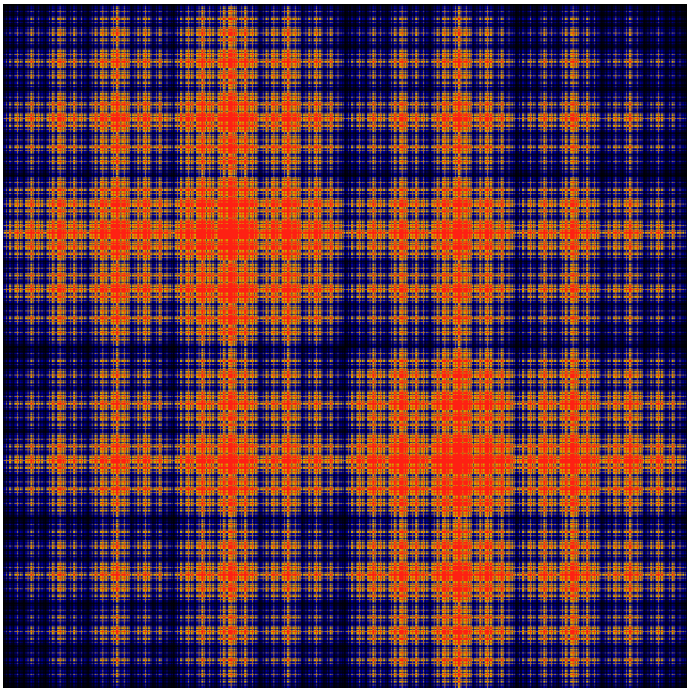
Applied Mathematics
and
Computing

Volume I



List of Contributors

J. Humpherys
Brigham Young University

J. Webb
Brigham Young University

R. Murray
Brigham Young University

J. West
University of Michigan

R. Grout
Brigham Young University

K. Finlinson
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Dr. J. Humpherys.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

`https://github.com/ayr0/numerical_computing`

as the original source of this work.
To view a copy of the Creative Commons Attribution 3.0 License, visit

`http://creativecommons.org/licenses/by/3.0/us/`

or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Lab 3

Essentials: NumPy

Why Arrays?

Problem 1 Let's begin with a simple demonstration of why arrays are important for numerical computation. Why use arrays when Python already has decently efficient list object? In this demonstration, we will try squaring a matrix. The matrix will be represented as a two dimensional list (i.e. a list of lists).

Write a function that will accept two matrices (two dimensional list), A and B, and return AB following the rules of matrix multiplication.

```
k = 10
a = [range(i, i+k) for i in range(0, k**2, k)]
```

Time how long your function takes to square matrices for increasing values of `k`.

Now import NumPy and create a NumPy array, b as shown below. We demonstrate how to square NumPy arrays as matrices below. `b*b` does not square the array, but rather multiplies b with itself element-wise. To get matrix multiplication for NumPy arrays, you must use `np.dot`

```
import numpy as np
b = np.array([range(i, i+k) for i in range(0, k**2, k)])
np.dot(b, b)
```

Time how long NumPy takes to square arrays for increasing sizes of `k`.

What do you notice about the time needed to square a two dimensional list vs. a two dimensional NumPy array?

NumPy

NumPy is a fundamental package for scientific computing with Python. It provides an efficient n-dimensional array object for fast computations. This lab will focus on

how to use these powerful objects. NumPy is commonly imported as shown below.

```
: import numpy as np
```

Before beginning this lab, it will be useful to understand certain concepts and terms used to describe NumPy arrays.

