Pulsar Searching in Python

Shana Li

Winter Term 2018

Abstract

As a continuation of the NAOC and FAST Pulsar Summer Project of 2017 [1], this project involves converting and optimizing a pulsar searching script, which was composed in Bash in the former project, into the Python programming language. By doing so, I became familiar with Python, a ubiquitous language used in the computational sciences, and produced a script that successfully and efficiently searches for pulsars in radio telescope data using the PRESTO package [2]. The streamlined Python script has consistently proven to be more efficient and equally effective as the older Bash script, and with further application, it can quickly recreate the results of the former script by running it on the 2010 Arecibo data from last summer, and identify new pulsars in data from other sources and epochs as well.

Contents

1 Introduction

Over the summer of 2017, I, along with fellow undergraduate Zach Komassa, travelled to China to work under professor Di Li of the National Astronomical Observatories of China (NAOC) and conduct research on pulsar astronomy. Along the way, I became incredibly familiar with the data processing pipeline for the Five-hundred-meter Aperture Spherical Telescope (FAST), even visiting the FAST site to gain a practical understanding of the process, and composed a Bash computer script that reads specific information from raw radio telescope data and utilizes Scott Ransom's PRESTO suite [2] to search for pulsars.

The project was overall very successful, as the script was able to process and identify strong pulsar candidates along with a confirmed pulsar in a series of data from the Arecibo telescope taken in 2010. However, the computation was very slow, taking about a week out of our ten weeks in China to run through most of the epoch's data. The experience was incredibly fulfilling as we did finally reach our goal, but undeniably, it left us wondering if we did things in the most efficient way.

1.1 Project Goals

Winter Term is the ideal opportunity to revisit my summer project and refine the data processing methods in my script. I chose to undergo this process in the Python language for a variety of reasons: Python is an omnipotent language in the computational sciences, and especially computational astronomy, which is a field I want to get involved in in the long run. Due to its readability, flexibility, and efficiency, it is an extremely popular choice for scientific data processing packages. Its rapidly expanding libraries also give programmers a boundless variety of available tools. Getting well-acquainted with Python will help me greatly in my computational physics career, and the programming skills that I develop over this time are essential to my classes and on-campus research.

Continuing my work on the FAST project, I can gain a more robust knowledge of the pulsar searching process, PRESTO processes, and the FAST pipeline, which is integral to my ongoing research in pulsar astronomy. Furthermore, the script that I compose will be more cohesive and efficient, and can be easily packaged and shared to the wider scientific community, where it can be used and extended by pulsar astronomers to expand the worldwide database of known pulsars.

1.2 Project Accomplishments

Over the duration of this project, I have gained fluency in the Python language, gained confidence in my Python programming skills, and am now comfortable coding in the language for larger projects. I have also acquired programming experience that is relevant to both my research and classes: after working extensively with Python imports and libraries, I now have a robust knowledge of the os and subprocess imports, string methods, and the processes that occur within the computer system when commands are executed.

I have successfully composed a Python version of my original pulsar searching script, which can accomplish the same tasks as its Bash predecessor with improvements in design, readability, and performance. The script has also been neatly packaged and uploaded onto GitHub [3], where its development process has been consistently tracked and its accompanying README file gives a summary of the program's capabilities and setup requirements. The GitHub repository allows the script to be easily shared, run, and expanded by astronomers anywhere.

This report details the process involved in developing the Python script, including the following:

1. Setup process involved in accessing and working on NAOC's server;
2. Each section of the script and its functionality, coding techniques involved in development, and optimization settings;
3. Runtime comparison of Python script with original Bash script.

2 Setup Procedure

Since I have a new Windows configuration instead of the Macbook I had worked on during the summer, I underwent a different procedure to set up an SSH connection to NAOC's server and open a VNC desktop window to work on a node in the cluster. To do so, I used PuTTY [4], an SSH client for Windows, and RealVNC's VNC Viewer software [5].

2.1 PuTTY Configuration

In PuTTY's configuration window, enter the IP address 159.226.170.68 using the port 33322 (as shown in figure 1) to connect to the fourth node of NAOC's cluster, which is the most powerful node for computational purposes. Afterwards, a terminal window (shown in figure 2) will pop up, and proceed to log in with authorized user credentials.

