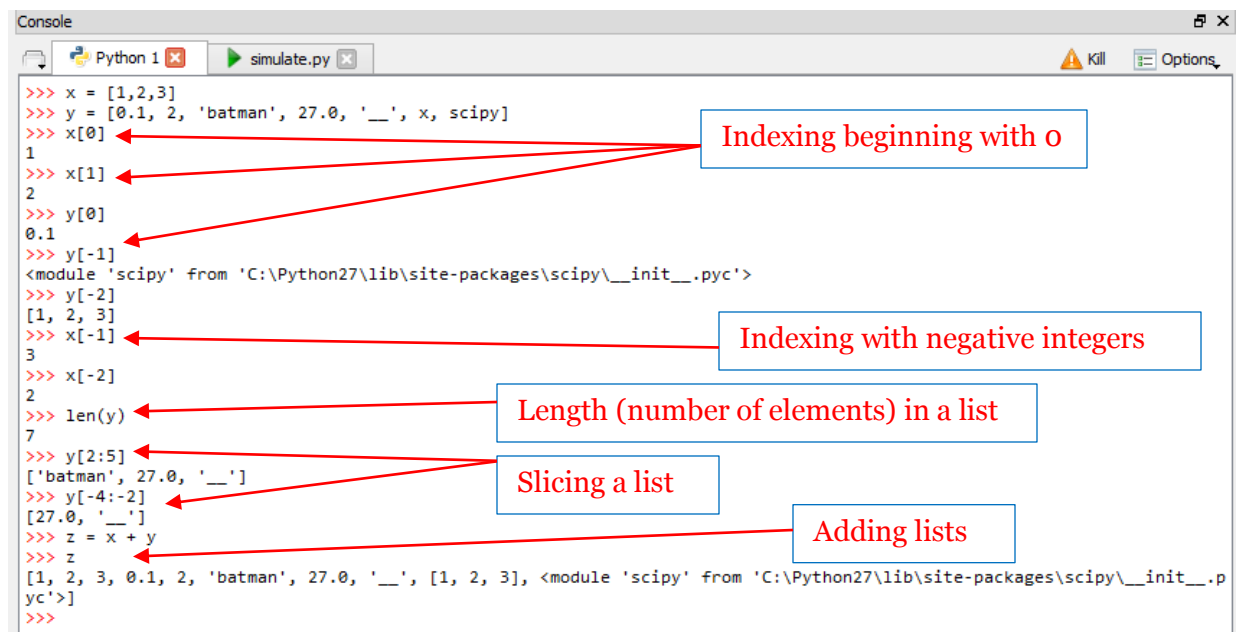# Fun with Numpy and Scipy

A gentle introduction (Dr. Vishwanath Dalvi)

The *Numpy* library (*import numpy* or *import scipy*) is a something worth familiarizing ourselves with. It is essentially the open-source community's answer to MATLAB: and a dashed fine answer it is too! All your linear algebra needs (and trust me, you have a *lot* of linear algebra needs!) met in one place. So lets begin...

But before we begin, let's look at a familiar Python data structure a little closer.

1. Indexing (and Pointers)

We are all already familiar with the *list* type of objects. Its just a list: but it is more than a simple list. It is a powerful *container* object. **Figure 1** illustrates the power of a list object. The list assigned to the variable y contains **pointers** to a wide variety of Python object. In fact, the pointers to *any* Python object can be contained in a list.

Figure 1: Illustration of the list *type. Two illustrative lists are shown: x, y. Elements of a list can be any Python object e.g. the list y contains elements that are integers, floats, text, another list (x) and a full Python module (scipy)! See how the elements of the list are accessed: in standard order with index starting at 0 and in reverse order with indexing starting at -1. Note also how the list is sliced and diced with sub-lists (we will come to this soon). A list is a powerful* **container** *object.* **Note***: the list only contains* pointers *to these objects and not the objects themselves: a subtle point that we will return to shortly.*

Before we move on, lets get this pointers thing out of the way. Python uses pointers by default. Consider Figure 2 which illustrates an implication of this. When we say $y = x$ what Python understands by it is: *assign to the variable y the pointer assigned to variable x.* Now both y and x refer *to the same object!* Which is why we got the surprise. This can be a source of great heartache and frustration later on. To preclude this, *never* say $y = x$ but always $y = x + [\,]$. This indicates to Python that y is a *new* object deserving its own pointer separation from x.

```
Console
   Python 1 ☒        ► simulate.py ☒
>>> a = 3
>>> b = a
>>> a, b                                As expected
(3, 3)
>>> b = 4
>>> a, b
(3, 4)
>>> x = [1, 2, 3]
>>> y = x
>>> x, y                                Surprise!!
([1, 2, 3], [1, 2, 3])
>>> y[1] = 4
>>> x, y
([1, 4, 3], [1, 4, 3])                  Make a New Object
>>> x = [1, 2, 3]
>>> y = x + []
>>> x, y
([1, 2, 3], [1, 2, 3])
>>> y[1] = 4
>>> x, y                                That's more like it!
([1, 2, 3], [1, 4, 3])
>>>
```

*Figure 2: Implications of the fact that Python uses pointers. Python variable names are actually pointers to a memory location. Hence the list x and y **point** to the **same memory location.** When we said x=y we merely assigned the **memory address** of x to y i.e. we are dealing with the **same** object. To make a separate, identical object use this trick: $y = x + [\,]$.*

Ok. Back to indexing.

