
[ ]:
```

3.1.3 First-order system with proportional control

Consider the simple feedback loop shown below



with $G_c = K_c$ and $G_p = \frac{1}{\tau s + 1}$

```
[1]: %matplotlib inline
```

```
[2]: import sympy
sympy.init_printing()
```

```
[3]: G_c = K_C = sympy.Symbol('K_C', positive=True)
```



```
[4]: s = sympy.Symbol('s')
tau = sympy.Symbol('tau', positive=True)
```

```
[5]: G_p = 1/(tau*s + 1)
G_p
```

```
[5]:
```

$$\frac{1}{s\tau + 1}$$

```
[6]: G_OL = G_p*G_c
```

```
[7]: from tbcontrol.loops import feedback
```

The target is to get $y_{SP} = y$

```
[8]: G_CL = feedback(G_OL, 1).cancel()
G_CL
```

```
[8]:
```

$$\frac{K_C}{K_C + s\tau + 1}$$

```
[9]: t = sympy.Symbol('t', positive=True)
```

```
[10]: general_timresponse = sympy.inverse_laplace_transform(sympy.simplify(G_CL/s), s, t)
general_timresponse
```

```
[10]:
```

$$\frac{K_C \left(e^{\frac{t(K_C+1)}{\tau}} - 1 \right) e^{-\frac{t(K_C+1)}{\tau}}}{K_C + 1}$$

```
[11]: import numpy
```

```
[12]: import matplotlib.pyplot as plt
```

```
[13]: y_func = sympy.lambdify((K_C, tau, t), general_timresponse, 'numpy')
```

```
[14]: smootht = numpy.linspace(0, 5)
```

```
[15]: def response(K_C=10, tau=10):
    y = y_func(K_C, tau, smootht)
    e = 1 - y
    fig, [ax_y, ax_e] = plt.subplots(2, 1)
    ax_y.plot(smootht, y)
    ax_y.axhline(1)
    ax_y.set_ylabel('Setpoint and y')

    ax_e.plot(smootht, e)
    ax_e.set_ylabel('Error')
```



```
[16]: from ipywidgets import interact
```

```
[17]: interact(response, K_C=(0, 100), tau=(0, 20))
```

```
interactive(children=(IntSlider(value=10, description='K_C'), IntSlider(value=10,
↪description='tau', max=20), ...
```

```
[17]: <function __main__.response(K_C=10, tau=10)>
```

Offset as function of gain

```
[18]: r = 1/s
```

```
[19]: y = r*G_CL
```

```
[20]: e = r - y
```

Use the final value statement to obtain eventual offset:

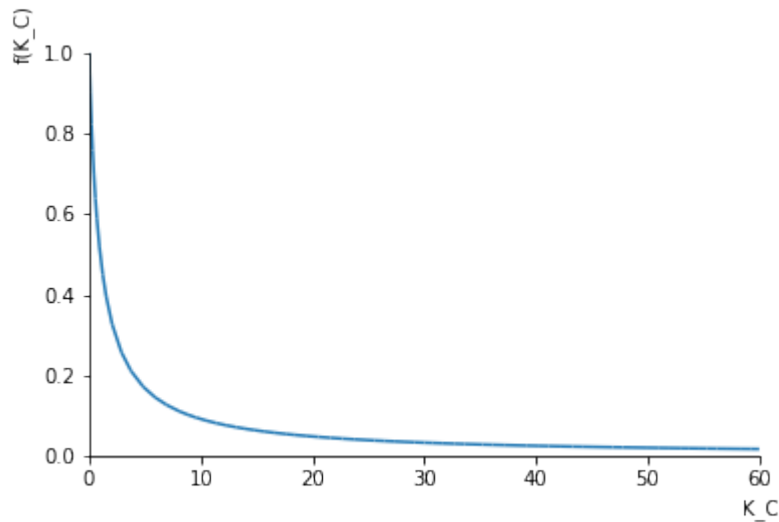
```
[21]: steady_offset = sympy.limit(s*e, s, 0)
steady_offset
```

```
[21]:
```

$$\frac{1}{K_C + 1}$$

Note the steady state offset is not a function of the system dynamics (time constant).

```
[22]: sympy.plot(steady_offset, (K_C, 0, 60))
```



```
[22]: <sympy.plotting.plot.Plot at 0x1141865f8>
```


Second order system with proportional control

```
[23]: import matplotlib.pyplot as plt
```

```
[24]: zeta = sympy.Symbol('zeta')
```

```
[25]: G = 1/(tau**2*s**2 + 2*tau*zeta*s + 1)
G
```

```
[25]:
```

$$\frac{1}{s^2\tau^2 + 2s\tau\zeta + 1}$$

```
[26]: G_CL = feedback(G*K_C, 1).cancel()
G_CL
```

```
[26]:
```

$$\frac{K_C}{K_C + s^2\tau^2 + 2s\tau\zeta + 1}$$

```
[27]: def response(new_K_C, new_tau, new_zeta):
    real_CL = G_CL.subs({K_C: new_K_C, tau: new_tau, zeta: new_zeta})
    timeresponse = sympy.inverse_laplace_transform(sympy.simplify(real_CL/s), s, t)
    sympy.plot(timeresponse, 1, (t, 0, 100))
    poles = sympy.solve(sympy.denom(sympy.simplify(real_CL)), s)
    plt.plot([sympy.re(p) for p in poles], [sympy.im(p) for p in poles], 'x',
    ↪markersize=10)
    plt.axhline(0, color='black')
    plt.axvline(0, color='black')
    plt.axis([-1, 1, -1, 1])
```

```
[28]: interact(response, new_K_C=(0., 100), new_tau=(0, 10.), new_zeta=(0, 2.));
```

```
interactive(children=(FloatSlider(value=50.0, description='new_K_C'),
    ↪FloatSlider(value=5.0, description='new_...
```

```
[ ]:
```

3.1.4 PID control on TCLab

This notebook and the associated `pidgui.py` allows you to play with a very basic position form discrete PID using either a modelled version or the real Temperature control lab.

```
[1]: from tclab.gui import NotebookUI
    from pidgui import PIDGUI
```

```
[2]: %matplotlib notebook
```

```
[3]: interface = NotebookUI(PIDGUI)
```

```
[4]: interface.gui
```



```

VBox(children=(HBox(children=(HBox(children=(Checkbox(value=False, description='Use_
↳model'), FloatSlider(value=1.0, description='Speedup', disabled=True, max=10.0,
↳min=1.0))), HBox(children=(Button(description='Connect', style=ButtonStyle()),
↳Button(description='Start', disabled=True, style=ButtonStyle()), Button(description=
↳'Stop', disabled=True, style=ButtonStyle()), Button(description='Disconnect',
↳disabled=True, style=ButtonStyle())))), HBox(children=(HBox(children=(Label(value=
↳'Timestamp:'), Label(value='No data'), Label(value=''))),
↳HBox(children=(Label(value='Session:'), Label(value='No data'), Label(value='')))),
↳ VBox(children=(HBox(children=(Button(description='Auto', disabled=True,
↳style=ButtonStyle()), Button(description='Manual', disabled=True,
↳style=ButtonStyle()))), HBox(children=(FloatSlider(value=1.0, description='Gain',
↳disabled=True), FloatSlider(value=100.0, description='$\\tau_I$', disabled=True),
↳FloatSlider(value=0.0, description='$\\tau_D$', disabled=True, max=10.0))),
↳HBox(children=(FloatSlider(value=30.0, description='Setpoint', disabled=True,
↳max=70.0, min=20.0), FloatSlider(value=0.0, description='Q1', disabled=True))))))
TCLab version 0.4.9dev
Simulated TCLab

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```

3.1.5 Programmatic interaction

We can interact with the interface while it is running. The controller is in the interface. **Note that you will have to start the controller and switch to auto to see the effect of these cells:**

```
[5]: controller = interface.controller
```

```
[6]: controller.pid.eint = 0
```

```
[ ]: controller.setpoint.value = 42
```

3.1.6 Advanced usage

Below we set up an experiment which will change the setpoint and increase the gain every 10 minutes.

```

[ ]: import tornado

def stepgain():
    if controller.setpoint.value == 45:
        controller.setpoint.value = 40
    else:
        controller.setpoint.value = 45
        controller.gain.value *= 1.1
    if controller.gain.value > 100:
        steptimer.stop()
        interface.action_stop(None)

minute = 60*1000 # a minute in milliseconds
steptimer = tornado.ioloop.PeriodicCallback(stepgain, 10*minute)

steptimer.start()

```


You can stop the timer by calling `.stop()`

```
[ ]: steptimer.stop()
```

3.1.7 Accessing the historian

The interface contains a historian. You can see the sessions it has stored like this:

```
[ ]: interface.historian.get_sessions()
```

You can roll the historian back to a session by using `load_session`. Note you shouldn't do this while the interface is connected.

```
[ ]: interface.historian.load_session(1)
```

3.1.8 More detailed analysis

We can analyse the results of the experiments we have made using Pandas:

```
[ ]: import pandas
```

```
[ ]: allresults = pandas.DataFrame.from_records(interface.historian.log, columns=interface.  
↪ historian.columns, index='Time')
```

3.1.9 Closed loop controlled responses

I discuss drawing these response qualitatively in [this video](#). Note that the responses that are drawn in the video match the responses drawn here, but the value of τ_I specified there doesn't show overshoot as in this notebook. The value in the video is $\tau_I = 10$, but that was obviously a bit too large to allow for the extreme oscillation. You can get a similar response to the one I sketched by using $\tau_I = 1$. The value below has been chosen to make the discussion in the video still hold in the same way.

```
[1]: from tbcontrol.loops import feedback  
import control  
import matplotlib.pyplot as plt  
import numpy  
%matplotlib inline
```

```
[2]: s = control.tf([1, 0], 1)
```

```
[8]: Gp = 1/(10*s + 1)  
PI = 5*(1 + 1/(10*s))  
P = 5
```

```
[9]: ts = numpy.linspace(0, 30)
```

Find the time where the error becomes zero by interpolating on the output response

```
[10]: t, y = control.step_response(feedback(PI*Gp, 1))  
errorzero = numpy.interp(1, y, t)
```


This function will plot a response for us

```
[11]: def plotresponse(ax, G, *args, **kwargs):
        ax.plot(*control.step_response(G, T=ts), *args, **kwargs)
        ax.axvline(errorzero, color='teal', linestyle='--')
```

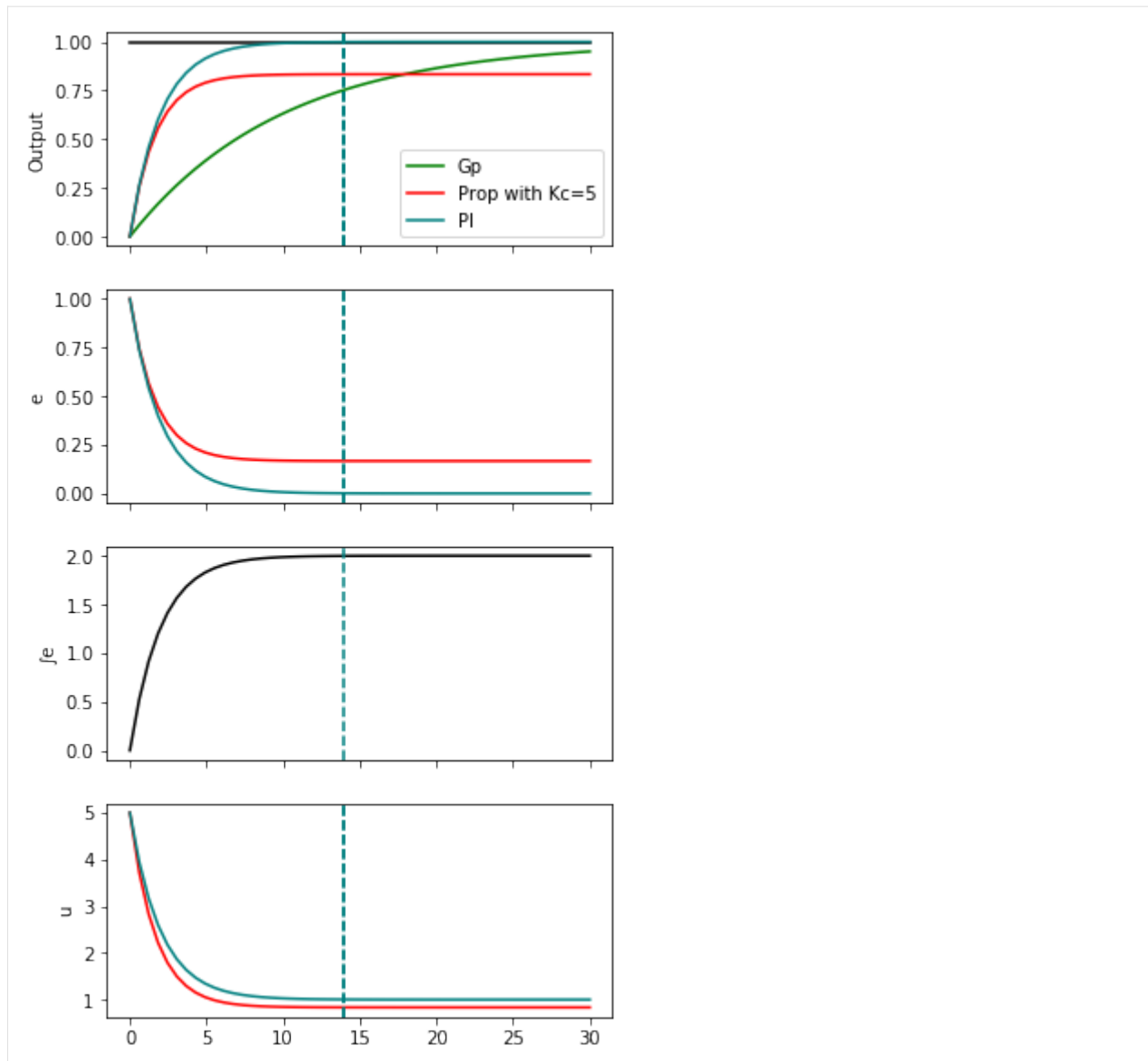
I'm trying to get all the colors to match the video here.

```
[12]: fig, (outputs, errors, errorint, u) = plt.subplots(4, 1, figsize=(5, 10), sharex=True)
        outputs.plot(ts, numpy.ones_like(ts), color='black')
        plotresponse(outputs, Gp, color='green', label='Gp')
        plotresponse(outputs, feedback(P*Gp, 1), color='red', label='Prop with Kc=5')
        plotresponse(outputs, feedback(PI*Gp, 1), color='teal', label='PI')
        outputs.set_ylabel('Output')
        outputs.legend()

        plotresponse(errors, 1 - feedback(P*Gp, 1), color='red')
        plotresponse(errors, 1 - feedback(PI*Gp, 1), color='teal')
        errors.set_ylabel('e')

        plotresponse(u, feedback(P, Gp), color='red')
        plotresponse(u, feedback(PI, Gp), color='teal')
        u.set_ylabel('u')

        plotresponse(errorint, (1 - feedback(PI*Gp, 1))/s, color='black')
        errorint.set_ylabel('∫e');
```

[]:

[]:

[]:

3.2 Laplace domain analysis of control systems

3.2.1 Closed loop stability

Stability of closed loop control systems appears to be easy to determine. We can just calculate the closed loop transfer function and invert the Laplace transform.

```
[1]: import sympy
sympy.init_printing()

[2]: %matplotlib inline

[3]: s = sympy.Symbol('s')

[4]: K_c, t = sympy.symbols('K_c, t', positive=True)
```

These are the systems from Example 10.4 in Seborg et al

```
[5]: G_c = K_c
G_v = 1/(2*s + 1)
G_p = G_d = 1/(5*s + 1)
G_m = 1/(s + 1)

[6]: K_m = sympy.limit(G_m, s, 0)

[7]: forward = G_c*G_v*G_p
backward = G_m

G_CL = K_m*forward/(1 + forward*backward)
```

```
[8]: sympy.simplify(G_CL)
```

```
[8]:
```

$$\frac{K_c (s + 1)}{K_c + (s + 1) (2s + 1) (5s + 1)}$$

```
[9]: y = G_CL/s
```

Now for the moment of truth. Uncomment the next line if you have a lot of time on your hands...

```
[10]: #yt = sympy.inverse_laplace_transform(y, s, t)
```

So that didn't really work as we expected. Can we at least calculate the roots of the denominator?

```
[11]: ce = sympy.denom(G_CL.simplify())
ce.expand()
```

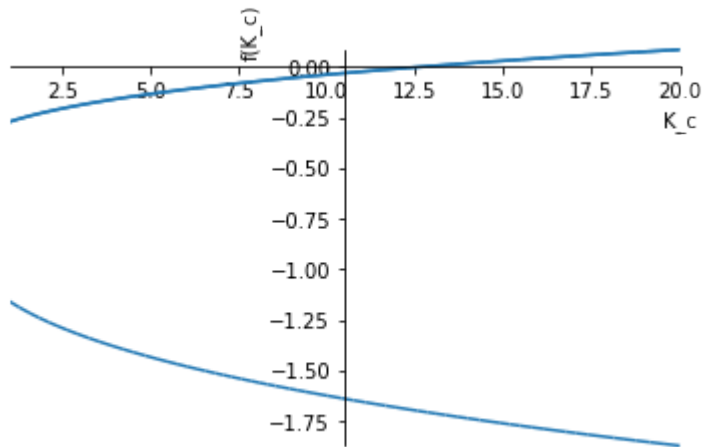
```
[11]:
```

$$K_c + 10s^3 + 17s^2 + 8s + 1$$

```
[12]: roots = sympy.solve(ce, s)
```

OK, that approach works, but is limited in the analytic case to 4th order polynomials


```
[13]: sympy.plot(*[sympy.re(r) for r in roots], (K_c, 1, 20))
```



```
[13]: <sympy.plotting.plot.Plot at 0x11b5e0978>
```

We can see that one root gets a positive real part around $K_c = 12.5$

3.2.2 Using the control library

We quickly run out of SymPy's abilities when calculating closed loop transfer functions. Let's try to do it with the control library instead:

```
[14]: import control
import numpy
import scipy.signal
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[15]: s = control.tf([1, 0], [1])
```

```
[16]: G_v = 1/(2*s + 1)
G_p = G_d = 1/(5*s + 1)
G_m = 1/(s + 1)
K_m = 1
```

```
[17]: from tbcontrol.loops import feedback
```

```
[18]: def closedloop(K_c):
    G_c = K_c

    G_CL = K_m*feedback(G_c*G_v*G_p, G_m)
    return G_CL
```

```
[19]: closedloop(2)
```

```
[19]:
```

$$\frac{20s^3 + 34s^2 + 16s + 2}{100s^5 + 240s^4 + 209s^3 + 103s^2 + 29s + 3}$$

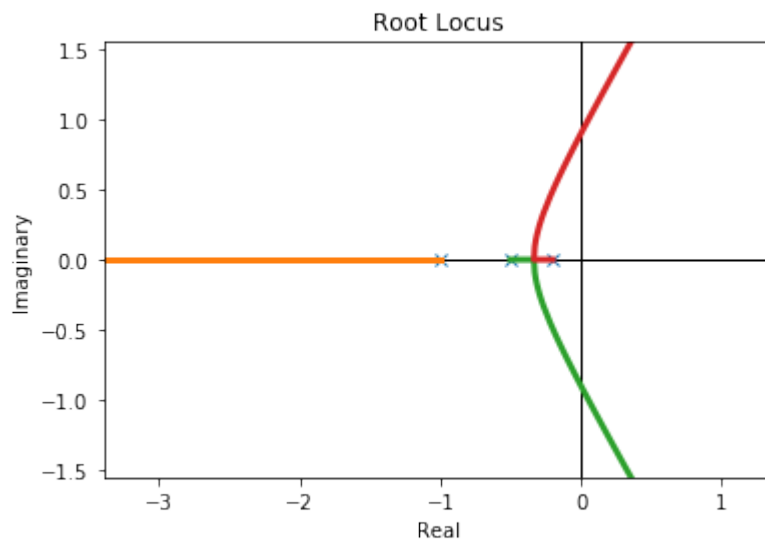

```
[20]: from ipywidgets import interact
```

```
[21]: smootht = numpy.linspace(0, 20)
```

```
[22]: loop = G_v*G_p*G_m
```

```
[23]: _ = control.rlocus(loop)
```

```
/Users/alchemyst/anaconda3/lib/python3.7/site-packages/matplotlib/figure.py:98:
↳MatplotlibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses the
↳earlier instance. In a future version, a new instance will always be created and
↳returned. Meanwhile, this warning can be suppressed, and the future behavior
↳ensured, by passing a unique label to each axes instance.
"Adding an axes using the same arguments as a previous axes "
```



```
[24]: def response(K_C):
    G_CL = closedloop(K_C)
    poles = G_CL.pole()
    plt.plot(*control.step_response(G_CL, smootht))
    _ = control.rlocus(loop)
    plt.scatter(poles.real, poles.imag, color='red')
```

```
[25]: interact(response, K_C=(1., 20.))
```

```
interactive(children=(FloatSlider(value=10.5, description='K_C', max=20.0, min=1.0),
↳Output()), _dom_classes=...
```

```
[25]: <function __main__.response(K_C)>
```

Now, the step response is calculated quickly enough that we can interact with the graphics using the slider!

Direct substitution

From our exploration above it is clear there is a zero of the characteristic equation at $K_C \approx 13$. Let's solve for this numerically:

```
[26]: def chareq(x):  
      Kc, omega = x  
      s = 1j*omega  
      ce = 1 + Kc*loop  
      ce_of_s = ce(s)  
      return ce_of_s.real, ce_of_s.imag
```

```
[27]: import scipy.optimize
```

```
[28]: scipy.optimize.fsolve(chareq, [13, 1])
```

```
[28]: array([12.6          ,  0.89442719])
```

How can we determine stability without calculating the roots? See the [next notebook](#).

3.2.3 Why do we need the Routh Array

In a [previous notebook](#) we showed that we can calculate the roots of the denominator of a closed loop transfer function to determine stability regions as a function of K_c . However, it became clear that analytic calculation of the roots would only work for lower-order systems.

Using numeric methods seemed to work OK, but involved trial-and-error.

Numeric root finding algorithms are also problematic. Consider finding the roots of $(1 + s)^{10}$. We can see that they should all be -1. Let's see how well `numpy.roots` does in finding them.

```
[1]: import numpy
```

```
[2]: numpy.roots([1, 2, 1])
```

```
[2]: array([-1., -1.])
```

```
[3]: polynomial = [1]  
     term = [1, 1]
```

```
[4]: for i in range(10):  
     polynomial = numpy.convolve(polynomial, term)
```

```
[5]: polynomial
```

```
[5]: array([ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1])
```

```
[6]: roots = numpy.roots(polynomial)  
     roots
```

```
[6]: array([-1.0486659 +0.01614412j, -1.0486659 -0.01614412j,  
          -1.02925286+0.04166079j, -1.02925286-0.04166079j,  
          -0.99899397+0.05030124j, -0.99899397-0.05030124j,  
          -0.9701264 +0.03974754j, -0.9701264 -0.03974754j,  
          -0.95296087+0.01496287j, -0.95296087-0.01496287j])
```


We're making up to 5% error and reporting non-negligible imaginary components, when we know the roots are actually real. So it's not that easy to make a call about the nature of the roots of high order polynomials by calculating them numerically. And it's not just because the algorithm isn't good enough. Evaluating one of the roots gives zero to many decimals. The problem is that computers use finite representations of these numbers.

```
[7]: numpy.polyval(polynomial, roots[0])
[7]: (-1.0769163338864018e-13+6.760217385881617e-15j)
```

3.2.4 A better way

The [Routh-Hurwitz stability criterion](#) provides an efficient check of stability for closed loop systems which avoids calculating the roots of a higher-order polynomial and is therefore less error prone if we have numeric coefficients and actually possible if we have symbolic coefficients (recall we cannot calculate the roots analytically for orders higher than 4).

```
[8]: import sympy
sympy.init_printing()

[9]: s = sympy.Symbol('s')

[10]: a_0, a_1, a_2, a_3, a_4 = sympy.symbols('a_0:5')
p = a_0 + a_1*s**1 + a_2*s**2 + a_3*s**3 + a_4*s**4
```

Note that we have to convert the expression above to a `Poly` object to recover all the coefficients.

```
[11]: p = sympy.Poly(p, s)
p
[11]: Poly(a4s4 + a3s3 + a2s2 + a1s + a0, s, domain = ℤ[a0, a1, a2, a3, a4])
```

This function constructs the Routh array as given in Seborg.

```
[12]: from tbcontrol.symbolic import routh

[13]: help(routh)
Help on function routh in module tbcontrol.symbolic:

routh(p)
    Construct the Routh-Hurwitz array given a polynomial in s

    Input: p - a sympy.Poly object
    Output: The Routh-Hurwitz array as a sympy.Matrix object
```

```
[14]: routh(p)
[14]:
```

$$\begin{bmatrix} a_4 & a_2 & a_0 \\ a_3 & a_1 & 0 \\ -\frac{a_1 a_4}{a_3} + a_2 & a_0 & 0 \\ \frac{a_0 a_3^2 + a_1(a_1 a_4 - a_2 a_3)}{a_1 a_4 - a_2 a_3} & 0 & 0 \\ a_0 & 0 & 0 \end{bmatrix}$$

(continues on next page)

(continued from previous page)

Let's try this on example 10.1

```
[15]: K_c = sympy.Symbol('K_c')
```

```
[16]: ce = 10*s**3 + 17*s**2 + 8*s + 1 + K_c
```

```
[17]: A = routh(sympy.Poly(ce, s))
A
```

```
[17]:
```

$$\begin{bmatrix} 10 & 8 \\ 17 & K_c + 1 \\ -\frac{10K_c}{17} + \frac{126}{17} & 0 \\ K_c + 1 & 0 \end{bmatrix}$$

For stability, the left hand column must have entries with all the same signs:

```
[18]: sympy.solve([e > 0 for e in A[:, 0]], K_c)
```

```
[18]:
```

$$-1 < K_c \wedge K_c < \frac{63}{5}$$

3.2.5 Root locus diagrams

Root locus diagrams show where the roots of the characteristic equation lie for different values of controller gain. The control library has a built-in function for plotting these diagrams.

```
[1]: import control
```

```
[2]: from matplotlib import pyplot as plt
```

```
[3]: %matplotlib inline
```

```
[4]: s = control.tf([1, 0], 1)
```

```
[9]: def rlocus(order, tau_p, K):
    Gp = 1/(tau_p*s + 1)**order
    Gc = 1

    L = Gp*Gc
    CL = K*L/(1 + K*L)
    control.root_locus(L);
    control.pzmap(CL)
    plt.title('')
```

```
[10]: from ipywidgets import interact
```

```
[11]: interact(rlocus, order=(1, 5), tau_p=(0.1, 2.), tau_i=(1., 20), K=(0.01, 200))
```



```
interactive(children=(IntSlider(value=3, description='order', max=5, min=1),
FloatSlider(value=1.05, descripti...
```

```
[11]: <function __main__.rlocus(order, tau_p, K)>
```

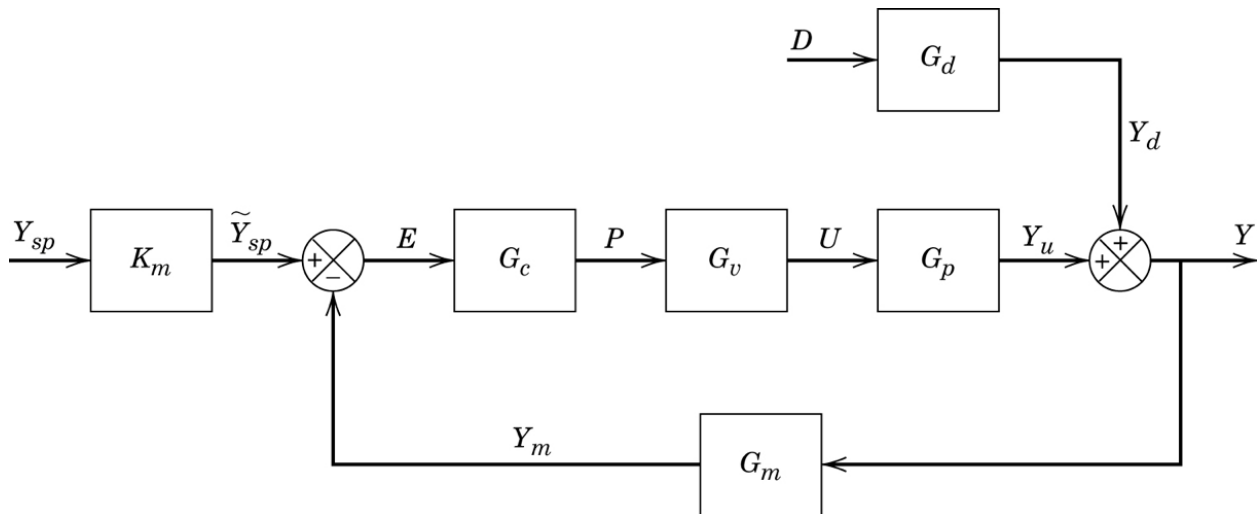
```
[ ]:
```

3.3 PID controller design, tuning and troubleshooting

```
[1]: import sympy
sympy.init_printing()
%matplotlib inline
```

3.3.1 Direct synthesis PID design

The direct synthesis design technique has a very appealing premise: we choose the desired closed loop behaviour and then rewrite the closed loop transfer function to find the controller which will give us this behaviour.



Specifically, we will specify what we want $\frac{Y}{Y_{SP}}$ to be, given that $D = 0$. We will also then calculate $\frac{Y}{Y_{SP}}$ from the block diagram and then solve for G_C .

```
[2]: s, G_C = sympy.symbols('s, G_C')
tau_c, phi = sympy.symbols('tau_c, phi', positive=True, nonzero=True)
```

Let's start by choosing a first order plus dead time response for our system. If any of G_v or G_p contain dead time, we cannot avoid that dead time in the response of our system to a setpoint change. Because sympy wants to typeset exponents with positive values, I will be using a transformation $\phi = -\theta$ in this notebook to get forms similar to the textbook.

```
[3]: desired_Y_over_Y_sp = sympy.exp(phi*s) / (tau_c*s + 1)
```

This is what the prototypical response we've specified looks like. You can see that τ_c specifies the desired speed of the response. Also notice that the gain is 1, so that the process eventually follows the set point.

```
[4]: from ipywidgets import interact
```



```
[5]: t = sympy.Symbol('t', positive=True)
def plotresponse(theta=(0, 3.), tau_c_in=(1., 5.)):
    desired_response = sympy.inverse_laplace_transform(desired_Y_over_Y_sp.subs({phi:
    ↪ -theta, tau_c: tau_c_in})/s, s, t)
    p = sympy.plot(desired_response, (t, 0, 10), show=False)
    p2 = sympy.plot(1, (t, 0, 10), show=False)
    p.append(p2[0])
    p.show()
interact(plotresponse);

interactive(children=(FloatSlider(value=1.5, description='theta', max=3.0),
    ↪ FloatSlider(value=3.0, description=...
```

Now, we calculate the closed loop transfer function. We will assume we have a model of the system called \tilde{G}

```
[6]: Gtilde = sympy.Symbol(r'\widetilde{G}')
actual_Y_over_Y_sp = Gtilde*G_C/(1 + Gtilde*G_C)
```

To find the controller expression which will achieve our desired response, we simply solve for desired = actual

```
[7]: G_C_solved, = sympy.solve(desired_Y_over_Y_sp - actual_Y_over_Y_sp, G_C)
G_C_solved
```

```
[7]:
```

$$\frac{e^{\phi s}}{\tilde{G}(s\tau_c - e^{\phi s} + 1)}$$

We will approximate the dead time in the denominator by a first order Taylor expansion. Note that this choice is not completely unique. In general, we will choose the approximation (Padé or Taylor) which results the correct order of transfer function in the next steps.

```
[8]: denom = sympy.denom(G_C_solved)
G_C_rational = G_C_solved*denom/denom.subs(sympy.exp(phi*s), 1 + phi*s)
```

```
[9]: G_C_rational.simplify()
```

```
[9]:
```

$$-\frac{e^{\phi s}}{\tilde{G}s(\phi - \tau_c)}$$

Now we can relate this to PID parameters for a general process. Here we have a PID controller expression.

```
[10]: K_C, tau_I, tau_D = sympy.symbols('K_C, tau_I, tau_D', positive=True, nonzero=True)
PID = K_C*(1 + 1/(tau_I*s) + tau_D*s)
PID.expand().together()
```

```
[10]:
```

$$\frac{K_C(s^2\tau_D\tau_I + s\tau_I + 1)}{s\tau_I}$$

For reference, we could also go for the ISA realizable controller, but then we'd need a different dead time approximation.

```
[11]: alpha = sympy.symbols('alpha')
```

```
[12]: ISA = K_C*(1 + 1/(tau_I*s) + tau_D*s/(alpha*tau_D*s + 1))
```



```
[13]: num, den = ISA.cancel().as_numer_denom()
```

```
[14]: num.collect(s)
```

```
[14]:
```

$$K_C + s^2 (K_C \alpha \tau_D \tau_I + K_C \tau_D \tau_I) + s (K_C \alpha \tau_D + K_C \tau_I)$$

And here we have a second order process with dead time.

```
[15]: K, tau_c, tau_1, tau_2, phi, theta = sympy.symbols('K, tau_c, tau_1, tau_2, phi, theta
→', positive=True)
G = K*sympy.exp(phi*s)/((tau_1*s + 1)*(tau_2*s + 1))
G
```

```
[15]:
```

$$\frac{K e^{\phi s}}{(s \tau_1 + 1)(s \tau_2 + 1)}$$

Our goal is to find the PID parameters which match the rational G_C we derived earlier.

```
[16]: target_G_C = G_C_rational.subs(Gtilde, G).expand().together()
```

We will create an object to hold on to equality in residual form ($G_c = G_{PID} \iff G_c - G_{PID} = 0$)

```
[17]: zeroeq = (target_G_C - PID).simplify()
```

```
[18]: numer, denom = zeroeq.as_numer_denom()
eq = sympy.poly(numer, s)
```

The following straightforward solution of the equations yields the correct result.

```
[19]: eqs = eq.coeffs()
```

```
[20]: eqs
```

```
[20]:
```

$$[-KK_C\phi\tau_D\tau_I + KK_C\tau_D\tau_I\tau_c - \tau_1\tau_2\tau_I, \quad -KK_C\phi\tau_I + KK_C\tau_I\tau_c - \tau_1\tau_I - \tau_2\tau_I, \quad -KK_C\phi + KK_C\tau_c - \tau_I]$$

```
[21]: sympy.solve(eqs, [K_C, tau_D, tau_I], dict=True)
```

```
[21]:
```

$$\left[\left\{ K_C : -\frac{\tau_1 + \tau_2}{K(\phi - \tau_c)}, \quad \tau_D : \frac{\tau_1\tau_2}{\tau_1 + \tau_2}, \quad \tau_I : \tau_1 + \tau_2 \right\} \right]$$

Note that neglecting the `dict=True` argument above does not currently work for Python 3.6 (see [this issue](#)). If the solution process fails for you, read on below.