After the connection has begun, if a new VNC window has not been previously created, enter the command:

```
> vncserver :xx
```

where XX is the VNC desktop number. I used the desktop number 38 for the duration of this project.

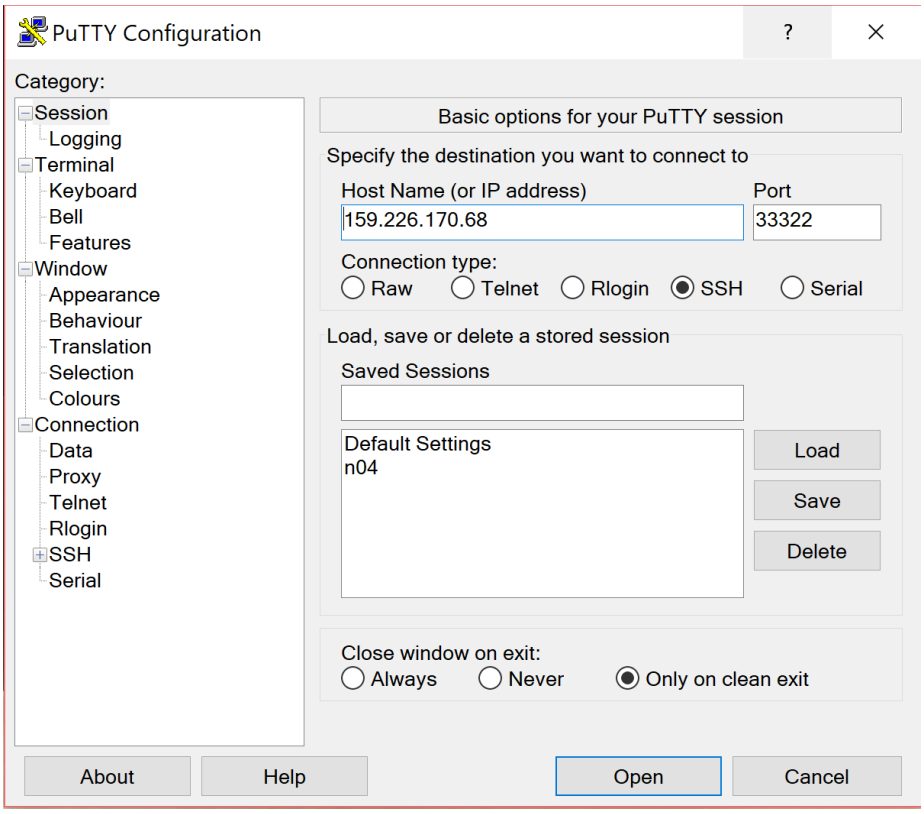

Figure 1: The PuTTY configuration window with the correct settings to log in to node four in NAOC's server.

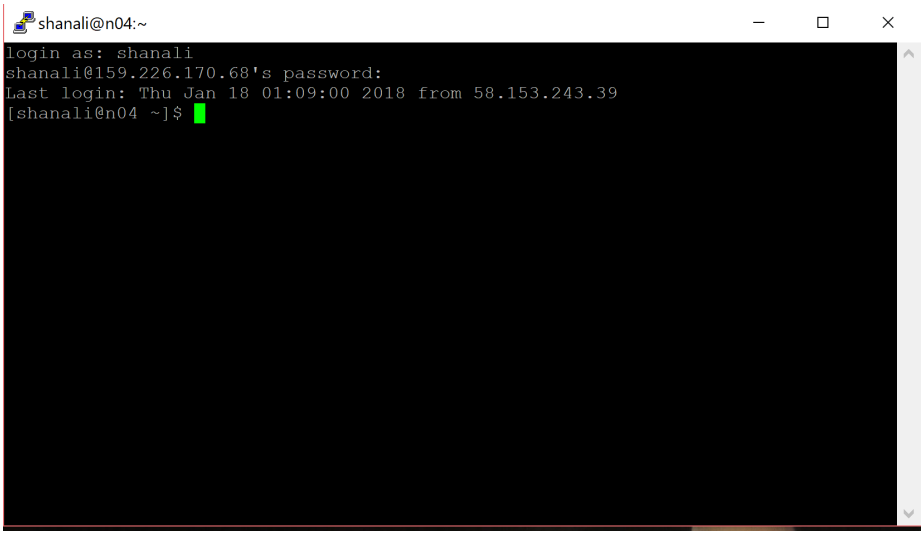

Figure 2: The PuTTY terminal window for logging in with user credentials.