We will need to refer to the Figure 3 shows a random list labelled x and how its elements can be accessed and how the list can be *sliced*. Pay careful attention to the illustration at the bottom of Figure 3 which shows how Python interpreters an *index*. The element that is referenced by an integer index is the one to the **immediate right** of the index.

```
>>> x = [1, 2, 3.0, 'four', 'five', 6, 7.0, 'eight', 'nine', 10]
>>>
>>>
>>>
>>> len(x)
10
>>> x[0], x[2], x[9]
(1, 3.0, 10)
>>> x[-10], x[-8], x[-1]
(1, 3.0, 10)
>>> x[2:7]
[3.0, 'four', 'five', 6, 7.0]
>>> x[-8:-3]
[3.0, 'four', 'five', 6, 7.0]
>>> x[:4]
[1, 2, 3.0, 'four']
>>> x[5:]
[6, 7.0, 'eight', 'nine', 10]
>>>
```

Figure 3: *Illustration of how python list-like/array-like objects are indexed. The illustration at the bottom of this figure shows that the indices are not **on** the elements of the list but rather to their left. Hence the element corresponding to a given index is the one to the **immediately to its right**. Hence x[2] returns 3.0 and x[-5] returns 6. We can also **slice** a list by specifying which indices to "cut" at: hence x[3,7] returns ['four', 'five', 6, 7.0]. This same slice can be obtained using x[-7, -3].*

Now let's see how to go about indexing an $N \times M$ array. Figure 4 shows a 4×3 array and various ways of extracting parts of it.

```
         Python 1 ☒        ▶ simulate.py ☒        ▶ errorestimation.py ☒                                    ⚠

>>> X = scipy.array([[0.1, 0.2, 0.3],[1.0, 2.0, 3.0],[10.0, 20.0, 30.0],[100.0,200.0,300.0]])
>>> X
array([[ 1.00000000e-01,   2.00000000e-01,   3.00000000e-01],
       [ 1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [ 1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [ 1.00000000e+02,   2.00000000e+02,   3.00000000e+02]])
>>> X.shape
(4, 3)
>>> X[3,2]
300.0
>>> X[1,2]
3.0
>>> X[:,2]
array([  0.3,    3. ,   30. ,  300. ])
>>> X[:,0]
array([  0.1,    1. ,   10. ,  100. ])
>>> X[0,:]
array([ 0.1,  0.2,  0.3])
>>> X[1,:]
array([ 1.,   2.,   3.])
>>> X[1:3,1]
array([  2.,   20.])
>>>
```

Single Column

Single Row

Extract Part of an Array

*Figure 4: Illustration showing indexing and slicing a 4 × 3 array. Note particularly how to extract a **single** row or column from the array. And then you can slice that row/column.*

So that's it. We are done with indexing.

2. Linear Algebra with Arrays

We have already seen the benefits of using arrays over lists in "A Gentle Introduction to Python". Go over that quickly and come back here. We will springboard from that. We really want to do linear algebra with arrays.

2.1 The Dot Product or Inner Product

Figure 5 shows the creation of a 3 × 4 array of floats from a list of lists. The "shape" of the array can be accessed from $x.shape$ attribute which returns a tuple $(nrow, ncol)$. The transpose of the array is simply $x.T$ whose shape is $(ncol, nrow)$. Since the arrays are not square, $x * x.T$ raises a **value error** saying it is impossible to **broadcast**: Python is unsure of how to go about this and raises an error. Broadcasting is something we will cover shortly. The takeaway here is that Python will only perform *unambiguous* tasks and raise an error for all others. There is no standard way of multiplying 3 × 4 array elementwise with a 4 × 3 array. However it is possible to do a *matrix multiplication of the two*. This is equivalent to taking a ***dot product*** of the two arrays (see Figure 5). For matrix multiplication $AB \neq BA$ i.e. it is **not commutative**. Hence we get different results for XX^T and for X^TX. You can (and should) verify that X^TX is what you would expect from standard matrix multiplication.

```
Console
  Python 1  ⊠       ▶ simulate.py ⊠
>>> x = [[0.1, 0.2, 0.3],[1.0,2.0,3.0],[10.0,20.0,30.0],[100.0,200.0,300.0]]
>>> x = scipy.array(x)
>>> x
array([[  1.00000000e-01,   2.00000000e-01,   3.00000000e-01],
       [  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [  1.00000000e+02,   2.00000000e+02,   3.00000000e+02]])
>>> print x.T
[[  1.00000000e-01   1.00000000e+00   1.00000000e+01   1.00000000e+02]
 [  2.00000000e-01   2.00000000e+00   2.00000000e+01   2.00000000e+02]
 [  3.00000000e-01   3.00000000e+00   3.00000000e+01   3.00000000e+02]]
>>> x.shape, x.T.shape
((4, 3), (3, 4))
>>> x*x.T
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,3) (3,4)
>>> scipy.dot(x.T, x)
array([[ 10101.01,  20202.02,  30303.03],
       [ 20202.02,  40404.04,  60606.06],
       [ 30303.03,  60606.06,  90909.09]])
>>> scipy.dot(x.T, x).shape
(3, 3)
>>> scipy.dot(x, x.T)
array([[  1.40000000e-01,   1.40000000e+00,   1.40000000e+01,
          1.40000000e+02],
       [  1.40000000e+00,   1.40000000e+01,   1.40000000e+02,
          1.40000000e+03],
       [  1.40000000e+01,   1.40000000e+02,   1.40000000e+03,
          1.40000000e+04],
       [  1.40000000e+02,   1.40000000e+03,   1.40000000e+04,
          1.40000000e+05]])
>>> scipy.dot(x, x.T).shape
(4, 4)
>>>
```

