

Optional Lab: Python, NumPy and Vectorization

A brief introduction to some of the scientific computing used in this course. In particular the NumPy scientific computing package and its use with python.

Outline

- [1.1 Goals](#)
- [1.2 Useful References](#)
- [2 Python and NumPy](#)
- [3 Vectors](#)
- [3.1 Abstract](#)
- [3.2 NumPy Arrays](#)
- [3.3 Vector Creation](#)
- [3.4 Operations on Vectors](#)
- [4 Matrices](#)
- [4.1 Abstract](#)
- [4.2 NumPy Arrays](#)
- [4.3 Matrix Creation](#)
- [4.4 Operations on Matrices](#)

```
In [1]: 1 import numpy as np    # it is an unofficial standard to use np for nump
        2 import time
```

1.1 Goals

In this lab, you will:

- Review the features of NumPy and Python that are used in Course 1

1.2 Useful References

- NumPy Documentation including a basic introduction: [NumPy.org \(https://NumPy.org/doc/stable/\)](https://NumPy.org/doc/stable/)
- A challenging feature topic: [NumPy Broadcasting \(https://NumPy.org/doc/stable/user/basics.broadcasting.html\)](https://NumPy.org/doc/stable/user/basics.broadcasting.html)

2 Python and NumPy

Python is the programming language we will be using in this course. It has a set of numeric data types and arithmetic operations. NumPy is a library that extends the base capabilities

of python to add a richer data set including more numeric types, vectors, matrices, and many matrix functions. NumPy and python work together fairly seamlessly. Python arithmetic operators work on NumPy data types and many NumPy functions will accept python data types.

3 Vectors

3.1 Abstract

Vectors, as you will use them in this course, are ordered arrays of numbers. In notation, vectors are denoted with lower case bold letters such as \mathbf{x} . The elements of a vector are all the same type. A vector does not, for example, contain both characters and numbers. The

$$\begin{array}{l} \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \\ \text{code} \end{array} \qquad \begin{array}{l} \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ \text{lecture} \end{array}$$

number of elements in the array is often referred to as the *dimension* though mathematicians may prefer *rank*. The vector shown has a dimension of n . The elements of a vector can be referenced with an index. In math settings, indexes typically run from 1 to n . In computer science and these labs, indexing will typically run from 0 to $n-1$. In notation, elements of a vector, when referenced individually will indicate the index in a subscript, for example, the 0^{th} element, of the vector \mathbf{x} is x_0 . Note, the x is not bold in this case.

3.2 NumPy Arrays

NumPy's basic data structure is an indexable, n -dimensional *array* containing elements of the same type (`dtype`). Right away, you may notice we have overloaded the term 'dimension'. Above, it was the number of elements in the vector, here, dimension refers to the number of indexes of an array. A one-dimensional or 1-D array has one index. In Course 1, we will represent vectors as NumPy 1-D arrays.

- 1-D array, shape (n,): n elements indexed [0] through [n-1]

3.3 Vector Creation

Data creation routines in NumPy will generally have a first parameter which is the shape of the object. This can either be a single value for a 1-D result or a tuple (n,m,...) specifying the shape of the result. Below are examples of creating vectors using these routines.

```
In [2]: 1 # NumPy routines which allocate memory and fill arrays with value
2 import numpy as np
3 a = np.zeros(4); print(f"np.zeros(4) : a = {a}, a shape = (4,)")
4 a = np.zeros((4,)); print(f"np.zeros(4,) : a = {a}, a shape = (4,)")
5 a = np.random.random_sample(4); print(f"np.random.random_sample(4): a = {a}, a shape = (4,)")
```

np.zeros(4) : a = [0. 0. 0. 0.], a shape = (4,), a data type = float64
np.zeros(4,) : a = [0. 0. 0. 0.], a shape = (4,), a data type = float64
np.random.random_sample(4): a = [0.49909867 0.46251609 0.83306502 0.92714106], a shape = (4,), a data type = float64

```
In [3]: 1 Some data creation routines do not take a shape tuple:
```

File "C:\Users\thoma\AppData\Local\Temp\ipykernel_11052\55466448.py", line 1
Some data creation routines do not take a shape tuple:
^
SyntaxError: invalid syntax

```
In [4]: 1 # NumPy routines which allocate memory and fill arrays with value but do not take a shape tuple
2 import numpy as np
3 a = np.arange(4.); print(f"np.arange(4.): a = {a}, a shape = (4,)")
4 a = np.random.rand(4); print(f"np.random.rand(4): a = {a}, a shape = (4,)")
```

np.arange(4.): a = [0. 1. 2. 3.], a shape = (4,), a data type = float64
np.random.rand(4): a = [0.94024719 0.16793347 0.80448453 0.01993634], a shape = (4,), a data type = float64

values can be specified manually as well.

```
In [5]: 1 # NumPy routines which allocate memory and fill with user specified values
2 import numpy as np
3 a = np.array([5,4,3,2]); print(f"np.array([5,4,3,2]): a = {a}, a shape = (4,)")
4 a = np.array([5.,4,3,2]); print(f"np.array([5.,4,3,2]): a = {a}, a shape = (4,)")
```

np.array([5,4,3,2]): a = [5 4 3 2], a shape = (4,), a data type = int32
np.array([5.,4,3,2]): a = [5. 4. 3. 2.], a shape = (4,), a data type = float64

