Python for Scientific Data Analysis

NumPy/SciPy

Section 5: Basic Linear Algebra with NumPy and SciPy

Linear Algebra is a cornerstone of computational mathematics. The simple linear algebra you probably learned in high school or undergrad consists of small vectors and matrices. But the real power with linear algebra rests in very large matrices and vectors. Since you cannot realistically solve these by hand, computers come in. Hence, the importance of numerical linear algebra.

Both NumPy and SciPy have numerous linear algebra routines. In this section, we will focus on the simplest ones with NumPy, describing how to do basic linear algebra operations.

Later in the course, after we have learned *Matplotlib*, *Pandas*, and *AstroPy* , we will build upon this foundation to describe more advanced linear algebra operations with both NumPy and SciPy and advanced modeling with SciPy. But for now, this short introduction to linear algebra and modeling will allow us to do useful things with packages.

Creating Vectors and Matrices with NumPy

Creating vectors and matrices for linear algebra operations is straightforward with NumPy. In fact, you have done them already. E.g.

```
a_vector=np.array([1,2,3,4,5,6]) #orientless
#array([1, 2, 3, 4, 5, 6])
a_vector.shape
#(6,)

a_row_vector=np.array([[1,2,3,4,5,6]])
#array([[1, 2, 3, 4, 5, 6]])
a_row_vector.shape
#(1, 6)

a_column_vector=np.array([1],[2],[3],[4],[5],[6])
#array([[1],
        [2],
        [3],
        [4],
        [5],
        [6]])

a_column_vector.shape
#(6, 1)

a_matrix=np.array([[1,2,3],[4,5,6]])
a_matrix.shape
#(2,3)
```

Note that the first example -- *a_vector* -- is orientationless: it is neither a row nor column vector but just a 1D list of numbers in NumPy. Orientation in NumPy is given by brackets `[ ]` . The outermost brackets group all of the numbers together into one object. Then, each additional set of brackets indicates a row: so a row vector has all numbers in one row, while a column vector has multiple rows, with each row containing one number. A matrix then will have multiple rows with each row containing more than one number.

Array arithmetic rules -- e.g. adding, subtracting -- follow those described in Section 2.

Basic Linear Algebra Operations

Vector Norms

The magnitude or *norm* of a vector is the distance from head to tail of the vector. It is computed using the standard Euclidean distance formula: the square-root of the sum of squared vector elements:

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

In Python this can be computed as `np.linalg.norm(v)`.

Note, since technically this is the L_2 norm the `v` has that subscript of `2`. There are other norms -- the L_1 ("Manhattan" norm) which is $\|v\|_1 = \sum_{i=1}^{n} |v_i|$ (i.e. the sum of the absolute value of array elements), etc. -- but you can read up on those if you want in case you don't already know them.

Here's an example of computing the vector norm with NumPy:

```
import numpy as np
a=np.array([[3,4,5,6,7]])

np.linalg.norm(a)
#11.61895003862225

np.linalg.norm(a,2) #note: same thing ... you are asking NumPy to compute
 the L2 norm
#11.61895003862225

np.linalg.norm(a,1) #note: different! ... you are asking NumPy to compute
 the L1 or 'Manhattan' norm
#7.0
```

Note the full capability of `norm` is complex and best left for official documentation, which is here:https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

Sometimes we don't want the vector as-is but want a corresponding *unit vector*, where $\|v\| = 1$. To produce this unit vector you have to take the input vector and divide by its norm.

Matrix Determinant

The *determinant* is a key property associated with a square matrix. It has a special importance when we later get to inverting matrices or do eigendecomposition of them. The determinant of matrix A is usually noted as $det(A)$ or |A|.

As you might remember, it's pretty easy to calculate determinants for small matrices, e.g. 2x2:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

For much larger matrices this gets quickly unwieldy, although you may remember how to use

cofactor expansion to compute the determinants of slightly larger matrices:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = 1 \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} - 2 \begin{bmatrix} 4 & 6 \\ 7 & 9 \end{bmatrix} + 3 \begin{bmatrix} 4 & 5 \\ 7 & 8 \end{bmatrix} = -3 + 12 - 9 = 0$$

Thankfully, we can just ask NumPy:

`matrix_det=np.linalg.det([matrix])` .