N-Dimensions

One, two, and three dimensional arrays are easy to visualize. But how do we visualize a four, ten, or fifteen dimensional array? NumPy arrays are best thought of as arrays within arrays. A one dimensional array consists of only elements. A two dimensional array is really just an array containing arrays which contain elements. Extending this metaphor, a three dimensional array is an array of arrays of arrays. Can you guess what a five dimensional array is? Let's define a three dimensional array.

```
: arr = np.random.randint(50, size=(5, 4, 3))
```

Arrays have several attributes. We use *shape* and *size* to describe the how big an array is. *Shape* tells how how many dimensions an array has and how big each dimension is. *Size* gives the total number of elements in the array.

```
: arr.shape
(5, 4, 3)
: arr.size
60
```

If we want to know how much memory an array takes to store, we can use `arr.nbytes`. The number of bytes is dependent on the data type (*dtype*) of the array. The data types that NumPy uses are different from Python data types. An integer in NumPy is not the same as an integer Python. Remembering this is vital. NumPy uses machine data types to speed up calculations. However, these datatypes are susceptible to a problem called *overflow*. A 64 bit integer has enough bits to represent integers between $-9,223,372,036,854,775,808$ and $9,223,372,036,854,775,807$. If we have an array with $-9,223,372,036,854,775,808$ and we decide to subtract 1, the integer wraps around and becomes $9,223,372,036,854,775,807$!

```
: arr.dtype
dtype('int64')
: arr.nbytes
480
```

Each element of the array has a unique address that describes it location. Indexing always starts at 0. Also, like Python lists, negative indices are valid and count from the tail of the array. We will discuss indexing in detail later in this lab.

```
: arr[0, 0, 0] # returns the first element of arr
: arr[-1, -1, -1] #returns the last element of arr
```

Creating Arrays

NumPy has several methods for creating and initializing arrays. When creating an array, we can optionally specify the data type that is stored in the array. NumPy arrays only store elements of the same data type, however, that data type can be any arbitrary object. The array order dictates how the array is laid out in memory. There is C order and Fortran order. C ordered arrays are also known as row-major arrays. This means that the fastest changing index correspond to the rows of the array. Fortran ordered arrays are column-major. Let's look at a few of the ways we can create arrays in NumPy.

- `np.array`: Makes an array from a Python list or tuple.
- `np.empty`: Allocates an array of a specific size without initializing the elements.
- `np.ones`: Allocates and array and initializes each element to 1.
- `np.zeros`: Allocates and array and initializes each element to 0.
- `np.identity`: Allocates a 2D array array with the main diagonal initialized to 1 and zeros everywhere else.

Indexing Arrays

Array Views and Copies

Before we begin accessing arrays, it is important to understand that NumPy has two ways of returning an array. Slice operations always return a *view* and fancy indexing always returns a *copy*. Understand that even though they may look the same, views and copies completely different.

Views are special arrays that reference other arrays. Changing elements in a view changes the array it references. Below we demonstrate the behavior of a view. Notice that `c` looks like a copy of `b`, but it is, in fact, not at all.

```
: b = np.arange(25).reshape(5,5); b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
: c = b[:]; c #looks like c is a copy of b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
: id(c) == id(b) #We have unique objects
False
: c[2] = 500; c
array([[  0,   1,   2,   3,   4],
       [  5,   6,   7,   8,   9],
       [500, 500, 500, 500, 500],
       [ 15,  16,  17,  18,  19],
```

```
       [ 20,  21,  22,  23,  24]])
: b #changing c also changed b!
array([[  0,   1,   2,   3,   4],
       [  5,   6,   7,   8,   9],
       [500, 500, 500, 500, 500],
       [ 15,  16,  17,  18,  19],
       [ 20,  21,  22,  23,  24]])
```

The reason that changing the array `c` also changed the array `b` is because `c` and `b` share the same memory, even though they are different Python objects. Views reduce the overhead of making copies of arrays and are useful when we want to change certain parts of the array.