2.2 VNC Viewer Settings

For the VNC Viewer program, a new connection should be set up while PuTTY is still connected. The VNC Server address is localhost:59XX, where XX is the desktop number as set up before (figure 3). Figure 4 shows the VNC window that the connection opens.

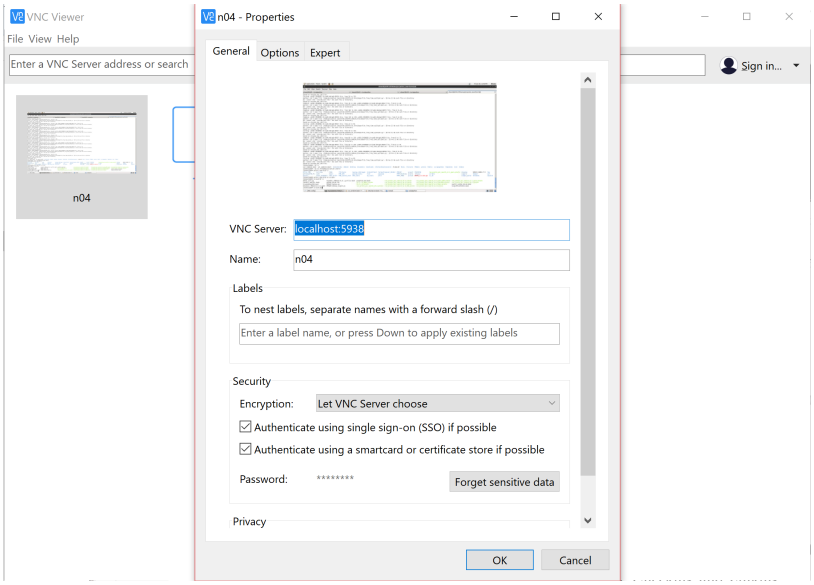

Figure 3: The VNC Viewer configurations for a remote desktop connection to the NAOC server.

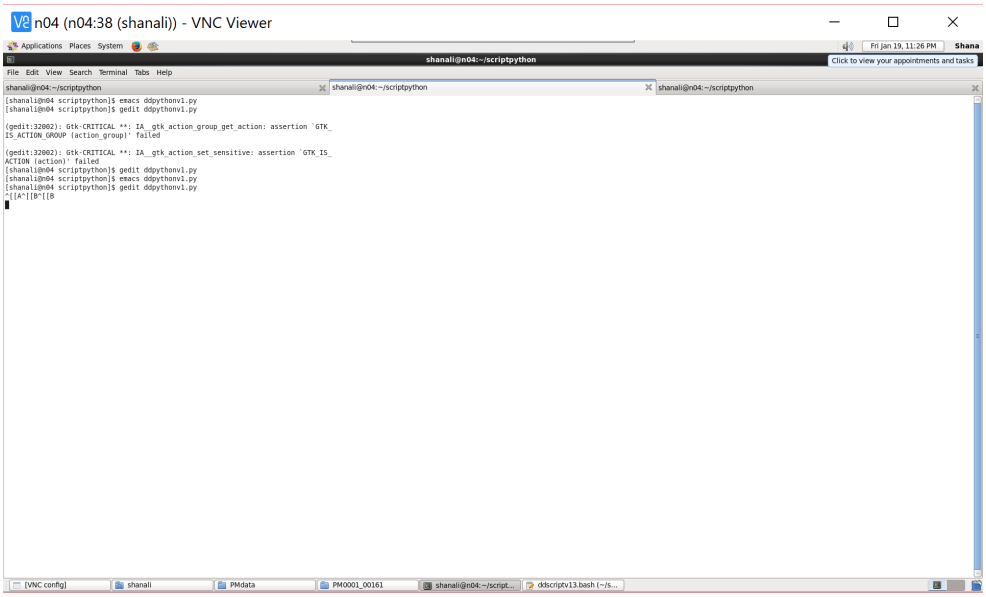

Figure 4: The VNC window that the connection opens, where all the coding, testing, and running is done.