*Figure 5: Illustration of an array, its transpose and the dot product (**scipy.dot**)*

Another important operation in linear algebra is *vector* transformation (rotation and scaling) by multiplication by a matrix. See Figure 6 for an illustration.

```
>>> X = scipy.array([[0.1, 0.2, 0.3],[1.0, 2.0, 3.0],[10.0, 20.0, 30.0]])
>>> X
array([[  0.1,   0.2,   0.3],
       [  1. ,   2. ,   3. ],
       [ 10. ,  20. ,  30. ]])
>>> X.shape
(3, 3)
>>> y = scipy.array([10.0, 20.0, 30.0])
>>> y.shape
(3,)
>>> y = y.reshape((1,3))
>>> y
array([[ 10.,  20.,  30.]])
>>> y.shape
(1, 3)
>>> y = y.reshape((3,1))
>>> y
array([[ 10.],
       [ 20.],
       [ 30.]])
>>> y.shape
(3, 1)
>>> yT = y.T
>>> yT
array([[ 10.,  20.,  30.]])
>>> yT.shape
(1, 3)
>>> scipy.dot(yT, X)
array([[ 321.,  642.,  963.]])
>>> scipy.dot(X, y)
array([[   14.],
       [  140.],
       [ 1400.]])
>>> scipy.dot(X, yT)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: matrices are not aligned
>>>
```

*Figure 6: Illustration of vector transformation by multiplication with a matrix. The vector **y** is a column matrix (standard for mathematical vectors) and hence **yT** is a row matrix. **X** is a square matrix. Hence only **yT** · **X** and **X** · **y** operations are allowed.*

Finally, here is a neat tip. There is a difference between a **vector** and a **column** or **row** matrix. This comes out clearly in Figure 7.

```
>>> A = scipy.array([[1,2,3],[4,5,6],[7,8,9]])
>>> x = scipy.array([10, 20, 30])
>>> A.shape
(3, 3)
>>> x.shape
(3,)
>>> y = x.reshape((3,1))
>>>
>>> y
array([[10],
       [20],
       [30]])
>>> scipy.dot(A,x)
array([140, 320, 500])
>>> scipy.dot(x,A)
array([300, 360, 420])
>>> scipy.dot(y.T, A)
array([[300, 360, 420]])
>>> scipy.dot(A, y)
array([[140],
       [320],
       [500]])
>>> |
```

*Figure 7: Illustration of the difference between a **vector** and a **row** or **column** matrix. A is a 3×3 matrix. x is a **vector** of length 3: hence $x.shape$ is (3,): not three rows or columns, just 3. We can reshape x as a column matrix y as shown. The shape of y is 3×1. Hence $y^T A$ and Ay are allowed operations (as shown). The result of these operations is the same as $x \cdot A$ and $A \cdot x$ respectively i.e. if you take an inner product of a square matrix with a **vector**, the vector is **automatically** recast as the appropriate row or column matrix.*

2.2 Broadcasting

Right. Let's see what **broadcasting** is. Figure 8 shows an illustration of how a scalar b is broadcast into an array to multiply another array X elementwise. The Figure 9 shows how a row vector **y** is broadcast into an array, whose rows are the vector **y** repeated, to multiply X and the Figure 10 shows how a column vector **y** is broadcast into an array, whose columns are the vector **y** repeated. In Figure 10 also note the use of the **reshape** method.

Note: *Broadcasting will also work for other arithmetic operations (+, -, / etc)*

```
    Python 1 ☒       ▶ simulate.py ☒
>>> X = scipy.array([[0.1, 0.2, 0.3],[1.0, 2.0, 3.0],[10.0, 20.0, 30.0], [100.0, 200.0, 300.0]])
>>> X
array([[  1.00000000e-01,   2.00000000e-01,   3.00000000e-01],
       [  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [  1.00000000e+02,   2.00000000e+02,   3.00000000e+02]])
>>> b = 10.0
>>> b*X
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [  1.00000000e+02,   2.00000000e+02,   3.00000000e+02],
       [  1.00000000e+03,   2.00000000e+03,   3.00000000e+03]])
>>> B = scipy.array(4*[[10,10,10]])
>>> B
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]])
>>> B*X
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [  1.00000000e+02,   2.00000000e+02,   3.00000000e+02],
       [  1.00000000e+03,   2.00000000e+03,   3.00000000e+03]])
>>>
```