A copy of an array is a separate array that is allocated separately.

```
: b = np.arange(25).reshape(5, 5); b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
: c = b.copy()
: id(c) == id(b) #we still have separate objects
False
: c[2] = 500
array([[  0,   1,   2,   3,   4],
       [  5,   6,   7,   8,   9],
       [500, 500, 500, 500, 500],
       [ 15,  16,  17,  18,  19],
       [ 20,  21,  22,  23,  24]])
: b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

Changing the data in a copy of an array, doesn't change the data in the original array. The two arrays address different locations in memory.

Slices

Each element of an array has a unique address that we can use to retrieve that element. Indexing NumPy arrays is syntatically the same as indexing Python lists. We will demonstrate on a random 2D array. Remember that slicing arrays always return views of the array. In this case, the indexing object is a Python tuple.

```
: arr = np.arange(25).reshape(5,5); arr
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
: arr[0, 0] #access the first element
0
: arr[-1, -1] #access the last element
```

```
24
: arr[0] #access the first row
array([0, 1, 2, 3, 4])
```

We can access ranges of elements using Python lists. NumPy, though, has a much faster, more concise way to select ranges using the `arr[start:stop:step]` notation.

```
: arr[::2] #get every other row.  equivalent to arr[range(0, len(arr),
     2)]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
: arr[::2, ::2] #get every other row and every other column
array([[ 0,  2,  4],
       [10, 12, 14],
       [20, 22, 24]])
: arr[3:, 3:] #extract lower right 2x2 subarray
array([[18, 19],
       [23, 24]])
: arr[:, 1] #extract second column
array([ 1,  6, 11, 16, 21])
```

Fancy Indexing

When the indexing object is an object other than a tuple, NumPy behaves slightly different. One difference is that fancy indexes always return a copy of an array instead of a view. There are two types of fancy indexes: boolean and integer. Boolean indexing returns an array of `True` or `False` values depending on some evaluating condition.

```
: bmask = (arr > 15) & (arr < 23)
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False,  True,  True,  True,  True],
       [ True,  True,  True,  False,  False]], dtype=bool)
: arr[bmask]
array([16, 17, 18, 19, 20, 21, 22])
: arr[(arr > 15) & (arr < 23)] #this is the shortened form
array([16, 17, 18, 19, 20, 21, 22])
: arr[~bmask] #invert the mask
```

```
: arr[(0, 2, 4), (0, 2, 4)] #grab every other element of diagonal
array([ 0, 12, 24])
: arr[range(0, 5, 2), range(0, 5, 2)] #same as above, but with ranges
array([ 0, 12, 24])
: arr[:, [0, -1]] #grab first and last column
array([[ 0,  4],
       [ 5,  9],
       [10, 14],
       [15, 19],
       [20, 24]])
```

Array Broadcasting

Array broadcasting allows NumPy to effectively work with arrays with sizes that don't match exactly. There are four basic rules to determine the behavior of broadcasted arrays

1. All input arrays of lesser dimension than the input array with largest dimension have 1's prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has a value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension.

To broadcast arrays, at least one of the following must be true.

1. All input arrays have exactly the same shape.
2. All input arrays are of the same dimension and the length of corresponding dimensions match or is 1.
3. All input arrays of fewer dimension can have 1 prepended to their shapes to satisfy the second criteria.

Problem 2 Explore array broadcasting. Create example for each of the three cases where arrays are broadcasted.

Saving Arrays

Sometimes it is desirable to save an array to a file to be used for later. NumPy provides several easy to use methods for saving and loading array data to files.

`np.save(file, arr)`	Save an array to a binary file
`np.savez(file, *arrs)`	Save multiple arrays to a binary file
`np.savetxt(file, arr)`	Save an array to a text file

`np.load(file)`	Load and return an array from a binary file
`np.loadtxt(file)`	Load and return an array from text file

Let's practice saving an array to a file and loading it again. Note that when saving an array, NumPy automatically appends the extension `.npy` if it does not already exist.

```
a = np.arange(30)
np.save('test_arr', a)
new_a = np.load('test_arr.npy')
np.savez('test_multi', a=a, new_a=new_a)
arrs = np.load('test_multi.npz')
```

The variable `arrs` points to a dictionary object with the keys `a` and `new_a` which reference the arrays that have been saved.