Or SciPy

`from scipy import linalg; matrix_det=linalg.det([matrix])` .

e.g.

```
import numpy as np
from scipy import linalg

a=np.array([[1,2,3],[4,5,6],[7,8,9]])
a.shape #3x3 matrix

matrix_det=np.linalg.det(a)
#0
matrix_det=linalg.det(a)
#0
```

Vector and Matrix Multiplication

The real power with NumPy's linear algebra routines come from multiplying large matrices and vectors, which we obviously cannot do by hand. Routines for how to do this are below ...

Dot Product

The `dot` operation does vector-vector, matrix-vector, and matrix-matrix multiplication of two arrays: `np.dot([first array],[second array])` . E.g. `np.dot(a,b)` .

Mathematically, what this operation does for each is:

The vector-vector operation for each element i, resulting in scalar s from vectors x and y:

$$s = \sum_i x_i y_i$$

The matrix-vector operation, for each i row and j column, resulting in vector y from matrix A and vector x:

$$y_i = \sum_j A_{ij} x_j$$

The vector-matrix operation, for each i row and j column, resulting in vector y from matrix A and vector x:

$$y_j = \sum_i x_i A_{ij}$$

The matrix-matrix operation, resulting in matrix C from matrices A and B:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Again, in all cases, the operation is `np.dot([first array],[second array])`

Note, it is quite easy to get broadcasting errors when doing matrix-vector or vector-matrix operations, so here are examples of how this works:

```
import numpy as np

a=np.array([[1,2],[3,4]]) #2x2 matrix 1, 2 for row =0 and 3,4 for row =1
b=np.array([[1,2]]) #a row vector
b2=np.array([[1],[2]]) #a column vector

#np.dot(a,b) #can't do this
np.dot(a,b2) #matrix-vector multiplication
#([[5],[11]]) #column vector

np.dot(b2,a) #can't do this
np.dot(b,a)  #vector-matrix multiplication
#([[ 7, 10]]) #row vector
```

Np.Matmul

NumPy provides another, simpler routine for *matrix* multiplication, called (unsurprisingly) `np.matmul([first matrix,second matrix])`.

E.g. in our previous example of matrices A and B this matrix multiplicaiton results in matrix C:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

For this case, the call is `np.matmul(A,B)`.

Now, there will be plenty of other matrix operations of greater sophistication and power. But we will get to these later.

Outer Product

Here's another useful operation: the *outer product*. The outer product of two vectors creates a matrix. In order to make this work, one vector must be a column vector and another must be a row vector.

e.g.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d & e \end{bmatrix} = \begin{bmatrix} ad & ae \\ bd & be \\ cd & ce \end{bmatrix}$$

In code, the outer product is computed as `np.outer(a,b)`. Below is an example:

```
v1=np.array([[1,2,3,4]])
#array([[1, 2, 3, 4]])

v2=np.array([[5],[6],[7]])*0.5
#array([[2.5],
#       [3. ],
#       [3.5]])

mat1=np.outer(v1,v2)
#array([[ 2.5,  3. ,  3.5],
#       [ 5. ,  6. ,  7. ],
#       [ 7.5,  9. , 10.5],
#       [10. , 12. , 14. ]])
```

Note also, the `np.dot` function can *also* compute an outer product PROVIDED that the first vector is a column vector and second is a row vector.

e.g.

```
b=np.array([[11,27]]) #a row vector
b2=np.array([[13],[32]]) #a column vector

result=np.dot(b2,b)

 #array([[143, 351],
      [352, 864]])

result2=np.outer(b2,b)

#array([[143, 351],
      [352, 864]])
```

Solving Basic Linear Equations

Now, we can take the power of numerical linear algebra to the next level by using it to solve systems of linear equations. *Many* coding problems I encounter are really just dressed up versions of a simple task to solve linear equations.

For instance, consider the following set of linear equations:

$x_1 + 4x_2 = 1$
$3x_1 + 4x_2 = -4$

Now, you could just solve this by hand fairly fast ($x_1 = -2.5$, $x_2 = 0.875$). But why do that when NumPy and SciPy will solve it for you & do it faster?

Recast the system of linear equations as a matrix equation of the form $\mathbf{Ax} = \mathbf{b}$. Here, $\mathbf{A}$ equals the coefficients in front of the variables: $\begin{bmatrix} 1 & 4 \\ 3 & 4 \end{bmatrix}$. And $\mathbf{b}$ equals the right-hand side of the equation: a column vector: $\begin{bmatrix} 1 \\ -4 \end{bmatrix}$. The unknown $\mathbf{x}$ equals the two variables x_1 and x_2.

In Python this is written as:

```
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix
```

We can then solve this system of linear equations by matrix *inversion*. Thankfully, NumPy and SciPy have a LOT of tools to do this in various ways. For now, we will focus on the most obvious, brute-force way: a simple inversion:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Here are various ways to solve it in NumPy:

`np.linalg.solve`

```
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

x=np.linalg.solve(a,b)
#array([[-2.5  ],
       [ 0.875]])
```

Using `np.linalg.inv` (the inversion function) and `np.dot`

```
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

np.linalg.inv(a).dot(b)
#array([[-2.5  ],
       [ 0.875]])
```

Using SciPy's `linalg.solve`

```
from scipy import linalg
linalg.solve(a,b)
#array([[-2.5  ],
       [ 0.875]])
```

Using SciPy's `linalg.inv` and `np.dot`

```
from scipy import linalg
a=np.array([[1,4,],[3,4]]) #a two-by-two matrix
b=np.array([[1],[-4]]) #a column matrix

linalg.inv(a).dot(b)
#array([[-2.5  ],
       [ 0.875]])
```
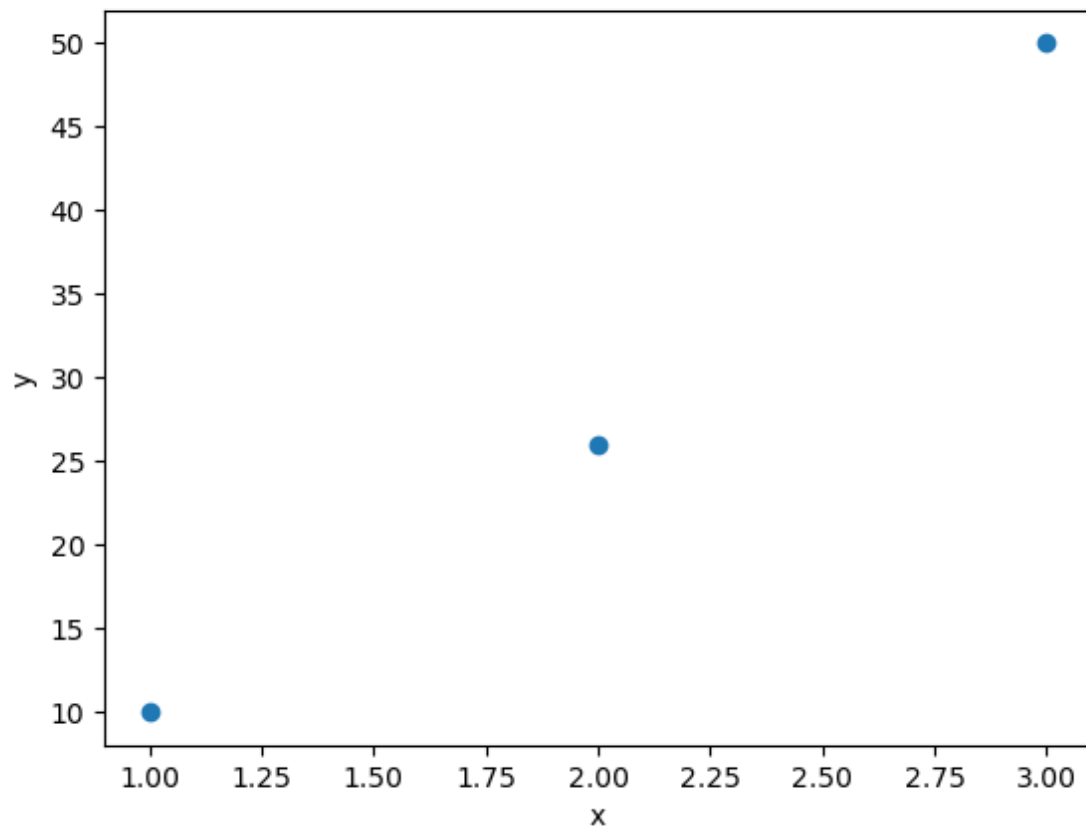
Simple Example: Curve Fitting

