- Zhuojun Chen
- A92108974

- COGS 118C - Assignment 0

This notebook has [15] points in total

The number of points for each question is denoted by []. Make sure you've answered all the questions and that the point total add up.

Lab 0 - Intro to Jupyter, Python, and numpy

... and Vectors, Arrays, Linear Algebra and Complex Numbers

In this lab, we will cover mathematical and programming concepts fundamental to neural signal processing. We will start with general Jupyter and python concepts, then move on to use numpy arrays/matrices.

Topics include:

- Jupyter notebook tips
- general python, including: types, functions, for loops, if/else statements
- numpy, array/matrix indexing
- multi-dimensional vectors, dot product
- complex numbers: rectangular vs. Euler form
- plotting
- how to find stuff you need: Googling, stackexchange, documentation

This notebook corresponds to material in the lecture slides for Lab1-math_review

---

Some Jupyter Tricks

Jupyter is an interactive interface that runs a Python "kernel" in the background, one of 3 possible kernels (other 2 being Julia and R). Each cell can be run (executed) separately, but they populate the same "name space", i.e., not independent. If you initialize and do some operations on a variable in one cell, and change it in another, that will affect the initial cell. It's similar to cells in a MATLAB script.

Basically, keep track of when and where you ran code, and before you submit anything final, **ALWAYS kill the notebook and re-run everything from the top** to make sure there weren't some loose variables defined somewhere else!

Some nifty shortcuts:

- Enter to enter edit mode (can enter text); ESC to enter command mode.
- In command mode:
  - A to open a new cell above, B to open one below, X to delete current cell.
  - M to change the cell to Markdown (fancy text, non-executable). Markdown is how all these instructions are written, and also where you will provide short-form responses in all the assignments. There are a lot more ways of formatting than plaintext, for example, **bold**.
- Shift-Enter to run a cell, which will execute the code, or set the Markdown text
- If you need to know what a function does, read its documentation string (docstring) by putting your cursor inside the brackets, and press shift+tab or shift+tab+tab to expand.

Press the little keyboard on the top for more shortcuts. Press the square or loop-arrow to kill and restart the session.

```
# [1] Q1: What is your name?
[1] Open a new cell at the VERY TOP of this notebook, make it a Markdown cell, and type, in a list:
- your name,
- your PID,
- and the course code, followed by " - Assignment 0"

and execute. It should look pretty nice.
```

Python Crash Course

I can't teach you all the python you need to know in 10 minutes, you'll have to struggle with it yourself. But these are some key concepts that make up most of the signal processing computations we will use.

COGS18 (https://cogs18.github.io/materials/00-Introduction) has all the stuff you need to know. You should've taken this course before, or an equivalent programming course. So the next section is a *light* review.

First, real basic stuff: types.

In [1]:
```
# This is a comment. Always comment your code!
# print() and type() are both native python functions
print(type(6)) # this is an integer
print(type(6.0)) # this is a float
print(type(True)) # this is a boolean
print(type('this is a string.')) # this is a string
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

Now, onto variables.

In [2]:
```
a = 5 # assign the value 5 to the variable a
b = 6.7 # similarly...

print(a)
print(b)
print(a==b)

a = b
print(a)
print(a==b)
```

```
5
6.7
False
6.7
True
```

Lastly, lists are python's native object for holding...well, lists of things.

In [3]:
```
# define a list
my_list = [1,2,3,4,5,6,7]
print(my_list)

# define a second list
my_other_list = [11,12,13,14,15,16,17]
print(my_other_list)

# now, add the two lists together
my_list + my_other_list
print(my_list+my_other_list)

print('Wait a minute...')
```

```
[1, 2, 3, 4, 5, 6, 7]
[11, 12, 13, 14, 15, 16, 17]
[1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16, 17]
Wait a minute...
```

Arrays with numpy

As you can see above, lists don't necessarily work the way we expect them to as Euclidean vectors. For one, adding two lists will combine them, not add numerically.

Due to this, and a slew of other reasons, we will be using `numpy`, the numerical python library.

In [4]:
```
# import our first library!
# import as simply creates a short form for you to refer to when you ca
# its module, and there are usually conventions.

import numpy as np
print(np)
```

```
<module 'numpy' from '/Users/zhuojun/anaconda3/lib/python3.6/site-package
s/numpy/__init__.py'>
```

Now, we can convert our lists into numpy arrays.

In [5]:
```
my_arr = np.array(my_list)
print(my_arr, type(my_arr))

my_arr2 = np.array(my_other_list)

# now let's try adding them (element-wise) again.
print('Adding:')
print(my_arr + my_arr2)

# we can also element-wise multiply them
print('Multiplying:')
print(my_arr*my_arr2)
```

```
[1 2 3 4 5 6 7] <class 'numpy.ndarray'>
Adding:
[12 14 16 18 20 22 24]
Multiplying:
[ 11  24  39  56  75  96 119]
```

[2] Q2: Basis Vectors in 2-dimension

[1] In the cell below, create two variables that represent two different vectors, the 2-dimensional standard bases i and j (https://en.wikipedia.org/wiki/Standard_basis).

[1] Then, perform vector addition using them by adding them together.

In [7]:
```
basis_x = np.array([1,0])
basis_y = np.array([0,1])

my_vec = basis_x + basis_y
print(my_vec) # print the result of their summation
```

