07 Numerical computation

September 13, 2020

1 Introduction

Working with numbers is central to almost all scientific and engineering computations. The topic is so important that there are many dedicated libraries to help implement efficient numerical computations. There are even languages that are specifically designed for numerical computation, such as Fortran and MATLAB.

NumPy (https://www.numpy.org/) is the most widely used Python library for numerical computations. It provides an extensive range of data structures and functions for numerical computation. In this notebook we will explore just some of the functionality. You will be seeing NumPy in other courses. NumPy can perform many of the operations that you will learn during the mathematics courses.

Another library, which largely builds on NumPy and provides additional functionality, is SciPy (https://www.scipy.org/). SciPy provides some more specialised data structures and functions over NumPy. If you are familiar with MATLAB, NumPy and SciPy provide much of what is available in MATLAB.

NumPy is a large and extensive library and this activity is just a very brief introduction. To discover how to perform operations with NumPy, your best resources are search engines, such as https://stackoverflow.com/.

1.1 Objectives

- Introduction to 1D and 2D arrays (vector and matrices)
- Manipulating arrays (indexing, slicing, etc)
- Apply elementary numerical operations
- Demonstrate efficiency differences between vectorised and non-vectorised functions

2 Importing the NumPy module

To make NumPy available in our programs, we need to import the module. It has become an informal custom to import NumPy using the shortcut '`np`':

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

3 Numerical arrays

We have already seen Python 'lists', which hold 'arrays' of data. We can access the elements of a list using an index because the entries are stored in order. Python lists are very flexible and can hold mixed data types, e.g. combinations of floats and strings, or even lists of lists of lists . . .

The flexibility of Python lists comes at the expense of performance. Many science, engineering and mathematics problems involve very large problems with operations on numbers, and computational speed is important for large problems. To serve this need, we normally use specialised functions and data structures for numerical computation, and in particular for arrays of numbers. Some of the flexibility of lists is traded for performance.

3.1 One-dimensional arrays

A one-dimensional array is a collection of numbers which we can access by index (it preserves order).

3.1.1 Creating arrays and indexing

To create a NumPy array of length 10 and initially filled with zeros:

[2]:
```
x = np.zeros(10)

print(x)
print(type(x))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<class 'numpy.ndarray'>
```

The default type of a NumPy array is `float`. The type can be checked with

[3]:
```
print(x.dtype)
```

```
float64
```

You cannot, for example, add a `string` to a `numpy.ndarray`. All entries in the array have the same type.

We can check the length of an array using `len`, which gives the number of entries in the array:

[4]:
```
print(len(x))
```

```
10
```

A better way to check the length is to use `x.shape`, which returns a tuple with the dimensions of the array:

[5]:
```
print(x.shape)
```

```
(10,)
```

`shape` tells us the size of the array in *each* direction. We will see two-dimensional arrays shortly (matrices), which have a size in each direction.

We can change the entries of an array using indexing,

[6]:
```
print(x)

x[0] = 10.0
x[3] = -4.3
x[9] = 1.0

print(x)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[10.   0.   0.  -4.3  0.   0.   0.   0.   0.   1. ]
```

Remember that indexing starts from zero!

There are other ways to create arrays, such as an array of 'ones':

[7]:
```
y = np.ones(5)
print(y)
```

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

an array of random values:

[8]:
```
y = np.random.rand(6)
print(y)
```

```
[0.13941305 0.70368634 0.83328857 0.84196633 0.3153394  0.53692712]
```

or a NumPy array from a Python list:

[9]:
```
x = [4.0, 8.0, 9.0, 11.0, -2.0]
y = np.array(x)
print(y)
```

```
[ 4.   8.   9. 11. -2.]
```

Two more methods for creating arrays which we will use in later notebooks are:

- `numpy.arange`; and
- `numpy.linspace`.

They are particularly useful for plotting functions. The function `arange` creates an array with equally spaced values. It is similar in some cases to `range`, which we have seen earlier. To create the array `[0 1 2 3 4 5]` using `arange`:

[10]:
```
x = np.arange(6)
print(x)
print(type(x))
```

```
[0 1 2 3 4 5]
<class 'numpy.ndarray'>
```

Note that ‘6’ is not included. We can change the start value, e.g.:

[11]:
```python
x = np.arange(2, 6)
print(x)
```

```
[2 3 4 5]
```

The function `linspace` creates an array with prescribed start and end values (both are included), and a prescribed number on values, all equally spaced:

[12]:
```python
x = np.linspace(0, 100, 6)
print(x)
```

```
[  0.  20.  40.  60.  80. 100.]
```

The `linspace` function is used extensively for plotting, as we will see in the next notebook.