Alternate solution

If the simple solution above didn't work, we can do it a little more manually. Look at those equations again

```
[22]: eqs
```

```
[22]: [-KK_C\phi\tau_D\tau_I + KK_C\tau_D\tau_I\tau_c - \tau_1\tau_2\tau_I, \quad -KK_C\phi\tau_I + KK_C\tau_I\tau_c - \tau_1\tau_I - \tau_2\tau_I, \quad -KK_C\phi + KK_C\tau_c - \tau_I]
```

With a little bit of help from us to choose the correct order to solve, we can get the solution in the book.

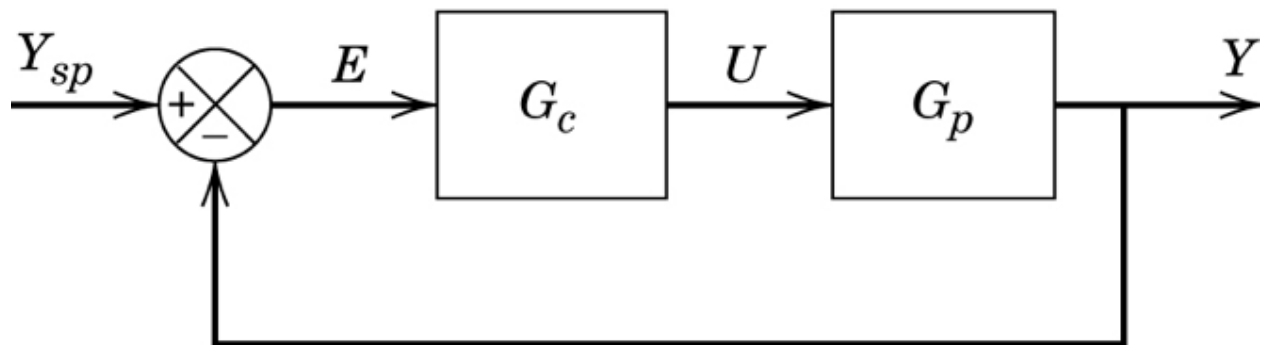
```
[23]: solution = {}
solution[K_C] = sympy.solve(eqs[1], K_C)[0]
solution[tau_D] = sympy.solve(eqs[0], tau_D)[0].subs(solution)
solution[tau_I] = sympy.solve(eqs[2], tau_I)[0].subs(solution).simplify()
solution
```

```
[23]: \left\{ K_C : -\frac{\tau_1 + \tau_2}{K(\phi - \tau_c)}, \quad \tau_D : \frac{\tau_1\tau_2}{\tau_1 + \tau_2}, \quad \tau_I : \tau_1 + \tau_2 \right\}
```

```
[ ]:
```

3.3.2 Minimal integral measures

One approach to finding controller parameters is to minimise some error measure with respect to the parameters. We will simulate a first order plus dead time system under PI control. The block diagram here is for simple feedback:



```
[1]: import numpy
import scipy.signal
import scipy.optimize
import matplotlib.pyplot as plt
from tbcontrol import blocksim
%matplotlib inline
```

```
[2]: # This is the 1,1 element of a Wood and Berry column (see eq 16-12)
K = 12.8
```

(continues on next page)

(continued from previous page)

```
tau = 16.7
theta = 1
```

```
[3]: ts = numpy.linspace(0, 2*tau, 500)
```

```
[4]: ysp = 1
```

```
[5]: Kc = 1
    tau_i = 1
```

```
[6]: def response(Kc, tau_i):
    Gp = blocksim.LTI('G', 'u', 'y', [K], [tau, 1], theta)
    Gc = blocksim.PI('Gc', 'e', 'u', Kc, tau_i)

    blocks = [Gp, Gc]
    inputs = {'ysp': lambda t: ysp}
    sums = {'e': ('+ysp', '-y')}

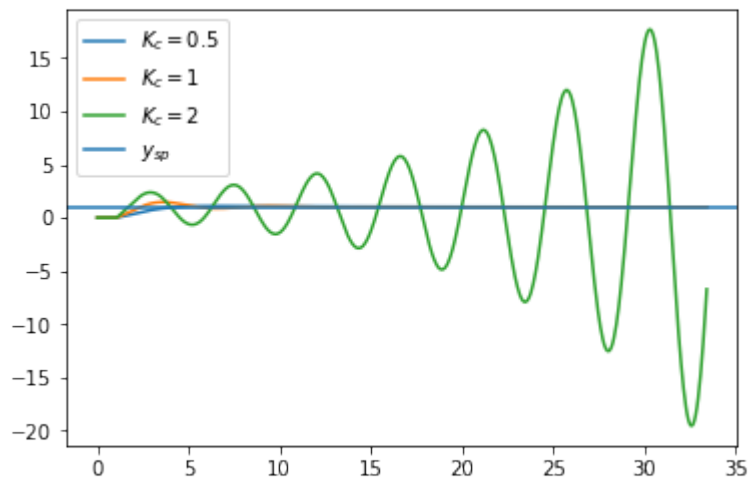
    diagram = blocksim.Diagram(blocks, sums, inputs)

    return numpy.array(diagram.simulate(ts) ['y'])
```

What does the setpoint response look like?

```
[7]: for Kc in [0.5, 1, 2]:
    plt.plot(ts, response(Kc, 10), label='$K_c={}$'.format(Kc))
plt.axhline(ysp, label='$y_{sp}$')
plt.legend()
```

```
[7]: <matplotlib.legend.Legend at 0x1c17a02e80>
```



These are the error measures in the book (eq 11-35 to 11-37). Note that the syntax $f(*parameters)$ with $parameters=(1, 2)$ is equivalent to $f(1, 2)$.

```
[8]: def iae(parameters):
    return scipy.integrate.trapz(numpy.abs(response(*parameters) - ysp), ts)
```



```
[9]: def ise(parameters):
      return scipy.integrate.trapz((response(*parameters) - ysp)**2, ts)
```

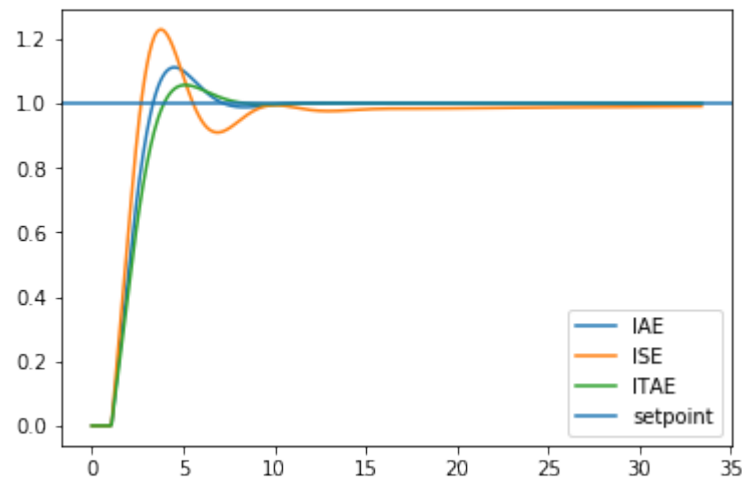
```
[10]: def itae(parameters):
       return scipy.integrate.trapz(numpy.abs(response(*parameters) - ysp)*ts, ts)
```

```
[11]: errfuncs = [iae, ise, itae]
```

Now we can find the optimal parameters for the various error measures.

```
[12]: %%time
       optimal_parameters = {}
       for error in errfuncs:
           name = error.__name__.upper()
           optimal_parameters[name] = scipy.optimize.minimize(error, [1, 10]).x
           print(name, *optimal_parameters[name])
           plt.plot(ts, response(*optimal_parameters[name]), label=name)
       plt.axhline(1, label='setpoint')
       plt.legend(loc='best')
```

```
IAE 0.6639499654277579 16.700104675285132
ISE 0.8519796593454986 25.498163370798167
ITAE 0.5856442797588636 16.69999502290863
CPU times: user 1min 26s, sys: 3 s, total: 1min 29s
Wall time: 1min 35s
```



We could also have used table 11.3, which is automated by a function in `tbcontrol` (see the *ITAE parameters for FOPDT system* notebook for more information on this function)

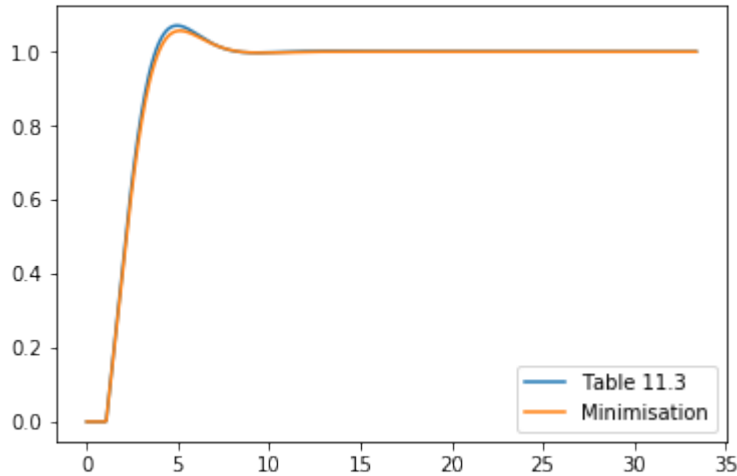
```
[14]: from tbcontrol.fopdtitae import parameters
```

```
[15]: Kc, tau_i = parameters(K, tau, theta, "Set point", "PI")
       print(Kc, tau_i)
0.6035259299979403 16.370626907724816
```

These values correspond approximately with those found through direct minimisation.


```
[16]: plt.plot(ts, response(Kc, tau_i), label='Table 11.3')
plt.plot(ts, response(*optimal_parameters['ITAE']), label='Minimisation')
plt.legend()
```

```
[16]: <matplotlib.legend.Legend at 0x1c1952be10>
```



```
[17]: itae([Kc, tau_i])
```

```
[17]: 4.025429557262326
```

```
[18]: itae(optimal_parameters['ITAE'])
```

```
[18]: 3.644873223762508
```

Note that the error we obtained via direct minimisation was lower than the one obtained via the table, as they have fitted a curve through the results.

3.3.3 ITAE parameters for FOPDT system

This notebook is a convenient interface to the `tbcontrol.fopdtitae` module, which calculates the values of the PI/PID controller settings based on Table 11.3 of Seborg, Edgar, Melichamp and Lewin (itself based on Smith and Corripio, 1997).

```
[1]: from tbcontrol import fopdtitae
```

We can get the parameters using the function `fopdtitae.parameters`: The default is disturbance parameters on a PI controller.

```
[2]: fopdtitae.parameters(K=1, tau=1, theta=1)
```

```
[2]: [0.859, 1.4836795252225519]
```


3.3.4 Interactive version

We'll build an interactive version by printing the parameters with their names and allowing for easy entry.

```
[3]: from ipywidgets import interact, FloatText
```

```
[4]: names = 'Kc', 'τI', 'τD'
```

This is the function which does the calculations. You can check the values in Example 11.5:

```
[5]: def tablefunction(K, tau, theta=1.07, type_of_input='Disturbance', type_of_controller=
    ↪ 'PI'):
    parameters = fopdtitae.parameters(K, tau, theta, type_of_input, type_of_
    ↪ controller)
    for name, value in zip(names, parameters):
        print(name, "=", value)
```

```
[6]: tablefunction(1.54, 5.93, 1.07, 'Disturbance', 'PI')
```

```
Kc = 2.9719324064107253
τI = 2.745987615154182
```

```
[7]: interact(tablefunction,
    K=FloatText(value=1.54), tau=FloatText(value=5.93), theta=FloatText(value=1.
    ↪ 07),
    type_of_input=['Disturbance', 'Set point'], type_of_controller=['PI', 'PID
    ↪ ']);

interactive(children=(FloatText(value=1.54, description='K'), FloatText(value=5.93,
    ↪ description='tau'), FloatT...
```

```
[ ]:
```

3.4 Frequency domain analysis of control systems

3.4.1 Stability in the frequency domain

The frequency domain allows us to find the stability of closed loop systems using only open loop transfer functions and simple operations.

This material is also covered in [this video](#). The GeoGebra sheet is available [here](#).

```
[1]: import numpy
    from matplotlib import pyplot as plt
    %matplotlib inline
```


Locating poles and zeros of a complex function

Let's construct a complex transfer function by specifying the poles, zeros and gain separately.

```
[2]: zeros = [1]
     poles = [-1 + 1j, -1 - 1j]
     gain = 1
```

```
[3]: from numpy.polynomial.polynomial import polyvalfromroots
```

```
[4]: def G(s):
     return gain*polyvalfromroots(s, zeros)/polyvalfromroots(s, poles)
```

It will be useful for us to be able to plot a complex curve easily

```
[5]: def plotcomplex(curve, color='blue', marker=None):
     plt.plot(numpy.real(curve), numpy.imag(curve), color=color, marker=marker)
```

```
[6]: def plotpz():
     for p in poles:
         plotcomplex(p, color='red', marker='x')
     for z in zeros:
         plotcomplex(z, color='red', marker='o')
```

This function will change the axes to be a cross through the origin and have an equal aspect ratio (so that a circle appears as a circle)

```
[7]: from tbcontrol.plotting import cross_axis
```

Let's construct a circular contour and see how the image of the contour moves around as the contour moves around. The image is $G(s)$ as s goes through a countour

```
[8]: from ipywidgets import interact
```

```
[9]: def plotsituation(contour):
     plotcomplex(contour)
     plotcomplex(G(contour), color='red')
     plotpz()
     cross_axis()
```

```
[10]: theta = numpy.linspace(0, 2*numpy.pi, 1000)
```

```
[11]: def argumentprinciple(centerreal=(-2., 2.), centerimag=(-2., 2.), radius=(0.5, 3)):
     contour = radius*numpy.exp(1j*theta) + centerreal + 1j*centerimag
     plotsituation(contour)
```

```
[ ]: interact(argumentprinciple)
```

You should be able to verify the **Cauchy argument principle** using the interaction above:

As s describes a simple contour enclosing N_p poles and N_z zeros, the image $G(s)$ encircles the origin $w = N_z - N_p$ times. w is the **winding number**.

Closed loop stability

Normally we will be looking at transfer functions of the form

$$\frac{GK}{1 + GK}$$

So we will want to check if the denominator of the above $(1 + GK)$ has roots in the RHP. To do this we can construct a special contour called the Nyquist D contour which encloses the whole of the RHP. It starts at the origin, then goes up to infinity, circles around at infinite distance from the origin in a clockwise direction, and then comes back up the imaginary axis. For most functions, the part at infinity just maps $1 + GK$ to 1 as GK goes to zero as s goes to infinity.

```
[ ]: omega = numpy.logspace(-2, 2, 1000)
Dcontour = numpy.concatenate([1j*omega, -1j*omega[::-1]]) # We're ignoring the
↪infinite arc
```

Let's assume that $K = 1$ and check if our system will be closed loop stable

```
[ ]: K = 1
```

```
[ ]: plotcomplex(K*G(Dcontour) + 1)
cross_axis(size=2)
```

Counting encirclements of the origin of $1 + GK$ is the same as counting encirclements of -1 by GK :

```
[ ]: def nyquistplot(K):
    plotcomplex(K*G(Dcontour))
    plotcomplex(-1, color='red', marker='o')
    cross_axis(size=2)
```

```
[ ]: nyquistplot(K=1)
```

This enables us to reason easily about the effect of the controller gain on stability:

```
[ ]: interact(nyquistplot, K=(0.5, 5.))
```

Nyquist stability criterion

Let N_P be the number of poles of $KG(s)$ encircled by the D contour and N_Z be the number of zeros of $1 + KG(s)$ encircled by the D contour. N_Z is the number of poles of the closed loop system in the right half plane. The resultant image shall encircle (clock-wise) the point $(-1 + j0)$ w times such that $w = N_Z - N_P$.

For a stable G this boils down to spotting when the Nyquist plot encircles the -1 point.

Bode stability criterion

Nyquist plots are hard to draw by hand, though, so we often use the Bode stability criterion instead. This works by noticing that, in order for the Nyquist graph to encircle the -1 point, the phase angle must reach -180° and the magnitude must be bigger than 1. We can draw a Bode diagram and a Nyquist diagram next to each other to see the effect of changing gains.

```
[ ]: def bodeplot(K):
    fig = plt.figure(figsize=(10,5))

    ax_gain = plt.subplot2grid((2, 2), (0, 0))
```

(continues on next page)

(continued from previous page)

```

ax_phase = plt.subplot2grid((2, 2), (1, 0))
ax_complex = plt.subplot2grid((2, 2), (0, 1), rowspan=2)

freqresp = K*G(1j*omega)

ax_gain.loglog(omega, numpy.abs(freqresp))
ax_gain.axhline(1, color='orange')
ax_gain.set_ylim([0.1, 10])
ax_gain.set_ylabel('|G|')

ax_phase.semilogx(omega, numpy.unwrap(numpy.angle(freqresp)) - numpy.
↪angle(freqresp[0])) # We know the angle should start at 0
ax_phase.axhline(-numpy.pi, color='green')
ax_phase.set_ylabel('∠G / rad')
ax_phase.set_xlabel('ω / (rad/s)')

plt.sca(ax_complex)
nyquistplot(K)

circle = numpy.exp(-1j*numpy.linspace(0, numpy.pi*2))
ax_complex.plot(circle.real, circle.imag, color='orange')
ax_complex.plot([-2, 0], [0, 0], color='green', linewidth=4, alpha=1, zorder=-1)

```

```
[ ]: interact(bodeplot, K=(0.5, 5.))
```

```
[ ]:
```

3.5 Advanced control methods

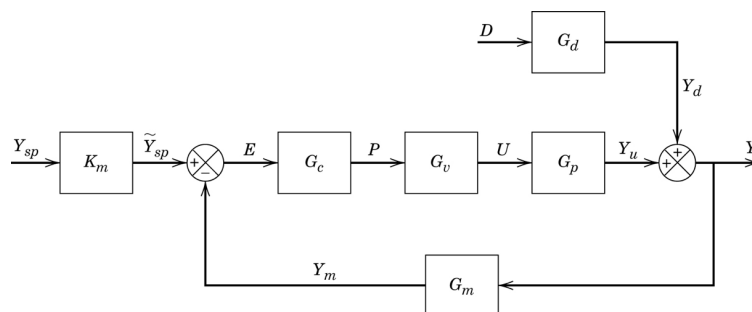
```
[1]: import matplotlib.pyplot as plt
      %matplotlib inline
```

```
[2]: import numpy
```

```
[3]: from tbcontrol import blocksim
```

3.5.1 Dead time reduces control performance

Let's first build a standard control loop with a disturbance.



We'll use the system and simulate the control performance with and without deadtime. $K_m = G_c = G_m = 1$ and

$$G_p = G_d = \frac{e^{-\theta s}}{(5s + 1)(3s + 1)}$$

```
[4]: inputs = {'ysp': blocksim.step(),
              'd': blocksim.step(starttime=40)}
```

```
[5]: sums = {'y': ('+yu', '+yd'),
             'e': ('+ysp', '-y'),
             }
```

For the dead time free system, we can use a high gain PI controller

```
[6]: def Gp(name, input, output, theta):
      return blocksim.LTI(name, input, output,
                          1, numpy.convolve([5, 1], [3, 1]), delay=theta)
```

```
[7]: Gp1 = Gp('Gp', 'p', 'yu', 0)
```

```
[8]: Gc1 = blocksim.PI('Gc', 'e', 'p',
                      3.02, 6.5)
```

For the system with dead time we need to detune the controller

```
[9]: Gp2 = Gp('Gp', 'p', 'yu', 2)
```

```
[10]: Gc2 = blocksim.PI('Gc', 'e', 'p',
                       1.23, 7)
```

```
[11]: Gd = Gp('Gd', 'd', 'yd', 2)
```

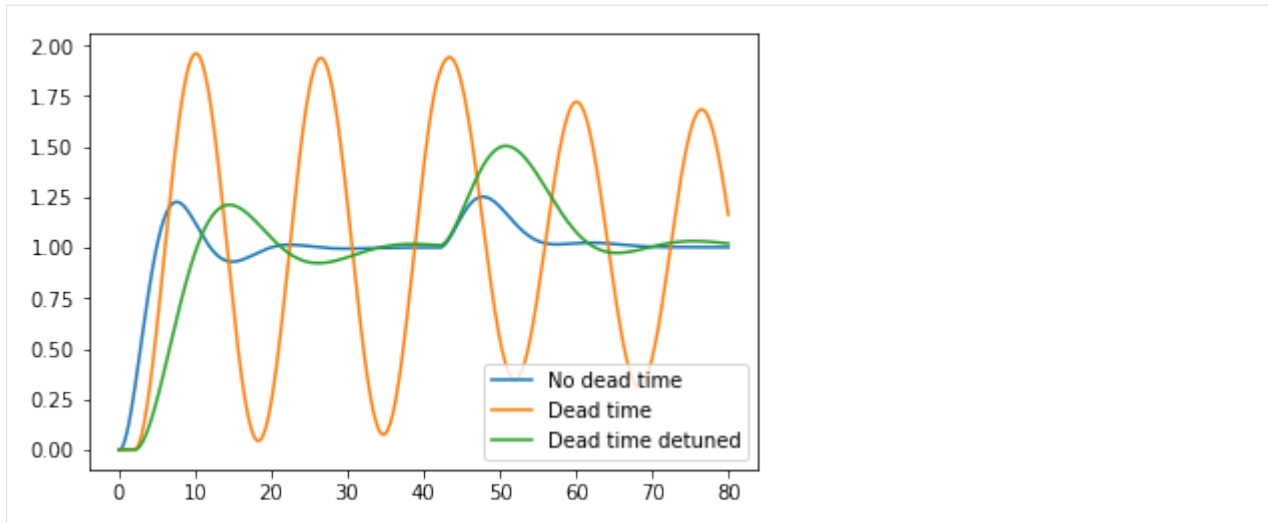
```
[12]: diagrams = {'No dead time': blocksim.Diagram([Gp1, Gc1, Gd], sums, inputs),
                 'Dead time': blocksim.Diagram([Gp2, Gc1, Gd], sums, inputs),
                 'Dead time detuned': blocksim.Diagram([Gp2, Gc2, Gd], sums, inputs)
                 }
```

```
[13]: ts = numpy.linspace(0, 80, 2000)
```

```
[14]: outputs = {}
```

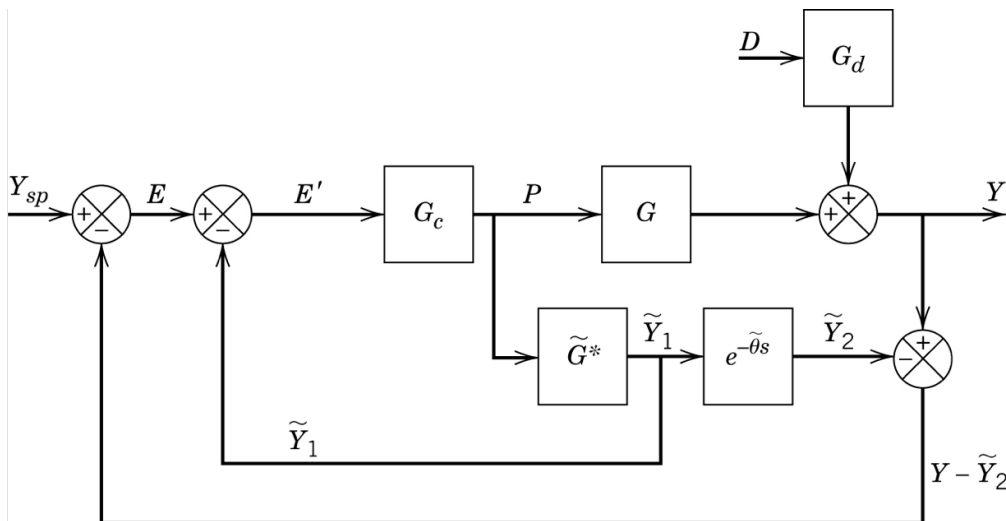
```
[15]: for description, diagram in diagrams.items():
      lastoutput = outputs[description] = diagram.simulate(ts)
      plt.plot(ts, lastoutput['y'], label=description)
      plt.legend()
```

```
[15]: <matplotlib.legend.Legend at 0x1c1f7ade10>
```

3.5.2 Smith Predictor

The Smith predictor or dead time compensator uses a dead time free model to do most of the control (\tilde{G}^*) and then subtracts the delayed prediction from the measurement to react only on the unhandled dynamics.



```
[16]: sums = {'y': ('+yd', '+yu'),
              'y-ytilde2': ('-ytilde2', '+y'),
              'e': ('+ysp', '-y-ytilde2'),
              'eprime': ('+e', '-ytilde1'),
              }
```

We'll use the dead time containing model from before

```
[17]: G = Gp2
      G.name = 'G'
```

But the controller which was tuned on the dead time free model


```
[18]: Gc = Gc1
      Gc.inputname = 'eprime'

[19]: Gtildestar = blocksim.LTI('Gtildestar', 'p', 'ytilde1',
                               1, numpy.convolve([5, 1], [3, 1]))

[20]: delay = blocksim.Deadtime('Delay', 'ytilde1', 'ytilde2', 2)

[21]: blocks = [Gc, G, Gd, Gtildestar, delay]

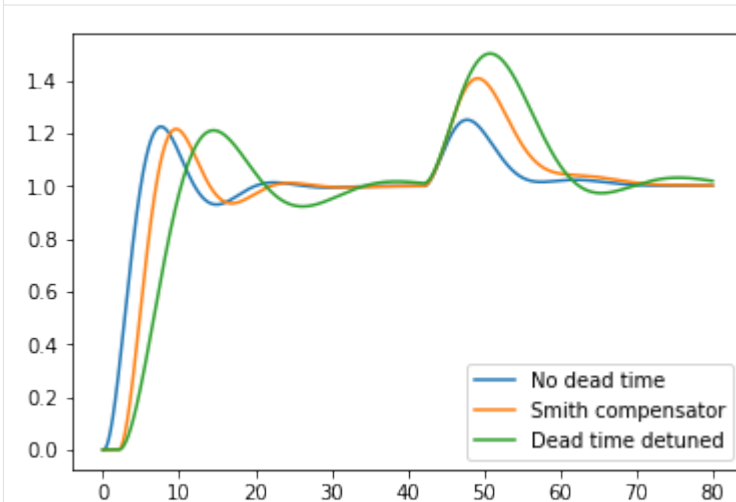
[22]: diagram = blocksim.Diagram(blocks, sums, inputs)
      diagram

[22]: PI: eprime → [ Gc ] → p
      LTI: p → [ G ] → yu
      LTI: d → [ Gd ] → yd
      LTI: p → [ Gtildestar ] → ytilde1
      Deadtime: ytilde1 → [ Delay ] → ytilde2

[23]: outputs['Smith compensator'] = diagram.simulate(ts)

[24]: for description in ['No dead time', 'Smith compensator', 'Dead time detuned']:
      plt.plot(ts, outputs[description]['y'], label=description)
      plt.legend()

[24]: <matplotlib.legend.Legend at 0x1c1f8d2358>
```



We see that the Smith Predictor gives us almost the same performance as the dead time free system, but that it cannot entirely compensate for the delay in the disturbance output because it doesn't have an undelayed measurement of it. It still does better on the disturbance rejection than the detuned PI controller we had to settle for with the dead time.

```
[ ]:
```


3.6 Discrete control and analysis

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[2]: import tbcontrol
tbcontrol.expectversion('0.1.3')
```

3.6.1 Numeric simulation

Let's start with a very simple numeric simulation of a proportional controller acting on a first order process $G = \frac{y}{u} = \frac{K}{\tau s + 1}$.

$$y(s)(\tau s + 1) = Ku(s) \quad (3.1)$$

$$\tau sy(s) + y(s) = Ku(s) \quad (3.2)$$

$$sy(s) = \frac{1}{\tau} (Ku(s) - y(s)) \quad (3.3)$$

$$\frac{dy}{dt} = \frac{1}{\tau} (Ku - y) \quad (3.4)$$

```
[3]: K = 3
tau = 2
```

```
[4]: Kc = 2
```

```
[5]: ts = numpy.linspace(0, 5, 1000)
dt = ts[1]
```

```
[6]: y_continuous = []
u_continuous = []
y = 0
sp = 1
for t in ts:
    e = sp - y
    u = Kc*e
    dydt = 1/tau*(K*u - y)

    y += dydt*dt

    u_continuous.append(u)
    y_continuous.append(y)
```

```
[7]: def plot_continuous():
    fig, [ax_u, ax_y] = plt.subplots(2, 1, sharex=True)
    ax_u.plot(ts, u_continuous)
    ax_u.set_ylabel('$u(t)$')
    ax_y.plot(ts, y_continuous)
    ax_y.axhline(1)
    ax_y.set_ylabel('$y(t)$')
```

(continues on next page)

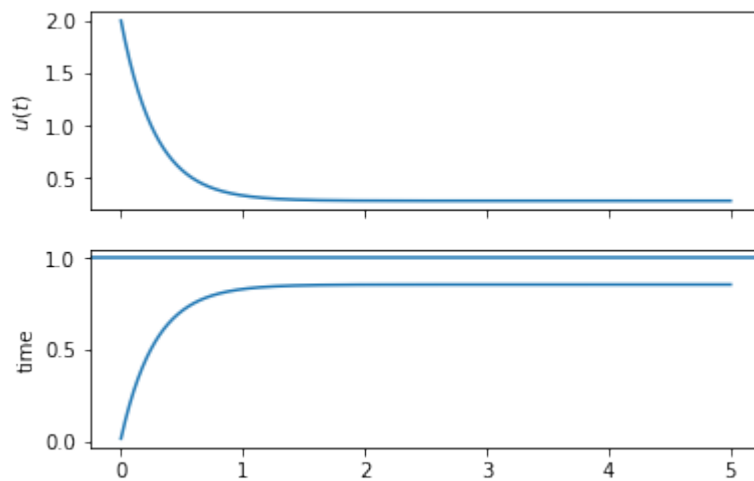
(continued from previous page)

```
ax_y.set_ylabel('time')

return ax_u, ax_y
```

```
[8]: plot_continuous()
```

```
[8]: (<matplotlib.axes._subplots.AxesSubplot at 0x11dfc79e8>,
      <matplotlib.axes._subplots.AxesSubplot at 0x11e022240>)
```



Now, let's use a discrete version of the same controller. We will assume a Zero Order Hold between the controller and the system.

The discrete controller will only run at the sampling points. Now, we have an integration timestep and a discrete timestep. We call the integration timestep dt and the sampling time Δt . We may think that if we use the above `for` loop to update t , it will eventually be equal to Δt , but this is not true in general. Instead, we set a target for the sampling time and check if we are at a time greater than that time, then set a new time one sampling time in the future.