*Figure 8: Illustration of broadcasting a scalar. When the 4×3 array X is multiplied by the scalar b, the result is the same as **broadcasting** the scalar as a 4×3 array with elements $B_{ij} = b$ and multiplying with X elementwise i.e. if $B*X=C$ then $C_{ij} = B_{ij} * X_{ij}$.*

```
>>> X = scipy.array([[0.1, 0.2, 0.3],[1.0, 2.0, 3.0],[10.0, 20.0, 30.0], [100.0, 200.0, 300.0]])
>>> X
array([[  1.00000000e-01,   2.00000000e-01,   3.00000000e-01],
       [  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  1.00000000e+01,   2.00000000e+01,   3.00000000e+01],
       [  1.00000000e+02,   2.00000000e+02,   3.00000000e+02]])
>>> y = scipy.array([10, 20, 30])
>>> y
array([10, 20, 30])
>>> X*y
array([[  1.00000000e+00,   4.00000000e+00,   9.00000000e+00],
       [  1.00000000e+01,   4.00000000e+01,   9.00000000e+01],
       [  1.00000000e+02,   4.00000000e+02,   9.00000000e+02],
       [  1.00000000e+03,   4.00000000e+03,   9.00000000e+03]])
>>> y*X
array([[  1.00000000e+00,   4.00000000e+00,   9.00000000e+00],
       [  1.00000000e+01,   4.00000000e+01,   9.00000000e+01],
       [  1.00000000e+02,   4.00000000e+02,   9.00000000e+02],
       [  1.00000000e+03,   4.00000000e+03,   9.00000000e+03]])
>>> Y = scipy.array(4*[[10,20,30]])
>>> X*Y
array([[  1.00000000e+00,   4.00000000e+00,   9.00000000e+00],
       [  1.00000000e+01,   4.00000000e+01,   9.00000000e+01],
       [  1.00000000e+02,   4.00000000e+02,   9.00000000e+02],
       [  1.00000000e+03,   4.00000000e+03,   9.00000000e+03]])
>>> Y
array([[10, 20, 30],
       [10, 20, 30],
       [10, 20, 30],
       [10, 20, 30]])
>>>
```

*Figure 9: Illustration of broadcasting a row vector. When the 4×3 array X is multiplied by the row vector y, the result is the same as **broadcasting** the vector as a 4×3 array with the rows as repeated vectors y.*

```
>>> X = scipy.array([[0.1, 0.2, 0.3],[1.0, 2.0, 3.0],[10.0, 20.0, 30.0], [100.0, 200.0, 300.0]])
>>> y = scipy.array([10.0, 20.0, 30.0, 40.0])
>>> y = y.reshape((4,1))
>>> y
array([[ 10.],
       [ 20.],
       [ 30.],
       [ 40.]])
>>> X*y
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  2.00000000e+01,   4.00000000e+01,   6.00000000e+01],
       [  3.00000000e+02,   6.00000000e+02,   9.00000000e+02],
       [  4.00000000e+03,   8.00000000e+03,   1.20000000e+04]])
>>> y*X
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  2.00000000e+01,   4.00000000e+01,   6.00000000e+01],
       [  3.00000000e+02,   6.00000000e+02,   9.00000000e+02],
       [  4.00000000e+03,   8.00000000e+03,   1.20000000e+04]])
>>> Y = scipy.array([3*[10.0],3*[20.0],3*[30.0],3*[40.0]])
>>> Y
array([[ 10.,  10.,  10.],
       [ 20.,  20.,  20.],
       [ 30.,  30.,  30.],
       [ 40.,  40.,  40.]])
>>> X*Y
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  2.00000000e+01,   4.00000000e+01,   6.00000000e+01],
       [  3.00000000e+02,   6.00000000e+02,   9.00000000e+02],
       [  4.00000000e+03,   8.00000000e+03,   1.20000000e+04]])
>>> Y*X
array([[  1.00000000e+00,   2.00000000e+00,   3.00000000e+00],
       [  2.00000000e+01,   4.00000000e+01,   6.00000000e+01],
       [  3.00000000e+02,   6.00000000e+02,   9.00000000e+02],
       [  4.00000000e+03,   8.00000000e+03,   1.20000000e+04]])
>>>
```

*Figure 10: Illustration of broadcasting a column vector. When the 4×3 array X is multiplied by the column vector y, the result is the same as **broadcasting** the vector as a 4×3 array with the columns as repeated vectors y. Note also the use of the **reshape** method.*

2.3 Elementary Linear Algebra

You can explore the *scipy.linalg* module by yourself for now. I will fill out this part later. I want to go ahead to solving differential equations.

3. Integration

The *scipy.integrate* library contains many modules for integration. There are modules for the trapezoidal rule, simpson rule etc.

However, the current favourite way to do integration is by *quadrature* which is simply trying to fit the function to a basis-set of easily integrable functions and then integrate those functions. The *Gaussian quadrature*[1] uses a set of *orthogonal polynomials* (Legendre, Chebyshev etc) to as the basis-sets for the fitting.

However, the most preferred quadrature right now is the *Curtis-Clenshaw* quadrature[2] implemented in *scipy* as *scipy.integrate.quad*[3]. Its use is illustrated in Figure 11.

```
>>> import scipy, scipy.integrate
>>> def f(x):
...     F = scipy.sin(x)
...     return F
...
>>> a = 0.0; b = scipy.pi
>>> int_result = scipy.integrate.quad(f, a, b)
>>> int_result
(2.0, 2.220446049250313e-14)
>>>
>>> -(scipy.cos(b) - scipy.cos(a))
2.0
>>>
>>> def g(x, A, B):
...     F = A*scipy.sin(B*x)
...     return F
...
>>> int_result = scipy.integrate.quad(g, a, b, args = (A,B))
>>> int_result
(1.3333333333333333, 1.15944764181609240e-12)
>>>
>>> -A/B*(scipy.cos(B*b) - scipy.cos(B*a))
1.3333333333333333
>>>
```

Labels pointing to the code: **Lower Limit**, **Upper Limit**, **Absolute Error of Integration**, **Result of Integration**, **Additional Arguments**

*Figure 11: Illustration of using scipy.integrate.quad for a simple integration. Please note how the module returns a **tuple** the first element of which is the result of the integration and the second element is the **estimated absolute error** of integration. Notice how **additional arguments** can be passed to the integration using the **args** keyword.*

---

[1] Wikipedia is your friend: https://en.wikipedia.org/wiki/Gaussian_quadrature

[2] Wikipedia is your friend: https://en.wikipedia.org/wiki/Clenshaw%E2%80%93Curtis_quadrature but for those to like it on the rocks:
http://homerreid.dyndns.org/teaching/18.330/Notes/ClenshawCurtis.pdf

[3] Three Cheers for Scipy! Please note that you have *all this work* absolutely free of cost! Try to pass it forward …

Note: *A couple of things important to note in* Figure 11. *The first thing to note is the* **you have to tell** *quad which function to integrate between which limits. And second, notice the you can* **pass a function as an argument!** *This is because, for Python, it is simply another object whose pointer you are passing. This is one of the reasons Python is such a powerful way to code!*

In Figure 11 notice how we passed additional arguments to the function using the **args** keyword. Experiment with this a little until it sinks it. We are going to use it extensively.

4. Ordinary Differential Equations

The *scipy.integrate.odeint* module packs a very advanced *marching method* (**Adams-Bashforth**[4] for non-stiff problems and **Backward-Differentiation Formula**[5] for stiff ones) to solve **initial value** problems.

4.1 Initial Value Problems

Here is a common initial value problem:

$$\frac{d^2 y}{dx^2} - 4y = e^x$$

$$y(0) = 1.0; \left.\frac{dy}{dx}\right|_{x=0} = 2.0$$

The solution is, of course[6]:

$$y = \frac{5}{4}e^{2x} + \frac{1}{12}e^{-2x} - \frac{1}{3}e^x$$

But suppose we could not solve this analytically and we are forced to ask the machine to help. But *how* do we ask? First off, we have to acknowledge that the machine speaks a very *formal* language. We have to tell it *explicitly* and *precisely* what we want.

But what do we want? An analytical expression? No, not really. What we can be satisfied with is a value of y_i for each value of x_i for $x_i \subset [x_1, x_2, \dots, x_n]$ i.e. for several values of x. *This* is something the computer can provide. And fairly easily. So let's see how.

A *marching* method essentially only solves *first order* problems. But this is not any trouble at all. We can easily split our second order problem into two first order ones, like so:

$$\frac{dy}{dx} = z; \frac{dz}{dx} = 4y + e^x$$

$$y(0) = 1.0; \ z(0) = 2.0$$

The solution will therefore be a set of $\{y_i\}$ and a set of $\{z_i\}$ for every value of x_i in $\{x_i\}$. So let's consider what we have: if we are given, at any point, a value of x, a value of y and a value of z,

---

[4] https://en.wikipedia.org/wiki/Linear_multistep_method
[5] https://en.wikipedia.org/wiki/Backward_differentiation_formula
[6] After some algebra which I won't bore you with …

then we are in a position to calculate $\frac{dy}{dx}$ and $\frac{dz}{dx}$. Further, once we know $\frac{dy}{dx}$ and $\frac{dz}{dx}$, we are in a position to find x, y and z for the next higher value of x.

The tuple (x, y, z) can be thought of as the **position vector** of a point in **phase-space.**

Hence, all we need to do to start up the marching method is:

1. Tell the machine how to calculate first derivatives at a given point in phase space.
2. Tell the machine at what point to begin (i.e. supply initial condition).
3. Tell the machine at which values of x you want to calculate y and z.
4. Give it all to the machine.
5. Interpret the results

So let's do this. You can follow along in Figure 12.

Step 1: So how do go about doing (1)? Really simple: **define a function** that takes in x, y, z and spits out $\frac{dy}{dx}$ and $\frac{dz}{dx}$. And if the derivatives are returned *in the same order* as the arguments are sent in, we don't need to specify which is which.

Hence, let's make a new Python module called $myfirstode.py$, import certain things and then write this function (we will call it *deriv*) in it. We have placed y *and* z in a vector $Y = [y, z]$.

Step 2: Now we need to supply the initial conditions. That is also really easy. We just need the vector Y at $x = 0$. Hence $Y_0 = [y_0, z_0]$.

Step 3: Now we have to tell it which values of x you want the solution at. That is also very simple: we use $scipy.linspace$ for 30 values of x between 0 and 1.

Step 4: Now to give it all to the machine! The syntax of *scipy.integrate.odeint*[7] defines precisely what goes where.

Step 5: The result is an array the columns of which are the solutions for each of the dependent variables (see Figure 12). These can be plotted against x to get a nice graph: see Figure 13.

---

[7] Google is *really* useful: http://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.integrate.odeint.html

```
1 import scipy
2 import scipy.integrate
3 import matplotlib.pyplot as plt
4
5 #Step 1:  Write the function deriv
6 def deriv(Y, x, a, b):
7     [y, z] = Y
8     dybydx = z
9     dzbydx = a*y + b*scipy.exp(x)
10    return [dybydx, dzbydx]
11
12 #Step 2: Define initial conditions
13 y0, z0 = 1.0, 2.0
14 Y0 = [y0, z0]
15
16 #Step 3:  Define the values of x at which you want the solutions
17 x = scipy.linspace(0.0, 1.0, 30)
18 a, b = 4.0, 1.0
19
20 #Step 4: Now give it all to the machine
21 Soln = scipy.integrate.odeint(deriv, Y0, x, args = (a, b))
22
23 #Step 5: Interpret the results.  Soln is a 10x2 matrix. The columns are y and z.
24 y = Soln[:,0]
25 z = Soln[:,1]
26
27 fig = plt.figure(); fig.show()
28 axy, axz = fig.add_subplot(121), fig.add_subplot(122)
29 axy.title.set_text('y vs x'); axz.title.set_text('z vs x')
30 axy.xaxis.label.set_text('x'); axy.yaxis.label.set_text('y')
31 axz.xaxis.label.set_text('x'); axz.yaxis.label.set_text('z')
32 axy.plot(x, y, 'r')
33 axz.plot(x, z, 'r')
34
35 #Lets also plot the theoretical value of y
36 y_theo = lambda x: (5.0/4.0)*scipy.exp(2*x) + (1.0/12.0)*scipy.exp(-2*x) - scipy.exp(x)/3.0
37 axy.plot(x, y_theo(x), 'c', linewidth = 5.0, alpha = 0.5)
38 fig.canvas.draw()
```

Note the order

alpha = degree of transparency

Figure 12: Illustration of the use of scipy.integrate.odeint. The graphs appear in Figure 13.
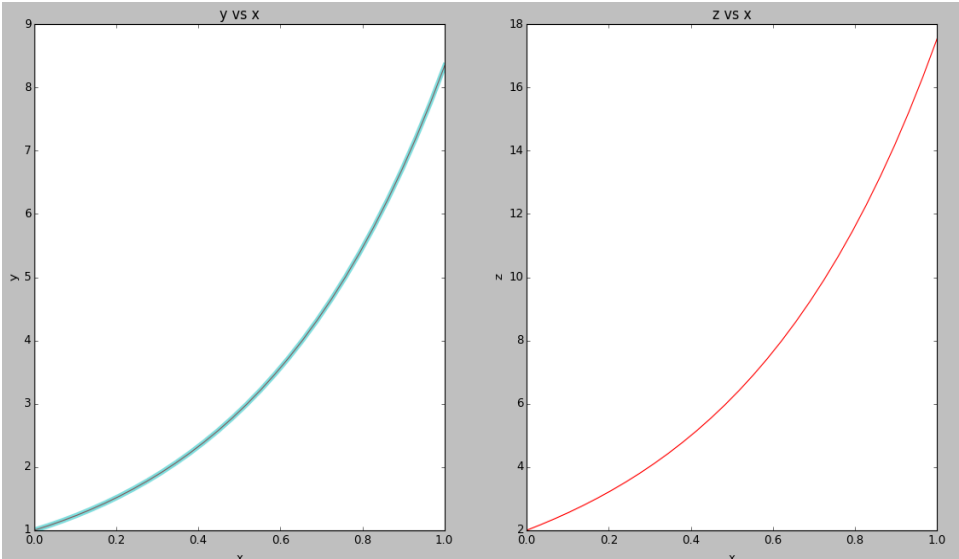

Figure 13: Results of the solution of Figure 12. The panel on the left has two curves: the numerical solution in red on which is superimposed the analytical solution in cyan.

4.2 Boundary Value Problems

But, of course, not all problems are initial value problems. Sometimes they are **boundary value problems**. Here is a common boundary value problem:

$$\frac{d\mathbf{Y}}{dz} = -\mathbf{h}(\mathbf{K_y}\mathbf{Y} - \mathbf{K_x}\mathbf{X})$$

$$\frac{d\mathbf{X}}{dz} = -\mathbf{h}(\mathbf{K_y}\mathbf{Y} - \mathbf{K_x}\mathbf{X})$$

$$\mathbf{Y}(z = 0) = \mathbf{Y_0}; \mathbf{X}(z = L) = \mathbf{X_L}$$

Messed up? What in seven blooming flower-beds am I trying to say? No need to get flustered: we have seen these before. Let's write it out in a less elegant way:

$$\frac{dY_i}{dz} = -h_i(K_{yi}Y_i - K_{xi}X_i)$$

Where: $K_{yi} = \frac{A_i}{\sum_{j=1}^{N} Y_j}$ and $K_{xi} = \frac{B_i}{\sum_{j=1}^{N} X_j}$. Seems familiar? This is the countercurrent contacting problem with $\mathbf{Y}$ as the vector of the molar flows of one of the phases and $\mathbf{X}$ as the vector of the molar flows of the other phase. Hence $A_i = H_i P_T$ in case of gas phase (H_i = Henry Coefficient ad P_T = total pressure) and $B_i = \rho_L$ = molar density of liquid.

Hence, for the above problem, $\mathbf{K_x}, \mathbf{K_y}, \mathbf{h}$ are $N \times N$ **diagonal** matrices.

Writing it in the matrix form is clearly a more elegant way of going about addressing them: both on paper **and** in code! For example, we can do the following manipulation:

$$\frac{d\mathbf{Y}}{dz} = \frac{d\mathbf{X}}{dz}$$

i.e.

$$\mathbf{Y_0} - \mathbf{Y} = \mathbf{X_0} - \mathbf{X}$$

i.e.

$$\mathbf{X} = \mathbf{Y} + (\mathbf{X_0} - \mathbf{Y_0})$$

Note: We have *no idea* what the value of $\mathbf{X_0}$ might be.

