Lab 2

NumPy and SciPy

Lab Objective: *Create and manipulate NumPy arrays and learn features available in NumPy and SciPy.*

Introduction

NumPy and SciPy[1] are the two Python libraries most used for scientific computing. NumPy is a package for manipulating vectors and arrays, and SciPy is a higher-level library built on NumPy. The basic object in NumPy is the *array*, which is conceptually similar to a matrix. However, unlike a matrix, which has two dimensions, a NumPy `array` can have arbitrarily many dimensions. NumPy is optimized for fast array computations.

The convention is to import NumPy as follows.

```
>>> import numpy as np
```

Learning NumPy

The strategies discussed in the section "Learning Python" of Lab 1 will also help you learn NumPy and SciPy. The following online resources are specific to SciPy.

- Official SciPy Documentation (`http://docs.scipy.org/doc/`)
- Sections 1.3 and 1.5 of the SciPy Lecture Notes (`http://scipy-lectures.github.io/`)

The remainder of this lab is a brief summary of the tools available in NumPy and SciPy, beginning with NumPy arrays.

[1]SciPy is also the name of a Python coding environment that includes the NumPy and SciPy libraries, as well as IPython, matplotlib, and other tools.

Arrays

Conceptually, a 1-dimensional array (called a 1-D array) is just a list of numbers. An n-dimensional array (or n-D array) is an array of $(n-1)$-dimensional arrays. Thus, any 2-D array is conceptually a matrix, and a 3-D array is a list of matrices, which can be visualized as a cube of numbers. Each dimension is called an *axis*. When a 2-D array is printed to the screen, the 0-axis indexes the rows and the 1-axis indexes the columns.

The NumPy array class is called `ndarray`. The simplest way to create an `ndarray` is to define it explicitly using nested lists.

```
# Create a 1-D array
>>> np.array([0, 3, 8, 6, 3.14])
array([0, 3, 8, 6, 3.14])

# Create a 2-D array
>>> ex1 = np.array([[1, 1, 2], [3, 3, 4]])
>>> ex1
array([[1, 1, 2],
       [3, 3, 4]])
```

You can view the length of each dimension with the `shape` command, and change the shape of an array with the `np.reshape()` function. The number of arguments passed to `reshape` tells NumPy the dimension of the new array, and the arguments specify the length of each dimension. An argument of `-1` tells NumPy to make that dimension as long a necessary.

```
# The 0-axis of ex1 has length 2
>>> ex1.shape
(2, 3)
>>> ex1.reshape(3, 2)
array([[1, 1],
       [2, 3],
       [3, 4]])
>>> ex1.reshape(-1)
array([1, 1, 2, 3, 3, 4])
```

Array objects also support the usual binary operators, including addition `+` and componentwise multiplication `*`.

Why Use Arrays?

NumPy arrays are drastically more efficient than nested Python lists for large computations. In this section we will compare matrix multiplication in Python and NumPy.

Problem 1. A matrix in NumPy is just a 2-D array. How can you multiply two 2-D NumPy arrays as matrices? Hint: Use the strategies outlined in the section "Learning Numpy".

Data Structure	k	Time (s)
Python List	10×10	0.0002758503
	100×100	0.1336028576
	1000×1000	200.4009799957
NumPy Array	10×10	0.0000109673
	100×100	0.0009210110
	1000×1000	2.1682999134

Table 2.1: Time for one computer to square a $k \times k$ matrix in Python and NumPy.

After doing the previous problem, you should know how to implement matrix multiplication in NumPy. On the other hand, a matrix in Python can be implemented as a list of lists. The following function will multiply two such matrices.

```
def arr_mult(A,B):
    new = []
    # Iterate over the rows of A.
    for i in range(len(A)):
        # Create a new row to insert into the product.
        newrow = []
        # Iterate over the columns of B.
        # len(B[0]) returns the length of the first row
        # (the number of columns).
        for j in range(len(B[0])):
            # Initialize an empty total.
            tot = 0
            # Multiply the elements of the row of A with
            # the column of B and sum the products.
            for k in range(len(B)):
                tot += A[i][k] * B[k][j]
            # Insert the value into the new row of the product.
            newrow.append(tot)
        # Insert the new row into the product.
        new.append(newrow)
    return new
```

arr_mult.py

Table 2.1 documents how long[2] one computer took to square a $k \times k$ matrix in both Python (using the function `arr_mult`) and Numpy (using the method you found in Problem 1) for various values of k. As you can see, NumPy is much faster. One reason for this is that algorithms in NumPy are usually implemented in C or Fortran.