3.1.2 Array arithmetic and functions

NumPy arrays support common arithmetic operations, such as addition of two arrays

[13]:
```python
x = np.array([1.0, 0.2, 1.2])
y = np.array([2.0, 0.1, 2.1])
print(x)
print(y)

# Sum x and y
z = x + y
print(z)
```

```
[1.  0.2 1.2]
[2.  0.1 2.1]
[3.  0.3 3.3]
```

and multiplication of components by a scalar,

[14]:
```python
z = 10.0*x
print(z)
```

```
[10.  2. 12.]
```

and raising components to a power:

[15]:
```python
x = np.array([2, 3, 4])
print(x**2)
```

```
[ 4  9 16]
```

We can also apply functions to the components of an array:

[16]:
```python
# Create an array [0, /2, , 3 /2]
x = np.array([0.0, np.pi/2, np.pi, 3*np.pi/2])
```

```
print(x)

# Compute sine of each entry
y = np.sin(x)
print(y)
```

```
[0.         1.57079633 3.14159265 4.71238898]
[ 0.0000000e+00  1.0000000e+00  1.2246468e-16 -1.0000000e+00]
```

The above has computed the sine of each entry in the array `x`.

Note that the function `np.sin` is used, and not `math.sin` (which was used in previous notebooks). The reason is that `np.sin` is more general - it can act on lists/arrays of values rather than on just one value. We will apply functions to arrays in the next notebook to plot functions.

We could have computed the sine of each array entry using `for` loops:

[17]:
```
y = np.zeros(len(x))
for i in range(len(x)):
    y[i] = np.sin(x[i])

print(y)
```

```
[ 0.0000000e+00  1.0000000e+00  1.2246468e-16 -1.0000000e+00]
```

but the program becomes longer and harder to read. Additionally, in many cases it will be much slower. You might see manipulation of arrays without indexing referred to as 'vectorisation'. When possible, vectorisation is a good thing to do. We compare the performance of indexing versus vectorisation below.

3.1.3 Performance example: computing the norm of a long vector

The norm of a vector x is given by:

$$\|x\| = \sqrt{\sum_{i=0}^{n-1} x_i x_i}$$

where x_i is the ith entry of x. It is the dot product of a vector with itself, followed by taking the square root. To compute the norm, we could loop/iterate over the entries of the vector and sum the square of each entry, and then take the square root of the result.

We will evaluate the norm using two methods for computing the norm of an array of length 10 million to compare their performance. We first create a vector with 10 million random entries, using NumPy:

[18]:
```
# Create a NumPy array with 10 million random values
x = np.random.rand(10000000)
print(type(x))
```

```
<class 'numpy.ndarray'>
```

We now time how long it takes to compute the norm of the vector using the NumPy function '`numpy.dot`'. We use the Jupyter 'magic command' `%time` to time the operation:

[19]:
```
%time norm = np.sqrt(np.dot(x, x))
print(norm)
```

```
CPU times: user 15.6 ms, sys: 1.49 ms, total: 17.1 ms
Wall time: 3.94 ms
1825.7561566402942
```

The time output of interest is '`Wall time`'.

> The details of how `%time` works are not important for this course. We use it as a compact and covenient tool to measure how much time a command takes to execute.

We now perform the same operation with our own function for computing the norm:

[20]:
```
def compute_norm(x):
    norm = 0.0
    for xi in x:
        norm += xi*xi
    return np.sqrt(norm)

%time norm = compute_norm(x)
print(norm)
```

```
CPU times: user 3.06 s, sys: 53.2 ms, total: 3.12 s
Wall time: 2.86 s
1825.7561566402892
```

You should see that the two approaches give the same result, but the NumPy function is more than 100 times faster, and possibly more than 100,000 times faster!

The message is that specialised functions and data structures for numerical computations can be many orders of magnitude faster than your own general implementations. On top of that, the specialised functions are much less likely to have bugs!

3.2 Two-dimensional arrays

Two-dimensional arrays are very useful for arranging data in many engineering applications and for performing mathematical operations. Commonly, 2D arrays are used to represents matrices. To create the matrix

$$A = \begin{bmatrix} 2.2 & 3.7 & 9.1 \\ -4 & 3.1 & 1.3 \end{bmatrix}$$

we use:

[21]:
```
A = np.array([[2.2, 3.7, 9.1], [-4.0, 3.1, 1.3]])
print(A)
```

```
[[ 2.2  3.7  9.1]
 [-4.   3.1  1.3]]
```

If we check the length of `A`:

```
[22]: print(len(A))
```

```
2
```

it reports the number of rows. To get the shape of the array, we use:

```
[23]: print(A.shape)
```

```
(2, 3)
```

which reports 2 rows and 3 columns (stored using a tuple). To get the number of rows and the number of columns,

```
[24]: num_rows = A.shape[0]
      num_cols = A.shape[1]
      print("Number of rows is {}, number of columns is {}.".format(num_rows,␣
       ↪num_cols))
```

```
Number of rows is 2, number of columns is 3.
```

We can 'index' into a 2D array using two indices, the first for the row index and the second for the column index:

```
[25]: A02 = A[0, 2]
      print(A02)
```

```
9.1
```

With `A[1]`, we will get the second row:

```
[26]: print(A[1])
```

```
[-4.   3.1  1.3]
```

We can iterate over the entries of `A` by iterating over the rows, and then the entry in each row:

```
[27]: for row in A:
          print("-----")
          for c in row:
              print(c)
```

```
-----
2.2
3.7
9.1
-----
-4.0
```

```
3.1
1.3
```

Warning: NumPy has a `numpy.matrix` data structure. Its use is not recommended as it behaves inconsistently in some cases.