These have all created a one-dimensional vector `a` with four elements. `a.shape` returns the dimensions. Here we see `a.shape = (4,)` indicating a 1-d array with 4 elements.

3.4 Operations on Vectors

Let's explore some operations using vectors.

3.4.1 Indexing

Elements of vectors can be accessed via indexing and slicing. NumPy provides a very complete set of indexing and slicing capabilities. We will explore only the basics needed for

the course here. Reference [Slicing and Indexing \(https://NumPy.org/doc/stable/reference/arrays.indexing.html\)](https://NumPy.org/doc/stable/reference/arrays.indexing.html) for more details.

Indexing means referring to *an element* of an array by its position within the array.

Slicing means getting a *subset* of elements from an array based on their indices.

NumPy starts indexing at zero so the 3rd element of an vector **a** is `a[2]` .

```
In [6]: 1 #vector indexing operations on 1-D vectors
        2
        3 a = np.arange(10)
        4 print(a)
        5
        6 #access an element
        7 print(f"a[2].shape: {a[2].shape} a[2] = {a[2]}, Accessing an element r
        8
        9 # access the last element, negative indexes count from the end
       10 print(f"a[-1] = {a[-1]}")
       11
       12 #indexs must be within the range of the vector or they will produce and
       13 try:
       14     c = a[10]
       15 except Exception as e:
       16     print("The error message you'll see is:")
       17     print(e)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
a[2].shape: () a[2] = 2, Accessing an element returns a scalar
```

```
a[-1] = 9
```

```
The error message you'll see is:
```

```
index 10 is out of bounds for axis 0 with size 10
```

3.4.2 Slicing

Slicing creates an array of indices using a set of three values (`start:stop:step`). A subset of values is also valid. Its use is best explained by example:

```
In [7]: 1 #vector slicing operations
2 a = np.arange(10)
3 print(f"a          = {a}")
4
5 #access 5 consecutive elements (start:stop:step)
6 c = a[2:7:1];    print("a[2:7:1] = ", c)
7
8 # access 3 elements separated by two
9 c = a[2:7:2];    print("a[2:7:2] = ", c)
10
11 # access all elements index 3 and above
12 c = a[3:];      print("a[3:]    = ", c)
13
14 # access all elements below index 3
15 c = a[:3];      print("a[:3]    = ", c)
16
17 # access all elements
18 c = a[:];       print("a[:]     = ", c)
19
a          = [0 1 2 3 4 5 6 7 8 9]
a[2:7:1] = [2 3 4 5 6]
a[2:7:2] = [2 4 6]
a[3:]    = [3 4 5 6 7 8 9]
a[:3]    = [0 1 2]
a[:]     = [0 1 2 3 4 5 6 7 8 9]
```

3.4.3 Single vector operations

There are a number of useful operations that involve operations on a single vector.

```
In [8]: 1 a = np.array([1,2,3,4])
2 print(f"a          : {a}")
3 # negate elements of a
4 b = -a
5 print(f"b = -a      : {b}")
6
7 # sum all elements of a, returns a scalar
8 b = np.sum(a)
9 print(f"b = np.sum(a) : {b}")
10
11 b = np.mean(a)
12 print(f"b = np.mean(a): {b}")
13
14 b = a**2
15 print(f"b = a**2     : {b}")
16
a          : [1 2 3 4]
b = -a      : [-1 -2 -3 -4]
b = np.sum(a) : 10
b = np.mean(a): 2.5
b = a**2     : [ 1  4  9 16]
```

3.4.4 Vector Vector element-wise operations

Most of the NumPy arithmetic, logical and comparison operations apply to vectors as well.