```
[9]: DeltaT = 0.5 # sampling time
```

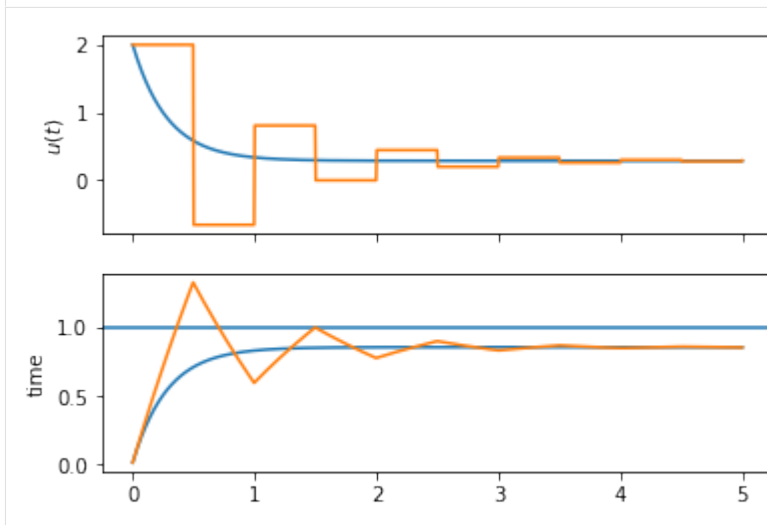
```
[10]: u_discrete = []
      y_discrete = []
      y = 0
      sp = 1
      next_sample = 0
      for t in ts:
          if t >= next_sample:
              e = sp - y
              u = Kc*e
              next_sample += DeltaT
              dydt = 1/tau*(K*u - y)
              y += dydt*dt

              u_discrete.append(u)
              y_discrete.append(y)
```

```
[11]: ax_u, ax_y = plot_continuous()
      ax_u.plot(ts, u_discrete)
      ax_y.plot(ts, y_discrete)
```



```
[11]: [<matplotlib.lines.Line2D at 0x11e1877f0>]
```



Notice the difference? Because the discrete controller only calculates its values at the sampling points and because the ZOH keeps its output constant, the discrete controller takes more action later on, in fact introducing some oscillation where the continuous controller could use arbitrarily large gain.

3.6.2 Symbolic calculation

Now we will try to replicate that last figure without doing numeric simulation. The continuous controller is trivially done via the Laplace transform:

```
[12]: import sympy
sympy.init_printing()
```

```
[13]: s = sympy.Symbol('s')
t = sympy.Symbol('t', positive=True)

Gc = Kc # controller
G = K/(tau*s + 1) # system

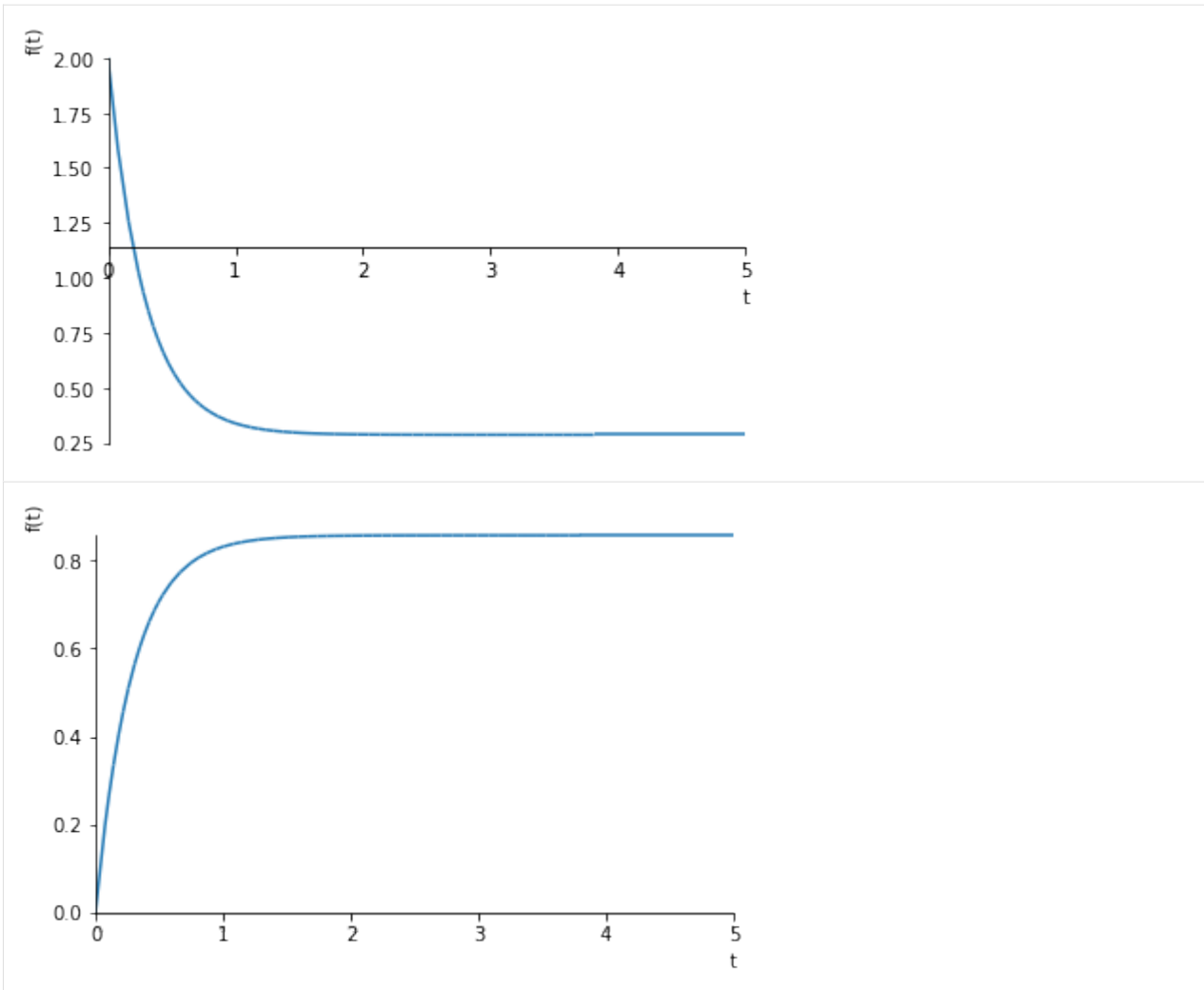
G_cl = Gc*G/(1 + Gc*G)

rs = 1/s # step input r(s)

ys = rs*G_cl # system output y(s)
es = rs - ys # error
us = Gc*es # controller output

yt = sympy.inverse_laplace_transform(ys, s, t)
ut = sympy.inverse_laplace_transform(us, s, t)
```

```
[14]: sympy.plot(ut, (t, 0, 5))
sympy.plot(yt, (t, 0, 5))
```

```
[14]: <sympy.plotting.plot.Plot at 0x11f90c4a8>
```

Now for the discrete controller. First we need some new symbols.

```
[15]: z, q = sympy.symbols('z, q')
```

We get the z transform of a sampled step from the table in the datasheet.

```
[16]: rz = 1 / (1 - z**(-1))
```

If we rewrite a z-transformed signal as a polynomial $r(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} \dots$, the coefficients give the values at the sampling points, so $a_0 = r(0)$, $a_1 = r(\Delta t)$, $a_2 = r(2\Delta t)$ and so on. We can obtain these coefficients easily using a Taylor expansion in sympy.

```
[17]: rz.subs(z, q**(-1)).series()
```

```
[17]: 1 + q + q2 + q3 + q4 + q5 + O(q6)
```

We can see clearly that all the coefficients are 1 for the step.

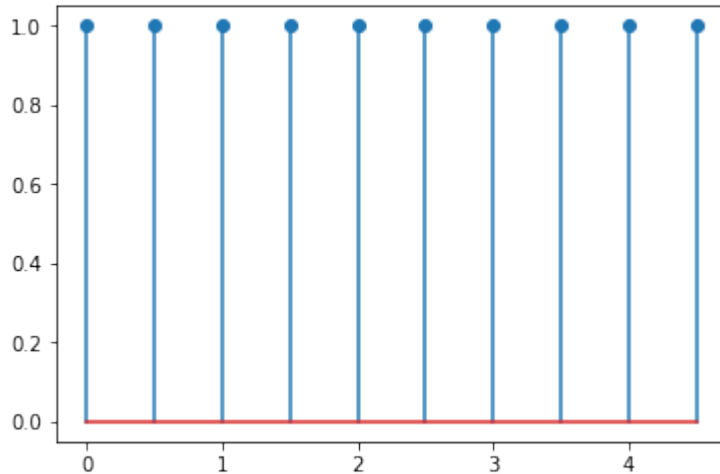
There is more detail in [this notebook](#) if you want to refresh your memory

The `tbcontrol.symbolic.sampledvalues` function automates this process:

```
[18]: from tbcontrol.symbolic import sampledvalues
```

```
[19]: def plotdiscrete(fz, N):
      values = sampledvalues(fz, z, N)
      times = [n*DeltaT for n in range(N)]
      plt.stem(times, values)
```

```
[20]: plotdiscrete(rz, 10)
```



Let's move on to the other transfer functions. The controller is simple:











```
[21]: Gcz = Kc
```

The controller is connected to a hold element (H) which is connected to the system itself (G). The z-transform of this combination can be written $\mathcal{Z}\{H(s)G(s)\}$. Remember, $H(s) = \frac{1}{s}(1 - e^{-\Delta Ts})$. Now we can show

$$\begin{aligned} \mathcal{Z}\left\{\frac{1}{s}(1 - e^{-Ts})G(s)\right\} &= \\ &= \mathcal{Z}\left\{\underbrace{\frac{G(s)}{s}}_{F(s)}(1 - e^{-Ts})\right\} \\ &= \mathcal{Z}\{F(s) - F(s)e^{-Ts}\} \\ &= \mathcal{Z}\{F(s)\} - \mathcal{Z}\{F(s)e^{-Ts}\} = F(z) - F(z)z^{-1} = F(z)(1 - z^{-1}) \end{aligned}$$

So the z transform we're looking for will be $F(z)(1 - z^{-1})$ with $F(z)$ being the transform on the right of the table of $\frac{1}{s}G(s)$.

$$\text{For } G(s) = \frac{K}{\tau s + 1}, F(s) = \frac{K}{s(\tau s + 1)}, F(z) = \frac{K(1-b)z^{-1}}{(1-z^{-1})(1-bz^{-1})}$$

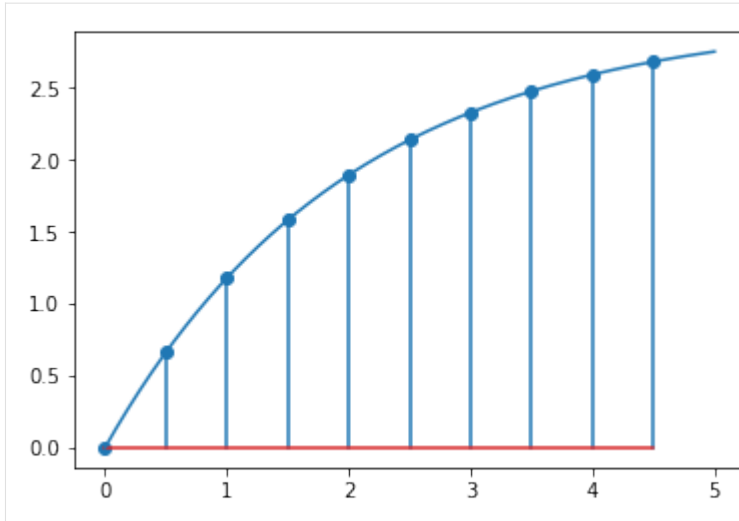
Time domain	Laplace-transform	z-transform ($b = e^{-aT}$)
Impulse: $\delta(t)$ 	1	1
Unit step: $u(t)$ 	$\frac{1}{s}$	$\frac{1}{1 - z^{-1}}$
Ramp: t 	$\frac{1}{s^2}$	$\frac{Tz^{-1}}{(1 - z^{-1})^2}$
t^n 	$\frac{n!}{s^{n+1}}$	$\lim_{a \rightarrow 0} (-1)^n \frac{\partial^n}{\partial a^n} \frac{1}{1 - bz^{-1}}$
e^{-at} 	$\frac{1}{s + a}$	$\frac{1}{1 - bz^{-1}}$
te^{-at} 	$\frac{1}{(s + a)^2}$	$\frac{Tbz^{-1}}{(1 - bz^{-1})^2}$
t^2e^{-at} 	$\frac{2}{(s + a)^3}$	$\frac{T^2bz^{-1}(1 + bz^{-1})}{(1 - bz^{-1})^3}$
$\sin(\omega t)$ 	$\frac{\omega}{s^2 + \omega^2}$	$\frac{z^{-1} \sin(\omega T)}{1 - 2z^{-1} \cos(\omega T) + z^{-2}}$
$\cos(\omega t)$ 	$\frac{s}{s^2 + \omega^2}$	$\frac{1 - z^{-1} \cos(\omega T)}{1 - 2z^{-1} \cos(\omega T) + z^{-2}}$
$1 - e^{-at}$ 	$\frac{a}{s(s + a)}$	$\frac{(1 - b)z^{-1}}{(1 - z^{-1})(1 - bz^{-1})}$

```
[22]: a = 1/tau
b = sympy.exp(-a*DeltaT)
Fz = K*(1 - b)*z**(-1)/((1 - z**(-1))*(1 - b*z**(-1)))
HGz = Fz - z**(-1)*Fz
```

Let's verify that this is correct by plotting the continuous response along with the discrete values:

```
[23]: plotdiscrete(rz*HGz, 10)
plt.plot(ts, K*(1 - numpy.exp(-ts/tau)))
```

```
[23]: [<matplotlib.lines.Line2D at 0x11df60c88>]
```

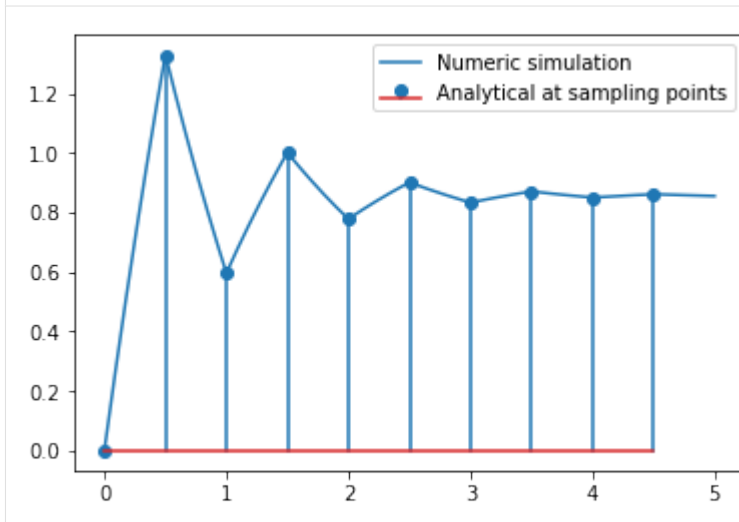



Now we have the discrete transfer functions, we can repeat the same calculation as before.

```
[24]: yz = rz*Gcz*HGz/(1 + Gcz*HGz)
```

```
[25]: plt.plot(ts, y_discrete)
      plotdiscrete(yz, 10)
      plt.legend(['Numeric simulation', 'Analytical at sampling points'])
```

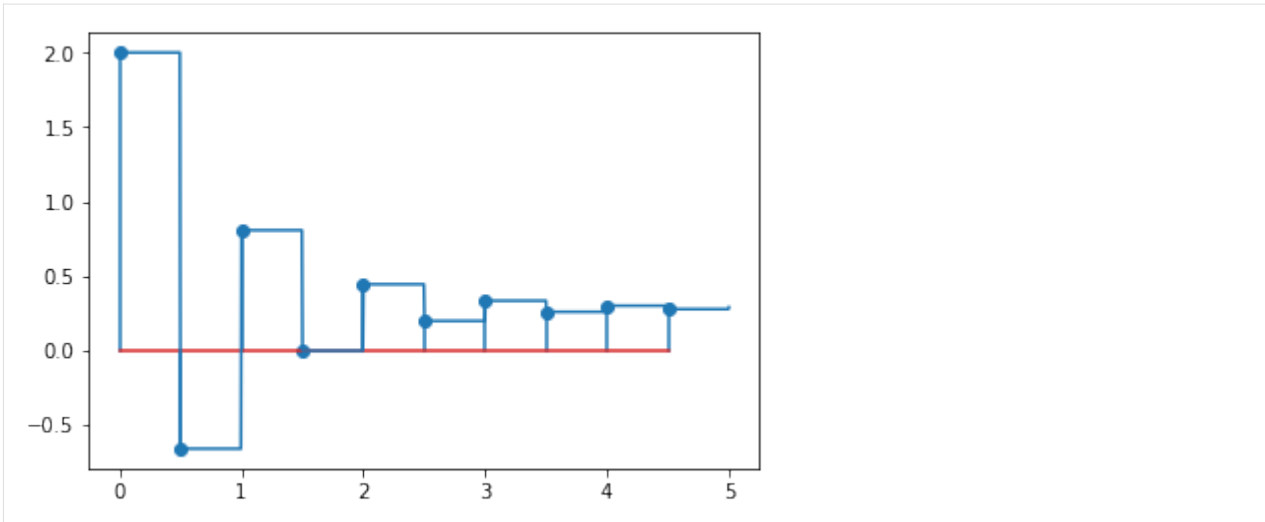
```
[25]: <matplotlib.legend.Legend at 0x11fef9e8>
```



So now we have recovered the response we calculated numerically before analytically. Let's see if we can reproduce the numeric values between the sampling points for the continuous system output. Our first step is to construct the Laplace transform of the controller output. Let's look again what that looked like:

```
[26]: ez = rz - yz
      uz = Gcz*ez
      plotdiscrete(uz, 10)
      plt.plot(ts, u_discrete)
```

```
[26]: [<matplotlib.lines.Line2D at 0x11ff33f28>]
```

We can interpret this as shifted pulse signals added together.

If we were trying to calculate the response of the system to a single pulse input, it would be simple.

Since

$$y(s) = G(s)u(s) \quad (3.5)$$

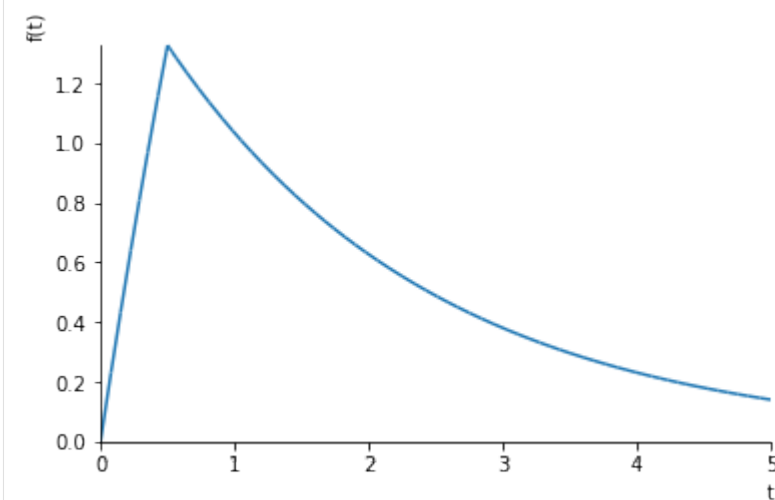
$$y(t) = \mathcal{L}^{-1}\{G(s)u(s)\} \quad (3.6)$$

and a pulse signal of width Δt and height v has Laplace transform $\frac{v}{s}(1 - e^{-\Delta t s}) = vH(s)$, the single output would be easy to obtain:

```
[27]: Hs = 1/s*(1 - sympy.exp(-DeltaT*s))
```

```
[28]: u_single_pulse = 2*Hs
      y_single_pulse = sympy.inverse_laplace_transform(G*u_single_pulse, s, t)
```

```
[29]: sympy.plot(y_single_pulse, (t, 0, 5))
```




```
[29]: <sympy.plotting.plot.Plot at 0x1201e82b0>
```

For subsequent pulses, we simply shift the pulse one time step up. So, given

$$u(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \dots = \sum_{i=0}^{\infty} a_i z^{-i}$$

If the output of the controller is zero-order held so that the held version of the signal is u_h , we can write

$$u_h(s) = a_0 H(s) + a_1 H(s)e^{-\Delta t s} + a_2 H(s)e^{-2\Delta t s} = \sum_{i=0}^{\infty} a_i H(s)e^{-i\Delta t s}$$

This maps quite elegantly to the following generator expression:

```
[30]: uhs = sum(ai*Hs*sympy.exp(-i*DeltaT*s)
              for i, ai in enumerate(sampledvalues(uz, z, 10)))
```

Equivalently, we can write

```
[31]: uhs = 0
a = sampledvalues(uz, z, 10)
for i in range(10):
    uhs += a[i]*Hs*sympy.exp(-i*DeltaT*s)
```

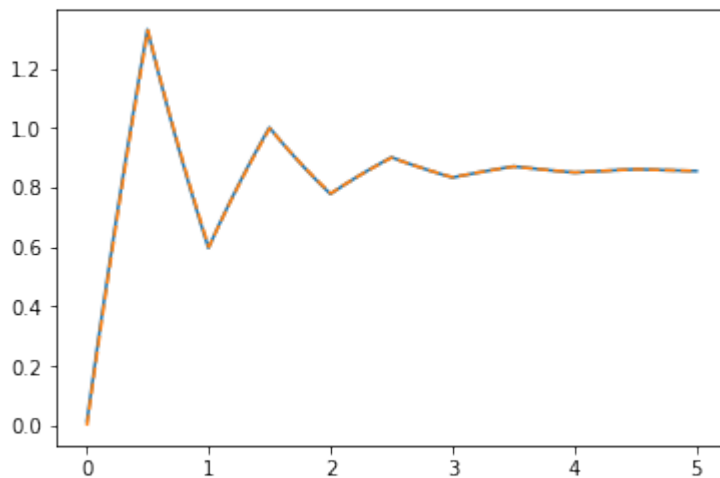
Now we can construct the continuous response like this:

```
[32]: ys = uhs*G
```

```
[33]: yt = sympy.inverse_laplace_transform(ys, s, t)
```

```
[34]: plt.plot(ts, y_discrete)
plt.plot(ts, tbcontrol.symbolic.evaluate_at_times(yt, t, ts), '--')
```

```
[34]: [<matplotlib.lines.Line2D at 0x11f9ee160>]
```



Notice that the analytical solution and numeric solution agree, but make sure you understand what the difference is in approach.

3.6.3 Discrete PI with ITAE parameters

This notebook reproduces Figure 17.10 in Seborg *et al* and also goes a little further

```
[1]: from tbcontrol import blocksim, fopdtitae

[2]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline

[3]: def limit(u, umin, umax):
    return max(umin, min(u, umax))

[4]: class DiscretePI(blocksim.Block):
    def __init__(self, name, inputname, outputname, K, tau_I, deltat, umin=-numpy.inf,
    ↪ umax=numpy.inf):
        super().__init__(name, inputname, outputname)

        self.K = K
        self.tau_I = tau_I
        self.deltat = deltat
        self.umin = umin
        self.umax = umax
        self.reset()

    def reset(self):
        self.state = 0
        self.output = 0
        self.eint = 0
        self.nextsample = 0

    def change_input(self, t, e):
        if t >= self.nextsample:
            self.eint += e*self.deltat
            self.output = self.K*(e + self.eint/self.tau_I)
            self.nextsample += self.deltat

        return limit(self.output, self.umin, self.umax)

    def change_state(self, x):
        self.state = x

    def derivative(self, e):
        return 0

[5]: class DiscretePI_vel(DiscretePI):
    def reset(self):
        self.state = 0
        self.output = 0
        self.e_k_1 = 0
        self.nextsample = 0

    def change_input(self, t, e_k):
        if t >= self.nextsample:
            self.output += self.K*((e_k - self.e_k_1) + self.deltat/self.tau_I*e_k)
            self.output = limit(self.output, self.umin, self.umax)
```

(continues on next page)

(continued from previous page)

```

        self.e_k_1 = e_k
        self.nextsample += self.deltat

    return self.output

```

The system is

$$G = G_v G_p G_m = \frac{-20e^{-s}}{5s + 1} = \frac{K_p e^{-\theta s}}{\tau_p s + 1}$$

It is not explicitly stated in the example, but $G_d = G$ is assumed.

```

[6]: Kp = -20
     taup = 5
     theta = 1

[7]: G = blocksim.LTI('G', 'u', 'yu',
                     Kp, [taup, 1], theta)

[8]: Gd = blocksim.LTI('G', 'd', 'yd',
                      Kp, [taup, 1], theta)

```

Note we can't just do $G = G_d$ because we need new names and a new state for the block.

```

[9]: sums = {'y': ('+yu', '+yd'),
            'e': ('-y', '+r')}

[28]: inputs = {'r': blocksim.zero,
               'd': blocksim.step()}

[11]: ts = numpy.linspace(0, 15, 2000)

[12]: deltats = [0.05, 0.25, 0.5, 1]

```

Reconstruction

Here is the direct reconstruction of Figure 11.10. Turns out they use a continuous PI controller with the parameters for a discrete controller.

```

[13]: def simulate(controller, discrete=True, umin=-numpy.inf, umax=numpy.inf):
        fig, (uaxis, yaxis) = plt.subplots(2, 1, figsize=(5, 10))
        for deltatt in deltats:
            Kc, tau_I = fopdtitae.parameters(Kp, taup, theta + deltatt/2)
            Gc = controller('Gc', 'e', 'u',
                           Kc, tau_I,
                           *((deltatt, umin, umax) if discrete else ()))
            diagram = blocksim.Diagram([G, Gd, Gc], sums, inputs)
            outputs = diagram.simulate(ts)
            uaxis.plot(ts, outputs['u'])
            yaxis.plot(ts, outputs['y'], label='Δt={}'.format(deltatt))
        uaxis.set_ylabel('u')
        yaxis.set_ylabel('y')

```

(continues on next page)

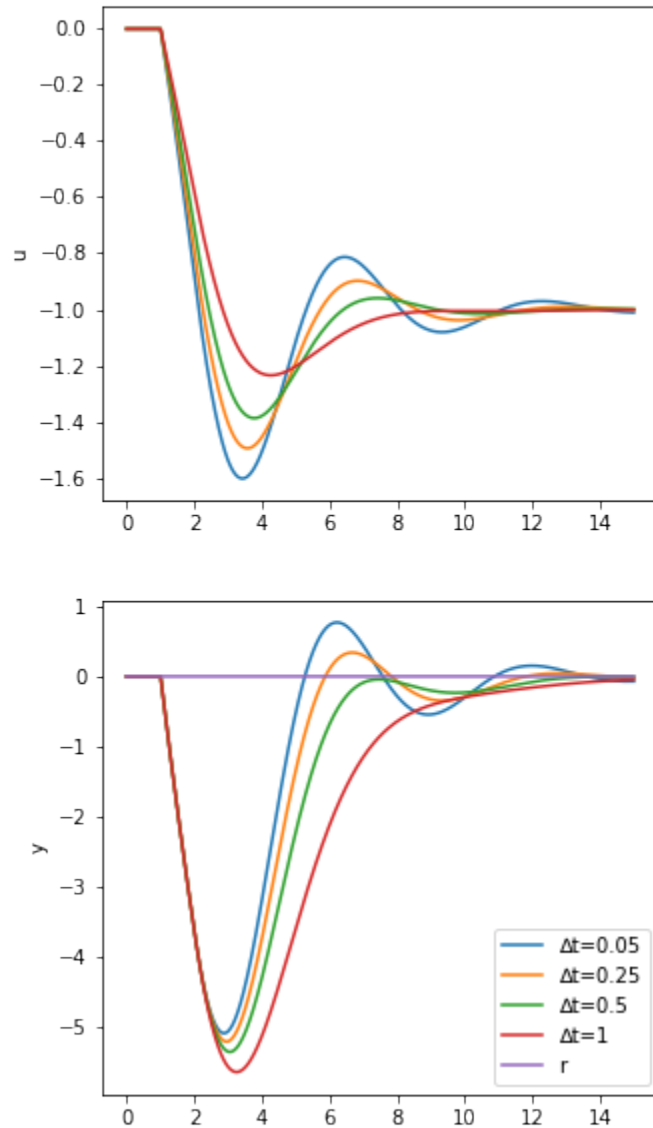
(continued from previous page)

```

yaxis.plot(ts, outputs['r'], label='r')
yaxis.legend()

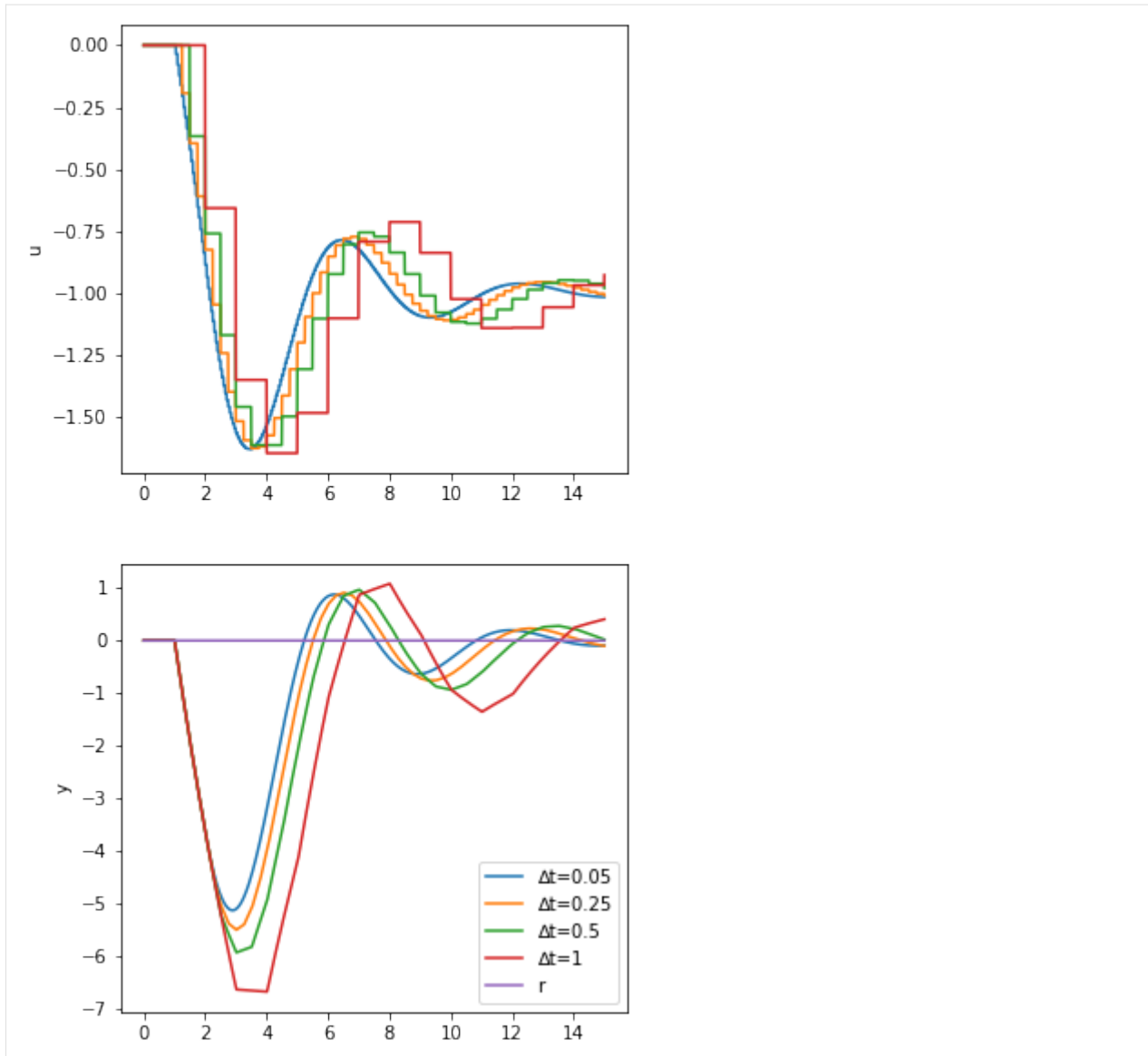
```

```
[14]: simulate(blocksim.PI, discrete=False)
```

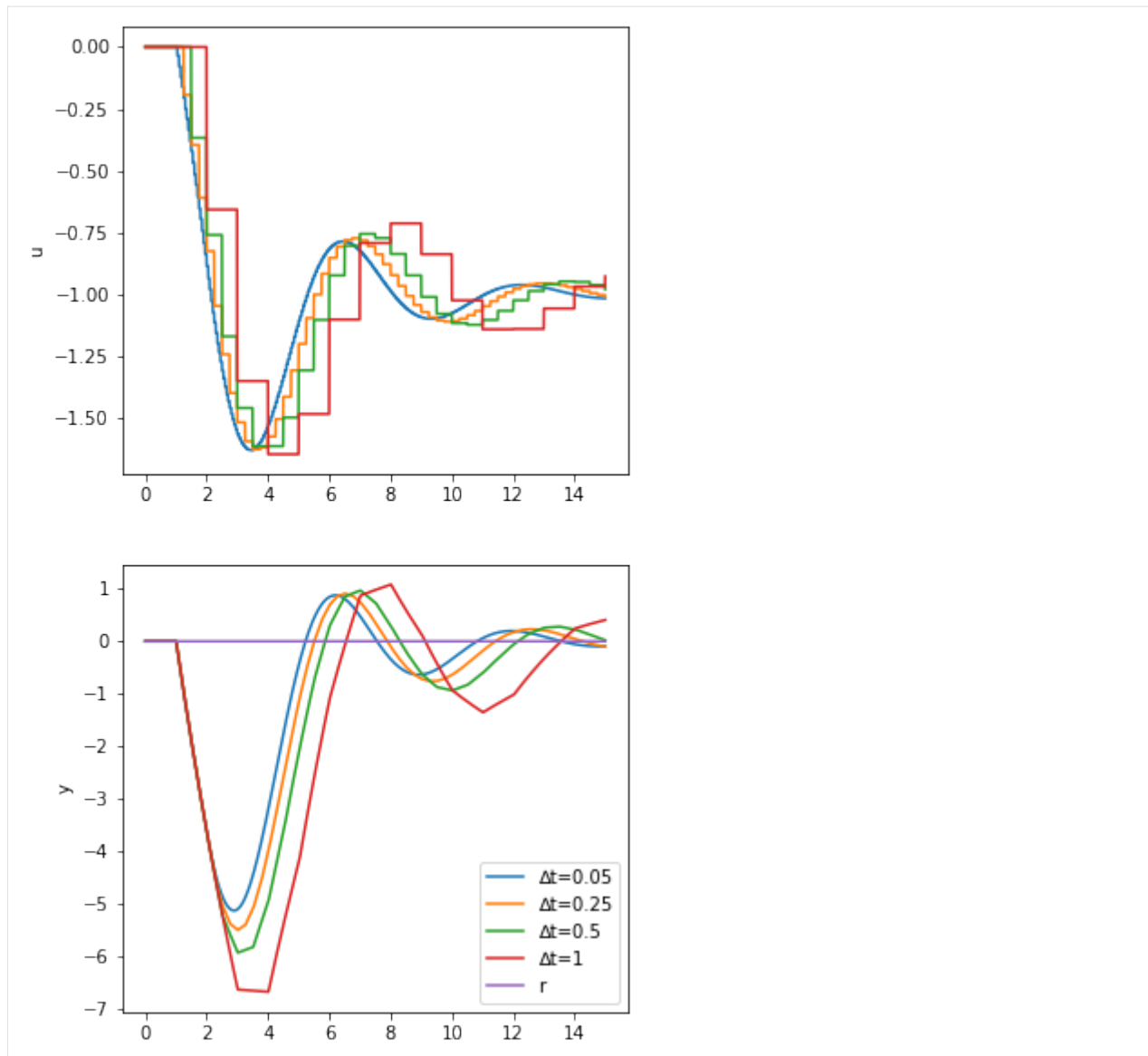


Discrete responses