Data Types

Unlike Python containers, a NumPy array requires all its elements to have the same data type. The data types used by NumPy arrays are machine-native and avoid the overhead of Python objects, meaning that they are faster to compute with. A

[2]You can replicate this experiment yourself. In IPython, you can find the execution time of a line of code by prefacing it with `%timeit`. If you aren't using IPython, you will need to use the timeit function documented here: `https://docs.python.org/2/library/timeit.html`.

Data type	Description
`bool`	Boolean
`int8`	8-bit integer
`int16`	16-bit integer
`int32`	32-bit integer
`int64`	64-bit integer
`int`	Platform integer (depends on platform)
`uint8`	Unsigned 8-bit integer
`uint16`	Unsigned 16-bit integer
`uint32`	Unsigned 32-bit integer
`uint64`	Unsigned 64-bit integer
`float16`	Half precision float
`float32`	Single precision float
`float64`	Double precision float (also `float`)
`complex64`	Complex number represented by two single precision floats
`complex128`	Complex number represented by two double precision floats (also `complex`)

Table 2.2: Native numerical data types available in NumPy.

NumPy `int` and a Python `int` are not the same; the former has been optimized to speed up numerical computations. Datatypes supported by NumPy are shown in Table 2.2.

Here are some examples of how to manipulate data types in NumPy.

```
# Access the data type of an array
>>> ex2 = np.array(range(5))
>>> ex2.dtype
dtype('int64')

# Specify the data type of an array
>>> ex3 = np.array(range(5), dtype=np.float)
>>> ex3.dtype
dtype('float64')
```

Creating Arrays

In addition to `np.array()`, NumPy provides efficient ways to create special kinds of arrays. The function `np.arange([start], stop, [step])` is similar to the Python function `range()`.

```
>>> np.arange(10, 20, 2)
array([10, 12, 14, 16, 18])
```

Use `np.linspace(start, stop, num=50)` to create an array of `num` numbers evenly spaced in the interval from `start` to `stop`.

```
>>> np.linspace(0, 32, 4)
array([  0.        ,  10.66666667,  21.33333333,  32.        ])
```

Function	Description
`diag`	Extract a diagonal or construct a diagonal array.
`empty`	Return a new array of given shape and type, without initializing entries.
`empty_like`	Return a new array with the same shape and type as a given array.
`eye`	Return a 2-D array with ones on the diagonal and zeros elsewhere.
`identity`	Return the identity array.
`meshgrid`	Return coordinate matrices from two coordinate vectors.
`ones`	Return a new array of given shape and type, filled with ones.
`ones_like`	Returns an array of ones with the same shape and type as a given array.
`zeros`	Return a new array of given shape and type, filled with zeros.
`zeros_like`	Return an array of zeros with the same shape and type as a given array.

Table 2.3: Some functions for creating arrays in NumPy.

We can even create arrays of random values chosen from probability distributions. These probability distributions are stored in the submodule `np.random`.

```
>>> np.random.rand(5) # uniformly distributed values in [0, 1)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])
```

Some other commonly used functions are `np.random.normal`, which samples from the normal distribution, and `np.random.randint`, which randomly selects integers from a range.

There are many functions for creating arrays besides these, some of which are described in Table 2.3. See `http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html` for more details.

Indexing and Slicing

Indexing for a 1-D NumPy array works exactly like indexing for a Python list. To access a single entry of a multi-dimenional array, say a 3-D array, you should use the syntax `f[i, j, k]`. While the syntax `f[i][j][k]` will also work, it is significantly slower because each bracket returns an array slice. Similarly, slicing an array works just like slicing a list, but with more dimensions.

```
>>> ex4 = np.arange(25).reshape((5,5))
>>> ex4
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]])
>>> ex4[4, -2]
23

# Extract the lower right 2x2 subarray.
>>> ex4[3:, 3:]
array([[18, 19],
       [23, 24]])

# Extract the second column. The returned array is 1-D.
>>> ex4[:, 1]
```

```
array([ 1,  6, 11, 16, 21])

# Reverse the order of the columns.
>>> ex4[:, ::-1]
array([[ 4,  3,  2,  1,  0],
       [ 9,  8,  7,  6,  5],
       [14, 13, 12, 11, 10],
       [19, 18, 17, 16, 15],
       [24, 23, 22, 21, 20]])
```