These operators work on an element-by-element basis. For example

$$c_i = a_i + b_i$$

```
In [9]: 1 a = np.array([ 1, 2, 3, 4])
        2 b = np.array([-1, -2, 3, 4])
        3 print(f"Binary operators work element wise: {a + b}")
```

Binary operators work element wise: [0 0 6 8]

Of course, for this to work correctly, the vectors must be of the same size:

```
In [10]: 1 #try a mismatched vector operation
        2 c = np.array([1, 2])
        3 try:
        4     d = a + c
        5 except Exception as e:
        6     print("The error message you'll see is:")
        7     print(e)
```

The error message you'll see is:
operands could not be broadcast together with shapes (4,) (2,)

3.4.5 Scalar Vector operations

Vectors can be 'scaled' by scalar values. A scalar value is just a number. The scalar multiplies all the elements of the vector.

```
In [11]: 1 a = np.array([1, 2, 3, 4])
        2
        3 # multiply a by a scalar
        4 b = 5 * a
        5 print(f"b = 5 * a : {b}")
```

b = 5 * a : [5 10 15 20]

3.4.6 Vector Vector dot product

The dot product is a mainstay of Linear Algebra and NumPy. This is an operation used extensively in this course and should be well understood. The dot product is shown below.

Vector Dot Product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = [\quad + \quad + \quad + \quad] =$$

The dot product multiplies the values in two vectors element-wise and then sums the result. Vector dot product requires the dimensions of the two vectors to be the same.

Let's implement our own version of the dot product below:

Using a for loop, implement a function which returns the dot product of two vectors. The function to return given inputs a and b :

$$x = \sum_{i=0}^{n-1} a_i b_i$$

Assume both a and b are the same shape.

```
In [12]: 1 def my_dot(a, b):
2         """
3         Compute the dot product of two vectors
4
5         Args:
6         a (ndarray (n,)): input vector
7         b (ndarray (n,)): input vector with same dimension as a
8
9         Returns:
10        x (scalar):
11        """
12        x=0
13        for i in range(a.shape[0]):
14            x = x + a[i] * b[i]
15        return x
```

```
In [13]: 1 # test 1-D
2 a = np.array([1, 2, 3, 4])
3 b = np.array([-1, 4, 3, 2])
4 print(f"my_dot(a, b) = {my_dot(a, b)}")
```

my_dot(a, b) = 24

Note, the dot product is expected to return a scalar value.

Let's try the same operations using `np.dot`.

In [14]:

```
1 # test 1-D
2 a = np.array([1, 2, 3, 4])
3 b = np.array([-1, 4, 3, 2])
4 c = np.dot(a, b)
5 print(f"NumPy 1-D np.dot(a, b) = {c}, np.dot(a, b).shape = {c.shape} ")
6 c = np.dot(b, a)
7 print(f"NumPy 1-D np.dot(b, a) = {c}, np.dot(a, b).shape = {c.shape} ")
8 print(c)
```

```
NumPy 1-D np.dot(a, b) = 24, np.dot(a, b).shape = ()
```

```
NumPy 1-D np.dot(b, a) = 24, np.dot(a, b).shape = ()
```

```
24
```

Above, you will note that the results for 1-D matched our implementation.

3.4.7 The Need for Speed: vector vs for loop

We utilized the NumPy library because it improves speed memory efficiency. Let's demonstrate:

In [15]:

```
1 import numpy as np
2 np.random.seed(1)
3 a = np.random.rand(10000000) # very large arrays
4 b = np.random.rand(10000000)
5
6 tic = time.time() # capture start time
7 c = np.dot(a, b)
8 toc = time.time() # capture end time
9
10 print(f"np.dot(a, b) = {c:.4f}")
11 print(f"Vectorized version duration: {1000*(toc-tic):.4f} ms ")
12
13 tic = time.time() # capture start time
14 c = my_dot(a,b)
15 toc = time.time() # capture end time
16
17 print(f"my_dot(a, b) = {c:.4f}")
18 print(f"loop version duration: {1000*(toc-tic):.4f} ms ")
19
20 del(a);del(b) #remove these big arrays from memory
```

```
np.dot(a, b) = 2501072.5817
```

```
Vectorized version duration: 15.6262 ms
```

```
my_dot(a, b) = 2501072.5817
```

```
loop version duration: 6648.1161 ms
```

So, vectorization provides a large speed up in this example. This is because NumPy makes better use of available data parallelism in the underlying hardware. GPU's and modern CPU's implement Single Instruction, Multiple Data (SIMD) pipelines allowing multiple operations to be issued in parallel. This is critical in Machine Learning where the data sets are often very large.

3.4.8 Vector Vector operations in Course 1

In Course 1, 2-D matrices are used to hold training data. Training data is m examples by n features creating an (m,n) array. Course 1 does not do operations directly on matrices but typically extracts an example as a vector and operates on that. Below you will review:

- data creation
- slicing and indexing

4.3 Matrix Creation

The same functions that created 1-D vectors will create 2-D or n-D arrays. Here are some examples

Below, the shape tuple is provided to achieve a 2-D result. Notice how NumPy uses brackets to denote each dimension. Notice further than NumPy, when printing, will print one row per line.

