

Summary of PHYS-495 Topics

Michael Camilo

Montclair State University

Abstract

In our first meeting, Dr. Ghosh gave a basic overview of neutron stars, relativity, and gravitational waves. We spoke of the Hubble constant and the methods by which the age of the universe can be determined. In later meetings, we worked on reinforcing my knowledge of basic Linux, Vim, Tmux, and Git commands to get an understanding of the kind of workflow one should have when coding. From here, having already had a background in Python, we moved on to applications of NumPy. Basic array manipulation and indexing, as well as boolean masking and indexing, were reviewed and applied to different projects.

The first project was to code a program that would find all the prime numbers from 0 up to the inputted value. Dr. Ghosh introduced a version of the code that was in C to give me an idea of a good algorithm to use, and we soon translated it into Python. The difference in computing speed became very apparent when compared to each other. Python code is interpreted, C's is compiled. The C code ran many times faster than the Python code. Numpy largely resolves this issue, as even though it is a python package, it runs in C. It is a means of coding in the simple syntax Python offers, with the efficiency C is known for. The next objective was to eliminate any loops in the Python code using arrays. Working with arrays is far more efficient than with loops. In my first attempt, I was successful at eliminating one of the for loops in the program; later Dr. Ghosh showed how both could be eliminated. Afterward, the NumPy-based program and the C program were compared. While the NumPy code was far faster than the original Python code, it was still outmatched by the C code. Well-written C code will always run faster than NumPy code, but NumPy does the job well enough.

The second assignment was to code a program that, when fed frequency, amplitude, starting and ending times, attenuation function, and initial phase values, would produce a

sinusoidal wave. Technically three are produced: the noise, the signal, and the total combined data. Under normal circumstances, the Laser Interferometer Gravitational-Wave Observatory (LIGO) receives constant noise and detects gravitational waves embedded inside the noise. The program produces a data set where the noise and signal have the same resolution when combined.

The last assignment's objective is to use this newly created waveform function to test methods of detecting gravitational-wave signals in raw data. Since the waveform function provides the actual properties of the signal, the extracted signal found in the detection function can be compared, and clearly show the accuracy of the method used. Cross-correlation is the first method that will be used to identify the location and properties of the signal. Detecting, isolating, and analyzing extracted signals is analogous to the work LIGO does, on a far smaller scale of complexity shown here.

Summary of PHYS-495 Topics

Neutron Stars can result from the collapse of high mass stars. During the lifespan of high mass stars, hydrogen, helium, carbon, neon, oxygen, and silicon fuse in that order, resulting in an iron core. At this point iron cannot be fused together, and the balance of fusion energy and the force of gravity is lost. The star now collapses on itself and implodes; this event is called a supernovae. The core is compressed by the now unhindered gravity, which squeezes protons and electrons together to form neutrons, resulting in a pool of neutrons. This compression explains the relative size of neutron stars compared to their parent star. There are multiple models on the structure of neutron stars, as many believe the pressure gravity caused in the formation of the neutron star went further, and split neutrons open, suggesting neutron stars are made up of quarks. If this is true, we should seriously consider renaming them to quark stars.

While Newtonian physics is great at explaining things that aren't too relatively massive or nonmassive, when it comes to things like supermassive, or miniscule, objects and systems moving at high speeds, it fails. One of the big questions in Newtonian physics was the cause of the force of gravity. The Theory of General Relativity was a new way of looking at physics. It introduces space-time, the fabric of the universe. This isn't too hard to wrap your head around, as the three dimensional space we are accustomed to is still present, time is the only new factor. Matter distorts and warps spacetime. Warped spacetime causes parallel lines to become geodesics, which at high distortions should converge at a single point. What we call gravity is really just something following a geodesic.

Disturbances in space-time called gravitational waves are an especially exciting discovery. Prominent ones are formed by the collision of binary star systems; the star types being neutron stars and black holes. When these stars orbit one another, they lose energy in the form of

gravitational waves, and as such they go closer until they merge. The merging event creates powerful gravitational waves that can be measured millions of years later on Earth. For neutron stars, as they get closer to each other, they warp, causing a severe variance in the distribution of mass of the entire system, causing this growth in gravitational wave strength. Gravitational waves hold information about the stars that collided, and can even be used to calculate Hubble's constant.

The Hubble Constant describes the speed at which the universe is expanding, which allows the age and history of the universe to be extrapolated. Gravitational waves can be analyzed to find the distance from Earth to the binary star-system that created them, and the light from the star system's galaxy can be observed for red shifting to find the speed at which it is moving away from us. Distance and velocity are the two variables needed to find the Hubble Constant, and gravitational waves mainly provide both of them.

After some time working with a UNIX operating system, I can see the charm. File management and directory navigation is far more efficient in UNIX than Windows. While UNIX's user interface gives any newcomer an unwelcome feeling, its simplicity and speed grows on you.

Memorable Commands:

- ls (List files in the directory)
- pwd (Show directory you are currently working in)
- mkdir (Create a new directory)
- rm (Remove a file)
- cp (Copy the contents of one file to another)

- mv (Rename a file)
- touch (Create a new file)
- cat (Show contents of file without opening it/ append file contents to another file)
- wc (Show number of words, lines, and bytes)
- cd .. (Move up one level in directory tree)
- cd (Move home/Move to specified directory)

Vim is a Linux text editor application whose keystrokes allow for maneuvering around a text more efficiently than with just single line position changes via the arrows. When learned, it is faster to code and write in use than with the mouse alternative.

Memorable Commands:

- vi (Open file using vim)
- h,j,k,l (Left, down, up, right)
- H (Hop top)
- M (Middle)
- L (Leap down)
- w (Forward to start of word)
- e (Forward to end of word)
- 0 (Start of line)
- \$ (End of line)
- o (open new line below current one)
- O (open new line above current one)

Tmux is a terminal multiplexer application for Unix systems that produces multiple terminal “sessions” that can be used interchangeably. This is especially useful when performing tasks that require the entire terminal to function, like running slow programs or working on different projects at the same time. An especially useful feature is that the tmux session will run as long as the computer is on, so even if the terminal is exited, the terminal will remain open. In cases where the terminal is exited without saving, progress on any work will not be lost.

Memorable commands:

- `tmux new -s ''` (Create session with given name)
- `tmux kill-session -t ''` (Delete session with given name)
- `tmux attach` (Attach to last session)
- `Ctrl + b $` (Rename session)
- `Ctrl + b d` (Detach from session)
- `Ctrl + b s` (Show all sessions)
- `Ctrl + b c` (Create window)
- `Ctrl + b &` (Close current window)
- `Ctrl + b %` (Split pane vertically)
- `Ctrl + b “` (Split pane horizontally)

Git tracks changes to files and is particularly useful when working on a single program with multiple contributors. Changes can be isolated to branches from the main branch changes to the program, and all branches can be combined with the main branch at the end to prevent an

overlapping of different changes throughout the process of developing the program. Contributors working on another contributor's buggy code by accident or contributors working on different versions of the program at the same time can be avoided with Git.

Memorable commands:

- `git init` (Make current directory a git repository)
- `git branch` (Create branch from main branch)
- `git checkout` (Go to the specified branch)
- `git add` (Makes git track file if untracked and moves it to staging area)
- `git commit` (Checkpoint your progress on added files)
- `git status` (Lists what going on in your repository)
- `git log` (Commit history)
- `git push` (Applies progress to the remote repository)
- `git pull` (Obtain any progress made in remote repository)
- `git diff` (Show differences in remote repository and current branch)
- `git merge` (Merge current branch with parent branch)

```
1 #include <stdio.h>
2 int main() {
3     int N, ii, jj, c=0;
4     float x;
5     printf("Give the number upto which you want the prime\n");
6     scanf("%d", &N);
7     printf("Find all the prime numbers up to %d\n", N);
8     for(ii=1; ii<=N; ii++){
9         int count = 0;
10        for(jj=1; jj<=ii; jj++){
11            x = (float)ii/(float)jj;
12            if(x == (int)x){
13                count += 1;
14            }
15            if (count >= 3){
16                break; /* We know it is not prime*/
17            }
18        }
19        if (count == 2){
20            c += 1;
21            printf("%d ", ii);
22        }
23    }
24    printf("\n\n");
25    printf("Total number of primes between 1 and %d = %d\n", N, c);
26 }
```

```
1 print("Give the number upto which you want the prime")
2 N = input()
3 N = int(N)
4
5 print("Find all the prime numbers up to {}".format(N))
6 c = 0
7 for ii in range(1,N+1):
8     count = 0
9     for jj in range(1,ii+1):
10         x = ii / jj
11
12         if x == int(x):
13             count += 1
14             if count >= 3:
15                 break
16         if count == 2:
17             c += 1
18             print(ii, end=" ")
19 print("\n\n")
20 print("Total number of primes between 1 and {} = {}".format(N,c))
```


In the C version of the prime number program, two for loops are utilized to obtain numbers up to each number leading to the imputed N. Conditional if statements are used to determine if a given number has

N = 4				
(1st Loop)		(2nd Loop)		
ii	jj	x		
1	1	1/1		✗
2	1,2	2/1	2/2	✓
3	1,2,3	3/1	3/2	3/3
4	1,2,3,4	4/1	4/2	4/3 4/4

more than 2 factors, to stop further searching, and declares if one of the numbers up to N is prime. This is a nested for loop approach. In the Python version of the program, the same approach is used. The only difference is the language used. While the Python version is shorter, it is much slower than the C code.

```
Give the number upto which you want the prime
100
Find all the prime numbers up to 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Total number of primes between 1 and 100 = 25
```

In the Numpy version of the program, one for loop and two if statements are dropped, already making

```
1 print("Give the number upto which you want the prime")
2 N = int(input())
3
4 print("Find all the prime numbers up to {}".format(N))
5 c = 0
6
7 for ii in range(N+1):
8     count = 0
9     jj = np.arange(1,ii+1) #Make array from 1 to ii (current loops number)
10
11     x = ii / jj #Make array for each number/ii's quotient
12
13     kk = np.round(x) #Make array for quotients, rounded
14
15     compare = x == kk #Make array of booleans, Trues are the factors of ii
16
17     if np.sum(compare) == 2: #If compare has 2 Trues, current ii has 2 factors (is prime)
18         print(ii, end=" ")
19         c+=1
20
21 print("\n\n")
22 print("Total number of primes between 1 and {} = {}".format(N,c))
```

it a faster version than its python counterpart. In each loop of the for loop, the number up to N and the created array jj are used to find the number's factors, and boolean indexing is used to identify if said number has more than 2 factors.

N = 4
(1st Loop)(Numpy Array)

ii	jj	x	decimal	rounded	compare	if
1	1	1/1	1	1	<u>I</u>	<u>-</u> ❌
2	1,2	2/1 2/2	2,1	2,1	<u>I,I</u>	Prime ✅
3	1,2,3	3/1 3/2 3/3	3,1.5,1	3,2,1	<u>I,F,I</u>	Prime ✅
4	1,2,3,4	4/1 4/2 4/3 4/4	4,2,1.3,1	4,2,1,1	<u>I,I,F,I</u>	<u>-</u> ❌

```

1 import numpy as np
2 print("Give the number upto which you want the prime")
3 N = input()
4 print("Find all the prime numbers up to {}".format(N))
5 N = int(N)
6 nums = np.arange(1, N+1)
7 x, y = np.meshgrid(nums, nums)
8 coeff = x/y
9 c = 0
10
11 # For each number find out in how many cases the division has no fraction #
12 is_divisible = coeff[:, nums-1].astype(int) == coeff[:, nums-1]
13 prime_cases = np.sum(is_divisible, axis=0) == 2
14 print(nums[prime_cases])
15 print("\n\n")
16 c = np.sum(prime_cases)
17 print("Total number of primes between 1 and {} = {}".format(N, c))

```

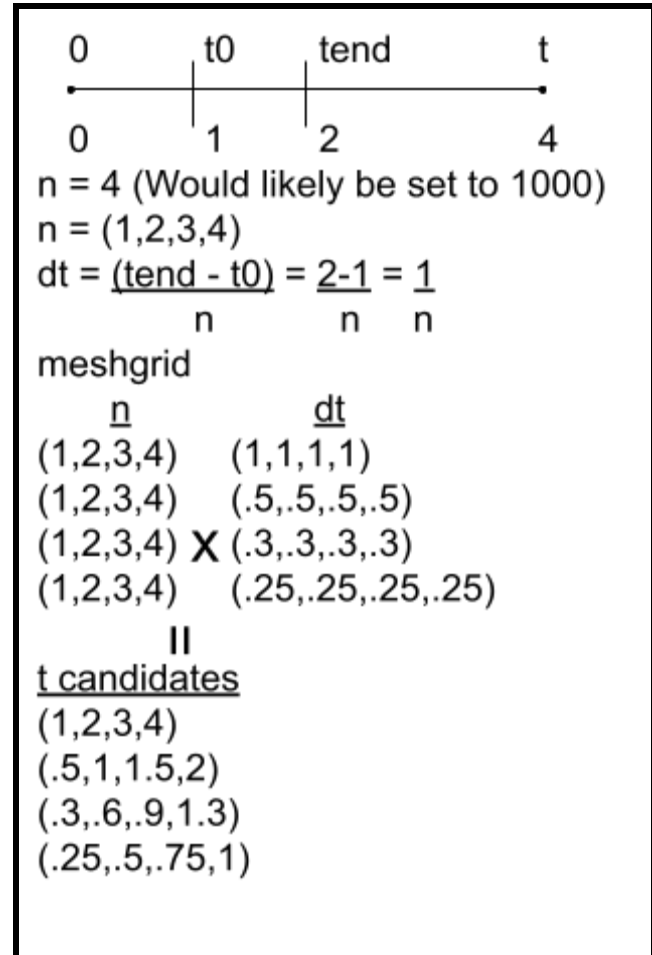
In the full Numpy version of the program, no for loops or conditional if statements are used. Array manipulation and indexing are at play now. The meshgrid function is in large part responsible for eliminating the need for loops. By creating two matrices of the nums array in different directions (x and y), it created the ii and jj lists we've seen in the last versions in an instant. From here, this code's ii and jj (x and y) are divided to find the factors of each number up to N. The whole factors are found by comparing the original coeff with the rounded version. A sum is found and for each row equal to 2 a True is returned in the prime_cases array. Boolean indexing is lastly used to find all the primes up to N.

N = 4
nums = (1,2,3,4)
meshgrid

x	y	coeff	decimal	round	isdivisible	primecase
1 2 3 4	1 1 1 1	1/1 2/1 3/1 4/1	1,2,3,4	1,2,3,4	<u>T</u> , <u>T</u> , <u>T</u> , <u>T</u>	1, 2, 2, 3
1 2 3 4	2 2 2 2	1/2 2/2 3/2 4/2	.5,1,1.5,2	1,1,2,2	F, <u>T</u> , F, <u>T</u>	❌✅✅❌
1 2 3 4	3 3 3 3	1/3 2/3 3/3 4/3	.3,.6,1,1.3	0,1,1,1	F, F, <u>T</u> , F	
1 2 3 4	4 4 4 4	1/4 2/4 3/4 4/4	.25,.5,.75,1	0,0,1,1	F, F, F, <u>T</u>	

For the first approach of the waveform function, a sample size (n) is sought for that will produce a dt that allows total data's time series to reach 0 from t_0 , by subtracting dt 's, without going negative. There is no such thing as a negative time! The dt had to also allow the time series to reach t from $tend$ without going over t . It makes an array for a potential n 's up to the selected n , 1000 at default, and uses it to find all dt 's for each of those n 's. With `meshgrid`, an n and dt matrix in the x and y direction were made and multiplied, as this would obtain all the times that could result from

each n and dt . The t candidates that actually equal the 0 to t_0 interval and $tend$ to t interval are isolated and the minimum value is used. From here padding and boolean manipulation are used to make sure the signal data is the right size, and that data size for the noise and signal are equal.



(T ,F,F,F)	(1 ,1,1,1)
(F, T ,F,F)	(.5, .5 ,.5,.5)
(F,F,F,F)	(.3,.3,.3,.3)
(F,F,F, T)	(.25,.25,.25, .25)

correctdt = .25 (lowest)

$$n = \frac{(tf-t_0)}{n} + \frac{(t_0-0)}{n} + \frac{(tend-t_0)}{n} + \frac{(t-tend)}{n} = totlen \quad (\text{sample size})$$

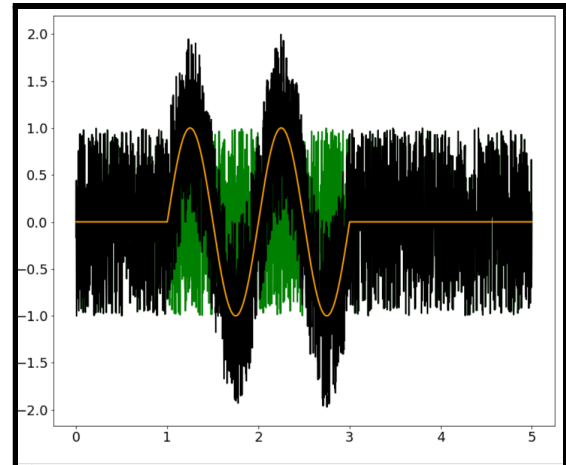
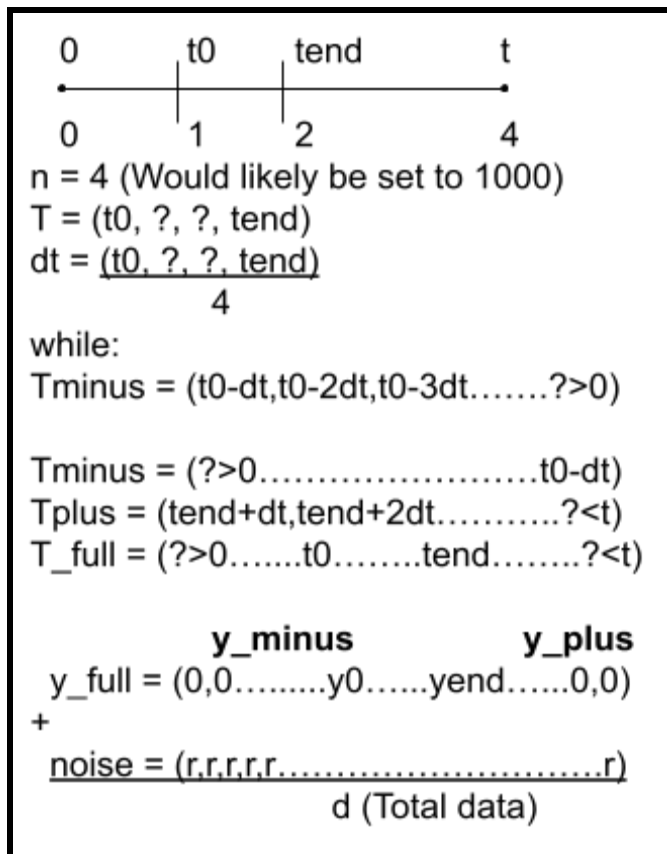
T: (1,2,3.....totlen) x correctdt

T: (0,dt,2d,t0.....tend.....t)

Y: (0,0... Y0... Yend... 0,0)

+
y: (r,r,r,r,r,r,r,r,r,r,r,r,r,r,r,r,r)
 d (Total Data)

The noise is made the same size as the signal's data size. Then, once added together, the total data of noise and an embedded is produced. In large part this program does not compromise on the time boundaries of the data. It searches for a sample size that works with the given inputs, but as a result the more detailed the inputs are rounding causes the returned time series to be slightly off the inputs timestamps and becomes very slow as well.



The latest and best approach to the waveform function is the compromise of the beginning and end timestamps of the total data, as the signal's timestamps are what really matter. For our purposes making the timeseries with proper

intervals under an absolute sample size is the faster algorithm of the two. By use of numpy's `arange` and `linspace` functions, the sample size and interval issue of the problem is made easy. `T` is the time series of the signal, `dt` is its rate of change or interval. By use of the while loop, an array from `t0` to 0 is made in intervals of `dt` that will be cut off before reaching a time before 0; it is reversed to be from 0 to `t0`. This `t_minus` is the time series from 0 to `t0`. The timeseries for `tend` to `t` is made with less care, as the end point should just be around the inputted `t`, and lastly they

are horizontally appended, making the full time series of the data. There is an occasional but inconsequential deviation in the dt's of the time series. From here the zeros_like function is used to pad the signal's data array. The noise is made in the length of the signal's padded data, and the two are combined to form the total data. The dt, T_full, and d are dumped onto a json file for further use.

Crosscor:

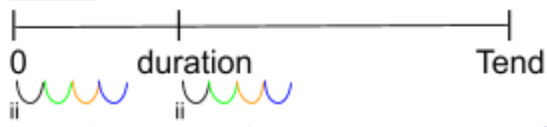
```

__init__:
reads/loads json
self.dt = data["df"]
self.t_full = data["t_full"]
self.data["d"]

template:
self.t : (0.....duration)
              (-gamma*t)
self.y = y(t) = sin(wt)

match:

```



```

0          duration          Tend
ii = 0 (starts as 0)
start and end
indice must
change at the
same in each
iteration of ii

while len(self.tfull[ii:]) >= len(self.y):
    i : 1 2 3 4
    time_slides : tdt 2dt 3dt 4dt
    match : sum(self.d[ii:len(template)+ii] * self.y)

```

(Y0y0+Y1y1+Y2y2+Y3y3+Y4y4...),
 match : (Y1y0+Y2y1+Y3y2+Y4y3+Y5y4...),
 (Y2y0+Y3y1+Y4y2+Y5y3+Y6y4...),

The cross correlation assignments utilizes our waveform function, and will search for the signal data in a pool of noise. This is done by utilizing a template waveform, and “sliding” it across the total waveform data. At each position, the waveform’s data points (Y) and the template’s data points(y) are multiplied and a match (M) is returned. The highest M returned is associated with the position of the waveform data that most closely resembles the

template data.. The great part of this assignment is that the accuracy of the cross correlation approach to finding embedded data can be displayed by utilizing the waveform function’s signal

data. Comparing the actual signal data and the “extracted signal data” will give us a nice test of accuracy.

To organize the multiple functions necessary for this process, and utilize variables across different functions, a class is used. The Crosscor class holds the template method and the match method. In its init method, after reading the data retrieved from running the waveform function, it isolates dt , T_full , and d from the file. The template method, given frequency, γ , and the duration of the signal, creates a time series with the same resolution the waveform function’s signal time series has, and creates displacement values. In the match method, a while loop is used to “slide” the template’s data across the waveform data. While the length of the waveform time series from index ii to the last value is greater than or equal to the length of the template timeseries, ii is iterated. Everytime ii is iterated, the waveform data, starting from an index of ii to the length of the template data, is multiplied to the template data, is summed up, and appended to the match array. With each iteration the displacement in time series, dt , is accounted for and appended to the $time_slides$ array. Each match value now has a time slide value that corresponds to it, which is essentially when finding the time in which the match is at its highest.