Fancy indexing is a second way to access elements of an array. There are two types of fancy indexing: boolean and integer. Boolean indexing uses an array of `True` or `False` values to determine which elements of the array to take. You can create true-false valued arrays by using logical operations on arrays. See `https://scipy-lectures.github.io/intro/numpy/operations.html#other-operations` for examples and `http://docs.scipy.org/doc/numpy/reference/routines.logic.html` for a full list of such operations.

```
# Logic operations on arrays result in true-false valued arrays.
>>> ex4mask = np.logical_and(ex4<23, ex4>15)
>>> ex4mask
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)

>>> ex4[ex4mask]
array([16, 17, 18, 19, 20, 21, 22])
```

Integer indexing uses Python lists to determine what array values to access.

```
# Return an array of elements ex4[0,0], ex4[0, 0], ex4[2,3], and ex4[4,1].
>>> ex4[[0, 0, 2, 4], [0, 0, 3, 1]]
array([ 0,  0, 13, 21])
# Take the first (0) and last (-1) columns.
>>> ex4[:, [0, -1]]
array([[ 0,  4], [ 5,  9], [10, 14], [15, 19], [20, 24]])
```

Fancy indexing can be used for assignment. For example, we can set all values of an array that are less than `10` to `0` in the following way.

```
>>> ex4[ex4<10] = 0
>>> ex4
array([[ 0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

Array Views and Copies

NumPy has two ways of returning an array. Slice operations and indexing always return a *view* and fancy indexing always returns a *copy*. Understand that even though they may look the same, views and copies are different.

A view of an array is a distinct object from the original array in Python, but it references the same place in memory. Thus, when you change elements in a view, you also change the array it references.

```
>>> ex5 = np.arange(5)

# Slicing produces a view of k.
>>> view_ex5 = ex5[:]
>>> view_ex5

# Check that m and k are distinct objects in Python.
>>> id(view_ex5) == id(ex5)
False

# Change the third element of view_ex5 to 500
# Changing view_ex5 also changes ex5.
>>> view_ex5[2] = 500
>>> view_ex5
array([  0,   1, 500,   3,   4])
>>> ex5
array([  0,   1, 500,   3,   4])
```

A copy of an array is a separate array with its own memory. Thus, when you change a copy of an array, you do not affect the original array. Because copying an array uses more memory and also more time, it should only be done when necessary. An array can be copied using the `np.copy()` function (also available as a method of the array object).

```
>>> copy_ex5 = np.copy(ex5)

# Check that j and n are distinct objects in Python.
>>> id(copy_ex5) == id(ex5)
False

# Change the third element of copy_ex5 to 1000
# Changing copy_ex5 does not affect ex5.
>>> copy_ex5[2] = 500
>>> copy_ex5
array([  0,   1, 1000,   3,   4])
>>> ex5
array([  0,   1, 500,   3,   4])
```

Whenever possible, the function `np.reshape()` returns a view. See the documentation for more information.

More Methods of NumPy Arrays

Some of the more common methods of NumPy arrays are described in Table 2.4. A more comprehensive list can be found at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html`.

Many of these methods have the option to operate *along an axis*. When called in this way on an n-D array, these methods return an $(n-1)$-D array (the specified axis is collapsed in the evaluation process).

Function	Description
`all`	returns True if all elements evaluate to True
`any`	returns True if any elements evaluate to True
`argmax`	indices of maximum value(s)
`argmin`	indices of minimum value(s)
`argsort`	indices that would sort the array
`astype`	casts a copy of an array to a different data type
`clip`	restrict values in an array to fit within a given range
`conj`	return the complex conjugate of the array
`copy`	return a copy of the array
`diagonal`	return a given diagonal of the array
`dot`	matrix multiplication
`max`	max element of the array
`mean`	average of the array
`min`	minimum element of the array
`vstack prod`	product of elements of the array
`ravel`	make a flattened version of an array, return a view if possible
`reshape`	return a view of the array with a changed shape
`round`	return a rounded version of the array
`sort`	sort the array in place
`std`	compute the standard deviation
`sum`	sum the elements of the array
`swapaxes`	return a view with the given axes swapped
`tolist`	return the array represented as a list or nested list
`trace`	return the sum of the elements along the main diagonal
`var`	return the variance of the array

Table 2.4: A few of the methods of NumPy arrays.

```
>>> ex6 = np.arange(9).reshape(3, 3)
>>> ex6
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Return the maximum value in the array
>>> ex6.max()
8

# Return the maximum values evaluated along the 0-axis
>>> ex6.max(axis=0)
array([6, 7, 8])

# Return the maximum values evaluated along the 1-axis
>>> ex6.max(axis=1)
array([2, 5, 8])
```

Problem 2. Write a function which accepts an integer n as input and does the following:

1. Creates an $n \times n$ array of `floats` randomly chosen from a normal distribution
2. Computes the mean of each row (use a built-in command)
3. Computes the variance of these means (use a built-in command).

As you increase n, what happens to the output of your function? This illustrates one version of the Law of Large Numbers, about which you will learn more later on.

Problem 3. One good application of array slicing is the Jacobi method for solving Laplace's equation, which is used to model steady-state heat flow on a square. This problem will help you implement the Jacobi method.

Make a function that accepts an array and a tolerance as input and does the following:

1. Makes a copy of the array.
2. Creates a variable to track the difference between the arrays. Initialize it as the tolerance parameter your function accepts.
3. While the difference is greater than or equal to the tolerance
   (a) Sets all points that are not on an edge of the new array equal to the average of their 4 immediate neighbors. Use the values from the old array for this computation. This should only take one line and should be based entirely on array slicing, NOT iterating through the array. (Hint: given a 2D array `A`, the slice `A[1:-1,1:-1]` references all non-edge entries, `A[:-2,1:-1]` references the upper neighbors, and `A[1:-1,2:]` references the right neighbors.)
   (b) Updates the difference to be the maximum of the absolute value of the new array minus the old one.
   (c) Copies the values from the new array into the old one (without creating a new array).

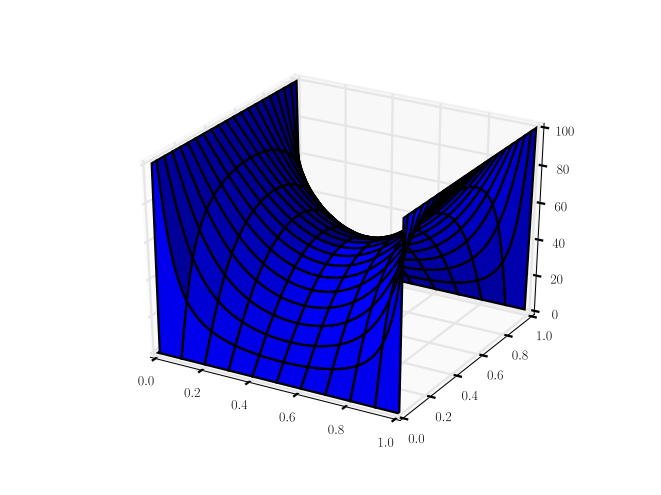
Now use the following code to generate a plot of your results.

```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
n = 100
tol = .0001
U = np.ones ((n, n))
```

```
U [:,0] = 100 # sets north boundary condition
U [:,-1] = 100 # sets south boundary condition
U [0] = 0 # sets west boundary condition
U [-1] = 0 # sets east boundary condition
# U has been changed in place (note that laplace is the name of
# the function in this case).
laplace(U, tol)
x = np.linspace(0, 1, n)
y = np.linspace(0, 1, n)
X, Y = np.meshgrid(x, y)
fig = plt.figure()
ax = fig.gca(projection = '3d')
ax.plot_surface (X, Y, U, rstride=5)
plt.show()
```

laplace_plot.py

It should resemble the following figure.



Iterating Through Arrays

Iterating through an array undoes most speed advantages of NumPy. You should avoid doing this whenever possible. You can often avoid iterating through arrays by using *array broadcasting* and *universal functions*, discussed in the next section.

It is occasionally valid to iterate through an array. The function `np.nditer()` will create an object that iterates through an array as quickly as possible.

NumPy and SciPy

We now introduce some additional features of NumPy and SciPy.

Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape. Two examples are addition and component-wise multiplication. Broadcasting is NumPy's way of extending such operations to accept some (not all) operands with different shapes. Broadcasting happens automatically whenever it is necessary.

To understand broadcasting, let us look at an example.

```
>>> A = np.ones(3);
>>> A
array([ 1.,  1.,  1.])
>>> B = np.vstack([1, 2, 3])
>>> B
array([[1],
       [2],
       [3]])
>>> A+B
array([[ 2.,  2.,  2.],
       [ 3.,  3.,  3.],
       [ 4.,  4.,  4.]])
```

We will now describe the algorithm used to obtain the result for `A+B` above. First, the shapes of `A` and `B` are lined up, starting at the far right, and 1's are prepended to the shorter tuple. So