```
[1 1]
```

Indexing, length, and summation of an array

When the vector is 2D, you know the array has length 2. But when we work with long signals later on, they will also be stored in these arrays, and we often need to know how many elements are in it for various purposes.

To access a particular element of an array, use square brackets. Python uses 0-indexing, meaning the first element is accessed with the index 0, instead of 1.

In [8]:
```
print(my_arr[0])
print(my_arr[-1])
```

```
1
7
```

In [9]:
```
# two ways of finding the length of an array

print(my_arr)
print(len(my_arr)) # len() is a native python function
print(my_arr.shape) # .shape is a property of numpy arrays only
```

```
[1 2 3 4 5 6 7]
7
(7,)
```

In [10]:
```
# two ways of finding the summation of an array
# note that this means adding together all the elements inside a single
print(sum(my_arr)) # len() is a native python function
print(my_arr.sum())# .sum() is a property of numpy arrays only
```

```
28
28
```

[2] Q3: Functions and for loops

[1] To brush up your python, create your own function that returns the summation of an array, **using only a for-loop**!

[1] Confirm that they return the same result as above.

In [11]:
```
# I've given you a hint for free: use the range() function to create an
# If you need to iterate through all the elements of an array, this has
print(range(6))
```

```
range(0, 6)
```

In [14]:
```
# Note that there are various ways to solve this problem, there's no co
# It's okay if your solution is different or more/less elegant, this is
# have the prerequisite exposure for coding algorithms
def my_sum(arr):
    #_FILL_IN_YOUR_CODE_HERE
    summation = 0
    for i in range(len(arr)):
        summation = summation + arr[i]
    return summation

print(my_sum(range(6)))
```

```
15
```

[2] Q4: Dot Product

[2] Consult the formula for the dot product below, create a function that performs the dot product (or inner product) between two arrays.

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

where N is the length (dimension) of the vectors. Note that the above uses zero-indexing, to be consistent with python.

In [19]:
```
def my_dotprod(arr1,arr2):
    # Hint: given what you already know, you only need a single line in
    result = my_sum(arr1 * arr2)
    return result

print(my_dotprod(my_arr, my_arr2))

# confirm that it gives same answer as the numpy function
print(np.dot(my_arr, my_arr2))

# Congratulations, you have just performed dot product in 7-dimensions.
```

```
420
420
```

[2] Q5: Dot Product of Basis

[1] What should the dot product of the two bases vectors you created above (i and j) be?

ANSWER: 0

[1] What is this property called?

ANSWER: Orthogonal Property of dot product.

---

[1] Confirm that this is the case in code below. You can use either your dot product function or numpy's.

In [ ]:
```
1
```

Complex Numbers

Complex numbers have a real and an imaginary component, which can be represented as a 2D (length-2) vector. The bases (axes), instead of x and y, are the real and imaginary number lines.

In python, we tack on a `j` to denote the imaginary component.

In [20]:
```
z = 1+1j
print(z)

# or you can construct a complex number like this:
w = complex(1,-1)
print(w)
```

```
(1+1j)
(1-1j)
```

Plotting vectors

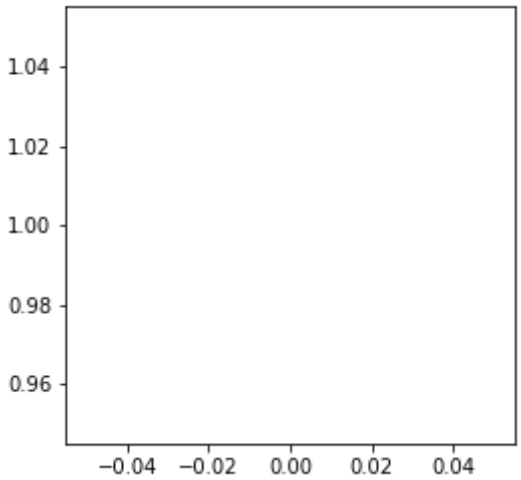
Now, we will import another library, for plotting things: matplotlib. More precisely, we're importing a specific module of the library, but don't worry about that, since we'll always be using the same line to import.

In [21]:
```
import matplotlib.pyplot as plt # do the import

# the next line is jupyter magic that always displays the plot, otherwi
%matplotlib inline
```

In [22]:
```
plt.figure(figsize=(4,4))
plt.plot(z)
```

```
/Users/zhuojun/anaconda3/lib/python3.6/site-packages/numpy/core/numeric.p
y:501: ComplexWarning: Casting complex values to real discards the imagin
ary part
  return array(a, dtype, copy=False, order=order)
```

Out[22]:
```
[<matplotlib.lines.Line2D at 0x119ba6da0>]
```



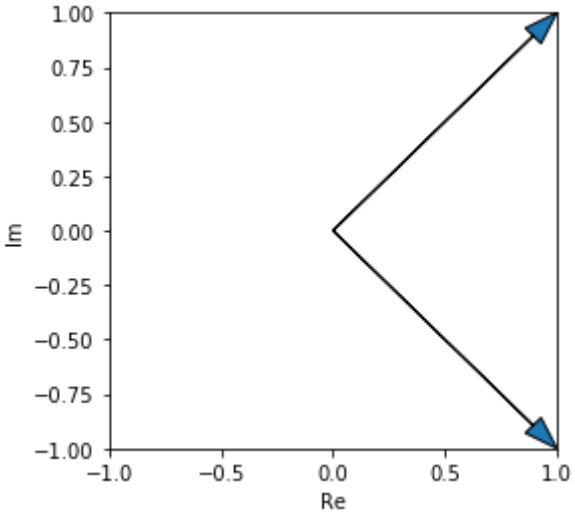
Well, nothing happened.

It's a little complicated, but as you can see above, matplotlib doesn't handle complex numbers in the intuitive way you'd imagine, so we have to specify the real and imaginary components separately.

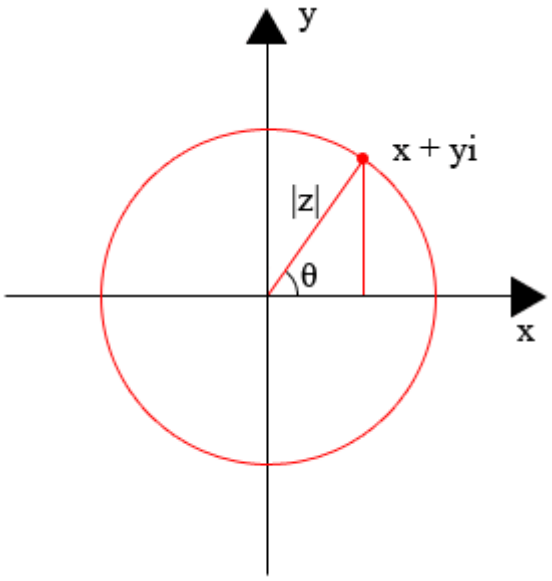
Also, we can use `plt.arrow()` to make the vector pretty.

In [23]:
```
# make a new figure and define its size
plt.figure(figsize=(4,4))

# draw the arrows
plt.arrow(0,0,z.real,z.imag, head_width=0.1, length_includes_head=True)
plt.arrow(0,0,w.real,w.imag, head_width=0.1, length_includes_head=True)

plt.xlim([-1,1]);plt.ylim([-1,1]) # set limits manually

plt.xlabel('Re');plt.ylabel('Im') # label the axes
```

Out[23]: `Text(0, 0.5, 'Im')`



[2] Q6: Complex Number in Polar Form



As defined above, the complex number z is in its rectangular form, which specifies its real and imaginary components. We can also represent it in its polar form. As the diagram shows above, that means specifying its magnitude and angle (or sometimes called phase).

[2] Note that the vector z is the hypotenuse of a right angled triangle with the real (x) and imaginary (y) components as its two sides. Express that in code below to find the magnitude and angle of z. Confirm that it's the same as the numpy results.

In [28]:
```
z_real = z.real
z_imag = z.imag

z_mag = np.sum(z_real**2 + z_imag**2)**0.5
z_ang = np.arctan(z_imag / z_real)

print(z_mag, z_ang)
print(np.abs(z), np.angle(z))
```

```
1.4142135623730951 0.7853981633974483
1.4142135623730951 0.7853981633974483
```

[3] Q7: Complex Number as Vectors

Conveniently, we can express the complex number as a length-2 array, and do vector computation with them as we did above.

[1] Looking at the plot where z and w are plotted as arrows, what is the angle (in radians or degrees) between those two vectors?

ANSWER: 90 degrees.

[1] What do you expect is the dot product of z and w? Why? Make an argument without explicitly computing it.

ANSWER: 0. Because w and z are orthogonal to each other, according to the orthogonal property, the result is 0.

[1] In the cell below, convert the complex number z and w into length-2 arrays, and confirm that the dot product is what you expected.

In [29]:
```
z_arr = np.array([1, 1])
w_arr = np.array([1,-1])
np.dot(z_arr,w_arr)
```

Out[29]: 0

End Survery

Please take a few minutes to fill out the following as it will help us to improve the following assignments & lectures.

Content:

What was one thing you learned from this lab & associated lectures?

ANSWER: I learned how to represent the complex number and how complex number works in Python.

What was one thing that you still found confusing after the lab, and need clarification?

ANSWER: I found I am good after doing the lab.

Style:

What was one thing you enjoyed about the formatting of this assignment (e.g., clarity, structure, guidance, etc.)?

Answer: The logic of this lab is pretty construtive and progressive, which makes the lab easy to follow and friendly to beginners.

What was one thing that you thought could use improvements on?

Answer: Have more visual explanation!

Thank you!