3 Coding Techniques and Optimization Measures

3.1 File Checking and Linking

As before, the beginning of the script contains an introductory title section for the script, which is run with two command line arguments: the directory of the file to process, and the name of the file. The following are implementation differences that have been programmed into the Python script:

1. The filename is now submitted inclusive of its file extension.
2. The script can now analyze files of different extensions as long as it is a valid data file, and not just files ending with .sf.
3. The naming of the variables within the program are now a lot more clear; the full filename (inclusive of the file extension) is denoted "FILENAME", with the all-caps indicating that the variable is final and will not be changed (this is also true for the start time and end time variables for calculating the program's runtime). There is now a different "foldername" variable that stores the name of the file without its file extension, which is for indicating the folder in which the data analysis takes place. This naming solves a lot of confusion, making code a lot more understandable for users and developers.
4. For checking the validity of the paths, the directory and filename are appended to a single directory to check, making the process theoretically twice as fast as before.
5. Instead of overwriting the folders for analyzing a data file, the Python program creates numbered duplicates of the folders to process a file more than once. This is a lot safer, and a lot more useful in debugging and using different parameters to re-process the same data.

```
1  STARTTIME = timeit.default\_timer()
2
3  print("************************************************************")
4  print("PRESTO data processing - now in Python \n")
5  print("by Shana Li \n")
6  print("Last updated 2018-01-31. \n")
7  print("Arguments should be in the format: DIRECTORY FILENAME.")
8  print("Filename SHOULD INCLUDE file extension (.sf).")
9  print("************************************************************ \n")
10
11 DIRECTORY = sys.argv[1]
12 FILENAME = sys.argv[2]
13 PATHNAME = DIRECTORY + "/" + FILENAME
14
15 #check validity of the path to file
16 if not os.path.exists(PATHNAME):
17         print("Invalid DIRECTORY and/or FILENAME. Please check input and try again
       . \n ************************************************************ \n")
18         exit(1)
19
```

```
#create new folder for data processing; if the file has already been processed
    here, create new folder with indexed name
#folder name starts out with the FILENAME without the extension
namearray = FILENAME.split(".")
fextension = namearray[len(namearray)-1]
foldername = FILENAME[:-(len(fextension)+1)]
x = 1
while os.path.exists("./" + foldername):
        if x == 1:
                foldername = foldername + "\_" + str(x)
        else:
                foldername = foldername[:-(len(str(x-1))+1)] + "\_" + str(x)
        x += 1
os.makedirs(foldername)
print("Data will be processed into the folder " + foldername + ". \n")

#create link to the file to process within the local folder
os.chdir(foldername)
os.symlink(PATHNAME, FILENAME)
print("Link to file " + FILENAME + " made. \n
    ********************************************************** \n")
```

3.2 rfifind

This part of the script runs the rfifind function to locate and mask RFI in the radio signals. The mask is used for multiple parts of the data processing sequence further down teh line. The differences introduced in this version of the script are:

1. The 'time' option for the rfifind command is stored in a variable instead of being hard coded in, allowing for easy alteration for multiple runs of the script.

```
print("********************************************************** \n")
#run rfifind
print("Running rfifind to detect and mask RFI: \n")
start = timeit.default_timer()
#time option for rfifind
rfitime = "2"
print("Filename: " + FILENAME + "\nTime: " + rfitime + " \n")

#spawn a shell process to run rfifind
os.system("rfifind " + FILENAME + " -time " + rfitime + " -o " + foldername + " >>
    /dev/null")

end = timeit.default_timer()
print("Time for rfifind: " + str(end - start) + " seconds.")
print("********************************************************** \n")
```

3.3 DDplan

This section extracts vital information from the binary data file to run DDplan.py with specified parameters to generate a de-dispersion plan correcting for dispersion of the radio signals. The

implementation differences include:

1. Because Python makes it difficult to read binary data directly, the formatted header of the file is catalogued by a shell process spawned by operations in the os library, and piped into a .txt file so that it can be read easily.