You can use treat curve fitting as a set of linear equations to solve and thus use matrix

inversions to determine a functional fit to the data.

Here's a super-simple example

Here we have three data points, where x = [1,2,3] and y = [10,26,50]. You might think (rightly) that this can be fit by a quadratic equation, or $y = a_o + a_1 x + a_2 x^2$.

So ... what do we know? when x = 1, then we have $10 = a_o + a_1 + a_2$. when x = 2, then we hae $26 = a_o + 2a_1 + 4a_2$. when x = 3, then we hae $50 = a_o + 3a_1 + 9a_2$.

Or ...

$$a_o + a_1 + a_2 = 10$$

$$a_o + 2a_1 + 4a_2 = 26$$

$$a_o + 3a_1 + 9a_2 = 50$$

I.e. a system of linear equations we can solve. First, we can verify that we do have a unique solution:
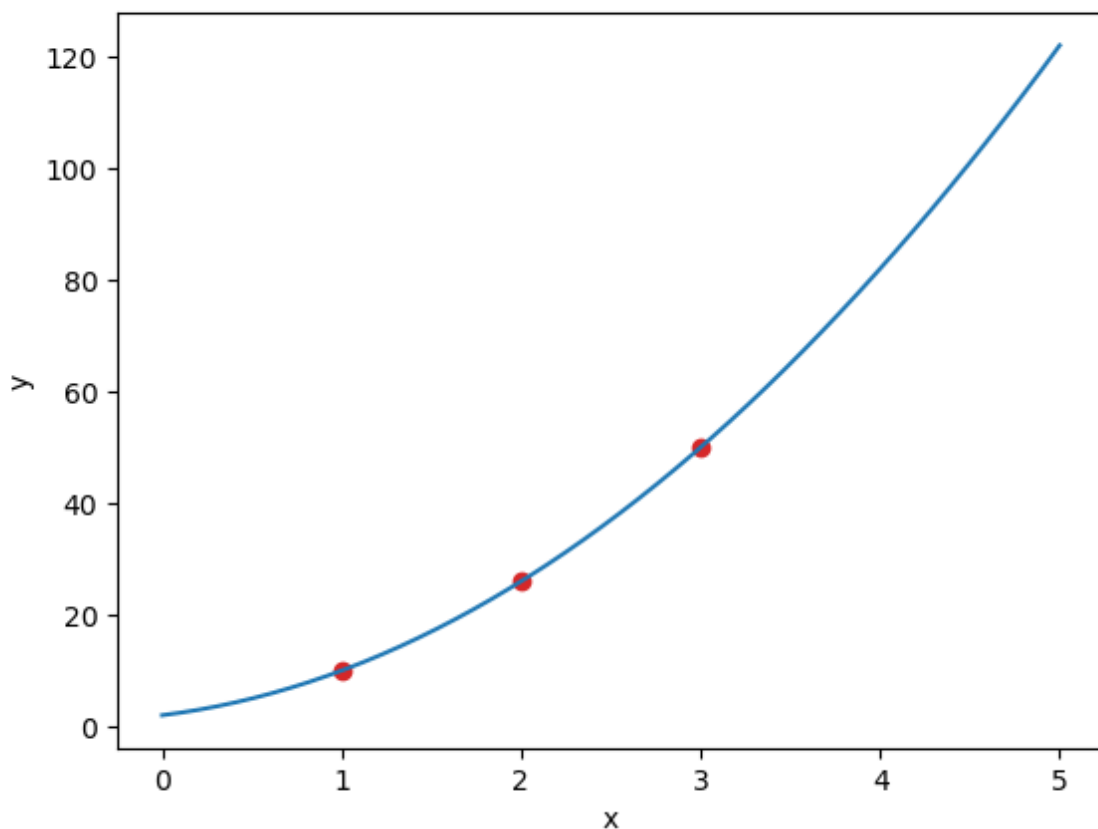
`np.linalg.det(A)` equals `2.0` .