This allows us to simplify our problem to this one[8]:

$$\frac{d\mathbf{Y}}{dz} = -\mathbf{h}\left((\mathbf{K_y} - \mathbf{K_x})\mathbf{Y} + \mathbf{K_x}(\mathbf{Y_0} - \mathbf{X_0})\right) = -\mathbf{h}(\mathbf{K_y}\mathbf{Y} - \mathbf{K_x}\mathbf{X})$$

with the boundary condition: $\mathbf{Y}(z = 0) = \mathbf{Y_0}$.

Yay! We have an **initial value problem!** Hooray! Hooray!

---

[8] Please check the algebra!

Umm... no. Not that I want to be a wet-blanket or anything, but we **still don't have any idea what X_0 is!** We only have X_L. And without X_0, we can't evaluate $\frac{dY}{dz}$ and the marching method won't start. We seem to have reached an *impasse*.

Hang on though! Why can't we *guess* an X_0 and see what we get at $z = L$ i.e. X_L^{guess} and if it is different from X_L we can correct it. *By jove that's brilliant!*[9]

And we have *just the thing* to correct it with: our old friend the Levenberg-Marquadt algorithm implemented in *scipy.optimize.leastsq*. We have already seen it in action before. Let's use it again.

To use the machine, we need numbers. So here they are:

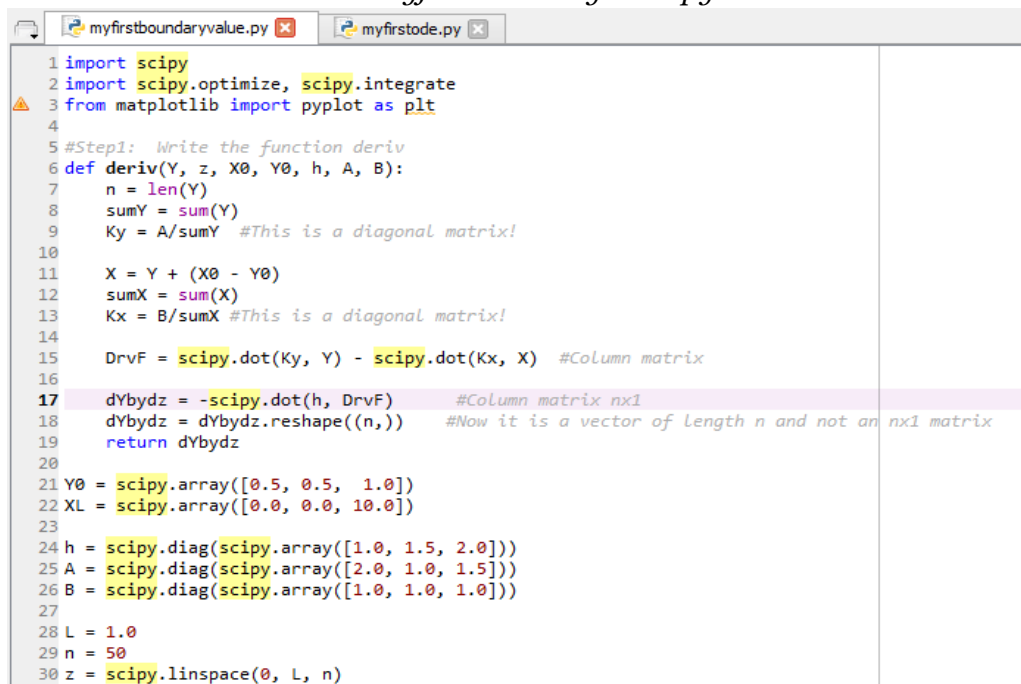$$Y_0 = [0.5, 0.5, 1.0]$$

$$X_L = [0.0, 0.0, 10.0]$$

$$h = diag([1.0, 1.5, 2.0])^{10}$$

$$A = diag([2.0, 1.0, 1.5]$$

$$B = diag([1.0, 1.0, 1.0])$$

$$L = 1.0$$

So lets start with a new file: *myfirstboundaryvalue.py*.

```python
import scipy
import scipy.optimize, scipy.integrate
from matplotlib import pyplot as plt

#Step1:  Write the function deriv
def deriv(Y, z, X0, Y0, h, A, B):
    n = len(Y)
    sumY = sum(Y)
    Ky = A/sumY   #This is a diagonal matrix!

    X = Y + (X0 - Y0)
    sumX = sum(X)
    Kx = B/sumX #This is a diagonal matrix!

    DrvF = scipy.dot(Ky, Y) - scipy.dot(Kx, X)   #Column matrix

    dYbydz = -scipy.dot(h, DrvF)       #Column matrix nx1
    dYbydz = dYbydz.reshape((n,))     #Now it is a vector of length n and not an nx1 matrix
    return dYbydz

Y0 = scipy.array([0.5, 0.5,  1.0])
XL = scipy.array([0.0, 0.0, 10.0])

h = scipy.diag(scipy.array([1.0, 1.5, 2.0]))
A = scipy.diag(scipy.array([2.0, 1.0, 1.5]))
B = scipy.diag(scipy.array([1.0, 1.0, 1.0]))

L = 1.0
n = 50
z = scipy.linspace(0, L, n)
```

Figure 14: First part of the boundary value problem. We have defined the derivative generator and the inputs.

---

[9] It *is* brilliant. And well known in the field for more than a century. It's called the "shooting method" to solve boundary value problems using methods developed for initial value problems.
[10] $diag(a)$ is a *diagonal matrix* $\mathbf{A}$ given by: $A_{ij} = \delta_{ij}a_i$ where a_i are the elements of the vector $\mathbf{a}$.

The Figure 14 shows the first part of the code where we have defined the derivative generator function (*deriv*) as well as the inputs.

Note: The number of dependent variables in the problem can be increased simply by adding elements to the input matrices Y, h, A, B and X. The code remains the same! This is why we want to use **vectors and arrays**.

Now we need the function that generates an error for the optimization. This is shown in Figure 15

```
31
32 def error(X0, z, XL, Y0, h, A, B):
33     Soln = scipy.integrate.odeint(deriv, Y0, z, args = (X0, Y0, h, A, B))
34     YL = Soln[-1,:]
35     XL_sm = YL + (X0 - Y0)
36     error = XL_sm - XL
37     return error
38
39 X0_guess = XL + 0.0
40 X0_opt, err = scipy.optimize.leastsq(error, X0_guess, args = (z, XL, Y0, h, A, B))
41 YY = scipy.integrate.odeint(deriv, Y0, z, args = (X0_opt, Y0, h, A, B))
42 XX = YY + (X0_opt - Y0)
```

Figure 15: The code to generate an error from a guess of X_0 and to call the Levenberg-Marquadt algorithm to correct the guess.

Now let's plot the **driving potential** i.e. $K_{yi}Y_i$ and $K_{xi}Xi$ for each component. The code appears in Figure 16 and the graph in Figure 17.

```
43
44 #Plotting the driving force
45 Kyy = scipy.array([A/sum(Y) for Y in YY])
46 Kxx = scipy.array([B/sum(X) for X in XX])
47 KyyYY = scipy.array([scipy.dot(Kyy[i], YY[i]) for i in xrange(len(YY))])
48 KxxXX = scipy.array([scipy.dot(Kxx[i], XX[i]) for i in xrange(len(XX))])
49
50 fig = plt.figure()
51 for i in xrange(len(Y0)):
52     ax = fig.add_subplot(1, len(Y0), i+1)
53     PotY = KyyYY[:,i]
54     PotX = KxxXX[:,i]
55     maxPot = max([max(PotY), max(PotX)])
56     ax.plot(PotY, z, 'r')
57     ax.plot(PotX, z, 'b')
58     ax.title.set_text('Driving Force for Component %d'%(i))
59     ax.xaxis.label.set_text('Potential')
60     ax.yaxis.label.set_text('z')
61     ax.axis([0.0, maxPot, 0.0, L])
62     ax.legend(['PhaseY','PhaseX'])
63 fig.canvas.draw()
64 fig.show()
```

Figure 16: Code for plotting the driving potential of each component. Try to figure out the code for yourself.
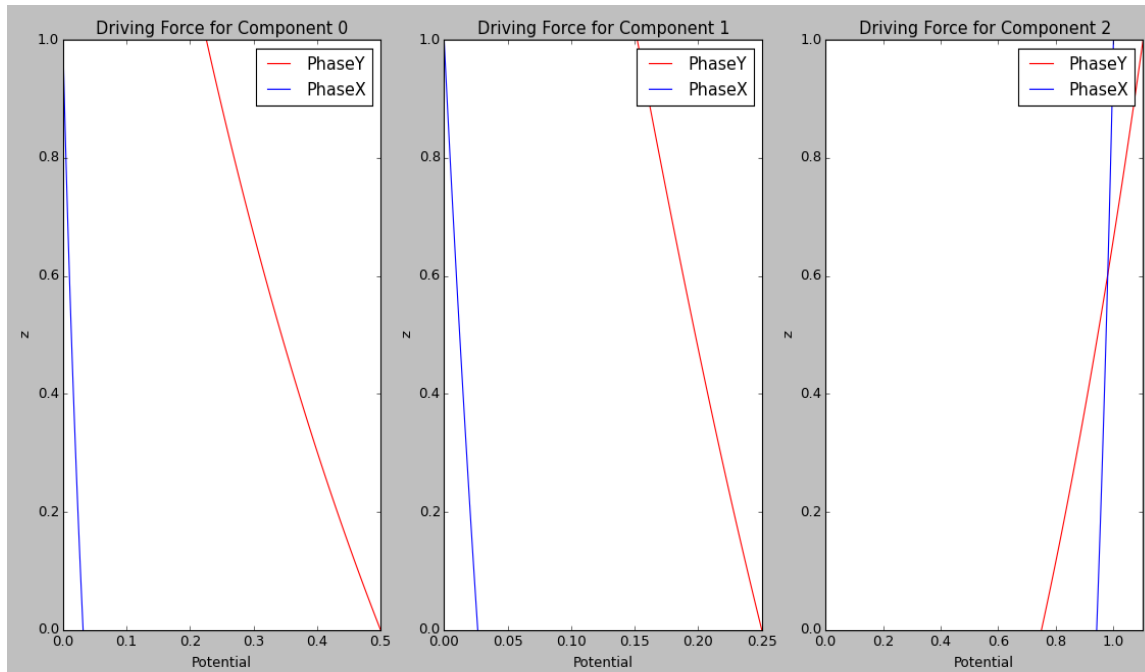
Figure 17: Plots of the driving potential for each component. Notice the graph of Component 2. Is it wrong? Why or why not?

There seems to be something unusual about the potential for *Component 2* in Figure 17. Can you express it formally? Why do you think this has happened? Is it wrong?

Do you see a man who is good at what he does? He will serve kings. He will not serve ordinary men.

- Proverbs 22:29