NumPy Basics

April 2, 2015

1 NumPy: Vectorized Array Processing in Python

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools we will use are built. Here are some of the things it provides:

- ndarray, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated *broadcasting* capabilities. We use "array" to refer to ndarrays colloquially.
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

The standard import line for NumPy is the following:

```
In [1]: import numpy as np
```

2 Arrays

NumPy arrays work fundamentally differently than the standard Python list, though they can be initialized by wrapping around a Python list, as below.

```
In [2]: my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9] # equivalently, range(1,10)
        my_array = np.array(my_list)
```

Arrays are indexed just like in standard Pyhon. For example,

```
In [3]: print my_array[0] # returns the first element of my_array
        print my_array[3:5] # returns the fourth and fifth elements of my_array
        print my_array[-2] # returns the second-to-last element of my_array
```

```
1
[4 5]
8
```

However, there are some immediate differences between Python lists and NumPy arrays. For example, consider these lines of code

```
In [4]: print my_list + my_list
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [5]: print my_array + my_array
```

```
[ 2  4  6  8 10 12 14 16 18]
```

```
In [6]: print my_array * 8
```

```
[ 8 16 24 32 40 48 56 64 72]
```

It looks as though `NumPy` arrays handle arithmetic operations differently than `Python` lists! Stay tuned to learn more about this.

2.1 Initializing `NumPy` arrays

There are four main ways to initialize an array in `NumPy`. The first is by doing what was shown above, wrapping an `NumPy` array around an existing `Python` list. A second common initialization is by using `arange`, which takes a starting point, an ending point (which is excluded), and an increment value, and generates the array. For example:

```
In [7]: np.arange(0,10,0.4)
```

```
Out[7]: array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ,  2.4,  2.8,  3.2,  3.6,  4. ,
                4.4,  4.8,  5.2,  5.6,  6. ,  6.4,  6.8,  7.2,  7.6,  8. ,  8.4,
                8.8,  9.2,  9.6])
```

This is similar to `Python`'s `range` function for lists. Another common way to initialize arrays is to use the `zeros` function, which generates an array of zeros for a given size. For example:

```
In [8]: np.zeros((2,4)) # note that np.zeros takes a tuple (m,n)
```

```
Out[8]: array([[ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.]])
```

Finally, sometimes you want to initialize an array to all 1s, so `NumPy` provides the `ones` function:

```
In [9]: np.ones((3,1)) # don't forget the tuple!
```

```
Out[9]: array([[ 1.],
               [ 1.],
               [ 1.]])
```

The full list of array creation functions are given in the following table:

2.2 Array creation functions

`array` Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default

`asarray` Convert input to ndarray, but do not copy if the input is already an ndarray

`arange` Like the built-in `range` but returns an `ndarray` instead of a list

`ones, ones_like` Produce an array of all 1's with the given shape and `dtype.ones_like` takes another array and produces a ones array of the same shape and dtype.

`zeros, zeros_like` Like `ones` and `ones_like` but producing arrays of 0's instead

`empty, empty_like` Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros` `eye, identity` Create a square `NxN` identity matrix (1's on the diagonal and 0's elsewhere)

Be careful using `empty`, because it stores the `ndarray` with whatever happens to be stored on the main stack. It's almost always preferable to initialize an "empty" array with `zeros`, because it is guaranteed what the values will be.

2.3 Data types

If necessary, you can specifiy the data type in the `ndarray` by including a `dtype=` parameter. The acceptable `dtype`s are:

`int8, uint8 (i1, u1)` Signed and unsigned 8-bit (1 byte) integer types

`int16, uint16 (i2, u2)` Signed and unsigned 16-bit integer types

`int32, uint32 (i4, u4)` Signed and unsigned 32-bit integer types

`int64, uint64 (i8, u8)` Signed and unsigned 64-bit integer types

`float16 (f2)` Half-precision floating point

`float32 (f4` **or** `f)` Standard single-precision floating point. Compatibile with `C` float

`float64 (f8` **or** `d)` Standard double-precision floating point. Compatibile with `C` double and `Python float` object