Now, solve:

```
b = np.array([10, 26, 50]).reshape((3, 1)) #convert b to a column vector
coeff= np.linalg.inv(A).dot(b) #matrix-vector multiplication

coeff
#array([[2.],
        [4.],
        [4.]])
```

This result tells you that $a_0 = 2$, $a_1 = 4$, and $a_2 = 4$ is a solution, which is confirmed by comparing the data to the model:

(Note: in this case, it is not a *unique* solution since the determinant of A, as we find now that we have coefficients, is zero ... but small details ...).

More Complex Example: LOCI

Okay fine, the above was a simplistic example. Does anything above help us to do actual research? Yes!

Let's pretend you are interested in imaging planets. You have one target image O^T where there is a planet hiding in noise (these are called "speckles"). You have a set of other images, O^R, with a very similar noise pattern (since "speckles" are *quasi-static*) from which you want to construct a *reference* image. These reference images could be that of a different star or the same star observed at a different parallactic angle (i.e. where north on your camera is pointed differently, so this hidden planet is in a different x,y position on your detector).

After constructing this reference image, you then subtract it from the target image to reveal planets. Sounds good? But how do you *optimially* combine these images?

Well, if you ignore some small details (e.g. signal loss from the planet when doing all of these, etc.) this is really a question of

"how do you optimially weight each of those reference images so that the subtraction minimizes the sum of the squared residuals." I.e.

$\min(\sigma^2)$, where

$$\sigma^2 = \sum_i (O^T - O_i^R)^2$$

and i is the pixel number (i.e. you are summing the residuals over all of the pixels of interest.

If you have a bunch of images, making up your reference set R this converts to

$$\sigma^2 = \sum_i (O^T - \sum_k c^k O_i^k)^2,$$

where c^k is the coefficient (i.e. a number) applied to the kth image in your reference set.

If you have found a *real* minimum of σ^2, then the partial derivative with respect to the coefficients goes to zero for **all** coefficients. E.g. for the jth coefficient then ...

$$\frac{\partial \sigma^2}{\partial c^j}$$

$$\partial \sigma^2 / \partial c^j = \sum_i -2O_i^j (O^T - \sum_k c^k O_i^k) = 0$$

Reversing the summation order you get

$$\sum_k c^k (\sum_i O_i^j O_i^k) = \sum_i O_i^j O_i^T$$

Now look at the left side of the equation. The inner summation is summing over all pixels for the jth reference image and set of kth reference images. The outer summation is a summation over

k. The right side, is multiplying the values of two particular images together O^j (from the reference set) and O^T your target image.

This is just *one* partial derivative, minimizing one coefficient. You can do this for every single possible coefficient. I.e. the left side of the equation becomes a single vector (c^k over all values of k) multiplied by a 2-D matrix (a square matrix, actually ... since the sum over k and the entire set of reference images = same dimensions.

I.e. we have a system of linear equations

$$A_{jk} = \Sigma_i\, O^j_i O^k_i$$

$$x^k = c^k$$

$$b_j = \Sigma_i\, O^j_i O^T_i$$

Ax=b

We can then *invert* the matrix A, multiply it by b to get the set of coefficients that minimize the noise!

Now, this does work really well.

Below shows an example with a bunch of fake planets inserted: top image is a simple median combination of reference images, bottom shows this *"Optimized Combination of Images"* approach. In the real algorithm, you do this subtraction in small patches of the image at once. So the name of the algorith is then the

"Locally Optimized Combination of Images" algorithm (Lafreniere et al. 2007, Astrophysical Journal, 660, 770)