```
[15]: simulate(DiscretePI)
```

```
[16]: simulate(DiscretePI_vel)
```

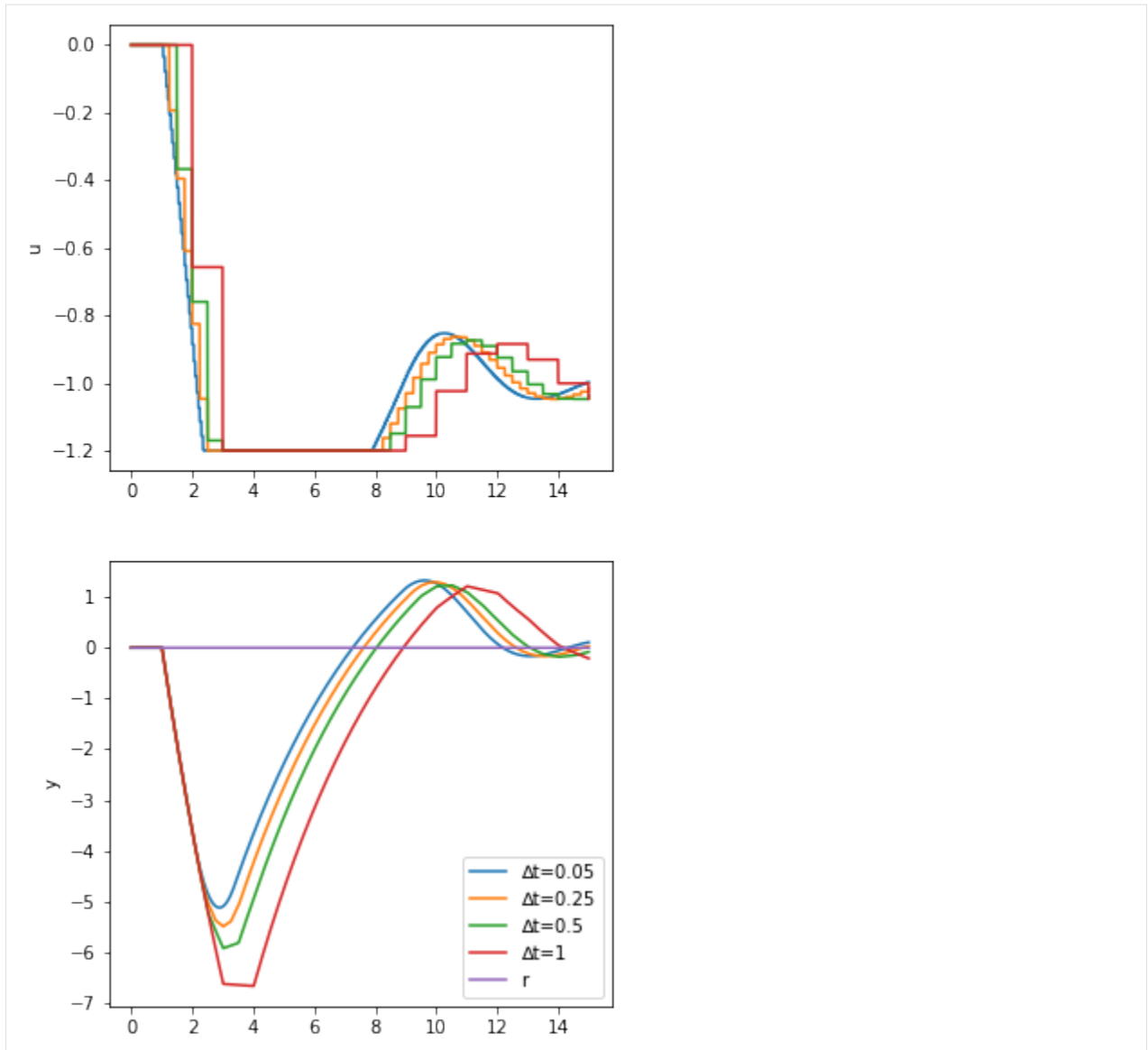



This looks different from the given figure, but the position form and velocity form look identical. Note that the analog PI controller gets better results than the digital approximations with the same controller parameters. “Better” in this context means that in general the error is smaller (note the worst case error in the analog case is about 5, while in the digital case it’s almost 7).

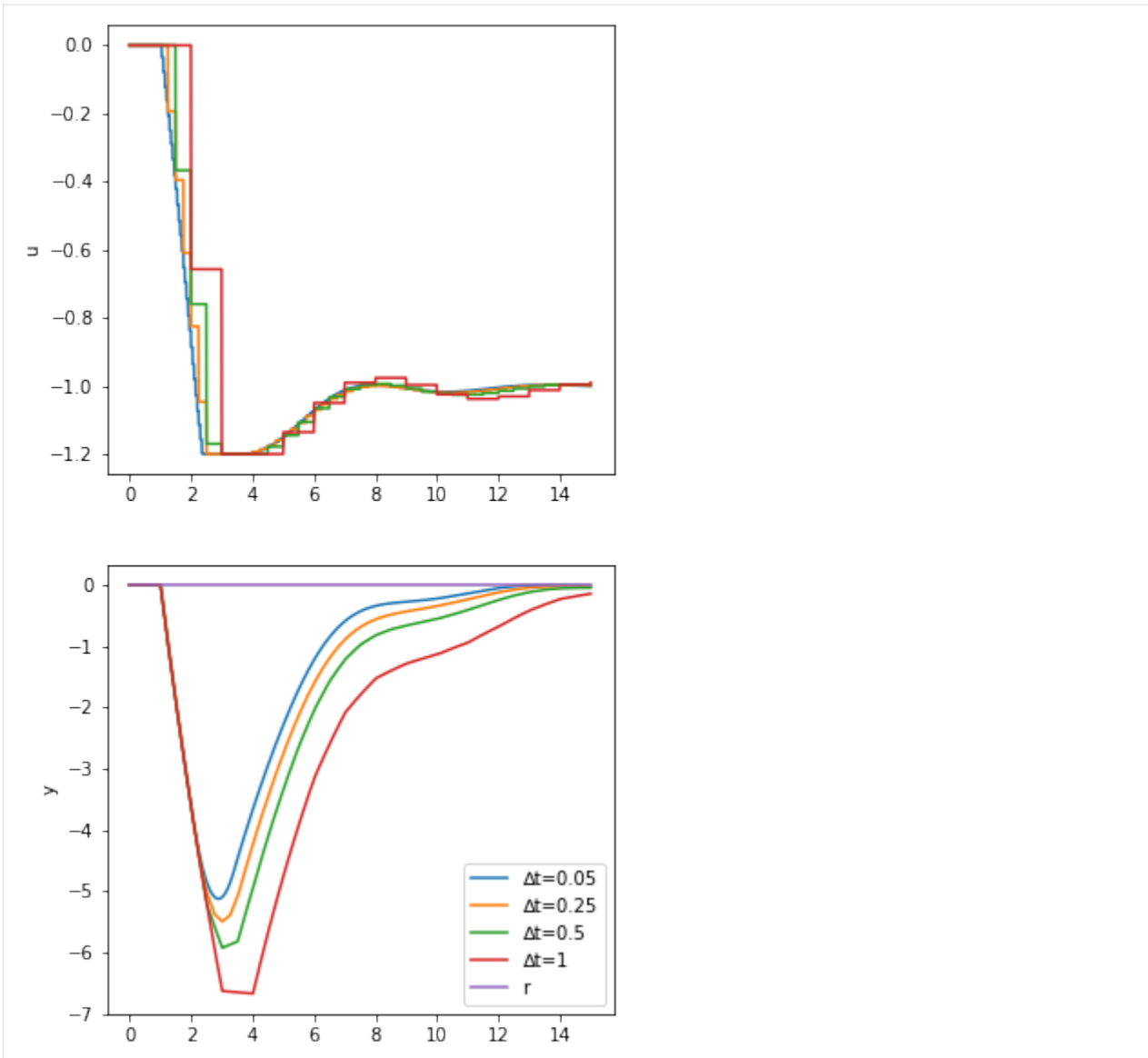
With output limits

Now, let’s include limiting behaviour

```
[19]: simulate(DiscretePI, umin=-1.2, umax=0)
```

```
[20]: simulate(DiscretePI_vel, umin=-1.2, umax=0)
```

The windup on the position form PID is very clear. In the “wound up” case, the output is stuck at the limit for a long time as the controller has to unwind the integral before moving away from the limit. In the velocity form, the output comes away from the limit much faster.

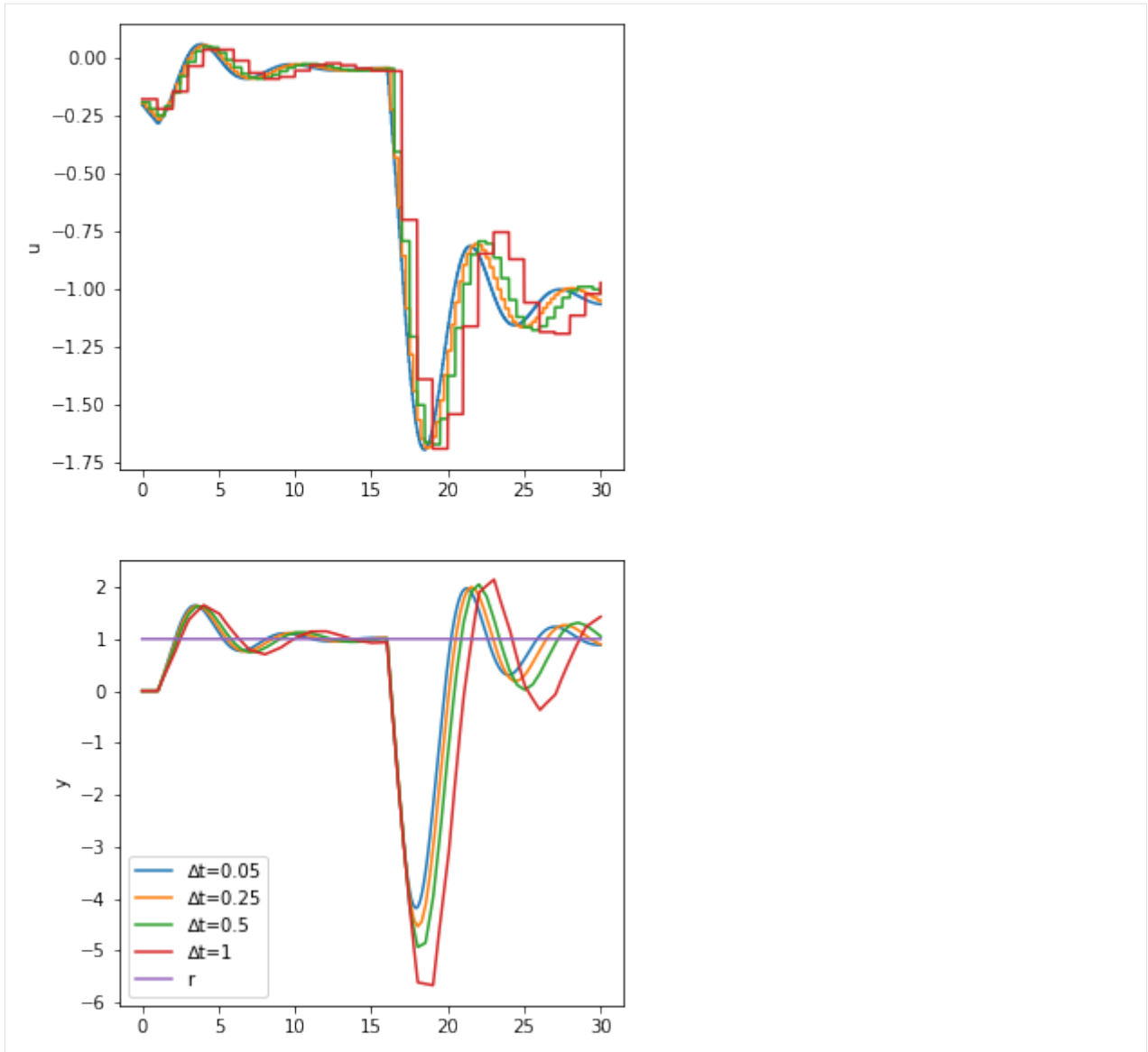
With setpoint tracking

Let’s include setpoint tracking as well.

```
[ ]: inputs['r'] = blocksim.step(starttime=0)
    inputs['d'] = blocksim.step(starttime=15)
```

```
[26]: ts = numpy.linspace(0, 30, 2000)
```

```
[27]: simulate(DiscretePI)
```

[]:

3.6.4 Dahlin controller

We will replicate the controller output in figure 17.11a. This notebook has a [video commentary](#).

Note I am replicating these results using analytic methods to show that the artefacts are not numerical but rather fundamental to the calculations. If you simply want to simulate the action of a discrete controller on a continuous system, have a look at the [Simple discrete controller simulation](#) notebook.

```
[1]: import sympy
sympy.init_printing()
import tbcontrol
tbcontrol.expectversion('0.1.3')
```


3.6.5 Discretise the system

We need to find the corresponding z transform of the hold element and the system. Since $H = 1/s(1 - e^{-Ts})$, we can find $F = G/s$, from there $f(t)$ and then work out the z transform

```
[2]: s, t = sympy.symbols('s, t')
Gs = 1/(15*s**2 + 8*s + 1)
Gs
```

```
[2]:
```

$$\frac{1}{15s^2 + 8s + 1}$$

```
[3]: f = sympy.inverse_laplace_transform(Gs/s, s, t).simplify()
sympy.nsimplify(sympy.N(f)).simplify()
```

```
[3]:
```

$$\theta(t) + \frac{3e^{-\frac{t}{3}}\theta(t)}{2} - \frac{5e^{-\frac{t}{5}}\theta(t)}{2}$$

We can see that $f(t)$ is the sum of 1 and two exponentials. It is easy to determine the corresponding z transforms from the table

Time domain	Laplace-transform	z -transform ($b = e^{-aT}$)
e^{-at}	$\frac{1}{s+a}$	$\frac{1}{1-bz^{-1}}$

```
[4]: z, q = sympy.symbols('z, q')
```

Now the sampling interval

```
[5]: T = 1 # Sampling interval
```

```
[5]: def expz(a):
    b = sympy.exp(-a*T)
    return 1/(1 - b*z**-1)
```

```
[6]: Fz = -5/2*expz(1/5) + 3/2*expz(1/3) + 1/(1 - z**-1)
```

```
[7]: Fz
```

```
[7]:
```

$$\frac{1.5}{1 - \frac{0.716531310573789}{z}} - \frac{2.5}{1 - \frac{0.818730753077982}{z}} + \frac{1}{1 - \frac{1}{z}}$$

Let's see if we did that right

```
[8]: import tbcontrol.symbolic
```

```
[9]: def plotdiscrete(fz, N):
    values = tbcontrol.symbolic.sampledvalues(fz, z, N)
    times = [n*T for n in range(N)]
    plt.stem(times, values)
```



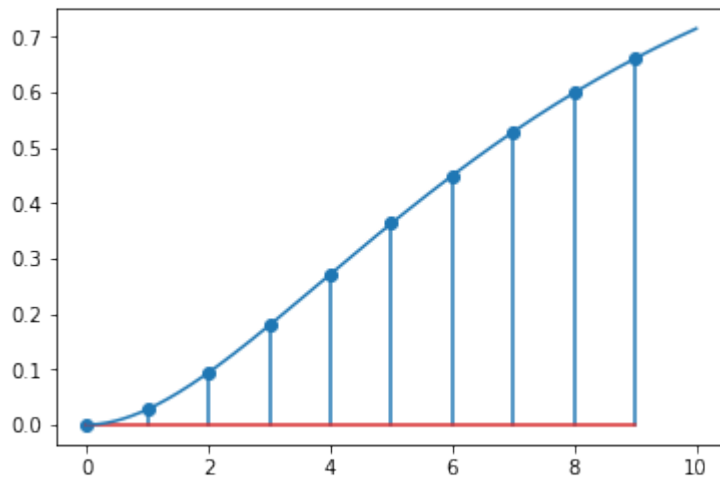
```
[10]: import matplotlib.pyplot as plt
      %matplotlib inline
```

```
[11]: import numpy
```

```
[12]: ts = numpy.linspace(0, 10, 300)
```

```
[13]: plotdiscrete(Fz, 10)
      plt.plot(ts, tbcontrol.symbolic.evaluate_at_times(f, t, ts))
```

```
[13]: [<matplotlib.lines.Line2D at 0x1205b3ef0>]
```



Here is the transform of the system and the hold element. See the *Discrete control* notebook for the derivation.

```
[14]: HG_z = Fz*(1 - z**(-1))
```

3.6.6 Dahlin Controller

The desired closed loop response is first order

```
[15]: theta = 0 # system dead time
      lambda = 1 # Dahlin's lambda
      h = theta # Dahlin's h
```

First order response in eq 17-63

```
[16]: N = theta/T
      A = sympy.exp(-T/lambda)
      yclz = (1 - A)*z**(-N-1)/(1 - A*z**(-1))
```

```
[17]: K = (1/HG_z*yclz/(1 - yclz)).simplify()
```

```
[18]: K
```

```
[18]: 1.0 (0.632120558828558z2 - 0.970470713623842z + 0.370831136111342)
      0.0279700831657292z2 - 0.00455601047319787z - 0.0234140726925314
```



```
[19]: #K = 0.632/(1 - z**-1) * (1 - 1.5353*z**-1 + 0.5866*z**-2)/(0.028 + 0.0234*z**-1)
```

We will model the control response to a unit step in reference signal

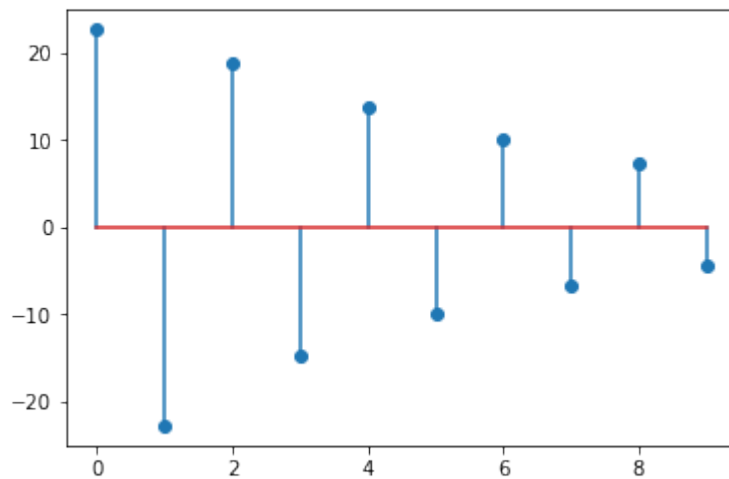
```
[20]: rz = 1/(1 - z**-1) # unit step in z
```

Now we can calculate the z-domain version of the controller output

```
[21]: uz = K/(1 + K*HG_z)*rz
```

```
[22]: N = 10
```

```
[23]: plotdiscrete(uz, N)
```



This oscillating controller output is known as “ringing”. It is an undesirable effect.

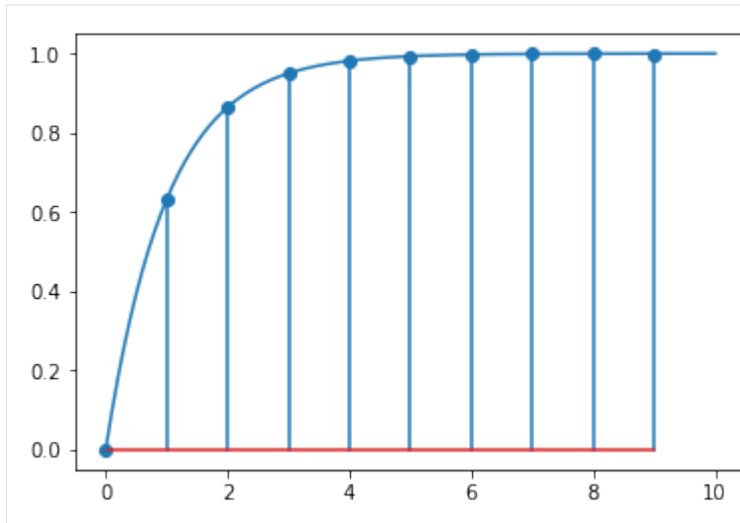
3.6.7 Continuous response

By design the controlled variable follows an exponential at the sampling points.

```
[24]: yz = K*HG_z/(1 + K*HG_z)*rz
```

```
[25]: plotdiscrete(yz, N)
plt.plot(ts, 1 - numpy.exp(-ts/lambda))
```

```
[25]: [<matplotlib.lines.Line2D at 0x120c51390>]
```

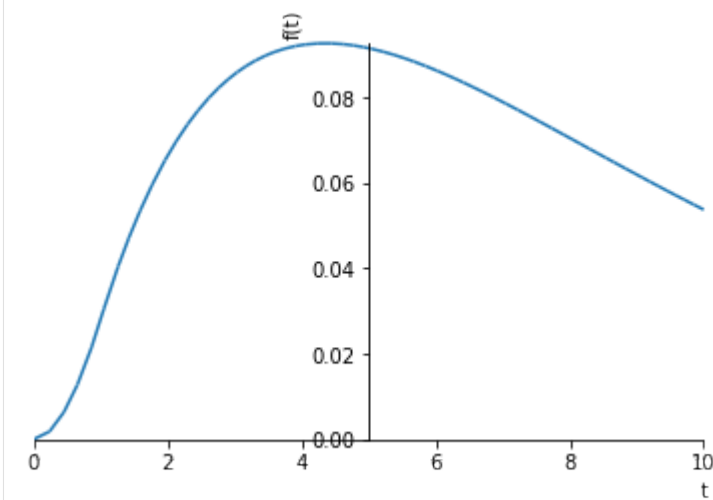
But what does it really look like between data points? First, let's construct the response of the system to a single sampling time length pulse input

```
[26]: p = f - f.subs(t, t-T)
```

Note that there is a slight issue with finding the value of this function at zero, so we avoid that point in plotting by starting at some small value which is not zero.

```
[27]: smallvalue = 0.001
```

```
[28]: sympy.plot(p, (t, smallvalue, 10))
```



```
[28]: <sympy.plotting.plot.Plot at 0x1209627f0>
```

Let's get the values of the controller output as a list:

```
[29]: u = tbcontrol.symbolic.sampledvalues(uz, z, N)
```

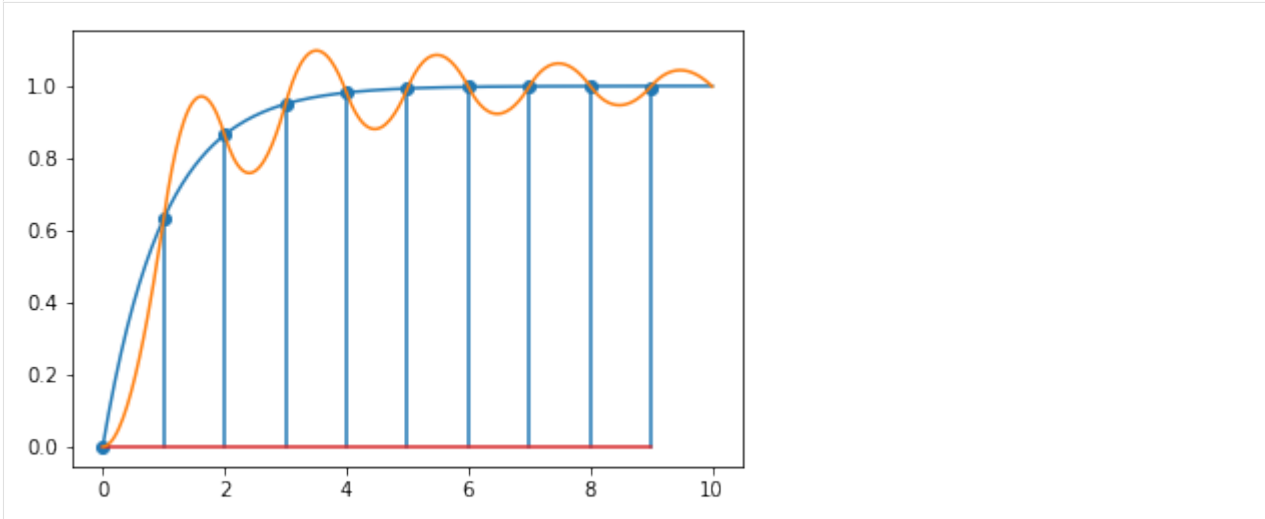
Now, we calculate the output of the system as the sum of the various pulse inputs.


```
[30]: yt = numpy.zeros_like(ts)
      for i in range(0, N):
          yt += [float(sympy.N(u[i]*p.subs(t, ti-i + smallvalue))) for ti in ts]
```

Finally, we present the discrete system response, the designed response and the analytical continuous response on the same graph.

```
[31]: plotdiscrete(yz, 10)
      plt.plot(ts, 1 - numpy.exp(-ts/λ))
      plt.plot(ts, yt)
```

```
[31]: [<matplotlib.lines.Line2D at 0x12126f8d0>]
```



```
[ ]:
```

3.6.8 Simple discrete simulation: Dahlin controller

This notebook replicates Figure 17.11 in Seborg et al without using analytic methods. If you want to get some insight into the math behind the z transform, applied to the same problem see the [Dahlin controller](#) notebook.

```
[1]: import tbcontrol
```

```
[2]: tbcontrol.expectversion('0.2.1')
```

```
[3]: from tbcontrol import blocksim
```

```
[4]: import matplotlib.pyplot as plt
      %matplotlib inline
```

```
[5]: import numpy
      import control
```


Simple discretisation

The control module supplies an easy access point to a transfer function which represents s.

```
[6]: s = control.TransferFunction.s
```

```
[7]: K = 1
     τ1 = 5
     τ2 = 3
     θ = 0
```

```
[8]: Δt = 1
```

```
[9]: assert θ % Δt == 0 # this is only correct when the delay is a multiple of the
     ↪ sampling time
     N = int(θ/Δt)
```

```
[10]: G = K / ((τ1*s + 1)*(τ2*s + 1))
```

Discretize G assuming a zero order hold in front of it, this corresponds to using Table 17.1

```
[11]: z = control.TransferFunction.z
```

```
[12]: Gz = G.sample(Δt)*z**(-N)
     Gz
```

```
[12]: 
$$\frac{0.02797z + 0.02341}{z^2 - 1.535z + 0.5866} \quad dt = 1$$

```

Do discrete transfer function math

Dahlin controller relationship

```
[13]: λ = Δt # this is a tuning parameters
```

```
[14]: A = numpy.exp(-Δt/λ)
     Gdc = 1/Gz * (1 - A)*z**(-N - 1) / (1 - A*z**(-1) - (1 - A)*z**(-N - 1)) # eq 17.65
```

After these calculations Gdc is

```
[15]: Gdc
```

```
[15]: 
$$\frac{0.6321z^4 - 0.9705z^3 + 0.3708z^2}{0.02797z^4 - 0.004556z^3 - 0.02341z^2} \quad dt = 1$$

```

Notice that there are “extra” orders of z above and below the line in that expression - we could simplify it. This is what `.minreal` does.

```
[16]: Gdc = Gdc.minreal()
```

```
[17]: Gdc
```



```
[17]:
```

$$\frac{22.6z^2 - 34.7z + 13.26}{z^2 - 0.1629z - 0.8371} \quad dt = 1$$

Much better.

Convert from positive to negative powers of z

We still have a problem, though, since the control library uses positive powers of z rather than negative powers of z . `tbcontrol.conversion` contains some methods to help convert such polynomials. Also notice that the `.num` and `.den` properties of the control library's `tf` objects are designed for MIMO systems, so they are nested lists corresponding with the index of the output and the input we want. Since we're just looking at one input and one output, we need the first element of the first element (`[0][0]`).

