Asteria Project Documentation

Albert van Kiel, Timo Hermsen, Robin Baneke,
Max van Hasselt, Robert Kleef and Menno Prinzhorn

January 21, 2019

Chapter 1

Introduction

During the course of this project, we are creating a free and open source framework that contains the generic algorithms and file handling for astronomical data sets. This framework will be modular. Similar to OpenCV, wherein specific modules can be added and disabled depended on the needs of a project. This framework will be implemented in Python and C++.

1.1 Current situation

The development of the application is split into multiple parts and distributed over two different teams, there are two teams on handling the in and output (team IO) and one implementing algorithms (team Algorithm). Each team is assigned to work on their own part of the project, at the end of each sprint the parts will be merged together, the result will be delivered to the product owners.

For the IO part the eScience Center is in need of an application that is able to both read and write filterbank files. There already is an application in use. However, the current application is outdated and therefore has to be rewritten. The scope of this project is to rewrite a new software stack with updated technology, such as using the GPU which is highly modular and re-usable.

The Algorithm part will be developing an application that detects pulsars in large datasets. The pulsars will be detected using the fast fourier transformation. Also, since the datasets are very large, we might want to do a parallel computing implementation for the algorithm in the future.

We are using Scrum as our project method. By applying Scrum we are able to make changes, based on the feedback given by our product owners, whenever necessary to the application.

Chapter 2

Plan of action

2.1 Mission

The purpose of the IO part of the project is to read and write filterbank files.

For the Algorithm part of the project the goal is to detect pulsars in large datasets using algorithms.

The solutions will first be implemented in Python and later in C++.

2.2 Monitoring performance

In order to monitor the performance of our development our product owners decided to plan weekly reviews. These reviews could be used to discuss the pace of our project. If our product owners are not satisfied with the pace of our development team, we could look at possible improvements.

2.3 Risks

- A shortage of knowledge about Python/C/C++
- An inflation of requirements
- A wrong estimation of required development time
- Lack of motivation

Chapter 3

Technical details

3.1 Requirements

The requirements can be divided into two different categories. There are both functional and non-functional requirements.

3.1.1 Non-functional

The non-functional requirements describe the technical requirements for the application.

Since some users might have problems understanding C++ code we will develop a standalone version in Python, since Python is readable even without a lot of programming knowledge. By doing this we give users the possibility to focus on the logic of the algorithm without needing C++ knowledge.

- Support for:
  MacOS High Sierra (x86_64), CentOS (x86_64), Raspbian(ARM x64)
- Python 3.6
- C++ 14
- Must be modular

3.1.2 Functional

All users:

- As user I want to read filterbank files in my program, so we can use the astronomical data in scientific programs.
- As user I want to write filterbank files in my program, so we can write astronomical data in scientific program

- As a user I want to downsample given input, so it can be used in a later stage.
- As user, I want clear exceptions when something is not set right, so i can more easily debug my program.

AUAS Students:

- As student I don't to worry about thread safety, so it's usable in highly threaded applications.
- As student I want that Asteria is easily imported, so I can easily import it in other projects.
- As a student, I want that Asteria runs on limited PC hardware, so I can test my projects on my personal computer.

Scientific programmer:

- As scientific programmer I want that my filterbank reader performs with very large datasets (+1TB of data), which is often the case in software
- As scientific programmer I want that my filterbank writer performs with very large datasets(+1TB of data), which is often the case in software
- As scientific programmer, I want to downsample large sizes the input time series(+1TB) so it can be used in astronomical projects.

3.2 Efficiency measurements

In this chapter we discuss the efficiency of the modules and its methods, by performing benchmarks and profiling and looking at the Big O of some methods. The only module we did not test for this chapter, is the plot module. We decided to exclude this module because the plots plays no role in the pipeline and is not very complex. Thus their complexity and performance being irrelevant for now.

Furthermore, for running the benchmarks and profiling we decided to use a Python script, rather than an Ansible or Bash script. We chose for Python because our team had more knowledge regarding Python than Ansible or Bash. Furthermore, we had to run the benchmarks on an old machine running CentOS 6, and installing dependencies like Ansible on this machine would be difficult. Bash could have been a possibility, however, since our team and the upcoming team might not have much knowledge regarding Bash, creating a Bash script to run the code could become a problem.

3.2.1 Computional complexity

In this section we describe the computional of each module we implemented in the Asteria library.

First, the complexity of reading the filterbank data is equal to $O(n * m)$, where n is equal to the amount of samples and m is equal to the amount of intermediate frequency channels. For selecting the filterbank data, when reading the filterbank data at once, the complexity is equal to $O(n)$.

Second, for the clipping module, the complexity can be described as $O(n + 2m)$; where n is the amount of samples and m is equal to the amount of columns.

Third, the complexity of the dedispersion module is equal to $O(2n + m + m^2)$. However, for the benchmarks we used the old dedispersion module, back when the complexity was equal to $O(n)$.

The timeseries, which includes downsampling, has a complexity that is equal to $O(2n)$.

The complexity of the discrete Fourier transformation is equal to $O(n^2)$, this means that the performance is equal to the square of the amount of samples.

For the fast Fourier transformation the complexity is equal to $O(n \log n)$, which means that the performance is equal to linearithmic of the amount of samples. And even though, the complexity of the fast Fourier transformation is already less than the discrete Fourier transformation, even more gains could be made when performing the fast Fourier transformations in parallel. A process that is impossible to do using the discrete Fourier transformation.

To test whether the expected Big O holds true when running the actual methods, we decided to benchmark the different methods. The results of

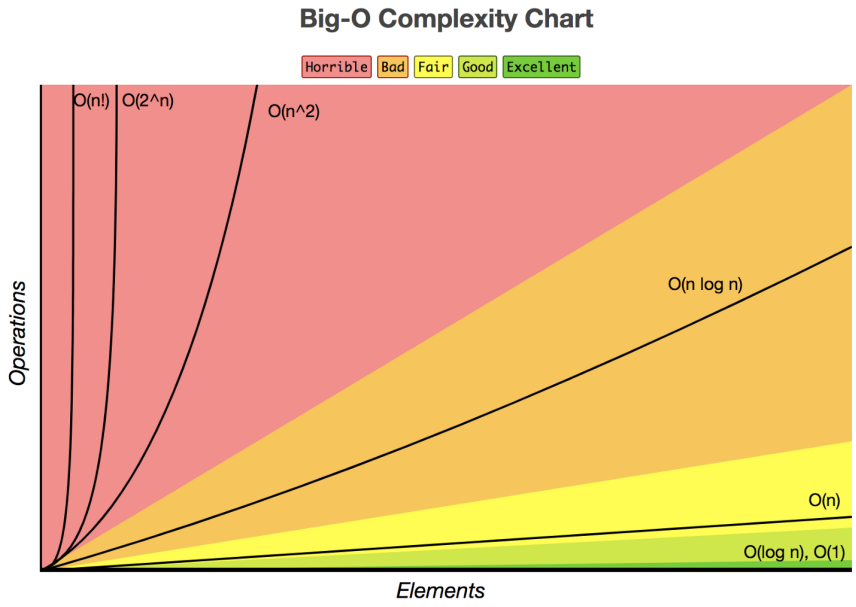

Figure 3.1: The Big O complexity chart displays the curve for each function and describes the performance using colors.

the benchmarks are described and explained in the upcoming paragraph.

3.2.2 Benchmarks

Furthermore, we measured and compared running the discrete Fourier transformation with the fast Fourier transformation. For each method, we ran it using different sample sizes. After performing the measurements we noticed that while the time it took to run the discrete Fourier transformation increased almost quadratically, the time it took to run fast Fourier transformation increased linearithmicly. This difference in efficiency becomes increasingly important when running the pipeline using larger sample sizes.

3.2.3 Profiling

For profiling we ran the pipeline module using a static filterbank file a 100 times with a total of 3070 and 49150 samples. The pipeline module ran and measured the time it took to run each method of the Asteria library. After measuring the time, we calculated the standard deviation to look if there were any outliers and after noticing no significant outliers, we used the mean of each method for visualizing the average measurements.

To be able to compare the efficiency of the methods, we decided to visualize both a horizontal bar chart and a pie chart for each method. By doing so, we hope to see whether there are any significant results that we

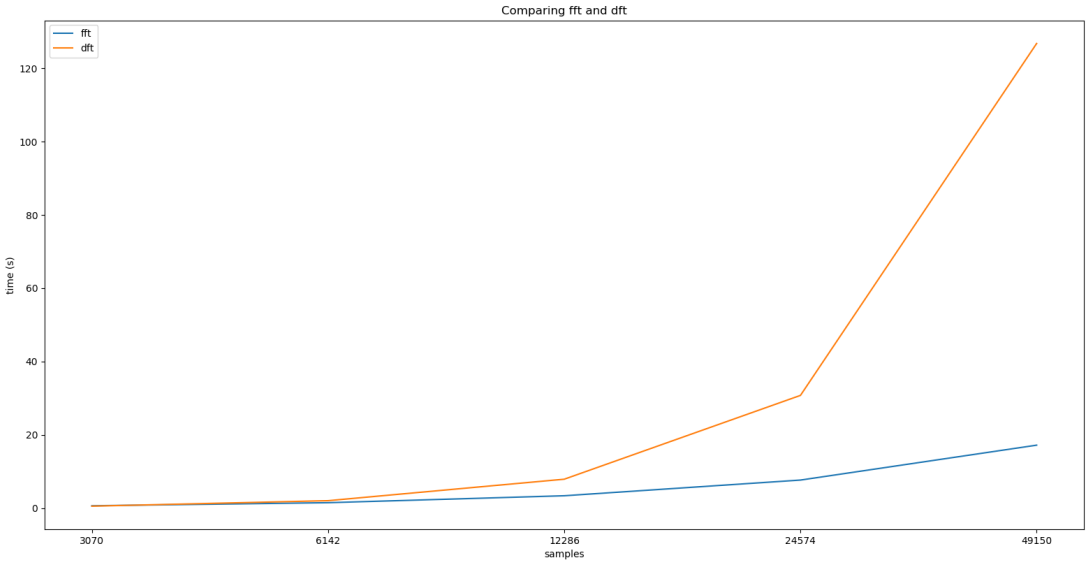

Figure 3.2: A linechart of the measurements of the discrete Fourier transformation vs the fast fourier transformation.

samples	DFT (s)	FFT (s)
3070	0.55	0.66
6142	2.05	1.50
12286	7.90	3.39
24574	30.77	7.67
49150	126.78	17.19

Figure 3.3: The table displays the exact values of the measurements.

would otherwise fail to notice. Furthermore, to see whether there are any relative changes when running different sample sizes, we also decided to display the bar chart for both the 3070 and 49150 samples.

After plotting the two different bar charts we noticed that there is a significant difference between the efficiency of the different methods. Especially the discerete Fourier transformation is incredibly slow when compared to the fast Fourier transformation (as discussed in the previous section). Furthermore, when the sample size increases, so does the difference between the discrete Fourier transformation and the other methods. Besides the discrete Fourier transformation there are no significant differences between the methods for the increased sample size.

The results show that the average time it takes to run the entire pipeline

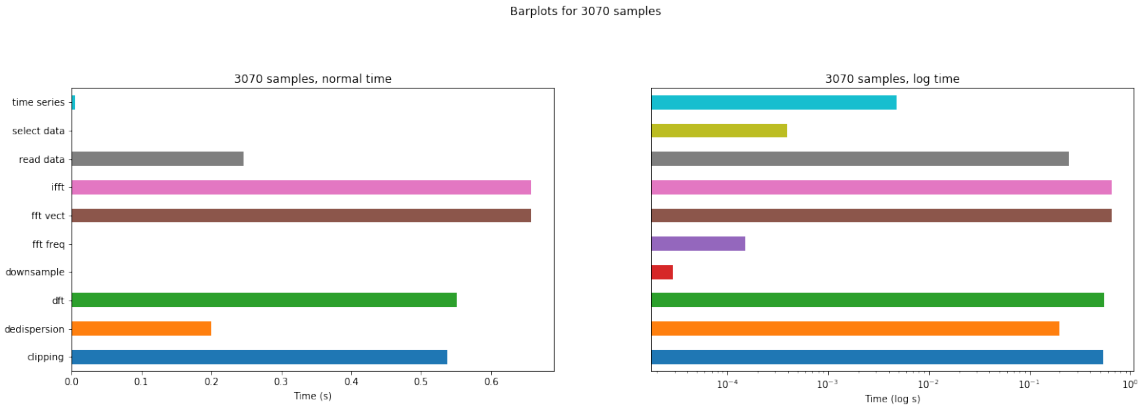

Figure 3.4: Measured time per method for 3070 samples.

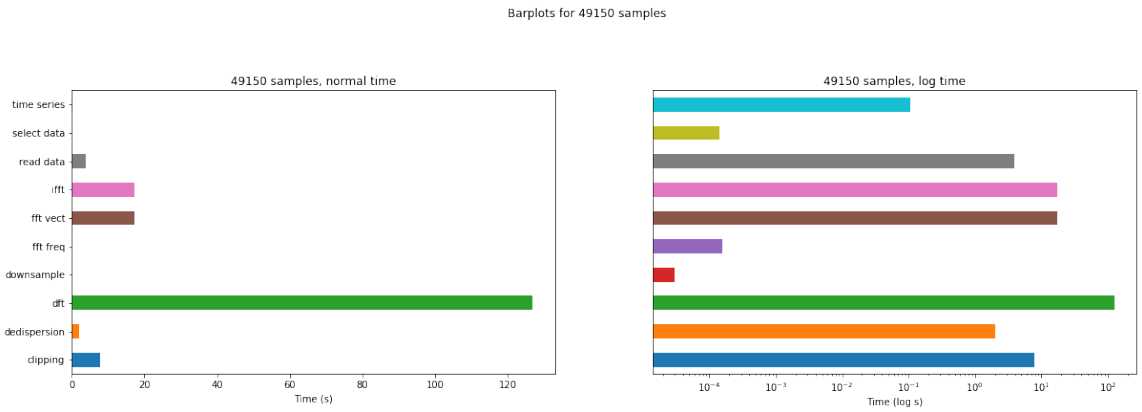

Figure 3.5: Measured time per method for 49150 samples.

is equal to around 175 seconds. Around 70% of that time is spend on calculating the discrete Fourier transformations. After that, the fast Fourier and inverse fast Fourier transformations make up for the largest part of the remaining time, followed by the clipping module. The distribution of the time is most efficiently displayed using a piechart, displayed below.

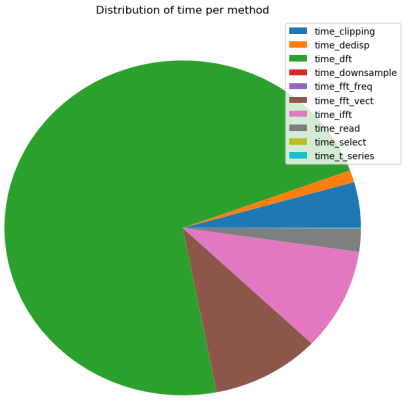

Figure 3.6: The piechart shows how the time of running a filterbank pipeline is distributed between the different methods.

3.3 What's next

In the previous chapters we looked back at the significant results we achieved. However, for the Asteria project to be succesfully adapted by both professional and amateur astrophysicists, a lot of changes still have to be made to the current Asteria project. This mostly concerns implementing additional features to the current Python pipeline and porting the project afterwards to C and C++.

3.3.1 Additional features in Python

The plans for which features to implement in the pipeline came from existing pulsar detection libraries like SigProc and Presto. Furthermore, some research papers describe which features should be created and in which order these features should be implemented in a pipeline.

A feature most pulsar detection libraries include, but ours is missing, is the feature of identifying birdies. This feature is used to identify bad frequencies after performing the fast Fourier transformations. These bad frequencies are begin identified saved in a file, so they can be easily removed in the future.

Furthermore, the harmonic summing is something we were advised to implement by the Escience Netherlands. The Harmonic summing is done after running the fast Fourier transformations and identifying the birdies. Harmonic summing is used to reconstruct power distributed throughout harmonics back in the Fourier domian. This is repeated multiple times to

increase the effect.

After that, the data is searched for single and periodic signals. Every single that surpasses the given threshold should be logged and saved for the upcoming detection processes. Then sifting is used to identify duplicate detections of the same signal at slightly different dispersion measures. These duplicates should be removed to avoid confusion.

After removing the duplicate signals, the candidates that remain should be phase-folded. In the folding process, the samples are transformed to display as a function of time. This changes the signal-to-noise ratio and thus makes it easier to spot pulsars, but removes the information that could be extracted from the frequecies. Again the information that is gained from this analysis should be stored for later use.

The last step is performing transient searches. The transient search smooths the time series using boxcar filters, before identifying the possible candidates using an appropirate signal-to-noise threshold.

Additionally machine learning could be used to identify the optimal parameters for setting thresholds.

Furthermore, the current library is missing a Command Line Interface to interact with it. This should be created when a larger portion of the pipeline is finished, to avoid the need to make changes to the Command Line Interface after a function has been updated.

3.3.2 Porting to C and C++

To make the library not only available to Python developers, but also to C and C++ developers, the library has to be ported to C and C++. When porting the library to C and C++, the focus should be on efficient usage of the memory.

Currently the library makes use of some nice features created by the Numpy library. Because the Numpy library is only available in Python, some function should be added to make porting to C possible. However, since the functions used are not complex, rewriting it in C and C++ should be possible.

Bibliography

[1] A Cameron. An investigation on pulsar searching techniques with the fast folding algorithm. 2017.

[2] S Chatterjee. Applications of the fourier transform, 2013.

[3] E Daniel Barr. Searching for pulsars with the effelsberg telescope. 2012.

[4] R Grootjans. Detection of dispersed pulsars in a time series by using a matched filtering approach. 2016.

[5] R James Lyon. Why are pulsars to hard to find? 2016.

[6] P Lazarus. Arecibo pulsar survey using alfa iv mock spectrometer data analysis, survey sensitivity, and the discovery of 40 pulsars. 2015.

[7] D Lorimer. Sigproc - v4.0: Pulsar signal processing programs. 2007.

[8] R Lynch. From the telescope to the collaboratory.

[9] S Ransom. Searching for pulsar with presto.

[10] P Scholz. The repeating fast radio burst frb 121102: multi-wavelength observations and additional bursts. 2016.

[11] A Sclocco. Real-time dedispersion for fast radio transient surveys, using auto tuning on many-core accelerators. 2016.

[12] E van Heerden. Data challenges in pulsar searches. 2017.