2. To look for the parameters required to run DDplan and other functions, the file is read through once and keywords are searched for at every line. This occurs a lot faster than in the Bash script, where the file is looked through every time a keyword is searched for.

```
print("************************************************************ \n")
print("Running DDplan.py to compose de-dispersion plan:")
start = timeit.default_timer()

#constant values
#low dm
ldm = "0"
#high dm
hdm = "4000"
#time resolution
tres = "0.5"


#make txt version of binary file header
fname = foldername + ".txt"
os.system("readfile " + FILENAME + " > " + fname)
f = open(fname, "r")

#obtain values from file
for line in f:
        #central frequency
        if re.search("Central freq", line):
                cfreq = line.split()[4]
        #number of channels
        if re.search("Number of channels", line):
                numchan = line.split()[4]
        #bandwidth
        if re.search("Total Bandwidth", line):
                bandw = line.split()[4]
        #sample time
        if re.search("Sample time", line):
                sampletime = str(float(line.split()[4]) / 1000000)
        #numout
        if re.search("Spectra per file", line):
                numout = line.split()[4]
                if int(numout) & 1:
                        numout = str(int(numout) - 1)
        #nsub
        if re.search("samples per spectra", line):
                nsub = line.split()[4]

#run DDplan.py
```

```
print("DM: 0 to 4000 \nTime resolution: " + tres + " seconds \nCentral Frequency:
    " + cfreq + " MHz \nNumber of Channels: " + numchan + "\nTotal Bandwidth: " +
    bandw + " MHz \nSample Time: " + sampletime + " seconds") + "\n"
os.system("DDplan.py -l " +  ldm + " -d " + hdm + " -f " + cfreq + " -b " + bandw
    + " -n " + numchan + " -t " + sampletime + " -r " + tres + " -o " + foldername
    + " | tee " + foldername + "_ddplaninfo.txt >> /dev/null")
print("Results saved in " + foldername + "_ddplaninfo.txt. \n")

end = timeit.default_timer()
print("Time for DDplan: " + str(end - start) + " seconds.")
print("********************************************************** \n")
```

3.4 prepsubband and realfft

Here, prepsubband is run on each call in DDplan, the results of which are exported onto a text file, and realfft is run to conduct a fast-Fourier transform of the data. No substantial implementation differences have been applied to this section.

```

print("********************************************************** \n")
print("Running subband de-dispersion using prepsubband:")
start = timeit.default_timer()

#call prepsubband on each call in DDplan, starting from line 14
i = 14
line = linecache.getline(foldername + "_ddplaninfo.txt", i)
while line != "\n":
        #put arguments into array
        args = line.split()
        #run prepsubband
        print("Low DM: " + args[0] + "\nDM step: " + args[2] + "\nNumber of DMs: "
    + args[4] + "\nNumout: " + str(float(numout)/float(args[3])) + "\nDownsample:
    " + args[3] + "\n")
        os.system("prepsubband -lodm " + args[0] + " -dmstep " + args[2] + " -
    numdms " + args[4] + " -numout " + str(int(numout)/int(args[3])) + " -downsamp
    " + args[3] + " -mask " + foldername + "_rfifind.mask -o " + foldername + " " +
     FILENAME + " >> /dev/null")
        i += 1
        line = linecache.getline(foldername + "_ddplaninfo.txt", i)

end = timeit.default_timer()
print("Time for de-dispersion: " + str(end - start) + " seconds.")
print("********************************************************** \n")

###############################################

print("********************************************************** \n")
print("Running realfft for Fourier transform:")
start = timeit.default_timer()

os.system("ls *.dat | xargs -n 1 --replace realfft {} >> /dev/null")

end = timeit.default_timer()
```

```
print("Time for realfft: " + str(end - start) + " seconds.")
print("************************************************************ \n")
```

3.5 accelsearch

In this section, the accelsearch function is run to search for periodic candidates in the processed signals. The main implementation differences that have been included are:

1. The 'zmax' parameter is saved as a variable instead of being hard coded in, so that it can be changed easily.

```
print("************************************************************ \n")
print("Running accelsearch to search for periodic candidates: \nUsing zmax = 0.")
start = timeit.default_timer()

#values to use for accelsearch
zmax = "0"

os.system("ls *.fft | xargs -n 1 accelsearch -zmax " + zmax + " >> /dev/null")

end = timeit.default_timer()
print("Time for accelsearch: " + str(end - start) + " seconds.")
print("************************************************************ \n")
```

3.6 accelsift

Here, ACCEL_sift.py is run to sift through the periodic candidates. The implementation improvements include:

1. The name of the text file in which the candidate information is saved contains the zmax value in its title, so that different runs of it with different zmax values can be easily compared.

```
print("************************************************************ \n")
print("Running ACCEL_sift.py to sift through periodic candidates:")
start = timeit.default_timer()

os.system("python $PRESTO/python/ACCEL_sift.py > cands_zmax_" + zmax + ".txt")
print("Results saved in cands_zmax_" + zmax + ".txt.")

end = timeit.default_timer()
print("Time for ACCEL_sift: " + str(end - start) + " seconds.")
print("************************************************************ \n")
```

3.7 prepfold

In the last section of the script, the prepfold function is run to fold multiple signals of periodic pulses to generate an accurate plot for the candidate. The implementation differences are:

1. Instead of looping through reading the file multiple times to make separate lists containing each set of information, the candidates' ACCEL_0 filename, .dat filename, candidate number, and DM are all saved into a 2D array, which is constructed fully by reading through the candidate file only once. This is a much faster option, since only one entity needs to be constructed, and the access time for an array is constant.

```
#make 2D list of best candidates and their attributes: ACCEL_0 filename (0), dat file name (1), candidate number (2), DM (3)
#period = str(float(words[7])/1000) (if ever needed)
cands = open("cands_zmax_" + zmax + ".txt", "r")
candlist = []
for line in cands:
        if re.search("ACCEL", line):
                words = line.split()
                accelstring = words[0].split(":")
                c = [accelstring[0], accelstring[0][:-8], accelstring[1], words[1]]
                candlist.append(c)

#loop prepfold through all viable candidates in candlist
for c in candlist:
        print("Running prepfold on candidate #" + str(candlist.index(c) + 1) + " of " + str(len(candlist)) + ":")
        #run prepfold command
        print("File: " + c[1] + "\nCandidate number: " + c[2] + "\nDM: " + c[3] + "\nnsub: " + nsub + "\n")
        os.system("prepfold -mask " + foldername + "_rfifind.mask -dm " + c[3] + " " + FILENAME + " -accelfile " + c[0] + ".cand -accelcand " + c[2] + " -noxwin -nosearch -o " + foldername + "_" + c[3] + " >> /dev/null")

#display png files
print("Folding finished, plots have been saved as the following .png files:")
for file in os.listdir(os.getcwd()):
    if file.endswith(".png"):
        print(file)

end = timeit.default_timer()
print("Time for folding candidates: " + str(end - start) + " seconds.")
print("********************************************************** \n")
```

4 Script Performance

The tables below document the runtimes of each section of the Bash and Python scripts, as well as the total time elapsed to process one file. To gather this data, each script is run ten consecutive times to process the single file "PM_0026_00511" from the batch of Arecibo files. The runtimes are taken through the date timestamp for Bash, and the default timer from the timeit import for Python. Average runtimes are calculated by omitting the maximum and minimum values of each section, and taking an average of the median eight times to account for occasional anomalies.

Table 1 contains the runtime information for the original Bash script, and table 2 contains the runtime information and an optimization percentage, which is the percentage decrease in runtime, based on the average runtimes of each section, when the script is converted to Python compared to its original Bash version. In other words, it is the difference between the average runtimes of the Bash and Python scripts for each section, divided by the runtime of the Bash script, expressed as a percentage value.

Section / Runtimes (s)	File preparation	rfifind	DDplan	prepsubband	realfft	accelsearch	accelsift	prepfold	Total
	4	53	1	235	335	268	6	368	1270
	10	52	0	237	335	271	5	366	1276
	10	52	0	232	354	254	4	368	1274
	11	51	0	230	350	260	5	366	1273
	8	51	0	232	*550*	*387*	6	368	*1602*
	6	51	0	229	361	268	6	367	1288
	10	51	1	228	343	310	4	368	1315
	7	52	2	227	346	342	7	374	1357
	9	51	1	232	*448*	349	*12*	370	*1472*
	7	52	0	239	398	284	5	371	1356
Average	8	52	1	232	353	290	5	369	1301

Table 1: Table of runtimes and average runtimes by section and overall for the original Bash script, based on ten trials. The bolded values have been regarded as anomalous, and were ignored in the calculation of the averages.

Section / Runtimes (s)	File preparation	rfifind	DDplan	prepsubband	realfft	accelsearch	accelsift	prepfold	Total
	0	52	1	231	354	248	5	366	1256
	1	51	1	227	326	258	4	363	1231
	1	51	0	234	342	234	4	362	1228
	0	51	0	234	342	238	3	364	1231
	0	52	0	232	339	251	5	364	1243
	0	52	0	229	329	249	3	364	1226
	0	52	0	232	335	238	4	363	1223
	1	51	0	236	337	245	5	360	1235
	0	51	0	240	354	251	5	365	1266
	1	51	0	241	345	228	4	367	1237
Average	0.4	51	0.1	234	340	244	4	364	1226
Improvement from Bash	95%	0%	60%	-1%	4%	16%	21%	1%	5%

Table 2: Table of runtimes and average runtimes by section and overall for the Python script, based on ten trials. The lowest row indicates the performance improvement percentages from the original Bash script.

4.1 Analysis

First and foremost, there are a few irregularities that drift significantly from the other data in the Bash times, leading to a few values being disregarded as anomalies when calculating the average. These variations are most likely caused by background processes consuming large amounts of

computing power that occurred while the program ran, making the process take longer. Hence, they have been corrected for by removing them entirely from the average calculations.

According to the data, all the sections in the program experience a decrease in runtime or stayed at the same efficiency after being converted and altered in Python. The rfifind, prepsubband, and prepfold sections are all functionally unchanged from before, and their slight differences in performance are negligible. Out of the sections which have experienced a significant decrease in runtime, some of them, such as file preparation or DDplan, take a marginal amount of time already, so don't impact the performance of the entire script much. Others that take up a more significant amount of time, such as accelsearch, have been sped up either due to implementation improvements in the section themselves or in other portions of the script, and contribute greatly to the overall performance of the script. However, some sections might have only appeared to be optimized due to coincidence and inaccuracy due to the small sample size, since their implementations are identical to the original. For example, in accelsift's case, it is apparent that because the times are so short and in both cases vary among each other by amounts greater than one second already, that the single second difference between the Bash and Python versions is negligible.

Overall, the program has experienced a 5% decrease in total runtime, which is statistically significant. Although for the data file in question, this percentage only corresponds to about 30 seconds out of 1300, the amount of time that can be saved in processing longer files of larger quantity is extremely notable. The performance increase saves both computational time and space; in the wider context of radio astronomy where hundreds of gigabytes of data are taken every day, a 5% decrease in runtime can make room for a great amount of data processing.

Of course, the runtime analysis conducted in this way is merely preliminary; the time measurement is hindered by the fact that the times can only be taken to the nearest second, and in an ideal case, there would be a lot more trials taken under a controlled computing environment where no other processes would run at the same time. Hence, a more accurate and conclusive analysis of the performance optimization could be done with more time and in a different environment.

5 Conclusion

All in all, this Winter Term project was successful in achieving my programming and educational goals. In a month's time, I have converted my original Bash pulsar searching script to Python, while implementing design and performance improvements to ultimately decrease the script's overall runtime. Along the way, I have familiarized myself with the Python programming language and some of its associated libraries, and am now more comfortable coding in the language for other projects in my academic courses and extracurricular research.

5.1 Next Steps

My project can be extended to converting the cut and combine script, which was written alongside the original Bash pulsar searching script as a tool to measure and cut out overlapping strips of radio telescope data to run through the pulsar searching process, to Python as well. Afterwards, the Arecibo drift scan data that I had originally worked with during the summer (which is currently stored in the Guizhou server that I do not yet have access to) can be run through both scripts, and the resulting plots can be looked through by eye to search for likely pulsar candidates. I have performed this process on a portion of the drift scan data during the summer, but I am yet to successfully process the entire batch of files from Arecibo.

References

[1] Shana Li, "Pulsar-Scripts-NAOC-FAST", *https://github.com/shanalii/Pulsar-Scripts-NAOC-FAST*

[2] Scott Ransom, "PRESTO", *http://www.cv.nrao.edu/∼sransom/presto/*

[3] Shana Li, "fast_script_python", *https://github.com/shanalii/fast_script_python*

[4] "Download PuTTY", *https://www.putty.org/*

[5] RealVNC Limited, "Download VNC Viewer", *https://www.realvnc.com/en/connect/download/viewer/*