```
A   (1-D array):     3
B   (2-D array): 3 x 1
```

becomes

```
A   (1-D array): 1 x 3
B   (2-D array): 3 x 1
```

For broadcasting to work, the dimensions must be compatible; that is, in a given axis, the lengths are equal, or one of the lengths is 1. Second, the arrays `A` and `B` are "stretched" one axis at a time until the lengths of their axes are the same. In each axis, if the lengths are different, the smaller array is copied along that axis (or "stretched"), until it is the size of the larger array. Conceptually, we are creating new 3×3 arrays A' and B' where

$$A' = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \text{and} \qquad B' = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}.$$

Finally, NumPy returns the sum $A' + B'$.

We emphasize that the "stretching" in this example is only conceptual, and no new array A' or B' is created. However, you should still be careful when broadcasting large arrays because you can fill the RAM on your computer, which can sometimes freeze the system. For a more detailed description of array broadcasting rules, see `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`.

Problem 4. Create a $100 \times 100 \times 3$ array of random integers taking values in the range $[0, 256]$. Such an array can represent an RGB image of 100×100 pixels, where each pixel is associated with an array of three integers indicating the amounts of red, green, and blue color present in that pixel. Use array broadcasting to multiply the red and green values by 0.5. (Such an operation would tone down the red and green colors and make the image appear more blue.)

Universal Functions

A universal function, or `ufunc`, operates on an array elementwise. It outputs an array of the same shape and datatype as the input array. Using a universal function is usually much faster than iterating through the array yourself.

Many scalar functions from the Python standard library have a universal analog that operates on arrays. For example, `math.sin()` operates on scalars, and `numpy.sin()` operates on arrays. If you have a simple operation that you want to perform elementwise on an array, you should see if SciPy has a universal function that will do it (it probably does). For a list of available `ufuncs`, see `http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs`.

Most universal functions also allow you to specify an output array, which must have the same shape as the input array. Doing so can reduce memory allocation.

```
>>> ex7 = np.arange(3, dtype=float)

# Take exp(ex7) and store the result in ex7.
>>> np.exp(ex7, out=ex7)
>>> ex7
array([ 1.        ,  2.71828183,  7.3890561 ])
```

Although universal functions also accept scalar inputs, they can be much slower than the corresponding standard library function. Thus, use standard library functions on scalars and universal functions on arrays.

Linear Algebra

Both NumPy and SciPy have a linear algebra library, but the SciPy library is larger. The SciPy linear algebra library is typically imported as follows.

```
from scipy import linalg as la
```

The linear algebra library contains several functions to construct special matrices, located in `linalg.special_matrices`. There are also functions that will invert matrices, find determinants and norms, solve linear systems and least squares problems, and find special matrix decompositions. You can read more about the linear algebra capabilities of SciPy in the documentation for the `linalg` module found at (`http://docs.scipy.org/doc/scipy/reference/linalg.html`).

Finally, the `scipy.linalg` library has a `matrix` class that is very similar to a 2-D NumPy array. The matrix class can be convenient when doing matrix operations. However, in such situations we still recommend using NumPy arrrays, which have many of the same features and are also compatible with all other SciPy operations.

Polynomials

The `np.poly1d` object represents a polynomial in NumPy. The constructor is called with the coefficients of the desired polynomial.

```
>>> poly_array = np.poly1d([3, 5, 1, 2, 0, 1])
>>> print poly_array
   5     4     3     2
3 x + 5 x + 1 x + 2 x + 1
```

The object `poly_array` represents the polynomial $3x^5+5x^4+x^3+2x^2+1$. NumPy provides many functions to operate on `poly1d` objects (see `http://docs.scipy.org/doc/numpy/reference/routines.polynomials.polynomial.html`).

Here is an example using the `poly1d` class. Recall that

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

The following function evaluates the n^{th} partial sum of this series at the value a.

```
1 from scipy.misc import factorial
2 def exp(a, n = 25):
      # Construct an array in reverse order from n to 0.
4     integers = np.arange(n, -1, -1)
      # Use broadcasting to compute coefficients
6     coefficients = 1. / factorial(integers)
      poly = np.poly1d(coefficients) # make polynomial object
8     return poly(a)
```

exp.py

The last two lines can be condensed by using the following command.

```
np.polyval(p, a)
```

Problem 5.

1. Use NumPy's polynomial objects to approximate the following series.

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!x^{2n+1}}{(2n+1)\,(n!)^2\,4^n}$$

   This series converges on $(-1, 1)$. Use your series approximation to approximate π. Hint: think of the powers of x that are not included in the series as having zero coefficients.

2. The lambert W function is the inverse of xe^x. Its Taylor series is below (note the index starts at 1).

$$W(x) = \sum_{n=1}^{\infty} \frac{(-n)^{n-1} x^n}{n!}$$

This series has a radius of convergence of $\frac{1}{e}$. Use the series to approximate a number x such that $xe^x = \frac{1}{4}$. Verify that your approximation is close.