```
[18]: import tbcontrol.conversion
```

```
[19]: Gdc_neg_num, Gdc_neg_den = tbcontrol.conversion.discrete_coeffs_pos_to_neg(Gdc.  
    ↪ num[0][0], Gdc.den[0][0])
```

```
[20]: Gdc_neg_num
```

```
[20]: [22.59988127611586, -34.69674036625465, 13.258134912008916]
```

```
[21]: Gdc_neg_den
```

```
[21]: [1.0, -0.1628886995509321, -0.8371113004490679]
```

Simple blocksim simulation

```
[22]: blocksim_G = blocksim.LTI('G', 'u', 'yu', G.num[0][0], G.den[0][0])
```

```
[23]: blocksim_Gdc = blocksim.DiscreteTF('Gc', 'e', 'u', 1, Gdc_neg_num, Gdc_neg_den)
```

```
[24]: diagram = blocksim.simple_control_diagram(blocksim_Gdc, blocksim_G, ysp=blocksim.  
    ↪ step(starttime=5))
```

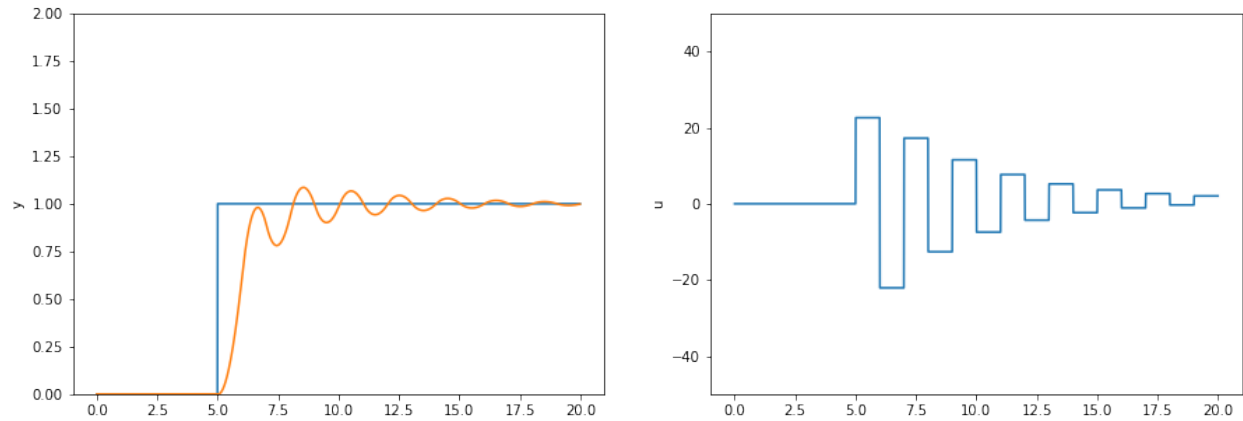
```
[25]: ts = numpy.arange(0, 20, 0.01)
```

```
[26]: result = diagram.simulate(ts)
```

```
[27]: def plot_outputs(result):  
    fig, (ax_y, ax_u) = plt.subplots(1, 2, figsize=(15, 5))  
  
    ax_y.plot(ts, result['ysp'])  
    ax_y.plot(ts, result['y'])  
    ax_y.set_ylim(0, 2)  
    ax_y.set_ylabel('y')  
  
    ax_u.plot(ts, result['u'])  
    ax_u.set_ylim(-50, 50)  
    ax_u.set_ylabel('u')
```



```
[28]: plot_outputs(result)
```



3.6.9 Noise models

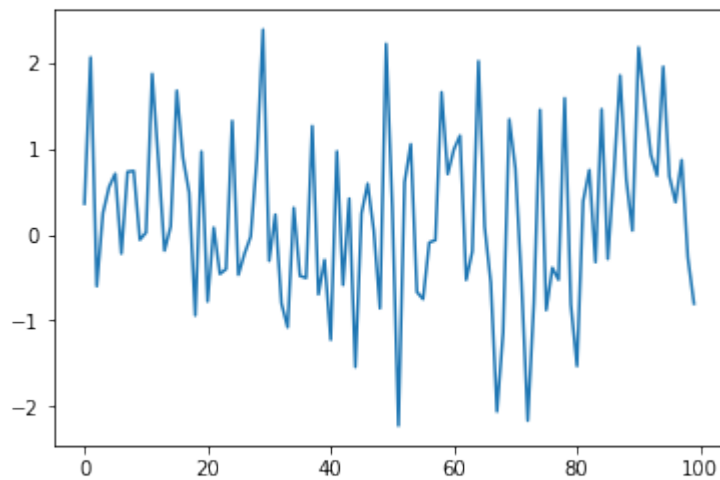
```
[1]: import matplotlib.pyplot as plt
      %matplotlib inline
```

```
[2]: import numpy
```

```
[7]: signal = numpy.random.randn(100)
```

```
[4]: plt.plot(signal)
```

```
[4]: [<matplotlib.lines.Line2D at 0x11cbac940>]
```



```
[5]: import scipy.signal
```

```
[6]: result = []
```

```
[8]: previous = 0
```

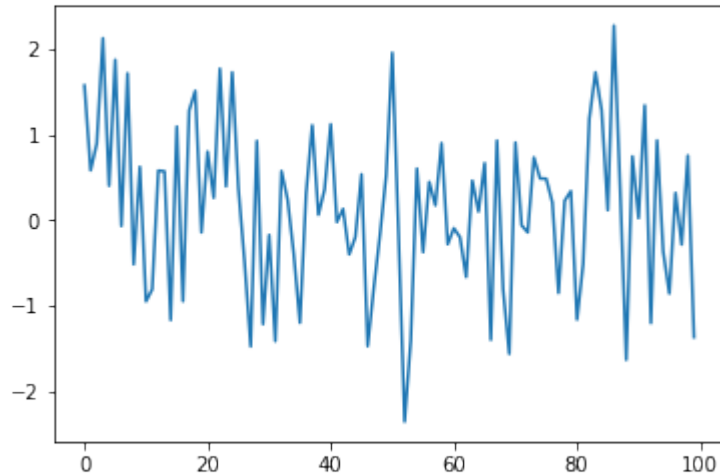


```
[9]: alpha = 0.95
```

```
[10]: for s in signal:
      news = previous*(1-alpha) + alpha*s
      previous = news
      result.append(news)
```

```
[11]: plt.plot(result)
```

```
[11]: [<matplotlib.lines.Line2D at 0x1c1eb4aa20>]
```

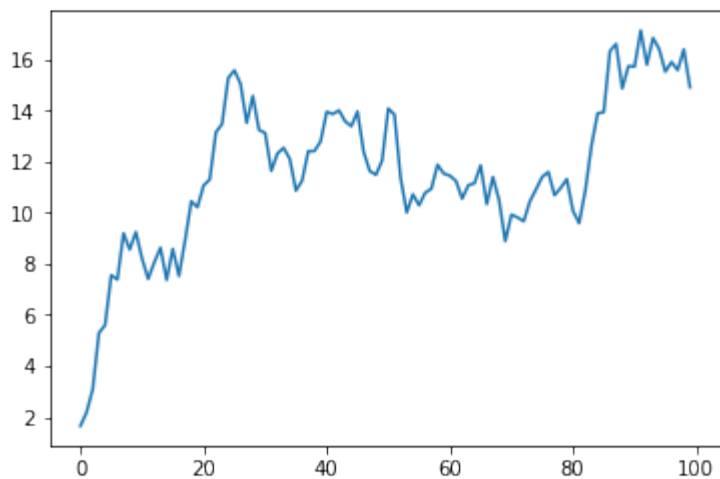


```
[13]: news = 0
```

```
[18]: result = numpy.cumsum(signal)
```

```
[19]: plt.plot(result)
```

```
[19]: [<matplotlib.lines.Line2D at 0x1c1ecd37b8>]
```



```
[ ]:
```

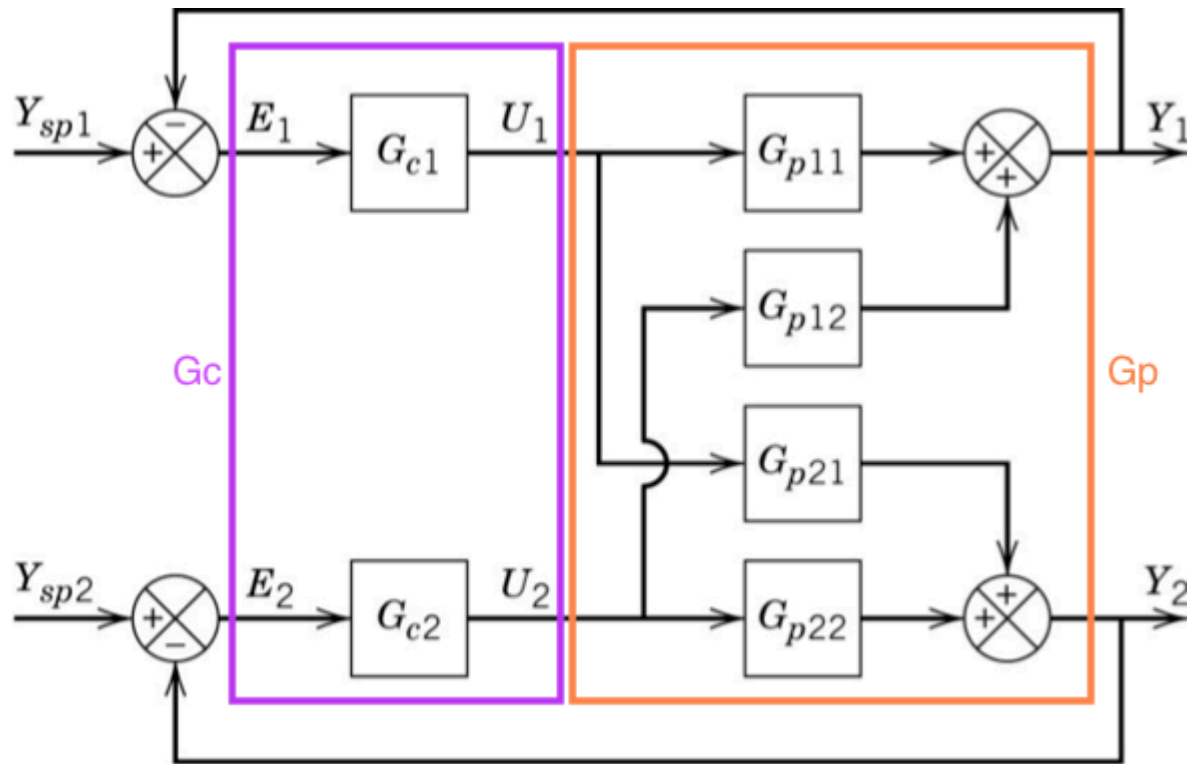

3.7 Multivariable control

3.7.1 Multivariable control

You may want to review the *Transfer function matrix* notebook before you look at this one.

Closed loop transfer functions for multivariable systems

Let's consider a 2×2 system with feedback control as shown below:



```
[1]: import sympy
sympy.init_printing()

[2]: G_p11, G_p12, G_p21, G_p22, G_c1, G_c2 = sympy.symbols('G_p11, G_p12, G_p21, G_p22, G_
↪c1, G_c2')
```

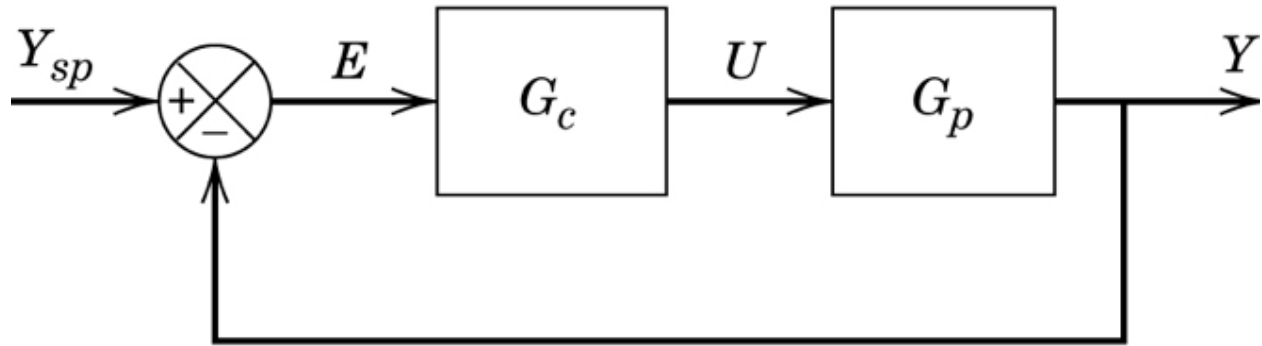
The matrix representation of the system is straightforwardly handled by `sympy.Matrix`:

```
[3]: G_p = sympy.Matrix([[G_p11, G_p12],
                        [G_p21, G_p22]])
```

The controller is a bit harder. Convince yourself you understand how the off-diagonal elements of G_c are zero in the diagram above.

```
[4]: G_c = sympy.Matrix([[G_c1, 0],
                        [0, G_c2]])
```

Now, we can redraw the block diagram using vectors for the signals and matrices for the blocks.



Let's derive the closed loop transfer function. There are three equations represented in the above diagram:

$$E = Y_{sp} - Y \quad (3.7)$$

$$U = G_c E = G_c (Y_{sp} - Y) \quad (3.8)$$

$$Y = G_p U = G_p G_c (Y_{sp} - Y) \quad (3.9)$$

$$(3.10)$$

Now, we can solve for Y :

$$\begin{aligned} Y &= G_p G_c (Y_{sp} - Y) \\ Y &= G_p G_c Y_{sp} - G_p G_c Y \\ Y + G_p G_c Y &= G_p G_c Y_{sp} \\ Y (1 + G_p G_c) &= G_p G_c Y_{sp} \end{aligned}$$

$$Y = (1 + G_p G_c)^{-1} G_p G_c Y_{sp}$$

end{align}

We can calculate the value of this function easily.

```
[5]: I = sympy.Matrix([[1, 0],
                       [0, 1]])
```

```
[6]: Gamma = sympy.simplify((I + G_p*G_c).inv()*G_p*G_c)
```

```
[7]: Gamma[0, 0]
```

$$\frac{G_{c1} (G_{c2} G_{p12} G_{p21} - G_{p11} (G_{c2} G_{p22} + 1))}{G_{c1} G_{c2} G_{p12} G_{p21} - (G_{c1} G_{p11} + 1) (G_{c2} G_{p22} + 1)}$$

```
[8]: Gamma[0, 1]
```

$$-\frac{G_{c2} G_{p12}}{G_{c1} G_{c2} G_{p12} G_{p21} - (G_{c1} G_{p11} + 1) (G_{c2} G_{p22} + 1)}$$

We notice that there is a common divisor in all the elements of Γ . This is due to the calculation of the inverse of $(I + G_p G_c)$, which involves calculation of the determinant $|I + G_p G_c|$.


```
[9]: Delta = (I + G_p*G_c).det()
Delta
```

```
[9]: 
$$G_{c1}G_{c2}G_{p11}G_{p22} - G_{c1}G_{c2}G_{p12}G_{p21} + G_{c1}G_{p11} + G_{c2}G_{p22} + 1$$

```

```
[10]: (Delta*Gamma).simplify()
```

```
[10]: 
$$\begin{bmatrix} G_{c1}(G_{c2}G_{p11}G_{p22} - G_{c2}G_{p12}G_{p21} + G_{p11}) & G_{c2}G_{p12} \\ G_{c1}G_{p21} & G_{c2}(G_{c1}G_{p11}G_{p22} - G_{c1}G_{p12}G_{p21} + G_{p22}) \end{bmatrix}$$

```

Characteristic equation

This leads us to conclude that we can calculate the characteristic equation of the closed loop transfer function as $|I + G_p G_c|$.

Notice that if we wanted the coupling the other way around, we could have worked with the same controller matrix permuted:

```
[11]: G_c*sympy.Matrix([[0, 1], [1, 0]])
```

```
[11]: 
$$\begin{bmatrix} 0 & G_{c1} \\ G_{c2} & 0 \end{bmatrix}$$

```

3.7.2 Multivariable Stability analysis

When we used proportional control on SISO systems we observed that there is usually an upper bound on the controller gain K_c above which the controlled system becomes unstable. Let's investigate the equivalent calculation for MIMO systems.

```
[1]: import sympy
sympy.init_printing()
%matplotlib inline
```

```
[2]: s = sympy.Symbol('s')
```

This matrix is from example 16.2 in Seborg

```
[3]: Gp = sympy.Matrix([[2/(10*s + 1), sympy.Rational('1.5')/(s + 1)],
[sympy.Rational('1.5')/(s + 1), 2/(10*s + 1)]]
Gp
```

```
[3]: 
$$\begin{bmatrix} \frac{2}{10s+1} & \frac{3}{2(s+1)} \\ \frac{3}{2(s+1)} & \frac{2}{10s+1} \end{bmatrix}$$

```

```
[4]: K_c1, K_c2 = sympy.symbols('K_c1, K_c2', real=True)
```

Unlike in SISO systems, we now have a choice of pairing. We will see that there are differences in the stability behaviour for the different pairings.


```
[5]: diagonal = True
```

```
[6]: if diagonal:
      Gc = sympy.Matrix([[K_c1, 0],
                        [0, K_c2]])
    else:
      Gc = sympy.Matrix([[0, K_c2],
                        [K_c1, 0]])
```

```
[7]: I = sympy.Matrix([[1, 0],
                      [0, 1]])
```

The characteristic equation can be obtained from the $|I + G_p G_c|$. I divide by 4 here to obtain a final constant of 1 like in the example to make comparison easier. Make sure you understand that any constant multiple of the characteristic equation will have the same poles and zeros.

```
[8]: charpoly = sympy.poly(sympy.numeral((I + Gp*Gc).det().cancel())/4, s)
```

Compare with Equation 16-20:

```
[9]: charpoly2 = sympy.poly(
      sympy.numeral(
        ((1 + Gc[0,0]*Gp[0,0])*(1 + Gc[1,1]*Gp[1,1]) - Gc[0,0]*Gc[1,1]*Gp[0,1]*Gp[1,
      ↪0])).cancel()
      )/4, s)
```

```
[10]: charpoly == charpoly2
```

```
[10]: True
```

Now that we have a characteristic polynomial, we can determine stability criteria using the `routh` function from `tbcontrol.symbolic`.

```
[11]: from tbcontrol.symbolic import routh
```

```
[12]: R = routh(charpoly)
```

```
[13]: R[0, 0]
```

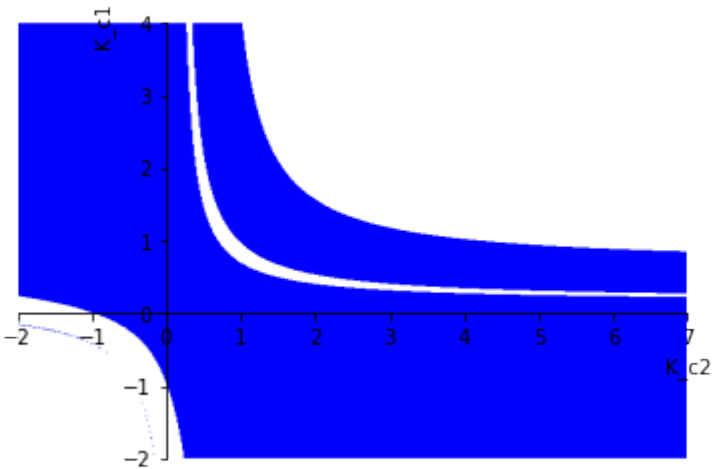
```
[13]: 100
```

All the remaining elements of the left hand row must be positive (the same sign as the first element)

```
[14]: requirements = True
      for r in R[1:, 0]:
          requirements = sympy.And(requirements, r>0)
```

The graph below is supposed to match the textbook, but as of 2019-03-30 it does not. This appears to be a bug in `plot_implicit`.

```
[15]: sympy.plot_implicit(requirements, (K_c2, -2, 7), (K_c1, -2, 4))
```

```
[15]: <sympy.plotting.plot.Plot at 0x1217574e0>
```

As an alternative, let's evaluate numerically

```
[16]: import numpy
```

```
[17]: import matplotlib.pyplot as plt
      %matplotlib inline
```

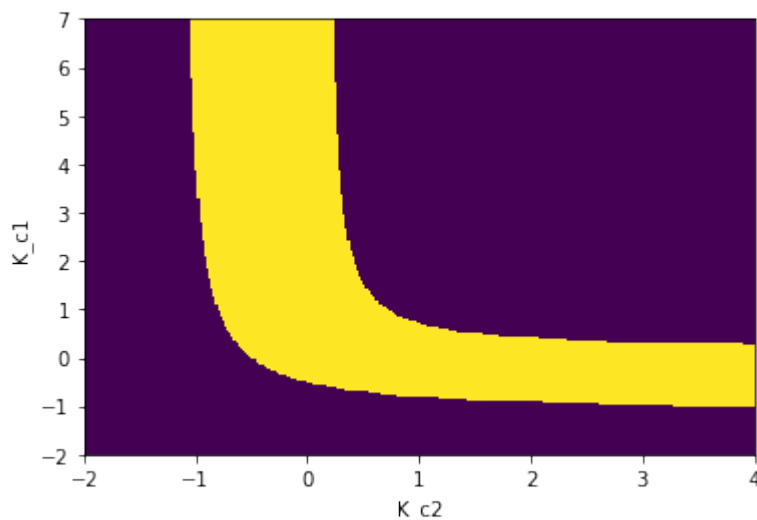
```
[18]: f = sympy.lambdify((K_c2, K_c1), requirements)
```

```
[19]: nK_c2, nK_c1 = numpy.meshgrid(numpy.linspace(-2, 4, 300), numpy.linspace(-2, 7, 300))
```

```
[20]: r = f(nK_c2, nK_c1)
```

```
[21]: plt.pcolor(nK_c2, nK_c1, r)
      plt.ylabel('K_c1')
      plt.xlabel('K_c2')
```

```
[21]: Text(0.5, 0, 'K_c2')
```



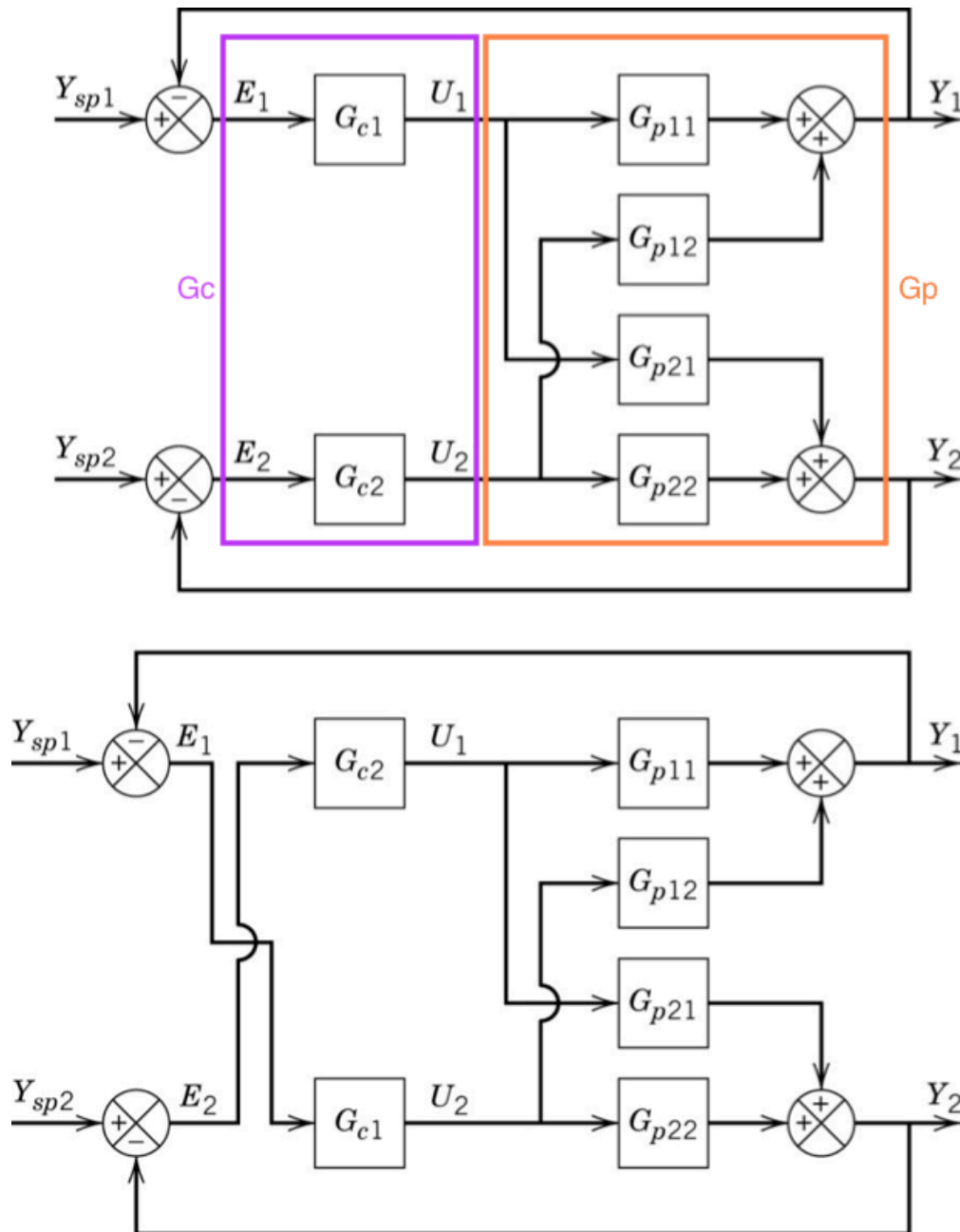
We can see that even this simple system can exhibit more complicated behaviour than we may expect from first order systems because of the extra loops formed by the controllers.

3.7.3 Multivariable pairing (RGA)

For a 2×2 system, we have 2 choices of pairing variables for distributed control:

Diagonal

Off-diagonal



$$G_{cd} = \begin{bmatrix} G_{c1} & 0 \\ 0 & G_{c2} \end{bmatrix}$$

$$G_{co} = \begin{bmatrix} 0 & G_{c2} \\ G_{c1} & 0 \end{bmatrix}$$

Bristol developed the Relative Gain Array to determine good pairings based on only the plant transfer function matrix G_p . The elements of the RGA are defined as

$$\lambda_{ij} \triangleq \frac{(\partial y_i / \partial u_j)_u}{(\partial y_i / \partial u_j)_y} = \frac{\text{open loop gain}}{\text{closed loop gain}}$$

We could build Λ by direct evaluation of the above derivatives near some point given a time-domain model, but if we already have a transfer function model, we can evaluate the steady-state gain matrix K by using the final value theorem.

```
[1]: import sympy
      sympy.init_printing()
```

```
[2]: s = sympy.Symbol('s')
```

```
[3]: def fopdt(k, theta, tau):
      return k*sympy.exp(-theta*s)/(tau*s + 1)
```

Using the system from example 16.5

```
[4]: G_p = sympy.Matrix([[fopdt(-2, 1, 10), fopdt(1.5, 1, 1)],
                        [fopdt(1.5, 1, 1), fopdt(2, 1, 10)]])
      G_p
```

```
[4]:
```

$$\begin{bmatrix} -\frac{2e^{-s}}{10s+1} & \frac{1.5e^{-s}}{s+1} \\ \frac{1.5e^{-s}}{s+1} & \frac{2e^{-s}}{10s+1} \end{bmatrix}$$

Unfortunately sympy cannot calculate limits on matrix expressions

```
[5]: #K = sympy.limit(G_p, s, 0)
```

But we can apply a function to the elements:

```
[6]: def gain(G):
      return sympy.limit(G, s, 0)
```

```
[7]: K = G_p.applyfunc(gain)
```

```
[8]: K
```

```
[8]:
```

$$\begin{bmatrix} -2 & 1.5 \\ 1.5 & 2 \end{bmatrix}$$

We can then calculate $\Lambda = K \otimes H$ where $H = (K^{-1})^T$:


```
[9]: Lambda = K.multiply_elementwise(K.inv().transpose())
      Lambda
```

```
[9]: 
$$\begin{bmatrix} 0.64 & 0.36 \\ 0.36 & 0.64 \end{bmatrix}$$

```

We can do the same calculation (faster) using numpy:

```
[10]: import numpy
```

```
[11]: def fopdt(k, theta, tau):
      return k*numpy.exp(-theta*s)/(tau*s + 1)
```

```
[12]: s = 0
```

```
[13]: K = numpy.matrix([[fopdt(-2, 1, 10), fopdt(1.5, 1, 1)],
      [fopdt(1.5, 1, 1), fopdt(2, 1, 10)]])
```

The `.A` attribute in matrices is the matrix as a `numpy.array`, which multiplies elementwise by default.

```
[14]: K.A*K.I.T.A
```

```
[14]: array([[0.64, 0.36],
      [0.36, 0.64]])
```

The numpy developers recommended that you should use `numpy.array` instead of `numpy.matrix` as much as possible. I find this makes the notation harder to read:

```
[15]: K = numpy.array([[fopdt(-2, 1, 10), fopdt(1.5, 1, 1)],
      [fopdt(1.5, 1, 1), fopdt(2, 1, 10)]])
```

```
[16]: K*numpy.linalg.inv(K).T
```

```
[16]: array([[0.64, 0.36],
      [0.36, 0.64]])
```

Simulation results

Let's simulate this system to get an idea of how the control works out

```
[17]: import tbcontrol
      tbcontrol.expectversion('0.1.4')
      from tbcontrol import blocksim
```

```
[18]: import numpy
```

```
[19]: N = 2
```

```
[20]: G = {}
```



```

[21]: gains = [[-2, 1.5],
              [1.5, 2]]
      taus = [[10, 1],
              [1, 10]]
      delays = [[1, 1],
                [1, 1]]

[22]: for inp in range(N):
      for outp in range(N):
          G[(outp, inp)] = blocksim.LTI(f"G_{outp}_{inp}", f"u_{inp}", f"yp_{inp}_{outp}"
↪",
                                         gains[outp][inp], [1, taus[outp][inp]], ↪
↪delays[outp][inp])

[23]: inputs = {'yssp_0': blocksim.step(),
                'yssp_1': blocksim.step(starttime=50)}

[24]: sums = {'y_{outp}': [f"+yp_{inp}_{outp}" for inp in range(N)] for outp in range(N)}
      for i in range(N):
          sums[f'e_{i}'] = [f'+yssp_{i}', f'-y_{i}']

[25]: sums
[25]: {'y_0': ['+yp_0_0', '+yp_1_0'],
      'y_1': ['+yp_0_1', '+yp_1_1'],
      'e_0': ['+yssp_0', '-y_0'],
      'e_1': ['+yssp_1', '-y_1']}

[26]: import matplotlib.pyplot as plt
      %matplotlib inline

[27]: def simulate(auto1=True, K1=-1, tauI1=10, auto2=True, K2=0.5, tauI2=10):
      controllers = {'Gc_0': blocksim.PI('Gc_0', 'e_0', 'u_0', K1, tauI1),
                    'Gc_1': blocksim.PI('Gc_1', 'e_1', 'u_1', K2, tauI2)}

      controllers['Gc_0'].automatic = auto1
      controllers['Gc_1'].automatic = auto2

      ts = numpy.arange(0, 100, 0.125)

      diagram = blocksim.Diagram(list(G.values()) + list(controllers.values()), sums, ↪
↪inputs)

      result = diagram.simulate(ts)

      # plt.figure()
      # plt.plot(ts, result['u_0'])
      # plt.plot(ts, result['u_1'])

      plt.figure()
      plt.plot(ts, result['y_0'])
      plt.plot(ts, result['yssp_0'])
      plt.plot(ts, result['y_1'])
      plt.plot(ts, result['yssp_1'])

```



```
[28]: from ipywidgets import interact

[29]: interact(simulate,
              auto1=[True, False], K1=(-2., 0), tauI1=(1., 50),
              auto2=[True, False], K2=(0., 2), tauI2=(1., 50))

interactive(children=(Dropdown(description='auto1', options=(True, False),
↵value=True), FloatSlider(value=-1.0...

[29]: <function __main__.simulate(auto1=True, K1=-1, tauI1=10, auto2=True, K2=0.5,
↵tauI2=10)>

[1]: import numpy
import matplotlib.pyplot as plt

[2]: %matplotlib inline
```

3.7.4 Eigenvalue problem

Matrix transformation is written as

$$\mathbf{y} = A\mathbf{x}$$

A different vector in the same direction can be written as scalar multiplication:

$$\mathbf{y} = \lambda\mathbf{x}$$

Equating these \mathbf{y} s yields:

$$A\mathbf{x} = \lambda\mathbf{x} \Rightarrow (A - \lambda I)\mathbf{x} = 0$$

$$\det(A - \lambda I) = 0$$

The eigenvalue problem can also be collected with Λ being a diagonal matrix containing all the eigenvalues and X containing the eigenvectors stacked column-wise. This leads to the eigenvalue decomposition:

$$AX = X\Lambda \Rightarrow A = X\Lambda X^{-1}$$

with

$$\Lambda = \text{diag}(\lambda_i)$$

If we try to find a similar decomposition with different constraints, we can write

$$A = UDV^H$$

If D is a diagonal matrix and U and V are **unitary**, this is the singular value decomposition.

In Skogestad

$$A = U\Sigma V^H$$

$$\Sigma = \text{diag}(\sigma_i)$$


```
[3]: from ipywidgets import interact
```

```
[4]: def plotvector(x, color='blue'):
      plt.plot([0, x[0,0]], [0, x[1,0]], color=color)
```

```
[5]: import matplotlib.patches as patches
```

Let's investigate the properties of this matrix:

```
[77]: A = numpy.matrix([[4, 3],
                       [2, 1]])
```

The eigenvectors and eigenvalues can be calculated as follows. We also calculate the output vectors associated with a unit vector input in the eigenvector directions.

```
[78]: A
```

```
[78]: matrix([[4, 3],
             [2, 1]])
```

```
[79]: v = numpy.asmatrix(numpy.random.random(2)).T
```

```
[80]: v = A*v
```

```
v = v/numpy.linalg.norm(v)
v
```

```
[80]: matrix([[0.89474813],
             [0.44657115]])
```

```
[81]: lambdas, eigenvectors = numpy.linalg.eig(A)
ev1 = lambdas[0]*eigenvectors[:, 0]
ev2 = lambdas[1]*eigenvectors[:, 1]
```

The singular values determine the main axes of the translation ellipse of the matrix. Note that the `numpy.linalg.svd` function returns the conjugate transpose of the input direction matrix.

```
[82]: U, S, VH = numpy.linalg.svd(A)
V = VH.H
ellipseangle = numpy.rad2deg(numpy.angle(complex(*U[:, 0])))
```

```
[83]: def interactive(scale, theta):
      x = numpy.matrix([[numpy.cos(theta)], [numpy.sin(theta)]]
                        y = A*x

      plotvector(x)
      plotvector(y, color='red')
      plotvector(ev1, 'green')
      plotvector(ev2, 'green')
      plotvector(V[:, 0], 'magenta')
      plotvector(V[:, 1], 'magenta')
      plt.gca().add_artist(patches.Circle([0, 0], 1,
                                          color='blue',
                                          alpha=0.1))
      plt.gca().add_artist(patches.Ellipse([0, 0], S[0]*2, S[1]*2,
```

(continues on next page)

(continued from previous page)

```

        ellipseangle,
        color='red',
        alpha=0.1))

plt.axis([-scale, scale, -scale, scale])
plt.axes().set_aspect('equal')
plt.show()
interact(interactive, scale=(1., 10), theta=(0., numpy.pi*2))

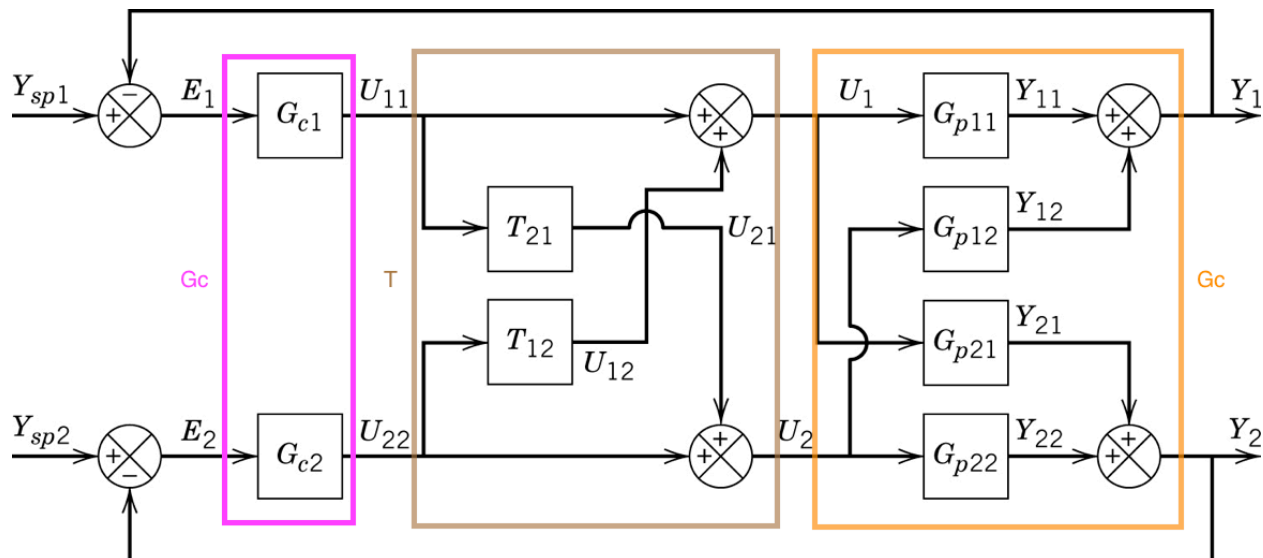
interactive(children=(FloatSlider(value=5.5, description='scale', max=10.0, min=1.0),
    ↳FloatSlider(value=3.141592653589793, description='theta', max=6.283185307179586),
    ↳Output()), _dom_classes=('widget-interact',))

```

[83]: <function __main__.interactive>

3.7.5 Decoupling

Given a general multivariable system with transfer function matrix G_p , a decoupler attempts to combine with the system to form a diagonal whole.



```
[1]: import sympy
sympy.init_printing()
```

```
[2]: G_p11, G_p12, G_p21, G_p22 = sympy.symbols('G_p11, G_p12, G_p21, G_p22')
G_p = sympy.Matrix([[G_p11, G_p12], [G_p21, G_p22]])
G_p
```

```
[2]:
```

$$\begin{bmatrix} G_{p11} & G_{p12} \\ G_{p21} & G_{p22} \end{bmatrix}$$

1. Inverse-based

Wouldn't it be nice if the system didn't have interaction? In other words, we could choose T such that we have this system with the same diagonal elements as the original system but zeros in the off diagonals.

```
[3]: G_s = sympy.Matrix([[G_p11, 0], [0, G_p22]])
      G_s
```

```
[3]: 
$$\begin{bmatrix} G_{p11} & 0 \\ 0 & G_{p22} \end{bmatrix}$$

```

Recalling that the combination of T and G_p in series is $G_p T$, we can solve for the decoupler directly

$$G_p T = G_s \therefore T = G_p^{-1} G_s$$

```
[5]: T = G_p.inv()*G_s
      T
```

```
[5]: 
$$\begin{bmatrix} \frac{G_{p11}G_{p22}}{G_{p11}G_{p22}-G_{p12}G_{p21}} & -\frac{G_{p12}G_{p22}}{G_{p11}G_{p22}-G_{p12}G_{p21}} \\ -\frac{G_{p11}G_{p21}}{G_{p11}G_{p22}-G_{p12}G_{p21}} & \frac{G_{p11}G_{p22}}{G_{p11}G_{p22}-G_{p12}G_{p21}} \end{bmatrix}$$

```

Let's see if that worked:

```
[6]: G_pT = G_p*T
      sympy.simplify(G_pT)
```

```
[6]: 
$$\begin{bmatrix} G_{p11} & 0 \\ 0 & G_{p22} \end{bmatrix}$$

```

Pros:

- Controller design can be based on open loop model
- Apparent dynamics (what the controller sees) are simple

Cons:

- T is often not physically realisable
- T is complicated

2. Zero off-diagonals

A more common strategy is to solve directly for the off-diagonal elements of set equal to zero.

So we just want

$$G_p T = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$$

Note the difference between the first method and this one - here we are not specifying the diagonal at all, we just want the off-diagonals to be zero.


```
[7]: T21, T12 = sympy.symbols('T21, T12')
T = sympy.Matrix([[1, T12],
                  [T21, 1]])

wantdiagonal = G_p*T

sol = sympy.solve([wantdiagonal[0,1], wantdiagonal[1, 0]], [T21, T12])
```

```
[8]: T.subs(sol)
```

```
[8]:
```

$$\begin{bmatrix} 1 & -\frac{G_{p12}}{G_{p11}} \\ -\frac{G_{p21}}{G_{p22}} & 1 \end{bmatrix}$$

So this is the classic/traditional decoupler shown in the diagram (with unit passthrough on the diagonals). This changes the transfer function the controller “sees” to

```
[9]: G_p*T.subs(sol)
```

```
[9]:
```

$$\begin{bmatrix} G_{p11} - \frac{G_{p12}G_{p21}}{G_{p22}} & 0 \\ 0 & G_{p22} - \frac{G_{p12}G_{p21}}{G_{p11}} \end{bmatrix}$$

Pros:

- Relatively simple design process
- Less complicated decoupler than the inverse-based method

Cons:

- Apparent plant may be higher order than the actual plant
- Still requires an inverse, may not be physically realisable (but more likely than method 1)

3. Adjugate method

The adjugate (previously called the adjoint) of a matrix will also diagonalise a system

```
[10]: T = G_p.adjugate()
T
```

```
[10]:
```

$$\begin{bmatrix} G_{p22} & -G_{p12} \\ -G_{p21} & G_{p11} \end{bmatrix}$$

```
[11]: G_p*T
```

```
[11]:
```

$$\begin{bmatrix} G_{p11}G_{p22} - G_{p12}G_{p21} & 0 \\ 0 & G_{p11}G_{p22} - G_{p12}G_{p21} \end{bmatrix}$$

Pros:

- Decoupler guaranteed to be physically realisable because it only requires “forward” models of the system.

Cons:

- Apparent plant now much higher order (look at the products in the $G_p T$ expression)

```
[ ]:
```

3.7.6 Model Predictive Control

The general idea of figuring out what moves to make using optimisation at each time step has become very popular due to the fact that a general version can be programmed and made very user friendly so that the intricacies of multivariable control can be handled by a single program.

In this notebook I will show how a single time step's move trajectory is calculated. We'll use the same system as we used for the *Dahlin controller*

```
[1]: import numpy
import scipy.signal
import scipy.optimize
import matplotlib.pyplot as plt
%matplotlib inline
```

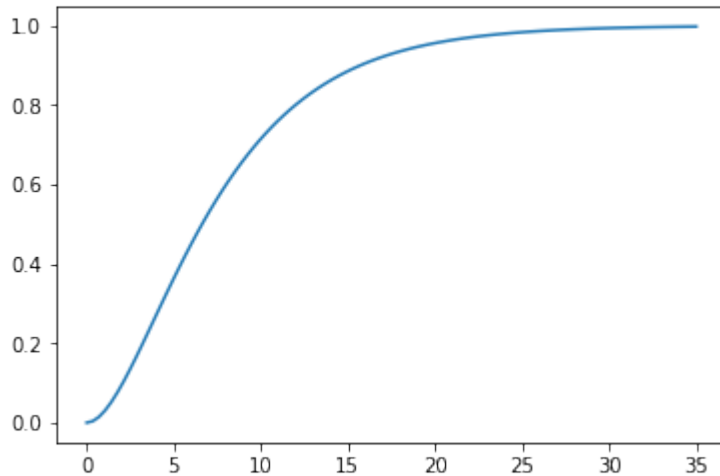
We start with a linear model of the system

$$G = \frac{1}{15s^2 + 8s + 1}$$

```
[2]: G = scipy.signal.lti([1], [15, 8, 1])
```

```
[3]: plt.plot(*G.step())
```

```
[3]: [<matplotlib.lines.Line2D at 0x1c146a8390>]
```



Our goal is to find out what manipulations must be made (changes to u) in order to get the system to follow a specific desired trajectory (which we will call r for the reference trajectory). We will allow the controller to make a certain number of moves. This is called the control horizon, M . We will observe the effect of this set of moves (called a “move plan”) for time called the prediction horizon (P).

Controller parameters


```
[4]: M = 10 # Control horizon
      P = 20 # Prediction horizon
      DeltaT = 1 # Sampling rate
```

```
[5]: tcontinuous = numpy.linspace(0, P*DeltaT, 1000) # some closely spaced time points
      tpredict = numpy.arange(0, P*DeltaT, DeltaT) # discrete points at prediction horizon
```

We choose a first order setpoint response similar to DS or Dahlin

```
[6]: tau_c = 1
      r = 1 - numpy.exp(-tpredict/tau_c)
```

For an initial guess we choose a step in u .

```
[7]: u = numpy.ones(M)
```

Initial state is zero

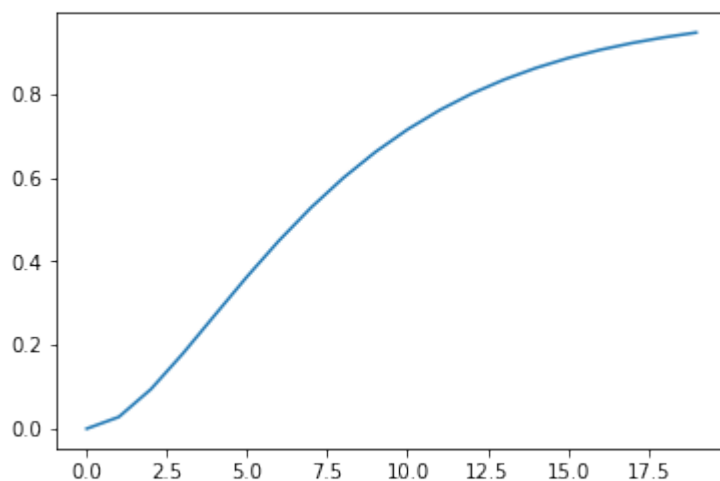
```
[8]: x0 = numpy.zeros(G.to_ss().A.shape[0])
```

```
[9]: def extend(u):
      """We optimise the first M values of u but we need P values for prediction"""
      return numpy.concatenate([u, numpy.repeat(u[-1], P-M)])
```

```
[10]: def prediction(u, t=tpredict, x0=x0):
      """Predict the effect of an input signal"""
      t, y, x = scipy.signal.lsim(G, u, t, X0=x0, interp=False)
      return y
```

```
[11]: plt.plot(tpredict, prediction(extend(u)))
```

```
[11]: [<matplotlib.lines.Line2D at 0x1c1475ccf8>]
```



```
[12]: def objective(u, x0=x0):
      """Calculate the sum of the square error for the control problem"""
      y = prediction(extend(u))
      return sum((r - y)**2)
```

This is the value of the objective for our step input:


```
[13]: objective(u)
[13]: 3.506966650333838
```

Now we figure out a set of moves which will minimise our objective function

```
[14]: result = scipy.optimize.minimize(objective, u)
      uopt = result.x
      result.fun
[14]: 0.0009187720232727162
```

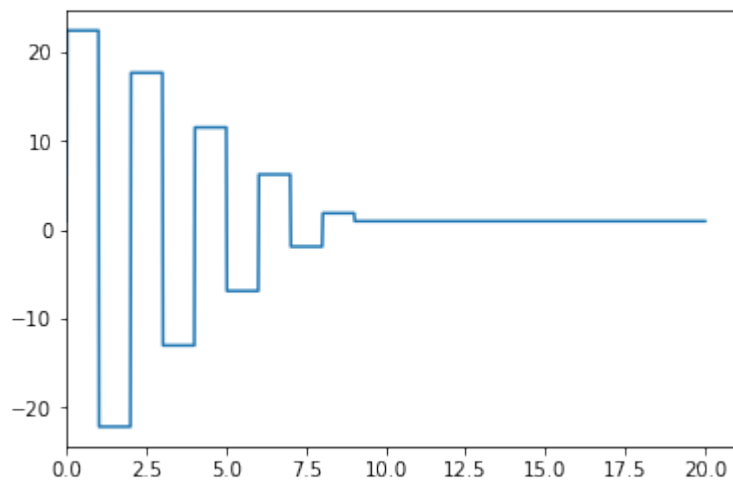
Resample the discrete output to continuous time (effectively work out the 0 order hold value)

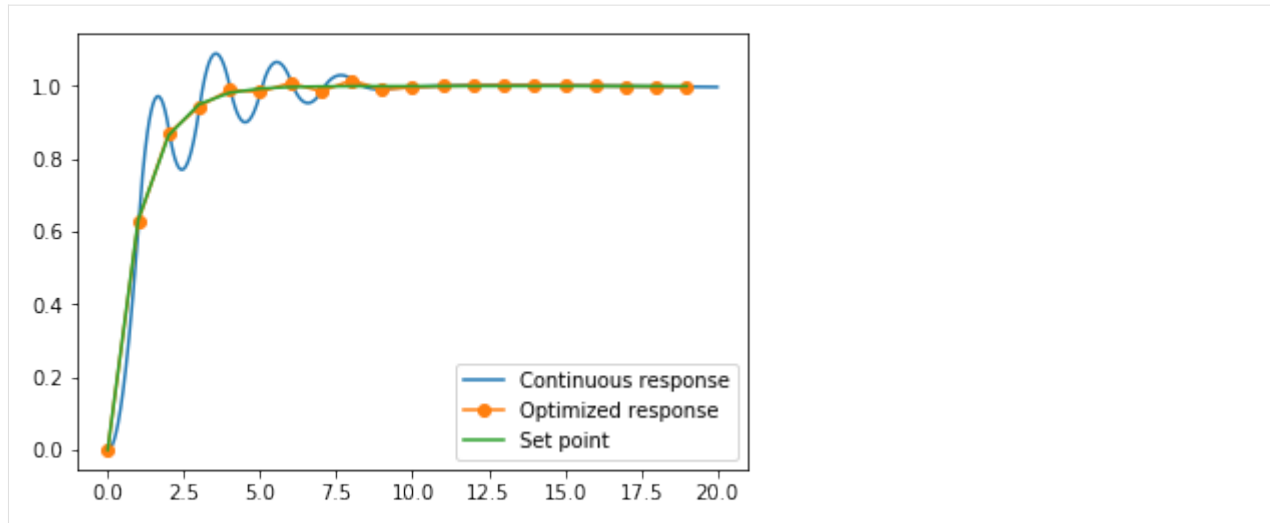
```
[15]: ucont = extend(uopt)[((tcontinuous-0.01)//DeltaT).astype(int)]
```

Plot the move plan and the output. Notice that we are getting exactly the output we want at the sampling times. At this point we have effectively recovered the Dahlin controller.

```
[16]: def plotoutput(ucont, uopt):
      plt.figure()
      plt.plot(tcontinuous, ucont)
      plt.xlim([0, DeltaT*(P+1)])
      plt.figure()
      plt.plot(tcontinuous, prediction(ucont, tcontinuous), label='Continuous response')
      plt.plot(tpredict, prediction(extend(uopt)), '-o', label='Optimized response')
      plt.plot(tpredict, r, label='Set point')
      plt.legend()
```

```
[17]: plotoutput(ucont, uopt)
```





One of the reasons for the popularity of MPC is how easy it is to change its behaviour using weights in the objective function. Try using this definition instead of the simple one above and see if you can remove the ringing in the controller output.

```
[18]: def objective(u, x0=x0):
    y = prediction(extend(u))
    umag = numpy.abs(u)
    constraintpenalty = sum(umag[umag > 2])
    movepenalty = sum(numpy.abs(numpy.diff(u)))
    strongfinish = numpy.abs(y[-1] - r[-1])
    return sum((r - y)**2) + 0*constraintpenalty + 0.1*movepenalty + 0*strongfinish
```

3.8 Control Practice

3.8.1 Control valve design

This is example 8.2 in Seborg, but worked a little differently to allow choice of R and C_{cv}

```
[1]: import numpy
import scipy.optimize
import matplotlib.pyplot as plt
from ipywidgets import interact
%matplotlib inline
```

```
[2]: # Constant pump head
DeltaPa = 40
# Guess for q
q0 = 100
```

The MEB reduces to quadratic form:

$$\Delta P_a = \Delta P_{hc} + \Delta P_v$$

$$\Delta P_a - a_{hc}q^2 - a_vq^2 = 0$$


```
[3]: def MEBcoeffs(l, R, Ccv, characteristic='eqperc'):
    ahc = 30/200**2
    if characteristic == 'linear':
        fl = 1
    elif characteristic == 'eqperc':
        fl = R*(1 - l)
    av = (1/(Ccv*fl))**2

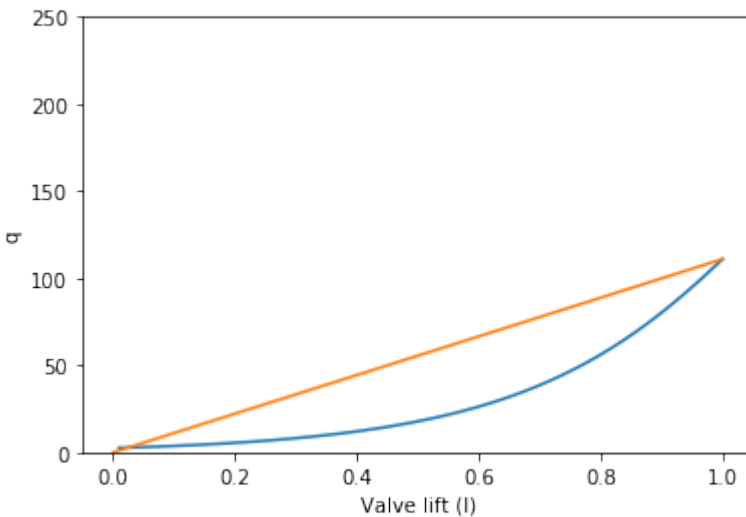
    return [-ahc - av, 0, DeltaPa]
```

```
[4]: def positive(x):
    return x[x>0][0]
```

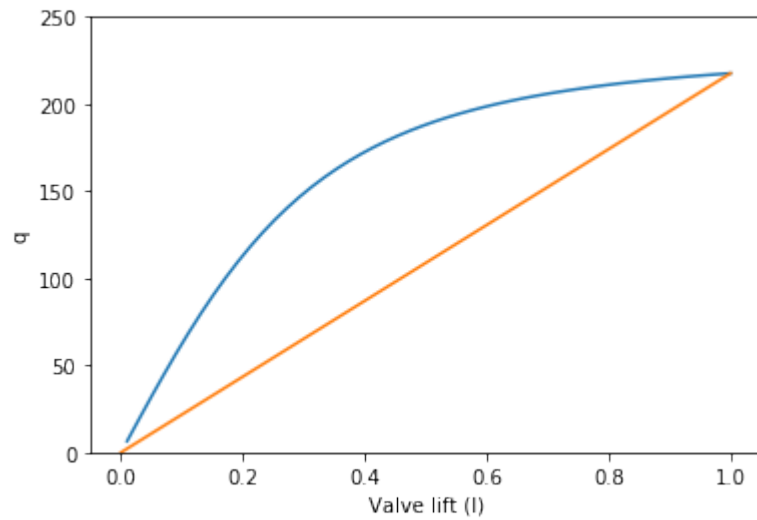
```
[5]: ls = numpy.linspace(0.01, 1)
```

```
[6]: def curve(R, Ccv, characteristic):
    qs = [positive(numpy.roots(MEBcoeffs(l, R, Ccv, characteristic))) for l in ls]
    plt.plot(ls, qs)
    plt.plot([0, 1], [0, max(qs)])
    plt.xlabel('Valve lift (l)')
    plt.ylabel('q')
    plt.ylim([0, 250])
```

```
[7]: curve(50, 20, 'eqperc')
```



```
[8]: interact (curve,
               R=(5., 100.),
               Ccv=(5., 200.),
               characteristic=['linear', 'eqperc'])
```

```
[8]: <function __main__.curve>
```


SIMULATION

This notebook times a couple of ways of integrating a number of tanks in series

```
[1]: import matplotlib.pyplot as plt
    %matplotlib inline
```

```
[2]: Ntimes = 1000
    Nstates = 20
    t_end = 100
    D = 2
```

```
[3]: import numpy
```

4.1 No delays

4.1.1 Normal way

Use a growing list for the history and an array for states

```
[4]: ts = numpy.linspace(0, t_end, Ntimes)
    dt = ts[1]
```

```
[5]: Fin = 2
    k = 1
    A = 1
```

```
[6]: %%time
    statehistory = []
    states = numpy.ones(Nstates)
    for i, t in enumerate(ts):
        Fout = states[0]*k
        dsdt = [1/A*(Fin - Fout)]
        for j in range(1, Nstates):
            Fin_j = k*states[j - 1]
            Fout_j = k*states[j]
            dsdt.append(1/A*(Fin_j - Fout_j))

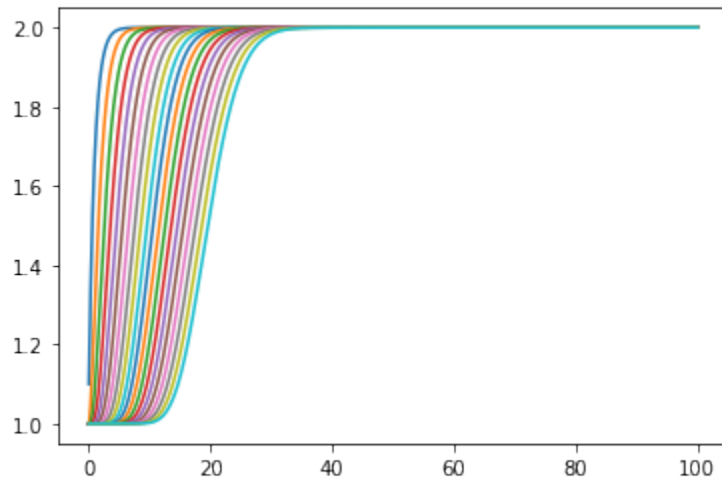
        states += numpy.array(dsdt)*dt

    # we have to copy because the above += is in place.
    statehistory.append(states.copy())
```



```
Wall time: 32 ms
```

```
[7]: plt.plot(ts, statehistory);
```



4.1.2 Preallocation

If you are used to Matlab you may imagine pre-allocating statehistory would save lots of time

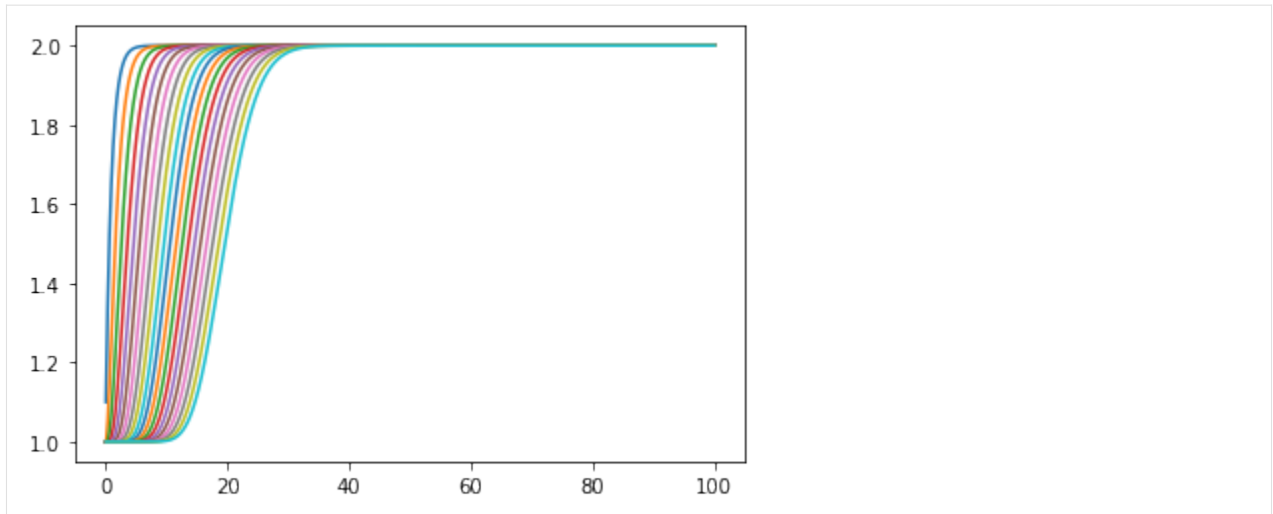
```
[8]: %%time
statehistory = numpy.empty((Ntimes, Nstates))
states = numpy.ones(Nstates)
for i, t in enumerate(ts):
    Fout = states[0]*k
    dsdt = [1/A*(Fin - Fout)]
    for j in range(1, Nstates):
        Fin_j = k*states[j - 1]
        Fout_j = k*states[j]
        dsdt.append(1/A*(Fin_j - Fout_j))

    states += numpy.array(dsdt)*dt

    statehistory[i, :] = states
```

```
Wall time: 32 ms
```

```
[9]: plt.plot(ts, statehistory);
```

Same result, much the same amount of time.

4.2 Dead time

Now, let's introduce a delay between each tank.

4.2.1 Lists and interp

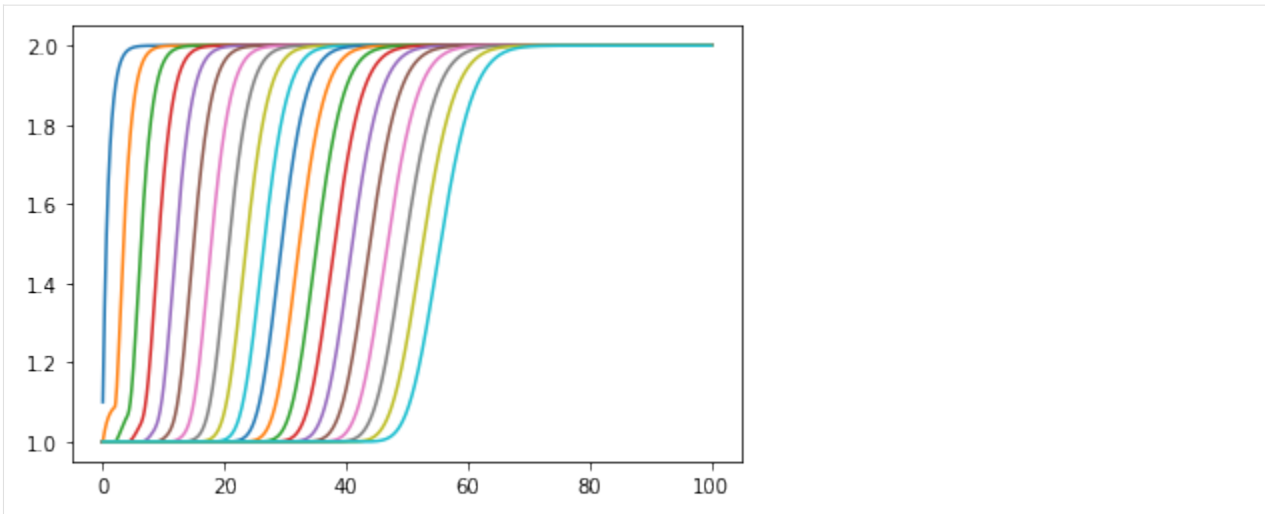
```
[10]: %%time
statehistory = [[] for _ in states]
states = numpy.ones(Nstates)
for i, t in enumerate(ts):
    Fout = states[0]*k
    dsdt = [1/A*(Fin - Fout)]
    for j in range(1, Nstates):
        delayed_Fin_j = k*(numpy.interp(t - D, ts[:i], statehistory[j-1]) if t > 0_
↪ else states[j-1])
        Fout_j = k*states[j]
        dsdt.append(1/A*(delayed_Fin_j - Fout_j))

    states += numpy.array(dsdt)*dt

    for j, s in enumerate(states):
        statehistory[j].append(s)
```

Wall time: 639 ms

```
[11]: plt.plot(ts, numpy.array(statehistory).T);
```

OK, that took a lot longer.

4.2.2 Approximate indexing

What if we just use indexing instead of interpolation?

```
[12]: %%time
statehistory = [[] for _ in states]
states = numpy.ones(Nstates)
for i, t in enumerate(ts):
    Fout = states[0]*k
    dsdt = [1/A*(Fin - Fout)]
    for j in range(1, Nstates):
        delayed_Fin_j = k*(statehistory[j-1][i - int(D/dt)] if t > D else states[j-1])
        Fout_j = k*states[j]
        dsdt.append(1/A*(delayed_Fin_j - Fout_j))

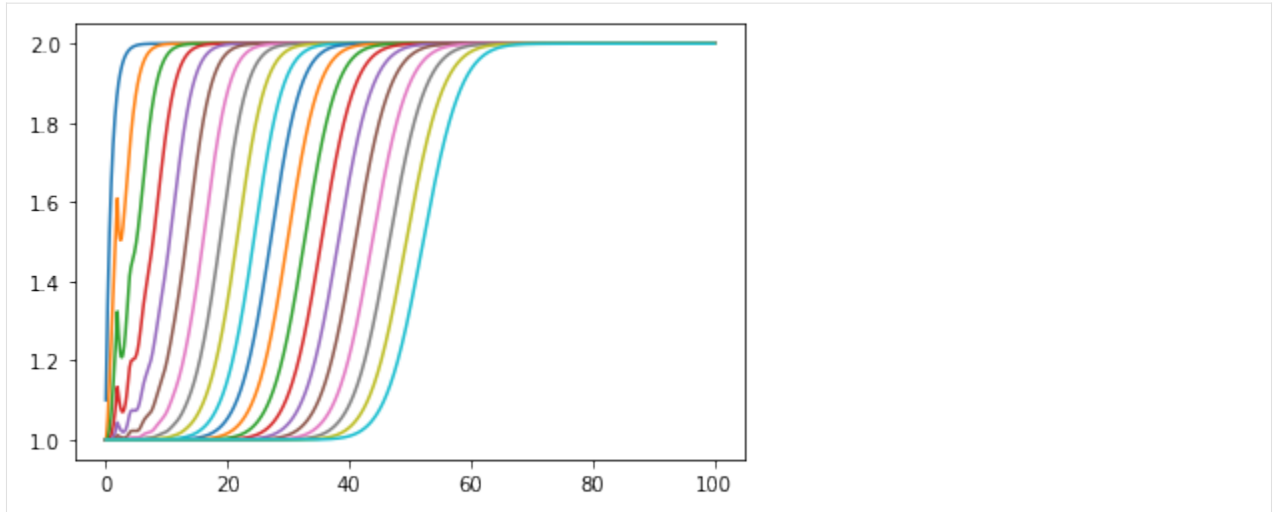
    states += numpy.array(dsdt)*dt

    for j, s in enumerate(states):
        statehistory[j].append(s)
```

Wall time: 49 ms

OK, we're back to almost the same time as before, but do we get the same result?

```
[13]: plt.plot(ts, numpy.array(statehistory).T);
```

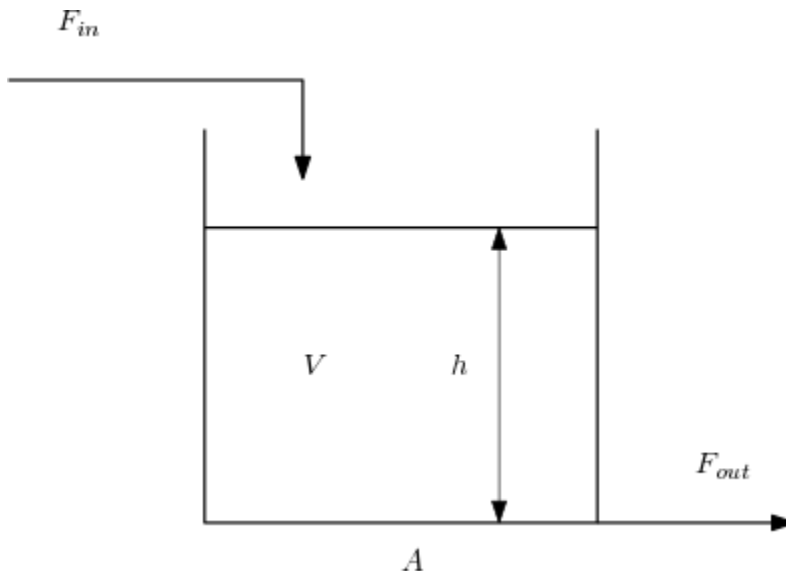



No, the rounding errors build up. If we use this strategy we had better choose a step size which divides cleanly into the dead time.

We covered the idea of simulating an arbitrary transfer function system in [a previous notebook](#). What happens if we have to simulate a controller (specified as a transfer function) and a system specified by differential equations together?

4.3 Nonlinear tank system

Let's take the classic tank system, with a square root flow relationship on the outflow and a nonlinear valve relationship.



```
:nbphinx-math: begin{align}
    \frac{dV}{dt} &= (F_{in} - F_{out}) \setminus h \setminus \frac{V}{A} \setminus f(x) \setminus \alpha^{x-1} \setminus F_{out} \setminus K f(x) \sqrt{h} \setminus
end{align}
```

```
[1]: import numpy
import matplotlib.pyplot as plt
%matplotlib inline
```


Parameters

```
[2]: A = 2
      alpha = 20
      K = 2
```

Initial conditions (notice I'm not starting at steady state)

```
[3]: Fin = 1
      h = 1
      V = A*h
      x0 = x = 0.7
```

Valve characterisitic

```
[4]: def f(x):
      return alpha**(x - 1)
```

Integration time

```
[5]: ts = numpy.linspace(0, 100, 1000)
      dt = ts[1]
```

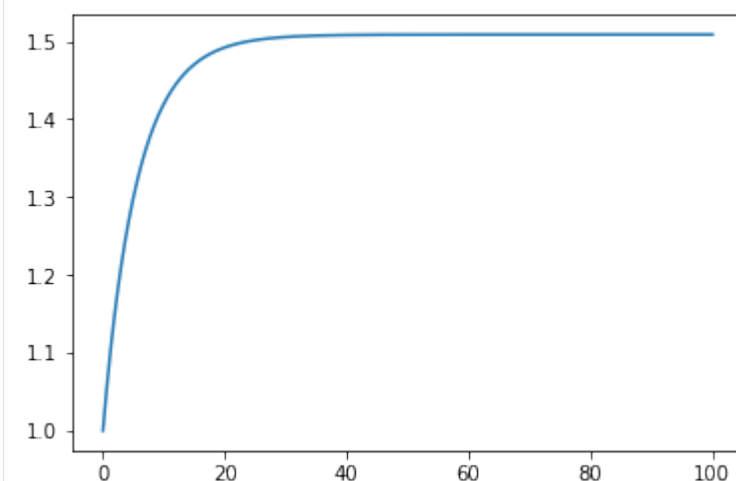
Notice that I have reordered the equations here so that they can be evaluated in order to find the volume derivative.

```
[6]: hs = []
      for t in ts:
          h = V/A
          Fout = K*f(x)*numpy.sqrt(h)
          dVdt = Fin - Fout
          V += dVdt*dt

          hs.append(h)
```

```
[7]: plt.plot(ts, hs)
```

```
[7]: [<matplotlib.lines.Line2D at 0x10e93ca20>]
```



4.4 PI Control

Now we can include a controller controlling the level by manipulating the valve fraction

```
[8]: import scipy.signal
```

Let's do a PI controller:

$$G_c = K_c \left(1 + \frac{1}{\tau_I s} \right) = \frac{K_C \tau_I s + K_C s^0}{\tau_I s + 0 s^0}$$

```
[9]: Kc = -1
     tau_i = 5
```

In versions of scipy < 1.0, Gc would automatically have a .A attribute. After 1.0, we need to convert to state space explicitly with .to_ss().

```
[10]: Gc = scipy.signal.lti([Kc*tau_i, Kc], [tau_i, 0]).to_ss()
```

```
[11]: hsp = 1.3
```

```
[12]: V = 1
     hs = []
     xc = numpy.zeros([Gc.A.shape[0], 1])
     for t in ts:
         h = V/A
         e = hsp - h # we cheat a little here - the level we use to calculate the error
                     ↪ is from the previous time step

         # e is in the input to the controller, yc is the output
         dxcdt = Gc.A.dot(xc) + Gc.B.dot(e)
         yc = Gc.C.dot(xc) + Gc.D.dot(e)

         x = x0 + yc[0,0] # x0 is the controller bias

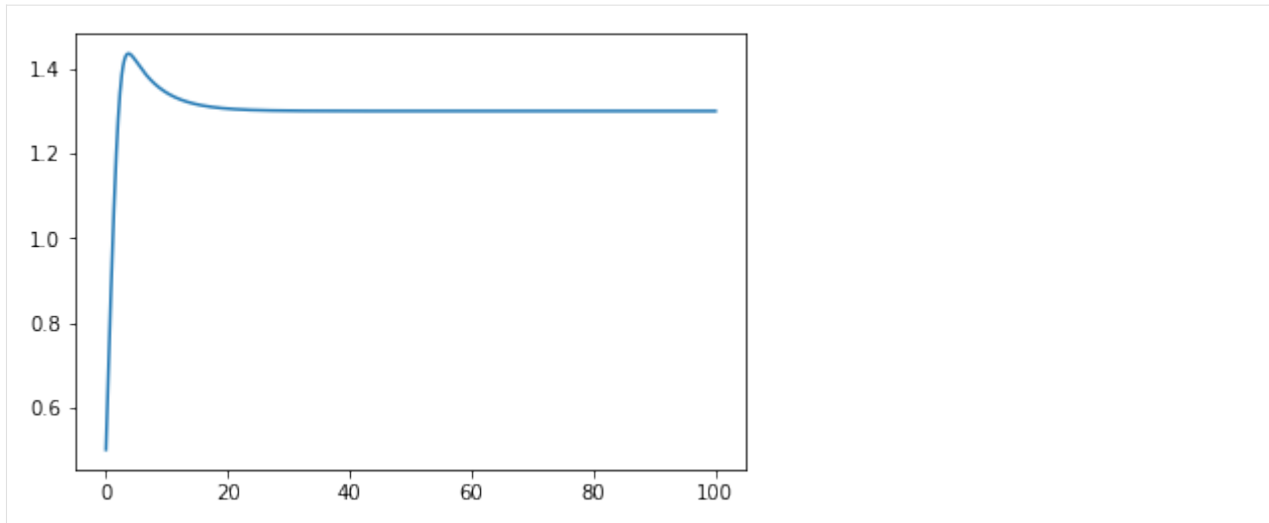
         Fout = K*f(x)*numpy.sqrt(h)
         dVdt = Fin - Fout

         V += dVdt*dt
         xc += dxcdt*dt

         hs.append(h)
```

```
[13]: plt.plot(ts, hs)
```

```
[13]: [<matplotlib.lines.Line2D at 0x1c1a0f31d0>]
```

4.5 Classes

You are already familiar with defining your own functions, like this:

```
[1]: def f(a):  
      return a
```

```
[2]: f(2)
```

```
[2]: 2
```

But what about defining your own types? First, let's remind ourselves of some built-in types:

```
[3]: a = 2
```

```
[4]: type(a)
```

```
[4]: int
```

OK, so there is a thing called an `int`.

```
[5]: int
```

```
[5]: int
```

We can make a new `int` by calling the type name as a function:

```
[6]: int()
```

```
[6]: 0
```

To build our own type, we can use the `class` keyword:

```
[7]: class MyClass:  
      pass
```



```
[8]: MyClass
[8]: __main__.MyClass

[9]: instance = MyClass()

[10]: type(instance)
[10]: __main__.MyClass
```

4.5.1 What is this good for?

The essence of object-oriented programming is allowing for data and algorithms to be combined in one place. Functions allow us to re-use algorithms, but classes allow us to combine them with local data:

```
[11]: class MyClass2:
        def __init__(self, name):
            print("I'm in __init__!")
            self.name = name

        def say_my_name(self):
            print(self.name)
```

```
[12]: s = MyClass2('Carl')
I'm in __init__!
```

```
[13]: s.say_my_name()
Carl
```

4.5.2 Objects must be “like” things

```
[14]: class Person:
        def __init__(self, name):
            self.name = name

        def display_name(self):
            print(self.name)
```

```
[15]: carl = Person("Carl Sandrock")
```

```
[16]: carl.display_name()
Carl Sandrock
```

```
[ ]:
```


4.6 Taking off the engine cover

How does Python “know” how to add two objects? Or what they should look like when printed to the console? Let’s dig into the underlying mechanisms that Python provides for this.

```
[1]: a = 2  
     b = 3
```

```
[2]: a + b
```

```
[2]: 5
```

What methods does `a` have?

```
[3]: dir(a)
```

```
[3]: ['__abs__',  
     '__add__',  
     '__and__',  
     '__bool__',  
     '__ceil__',  
     '__class__',  
     '__delattr__',  
     '__dir__',  
     '__divmod__',  
     '__doc__',  
     '__eq__',  
     '__float__',  
     '__floor__',  
     '__floordiv__',  
     '__format__',  
     '__ge__',  
     '__getattr__',  
     '__getnewargs__',  
     '__gt__',  
     '__hash__',  
     '__index__',  
     '__init__',  
     '__init_subclass__',  
     '__int__',  
     '__invert__',  
     '__le__',  
     '__lshift__',  
     '__lt__',  
     '__mod__',  
     '__mul__',  
     '__ne__',  
     '__neg__',  
     '__new__',  
     '__or__',  
     '__pos__',  
     '__pow__',  
     '__radd__',  
     '__rand__',  
     '__rdivmod__',  
     '__reduce__',  
     '__reduce_ex__',  
     '__repr__',
```

(continues on next page)

(continued from previous page)

```

'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__trunc__',
'__xor__',
'bit_length',
'conjugate',
'denominator',
'from_bytes',
'imag',
'numerator',
'real',
'to_bytes']

```

All those methods with `__` on both sides are methods that are not normally shown in the tab completion list for the object. They are often called “dunder” methods for brevity, so you would say “dunder abs” for `__abs__`. In the documentation for Python these are called “special methods”.

Special methods are how some fundamental properties of objects are implemented in Python. For instance, you can safely imagine that `a + b` is translated to

```

[4]: a.__add__(b)
[4]: 5

```

This is the mechanism by which different kinds of objects can do very different kinds of things when `+` is used on them:

```

[5]: a = '2'
     b = '3'

```

```

[6]: a + b
[6]: '23'

```

```

[7]: a.__add__(b)
[7]: '23'

```

We’ve already encountered one special method: `__init__`. Let’s build a class which stores a value internally.

```

[8]: class TestClass:
     def __init__(self, value):
         self.value = value

```


We can now create objects of class `TestClass`:

```
[9]: a = TestClass(2)
     b = TestClass(3)
```

But they are a bit hard to use. For one, they don't display anything meaningful when we display them

```
[10]: a
[10]: <__main__.TestClass at 0x10750fcc0>
```

We can extend `TestClass` with a `__repr__` method. This is short for representation and is used in the console and the notebook to show an object. By convention, the `__repr__` method returns a string that could be copy-pasted to create the object.

```
[11]: class TestClass:
     def __init__(self, value):
         self.value = value
     def __repr__(self):
         return "TestClass({})".format(self.value)
```

```
[12]: a = TestClass(2)
     b = TestClass(3)
```

```
[13]: a
[13]: TestClass(2)
```

Great, at least we can see what we're working with now. But let's say we want to make this class be able to support addition. At the moment, this doesn't work:

```
[14]: # a + b
     # -----
     # TypeError                                Traceback (most recent call last)
     # <ipython-input-14-f96fb8f649b6> in <module>()
     # ----> 1 a + b
     #
     # TypeError: unsupported operand type(s) for +: 'TestClass' and 'TestClass'
```

```
[15]: class TestClass:
     def __init__(self, value):
         self.value = value

     def __repr__(self):
         return "TestClass({})".format(self.value)

     def __add__(self, other):
         return TestClass(self.value + other.value)
```

```
[16]: a = TestClass(2)
     b = TestClass(3)
```

```
[17]: a + b
[17]: TestClass(5)
```

This should give you a glimpse into how libraries like SymPy or Numpy obtain their effects. They are using these mechanisms to make objects which “do the right thing” when they are added together, divided and multiplied, as well as

giving them “normal” non-underscore methods for additional manipulation. All of the objects you have used to do math so far have implemented these operations. Think of `sympy.Symbol` or the `numpy.array`.

```
[ ]:
```

4.7 Objects

Have a look at the *Hybrid system simulation* notebook before reading this one.

In the `for` loop we used to implement the Euler integration in that notebook, it was clear that we were working with two systems - the tank system and the controller. We can see some similarities between the two systems: both of them have an input (x for the tank, e for the controller) and an output (h for the tank, x for the controller), both of them have an internal state (V and the integral of the error). However, we ended up having to repeat many calculations for the two of them, and the code corresponding to the two systems ended up in different places between the two of them.

Objects allow us to group each system’s equations and data together so that our loop can be cleaner and the parts of the code that are specific to each system can be all in one place.

```
[1]: import numpy
import scipy.signal
import matplotlib.pyplot as plt
%matplotlib inline
```

4.7.1 Tank system

We start by defining a `class` for the tank system:

```
[2]: class TankSystem:
    def __init__(self, A, alpha, K, V, Fi):
        """ This special function gets called when an object of this class is created """
        self.A = A
        self.alpha = alpha
        self.K = K
        self.Fi = Fi
        self.change_state(V)

    def f(self, x):
        return self.alpha**(x - 1)

    def change_input(self, x):
        self.Fo = self.K*self.f(x)*numpy.sqrt(self.h)

    def change_state(self, V):
        self.state = self.V = V
        self.output = self.h = self.V/self.A

    def derivative(self, x):
        self.change_input(x)
        dVdt = self.Fi - self.Fo
        return dVdt
```

The functions defined inside the class are known as methods and are called as `object.method(arguments)`. You may imagine that `object.method(arguments)` gets translated to `method(object, arguments)` before it is called, so the `self` argument to methods will be the object whose method is called.

4.7.2 PI Controller

And another for the PI controller.

```
[3]: class PIController:
    def __init__(self, Kc, tau_i, bias):
        self.Kc = Kc
        self.G = scipy.signal.lti([Kc*tau_i, Kc], [tau_i, 0])
        self.Gss = self.G.to_ss()
        self.change_state(numpy.zeros((self.Gss.A.shape[0], 1)))
        self.bias = self.output = bias
        self.y = self.bias

    def change_input(self, u):
        self.y = self.Gss.C.dot(self.x) + self.Gss.D.dot(u) + self.bias
        self.output = self.y[0, 0] # because y is a matrix, and we want a scalar.
        ↪ output

    def change_state(self, x):
        self.x = self.state = x

    def derivative(self, e):
        return self.Gss.A.dot(self.x) + self.Gss.B.dot(e)
```

```
[4]: ts = numpy.linspace(0, 100, 1000)
    dt = ts[1]
```

```
[5]: sp = 1.3
```

4.7.3 Generic integration

Now we can integrate. Notice that in the code below there is no specific reference to tanks or PI controllers. To change the type of controller or the system, we can change the kinds of objects we create as `system` and `controller`

```
[6]: def control_simulation(system, controller):
    outputs = []
    for t in ts:
        system.change_input(controller.output)

        e = sp - system.output

        controller.change_input(e)

        system.change_state(system.state + system.derivative(controller.output)*dt)
        controller.change_state(controller.state + controller.derivative(e)*dt)

        outputs.append(system.output)
    return outputs
```

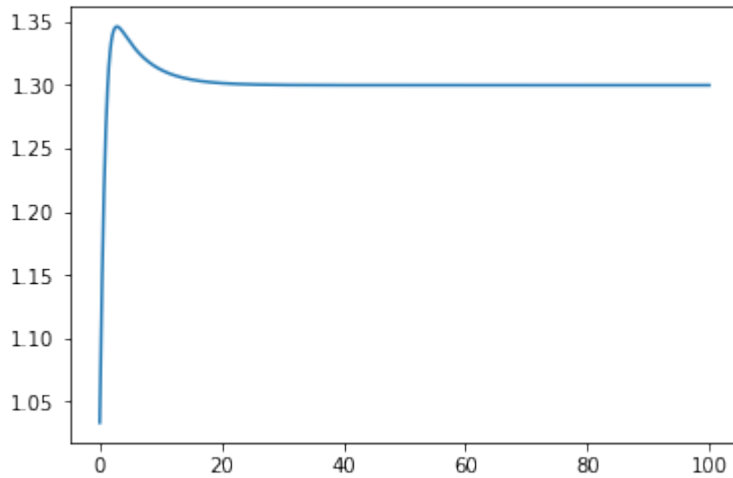
```
[7]: system = TankSystem(A=2, alpha=20, K=2, V=2, Fi=1)
    controller = PIController(Kc=-1, tau_i=5, bias=0.7)
```

```
[8]: outputs = control_simulation(system, controller)
```



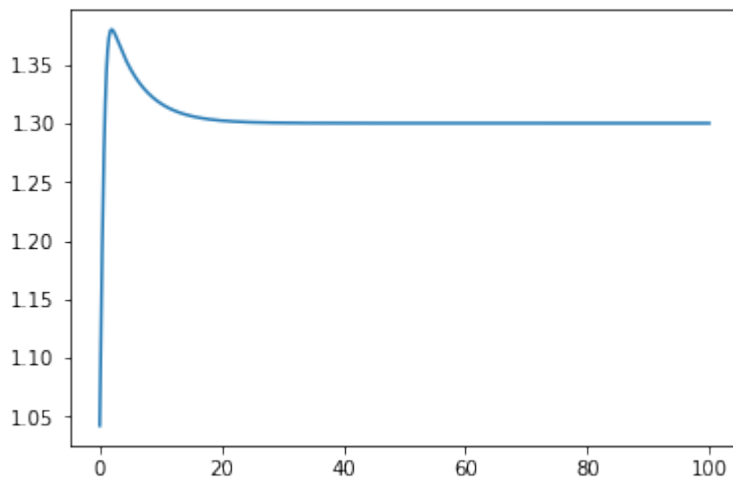
```
[9]: plt.plot(ts, outputs)
```

```
[9]: [<matplotlib.lines.Line2D at 0x111796550>]
```



This allows us to simulate different situations easily.

```
[10]: outputs = control_simulation(system=TankSystem(A=2, alpha=10, K=2, V=2, Fi=1),
                                   controller=PIController(Kc=-2, tau_i=5, bias=0.5))
plt.plot(ts, outputs);
```



4.7.4 Re-using the interface

Note that we could also define a completely new system. The controller doesn't "know" it is a level controller in this case, so let's build a class which would represent an LTI system under control:

```
[11]: class LtiSystem:
        def __init__(self, numerator, denominator):
            self.G = scipy.signal.lti(numerator, denominator)
            self.Gss = self.G.to_ss()
            self.change_state(numpy.zeros((self.Gss.A.shape[0], 1)))
```

(continues on next page)

(continued from previous page)

```

        self.y = self.output = 0

    def change_input(self, u):
        self.y = self.Gss.C.dot(self.x) + self.Gss.D.dot(u)
        self.output = self.y[0, 0]

    def change_state(self, x):
        self.x = self.state = x

    def derivative(self, e):
        return self.Gss.A.dot(self.x) + self.Gss.B.dot(e)

```

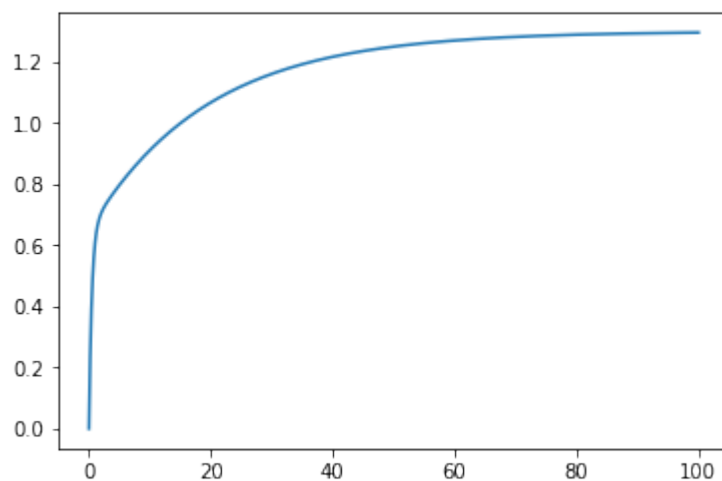
Now we can simulate the response of an arbitrary LTI system to an arbitrary controller as long as we have classes to represent their behaviour!

```

[12]: outputs = control_simulation(system=LtiSystem(1, [1, 1]),
                                controller=PIController(Kc=1, tau_i=10, bias=0))
plt.plot(ts, outputs)

[12]: [<matplotlib.lines.Line2D at 0x111957e10>]

```



Look how easy it becomes to compare control performance:

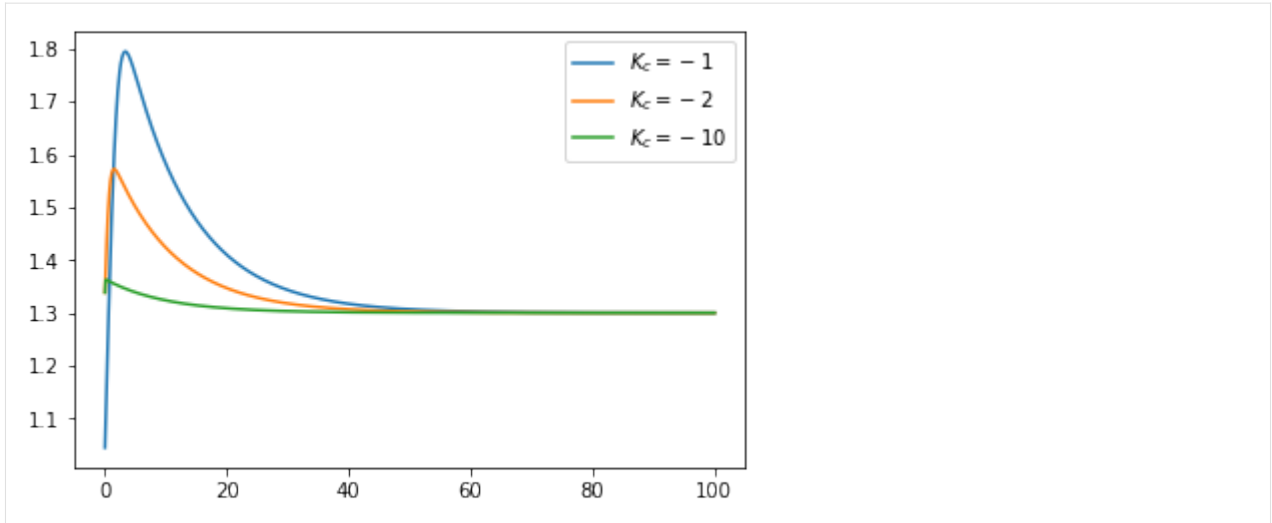
```

[13]: system = TankSystem(A=2, alpha=10, K=2, V=2, Fi=1)

        controllers = [PIController(Kc=K, tau_i=10, bias=0) for K in [-1, -2, -10]]
        for controller in controllers:
            outputs = control_simulation(system, controller)
            plt.plot(ts, outputs, label=r'$K_c={}$'.format(controller.Kc))
        plt.legend()

[13]: <matplotlib.legend.Legend at 0x1119a25c0>

```

4.8 A discrete controller class

Let's extend the example from our previous object-oriented simulation notebook to discrete controllers.

```
[1]: import numpy
import scipy.signal
import matplotlib.pyplot as plt
%matplotlib inline
```

We start by defining a class for the tank system again:

```
[2]: class TankSystem:
    def __init__(self, A, alpha, K, V, Fi):
        """ This special function gets called when an object of this class is created"
        ↪ """
        self.A = A
        self.alpha = alpha
        self.K = K
        self.Fi = Fi
        self.change_state(V)

    def f(self, x):
        return self.alpha**(x - 1)

    def change_input(self, x):
        self.Fo = self.K*self.f(x)*numpy.sqrt(self.h)

    def change_state(self, V):
        self.state = self.V = V
        self.output = self.h = self.V/self.A

    def derivative(self, x):
        self.change_input(x)
        dVdt = self.Fi - self.Fo
        return dVdt
```

But this time we'll do a discrete controller:


```
[3]: class DiscreteController:
      def __init__(self, DeltaT, Kc, tau_i, bias):
          self.DeltaT = DeltaT
          self.Kc = Kc
          self.tau_i = tau_i
          self.next_sample_time = 0
          self.error_sum = 0
          self.bias = self.output = bias

      def update(self, u, t):
          if t >= self.next_sample_time:
              self.error_sum += u
              self.output = self.Kc*(u + 1/self.tau_i*self.error_sum)
              self.next_sample_time += self.DeltaT
```

I've changed the way we update the controller here to accomodate the fact that discrete controllers don't have states to integrate, so we don't need the derivative method. The way I've built this class requires that the update method be called with strictly ascending times in t . Note that this class corresponds very closely to a physical box. All the sampling is taking place inside the class and it is designed to be updated in a "normal" Euler integration loop. See how using classes makes the calculations which are about a particular physical object (the controller) stay together in the code rather than being spread out in lots of places?

Of course, we still have to test that this works:

```
[4]: ts = numpy.linspace(0, 100, 1000)
      dt = ts[1]
```

```
[5]: sp = 1.3
```

I've changed this function to use the new update methods, but otherwise it is similar to the previous one.

```
[6]: def control_simulation_discrete(system, controller):
      outputs = []
      for t in ts:
          system.change_input(controller.output)

          e = sp - system.output

          controller.update(e, t)

          system.change_state(system.state + system.derivative(controller.output)*dt)

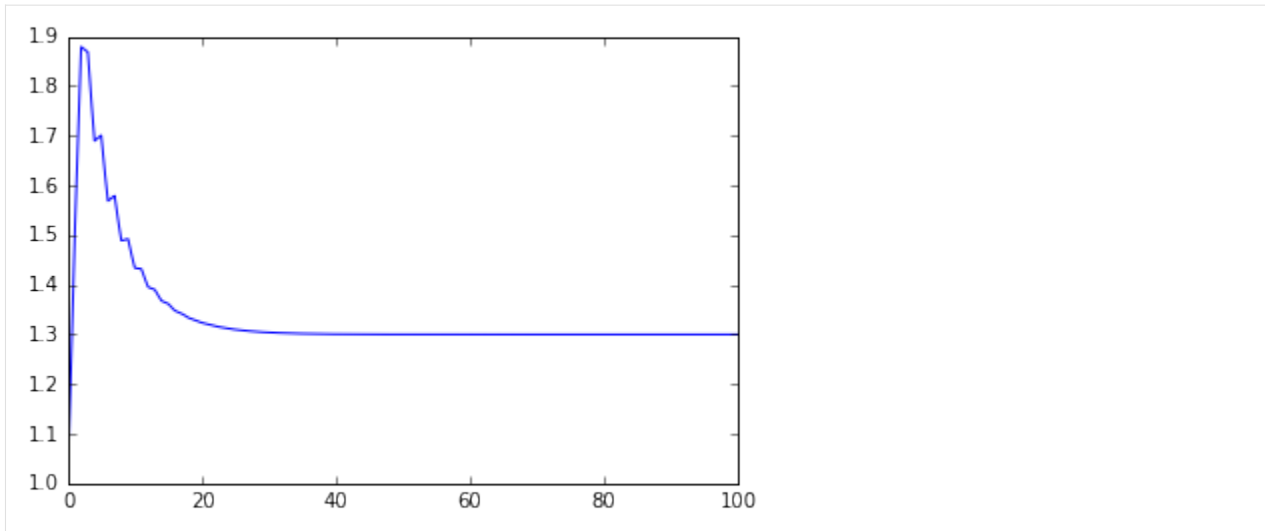
          outputs.append(system.output)
      return outputs
```

```
[7]: system = TankSystem(A=2, alpha=20, K=2, V=2, Fi=1)
      controller = DiscreteController(DeltaT=1, Kc=-1, tau_i=5, bias=0.7)
```

```
[8]: outputs = control_simulation_discrete(system, controller)
```

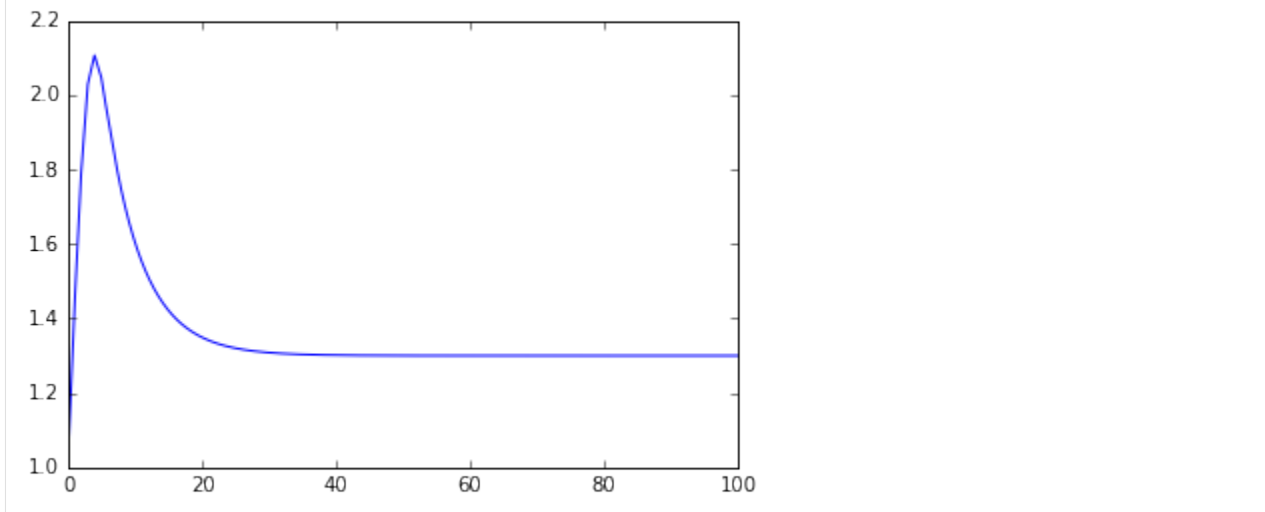
```
[9]: plt.plot(ts, outputs)
```

```
[9]: [<matplotlib.lines.Line2D at 0x10cf877f0>]
```

We can still simulate different situations easily.

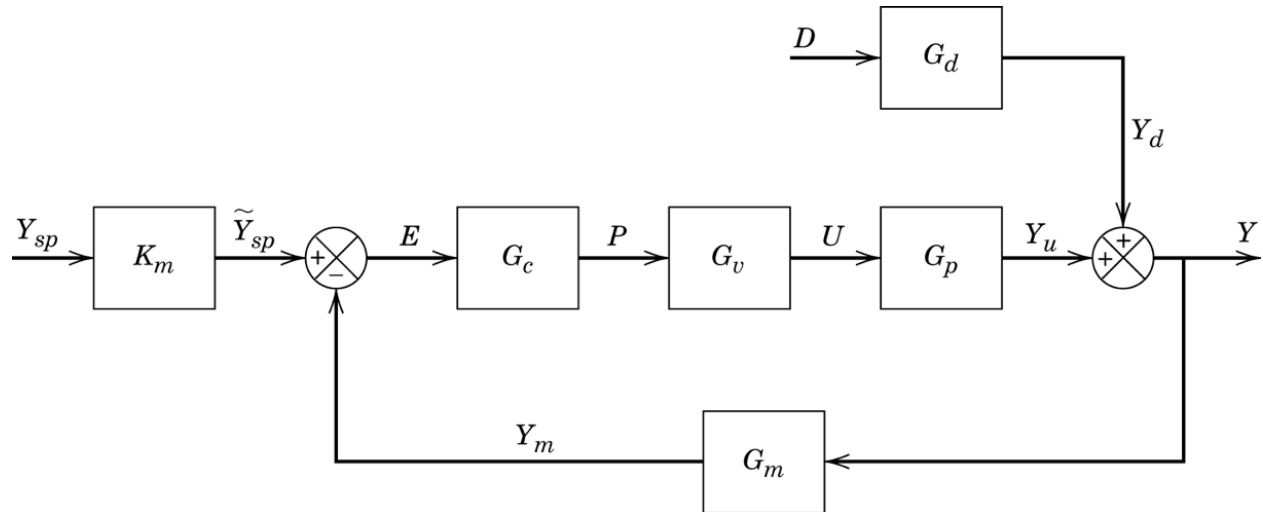
```
[10]: outputs = control_simulation_discrete(system=TankSystem(A=2, alpha=10, K=2, V=2,
↪ Fi=1),
                                controller=DiscreteController(DeltaT=1, Kc=-0.5,
↪ tau_i=5, bias=0.5))
plt.plot(ts, outputs);
```



```
[ ]:
```


4.9 Blocksim

`tbcontrol.blocksim` is a simple library for simulating the kinds of block diagrams you would encounter in a typical undergraduate control textbook. Let's start with the most basic example of feedback control.



```
[1]: import tbcontrol
     tbcontrol.expectversion("0.1.1")
```

```
[2]: from tbcontrol import blocksim
```

Our first job is to define objects representing each of the blocks. A common one is the LTI block

```
[3]: Gp = blocksim.LTI('Gp', 'u', 'y', 10, [100, 1], 50)
```

```
[4]: Gp
```

```
[4]: LTI: u → [ Gp ] → y
```

We'll use a PI controller

```
[5]: Gc = blocksim.PI('Gc', 'e', 'u', 0.1, 50)
```

```
[6]: Gc
```

```
[6]: PI: e → [ Gc ] → u
```

Once we have the blocks, we can create a Diagram.

Sums are specified as a dictionary with the keys being the output signal and the values being a tuple containing the input signals. The leading + is compulsory.

The inputs come next and are specified as functions of time. `Blocksim.step()` can be used to build a step function.

```
[7]: diagram = blocksim.Diagram([Gp, Gc],
                                sums={'e': ('+y_sp', '-y')},
                                inputs={'y_sp': blocksim.step()})
```

```
[8]: diagram
```



```
[8]: LTI: u → [ Gp ] → y
    PI: e → [ Gc ] → u
```

Blocksim is primarily focused on being able to simulate a diagram. The next step is to create a time vector and do the simulation.

```
[9]: import numpy
```

The time vector also specifies the step size for integration. Since blocksim uses Euler integration internally you should choose a time step which is at least 10 times smaller than the smallest time constant of all the blocks. The timespan is of course dependent on what you are investigating.

```
[10]: ts = numpy.arange(start=0, stop=1000, step=1)
```

```
[11]: simulation_results = diagram.simulate(ts, progress=True)
      HBox(children=(IntProgress(value=0, max=1000), HTML(value='')))
```

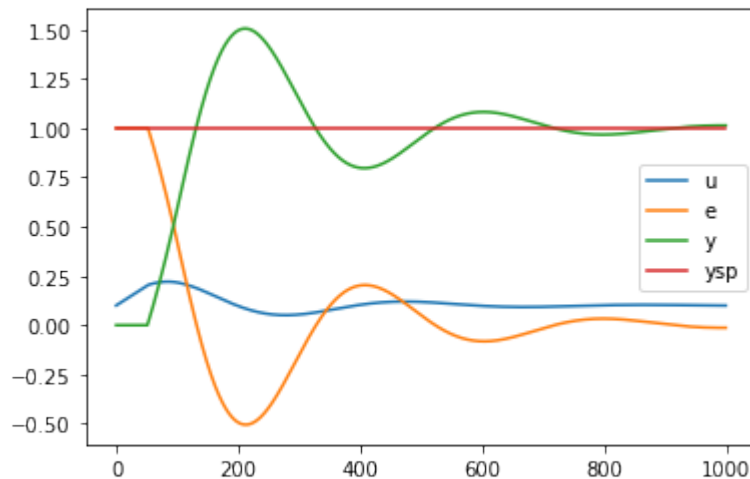
The result of `simulate()` is a dictionary containing the simulation results.

```
[12]: import matplotlib.pyplot as plt
```

```
[13]: %matplotlib inline
```

```
[14]: for signal, value in simulation_results.items():
      plt.plot(ts, value, label=signal)
      plt.legend()
```

```
[14]: <matplotlib.legend.Legend at 0x1c19f872b0>
```



4.9.1 Re-using parts of a diagram

Let's compare the output of a PI and a PID controller on this system. We've already got the PI response, which we should store.

```
[15]: y_pi = simulation_results['y']
```

Let's swap out the PI controller for a PID.

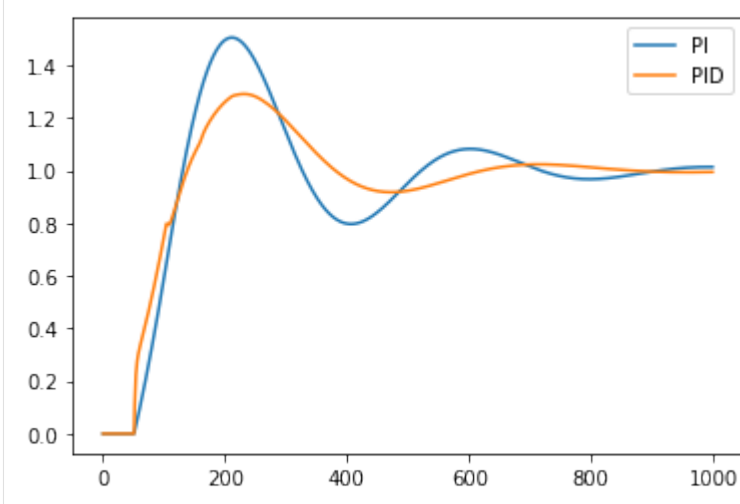
```
[16]: Gc_pid = blocksim.PID('Gc', 'e', 'u', 0.1, 50, 25)
```

```
[17]: diagram.blocks = [Gp, Gc_pid]
```

```
[18]: simulation_results = diagram.simulate(ts, progress=True)
      HBox(children=(IntProgress(value=0, max=1000), HTML(value='')))
```

```
[19]: plt.plot(ts, y_pi, label='PI')
      plt.plot(ts, simulation_results['y'], label='PID')
      plt.legend()
```

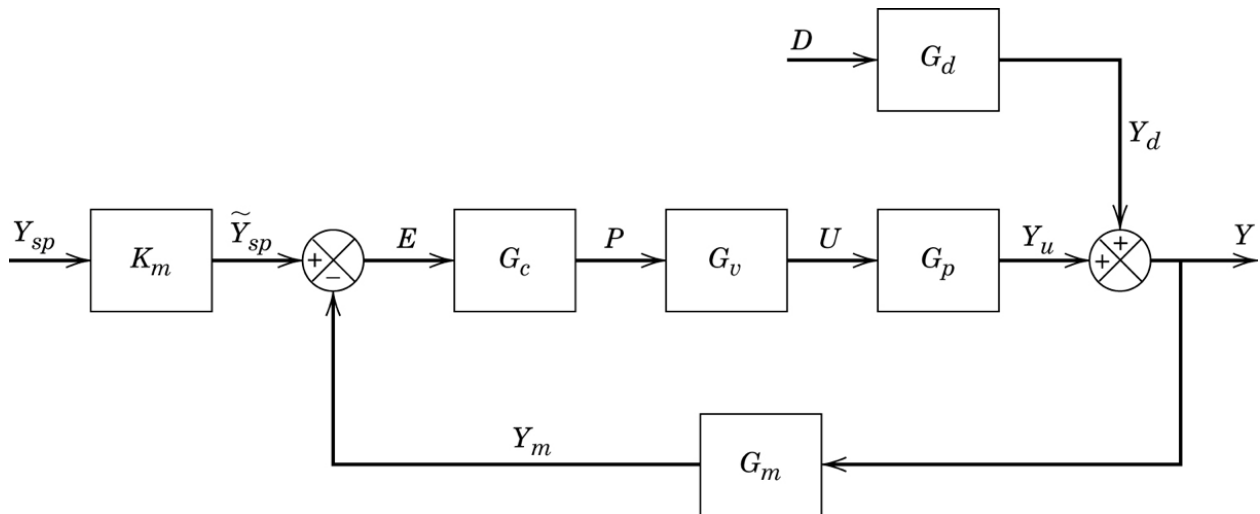
```
[19]: <matplotlib.legend.Legend at 0x1c1a113d68>
```



We can see that adding the derivative action has improved control.

4.10 Disturbances

We can simulate a more complicated block diagram with a disturbance.



```
[20]: Km = blocksim.LTI('Km', 'ysp', 'ytildesp', 1, 1)
      Gc = blocksim.PI('Gp', 'e', 'p', Kc=8, tau_i=10)
      Gv = blocksim.LTI('Gv', 'p', 'u', 1, 1)
      Gp = blocksim.LTI('Gp', 'u', 'yu', [1], [10, 1])
      Gd = blocksim.LTI('Gd', 'd', 'yd', [1], [10, 1])
      Gm = blocksim.LTI('Gm', 'y', 'ym', [1], [1, 1])

      blocks = [Km, Gc, Gv, Gp, Gd, Gm]
```

```
[21]: sums = {'e': ('+ytildesp', '-ym'),
              'y': ('+yd', '+yu')}

      inputs = {'ysp': blocksim.step(),
                'd': blocksim.step(starttime=50)}
```

```
[22]: diagram = blocksim.Diagram(blocks, sums, inputs)
```

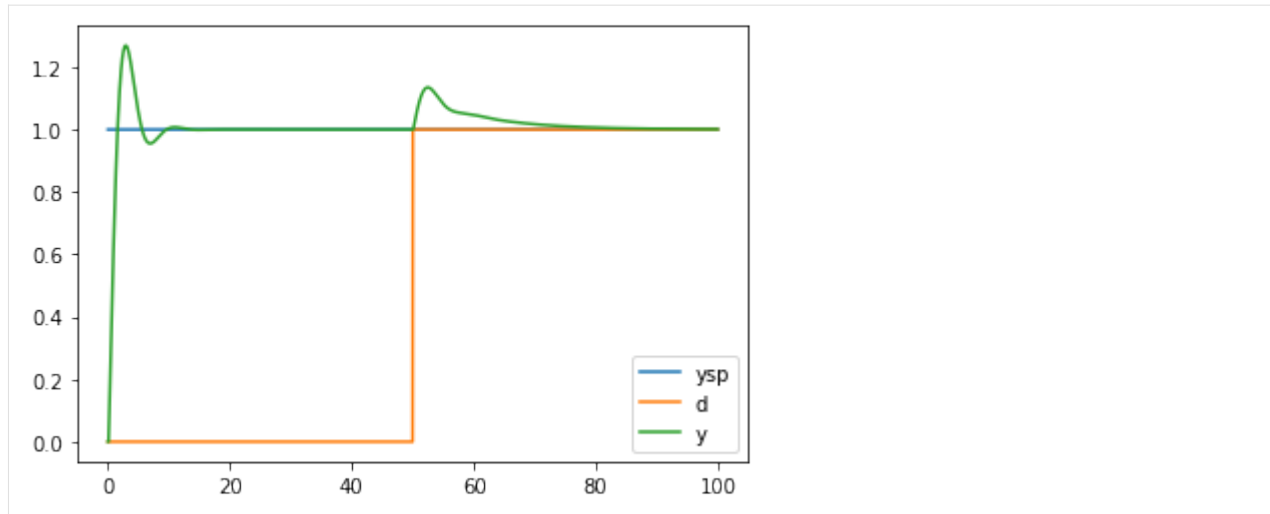
```
[23]: ts = numpy.arange(start=0, stop=100, step=0.05)
```

```
[24]: results = diagram.simulate(ts, progress=True)

      HBox(children=(IntProgress(value=0, max=2000), HTML(value='')))
```

```
[25]: for name in ('ysp', 'd', 'y'):
      plt.plot(ts, results[name], label=name)
      plt.legend()
```

```
[25]: <matplotlib.legend.Legend at 0x1c19feccf8>
```

4.11 Algebraic equations

Sometimes it is useful to be able to handle non-linear calculations in block diagrams. This deviates from the strict interpretation of block diagrams but can be useful for instance in calculating the response of a controller with output limits.

```
[26]: import numpy

[27]: def limit(t, u):
      return numpy.clip(u, 0, 0.2)

[28]: Gp = blocksim.LTI('Gp', 'unlimited', 'y', 10, [100, 1], 50)
      Gc = blocksim.PI('Gc', 'e', 'u', 0.1, 50)

[29]: limiter = blocksim.AlgebraicEquation('Limiter', 'u', 'unlimited', limit)

[30]: diagram = blocksim.Diagram([Gp, Gc, limiter],
                                sums={'e': ('+ysp', '-y')},
                                inputs={'ysp': blocksim.step()})

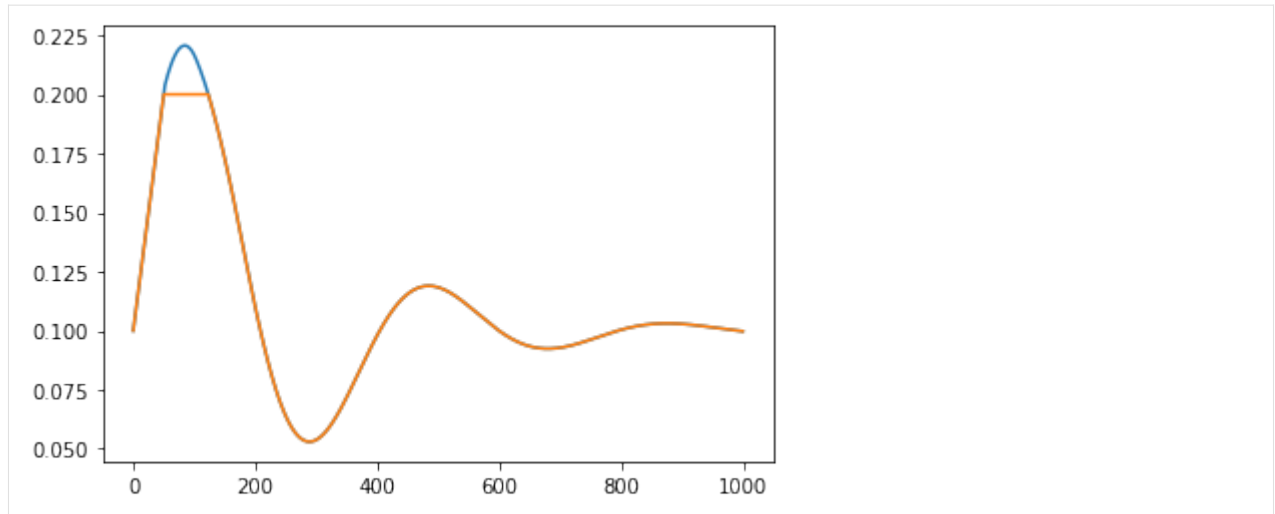
[31]: ts = numpy.arange(start=0, stop=1000, step=1)

[32]: simulation_results = diagram.simulate(ts)

[33]: diagram
LTI: unlimited → [ Gp ] → y
PI: e → [ Gc ] → u
AlgebraicEquation: u → [ Limiter ] → unlimited

[34]: plt.plot(ts, simulation_results['u'])
      plt.plot(ts, simulation_results['unlimited'])

[34]: [<matplotlib.lines.Line2D at 0x1c1a16ad68>]
```

We can see the effect of the limiter clearly in the above figure

[]:

TEMPERATURE CONTROL LAB (TCLAB)

5.1 FOPDT fit

My tests of my TCLab around $Q_1 = 50$ have resulted FOPDT model with $\tau_p = 150$ s, $K_p = 0.38$ and $\theta = 15$ s

```
[61]: from tclab import runexperiment
```

```
[39]: steptime=1000
      Qbar = 50
      deltaQ = 10
```

```
[40]: def steptest(t, lab):
      lab.Q1(Qbar if t < steptime else Qbar + deltaQ)
```

```
[78]: %matplotlib notebook
```

```
[79]: experiment = runexperiment(steptest, connected=True,
                                plot=True, twindow=1000,
                                time=1000,
                                speedup=1,
                                dbfile='sinetest.db')
```

TCLab version 0.4.6dev

NHduino connected on port /dev/cu.wchusbserial1410 at 115200 baud.

TCLab Firmware 1.3.0 Arduino Uno.

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

TCLab disconnected successfully.

```
[16]: import numpy
      from matplotlib import pyplot as plt
```

```
[17]: %matplotlib inline
```

```
[21]: # h = experiment.historian
```

```
[19]: from tclab import Historian
```



```
[20]: h = Historian(sources=({'Q1', lambda: [1, 2, 3, 4]}, ('Q2', None), ('T1', None), ('T2', None)), dbfile='sinetest.db')
```

```
[22]: h.get_sessions()
```

```
[22]: [(2, '2018-03-06 18:45:27', 13710),
      (12, '2018-03-07 14:55:43', 2001),
      (15, '2018-03-07 18:43:39', 7526),
      (25, '2018-03-08 05:34:09', 5523),
      (27, '2018-03-08 07:10:23', 4873),
      (28, '2018-03-08 12:59:29', 55),
      (29, '2018-03-08 13:00:31', 116),
      (30, '2018-03-08 13:02:35', 1001),
      (31, '2018-03-08 13:25:17', 2001),
      (32, '2018-03-08 14:30:19', 0),
      (33, '2018-03-08 14:30:32', 891),
      (34, '2018-03-08 14:46:08', 536),
      (35, '2018-03-08 14:55:18', 132),
      (36, '2018-03-08 15:02:28', 2001),
      (37, '2018-03-09 04:37:56', 0),
      (38, '2018-03-09 04:39:17', 0)]
```

```
[30]: h.load_session(12)
```

```
[62]: tau_p = 150
      K_p = 0.38
      theta = 15
```

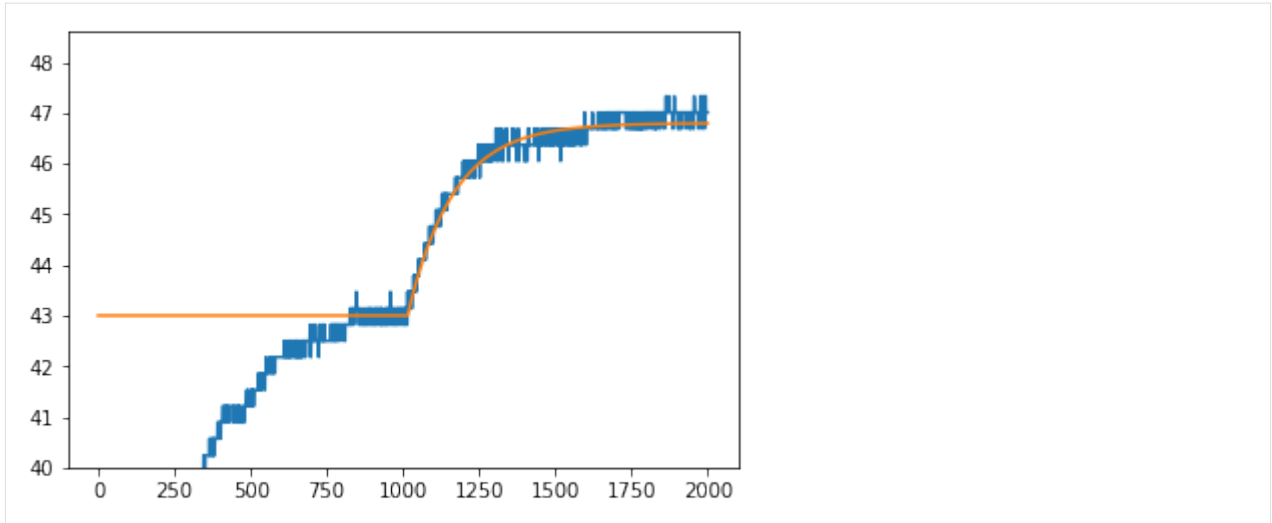
```
[63]: T1_0 = 43
```

```
[64]: t = numpy.array(h.t)
```

```
[65]: resp = numpy.maximum(deltaQ*K_p*(1 - numpy.exp(-(t - theta - steptime)/tau_p)), 0) + T1_0
```

```
[66]: plt.figure()
      plt.plot(h.t, h.logdict['T1'])
      plt.plot(h.t, resp)
      plt.ylim(ymin=40)
```

```
[66]: (40, 48.603)
```

[]:

5.2 TCLab in the frequency domain

```
[1]: import numpy
```

```
[2]: %matplotlib inline
```

```
[3]: tau_p = 150
      K_p = 0.33
      theta = 15
```

```
[4]: omega = numpy.logspace(-3, -2)
      s = omega*1j
```

```
[5]: G = K_p/(tau_p*s + 1)*numpy.exp(-theta*s)
```

```
[6]: import matplotlib.pyplot as plt
```

Let's choose 5 logarithmically spaced points around the corner frequency

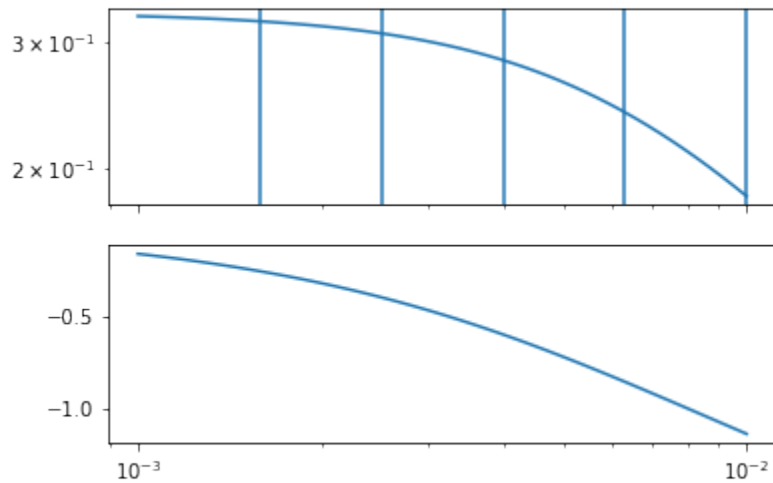
```
[7]: freqs = numpy.logspace(-2.8, -2, 5)
```

```
[8]: def plotfreqs(ax):
      for freq in freqs:
          ax.axvline(freq)
```

```
[9]: def bode(omega, G, gainax=None, phaseax=None, phasecorr=0):
      if gainax is None:
          fig, (gainax, phaseax) = plt.subplots(2, 1, sharex=True)
          gainax.loglog(omega, numpy.abs(G))
          angle = numpy.angle(G)
          phaseax.semilogx(omega, numpy.unwrap(angle) + phasecorr)
      return gainax, phaseax
```



```
[10]: gainax, phaseax = bode(omega, G)
      plotfregs(gainax)
```



5.2.1 Direct frequency domain tests

```
[11]: Qbar = 50
      deltaQ = A = 10
```

How long does one sine wave take to repeat?

$$P = 2\pi/\omega$$

```
[12]: P = 2*numpy.pi/freqs
```

We'll do a couple of repeats.

```
[13]: nperiods = 3
```

```
[14]: switch_times = numpy.concatenate([[0], numpy.cumsum(P*nperiods)])
```

```
[15]: switch_times
```

```
[15]: array([ 0.          , 11893.26574888, 19397.409123   , 24132.20349893,
          27119.65678502, 29004.61237718])
```

```
[16]: from tclab import runexperiment, labtime
```

```
[17]: def send_sine_wave(t, lab):
      print(f'\rTime: {t} Last sleep: {labtime.lastsleep} ', end='')
      step = numpy.max(numpy.nonzero(switch_times <= t))
      lab.Q1(Qbar + A*numpy.sin(freqs[step]*(t - switch_times[step])))
```

```
[18]: totaltime = sum(nperiods*P)
      totaltime
```



```
[18]: 29004.612377178186
```

Experiment will take this many hours

```
[19]: totaltime/60/60
```

```
[19]: 8.056836771438386
```

Looks like we'll be running it overnight.

```
[57]: %%time
experiment = runexperiment(send_sine_wave, connected=True,
                           plot=False, twindow=1000,
                           time=int(totaltime),
                           speedup=1,
                           dbfile='sinetest.db')
```

```
TCLab version 0.4.6dev
NHduino connected on port /dev/cu.wchusbserial1410 at 115200 baud.
TCLab Firmware 1.3.0 Arduino Uno.
Time: 4872.0 Last sleep: 0.8694100379943848 TCLab disconnected successfully.
CPU times: user 15.5 s, sys: 28 s, total: 43.6 s
Wall time: 1h 21min 16s
```

```
[37]: import tclab
```

```
[38]: h = tclab.Historian(('Q1', lambda: [0, 0, 0, 0]),
                        ('Q2', None),
                        ('T1', None),
                        ('T2', None)), dbfile='sinetest.db')
```

```
[39]: # h = experiment.historian
```

```
[40]: h.get_sessions()
```

```
[40]: [(2, '2018-03-06 18:45:27', 13710),
      (12, '2018-03-07 14:55:43', 2001),
      (15, '2018-03-07 18:43:39', 7526),
      (25, '2018-03-08 05:34:09', 5523),
      (27, '2018-03-08 07:10:23', 4873),
      (28, '2018-03-08 12:59:29', 55),
      (29, '2018-03-08 13:00:31', 116),
      (30, '2018-03-08 13:02:35', 1001),
      (31, '2018-03-08 13:25:17', 2001),
      (32, '2018-03-08 14:30:19', 0),
      (33, '2018-03-08 14:30:32', 891),
      (34, '2018-03-08 14:46:08', 536),
      (35, '2018-03-08 14:55:18', 132),
      (36, '2018-03-08 15:02:28', 2001),
      (37, '2018-03-09 04:37:56', 0),
      (38, '2018-03-09 04:39:17', 0),
      (39, '2018-03-09 04:45:49', 0),
      (40, '2018-03-09 04:46:44', 0),
      (41, '2018-03-09 08:22:48', 0),
      (42, '2018-03-09 08:23:20', 0)]
```



```
[41]: import pandas
```

```
[42]: %matplotlib inline
```

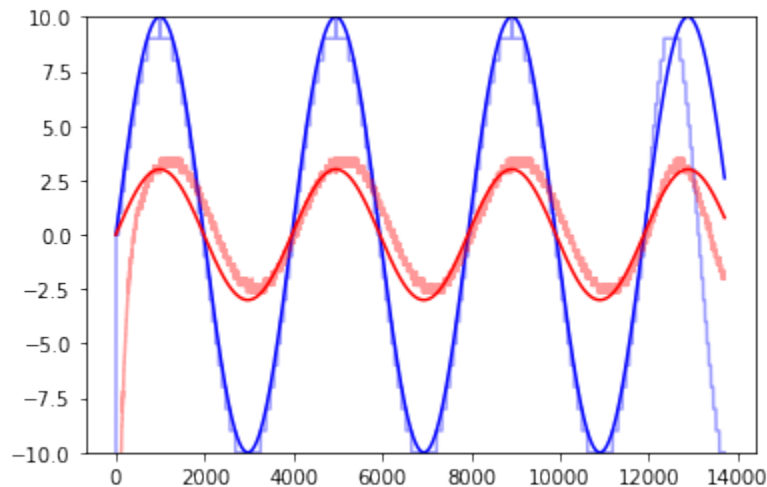
```
[43]: # h = experiment.historian
```

```
[44]: sine_sessions = [2, 15, 25, 27]
```

```
[45]: def sinefit(session, freq, Tbar=40, inphase=0, gain=0.4, phase=0):
    h.load_session(session)
    t = numpy.array(h.t)
    Q1 = numpy.array(h.logdict['Q1'])
    Q1_sine = A*numpy.sin(t*freq + inphase)
    T1 = numpy.array(h.logdict['T1'])
    T1_sine = A*gain*numpy.sin(t*freq + inphase + phase)
    plt.plot(t, Q1 - Qbar, color='blue', alpha=0.4)
    plt.plot(t, Q1_sine, color='blue')
    plt.plot(t, T1 - Tbar, color='red', alpha=0.4)
    plt.plot(t, T1_sine, color='red')
    plt.ylim(-A, A)
    print(f'Gain={gain}, Phase={phase-inphase}')
```

```
[46]: sinefit(2, freqs[0], 40, 0, 0.3, 0, )
```

Gain=0.3, Phase=0



```
[47]: from ipywidgets import interact
```

```
[48]: interact(sinefit,
    session=sine_sessions,
    freq=freqs,
    Tbar=(30., 50.),
    inphase=(0, 2*numpy.pi),
    gain=(0., 1., 0.01),
    phase=(-2*numpy.pi, 0),)
```

```
interactive(children=(Dropdown(description='session', options=(2, 15, 25, 27),
↵value=2), Dropdown(description='freq', options=(0.001584893192461114, 0.
```

(continues on next page)

(continued from previous page)

```

→002511886431509582, 0.003981071705534973, 0.00630957344480193, 0.01), value=0.
→001584893192461114), FloatSlider(value=40.0, description='Tbar', max=50.0, min=30.
→0), FloatSlider(value=0.0, description='inphase', max=6.283185307179586),
→FloatSlider(value=0.4, description='gain', max=1.0, step=0.01), FloatSlider(value=0.
→0, description='phase', max=0.0, min=-6.283185307179586), Output()), _dom_classes=(
→'widget-interact',))

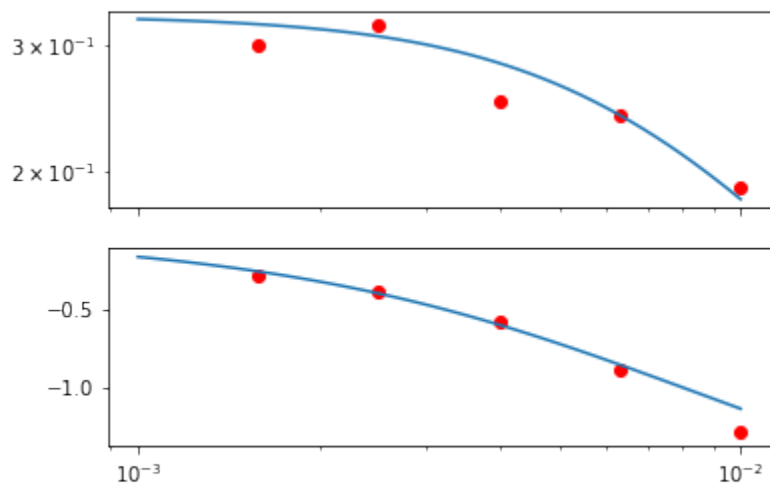
```

```
[48]: <function __main__.sinefit>
```

```
[124]: gains = [0.3, 0.32, 0.25, 0.24, 0.19]
      phases = [-0.28, -0.38, -0.58, -0.88, -1.28]
```

```
[158]: gainax, phaseax = bode(omega, G)
      gainax.scatter(freqs, gains, color='red')
      phaseax.scatter(freqs, phases, color='red')
```

```
[158]: <matplotlib.collections.PathCollection at 0x11a430ba8>
```



5.2.2 FFT based bode diagram

```
[20]: next_values = {'time': 0, 'value': +A}
      def random_noise(t, lab):
          if t > next_values['time']:
              next_values['time'] += numpy.random.uniform(50, 300)
              next_values['value'] = -A if next_values['value'] == +A else +A
              lab.Q1(Qbar + next_values['value'])

```

```
[21]: next_values
```

```
[21]: {'time': 0, 'value': 10}
```

```
[22]: %matplotlib notebook
```

```
[23]: %%time
      experiment = runexperiment(random_noise, connected=True,
                                plot=True, twindow=2000,
```

(continues on next page)

(continued from previous page)

```

        time=8*60*60,
        speedup=1,
        dbfile='sinetest.db',
    )

```

TCLab version 0.4.6dev

NHduino connected on port /dev/cu.wchusbserial1410 at 115200 baud.

TCLab Firmware 1.3.0 Arduino Uno.

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

TCLab disconnected successfully.

```

-----
RuntimeError                                Traceback (most recent call last)
<timed exec> in <module>()

~/Documents/Development/TCLab/tclab/experiment.py in runexperiment(function,
↳connected, plot, twindow, time, dbfile, speedup, synced)
    99     """
    100     with Experiment(connection, plot, twindow, time, dbfile, speedup, synced)
↳as experiment:
--> 101         for t in experiment.clock():
    102             function(t, experiment.lab)
    103         return experiment

~/Documents/Development/TCLab/tclab/experiment.py in clock(self)
    78         else:
    79             times = range(self.time)
--> 80             for t in times:
    81                 yield t
    82                 if self.plot:

~/Documents/Development/TCLab/tclab/labtime.py in clock(period, step, tol, adaptive)
    98             'Step size was {} s, but {:.2f} s elapsed '
    99             '({:.2f} too long). Consider increasing step.')
--> 100             raise RuntimeError(message.format(step, elapsed, elapsed-
↳step))
    101             labtime.sleep(step - (labtime.time() - start) % step)
    102             now = labtime.time() - start

RuntimeError: Labtime clock lost synchronization with real time. Step size was 1 s,
↳but 3.22 s elapsed (2.22 too long). Consider increasing step.

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```
[181]: h = experiment.historian
```

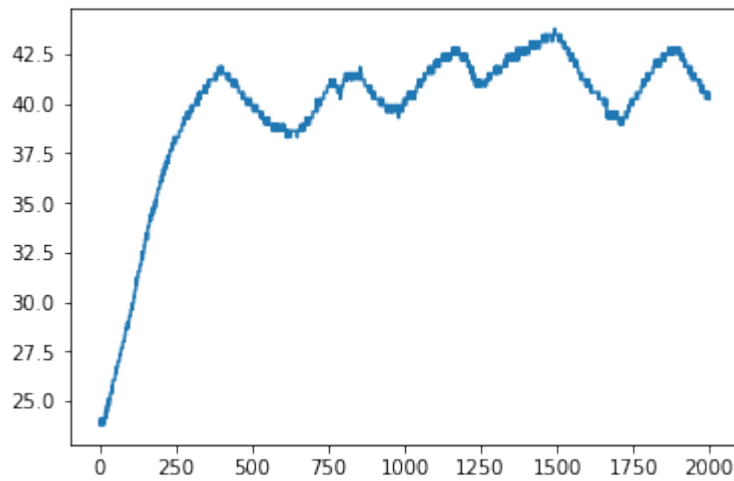
```
[30]: h.load_session(36)
```

```
[31]: %matplotlib inline
```

```
[32]: plt.plot(h.t, h.logdict['T1'])
```



```
[32]: [<matplotlib.lines.Line2D at 0x118ee7a20>]
```



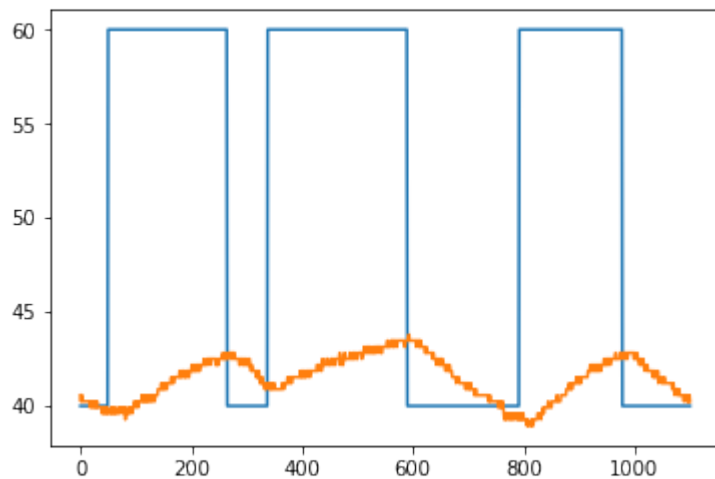
```
[33]: import numpy
```

```
[72]: startcut = 900
```

```
[73]: Q1 = numpy.array(h.logdict['Q1'][startcut:])
      T1 = numpy.array(h.logdict['T1'][startcut:])
```

```
[74]: plt.plot(Q1)
      plt.plot(T1)
```

```
[74]: [<matplotlib.lines.Line2D at 0x1176f1278>]
```



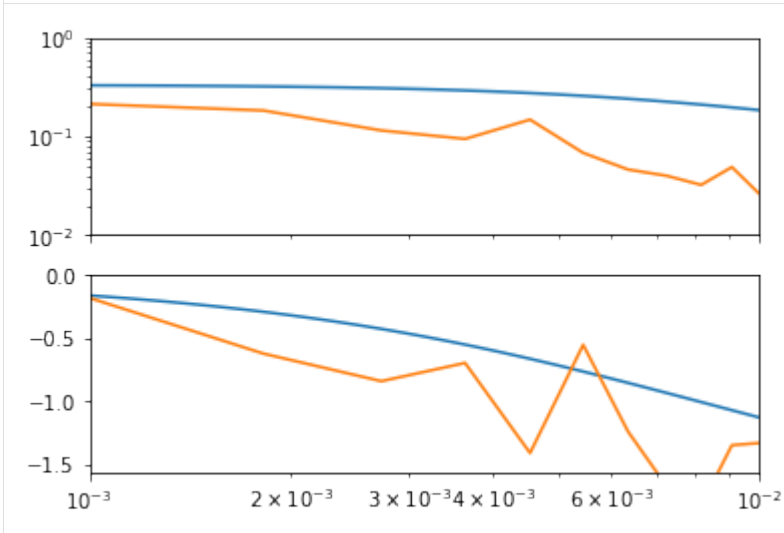
```
[75]: u = numpy.fft.rfft(Q1 - Qbar)
      y = numpy.fft.rfft(T1 - numpy.mean(T1))
```

```
[76]: omegafft = numpy.fft.rfftfreq(len(Q1), 1)
      Gfft = y/u
```



```
[160]: ga, pa = bode(omega, G)
bode(omegafft, Gfft, ga, pa, numpy.pi/4)
plt.xlim(1e-3, 1e-2)
ga.set_ylim(0.01, 1)
pa.set_ylim(-numpy.pi/2, 0)
```

```
[160]: (-1.5707963267948966, 0)
```



```
[ ]:
```

```
[ ]:
```


SEARCH PAGE

- search