`float128 (f16` **or** `g)` Extended floating point

`complex64, complex128, complex256 (c8, c16, c32)` Complex numbers represented by 32, 64, or 128 floats, respectively

`bool (?)` Boolean type storing `True` and `False` values

`object (O)` Python object type

`string_ (S)` Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use `S10`

`unicode_ (U)` Fixed-length unicdoe type (number of bytes platform specific). Same specification semantics as `string_` (e.g. `U10`).

```
In [10]: fidentity = np.eye (5, dtype='float16')
         bidentity = np.eye (5, dtype='bool')
         sidentity = np.eye (5, dtype='string_')

         print fidentity, "\n\n"
         print bidentity, "\n\n"
         print sidentity
```

```
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
```

```
[[ True False False False False]
 [False  True False False False]
 [False False  True False False]
 [False False False  True False]
 [False False False False  True]]
```

```
[['1' '' '' '' '']
 ['' '1' '' '' '']
 ['' '' '1' '' '']
 ['' '' '' '1' '']
 ['' '' '' '' '1']]
```

2.4 Accessing array elements

`NumPy` arrays can be accessed in similar fashion to `Python list` types. For example, given an array

```
In [11]: a = np.array([1., 2., 3.])
```

we can access elements in the natural sense:

```
In [12]: print a[0]
         print a[:1]
         print a[::-1]
```

```
1.0
[ 1.]
[ 3.  2.  1.]
```

There are two ways to access elements for multidimensional arrays. The most natural accessing syntax originates from the following observation:

```
In [13]: a = np.array([[1., 2.], [3., 4.]])
         a[0]
```

```
Out[13]: array([ 1.,  2.])
```

In a 2x2 matrix, accessing the first element returns the first row vector of the matrix. Thus, we can treat `a[0]` as its *own* array, so we can access any of its elements by using the format:

```
In [14]: print a[0][0], a[0][1]
```

```
1.0 2.0
```

This is called the recursive approach. There is another approach, which is unique to `NumPy` arrays, which lets you overload the arguments in the square brackets:

```
In [15]: print a[0,0], a[0,1]
```

```
1.0 2.0
```

Exercise Take a large array, and use `%timeit` to determine which accessor approach is more efficient.

```
In [16]: %timeit a[0][0]
         %timeit a[0,0]
```

```
The slowest run took 87.64 times longer than the fastest. This could mean that an intermediate result is
1000000 loops, best of 3: 239 ns per loop
The slowest run took 27.72 times longer than the fastest. This could mean that an intermediate result is
10000000 loops, best of 3: 112 ns per loop
```

The moral of the story is, *use the `NumPy` specific approach!*

2.5 Operations between `ndarrays` and scalars

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. This is usually called *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise.

Try it! Run the following code to get a sense of how `ndarray` operations with scalars work.

```
In [17]: myarr = np.arange(1.,11.,1.)
         print myarr * 2.
         print myarr - 6.
         print myarr * myarr
         print np.log(myarr)
```

```
[  2.   4.   6.   8.  10.  12.  14.  16.  18.  20.]
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
[   1.    4.    9.   16.   25.   36.   49.   64.   81.  100.]
[ 0.          0.69314718  1.09861229  1.38629436  1.60943791  1.79175947
  1.94591015  2.07944154  2.19722458  2.30258509]
```

Exercise Write a function that takes a tuple (m, n) and a number x generates an array of containing the value x.

```
In [18]: def val_arr((m,n),x):
             return np.ones((m,n)) * x

         val_arr((5,3), 3)
```

```
Out[18]: array([[ 3.,  3.,  3.],
                [ 3.,  3.,  3.],
                [ 3.,  3.,  3.],
                [ 3.,  3.,  3.],
                [ 3.,  3.,  3.]])
```

Note that you can extract a sub-array from a main array not only by slicing sections but by choosing iterations. For example,

```
In [19]: print my_array
         print my_array[::2] # the sub-array starting at 0 to the end with even index
```

```
[1 2 3 4 5 6 7 8 9]
[1 3 5 7 9]
```

Write a function that declares an $n \times n$ “checkerboard” array of 0s and 1s.

```
In [20]: def checkerboard(n):
             checks = np.zeros((n,n))
             checks[0::2, 0::2] = 1
             checks[1::2, 1::2] = 1
             return checks

         checkerboard(5)
```

```
Out[20]: array([[ 1.,  0.,  1.,  0.,  1.],
                [ 0.,  1.,  0.,  1.,  0.],
                [ 1.,  0.,  1.,  0.,  1.],
                [ 0.,  1.,  0.,  1.,  0.],
                [ 1.,  0.,  1.,  0.,  1.]])
```

3 Lab: Invertible Matrices

NumPy provides powerful methods for randomly-generated arrays using the np.random class. The simplest random-number generation task is to produce a random number x that is a member of the set $[0, 1)$; that is, $0 \leq x < 1$. In NumPy, this is achieved by:

```
In [21]: np.random.rand()
```

```
Out[21]: 0.2970259998502053
```

Notice that if you run this cell many times, you get a different result. np.random.rand() takes n arguments which determine the shape of the output. For example, to get a 5-dimensional random vector, we can write

```
In [22]: np.random.rand(5)
```

```
Out[22]: array([ 0.13045305,  0.46411557,  0.43089943,  0.81149696,  0.32888679])
```

Each additional argument specifies another "axis" of the array output. So if you give two arguments, it produces a matrix; three arguments produces a "cube" matrix; and n arguments produces an n-tensor.

The class np.random has many more random array capabilities which you can find here. We will just use np.random.rand for this lab.

Exercise

1. Generate a three-dimensional random vector whose entries range between 4 and 8.
2. Generalize this process into a function that produces an n-dimensional random vector whose components are elments of $[a, b)$.

```
In [23]: np.random.rand(3) * 4. + 4.
```

```
Out[23]: array([ 5.82191303,  7.81879127,  7.03432306])
```

```
In [24]: def generate_random_vector(n, a, b):
             return np.random.rand(n) * (b - a) + a
```

3.1 Main lab

Consider the following problem: given an $n \times n$-dimension matrix A, what is the probability that A is invertible? One reasonable interpretation of this problem is to ask how frequently a randomly-generated matrix is invertible. Your task is to

- write a function that generates a random $n \times n$ matrix whose values fall in the set $[-1, 1)$,
- write a function to determine if such a matrix is invertible,
- write a function to approximate the probability of such an event by considering N matrices and by recording the number of these that are invertible.

Recall that a matrix A is invertible if and only if $\det(A) \neq 0$. To this end, you might want to check out linalg, `NumPy`'s linear algebra class.

Also, because we are dealing with numerical precision, it probably is best not to expect that the determinant will ever *actually* be 0. Instead, we suggest including an error tolerance.

```
In [25]: def random_matrix(n):
             return np.random.rand(n,n) * 2.0 - 1.0

         def is_invertible(mat):
             return np.abs(np.linalg.det(mat)) > 1e-10

         def simulation(n):
             counts = 0.0
             for i in range(n):
                 if is_invertible(random_matrix(n)):
                     counts += 1.0
             return counts / n
```

```
In [26]: simulation(10)
```

```
Out[26]: 1.0
```

It turns out that the overwhelming majority of random matrices are invertible. This simulation supports the theory, which you can read about on this StackExchange post. Hopefully, you're beginning to see the power of programming in problem-solving.

4 Lab: Cellular Automata

(From Nicolas P. Rougier's Numpy tutorial)

We will construct a simulation of John Conway's [Game of Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life) using the skills we have learned about `NumPy`.

> The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The "game" is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.
>
> The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
>
> 1. Any live cell with fewer than two live neighbours dies, as if by needs caused by underpopulation.
> 2. Any live cell with more than three live neighbours dies, as if by overcrowding.
> 3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
> 4. Any dead cell with exactly three live neighbours becomes a live cell.

The initial pattern constitutes the ‘seed’ of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed – births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

4.1 Getting started

The first thing to do is to create the proper `numpy` array to hold the cells. This can be done very easily:

```
In [27]: Z = np.array([[0,0,0,0,0,0],
                       [0,0,0,1,0,0],
                       [0,1,0,1,0,0],
                       [0,0,1,1,0,0],
                       [0,0,0,0,0,0],
                       [0,0,0,0,0,0]],dtype='int64')
```

You don’t have to specify the data type, as `NumPy` will interpret an array of integers as an `int64` data type anyways. Sometimes, though, it’s good practice to be specific. You can always check what datatype an array is by running

```
In [28]: Z.dtype

Out[28]: dtype('int64')
```

We can also check the shape of the array to make sure it is 6x6:

```
In [29]: Z.shape

Out[29]: (6, 6)
```

You already know how to access elements of `Z`. Write a statement to obtain the the element in the 3rd row and the 4th column.

```
In [30]: Z[2,3]

Out[30]: 1
```

Better yet, we can access a subpart of the array using the slice notation. Obtain the subarray of `Z` containing the rows 2-5 and columns 2-5, and set it equal to an `ndarray` labelled `A`.

```
In [31]: A = Z[1:5,1:5]
```

Be mindful of array pointers! Look at what happens when you run the following code:

```
In [32]: A[0, 0] = 9
         print A, "\n\n"
         print Z

[[9 0 1 0]
 [1 0 1 0]
 [0 1 1 0]
 [0 0 0 0]]


[[0 0 0 0 0 0]
 [0 9 0 1 0 0]
 [0 1 0 1 0 0]
 [0 0 1 1 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

We set the value of A[0,0] to 9 and we see immediate change in Z[1,1] because A[0,0] actually corresponds to Z[1,1]. This may seem trivial with such simple arrays, but things can become much more complex (we'll see that later). If in doubt, you can use `ndarray.base` to check easily if an array is part of another one:

```
In [33]: print Z.base is None
         print A.base is Z

True
True

In [34]: A[0,0] = 0 # put A (and Z) back to normal.
```

4.2 Counting neighbors

We now need a function to count the neighbours. We could do it the same way as for the python version, but this would make things very slow because of the nested loops. We would prefer to act on the whole array whenever possible, this is called *vectorization*.

First, you need to know that you can manipulate Z as if (and only as if) it was a regular scalar:

```
In [35]: 1 + (2*Z + 3)

Out[35]: array([[4, 4, 4, 4, 4, 4],
                [4, 4, 4, 6, 4, 4],
                [4, 6, 4, 6, 4, 4],
                [4, 4, 6, 6, 4, 4],
                [4, 4, 4, 4, 4, 4],
                [4, 4, 4, 4, 4, 4]])
```

If you look carefully at the output, you may realize that the ouptut corresponds to the formula above applied individually to each element. Said differently, we have
`(1+(2*Z+3))[i,j] == (1+(2*Z[i,j]+3))`
for any `i,j`.

Ok, so far, so good. Now what happens if we add `Z` with one of its subpart, let's say `Z[-1:1,-1:1]` ?

```
In [36]: Z + Z[-1:1, -1:1]


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-36-6833c4de92a5> in <module>()
    ----> 1 Z + Z[-1:1, -1:1]


        ValueError: operands could not be broadcast together with shapes (6,6) (0,0)
```

This raises a `Value Error` but more interestingly, numpy complains about the impossibility of *broadcasting* the two arrays together. Broadcasting is a very powerful feature of numpy and most of the time, it saves you a lot of hassle. Let's consider for example the following code:

```
In [37]: Z + 1
```

```
Out[37]: array([[1, 1, 1, 1, 1, 1],
                [1, 1, 1, 2, 1, 1],
                [1, 2, 1, 2, 1, 1],
                [1, 1, 2, 2, 1, 1],
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 1, 1]])
```

How can a matrix and a scalar be added together? Well, they can't. But `NumPy` is smart enough to guess that you actually want to add 1 to each of the element of `Z`. This concept of broadcasting is quite powerful and it will take you some time before masterizing it fully (if even possible).

However, in the present case (counting neighbours if you remember), we won't use broadcasting (uh?). But we'll use vectorize computation using the following code:

```
In [38]: N = np.zeros(Z.shape, dtype=int)

         N[1:-1,1:-1] += (Z[ :-2, :-2] + Z[ :-2,1:-1] + Z[ :-2,2:] +
                          Z[1:-1, :-2]                + Z[1:-1,2:] +
                          Z[2:  , :-2] + Z[2:  ,1:-1] + Z[2:  ,2:])
```

```
In [39]: N
```

```
Out[39]: array([[0, 0, 0, 0, 0, 0],
                [0, 1, 3, 1, 2, 0],
                [0, 1, 5, 3, 3, 0],
                [0, 2, 3, 2, 2, 0],
                [0, 1, 2, 2, 1, 0],
                [0, 0, 0, 0, 0, 0]])
```

To understand this code, have a look at the figure below:

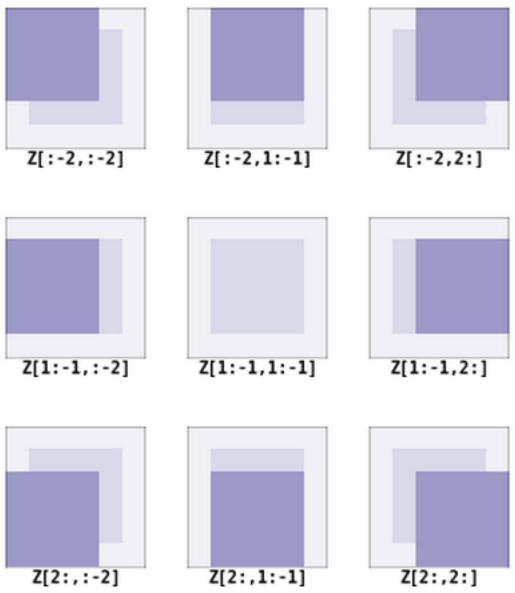


Figure 1: Slicing ndarrays

What we actually did with the above code is to add all the darker blue squares together. Since they have been chosen carefully, the result will be exactly what we expected. If you want to convince yourself, consider a cell in the lighter blue area of the central sub-figure and check what will the result for a given cell.

4.3 Cell Iteration

In a first approach, we can write the iterate function using the argwhere method that will give us the indices where a given condition is True.

```
In [40]: def iterate(Z):
             # Iterate the game of life : naive version
             # Count neighbours
             N = np.zeros(Z.shape, int)
             N[1:-1,1:-1] += (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
                              Z[1:-1,0:-2]                + Z[1:-1,2:] +
                              Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])
             N_ = N.ravel()
             Z_ = Z.ravel()

             # Apply rules
             R1 = np.argwhere( (Z_==1) & (N_ < 2) )
             R2 = np.argwhere( (Z_==1) & (N_ > 3) )
             R3 = np.argwhere( (Z_==1) & ((N_==2) | (N_==3)) )
             R4 = np.argwhere( (Z_==0) & (N_==3) )

             # Set new values
             Z_[R1] = 0
             Z_[R2] = 0
             Z_[R3] = Z_[R3]
             Z_[R4] = 1

             # Make sure borders stay null
             Z[0,:] = Z[-1,:] = Z[:,0] = Z[:,-1] = 0
```

Even if this first version does not use nested loops, it is far from optimal because of the use of the 4 argwhere calls that may be quite slow. We can instead take advantages of numpy features the following way.

```
In [41]: def iterate_2(Z):
             # Count neighbours
             N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
                  Z[1:-1,0:-2]                + Z[1:-1,2:] +
                  Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])

             # Apply rules
             birth = (N==3) & (Z[1:-1,1:-1]==0)
             survive = ((N==2) | (N==3)) & (Z[1:-1,1:-1]==1)
             Z[...] = 0
             Z[1:-1,1:-1][birth | survive] = 1
             return Z
```

Now, let's throw together a simple visualization for our Game of Life! Don't worry about the matplotlib code, we'll deal with it later.

```
In [42]: import matplotlib.pyplot as plt
         from matplotlib import animation
         from IPython.display import Image
         %matplotlib inline
```

The cell below saves our animated Game of Life as a .gif format on your computer.

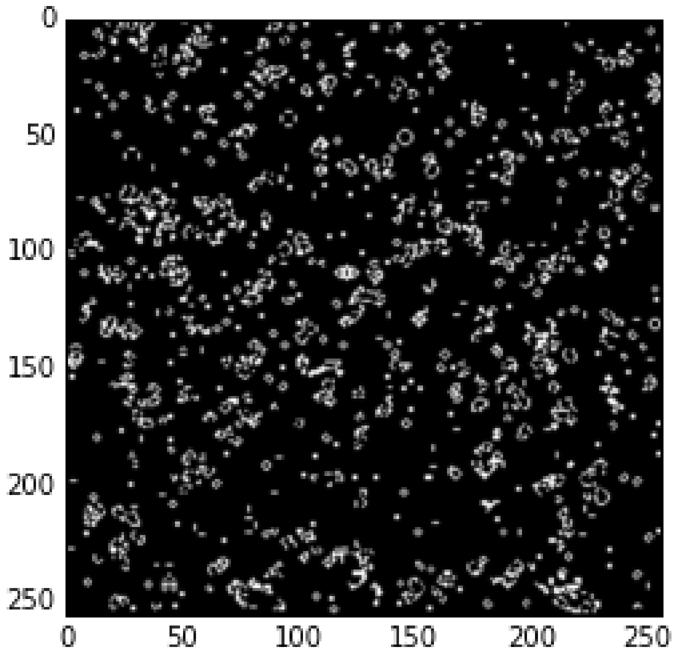
```
In [43]: Z = np.random.randint(0,2,(256,256))
         fig = plt.figure()
         ax = plt.axes()

         im = plt.imshow(Z, cmap='gray', interpolation='bicubic')

         def updatefig(*args):
             im.set_array(iterate_2(Z))
             return im,

         ani = animation.FuncAnimation(fig, updatefig, interval=200, blit=True)

         ani.save('demonstration.gif', writer='imagemagick', fps=10)
```



And now, load up the `.gif` file with:

```
In [44]: Image(url='demonstration.gif')

Out[44]: <IPython.core.display.Image object>
```

5 Lab: Gradient Descent

For those of you who have taken linear algebra, or have dealt with problems involving solutions to systems of linear equations such as

$$\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\
&\vdots \\
a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n
\end{aligned}$$

you might be familiar with Gaussian elimination, which provides a systematic way to find a solution, if one exists. Now, we can compactly write this system of equations as a *matrix-vector* equation $A\vec{x} = \vec{b}$, where

$$(A_{i,j}) = a_{i,j},$$

$$\vec{x} = (x_1, x_2, \cdots, x_n)^T,$$

and

$$\vec{b} = (b_1, b_2, \cdots, b_n)^T.$$

5.1 Problems with Gaussian Elimination

Gaussian elimination has some significant advantages. First off, it will always find a solution, if one exists, in a finite number of steps. This is a powerful property, but it comes at a cost. For an $n \times n$ matrix A, Gaussian elimination is an $\mathcal{O}(n^3)$ algorithm, which means that the number of steps required to solve the system is proportional to the **cube** of the number of rows or columns of the matrix.

This isn't so bad for an 2×2 matrix, but it is downright impossible for an 10000×10000 one. So, we need to find a better approach.

5.2 Theory for Gradient Descent

One alternative is the *method of gradient descent.* Consider a function

$$f(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{b}^T\vec{x} + c,$$

where A is a symmetric matrix, and c is a scalar constant. Now, this implies that $f : \mathbb{R}^n \to \mathbb{R}$, which means that f is a scalar function of n variables.

Specifically, f is a *quadratic* function, and it has a minimum. One can show that gradient of f is given by

$$\nabla f(\vec{x}) = A\vec{x} - \vec{b}$$

for symmetric matrices A. This means that by finding an $\vec{x}$ that satisfies $\nabla f(\vec{x}) = 0$ will simultaneously solve the system of equations $A\vec{x} = \vec{b}$.

You can visualize f as a quadratic bowl, and the method of gradient descent looks for the minimum of the bowl.

In the above image, we have the paraboloid formed by f, as well as the iterations of solutions given by the gradient descent algorithm. Alternatively, you can think about the contours of f, and for this you can think of gradient descent as updating the locations of the guesses in the following image.

Now, the gradient of a function points in the direction of greatest-initial increase of the function. Perhaps mysteriously, the *antigradient*, $-\nabla f$ points in the direction of greatest-initial *decrease* of f. So, to minimize $f(\vec{x})$, we can follow $-\nabla f$ all the way down until we reach the bottom of the bowl.

Note: From here on out, $\vec{x}$ and $\vec{b}$ will *always* denote vectors, so we will not include the $\vec{\cdot}$ notation.

5.3 Constructing the algorithm

Suppose x is the solution to the equation $Ax = b$. Imagine that you're given a *guess* solution $x_{(i)}$. Then the *residual* between the ideal solution and the guess is given by

$$r_{(i)} = b - Ax_{(i)}.$$

Coincidentally, $r_{(i)} = -\nabla f(x_{(i)})$, which means that when we update to $x_{(i+1)}$ we want to move in the direction of $r_{(i)}$.

So, when we update to $x_{(i+1)}$, we will update according to the rule

$$x_{(i+1)} = x_{(i)} + \alpha r_{(i)},$$

where α is some number.

What should α be? We want to choose α that makes the biggest improvement in one single step. It turns out that after doing some calculations, this α is given by

$$\alpha = \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}}.$$

5.3.1 Gradient Descent Pseudocode

Suppose you are given a matrix A that is square and symmetric, along with a vector b. Here is how Gradient Descent works:

1. Set $i = 0$.
2. Initialize x_0.
3. Set $r_i = b - Ax_i$.
4. Let $\alpha = \dfrac{r_i^T r_i}{r_i^T A r_i}$
5. Set $x_{i+1} = x_i + \alpha r_i$.
6. Increment i.
7. Go back to step (3).

5.3.2 Terminating the algorithm

When $f(x_{(i)})$ is very close to the minimum, the vector $x_{(i)}$ is very close to the solution x. So, the residual vector $r_{(i)}$ is very close to the zero vector. In other words, the *magnitude* of $r_{(i)}$ is approximately 0.

We can use this to build a terminating condition. Since the magnitude of $r_{(i)}$ is given by

$$\|r_{(i)}\| = r_{(i)}^T r_{(i)},$$

we can tell our algorithm to stop as soon as $\|r_{(i)}\|$ is below some error tolerance.

5.4 Try It!

We are now going to build and test an algorithm that applies the gradient descent method. Use the Gradient Descent Pseudocode to build a gradient descent algorithm. Remember, this function will take a symmetric matrix and a vector as inputs.

5.4.1 Test

Try testing your algorithm with the matrices: $A = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}$ and $b = \begin{bmatrix} 2 \\ -8 \end{bmatrix}$ Your algorithm should produce: $x = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$

5.4.2 Real Test

Now it is time to test your algorithm with an arbitrary large matrix. First, we have to generate a large matrix

```
def generate_large_matrix(n):
    A = np.random.random_intergers(-100,100,(n,n)) # Generate a square matrix with int entries in [-100
    A = A.T.dot(A) #Makes a matrix symmetric
    A[A<0] = 0 #Makes all matrix entries positive
    return A
```

Now we have to generate a vector

```
def generate_vector(n):
    b = np.random.random_intergers(-100,100,n)
    return b
```

Input the generated matrix and vector into your algorithm. (Make sure that the n for the large matrix is the same as the n for the vector.)

Did it work? (Hopefully, yes.)