3.2.1 2D array (matrix) operations

For those who have seen matrices previously, the operations in this section will be familiar. For those who have not encountered matrices, you might want to revisit this section once matrices have been covered in the mathematics lectures.

Matrix-vector and matrix-matrix multiplication We will consider the matrix A:

$$A = \begin{bmatrix} 3.4 & 2.6 \\ 2.1 & 4.5 \end{bmatrix}$$

and the vector x:

$$x = \begin{bmatrix} 0.2 \\ -1.1 \end{bmatrix}$$

[28]:
```
A = np.array([[3.4, 2.6], [2.1, 4.5]])
print("Matrix A:\n {}".format(A))

x = np.array([0.2, -1.1])
print("Vector x:\n {}".format(x))
```

```
Matrix A:
 [[3.4 2.6]
 [2.1 4.5]]
Vector x:
 [ 0.2 -1.1]
```

We can compute the matrix-vector product $y = Ax$ by:

[29]:
```
y = A.dot(x)
print(y)
```

```
[-2.18 -4.53]
```

Matrix-matrix multiplication is performed similarly. Computing $C = AB$, where A, B, and C are all matrices:

[30]:
```
B = np.array([[1.3, 0], [0, 2.0]])

C = A.dot(B)
print(C)
```

```
[[4.42 5.2 ]
 [2.73 9.  ]]
```

The inverse of a matrix (A^{-1}) and the determinant ($\det(A)$) can be computed using functions in the NumPy submodule `linalg`:

[31]:
```
Ainv = np.linalg.inv(A)
print("Inverse of A:\n {}".format(Ainv))

Adet = np.linalg.det(A)
print("Determinant of A: {}".format(Adet))
```

```
Inverse of A:
 [[ 0.45731707 -0.26422764]
 [-0.21341463  0.34552846]]
Determinant of A: 9.839999999999998
```

NumPy is large library, so it uses sub-modules to arrange functionality.

A very common matrix is the *identity matrix* I. We can create a 4×4 identity matrix using:

[32]:
```
I = np.eye(4)
print(I)
```

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

4 Array slicing

When working with arrays, it is often useful to extract subsets of an array. We might want just the first 3 entries of a long array, or we might want the second column of a 2D array (matrix). These operations are known as *array slicing* (https://en.wikipedia.org/wiki/Array_slicing).

We will explore slicing through examples. We start by creating an array filled with random values:

[33]:
```
x = np.random.rand(5)
print(x)
```

```
[0.35912728 0.68416347 0.26731711 0.79101538 0.85003759]
```

Below are some slicing operations:

[34]:
```
# Using ':' implies the whole range of indices, i.e. from 0 -> (length-1)
y = x[:]
print("Slice using '[:]' {}".format(y))

# Using '1:3' implies indices 1 -> 3 (not including 3)
y = x[1:3]
print("Slice using '[1:3]': {}".format(y))
```

```
# Using '2:-1' implies indices 2 -> second-from-last
y = x[2:-1]
print("Slice using '[2:-1]': {}".format(y))

# Using '2:-2' implies indices 2 -> third-from-last
y = x[2:-2]
print("Slice using '[2:-2]': {}".format(y))
```

```
Slice using '[:]' [0.35912728 0.68416347 0.26731711 0.79101538 0.85003759]
Slice using '[1:3]': [0.68416347 0.26731711]
Slice using '[2:-1]': [0.26731711 0.79101538]
Slice using '[2:-2]': [0.26731711]
```

> Note the use of the index `-1`. The index `-1` corresponds to the last entry in the array, and `-2` the second last entry, etc. This is convenient if we know how far in from the end of an array a desired entry is. By using negative indices we can express this without reference to the length of the array.

If we want a slice to run from the start of an array, or to the end of an array, we do:

[35]:
```
# Using ':3' implies start -> 3 (not including 3)
y = x[:3]
print("Slice using '[:3]': {}".format(y))

# Using '4:' implies 4 -> end
y = x[4:]
print("Slice using '[4:]': {}".format(y))

# Using ':' implies start -> end
y = x[:]
print("Slice using '[:]': {}".format(y))
```

```
Slice using '[:3]': [0.35912728 0.68416347 0.26731711]
Slice using '[4:]': [0.85003759]
Slice using '[:]': [0.35912728 0.68416347 0.26731711 0.79101538 0.85003759]
```

Slicing can be applied to 2D arrays:

[36]:
```
B = np.array([[1.3, 0], [0, 2.0]])
print(B)

# Extract second row
r = B[1, :]
print(r)
```

```
[[1.3 0. ]
 [0.  2. ]]
[0. 2.]
```

There is more to array slicing syntax, for example it is possible to extract every 3rd entry. If you need to extract a sub-array, check first if you can do it using the compact array slicing syntax.

5 Exercises

Complete now the 07 Exercises notebook.