In [17]:

```
1 a = np.zeros((1, 5))
2 print(f"a shape = {a.shape}, a = {a}")
3
4 a = np.zeros((2, 1))
5 print(f"a shape = {a.shape}, a = {a}")
6
7 a = np.random.random_sample((1, 1))
8 print(f"a shape = {a.shape}, a = {a}")
```

```
a shape = (1, 5), a = [[0. 0. 0. 0. 0.]]
a shape = (2, 1), a = [[0.]
[0.]]
a shape = (1, 1), a = [[0.44236513]]
```

One can also manually specify data. Dimensions are specified with additional brackets matching the format in the printing above.

In [18]:

```
1 # NumPy routines which allocate memory and fill with user specified val
2 a = np.array([[5], [4], [3]]); print(f" a shape = {a.shape}, np.array
3 a = np.array([[5], # One can also
4 [4], # separate values
5 [3]]); #into separate rows
6 print(f" a shape = {a.shape}, np.array: a = {a}")
```

```
a shape = (3, 1), np.array: a = [[5]
[4]
[3]]
a shape = (3, 1), np.array: a = [[5]
[4]
[3]]
```

4.4 Operations on Matrices

Let's explore some operations using matrices.

4.4.1 Indexing

Matrices include a second index. The two indexes describe [row, column]. Access can either return an element or a row/column. See below:

```
In [19]: 1 #vector indexing operations on matrices
2 a = np.arange(6).reshape(-1, 2) #reshape is a convenient way to creat
3 print(f"a.shape: {a.shape}, \na= {a}")
4
5 #access an element
6 print(f"\na[2,0].shape: {a[2, 0].shape}, a[2,0] = {a[2, 0]}, type
7
8 #access a row
9 print(f"a[2].shape: {a[2].shape}, a[2] = {a[2]}, type(a[2]) = {ty
10
```

```
a.shape: (3, 2),
a= [[0 1]
     [2 3]
     [4 5]]
```

```
a[2,0].shape: (), a[2,0] = 4, type(a[2,0]) = <class 'numpy.int32'> A
ccessing an element returns a scalar
```

```
a[2].shape: (2,), a[2] = [4 5], type(a[2]) = <class 'numpy.ndarray'>
```

It is worth drawing attention to the last example. Accessing a matrix by just specifying the row will return a *1-D vector*.

Reshape

The previous example used [reshape \(https://numpy.org/doc/stable/reference/generated/numpy.reshape.html\)](https://numpy.org/doc/stable/reference/generated/numpy.reshape.html) to shape the array.

```
a = np.arange(6).reshape(-1, 2)
```

This line of code first created a *1-D Vector* of six elements. It then reshaped that vector into a *2-D array* using the reshape command. This could have been written:

```
a = np.arange(6).reshape(3, 2)
```

To arrive at the same 3 row, 2 column array. The -1 argument tells the routine to compute the number of rows given the size of the array and the number of columns.

4.4.2 Slicing

Slicing creates an array of indices using a set of three values (`start:stop:step`). A subset of values is also valid. Its use is best explained by example:

In [20]:

```
1 #vector 2-D slicing operations
2 a = np.arange(20).reshape(-1, 10)
3 print(f"a = \n{a}")
4
5 #access 5 consecutive elements (start:stop:step)
6 print("a[0, 2:7:1] = ", a[0, 2:7:1], ", a[0, 2:7:1].shape =", a[0, 2:7:1].shape)
7
8 #access 5 consecutive elements (start:stop:step) in two rows
9 print("a[:, 2:7:1] = \n", a[:, 2:7:1], ", a[:, 2:7:1].shape =", a[:, 2:7:1].shape)
10
11 # access all elements
12 print("a[:, :] = \n", a[:, :], ", a[:, :].shape =", a[:, :].shape)
13
14 # access all elements in one row (very common usage)
15 print("a[1, :] = ", a[1, :], ", a[1, :].shape =", a[1, :].shape, "a 1-D array")
16 # same as
17 print("a[1] = ", a[1], ", a[1].shape =", a[1].shape, "a 1-D array")
```

```
a =
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
a[0, 2:7:1] = [2 3 4 5 6] , a[0, 2:7:1].shape = (5,) a 1-D array
a[:, 2:7:1] =
[[ 2  3  4  5  6]
 [12 13 14 15 16]] , a[:, 2:7:1].shape = (2, 5) a 2-D array
a[:, :] =
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]] , a[:, :].shape = (2, 10)
a[1, :] = [10 11 12 13 14 15 16 17 18 19] , a[1, :].shape = (10,) a 1-D array
a[1] = [10 11 12 13 14 15 16 17 18 19] , a[1].shape = (10,) a 1-D array
```

Congratulations!

In this lab you mastered the features of Python and NumPy that are needed for Course 1.

In []: