ASTRON

FF-Flagger: A New RFI-Mitigation Pipeline For Transient Detection at WSRT.

Masters Thesis

University of Amsterdam

Anton Pannekoek Institute for Astronomy

Author:

Jedda Boyle

Supervisors:

Dr. Joeri van Leeuwen

Dr. Alessio Sclocco

ABSTRACT

A dearth of data is limiting our understanding of fast radio bursts (FRBs). The data rates of modern radio telescopes exceed storage capacity so online signal-processing pipelines that scan sky data in real time searching for FRBs are required. However, this search is hindered by terrestrial radio frequency interference (RFI), a problem which is only getting worse as the radio bandwidth becomes increasingly congested. This thesis proposes a software-based solution to the problem of RFI for the FRB-detection effort at the Westerbork Synthesis Radio Telescope (WSRT). The solution is a novel thresholding-based real-time GPU-accelerated RFI-mitigation pipeline called FF-Flagger. FF-Flagger is shown using WSRT data to reduce the number of false positives and increase the number of true positives while meeting the computational constraints of the transient-detection pipeline. Additionally, it is shown to outperform the current default RFI-mitigation pipeline at WSRT, the AOFlagger, for transient detection.

TABLE OF CONTENTS

Page

Page

1. INTRODUCTION

It is hard not to be awed by gazing at a seemingly tranquil night sky, and our astonishment would only increase if we could see the vigorous hotchpotch lurking just below the surface of what our eyes reveal. Radio telescopes allow us to see below the optically visible universe and this ability has exposed innumerable new scientific mysteries, key among them the recent discovery of fast radio bursts - extremely high-energy events occurring outside the Milky Way that last fractions of a second. This thesis is a contribution towards our use of radio telescopes to better understand fast radio bursts.

The source of fast radio bursts (FRBs) is currently unknown, as is their mechanism. The discovery of fewer than 50 FRBs has been published and this dearth of data is hampering the astronomical community's current efforts to unravel the mystery [1]. Consequently, the hardware and software of radio telescopes around the world, including the Westerbork Synthesis Radio Telescope (WSRT), are now being optimised to scan the sky for new FRBs. These scans are increasingly impaired by terrestrial signal due, on the one hand, to the acute sensitivity of modern radio telescopes and, on the other, to increasingly congested radio bandwidths. This terrestrial noise in the signal detected by radio telescopes is known as radio frequency interference (RFI). Mitigating the deleterious effects of RFI on the FRB-detection effort at WSRT is the focus of this thesis.

The technological progress of radio-telescope hardware is allowing for increasingly sensitive and expansive observations of the sky and as a result the data rates are rapidly growing. Already radio telescopes such as The Low Frequency Array (LOFAR) generate more data in a given amount of time than there is internet traffic

in the whole of The Netherlands and these numbers are only going to increase with next-generation radio telescopes such as the Square Kilometre Array (SKA). These enormous data rates have made the storing of observations no longer feasible so all stages of the data-processing pipeline, including the RFI-mitigation module, need to run in real time.

As part of a transient-detection pipeline designed to discover FRBs, an RFI-mitigation module needs to adhere to two key requirements:

1. The RFI-mitigation module should run in real time and have linear-time complexity to ensure that it scales into the future as the rates of data generated by radio telescopes increase.

2. The RFI-mitigation module should improve the accuracy of the transient-detection pipeline. This means that the number of false positives must be decreased while the number of FRBs detected is not.

In this thesis we propose a new RFI-mitigation pipeline called FF-Flagger and implement it as part of the Apertif Monitor for Burst Encountered in Real-time (Amber) - the real-time transient-detection pipeline being used at WSRT to find FRBs. The FF-Flagger is based on a new thresholding algorithm called edge-thresholding which is also proposed for the first time in this thesis. The FF-Flagger is one of the first RFI-mitigation pipelines specifically designed to work in conjunction with a transient-detection pipeline.

Using data from WSRT, the FF-Flagger is shown to adhere to the two specifications listed above as well as to accommodate memory constraints specific to Amber at WSRT. Currently the default RFI-mitigation pipeline at LOFAR and at WSRT is the AOFlagger. In this thesis we show that the FF-Flagger outperforms the AOFlagger in terms of transient detection.

1.1 Background

1.1.1 Radio Astronomy

Radio astronomy studies the universe by observing it in the radio bandwidth between 3 kHz and 300 GHz of the electromagnetic spectrum - the electromagnetic field propagating through space-time [2]. Radio astronomy is fundamentally no different to standard optical astronomy which also observes the electromagnetic spectrum but in the range of frequencies our eyes and those of most sighted animals are able to detect. The electromagnetic spectrum includes, in order of increasing frequency, radio waves, infrared, visible light, ultraviolet, X-rays, and gamma rays [3].

Radio astronomy was born in 1933 when Karl Jansky published his discovery that the universe emits radio waves that are detectable on the surface of the earth [3]. As with many great discoveries Jansky's detection of radio signals was an accident. He had built a radio antenna to research radio-frequency interference coming from thunderstorms and found an unknown signal that correlated with the rotation of the earth. Once he had ruled out his initial hypothesis, that the interference was coming from the sun, he determined that the Milky Way was the source. By 1937 amateur astronomer Grote Reber, inspired by Jansky's discovery, had made the first purpose-built radio telescope [3].

To understand why it is interesting to study the universe at different bandwidths of the electromagnetic spectrum consider figure 1.1, which shows an image of the Milky Way in (a) radio, (b) infrared, (c) visible, and (d) X-ray wavelengths. As this figure shows, different bandwidths of the electromagnetic spectrum reveal new perspectives on the universe. Most astronomy is done in the radio and visible bandwidths because these waves aren't blocked by the earth's atmosphere and are therefore detectable on the surface of the earth [3]. Observing other bandwidths requires launching telescopes into orbit. NASAs Fermi Space Telescope, for instance, is currently in orbit

detecting gamma radiation and there are even some radio telescopes in orbit, but, unsurprisingly, building and maintaining a telescope in orbit is vastly more expensive and difficult than on the surface of the earth [4] [5]. For this reason most telescopes are terrestrial and they therefore have to observe in the radio or visible bandwidths.

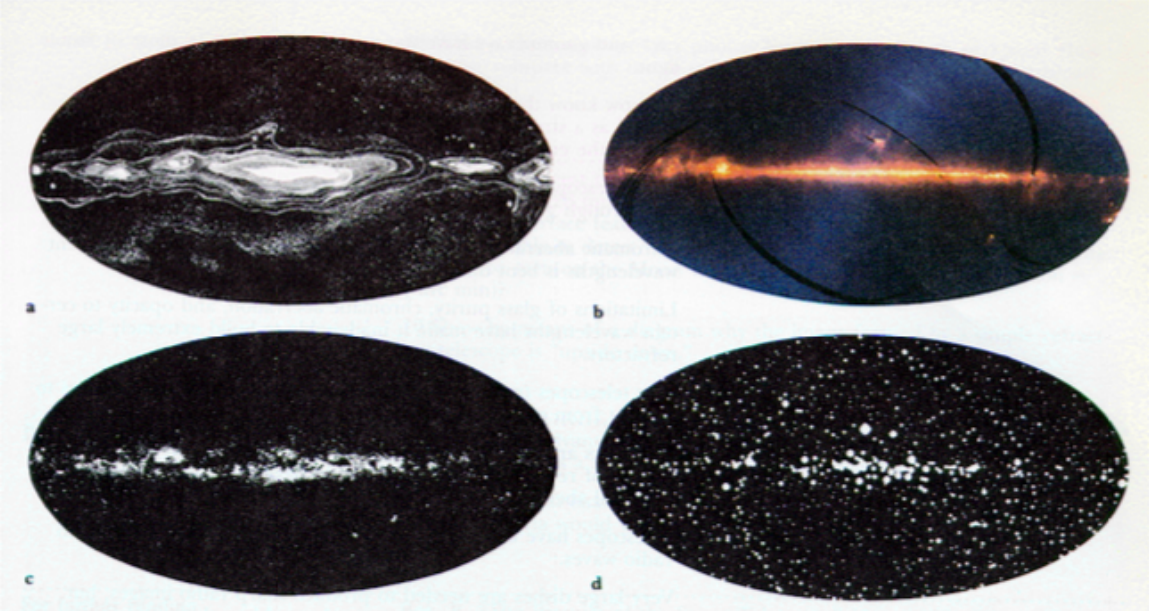

Fig. 1.1. An image of the Milky Way stretching horizontally across the picture in (a) radio, (b) infrared, (c) visible, and (d) X-ray wavelengths. This image was generate from Griffith Observatory and JPL. [3]

.

Radio telescopes come in many different shapes and sizes and serve many different purposes. At one end of the spectrum there is the enormous single-dish five-hundred-metre Aperture Spherical Radio Telescope and at the other end there is the dish-less omnidirectional antennae array used at The Low-Frequency Array (LOFAR) [6] [7]. All these radio telescopes can convert detected electromagnetic radiation into various data artefacts that are useful for radio astronomers.

In this thesis the data used is on one such artefact, the time-frequency intensity array. A time-frequency intensity array is generated by subdividing the radio bandwidth being observed by the radio telescope into smaller bandwidths -the frequency channels. The intensity of the electromagnetic radiation in each of these frequency channels is then recorded at fixed time intervals. The result is a two-dimensional array where the y-dimension records different frequency bandwidths, the x-dimension records different time samples and the data contained within the array is an integer representing the intensity of the electromagnetic radiation in a specific bandwidth at a specific time.

With this cursory understanding of the mechanism generating radio waves and the resulting data format produced by radio telescopes it is now time to consider what is observable.

1.1.2 Transients, Pulsars and Fast Radio Bursts

Mechanisms such as pulsar coherent radio emission populate the observations of radio telescopes with detectable electromagnetic radiation [8]. Sources of this electromagnetic radiation can be broken down into three general groups, namely discrete point sources such as stars, discrete extended sources such as galaxies, and background radiation (partly made up of the remaining glow from the big bang [3]). All of these sources are studied by radio astronomers but in this thesis we focus on fast radio bursts, which are a specific type of transient discrete point source. Transient sources, as the name suggests, are dynamic sources that are observable for brief moments of time.

One of the most studied types of transient sources are pulsars. Pulsars are formed when a star becomes unable to support its own mass and collapses into a nova or supernova. As the resulting gas cloud dissipates a relatively small dense core with a

diameter of several kilometres remains. The resulting pressure is high enough to fuse protons and electrons into neutrons, and a neutron star is born. Neutron stars have extremely powerful magnetic fields and if the axis of rotation does not correspond to the axis of the magnetic field the rotations will create periodic oscillations in the electromagnetic spectrum - much like a galactic lighthouse. These rotating neutron stars are known as pulsars. The first pulsar was discovered in 1967 and since then thousands more have been identified [3].

Measuring the slowly decreasing rotation frequency of pulsars, which rotate up to hundreds of times per second, has helped to confirm aspects of Einsteins theory of general relativity and the study of pulsars more generally has provided insights into the properties of neutron star physics, the galactic gravitational potential and magnetic field, the interstellar medium, celestial mechanics, planetary physics and cosmology [9].

In this thesis our interest in pulsars is their relation to fast radio bursts (FRBs). Introducing FRBs is difficult because nobody knows exactly what they are. We observe them as non-periodic short bursts of electromagnetic radiation where each burst lasts a few milliseconds. Unlike the case with pulsars, we cannot explain how they are formed or what causes them [10]. FRBs are a rare and true scientific mystery.

As was the case with Jansky's discovery in 1933, the detection of FRBs was an accident. In 2007 Duncan Lorimer from the West Virginia University in Morgantown was searching through old data from the Parkes radio telescope in Australia and came across what appeared to be a non-repeating radio burst, similar to a pulsar. Unable to observe a second burst he had only a single data point to validate what he had seen - a process further hindered by a rather embarrassing recreation of a similar signal by the opening and closing of microwaves in the telescope canteen! His observation was subsequently validated as the first detection of an FRB. Only a handful of new

FRBs have been discovered since.

Not knowing the source or mechanisms of FRBs has naturally attracted the intrigue of the astronomical community. Many theories for FRBs have been proposed - for instance, that they are evaporating black holes, colliding neutron stars or enormous magnetic eruptions - but, so far, even the best models fail to account for all the observations [11]. Even a non-periodic repeating FRB has been discovered, ruling out theories assuming the process to be a cataclysmic event [12]. It seems reasonable that no one model will account for all FRBs, especially given that some repeat and some do not, and it is therefore likely that there is more than a single processes behind FRBs.

Another feature, apart from periodicity, that makes FRBs different from pulsars is that they come from a more energetic source, meaning that they emit more radio radiation and therefore are detectable from further away. This means they can be used to answer questions pulsars cant, for example about the nature of the intergalactic medium [11].

To answer the many mysteries of FRBs, we need data about them. To date, we have discovered fewer than 50 FRBs in comparison to thousands of pulsars [1]. This dearth of data has led to both astronomers and telescopes worldwide being allocated to the discovery of more FRBs. This thesis is one part of that effort.

FRBs and pulsars emit broadband radiation for fractions of a second [9]. Before this radiation reaches earth it travels for many light-years through the intergalactic median, which is filled with extremely tenuous plasma. These gases interfere with the wavelengths and cause the longer waves to arrive later than the short ones. As a result FRBs and pulsars show up in a time-frequency intensity array as a broadband-dispersed burst [9]. The degree to which a burst has been dispersed is a value known

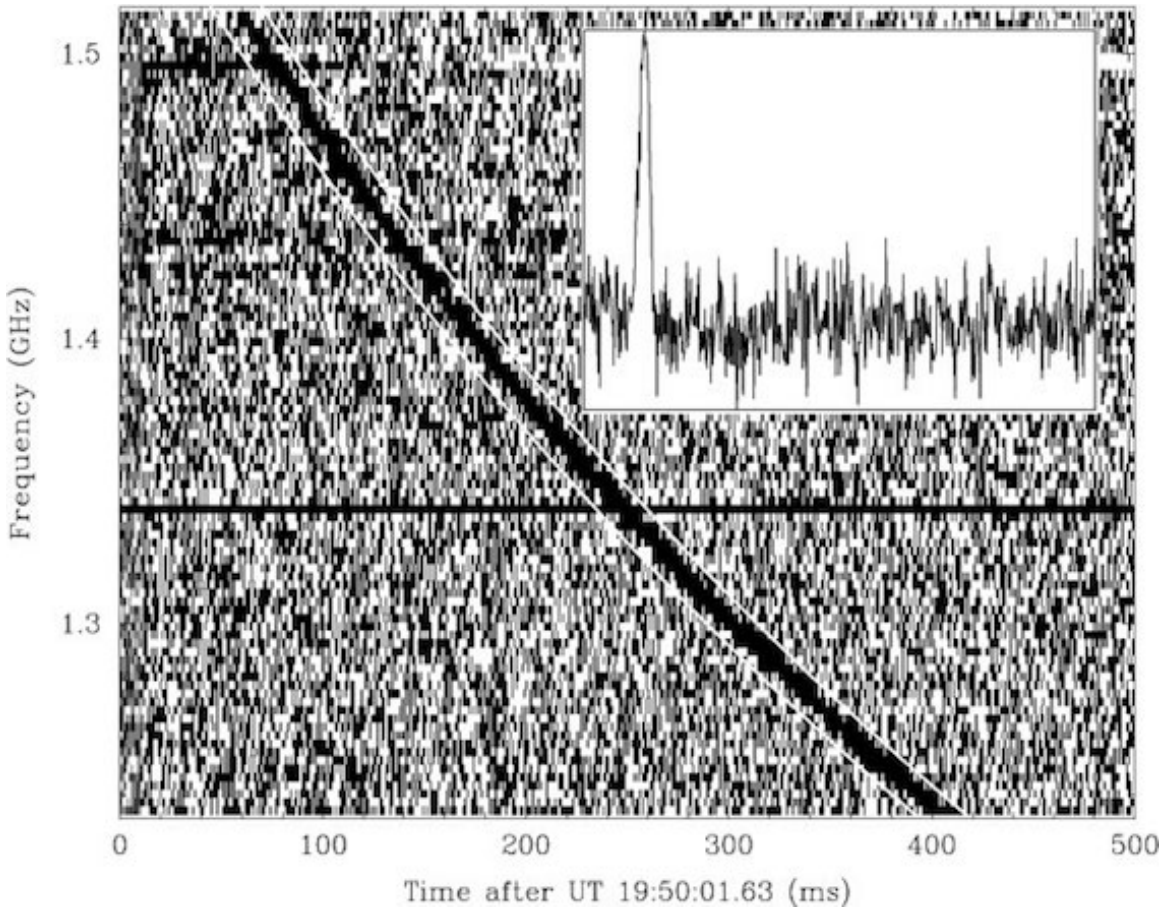

Fig. 1.2. Figure showing the dispersed time-frequency intensity array and de-dispersed pulse profile of the Lorimer burst - the first detected FRB [13].

as the dispersion measure (DM). As an example consider the time-frequency intensity array of the first FRB, discovered by Lorimer, which is shown in figure 1.2. The inset plot shows the summed intensity of the FRB after the data has been de-dispersed to account for the delay caused by travelling through the intergalactic medium.

As we can see in figure 1.2 the FRB is clearly visible and on an almost-Gaussian background but this is unfortunately not always the case. Often the signal is obscured by noise coming from terrestrial sources of radio waves, known as RFI.

1.1.3 Radio Frequency Interference

Unfortunately not all radio-frequency radiation detected by radio telescopes comes from celestial sources. Our planet is full of technologies emitting radio waves that pollute the data recorded by radio telescopes. This terrestrial interference is RFI.

The issue of RFI is a growing concern for radio astronomy because an ever-increasing number of technologies is congesting the radio bandwidth and, at the same time, the heightened sensitivity of radio telescopes is making this congestion more apparent. Sources of RFI are everywhere. Aeroplanes and satellites cause RFI in the sky while sources such as cellphones, FM antennae, microwaves and TV broadcasting produce RFI on the ground. In many cases this terrestrial signal is actually brighter than celestial signal because the inverse square law of propagation - which states that signal dissipates from the source at a quadratic rate with its distance - means that celestial signal is often weak by the time it reaches earth [3].

The best form of RFI mitigation is proactive - when RFI is removed at source. You can build radio telescopes in remote regions where there isn't much interference, for example, or regulation can be used to enforce radio quiet zones around radio telescopes. These proactive measures are effective and are currently an important part

of the RFI-mitigation effort at many radio telescopes but they are not sufficient because avoiding all RFI this way is impossible. Even the radio telescope itself is full of complex electronics and, although these are covered to avoid the worst of their RFI, some remains unavoidable. Radio quiet zones are becoming harder to create as other industries, often with more financial backing, such as wireless communications and data broadcasting are competing for space in the radio bandwidth. Planes and satellites are also unavoidable for a radio telescope in any location - although the location of some are known and observations can be planned accordingly.

Given these limitations of proactive RFI mitigation, it is necessary that RFI-mitigating software is part of a telescopes signal-processing pipeline. To separate celestial signal from terrestrial noise while processing data, a telescopes RFI-mitigation pipelines utilise the fact that signal and noise have a different structure. RFI is usually bright, not dispersed and narrowband or short in time - all features not exhibited by FRBs and other celestial sources.

The efficacy of software-based RFI mitigation is limited by the enormous amount of data rates produced by modern radio telescopes, often making complicated noise-reduction algorithms unfeasible. This thesis therefore concentrates on real-time linear-complexity RFI-mitigation algorithms that use the features of RFI to separate it from the signal in the time-frequency intensity arrays produced by radio telescopes.

1.2 Technical Details

In this section the technical details of the data, software, hardware and science case for the WSRT transient detection effort, required for understanding this thesis, are described.

1.2.1 Westerbork Synthesis Radio Telescope

In this thesis we focus on the FRB-detection efforts at the Westerbork Synthesis Radio Telescope (WSRT) in the northeast of the Netherlands. While all the data, software and results used in this thesis take into account the constraints and requirements of the FRB-detection project at Westerbork the results are also meaningful for other transient-detection projects at telescopes such as the Low-Frequency Array (LOFAR) and the Square Kilometre Array (SKA).

WSRT is an interferometric array made up of 14 dishes each 25-metre in diameter arranged 2.5 kilometres apart on an east-to-west line. In 12 of the dishes the receiver in the focus of the dish is being replaced by an array of receivers known as a Phased Array Feed. The commissioning for this upgrade, which is known as Aperture Tile in Focus (Apertif), is ongoing. The new receivers observe in the bandwidth between 1000-1750 MHz and drastically increase the field of view. [14]. An area of the sky that the telescope is able to view is called a beam and as a result of the Phased Array Feed the 12 upgraded dishes are able to observe 40 beams on the sky instead of just one. See figure 1.3.

Currently, the largest planned radio telescope is the SKA. Once completed, it will include thousands of dishes and more than a million antennae distributed between South Africa and Australia [16]. A key goal of the SKA is to increase the field of view and one of the possible technologies for doing this is a Phased Array Feed such as that being used for the Apertif upgrade. So while Apertif will increase the field of view at WSRT, it also aims to determine the feasibility of using a Phased Array Feed at the SKA.

To fully utilise the benefits of the Apertif upgrade, all the signal processing at WSRT, as well as the RFI mitigation, is done using the The Apertif Radio Transient Sys-

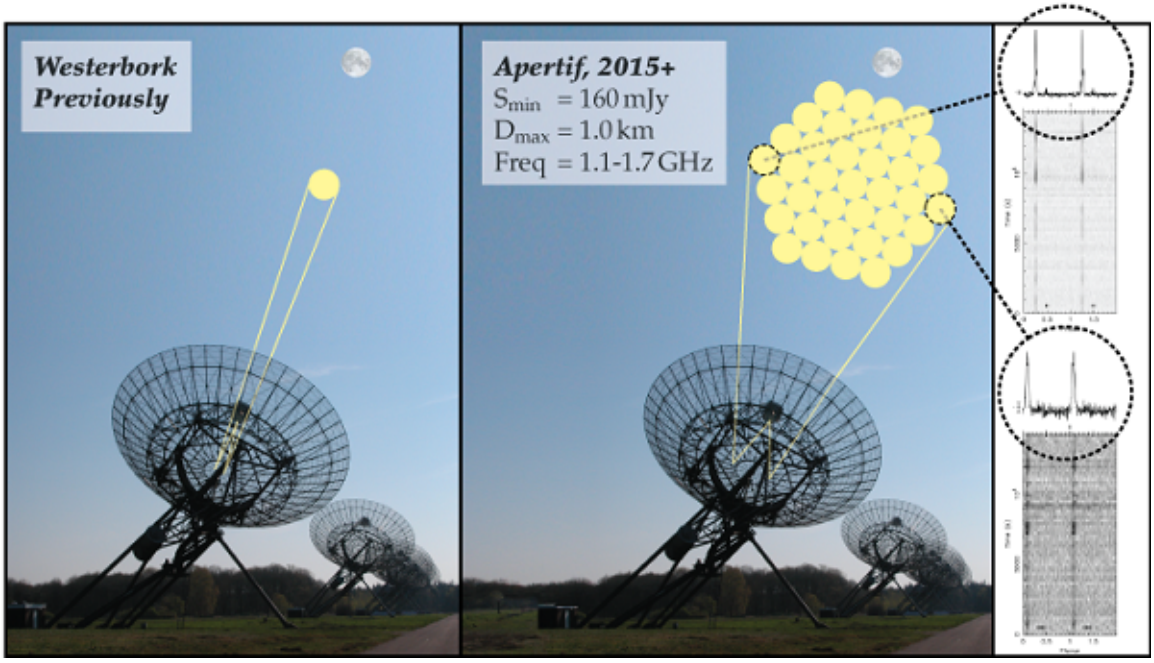

Fig. 1.3. The Westerbork Synthesis Radio Telescope, with a graphical representation of the on-sky beams before and after the Apertif upgrade. [15]

.

tem On Apertif (ARTS). ARTS is a hybrid machine of FPGAs and a 2-petaflop GPU cluster. The transient-detection and RFI-mitigation pipelines run on the GPU cluster [15].

1.2.2 The Data Processing Pipeline at WSRT

In this section an overview of the data flow from the telescope to storage drive is described. The description is based on WSRT and Amber - the transient detection pipeline at WSRT [17]. This is not a full technical specification. The aim is to provide sufficient detail for this thesis while also avoiding minutiae.

An overview of the data flow and software components of the pipeline can be seen

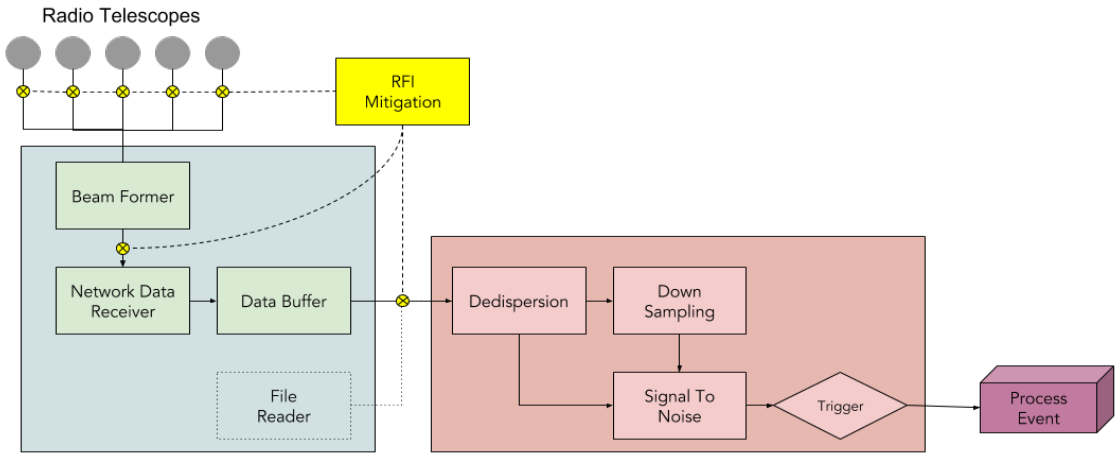

Fig. 1.4. An high-level overview of the data flow at WSRT. The red block represents the stages of the pipeline done by the transient detection software, Amber or Heimdall.

in figure 1.4. The pipeline is divided into three parts. The first, shown in green in figure 1.4, is a data preparation stage that converts the radio signal into a data form usable for the later stages. The second stage, shown in red, is a transient detection pipeline and it searches through the data filtering for celestial signal of interest in an attempt to reduce the data burden on the final stage. The final stage, shown in purple in figure 1.4, is a convolution neural network that determines if the data is from a transient or not. If the data is believed to be from a transient it is saved to disk.

The first component of the data preparation stage is the beam former which receives digitised radio signal for each beam, region of the sky being observed, from the radio telescope's receivers. The data from each beam will arrive at each dish at slightly different times and angles because the dishes at WSRT are sprawled. The beam former accounts for this by performing the inverse transformation to align the data observed by different dishes coming from the same on-sky point or beam. Currently, beam forming is implemented at WSRT using FPGAs.

The formed beams are then sent through the network receiver to the data buffer at the ARTS compute centre which contains the hardware for the data processing. Each new block of data is stored in a first-in-first-out ring buffer that stores a block of sky data that steps forward in time. Each block of sky data from a window of time is known as an event and is forwarded to the next stage of the pipeline.

All of the experiments in this project were done using saved data so the data preparation stage of the pipeline is replaced by a file reader.

The second component of the data pipeline, the transient detection pipeline which is shown in red in figure 1.4, sifts through data in search of events with a high signal-to-noise ratio which could be pulsars or FRBs. Given that radio telescopes such as WSRT produce an enormous amount of data - in some cases in excess of 4 terabytes per second - giving all the data to the convolutional neural network is not feasible. Therefore, the aim of the transient detection pipeline is to look at the data in real time and use some metric to determine if it should be processed further. The strategy is to filter out events using the signal-to-noise ratio. Only events with a signal-to-noise ratio above a certain threshold are forwarded. In this project we focus on the transient detection pipeline Amber which is a GPU accelerated transient detection pipeline. Amber was designed with WSRT in mind but it can be used on other telescopes as well. The FRB detection experiment at WSRT will be using Amber. Another GPU accelerated transient detection pipeline, called Heimdall, is also used in this project at because some components of Amber are still being developed.

The first stage of Amber is de-dispersion. As previously mentioned, the signal detected by the telescope is dispersed in time by the interstellar medium it has travelled through to get to the receiver. The amount of dispersion is determined by the distance of the source and the interstellar medium, neither of which are known by the receiver. Therefore, for each event Amber needs to try thousands of candidate dis-

persion measures and for each dispersion measure Amber computes the de-dispersed data and signal-to-noise ratio [17].

The next step of the transient detection pipeline is to integrate the de-dispersed data in the time dimension by collapsing across multiple time samples. The idea is that if we can add all signal from an FRB into a single bin we will get a more accurate measurement of the signal-to-noise ratio. We want to integrate the time-sample by exactly the number of time-bins the FRB occurs in. Again, as with de-dispersion, this value isn't known beforehand so we try multiple widths and compute the signal-to-noise ratio of each down-sampling width.

For each dispersion measure and time-dimension integration width the signal-to-noise ratio is computed. Amber computes the signal-to-noise ratio by taking the sum of each integrated time sample and computing the difference between the maximum value and the medium value divided by the median absolute deviation of the time sample sums (The median absolute deviation is explained in more detail in a later section). If the signal-to-noise ratios is above the threshold then the event causes Amber to trigger.

Each event that causes Amber to trigger is forwarded to the final stage of the pipeline - a convolutional neural network [18]. The input to the convolutional neural network is the time-frequency intensity array, the de-dispersed pulse profile, the dispersion-measure time array and the multi-beam information. Generating these data products for the convolutional neural network is the bottleneck that restricts how many events can be forwarded from the transient detection pipeline. If the neural network decides with some level of confidence that the event is a transient then the data is written to disk and saved for further analysis.

Setting the threshold for the signal-to-noise ratio at which Amber forwards an event

and setting the threshold at which the convolutional neural network writes data to disk are both balancing acts. If these thresholds are too high we run the risk of throwing away valuable data and if these thresholds are too low the data rates will exceed the hardware capabilities.

1.2.3 WSRT Time-Frequency Intensity Data

The RFI-mitigation pipeline for this thesis is implemented as part of Amber. In this section the data, which is time-frequency intensity arrays, that is the input for Amber from WSRT, is described.

The transient-detection effort at WSRT uses the Apertif upgrade so only the 12 upgraded dishes are used. Each of these has 40 on-sky beams. This means that in total $12 \times 40 = 480$ time-frequency intensity arrays need to be processed by the RFI-mitigation and transient-detection pipelines. Each time-frequency intensity array contains 8-bit unsigned integers from 1536 frequency channels with a sampling time of 40.96 microseconds. Because the RFI-mitigation pipeline is implemented as part of Amber, the Amber configuration currently being used at WSRT is assumed. This configuration processes the data in one-second-long events of 25,000 de-dispersed time samples. De-dispersion creates zero values at the boundaries due to the roll operation. Therefore, to ensure that all the cells in de-dispersed data array have valid non-zero value the de-dispersion is done on a larger window of 43,657 samples. RFI mitigation is done before de-dispersion so the number of samples that need to be processed per second is 43,657, although this number could change for different configurations of Amber.

Putting this all together, we see that for each second of observation the RFI-mitigation pipeline needs to process $1536 \times 43,657$ 8-bit unsigned integers for each on-sky beam

from each telescope dish. In total that is $12 \times 40 \times 1536 \times 43,657$ 8-bit unsigned integers or 32.2 gigabytes of data which needs to be processed in real time.

1.2.4 RFI Environment at WSRT

RFI can come in many different forms from many different sources. However, at WSRT the majority of RFI falls into just two different categories.

Most of the RFI detected at WSRT is either short in time and narrow in frequency or short in time and across all frequency channels. If the narrow bandwidth bursts happen to align in a certain configuration they can cause the transient-detection pipeline to incorrectly trigger and, in the second instance, the broadband RFI can easily be misidentified as a transient with a low dispersion measure. Both forms of RFI can also cause the detection of transients at the wrong dispersion measure which means that the input data artefacts for the convolutional neural network will be generated incorrectly and a transient could be misidentified.

Consider figure 1.5 which shows the time-frequency intensity arrays of three false triggers. These were the three false triggers with the largest signal-to-noise ratio from an observation without any transients. Both of the common types of RFI are visible in these plots.

While figure 1.5 shows just a few examples, they are a representative sample of many of the false triggers detected at WSRT. For this reason the RFI-mitigation algorithms considered for this thesis aim to mitigate these two types of RFI.

1.3 Thesis Specification

The key goal of this thesis is to implement and validate a GPU-accelerated real-time RFI-mitigation pipeline that works as part of the data-processing pipeline, de-

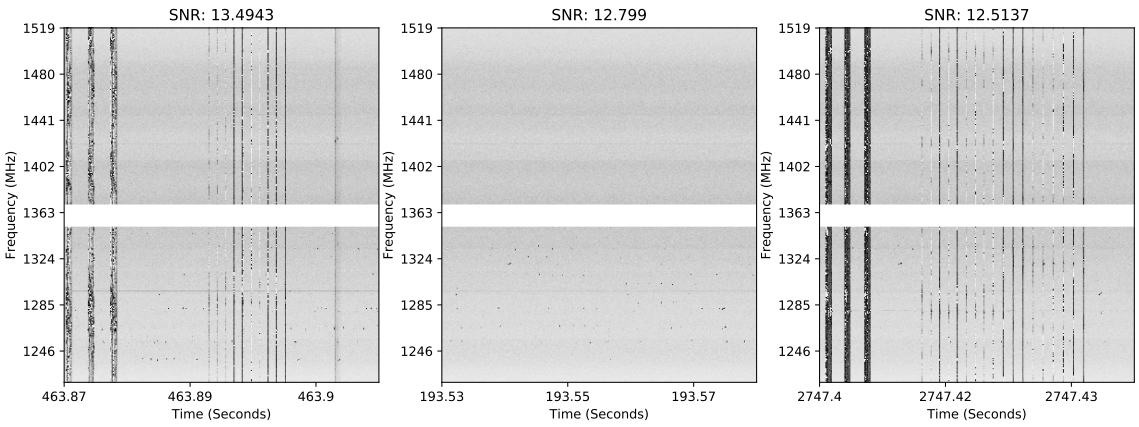

Fig. 1.5. The time-frequency intensity arrays of three high signal-to-noise false triggers observed at WSRT.

scribed in section 1.2.2, at WSRT. The results should also be significant for other transient-detection projects.

As stated in the introduction, all RFI-mitigation implementations for transient-detection pipelines should improve the accuracy of the transient-detection pipeline. This means that the number of false positives must be decreased while the number of FRBs detected is not. It is believed that using ARTS it would be possible to generate the data artefacts required for the convolutional neural network at a rate of a few hundred per hour. This means that RFI mitigation needs to reduce the number of false triggers to under that threshold.

The accuracy of the RFI mitigation needs to be balanced against computational cost. For this thesis, only linear-time complexity algorithms are considered because data rates at radio telescopes are increasing and linear-time complexity helps to ensure that the RFI-mitigation pipeline will scale into the future.

Additionally, at WSRT Amber is currently unable to process the data from all the

12 telescopes dishes due to GPU memory constraints. This problem is currently being worked on but an RFI-mitigation module that uses a significant amount of extra memory would not be viable as using all 12 telescopes beams is necessary.

The third performance requirement is that the RFI-mitigation pipeline runs in real time. The exact run-time of all the other components in the data-processing pipeline is unknown. In this thesis we define the real-time constraints for the RFI-mitigation model as a tenth of a second per second of on-sky data. This number is certainly not exact but seems reasonable.

As explained in sections 1.2.2 and 1.2.3 there is a total of 40 on sky-beams. Each on sky-beam, which consists of 12 time-frequency intensity arrays each of dimension 1536×43657, is processed by one node at ARTS. Each node of ARTS has a 4 Nvidia GeForce GTX 1080 Ti GPUs, one of which is available for the RFI-mitigation. This processing may not take more than 10% of the overall available compute time. Thus the 12 arrays, together containing a second of data, must the processed in 100 milliseconds or less. This means that the real-time constraints per 1536×43657 time-frequency intensity array for this thesis is 8000 microseconds.This is the real-time performance metric we use in this thesis.

Throughout this thesis the performance requirements listed here are taken into consideration to ensure that the end result is a viable RFI-mitigation pipeline.

1.4 Related Work

Researchers have proposed many solutions to RFI, but most are designed for imaging applications. There are very few if any RFI-mitigation algorithms specifically designed for transient detection.

Although there is some overlap between these two application domains, RFI-mitigation algorithms for imaging are not all suitable for transient detection and therefore we cannot simply reuse all of this research. For example, kurtosis and surface-fitting methods proposed in [19] and [20] assume that the signal is smooth - not always true of transients, which can have sharp features. Besides incorrect assumptions, another issue with many imaging RFI-mitigation algorithms is that they are not computationally efficient enough for online data processing pipelines. For example, in [21] convolution neural-networks are proposed and shown to be effective but the data rates at WSRT make it unfeasible to pass all the data through convolutional neural networks without down-sampling - which would tarnish the precision of the RFI mitigation. The morphological detection algorithms such as that proposed in [22] require a RFI mask, also an unfeasible computational requirement for transient detection at WSRT.

Another set of RFI-mitigation algorithms designed for imaging that are not viable for this thesis are those that do not fit into the current data-processing pipeline. For example, in [23] and [24], RFI mitigation is done using a reference antenna. This antenna is tuned to be more sensitive to RFI than the telescope dishes doing the observation. The signal from the reference antenna is transformed in phase and gain to correctly scale the RFI and then the reference data is subtracted from the original data to get a clean data stream [23] [24]. While this approach has been shown in [23] to be effective for pulsar detection , using a reference antenna is not an easy option for RFI mitigation at WSRT because it would require new hardware. At interferometric arrays such as WSRT, RFI-mitigation can also be done at the beam-forming stage as RFI is often only observable in a single beam [25]. This is done by using spatial co-variance matrices between data observed by different dishes to estimate a spatial signature vector that allows for spatial filtering [25]. However, RFI-mitigation at the beam-forming stage is not the focus of this thesis.

The methods we have discussed so far are either too computationally costly, make

incorrect assumptions for transient detection or do not fit into the WSRT data-processing pipeline. Thresholding algorithms, on the other hand, are computationally efficient and ideally suited to being implemented on GPUs as part of the transient-detection software, meaning that they fit perfectly into the data-processing pipeline at WSRT. An example of a thresholding algorithm is the Asynchronous Pulse Blanking algorithm that thresholds the squared values in the data stream based on the mean and standard deviation of the squared values [26]. However, Asynchronous Pulse Blanking is designed for mitigating against RFI pulses and therefore is not suitable for transient detection.

One of the most popular thresholding algorithms for imaging is sum-thresholding, which assumes RFI is more intense over a given data range than signal, an assumption that should hold most of the time for transient detection. Sum-thresholding is also of particular interest to this thesis because it is the key RFI-mitigation algorithm in the AOFlagger which is currently the default RFI-mitigation pipeline at WSRT [27].

2. THEORY

In this chapter the RFI-mitigation algorithms used in this thesis are described along with some discussion of the methods and underlying theory.

2.1 Robust Statistics and Outlier Detection

The data observed at WSRT is mostly Gaussian noise occasionally interrupted by outliers which are either a transient or RFI. RFI mitigation is the task of separating signal from data and performing outlier detection on data which isn't signal.

Effective outlier detection requires being able to compute statistics that are representative of the data distribution and not of the outliers. This can be done by using large data sets with few outliers or by using robust statistics - statistics that are not easily disrupted by outliers. One measure of robustness is a statistics breakdown point defined as how much of a sample can be contaminated by arbitrary data before the statistic becomes unbounded. The mean has a breakdown point of zero percent because only a single data point needs to be replaced to make any arbitrary change to the value of the mean. In comparison the median has a breakdown point of fifty percent.

Outlier detection is a well studied problem but many of the outlier-detection methods that have been shown to perform well - such as clustering, regression, machine-learning and boundary fitting - have non-linear time-complexity and are therefore computationally too costly for our needs [28].

A sigma cut is a simple linear time complexity outlier detection method suited to

one-dimensional data. The mean and standard deviation of the data set is computed and all data points that differ from the mean by more than some threshold number of standard deviations are flagged as outliers [29].

A sigma cut is not a robust outlier-detection method because it relies on the mean and standard deviation, both of which have a breakdown point of zero percent. However, it only requires two passes over the data, one pass to compute the mean and standard deviation and a final pass to flag outliers. Computing the mean and standard deviation can be decomposed into a reduction operation, which is efficient on the GPU. For these reasons it is useful and used throughout this thesis.

Instead of using the standard deviation and mean we can use the more robust median and median absolute deviation. For a data set X the median absolute deviation (MAD) is defined as [30]:

$$\text{Median}(|x_i - \text{Median}(X)|)$$

A data point is then identified as an outlier if it deviates more than some threshold number of MADS from the median.

MAD outlier detection is more robust than a sigma cut but it requires the computation of two medians, which requires sorting which, in turn, generally requires $O(n \log n)$ time and $O(n)$ space if the original data needs to be left untouched. The time complexity and space requirements of sorting limits the applicability of MAD outlier detection for the purposes of this thesis.

2.2 Flagging Broadband Zero DM RFI

A lot of the RFI at WSRT is broadband, as is an FRB, but unlike an FRB, all the signal is in a single or just a few time samples, because RFI is not dispersed. Examples of this type of RFI can be seen in figure 1.5. It is often the cause of false

triggers. In this section we discuss RFI-mitigation strategies for this type of RFI.

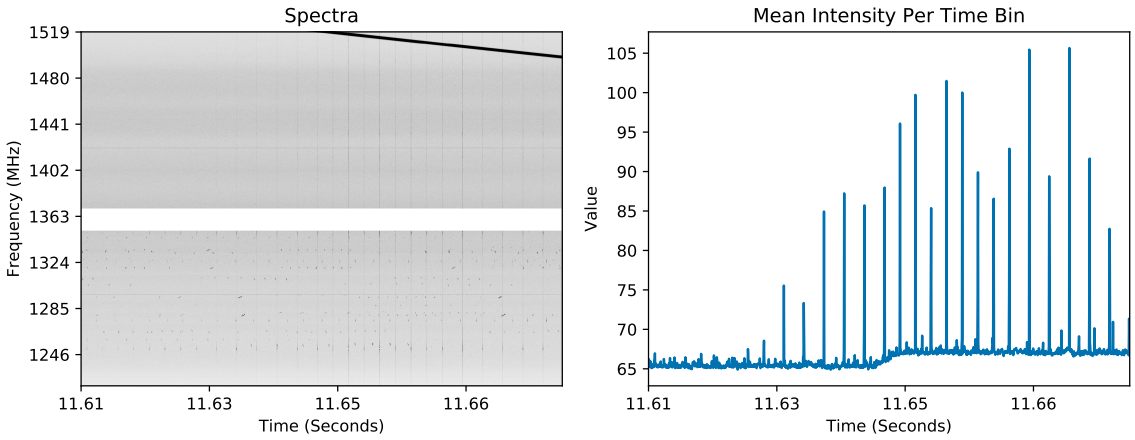

Fig. 2.1. Left: A time-frequency intensity array of an WSRT observation with real RFI and a simulated FRB. Right: The mean intensity of each time-sample.

Removing zero-DM RFI is done by computing the central tendency of the detected values in each time sample and then doing outlier detection on the resulting values. In figure 2.1 an event and the mean value of each time sample in the event is shown. Figure 2.1 makes it clear that this type of RFI mitigation works because some time samples are contaminated with RFI and these samples are easily visible in the mean space. Therefore it makes sense to do outlier detection on the mean values. However, figure 2.1 also indicates some limitations of this method of RFI mitigation. In the right-hand corner of figure 2.1 the edge of a bright FRB can be seen. The FRB increases the mean of the sample which creates a step in the mean values. For this event there is actual RFI which will increase the deviation sufficiently to make the impact of the FRB negligible. However, if the event has no RFI and the FRB has a high intensity with a low DM it is possible that the FRB is identified as RFI and all the time samples where it occurs are flagged. Also some RFI, which can also be observed in figure 2.1, isn't broadband but is intense enough to create a spike in the

mean value. If this is the case then all the frequency channels in the time sample will be flagged as RFI when only a few frequency channels are actual RFI.

The efficacy of doing outlier detection on the central tendencies of time bins is dependent on which measure of central tendency is used.

2.2.1 Measures of Central Tendency

There are many measures of central tendency, and in this thesis we consider the median, mean, trimmed mean and root mean square (RMS). Using pigeonhole sorting (see section 3.2.1) we can efficiently do MAD outlier detection on time samples if the measure of central tendency returns integer values within a small range. The median would fulfil this requirement because all the values are 8-bit integers. Most measures of central tendency result in a float value, which means MAD outlier detection cannot be done in linear time, in which case a sigma cut is used.

The mean is the most obvious choice and is easy to compute. However, using the mean could cause inaccuracies because it is not a robust measure of central tendency which means FRBs or narrowband RFI could cause all the frequency channels in a time sample to be flagged as RFI when only a few frequency channels are actually contaminated. This can be seen in figure 2.1.

A more robust and also easily computable alternative to the mean is the trimmed mean. The trimmed mean is computed by calculating a p-percentile point which is a value for which $p\%$ of the data is less than. The trimmed mean is then the standard mean of all the data points less than the p-percentile point. Using pigeonhole sorting (see section 3.2.1) the p-percentile point of a time sample can be efficiently computed.

Another alternative measure of central tendency, which is used as part of the AOFlag-

ger, is the RMS, which has many applications within electrical engineering and signal processing. The RMS is the square root of the mean of the squared data, which is:

$$\sqrt{\frac{1}{n}\sum_{i}^{n} x_i^2}$$

2.3 Flagging Ephemeral Narrowband RFI

In section 2.2 methods for flagging all the frequency channels in a time samples were discussed but much of the RFI observed at WSRT only pollutes a few frequency channels. For this type of RFI we use methods that look at small windows of data instead of all the frequency channels in a time sample and decide whether the window is RFI or not. This class of RFI-mitigation algorithms is called thresholding algorithms. The AOFlagger uses sum-thresholding for this application and in this thesis a thresholding algorithm called edge-thresholding is proposed.

Thresholding algorithms work by computing a value for each window of data and doing outlier detection on the resulting values. Ideally we would do a sigma cut or MAD outlier detection on the values of all the windows but these are computationally costly operations and storing the results from all the windows isn't feasible because the number of values that need to be stored is a multiple of the window size and the data size. For this reason a window is flagged if its value is greater than some threshold value because this outlier detection method requires no awareness of the other window values.

2.3.1 Sum-Thresholding

Sum-thresholding is an iterative algorithm that runs in linear-time and is well suited to being implemented on the GPU. It is a key RFI-mitigation algorithm in the AOFlagger and has been shown to work well on WSRT data for imaging [27].

For each iteration, sum-thresholding takes a data array and a boolean mask as input along with parameters ω_i and t_i, which are the window size and threshold. The mask labels data points that have already been identified as RFI by an earlier stage of the RFI pipeline or an earlier iteration of sum-thresholding. For every possible window of data with size ω_i the window is flagged as RFI if the mean of the unflagged data points in the window is greater than the threshold [27]. As an example given the following window of data and mask:

$$\text{Data} = [6, 5, 7, 3, 7, 9]$$
$$\text{Mask} = [1, 0, 0, 1, 1, 0]$$

we would have a sum-threshold value of $(5 + 7 + 9)/3 = 7$ and the entire window would be flagged as RFI if t_i was set to be less than 7. Generally speaking we would compare the threshold against the absolute value of the masked window mean but in our scenario all data points are positive so this isn't necessary

The Python code for the sequential linear implementation of sum-thresholding is shown below:

```
"Data and mask are one dimensional arrays."
"w := window size"
"t := threshold"
def sum_threshold(data, mask, w, t):
    temp_mask = np.copy(mask)

    "Sum of valid points in window."
    window_sum = 0

    "Count of valid points in window."
    count = 0

    "Compute count and window_sum for first window."
```

```
    for i in range(w):
        window_sum += data[i] * (1 - mask[i])
        count += (1 - mask[i])

    "Flag first window."
    if (window_sum > t * count):
        temp_mask[:w] = 1

    "Slide window down the data."
    for i in range(w, len(data) - w):
        window_sum += data[i + w] *
                      (1 - mask[i + w])
        count += (1 - mask[i + w])

        window_sum -= data[i] * (1 - mask[i])
        count -= (1 - mask[i])

        "Flag window"
        if (window_sum > t * count):
            temp_mask[i:i + w] = 1

    mask = temp_mask
```

For any given window size, the code is $O(n)$. Therefore, in sum-thresholding's purest formulation - when it runs on all window sizes less than n - we would have an $O(n^2)$ algorithm. While this is probably good for RFI detection it is not feasible for the WSRT application that is focused on in this thesis. By restricting the maximum window size to a fixed value less than n and doing a fixed number I we get a linear-time complexity algorithm with computational complexity $O(Wn)$ where W is the

maximum window size. Therefore using large window sizes is quite feasible for the sequential implementation.

In [27] sum-thresholding works by picking a maximum number of iterations and doubling the window size for each iteration giving us $w_i = 2^i$. Doubling the window size is done to ensure that large windows are tested for RFI without too much computational overhead. It is also proposed in [27] that the threshold is decreased for larger window sizes so in this thesis the threshold T_i is $(\alpha \times \text{median})/i$ where the median is over the entire data set.

2.3.2 Edge-Thresholding

Edge-thresholding is proposed for the first time in this thesis and is one of the first thresholding algorithms designed specifically for transient detection. As is the case with sum-thresholding, edge-thresholding is an iterative algorithm reliant on processing the data in windows of increasing size. Unlike sum-thresholding, edge-thresholding takes into consideration the structure of the signal and of the RFI in an attempt to more accurately mitigate RFI.

Sum-thresholding works by labelling a window of data as RFI if the window's masked mean is above some threshold. Edge-thresholding works by labelling a window of data as RFI if the window's edge-filter is above some threshold. Given a window of data $x = (x_0, ..., x_{\omega-1})$ with window size ω the edge-filter is a function defined as:

$$F(x) = \min\left(|f(x) - x_0|, |f(x) - x_\omega|\right) \tag{2.1}$$

where f is any function measuring the central tendency of the window x. A natural choice for f would be the mean or median. The edge-filter is an approximate metric function of the difference between the window and its boundary values.

In essence edge-thresholding is thresholding on the deviation between values and

therefore it's natural to use some number of standard deviations or median absolute deviations as the threshold. Given that the MAD is more robust and it is efficiently compatible using pigeon-hole sort (see section 3.2.1) this thesis proposes flagging windows whose edge-filtered value is greater than some threshold number of median absolute deviations. The python implementation is shown below:

```
"Data and mask are one dimensional array."
"w := window size"
"t := threshold"
def median_edge_threshold(data, mask, w, t):

  "Compute MAD."
    median = np.median(data)
    mad = np.median(np.abs(data - median))

    "Slide window down the data."
    for i in range(len(data)):

            "Compute window statistic"
            window_stat = np.median(data[i:i + w])

            "Compute edge-filter value"
            t_value = min(abs(window_stat - data[i - 1]),
                          abs(window_stat - data[i + w]))

            "Threshold edge-filter value"
            if t_value > 1.4826 * t * mad:
                mask[i:i + w] = 1
```

Edge-thresholding can also be done without a measure of centrality. For example we can threshold the minimum absolute difference of each value in the window with the boundary values. We call this pointwise edge-thresholding. It tries to mitigate the deleterious effects of outliers in the window which could cause big windows to be flagged as RFI when only a single or a few points are actually RFI. This could also be done with more robust statistics but they are usually more expensive to compute. To make pointwise edge-thresholding more concrete consider the Python implementation:

```
"Data and mask are one dimensional array."
"w := window size"
"t := threshold"
def pointwise_edge_threshold(data, mask, w, t):

     "Compute MAD."
    median = np.median(data)
    mad = np.median(np.abs(data - median))

    for i in range(1, len(data) - w):

        "Loop of each value in the window."
        for j in range(i, i + window_size):

            "Compute edge-filter value"
            t_value = min(abs(data[j] - data[i - 1]),
                          abs(data[j] - data[i + w]))

            "Threshold edge-filter value"
            if t_value > 1.4826 *  t * mad:
                mask[j] = 1
```

2.3.3 Discussion and Comparison of Thresholding Methods

Edge-thresholding and sum-thresholding are similar but they make slightly different assumptions about the nature of signal and of noise. When trying to separate signal from noise it is useful to consider differences in their structure and so to determine what should be thresholded.

Sum-thresholding makes the assumption that signal is on average less intense than noise. In most cases this assumption holds, especially for imaging applications but it is not always true for transient detection. Many transients, and FRBs in particular, are generated by enormous amounts of energy and are therefore in some cases the most intense feature in an event. In these cases sum-thresholding will flag either the signal and RFI or neither, depending on the threshold.

Instead of intensity, edge-thresholding takes advantage of two other features that signal exhibits but RFI doesn't. The first is width. Most detected RFI at WSRT is extremely short in time. Some examples are less than a single 40-microsecond time sample long. Second, signal has a pseudo Gaussian profile due to finite frequency resolution of the telescope which causes DM smearing. This is particularly noticeable in WSRT data because the frequency channels in the data received by the RFI-mitigation pipeline have larger bandwidths than those actually detected by the telescope, due to binning of frequency channels before the RFI mitigation. For these reasons the edge-filtering operation filters for narrow peaks in the data. Given these assumptions, edge-thresholding is susceptible to flagging short, bright, low DM FRBs, depending on the threshold and window size.

The performance of edge-thresholding is dependent on which measure of centrality is used for each window. Obviously the computational cost is greater if we use the median or some other robust statistic rather than the mean. When the mean is used

for edge-thresholding the computational cost is almost identical to sum-thresholding, the only difference being that edge-thresholding requires more operations to calculate the threshold equality. To make this clear consider that the Python implementation of sum-thresholding in section 2.3.1 could be converted to a mean edge-thresholding implementation by changing only the equality checking the threshold.

The major resource difference between these formulations of sum-thresholding and edge-thresholding is the fact that sum-thresholding masks the input and edge-thresholding does not. Edge-thresholding would probably be more accurate at identifying RFI if it masked its input based on RFI detected in previous iterations or by earlier stages of the pipeline, especially when using non-robust statistics such as the mean. Without a mask, a single large value could cause an entire window to be flagged as RFI even if it was the only outlier value. Using a mask should mitigate this problem because the outlier value would have already been flagged by an earlier iteration.

Similarly, without an input mask, edge-thresholding could incorrectly flag a window as RFI if the window spans between RFI-contained data. In this case the window would have a low mean value because its contents are RFI free but the window could still have a large edge-filter value if the edges, which the window is compared against, are RFI. Again, a mask would help with this, if the edge-values would already have been masked as RFI or excised by an earlier iteration.

Despite these benefits of having a mask, edge-thresholding without one is proposed in this thesis because the additional memory and computational requirements of maintaining a mask make it infeasible for an RFI-mitigation pipeline at WSRT.

2.4 Morphological Detection

All of the RFI-mitigation algorithms discussed so far have required a threshold parameter which needs to be set high enough that FRBs and other transients are not incorrectly flagged. This means that some RFI is likely to slip under the threshold and be missed. One solution to this problem, using the assumption that RFI is usually non-uniformly distributed, is to extend the mask and flag samples that are sufficiently close to regions of strong RFI. This approach is known as morphological detection.

2.4.1 Dilation

The simplest approach to morphological detection is dilation of the RFI mask [31] - for every flag in the RFI mask you just extend the mask to also include a box of some size around the flagged sample. In [31] a box of size 5 was shown to be effective.

Dilation has the advantage of being computationally efficient and does not require a mask if the dilation operation is done at the moment of RFI detection. Despite these benefits dilation suffers from being inaccurate and it isn't scale invariant, meaning that it doesn't generalise to changes in the input data granularity or changes in the size of RFI features.

2.4.2 Scale Invariant Rank Operator

The scale-invariant rank (SIR) operator is a one-dimensional morphological dilation technique that changes the size of the dilation based on the flag density in a region of the mask [22]. The dilation size is increased when the mask is dense and decreased when it is not. The SIR operator solves both issues of dilation - namely, that it is inaccurate and not scale invariant.

The SIR operator takes a mask as input and flags all contiguous windows of data

in the mask such that the ratio of flagged samples in the window is less than or equal to a threshold called the density ratio threshold. The density ratio threshold impacts the aggressiveness of the SIR operator. If the density ratio threshold is one then all samples are flagged and if it is zero then no additional samples are flagged. A naive implementation of the SIR operator would require $O(n^2)$ time because the flag density of all windows needs to be computed and checked against the density ratio threshold. However, in [22] an exact linear-time complexity algorithm was proposed (see section 3.4.1).

Having a linear-time complexity implementation of the SIR operator is useful because it allows the RFI pipeline to maintain the linear-time complexity requirement. However, the linear-time complexity method requires $O(3n)$ additional memory. As discussed in section 1.2.3 the total data rate of the RFI-mitigation pipeline per second is 32.2 gigabytes of data so $O(3n)$ would require more than 100 gigabytes of data which is clearly not feasible even with the 40 GPUs of the ARTS cluster. Even though this number could probably be reduced, the memory requirements would still not be feasible. On top of this, the linear algorithm requires a prefix-sum operation which can be efficiently optimised on the GPU but also requires additional memory.

The SIR operator clearly is not ideal for the RFI-mitigation pipeline at WSRT - the method being either $O(n^2)$ with no additional memory or $O(n)$ with too much additional memory.

2.5 The AOFlagger

The AOFlagger is an iterative RFI-mitigation pipeline developed for imaging applications at WSRT and LOFAR [20]. It is the default RFI-mitigation pipeline at both of these telescopes. However, the AOFlagger has not yet been tested for the application of transient detection and it has not been fully ported to a GPU although

some initial attempts have been made.

An overview of the AOFlagger is shown in figure 2.2. Each iteration of the AOFlagger contains a sum-thresholding operation in time and frequency followed by a sigma cut of the RMS values of time samples and frequency channels and a two-dimensional Gaussian high-pass filter operation. The Gaussian high-pass filter operation is executed to subtract sharp edges caused by strong sources. This is done because the signal is assumed to be smooth and the RFI to be sharp. The time-sample sigma cut, which is labelled as 'mask bad channels/times' in figure 2.2, includes the mask from previous iterations of sum-thresholding into the RMS calculation. At the beginning of each iteration sum-thresholding is executed with a larger window size and lower threshold than the previous iteration. In this thesis the window size is increased at rate $\omega_i = 2^i$ and the threshold is decreased at rate $t_i = \alpha/i$ where i is the loop counter and α is an aggressiveness parameter. The algorithm also includes a re-sampling step before and after the Gaussian high-pass filter because this operation was shown not to require full resolution and after down-sampling the computational cost was reduced. Finally, once all iterations are complete the output mask is forwarded through the SIR operator before being returned to the next stage of the data processing pipeline.

2.6 RFI Excision Methods

Mitigate the negative effects of RFI in radio telescope data requires identifying the RFI, which has been the focus of the previous sections. This section discusses what to do with the RFI once it has been identified.

Many RFI-mitigation pipelines and algorithms maintain a mask data structure that records the locations of the RFI-contaminated data. Using the mask has the advantage that no data is lost since the original data is left untouched. Furthermore, you can analyse the mask to get a better understanding of the RFI environment, which

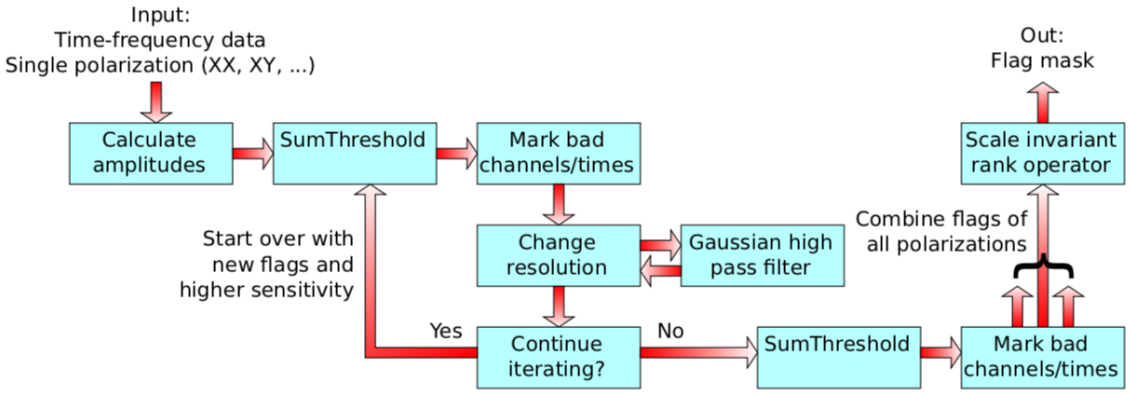

Fig. 2.2. An overview of the AOFlagger's RFI-mitigation pipeline [27].

is important from an astronomical perspective and useful for parameter tuning of the RFI mitigation. Once you have a mask of the RFI-contaminated data it can be forwarded along with the original data to the next stages of the pipeline. Forwarding the mask and the data rather than just the masked data gives more flexibility and information to the subsequent stages of the detection pipeline.

However, using a mask also has various major drawbacks. For one, it creates a dependency between the RFI-mitigation module and the rest of the pipeline. To make an RFI-mitigation pipeline more portable the pipeline should excise the identified RFI before forwarding the data. This is necessary for an RFI-mitigation pipeline to be compatible with Amber, since Amber is not developed to work with a mask.

Given that forwarding the RFI mask clearly isn't feasible, a decision needs to be made about how to excise the data - replacing the RFI-contaminated data with new values. Using zeros seems natural and is computationally efficient but zeros can affect the signal-to-noise statistics. For instance, zeros will impact the median and the median absolute deviation statistics used in Ambers signal-to-noise calculation. Other constant values are also computationally efficient but its difficult to choose a good value and the effectiveness of a chosen value can even vary during the course of a single observation. So instead, some value correlated to the data around the RFI contaminated point is usually used. For example, Heimdall replaces RFI with a random value from the surrounding data points. However, this suffers from needing the costly generation of many random numbers and uncoalesced memory accesses.

For this thesis, replacing RFI with the median of the frequency channel or with the mean of the event were considered. We found that using the median of each frequency channel is the more effective option. In one case, after RFI mitigation Heimdall triggered incorrectly 1477 times when using the mean and only triggered incorrectly 624 times when using the median of each frequency channel which is approximately a 50%

reduction in the false positive trigger rate. In both instances the same RFI mask was used. Using the median of frequency channels also has the philosophical advantage that the data being added is actual data. Also, using the methods described in section 3.2.1, computing the median of frequency channels can be done efficiently in linear time.

For this thesis using a mask to excise data has other problems - the biggest being the additional memory a mask would require. As it is, Amber at WSRT already has GPU memory constraints that prevent it from running on the 12 telescope dish beams. An additional memory block for a mask would only exacerbate this problem. Additionally, constantly reading and writing to a mask, plus the final read of the mask required to excise the RFI, would add to the memory bottleneck. In some experiments it was found that maintaining a mask uses more than half of the RFI-mitigation modules compute time.

For these reasons it is not feasible for the RFI-mitigation pipeline in this thesis to maintain a mask. Therefore, RFI needs to be excised the moment it is detected. Without precaution this can create data races since the pipeline runs with multiple active threads at a time.

3. IMPLEMENTATION DETAILS

In this section we describe the implementations of the RFI-mitigation algorithms described in section 2. All of the host code in this section was implemented in C++ and the device code was implemented using OpenCL. This choice was made because this is the same software stack used to develop Amber.

The performance results discussed in this section were executed on the The Distributed ASCI Supercomputer 5 and all code was run on a Pascal generation Nvidia GTX TitanX. The kernels were run 5 times to warm up the GPU and then the average execution time over a 1000 subsequent runs of the kernel with randomised input data is reported.

3.1 Time Sample Sigma Cut

In this section the implementation of the mean time-sample sigma cut used in this thesis is described. A mean time-sample sigma cut works by computing the mean and standard deviation of the time-sample means and then flagging time-samples whose means values deviate more than some threshold number of standard deviations from the mean of the time-sample means.

3.1.1 Mean, Standard Deviation and Reduce

The mean and standard deviation are crucial for doing a sigma cut of the time samples and therefore being able to compute these statistics efficiently is crucial for the pipelines overall performance. Given that both of these operations require summing over a set of data points, an efficient sum reduce kernel is required.

To compute the mean we do a sum reduce over the data and divide by the length of the data. Ideally we would have liked to compute both the mean and standard deviation of the data by computing the mean and the square mean in a single pass. However, this method resulted in numerical instabilities and required an extra temporary data array to do two reduces in parallel. Instead, the mean is computed in a single pass and the standard deviation is calculated as $\sum_{i=1}^{n} \frac{(x_i-\mu)^2}{n-1}$ in a second pass. To avoid numerical instabilities with this method it is important that the division on each data point was done before the summation to prevent the values becoming too large. This two-pass method requires the same amount of compute but more global memory accesses. If the two pass solution becomes too expensive there are numerically stable ways to compute the mean and standard deviation in a single pass, as is done in Amber.

We start by focusing on a standard one-dimensional reduce used to compute the mean and standard deviation of time-sample means, so that we can flag time samples with a sigma cut. (The pipeline also requires a column reduce to calculate the mean of time samples but we focus on this in a later section.) Three one-dimensional reduces were implemented for this thesis.

The first - a naive reduce - uses a single work group. The work group steps forward along the data where each thread's step size is the size of the work group and each thread starts at the index of its thread index within the work group. As the work group slides along the data, each thread performs a local reduce in shared local memory. Once the work group has looped over all the data the local shared memory contains partially reduced results. A final reduce is done on the shared local memory using a tree approach (explained in next paragraph). This implementation benefits from a small parallel overhead since there is only a single kernel call and from there being no barriers while iterating over the global memory. Although barriers are

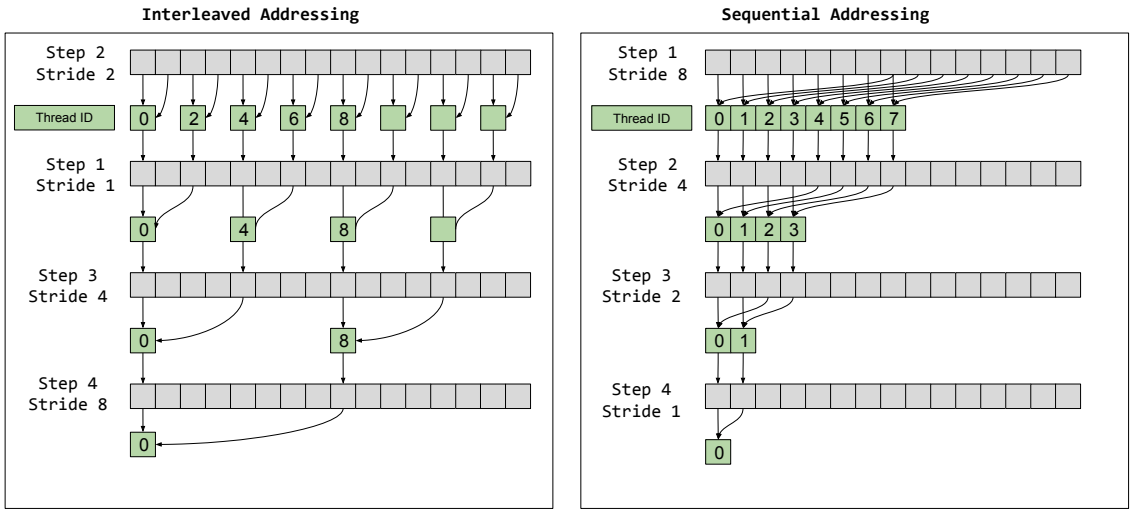

Fig. 3.1. Graphical overview of the local shared memory reduce patterns of two tree-based reduce implementations.

needed for the reduce in shared local memory if the number of threads in the work group is more than a warp. But the clear disadvantage of this method is that the parallelism of the GPU is not being fully utilised by a single work group.

To increase the parallelism of the reduce, we use a tree approach as described in [32]. The idea is to give each work group a segment of the data to reduce. The partially reduced result from each work group is then reduced further by another work group as the next step. Because there is no global barrier between work groups each level of the reduce tree requires a new kernel call. Each layer of the tree has the same logic so the kernel calls are iterative. The depth of the reduce tree is logarithmic with respect to the number of work groups. Two tree-based reduces were implemented in this project. A graphical representation of these methods is shown in figure 3.1.

A tree-based reduce contains two key components. The first is reading global data into shared local memory and the second is doing the reduce in the shared local data.

First let's focus on the reduce in shared local data.

The reduce in shared local data is iterative. During each loop, threads do a single addition between two partially reduced values from earlier iterations. The order of these additions and the allocation of work to threads can have a big impact on the performance of the algorithm.

Interleaved addressing - where each thread combines its own and its neighbouring partially reduced results - is the natural way to do this reduction. A graphical representation of this is shown in figure 3.1. The stride s grows at rate 2^i where i is the loop index starting at zero. Therefore at each iteration the number of partially reduced results and the number of active threads halves. This means that the reduction requires $\log_2$ iterations.

Interleaved addressing, as shown in figure 3.1, presents a few issues. Firstly, at each iteration every thread with an index that does not divide the stride is inactive. This means that many of the threads in a warp do nothing. In the worst case, where the stride is greater than 32, only a single thread in a warp is doing computation. To solve this problem we need contiguous allocation of active warps where all the threads with a global index below some number are active and all threads above this number are inactive. If this is the case we have at most one warp that contains active and inactive threads, which means the minimum number possible of inactive threads are wastefully run using processor resources.

In addition, we want contiguous threads to access contiguous shared local memory locations to avoid shared local memory-bank conflicts - where two threads in the same half warp are simultaneously trying to access different memory locations in the same shared local memory bank. For example, consider solving the contiguous active thread allocation problem by assigning each thread to the partially reduced value at

index $2s \times \text{tid}$ where tid is the local thread index. This results in contiguous active threads but each thread will need to access local memory bank $2s \times \text{tid} \mod 32$ because GPUs have 32 shared local memory banks. Memory transactions are processed in 16-thread half warps so if $s = 2$ thread 0 and 8, both of which are in the same half warp, will need access to the same shared local memory bank, creating a shared local memory-bank conflict.

Sequential memory addressing, as shown in figure 3.1, solves both the inactive thread and the shared local memory-bank conflict issues by having contiguous local memory accesses and contiguous active thread allocation.

To further optimise our tree-based reduce we can also do the first reduction step when reading from global memory, thus halving the number of threads required.

Also, each reduction step requires a barrier - unless the number of values being reduced is fewer than than 32, in which case this is done within a warp. Therefore you can have a separate loop without a barrier when the number of values being reduced is less than 32.

If we know the work group size beforehand we can unroll the reduction loops so that no unnecessary computation is done. We found that before unrolling loops the best performance was achieved with 128 threads per work group so we decided to unroll the loops accordingly.

In figure 3.2 the performance of the three reduce implementations is shown. All the results in this plot have been normalised against the basic tree reduction implementation, which used interleaved addressing to do the reduction on shared local memory. The naive reduce, as described above, uses a single work group sliding along the data while keeping partial results in shared local memory. The optimised tree reduce does

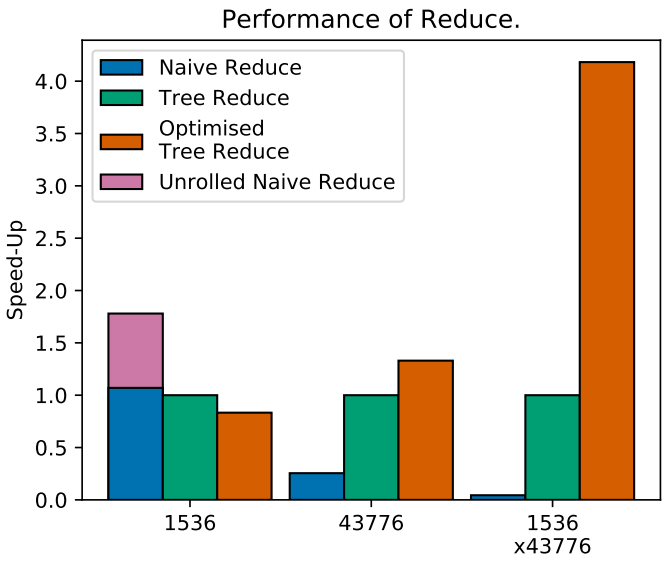

Fig. 3.2. Performance of the different reduce implementations for each of the relevant input data dimensions.

the initial reduce step while accessing global memory, uses sequential addressing, the local memory reduction is unrolled and the final warp level reduction is done without barriers. An RFI event has 1536 frequency channels and 43776 time samples so the plot shows the performance of the reduce on all the relevant input sizes that could be needed given these event dimensions.

The results in figure 3.2 show that the basic tree reduce is never the optimal method to use. For larger inputs the optimised tree-based implementation was up to 94 times faster than the naive implementation and four times faster than the basic tree implementation. For doing reduces over the frequency channels, the basic naive implementation is faster than both the tree-based implementations. This speed-up is almost double if we unroll the reduction on shared local memory. This is because the naive implementation has a lower parallel overhead and when only reducing 1536 values the method's lack of parallelism isn't deleterious to the performance. This is an important result because computing the mean of the time samples requires reducing over the frequency channels, which is most efficiently done using this simple naive kernel.

3.1.2 Computing the Mean of Time Samples

Doing RFI mitigation on time samples using measures of central tendency - mean, trimmed mean and RMS - is an important part of the RFI pipeline proposed in this thesis. In section 4.1 it is shown that using the RMS were not optimal. So we need efficient algorithms to compute the mean and trimmed mean of time samples. The data orientation of an event has the frequency channels along the first dimension and time samples along the second dimension. This means that we need efficient algorithms that work on matrix columns.

Generally doing operations on matrix columns results in bad memory-access pat-

terns. A common solution to this problem is to perform a transpose on the data to allocate it more efficiently. Once we've transposed the data we can perform a reduction on the rows by extending the naive reduce implementation described in section 3.1.1 into a two-dimensional one. This means each work group is assigned a data row. Given that the reduce is over 1536 frequency channels this is the most efficient reduce operation. We can also use the methods described in section 3.2.1 to compute the nth percentile and so compute the trimmed mean. Since GPU transpose is computationally efficient this approach results in a fast matrix-column mean and trimmed-mean operation. However, standard efficient GPU transpose operations require a duplication of memory.

As discussed the memory constraints on the GPU are already incredibly tight and storing a duplication of the data for a transpose of each beam's data is just not feasible. This means we need to implement an efficient in-place GPU transpose. For square matrices this is quite efficient because each work group can be assigned a block location and the transpose of that block location. The transpose of these two blocks can be done by the work group in shared local memory, avoiding the need to duplicate the data in global memory. However, the time-frequency intensity arrays at WSRT aren't square so we would need a rectangular in-place GPU transpose. With a non-square matrix, the data points that need to be swapped result in irregular memory-access patterns that can result in terrible performance. Solution to the access problems on non-square matrix transpose have been proposed in [].

Given the inefficient memory-access patterns of non-square in-place GPU transpose, an alternative solution is to avoid the transpose and compute the mean and trimmed mean on the columns of a matrix. However, on columns we cannot efficiently use the methods in section 3.2.1 to compute a nth percentile which means the trimmed mean isn't feasible without a non-square in-place GPU transpose. For this reason only a column mean operation is implemented for time-sample sigma cutting.

The implementation to compute the means of a matrix's columns is as follows. Divide the matrix into two-dimensional tiles and assign a work group to each tile. The work group computes the partial sum reduce of the tiles columns in shared local memory. Once these partial results are complete, the results in the shared local memory are written to global memory. Doing the partial reduce in shared local memory avoids atomic write conflicts on the global output data.

The size and therefore number of tiles can dramatically affect the performance of column-mean reduce. Having too few tiles results in too much work per work group. Since multiple work groups can be active at a time, the writes to global memory need to be atomic and therefore having too many tiles can result in atomic write collisions.

This algorithm has many factors that influence its performance and it is therefore a good example of a situation where automatic tuning would be beneficial to the performance of the pipeline. In our experimentation we found that tiles of dimension 128×1024 and work groups of size 4×256 were optimal for a standard event of size 1536×43657. With these parameters it took 838 microseconds to compute the mean of all 43657 time samples.

The fully optimised kernel using the unrolled naive reduce described above for computing row means took 738 microseconds. From a performance perspective, this means we would need a non-square in-place GPU transpose that takes less than 50 microseconds to transpose a matrix of 1536×43657 for this to be more efficient. This would also make using truncated-means feasible.

Given that the mean is shown in section 4.1 to perform well and that a non-square in-place GPU transpose is complicated to implement and has inefficient memory-access

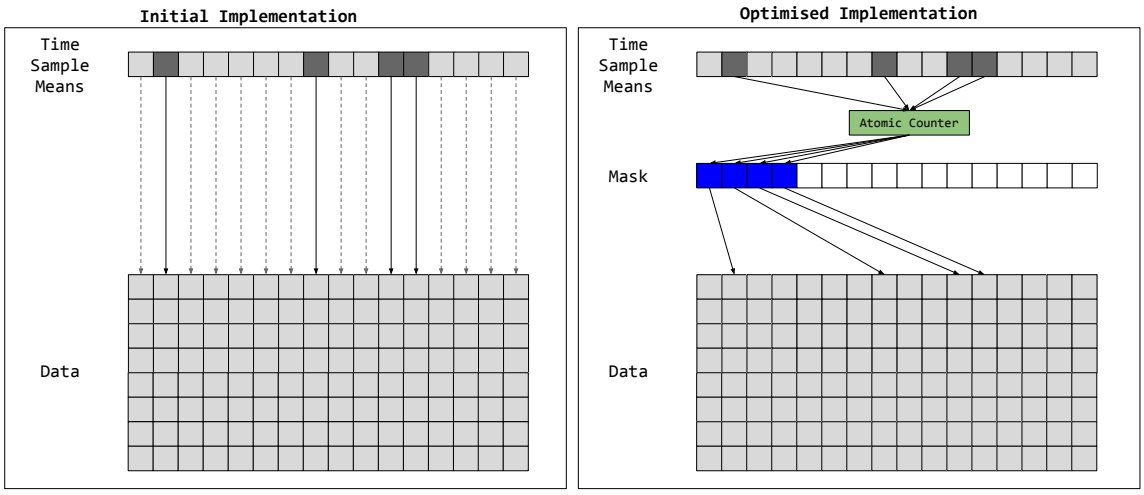

Fig. 3.3. Overview of two time-sample sigma cut RFI excision implementations.

patterns, computing the mean directly on columns was used for the pipeline proposed in this thesis.

3.1.3 Sigma Cut and Time Sample Excision

After calculating the column means, we can use the optimised tree-based reduce implementation described in section 3.1.1 to compute the mean and standard deviation of the column means. Every time sample whose mean value deviates from the mean of the time-sample means more than a threshold number of standard deviations is flagged as RFI.

In the left-hand plot of figure 3.3 an overview of the initial implementation of the sigma cut is shown. Every time-sample mean is assigned a thread. Each thread checks to see if the absolute mean mean difference is above the threshold. If it is, the thread continues to excise the column of data associated with that time sample. The advantage of this method is that no additional data is required for a mask. The

disadvantage is that the flagged time samples are generally more sparse than 32 and therefore most of the data excision is done by an entire warp with only a single active thread. Consequently, the performance is really bad and it takes 12275 microseconds to process a single event of size 1536×43657.

The performance of this initial implementation isn't acceptable and therefore an alternative solution that uses more memory is implemented. A graphical representation of this is shown in the right-hand plot of figure 3.3. The solution is to have a mask that stores the indices of the flagged samples. Access to the mask is controlled by an atomic counter which is used by threads to get the next valid write location in the mask. The atomic counter also records the number of valid values in the mask. A second kernel is then launched that reads the mask and excises the data. This method ran in 1549 microseconds, which is roughly an eightfold speed-up. The second implementation is faster than the first because it doesn't waste threads in the excision step. The second method also has the advantage that you can do multiple flagging iterations over the time-sample means and then only do a single excision step at the end. This is useful if multiple sigma-cut iterations are done.

The disadvantage with the second implementation is that the atomic counter can cause threads to block if there is a write collision. However, the low density of flagged time samples means that this is not a big issue. A more limiting factor for this method is the size of the mask. As we know, memory is scarce on the GPU. In general only a couple hundred of the 43657 time samples are flagged so the memory for the mask doesn't need to be that big but any mask size less than the number of time samples could result in the mask overflowing. Currently the mask is 43657 integers big but, to be more memory efficient, a smaller mask with a procedure for the rare case when the mask overflows would be more memory efficient.

3.2 Thresholding Methods

For this thesis a GPU accelerated implementation of both Sum-Thresholding and Edge-Threshold is implemented. Both implementations described in this section do thresholding along the rows of the input data. Sum-Thresholding requires a median of the row for the threshold and edge-thresholding requires a MAD for the threshold.

3.2.1 Computing Medians and MADs

This section describes how to compute the median and the median absolute deviation (MAD) of frequency channels in linear time without using any additional global memory. Both the median and MAD are robust statistics that are used for the thresholds of sum-thresholding and edge-thresholding as well as for RFI excision.

Computing the MAD requires computing two medians. In general, computing a median requires $O(n \log n)$ time and $O(n)$ additional memory to leave the original data unaltered due to the dependency of computing the median on sorting. To avoid this problem we can use a feature of the data format at WSRT and at radio other telescopes.

As already mentioned, every data point at WSRT is a single byte representing an unsigned integer. This means that there are in total 256 possible values. Sorting an array of these data points can be done in linear $O(n + 256)$ time by using pigeonhole sorting, which works as follows. If we have k possible integers, an array of size k - known as the pigeonhole array - is initialised to zeros. Once this is done the algorithm loops over the data to be sorted and the value of each data point is used to index into the pigeonhole array and increment the value stored in the location by one. The result is a pigeonhole array that contains the count of each value occuring in the data being sorted. Looping over this pigeonhole array and keeping track of the cumulative sum returns the median, which is the index of the pigeonhole array where the cumulative

sum is half the size of the original data.

Pigeonhole sorting has time complexity $O(n + k)$ and is therefore efficient when k is small, as is the case for us. The downside of pigeonhole sorting is that it doesn't parallelise well due to the bottleneck of multiple threads needing to write to the pigeonhole array at the same time. Therefore, it is a more efficient algorithm for computing the median of frequency channels, rather than of an entire event, because each frequency channel will have its own pigeonhole array resulting in fewer write collisions.

The implementation in this thesis assigns multiple GPU threads to each frequency channel to optimise the cache line data sharing between threads. The number of threads per frequency channel is a balancing act between data sharing and minimising atomic write collisions resulting from many threads trying to access the pigeonhole array at the same time. The pigeonhole array is stored in shared local memory and therefore this method requires no additional global memory. Within a work group, the maximum number of frequency channels computed at a time is limited by the size of shared local memory because each frequency channel requires 256×4 bytes of memory for the pigeonhole array. This data could be reduced by using 16-bit integer shorts because the maximum sum of each pigeonhole is 43776, the number of time samples, which is less than 2^{16}. However, using 16-bit integer shorts might result in more shared-memory-bank conflicts because the width of each shared memory bank is 32-bits or 64-bits depending on the GPU.

3.2.2 Edge-Thresholding

For this thesis, both mean and pointwise edge-thresholding were implemented. Four optimisations were applied to each version. Mean edge-thresholding was implemented because it is computationally efficient. Pointwise edge-thresholding was

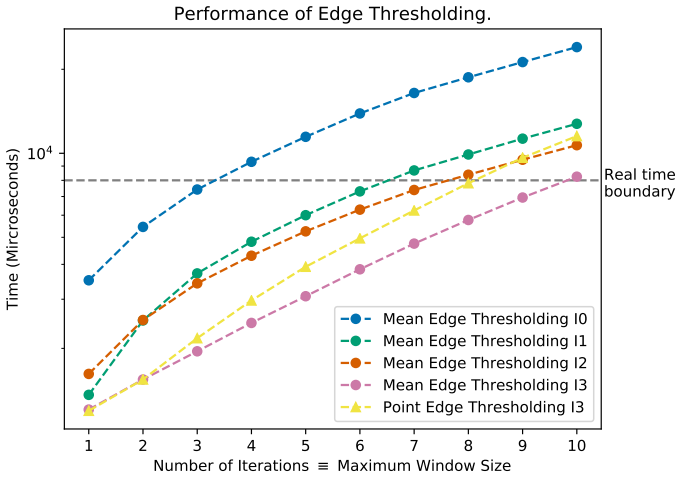

Fig. 3.4. Performance of mean and pointwise edge-thresholding on a log scale plot with the real-time performance boundary for 4 different optimisations.

implemented to compensate for mean edge-thresholdings inaccuracy - due to the mean being a non-robust statistic - and for the lack, in both instances, of an input mask. A third option, median edge-thresholding, could also have compensated for these limitations, but it was found to be too computationally costly and so pointwise edge-thresholding was chosen. The performance results are shown in figure 3.4.

The first implementation of mean edge-thresholding assigns a thread to each data point. This thread is responsible for calculating the edge-filter of all windows that begin at this data point and thresholding each window. Since adjacent threads share data, each thread first loads its data point into shared local memory. If the threads allocation is close to the edge of the work group, it also loads a boundary value. At most every thread does two transfers from global into shared memory. After a barrier assuring that all shared local memory is valid, edge-thresholding can be done with that shared local memory. The performance of this implementation is plotted in figure 3.4 as implementation zero.

Implementation zero's performance suffered from the number of threads being launched and from the relatively small amount of work allocated to each thread. The overhead from launching 1536×43776 threads used more than a quarter of the entire run time. The reason for having so many threads is the 43776 time samples. To reduce the number of threads our next implementation - implementation one - launches a single work group of threads for the entire frequency channel. This work group steps forward in step sizes equal to the dimension of the work group. At every step, every thread within the data range is associated with a data point. As in implementation zero, each thread is responsible for doing edge-thresholding on every window that starts with the data point it is currently assigned. When the thread is done with its current data point, it steps forward, rather than returning as it does in implementation zero. As we can see in figure 3.4 implementation one improved the performance of edge-thresholding.

Implementation two is identical to implementation one except that at every step the block of threads reads the global data into shared local memory where the edge-thresholding can be done. Again this is useful because adjacent threads share data. As we can see in figure 3.4 using shared local memory improves performance for all window sizes greater than two. For a window size of one there is less data sharing between adjacent threads which means data sharing cannot offset the overhead of loading global memory into shared local memory.

The final optimisation, plotted in figure 3.4 as implementation three, involves unrolling the inner loop that computes the mean for each window size. This loop is executed multiple times for every data point which means there is a big gain in performance. Additionally the number of loop iterations is generally low so not an excessive number of loop iterations need be unrolled.

Figure 3.4 also plots the performance of pointwise edge-thresholding with all the

four optimisations implemented. Pointwise edge-thresholding performs slightly less well than mean edge-thresholding because the threshold condition needs to be checked for every value in the window rather than just once for the entire window, as is the case with mean edge-thresholding. This is particularly costly because computing the threshold condition requires two absolute value operations and a single minimum operation. This explains why the performance of pointwise edge-thresholding gets worse compared to mean edge-thresholding as the window size increases because it requires relatively more threshold conditions need to be checked as the window size increases.

These optimisations of edge-thresholding made the algorithm approximately three times faster for all numbers of iterations shown in figure 3.4. This speed-up is crucial for the efficacy of edge-thresholding given how much time the operation uses relative to the time constraints. After the optimisations the maximum number of iterations possible with respect to our real-time constraints went from three to nine and in the time that implementation zero does one iteration implementation three is able to do six.

3.3 Sum-Thresholding

A GPU-accelerated sum-thresholding implementation is delivered as part of this thesis. Sum-thresholding and edge-thresholding are similar algorithms and the same algorithmic structure is used for the implementation of sum-thresholding as is used in section 3.2.2 for the implementation of edge-thresholding. The difference with this implementation of sum-thresholding is that it requires an input mask, which makes its performance slightly worse than mean edge-thresholding because there is some additional memory overhead.

3.4 Morphological Detection

In this thesis the SIR operator is the only morphological detection method implemented as a separate RFI-mitigation algorithm. An implementation of dilation that extended the number of values written by the edge-thresholding operation to the output mask was experimented with but it was found to be too time consuming because it resulted in edge-thresholding needing to make many more writes to global memory.

3.4.1 Scale Invariant Rank Operator

In this section the linear implementation of the SIR operator as proposed in [22] is described. The SIR operator is the only RFI-mitigation algorithm dicsussed in this thesis that is not implemented on the GPU. The reason for this is that, as discussed in section 2.4.2, the only linear SIR operator algorithm has additional memory requirements that make it not viable.

The python implementation of the linear SIR is shown below. The algorithm clearly requires $O(3n)$ additional memory and a prefix sum operation which also requires additional memory if implemented on the GPU. For these reason the SIR operator was implemented in C++ and memory is copied to and from the device when the function is called. This is clearly not feasible for the final RFI-mitigation pipeline but it is sufficient for experimenting with the astronomical performance of the SIR operator.

```
"mask is one dimensional"
    "1 :=  flagged sample"
    "0 :=  clear sample"
"eta := density ratio threshold"
"n := mask length"
def sir_operator(mask, eta, n)
```

```
partial_sum = np.zeros(n)
partial_min_idx = np.zeros(n)
partial_max_idx = np.zeros(n)

partial_sum[0] = eta - 1 + mask[0]
for i in range(1, n - 1):
    partial_sum[i + 1] = partial_sum[i]
                        + eta - 1 + mask[i]
"Calculate partial min index of partial sum"
for i in range(1, n):
    partial_min_idx[i] = partial_min_idx[i - 1]
    if partial_sum[partial_min_idx[i]] > partial_sum[i]:
                partial_min_idx[i] = i;

"Calculate partial max index of partial sum"
partial_max_idx[-1] = n;
for i in range(n - 1, -1, -1):
    partial_max_idx[i] = partial_max_idx[i + 1]
    if partial_sum[partial_max_idx[i]] < partial_sum[i + 1]):
        partial_max_idx[i] = i + 1

    "Use partial sum and indexes to calculate SIR output mask."
for i in range(0, n):
    if partial_sum[partial_max_idx[i]] >=
    partial_sum[partial_min_idx[i]] :
        mask[i] = 1
    else:
        mask[i] = 0;
```

4. EXPERIMENTATION

In this section the RFI mitigation algorithms described in the theory section (section 2) are run on real sky data observed at WSRT. Simulated FRBs are injected into clean data using the ARTS-analysis software tool-kit [33]. To include real signal an observation of the Crab Pulsar is also used.

During the course of this thesis Amber was undergoing commissioning and feature changes. For this reason Heimdall is the transient-detection pipeline used in this section because it is currently stable.

A transient is considered to be found if:

1. Heimdall triggers within a tenth of a second of the FRB's arrival time. This time window allows for some discrepancies in the detected arrival time due to de-dispersion.

2. The trigger's DM has a relative error to FRB's actual DM which is less than 20 percent. The approximately correct DM is required because an incorrect DM will cause the de-dispersed data artefacts forwarded to the convolutional neural network to be incorrect which could cause the classifier to miss identify the FRB.

The number of triggers reported are the number of triggers after clustering. The clustering algorithm simply picks the trigger with the highest signal to noise in a 100 millisecond window to prevent the same event causing multiple triggers.

4.1 Zero DM RFI Excision

In this section we investigate the effectiveness of doing outlier detection on time sample measure of centrality. This RFI-mitigation strategy is designed to flag zero DM broadband RFI. As discussed in section 2.2.1 the measures of centrality considered for this thesis are the mean, trimmed mean and RMS values. A sigma cut is used to detect outliers - values that diverge more than a threshold number of standard deviations from the mean are excised. All experiments were done with a 2.5 sigma threshold and the truncation was set to ignore values above the 90th percentile.

4.1.1 Trigger Reduction

Determining the impact each of these three RFI mitigation methods have on the number of false triggers is a good place to start. The experiments in this section were done on 10 minutes of sky data observed from 02:23:27 on the morning of 04-07-2018 in direction RA23DEC58. This data is known to contain no signal and therefore all triggers are false triggers. Heimdall triggered 214 times on the uncleaned data, which is a rate over 1200 triggers in an hour. This rate is much more than the process-able rate of a few hundred triggers in an hour. So without RFI mitigation it would be impossible to process this file in real-time.

The left-hand plot of figure 4.1 shows the impact on the number of false triggers after a single pass over the data by each of the algorithms. The key observation is that all three methods reduce the number of false triggers. The methods also change the distribution of the signal-to-noise ratio of these triggers. Because there is no actual signal in the data the signal-to-noise ratio is a measure of the RFI. Therefore, if only fragments of the RFI are excised it makes sense that some high signal-to-noise triggers will have a lower signal-to-noise ratio after RFI mitigation. This explains why the number of triggers with a signal-to-noise ratio less than 20 increases.

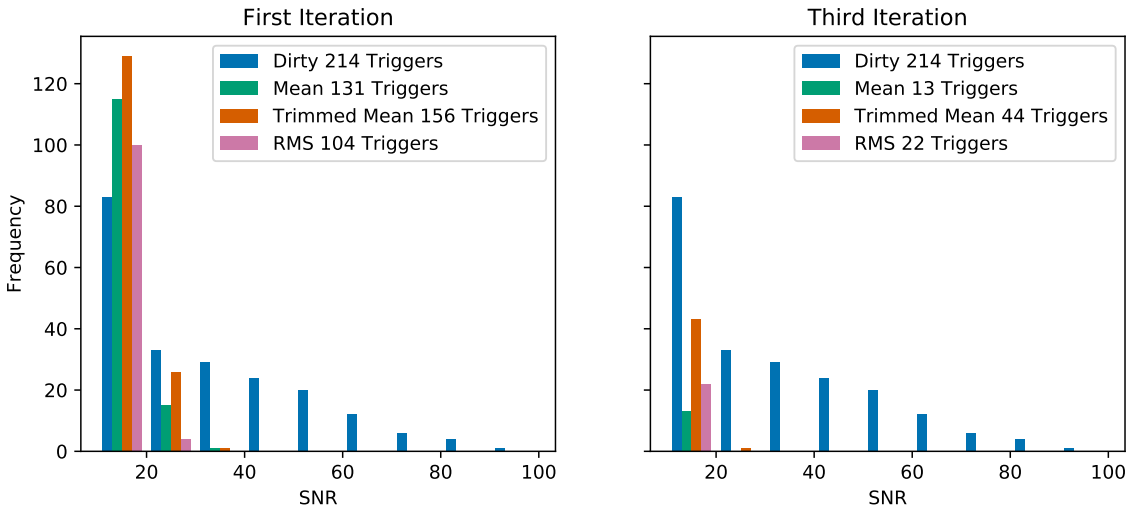

Fig. 4.1. The number of false triggers with a signal-to-noise ratio above 10 after zero DM RFI excision from 10 minutes of sky data observed in direction RA23DEC58.

This downwards shift in the signal-to-noise distribution suggests that more iterations might shift some of the triggers even further down and crucially below the signal-to-noise threshold. The right-hand plot of figure 4.1 shows the false triggers after three iterations. This plot makes it clear that multiple iterations increase the trigger reduction. In this experiment it was found that the fourth iteration made no tangible difference to the number of triggers. The optimal number of iterations will depend on the data but the key point is that more than one iteration can be effective.

Using the mean reduced the number of false triggers by 94 percent - which was the most effective of the four measures of centrality that we tried. For all three methods the false trigger rate dropped below 300 triggers per hour, making it feasible to process the observation in real-time. One caveat to these results is that the observation is not representative of all the observations that are made at WSRT. Some observations aren't as contaminated by this type of RFI, but these results prove that these methods can be effective at reducing the number of false triggers.

4.1.2 Transient Detection After Zero DM RFI Excision

While reducing false triggers is crucial, this should not be at the cost of obscuring FRBs. To test the impact of time sample mitigating we simulated low DM FRBs into real sky data. These tests were done with low DM events because low DM signal is similar to the RFI we are mitigating and therefore more likely to be obscured. We tested with DM 25, 50, 75 and 100. The signal-to-noise ratios calculated by Heimdall of each FRB before and after cleaning is plotted in figure 4.2.

We want the FRBs in the clean files to have equal or higher signal-to-noise ratios than in the dirty files. The straight line plotted in figure 4.2 shows this relation and therefore points above the line indicate that the RFI mitigation is working. Fortunately, the trimmed mean and RMS both seem not to affect the signal-to-noise ratios, while

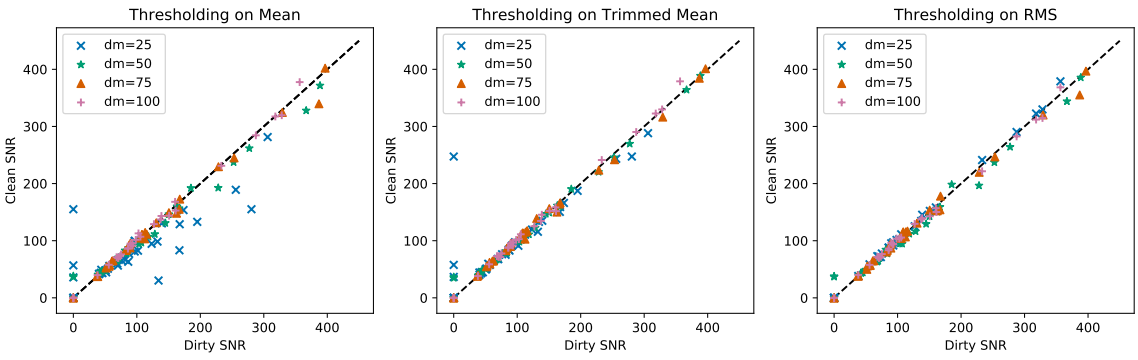

Fig. 4.2. The signal-to-noise ratio of 150 simulated FRBs before and after zero DM RFI excision.

the mean method appears most likely to reduce the signal-to-noise ratio of FRBs, especially for DM 25 FRBs.

If an FRB is not detected in one file it is plotted in figure 4.2 as having zero signal-to-noise in that file. This means that points along the y-axis are all FRBs that were not detected before cleaning. This is usually due to the fact that the RFI causes Heimdall to incorrectly identify the DM of the event. Both the mean and trimmed mean methods actually increase the number of FRBs detected. These newly detected FRBs mostly have DM 25 or sometimes DM 50. This tells us that in this experiment low DM FRBs are more likely to be obscured by RFI. This makes sense given that low DM FRBs occur in fewer time samples and this means that every contaminated sample is obscuring a relatively larger portion of the signal. Figure 4.2 also shows that there are no FRBs obscured by the RFI mitigation - which is crucial.

As a final proof that using measures of centrality to mitigate time samples does not obscure signal we ran these methods on real data from the Crab pulsar, a bright low-DM source and therefore ideal for the test. In the observation the Crab pulsar pulsed 366 times of which 359 were detected in the dirty file. After cleaning with all three methods the number of detected pulses was 360. So, even with real signal these RFI mitigation methods obscured no pulses and slightly improved the detection accuracy.

4.1.3 Resilience to High Signal-to-Noise Transients

In figure 4.2 the reduced signal-to-noise ratio of the DM 25 FRBs after mitigating time samples using the mean suggests that at some point a low DM FRB will become bright enough for these methods to obscure it. To test this, an FRB with fixed DM but increasing intensity was repeatedly injected into the same block of sky data. The impact of the RFI mitigation algorithm on the signal-to-noise ratio before and after

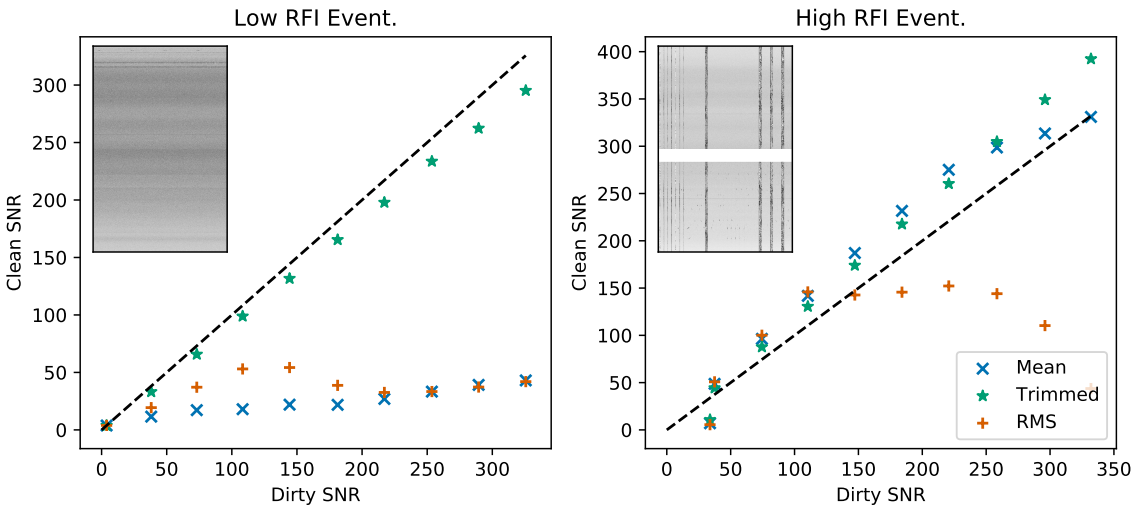

Fig. 4.3. Signal-to-noise of a fixed DM FRB before and after zero DM RFI excision where the FRBs intensity is being increased. The inset plot shows a slice of the event the FRB is injected into.

cleaning was measured as the injected FRB's brightness was increased. The results can be seen in figure 4.3. The inset plots in figure 4.3 show a slice of the event the FRB was injected into. For these plots the signal-to-noise ratio was calculated as the difference between the maximum and median value divided by the MAD of the total intensity of each time sample after de-dispersion without down-sampling. This means that the signal-to-noise ratios in figure 4.3 are not directly comparable to the signal-to-noise ratio that Heimdall returns because Heimdall uses a different algorithm to compute the signal-to-noise ratio. Nevertheless these are useful plots because they make it possible to compare the sensitivity of the different measures of centrality to bright low DM FRBs.

In both plots of figure 4.3 we want the points to be on or above the straight line. This indicates that the RFI mitigation isn't reducing the signal-to-noise ratio of the transient.

The left-hand plot of figure 4.3 shows the results when the FRB is injected into an event with no visible RFI. Its clear that as the injected signal-to-noise ratio increases both the mean and RMS methods decrease the signal-to-noise ratio in the clean file. Meanwhile, the trimmed mean tends more closely to the straight line. This indicates that the trimmed mean is less likely to obscure an FRB than the other methods in a low RFI event.

In the right-hand plot of figure 4.3, which shows the scenario when the event has a lot of RFI, we see that the trimmed mean method improves the SNR. At first the RMS method improves the signal-to-noise ratio but this quickly changes as the intensity of the injected FRB is increased. The same pattern appears for the mean method but the point at which the signal-to-noise ratio starts to be decreased is at a higher injected signal-to-noise ratio. Again the trimmed mean method performs best.

4.1.4 Conclusions

The results in this section show that outlier detection using any of the three measures of centrality is an effective RFI mitigation strategy. These methods reduce the number of triggers and either increase or maintain the number of detected FRBs. The RMS method did not have any advantage over the other two methods. Its trigger reduction was worse than mean and its sensitivity to bright FRBs was worse than both other methods. The mean method reduced the most number of false triggers but was also most likely to obscure an FRB. In contrast the trimmed mean method was best at avoiding FRBs but also worst at removing false triggers. However, the trimmed mean has the advantage that it is parameterisable in the sense that the level of truncation can be changed to balance between false triggers and FRB obfuscation depending on the RFI environment.

4.2 Thresholding

In this section experiments to determine the astronomical precision of both sum and edge-thresholding for transient detection are done on real WSRT sky-data and simulated FRBs.

4.2.1 Edge-Thresholding A single Frequency Channel

As this is the first formulation of edge-thresholding, we start by showing here how it works - analysing what edge-thresholding does to a single frequency channel.

A time-frequency intensity array is shown in the left-hand plot of figure 4.4. An FRB and some narrowband RFI are clearly visible. The right-hand plot of figure 4.4 shows the 1325 MHz frequency channel of the time-frequency intensity array. In the single channel the RFI is visible as spikes and the FRB shows up as a pseudo-Gaussian hump.

The right-hand plot of figure 4.4 also shows the result of the edge-filter operation (see section 2.3.2) using two different window sizes. When the window size is three, the edge-filtered data still contains visible artefacts from the RFI but not from the FRB. This is due to the width and smooth shape of the FRB, which makes no FRB value significantly different from its neighbours. In contrast, the RFI is narrow and sharp ,which means RFI values do differ from their neighbours, causing edge-filtering to return a large value. This ability of edge-filtering to separate RFI and FRBs is the key to edge-thresholdings success because it makes it possible to threshold in a space where only RFI is visible. However, as is also shown in figure 4.4 with $\omega = 100$, if the window size becomes too large relative to the size of the FRB, edge-filtering will no longer avoid the FRB. This shows that edge-thresholding can flag data regions smaller than the window size. Therefore edge-thresholding is susceptible to flagging FRBs that are narrower than the window size.

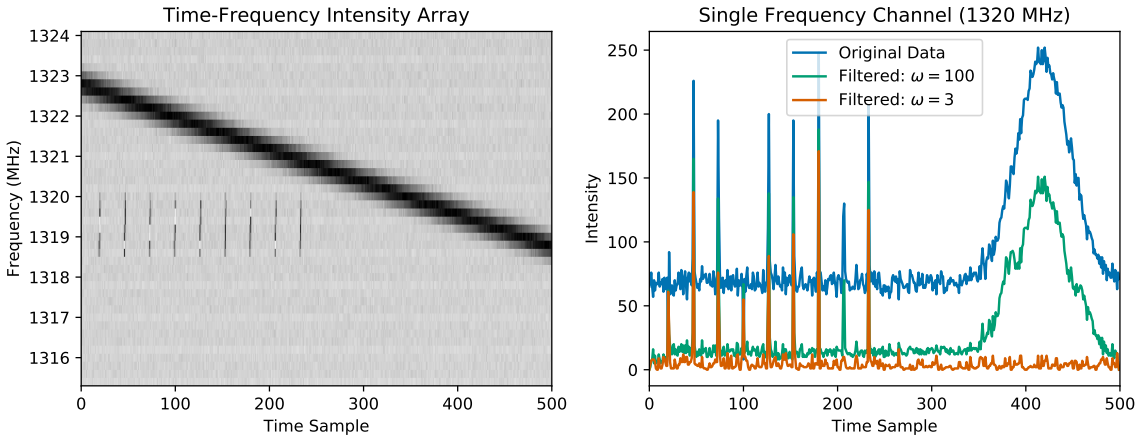

Fig. 4.4. The edge-filter result of various window sizes on a frequency channel that includes RFI and an FRB.

Using a too-large window size for edge-filtering can incorrectly flag signal but, conversely, using a too-small window size can cause edge-thresholding to miss RFI. An example of this is shown in figure 4.5. Both spikes visible in the left-hand plot of figure 4.5 are RFI and both are wider than four time samples. As a result, when the edge-filter window size is four both spikes are missed by the filter. If the window size is increased to eight both spikes are detected because their width is less than eight. However, when the window size is increased to 10 the process again becomes inaccurate - which can be seen as the black dashed line in figure 4.5. Whats happened in this case is that the window is so large it spans two RFI peaks. Because edge-thresholding filters on the difference between the window's internal and edge data it can incorrectly flag clean data when - as in this example - the internal data is clean and the edge data is RFI. Ideally we would have an input mask giving us the information from earlier iterations when smaller window sizes were used. In these earlier iterations the RFI peaks would have been masked and therefore this inaccuracy wouldn't occur. But as discussed previously, an input mask isn't viable so this inaccuracy needs to be tolerated or mitigated in other ways.

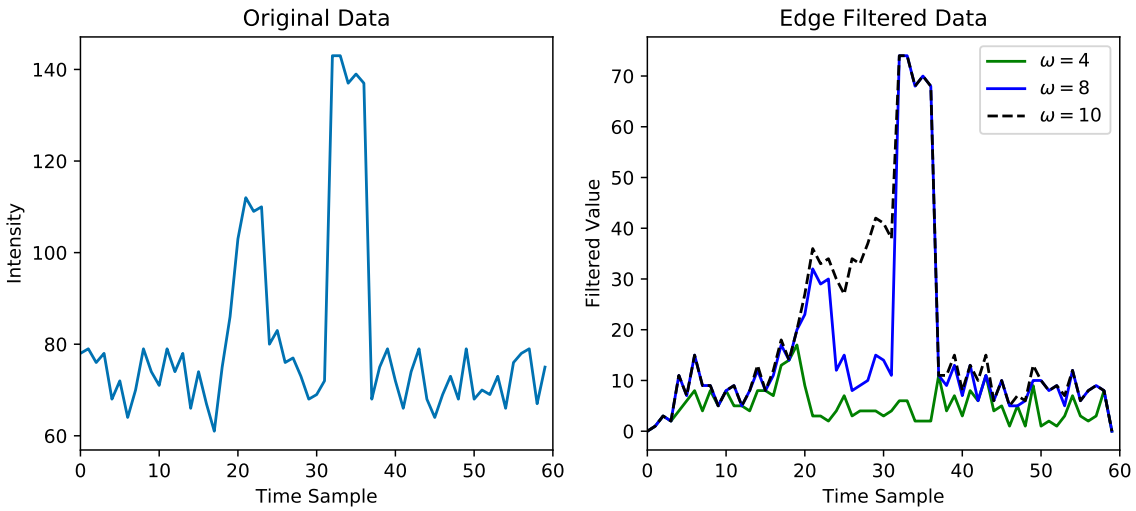

Fig. 4.5. Left: Uncleaned data from a single frequency channel of a WSRT observation with two clear RFI peaks. Right: The results of edge-filtering the uncleaned data with different window sizes.

The results in figure 4.4 and figure 4.5 make the importance of picking the correct window size clear. A too-large window can obscure FRBs or incorrectly flag clean data while a too-small window can result in RFI being missed.

4.2.2 Comparison of Mean, Point and Median Edge-Thresholding

Along with picking a window size, the choice of the statistic used to measure the window of data is an important factor determining the performance of edge-thresholding. In figure 4.6 the results of mean, median and point edge-filtering with a window size of five are shown as is the original uncleaned time-frequency intensity array.

The mean statistic is a tempting choice because computing the mean of each window is computationally efficient and the computation can easily be reused for larger win-

dows. However, the sensitivity of the mean to outliers makes mean edge-thresholding inaccurate. This effect can be seen in figure 4.6, where only a single point of RFI is causing an entire window of size five to be flagged. RFI smaller than five time samples should have already been detected in earlier iterations but the information cannot be communicated between iterations without an input mask. So, again, having an input mask would be ideal but is unfeasible.

In theory, another way to minimise inaccuracies would be to use a more robust statistic such as the median. As is shown in figure 4.6 using the median does improve accuracy - but the additional memory and computational cost of computing the median for every window makes this impractical.

These problems with using the mean and the median led to the development of pointwise edge-thresholding. The results are also shown in figure 4.6, where it can clearly be seen that pointwise edge-thresholding is more accurate than both mean and median edge-thresholding without the need for a costly sort operation or input mask.

4.2.3 Edge-Thresholding WSRT Observations

Examples of edge-thresholding at work were shown in section 4.2.1. In this section the experiments are scaled-up to determine how edge-thresholding affects the performance of a transient-detection pipeline. Point and mean edge-thresholding are considered along with masked-mean edge-thresholding - which uses an input mask from previous iterations to exclude previously masked data in all computations.

Edge-thresholding needs to not obscure FRBs. FRBs were inserted into data from two observations known to include no signal from a transient. The first observation in direction RA04DEC60 had 332 false triggers in 10 minutes. The second observa-

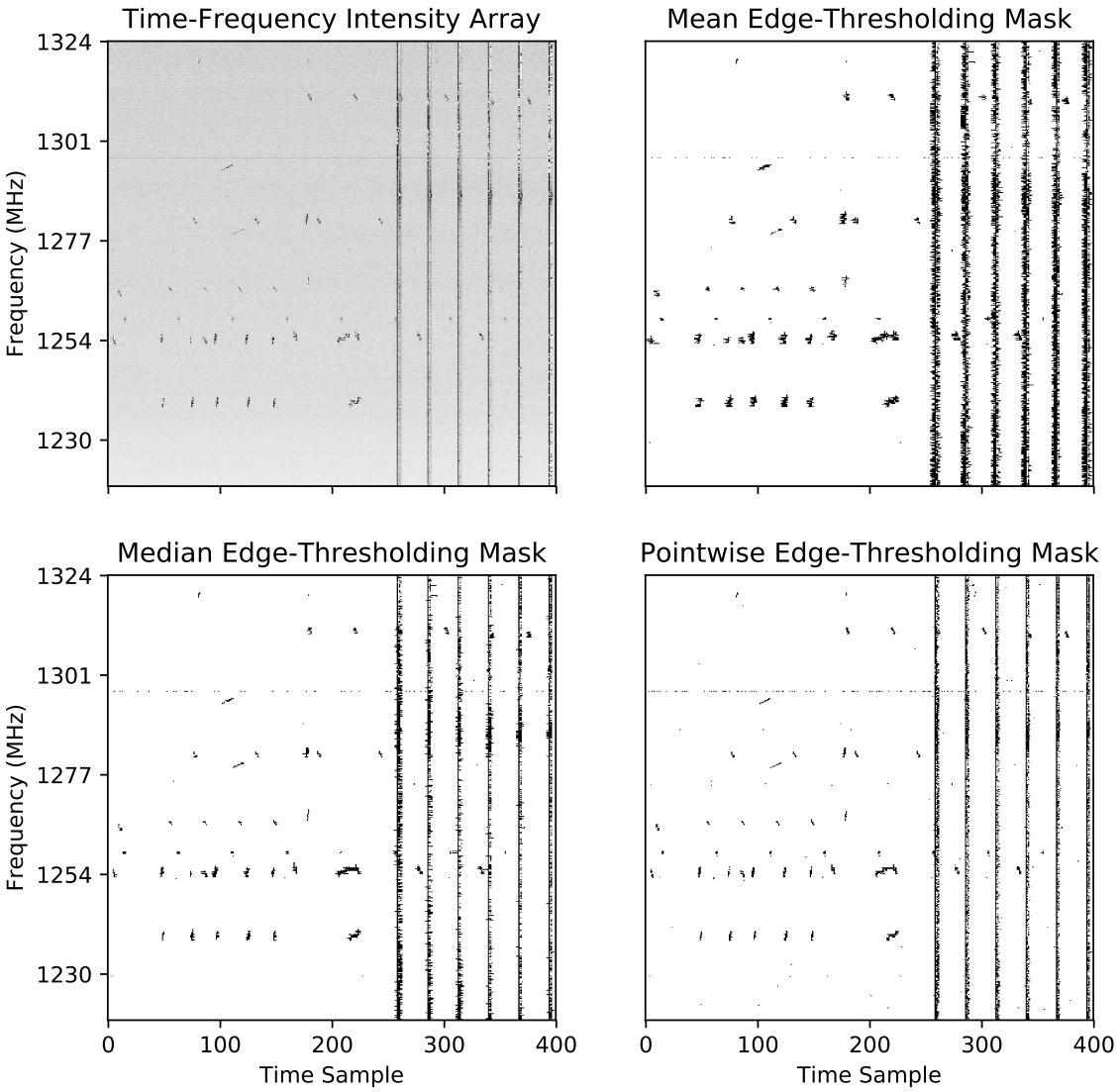

Fig. 4.6. A time-frequency intensity array of a WSRT observation with clear RFI and the resulting mask from three different version of edge-thresholding.

tion in direction RA23DEC54, which was a lot more contaminated by RFI, falsely triggered 13,164 times in 10 minutes. The FRB detection results in both files before and after edge-thresholding are shown in figure 4.7.

The FRB-detection rate after edge-thresholding in the less-contaminated file is shown in the top three plots of figure 4.7. Edge-thresholding - be it mean, masked-mean or pointwise edge-thresholding - has clearly made no difference to the detection of FRBs. This is good news because it suggests that in an ideal scenario, where the data contains only Gaussian noise and signal, edge-thresholding will not obscure FRBs.

In the file with more RFI the results are slightly different, as is shown in the bottom three plots of figure 4.7. Here the good news is that all three methods of edge-thresholding result in new FRBs being detected - the cleaner files making it easier for the correct DM to be detected by Heimdall. But theres bad news too: all three methods also obscure some FRBs. For point and masked-mean edge-thresholding the number of newly detected FRBs is more than the number of obscured FRBs, but for mean edge-thresholding the reverse is true.

Mean edge-thresholding was shown in section 4.2.1 to be inaccurate and the experiments in this section indicate that its inaccuracy hinders the detection of FRBs in a high-RFI environment. The assertion that this can be blamed on the mean being a non-robust statistic is supported by the fact that masked-mean edge-thresholding, which mitigates the sensitivity of the mean to outliers, does not obscure as many FRBs.

An example of an FRB obscured by mean edge-thresholding is shown in figure 4.8. The two inset plots of figure 4.8 show the de-dispersed pulse profile and the red dashed lines show the arrival and end time of the FRB. The occurrence of the FRB aligns perfectly with the RFI. So when the RFI is inaccurately cleaned it also obscures the

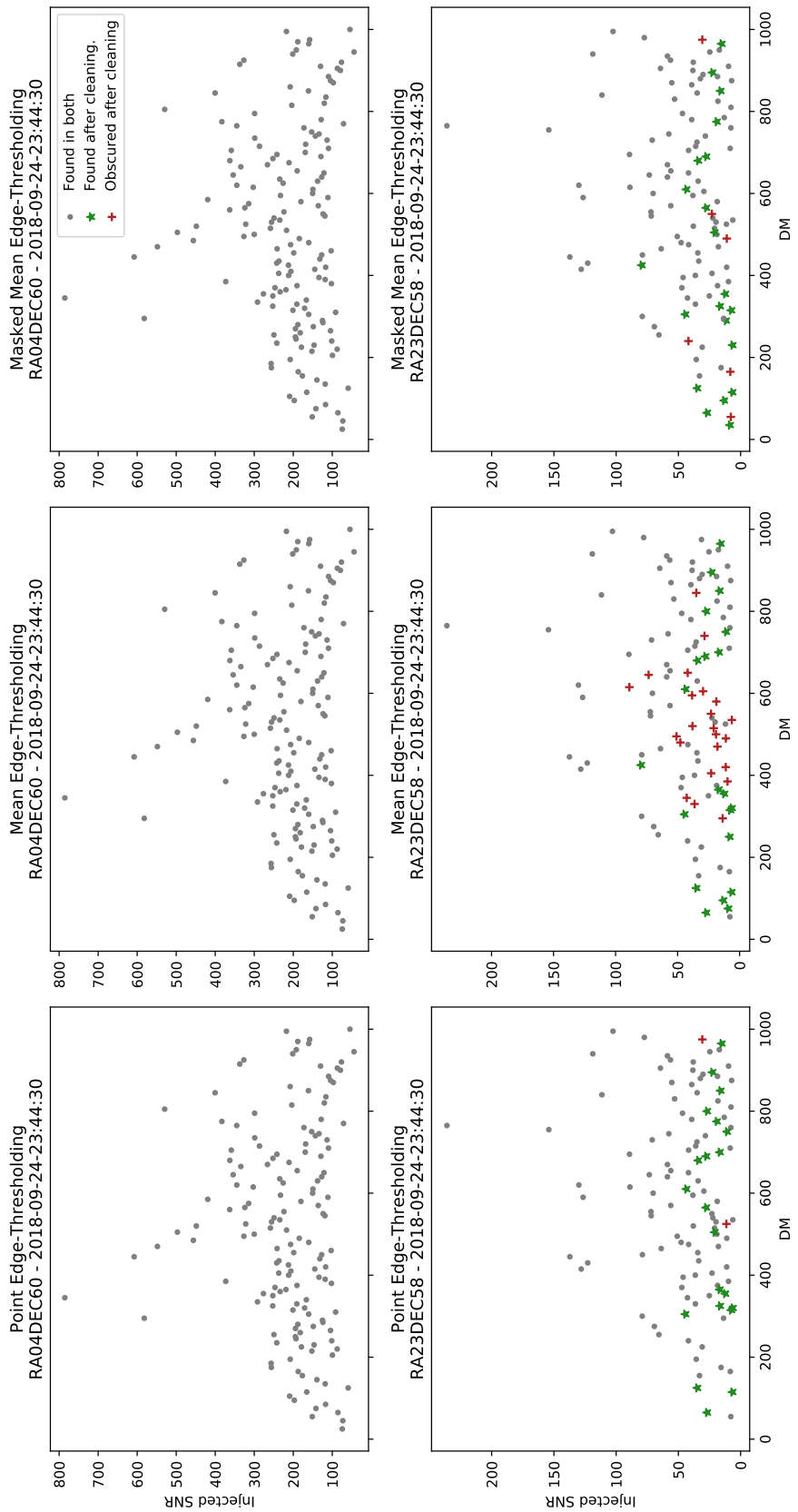

Fig. 4.7. Plots showing the detected FRBs before and after different edge-thresholding implementations on two different WSRT observations.

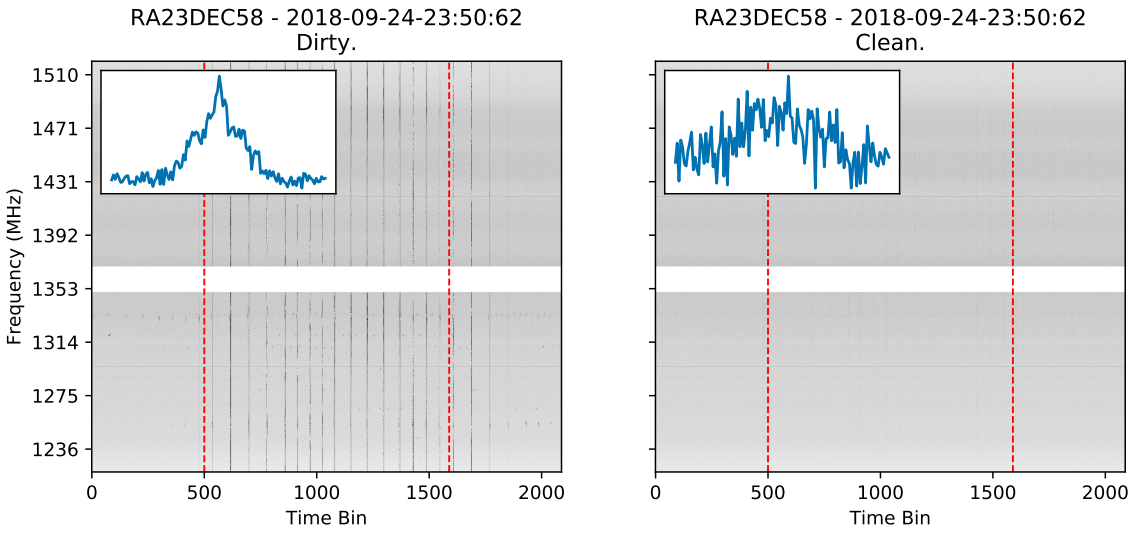

Fig. 4.8. The time-frequency intensity array of an FRB in an unclean event before and after mean edge-thresholding.

FRB. As can be seen in the de-dispersed pulse profiles the cleaned file no longer contains the pulse of the FRB.

The FRB in figure 4.8 was obscured by mean edge-thresholding but not by pointwise edge-thresholding - thanks to its greater accuracy. Figure 4.7 shows many FRBs being obscured by mean edge-thresholding but not by pointwise edge-thresholding. Whats more, pointwise edge-thresholding is clearly better at detecting FRBs. Overall, using pointwise edge-thresholding resulted in 163 FRBs being detected - 20 more than the 143 detected using mean edge-thresholding. This is a 12 percent improvement.

The significant increase in the number of FRBs detected after pointwise edge-thresholding is somewhat diminished by the fact that two have been obscured by it. Obviously it would be helpful to understand what has caused pointwise edge-thresholding to miss these two FRBs. In figure 4.9 the pulse profile of both FRBs before and after cleaning is shown. In both cases the FRB is easily visible in the clean file and in the left-hand plot the upward slope, which is probably the result of RFI, is cleaned. This suggests the cleaning of RFI is working well, so it is surprising that Heimdall has missed the FRB afterwards. In this regard, Heimdall is black-box, making it difficult to identify the cause of this error.

So far we have discussed the impact of edge-thresholding on the detection of FRBs. The next step is to determine if edge-thresholding also reduces the number of false triggers. Table 4.1 provides the answer. The number of false triggers is reduced by both mean and pointwise edge-thresholding. Fortunately, given pointwise edge-thresholdings good FRB-detection rates, it also results in the largest reduction in the number of false triggers. The largest percent decrease in the number of false triggers comes using a window size of one. The additional reduction from larger windows is comparatively marginal. This is good news given the additional computational cost

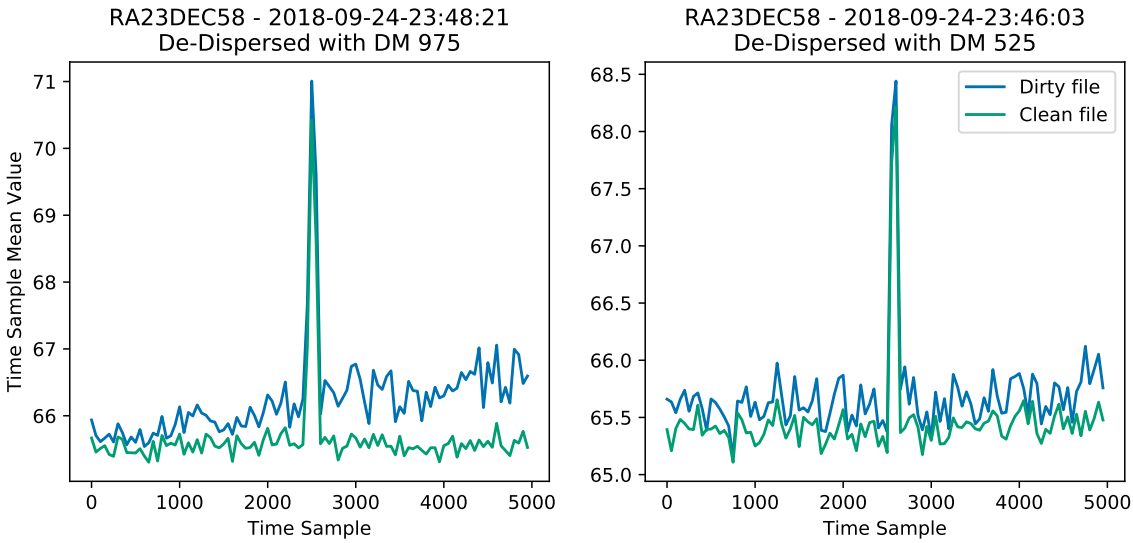

Fig. 4.9. The de-dispersed pulse profile of two FRBs in cleaned and uncleaned data.

Pointwise edge-thresholding	Dirty	$\omega \leq 1$	$\omega \leq 5$	$\omega \leq 10$	% Decrease
RA04DEC60 from 23:44 to 23:55	13164	933	564	435	96.69%
RA23DEC58 from 23:44 to 23:555	332	135	133	124	62.93%
Mean Edge-Thresholding	Dirty	$\omega \leq 1$	$\omega \leq 5$	$\omega \leq 10$	% Decrease
RA04DEC60 from 23:44 to 23:55	13164	933	710	723	94.5 %
RA23DEC58 from 23:44 to 23:55	332	135	169	172	48.19%

Table 4.1
The number of false triggers with a signal-to-noise ratio above 8 after pointwise and mean edge-thresholding on two different observations, one in direction RA04DEC60 and one in direction RA23DEC58. $\omega \leq 5$ and $\omega \leq 10$ indicates that edge-thresholding has been executed iteratively with all window sizes less than 5 and 10 respectively.

of large windows and the other negatives associated with them, as discussed in section 4.2.1. The results in table 4.1 reveal another disadvantage of large windows namely that, when using mean edge-thresholding, the number of false triggers increases as the window size increases. In fact, throughout the experimentation for this thesis, it was found that too-aggressive RFI mitigation actually increases the number of false triggers.

4.3 Sum-Thresholding Compared to Edge-Thresholding

In this thesis we propose edge-thresholding as an alternative to sum-thresholding and to this end the precision of sum-thresholding and edge-thresholding on the same input data is in this section compared. The results of one representative experiment is plotted in figure 4.10.

For sum-thresholding there is a threshold associated with every window size and all windows with an masked mean intensity above the threshold are flagged. This means

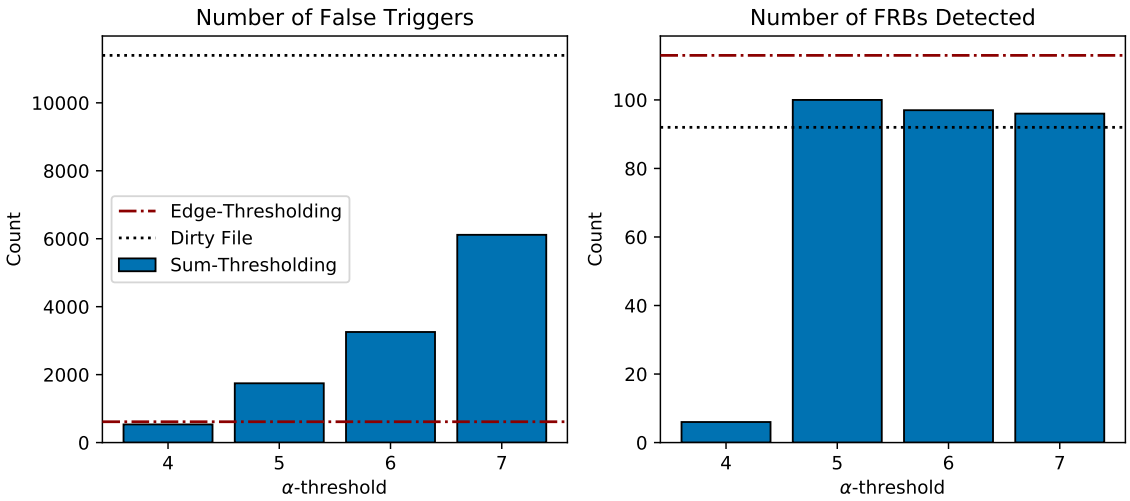

Fig. 4.10. A comparison of the FRB detection and false trigger rate of sum-thresholding with a dirty file and a file file cleaned with edge-thresholding.

that without obscuring FRBs sum-thresholding can only flag RFI that is brighter than all FRBs. This can be effective because a lot of RFI is extremely bright and many FRBs have a low signal-to-noise per pixel before de-dispersion and integration over the frequency channels. This observation was confirmed by experimentation, an example of which can be seen in figure 4.10, which proved that sum-thresholding can reduce the number of false triggers without decreasing the number of FRBs detected.

However, there is also a lot of RFI that causes false triggers but isn't necessarily bright in comparison to the brightest FRBs and sum-thresholding, unlike edge-thresholding, is unable to mitigate this type of RFI. As a result sum-thresholding is unable to reduce the number of false triggers as aggressively as edge-thresholding without obscuring FRBs. Again this was confirmed with experiments. As can be seen in figure 4.10, for sum-thresholding to achieve a comparable false trigger reduction to edge-thresholding many of the FRBs are also obscured. Additionally the more aggressive edge-thresholding is not only able to reduce more false triggers but also allows for more FRBs to be detected.

The results of the experiments such as those shown in figure 4.10 indicate that edge-thresholding is more effective at mitigating RFI than sum-thresholding. Additionally, unlike sum-thresholding, the pointwise edge-thresholding implementation used for these experiments does not require a mask. This means that it is not only more accurate but also more efficient with computational resources. We showed in section 4.2.3 that the accuracy of edge-thresholding is improved with an input mask so this comparison could be even more favourable to edge-thresholding if both methods were allowed to use an input mask.

As discussed, the memory and computational requirements at WSRT means that maintaining a mask is not viable. So if sum-thresholding was to be implemented as part of the RFI mitigation pipeline at WSRT it would have to be without a mask.

As we know the mean is not a robust statistic and therefore without an input mask sum-thresholding would be much less accurate because a single outlier data point could cause larger windows to be flagged. As was shown in section 4.2.3 using an unmasked mean in edge-thresholding resulting in many FRBs being obscured in high RFI environments and this could also be a problem with sum-thresholding if no mask was being used.

4.4 Scale Invariant Rank Operator

The Scale Invariant Rank (SIR) operator looks at the structure of the mask to identify other data points that are probably also RFI. It has to be used in conjunction with other RFI mitigation algorithms that determine the input mask. It's often used as the final stage of an RFI pipeline before data excision. In [27] the SIR operator was shown to work well in conjunction with sum-thresholding for imaging purposes. In this section the efficacy of the SIR operator in conjunction with edge-thresholding is tested for transient detection.

To test the SIR operator we use edge-thresholding to generate an RFI mask. The mask is passed through the SIR operator and both the original and modified mask are used to excise the RFI from the data. Then both data outputs are passed through Heimdall and the resulting triggers are compared. One example, with an edge-threshold of 3.5 and 10 iterations, is shown in figure 4.11. The left hand plot of figure 4.11 shows the relative increase in the signal-to-noise ratio of detected FRBs with and without the mask being passed through the SIR operator. The right hand plot of figure 4.11 shows the extra trigger from the SIR operator.

These experiments indicate that the SIR operator modestly decreases the number of false triggers. In figure 4.11 the SIR operator resulted in a 25 percent reduction in the number of triggers, which seems quite good until one notes that most of the

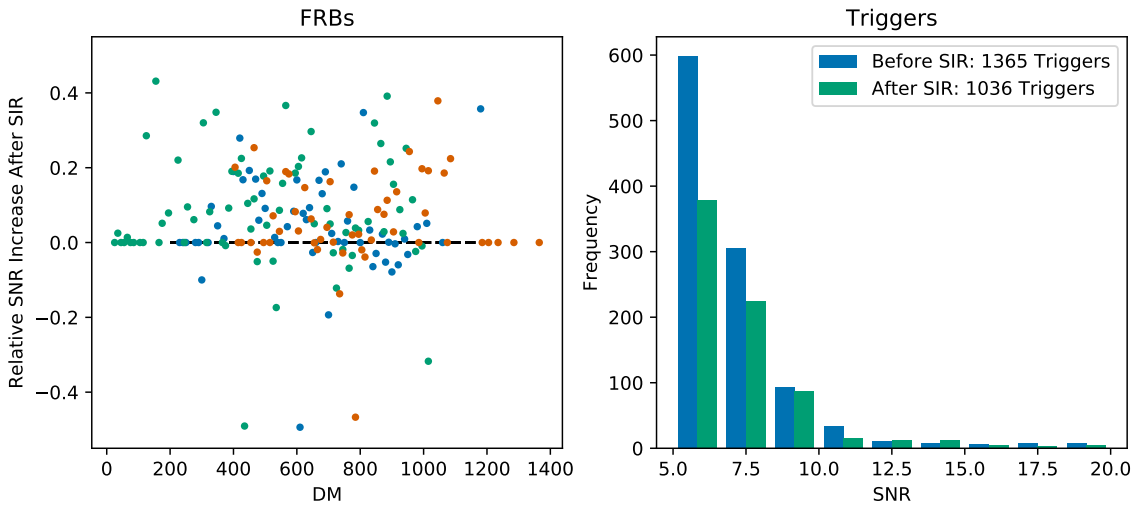

Fig. 4.11. Left: A figure showing the relative change in the signal-to-noise ratio of detected FRBs when the SIR operator is used. Right: The trigger reduction resulting from the SIR operator.

removed triggers have a low signal-to-noise ratio. For triggers with a signal-to-noise ratio above 10, the reduction in the number of triggers is much lower. In the example shown in figure 4.11 the number of triggers with a signal-to-noise ratio above 10 is only reduced from 353 to 344. So the SIR operator only makes a significant difference to the number of false triggers if the FRB search is parameterised to find FRBs with a low signal-to-noise ratio.

While the SIR operator results in marginal decreases in the number of false triggers it has a considerable impact on the signal-to-noise ratio of detected FRBs. In the left-hand plot of figure 4.11 the relative signal-to-noise increase is shown for FRBs detected in a file cleaned with edge-thresholding alone and in another file cleaned with edge-thresholding plus the SIR operator. FRBs along the zero vertical line have exactly the same signal-to-noise ratio before and after the SIR operator. There are many of these FRBs but the majority are above the vertical line, which indicates that the SIR operator has increased the signal-to-noise ratio at which these FRBs were

detected. Remarkably, over all the FRBs that were detected in both files the SIR operator resulted in an average absolute signal-to-noise increase of 92.54 per FRB. Additionally, the number of FRBs detected increased by four from 159 to 163. While it's good to know that the SIR operator will on average increase the signal-to-noise ratio of detected FRBs, its important to note that this is not always the case. As you can see in figure 4.11 some of the FRBs have a relative signal-to-noise ratio decrease of more than 40 percent, although this did not prevent them from being detected.

5. FF-FLAGGER

The FF-Flagger, proposed for the first time in this thesis, is a real-time GPU-accelerated RFI-mitigation pipeline designed to work with Amber at WSRT's FRB-detection project. The theory from section 2 and the results from section 4 are used to endorse the formulation of the FF-Flagger described here.

This thesis also includes an implementation of the FF-Flagger as a component of Amber. The implementation details and performance results of the FF-Flagger as a component of Amber are also discussed in this section.

5.1 Specification

Like the AOFlagger, the FF-Flagger is an iterative RFI-mitigation pipeline that contains thresholding and time-sample mitigation steps. As discussed in section 1.2.4 the most common forms of RFI at WSRT are broadband or narrowband ephemeral RFI and these are therefore the types of RFI that the FF-Flagger is optimised to mitigate. Mitigating the narrowband RFI is done by pointwise edge-thresholding each frequency channel along the time dimension. Although not its original intention, pointwise edge-thresholding has proved also to be quite effective at mitigating broadband RFI. Mitigating the broadband zero-DM RFI is then done using a sigma cut on the mean of time samples.

The FF-Flagger has three modes. Mode one computes the median and MAD of each frequency channel and then does multiple iterations of pointwise edge-thresholding. Mode two computes the median of each frequency channel and then does multiple sigma cuts on the means of the time samples. Mode three combines the two methods,

starting with the computation of the medians and MADs of the frequency channels and then doing multiple iterations of pointwise edge-thresholding followed by multiple iterations of sigma cuts on the time-sample means. The choice of which mode is optimal depends on the RFI environment and the computational restrictions of the system.

For each iteration of both stages of the pipeline the thresholds are kept constant. At the end of each iteration of the sigma cut, RFI is excised so the next iteration, which recomputes the mean and standard deviation, is able to detect new outliers with the same threshold. The window size used for pointwise edge-thresholding always starts at one and is incremented by one every iteration.

All detected RFI is replaced by the median intensity of the frequency channel that contains the RFI-contaminated data point. As discussed in section 2.6 this is an effective RFI-excision scheme. Pointwise edge-thresholding does not use an output mask so RFI is excised by the pointwise edge-thresholding kernel itself. Computing the median and MAD is a costly operation but also necessary for RFI excision and for calculating the pointwise edge-thresholding threshold. To offset this cost, the median and MAD are only recomputed every $n-th$ event to save compute time. This works because the median and MAD are robust statistics and the observational environment of the telescope is quite stable over a few seconds. The frequency with which the median and MAD are recomputed is a parameter of the FF-Flagger.

To use the FF-Flagger the following parameters need to be chosen:

- A mode, determining which stages of the pipeline are executed.
- A threshold for pointwise edge-thresholding - in other words, a threshold number of MADs that the edge-filter value must have for the point to be flagged as RFI.

- The maximum window size, starting at size one, that is used for pointwise edge-thresholding.
- A threshold for the sigma cut of time samples - in other words, a threshold number of standard deviations from the mean that is required for a time sample to be flagged.
- The number of time-sample sigma-cut iterations done.
- A frequency determining how many events are mitigated before the median and MAD are recomputed.

5.1.1 Implementation Within Amber

The FF-Flagger has been implemented in C++ and OpenCL to make it easily compatible with Amber. It is implemented as a C++ class with a constructor that takes into account Amber-specified OpenCL environment variables. Therefore using the FF-Flagger with Amber just requires including the class and calling the correct constructor. The current version of the FF-Flagger works on only one telescope-dish beam at a time.

The FF-Flagger is also implemented as a free-standing RFI-mitigation pipeline that reads and writes filterbank files. Although intended to be used with Amber, the FF-Flagger could easily be included in other transient-detection pipelines. Within Amber the FF-Flagger reads an RFI-configuration file that specifies the parameters to be used. Additionally, the FF-Flagger comes with a suite of unit tests that validate the correctness of the GPU kernels.

The implementation of pointwise edge-thresholding described in section 3.2.2 is used within Amber with one variation: no output mask is used. In experimentation section of edge-thresholding, section 4.2, the location of RFI values were stored using an

output mask, which is not viable at WSRT with Amber. Instead, the version implemented with Amber writes directly back to the input data rather than to an output mask. Because we want to avoid atomic operations, the choice of no output mask results in data races because multiple threads read and write to the input data. This should not be a problem because there is no issue with excising an RFI-contaminated data point multiple times. In fact, writing directly back to the input data is similar to having an input mask because there is a chance (though no guarantee) that when using bigger window sizes smaller RFI features will already have been excised by an earlier iteration of pointwise edge-thresholding with a smaller window size. This is similar to having an input mask, which was shown in section 4.2 to increase accuracy. In short, using no output mask and writing directly back to the input array could actually improve performance.

pointwise edge-thresholding uses the MAD of each frequency channel to threshold windows and the median of each frequency channel is to excise RFI. The median and MAD of each frequency channel is an unsigned 8-bit integer so the additional memory required for pointwise edge-thresholding is two unsigned 8-bit integers for every frequency channel.

The time-sample sigma-cut implementation described in section 3.1 is used for the Amber implementation of the FF-Flagger. While this implementation is efficient it requires quite a lot of additional memory. A device buffer of floats is required to store the mean of each time sample and a buffer of equal size is required for doing the reduce operation because the tree-based GPU reduce, as described in section 3.1.1, requires a temporary array if it is to leave the original data untouched - which is required in our case because the time-sample means also need to be used to compute the standard deviation and in later iterations of the sigma cut. Additionally an array on integers is required to store the index of the time samples that are identified as RFI. At most this array needs be the same size as the number of time samples but this clearly is

seldom the case because it is rare for every time sample in an event to be flagged. With some experimentation it was found that using a sigma-cut threshold of 2.5 the number of flagged time samples in a 43,776 time-sample event never went above 10,000. Still limiting the size of the array would require some specified behaviour in the rare case that it overflows. For the sake of simplicity, the current implementation of the FF-Flagger uses an array the same size as the number of time samples to be safe. An overview of the additional memory requirements of the FF-Flagger can be seen in figure 5.1. This figure shows the additional memory requirements for a single beam, meaning that 12 times the memory shown in figure 5.1 would be required to process all the beams. This means that the FF-Flagger needs less than 6.3 megabytes of additional memory per GPU.

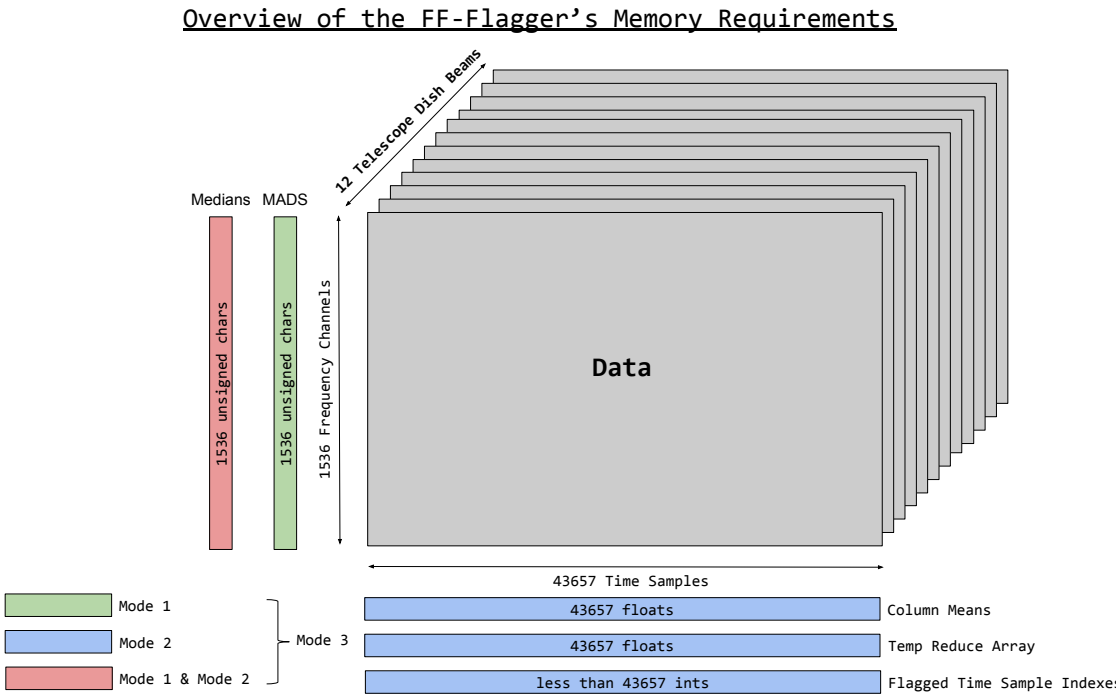

Fig. 5.1. An overview of the FF-Flagger's memory requirements for a single on-sky beam. Mode 1 is pointwise edge-thresholding, mode 2 is a time-sample sigma cut and mode 3 is the FF-Flagger pipeline (both).

5.1.2 Comparison with AOFlagger

The AOFlagger is the current default RFI-mitigation pipeline at WSRT and LOFAR and it has been shown to be effective as an RFI-mitigation pipeline for imaging purposes at both of these telescopes. The FF-Flagger tries to use what is good about the AOFlagger but re-purpose it to the transient-detection application. For this reason the structure of the FF-Flagger is similar to that of the AOFlagger in the sense that both pipelines are iterative and contain a thresholding and time-sample sigma cut.

The iteration patterns of the two pipelines are different. The FF-Flagger does all the thresholding iterations before doing the time-sample sigma-cut iterations. This is in contrast to the AOFlagger which does one threshold and one sigma-cut operation for each iteration. The reason for this choice with the FF-Flagger is that it is more efficient to do all the thresholding iterations in a single GPU-kernel call and so avoid parallel overhead. In particular, the overhead from loading global memory in shared local memory is reduced because the shared local memory can be used for all the iterations rather than requiring the shared local memory to be reloaded for a later iteration, as is the case with the AOFlagger.

For the time-sample sigma cut, the AOFlagger uses the RMS of each time sample while the FF-Flagger uses the mean of each time sample. This decision was made because, as shown in section 4.1, the mean is better at reducing the number of false triggers and in all cases was equal to, or better than, the RMS in terms of obscuring bright low-DM FRBs. The mean is also a slightly more efficient statistic to compute because less square operations are required.

For thresholding, the FF-Flagger uses edge-thresholding rather than sum-thresholding as is the case with the AOFlagger. As shown in section 4.3, edge-thresholding is able

to use a tighter threshold because it makes more assumptions about the structure of RFI and signal. This means that more false triggers can be mitigated without obscuring FRBs.

The AOFlagger also includes a down-sampling stage, a Gaussian high-pass filter and the SIR operator. As discussed in section 2.4.2 the SIR operator is not feasible for the FF-Flagger because the linear-time complexity SIR implementation requires $O(3n)$ additional memory. The AOFlagger includes a down-sampling stage to reduce the performance requirements of later iterations but this would also require additional memory for storing the down-sampled data and therefore is also not feasible for the FF-Flagger. Finally, the Gaussian high-pass filter makes the assumption that the signal of interest is smooth [27]. While this holds for imaging applications, this assumption is less true for transient detection and therefore is unlikely to be as effective as part of the FF-Flagger.

Another crucial difference between the two methods is that the FF-Flagger doesn't maintain a mask. It would be helpful to see the RFI mask in order to do analysis of the RFI environment but the memory constraints at WSRT make this impossible. Future versions of the FF-Flagger could use some additional memory to store statistics regarding the RFI excised data but the current implementation does not offer this functionality.

The AOFlagger also uses variable threshold values that alter for every iteration. This would probably also be a beneficial upgrade to any future version of the FF-Flagger.

5.2 Results

In section 4 the key RFI-mitigation algorithms, pointwise edge-thresholding and a time-sample mean-sigma cut, were shown to perform well on a small amount of data

from just a couple of observations. To make the results in this thesis more rigorous, the final version of the FF-Flagger pipeline as described in section 5.1.1 is executed on multiple longer observations. The results are shown for mode 1, which is pointwise edge-thresholding, mode 2, which is the time-sample mean-sigma cut, and mode 3 which is pointwise edge-thresholding followed by a time-sample mean-sigma cut.

On each observation, two iterations of pointwise edge-thresholding and three sigma-cut iterations are executed. The exact time taken by FF-Flagger is dependent on the dirtiness of the observation. For this reason the performance results of the FF-Flagger on the CasA observation, which is the most contaminated observation, are plotted in figure 5.2. Figure 5.2 also shows the average amount of time required by each kernel of the FF-Flagger to process a single second of data. The real-time requirement is 8000 microseconds as these results are for a single on-sky beam and therefore all three modes run in real time.

Computing the frequency-channel medians and doing pointwise edge-thresholding are the most costly parts of the pipeline. It was shown in section 2.6 that using the medians to excise RFI reduces the number of false triggers and it was shown in section 4.2.3 and table 5.1 that pointwise edge-thresholding is effective, which suggests both kernels are worth the cost. To determine whether speed-up of these kernels is possible - given the hardware and the compute that they are doing - would require the analysis of a performance model such as the roof-line model.

The astronomical results of the FF-Flagger can be seen in table 5.1. The data set includes a 30-minute long observation of the Crab Pulsar, which is a bright low-DM transient, perfect for testing whether the FF-Flagger is likely to obscure signal. The rest of the data has 1000 simulated FRBs injected but some of the FRBs - those with a signal-to-noise ratio below 8 - are ignored. All false triggers with a signal-to-noise ratio below 8 are also excluded.

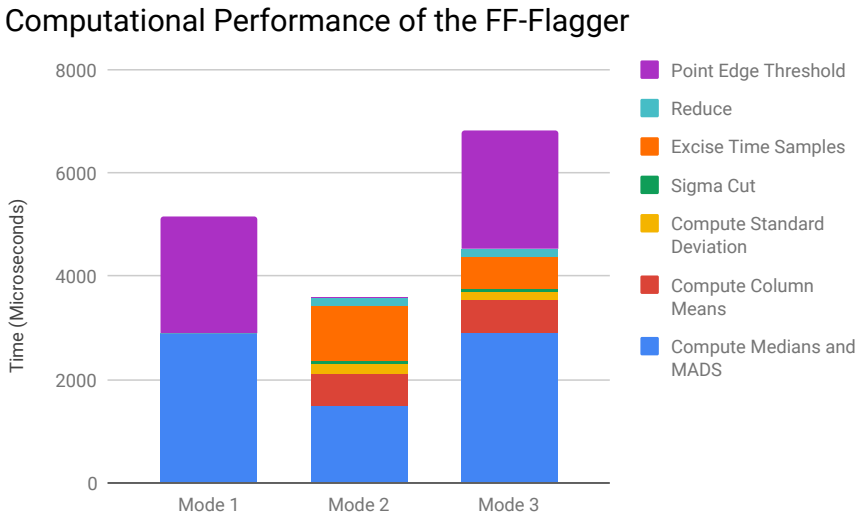

Fig. 5.2. The performance of each mode of the FF-Flagger broken down into the time required by each kernel. The numbers are the average time required per time-frequency intensity array with dimension $1536 \times 43,657$ over 30 minutes of RFI contaminated sky data.

Looking at the Crab pulsar data in table 5.1 it is clear that the FF-Flagger can be very effective. The number of pulses detected is increased by more than 60% and the reduction of false triggers is above 98% for all three modes. For mode 3 - the entire FF-Flagger pipeline - the number of false triggers is, impressively, zero. This level of performance is achievable because the RFI-environment of the observation happens to include only the type of RFI that the FF-Flagger is designed to mitigate and because the signal is all regular. This means that threshold parameters can be set aggressively as only a single type of signal needs to be avoided.

The CasA file is another observation where the FF-Flagger is very effective. There is more than a 20% increase in the number of FRBs detected and up to a 91% decrease in the number of false triggers. This observation also shows the value of combining pointwise edge-thresholding and a time-sample mean-sigma cut because each individual component is far less effective than the two methods combined.

Both the Crab pulsar and CasA are highly contaminated with RFI but, in contrast, 12RTRA04DEC60 and M31 have relatively few false triggers. Nevertheless, the false-trigger reduction is still good, around two thirds, but more importantly the number of FRBs is not reduced. This matters because it shows that even in a low-RFI environment such as these the FF-Flagger is unlikely to obscure FRBs.

The most disappointing results come from P1858-0814. In this file the number of FRBs detected is reduced by mode 1 and mode 3, which indicates that pointwise edge-thresholding is obscuring FRBs, although in return the false-trigger rate is reduced by up to 98%.

The unusual feature of P1858-0814, which could explain why pointwise edge-thresholding reduces the number of FRBs detected, is that the observation had an extremely low

intensity. The mean intensity of events was a little more than 8 when the mean intensity of other observations was in the 60-to-130 range. For mode 2 - using only a time-sample mean-sigma cut - the number of FRBs detected is increased but there is almost no reduction in the number of false triggers.

The reduction in FRBs detected in P1858-0814 proves that there is no guarantee that the FF-Flagger will not obscure FRBs. In fact, in most cases where there is a large amount of RFI, as was shown in figure 4.7 in section 4.2.3, the FF-Flagger does obscure some FRBs. Fortunately though, as the results in table 5.1 show, cleaning the data generally results in more new FRBs being revealed than the number being obscured.

Overall, these experiments suggest that mode 3 - the entire FF-Flagger pipeline - is optimal. In some files, such as CasA and M31, the two stages of the pipeline work additively and therefore both are necessary. But in other files, such as FRB170922 and P1858-0814, only one of the stages in the pipeline is actually effective, and therefore the other stage could have been dropped to save computational time. Similarly, in some cases, such as 12RTRA04DEC60 and the Crab Pulsar, both stages of the pipeline are roughly equally effective and the gain from doing both is marginal. However, since this pipeline needs to execute online it is difficult to know beforehand which stage will be required or whether only one stage will be required. It makes sense to rather run the entire pipeline every time and so be certain that all possible benefits are being obtained - especially considering that running both stages never reduces the astronomical performance.

This challenge for the FF-Flagger of not knowing beforehand which stages of the pipeline will be effective stems from not knowing the RFI environment beforehand - and that, in turn, makes it difficult to tune the parameters. To get the results shown in table 5.1 the parameters had to be aggressively tuned. The incorrect parameters

can easily obscure FRBs or increase the number of false triggers, so a production-ready version of the FF-Flagger would need a way to parameterise itself online to obtain optimal performance.

The results in this section have shown that the FF-Flagger can run in real time and is effective on a multitude of different WSRT observations. The results also show that, in addition to not obscuring simulated signal, the FF-Flagger can also avoid obscuring real signal. Still to be resolved is the challenge of parameterising the pipeline online.

		Transients Detected		False Triggers	
Source Name	RFI-Mitigation	Original	% Increase	Original	% Reduction
Crab Pulsar	Original Data	941	-	17254	-
	Mode 1	1573	67.16 %	239	98.61 %
	Mode 2	1593	69.29 %	138	99.20 %
	Mode 3	1590	68.97 %	0	100 %
CasA	Original Data	555	-	25465	-
	Mode 1	677	21.98%	4578	82.02 %
	Mode 2	701	26.30 %	15196	40.33 %
	Mode 3	717	29.19%	2102	91.75 %
12RTRA04DEC60	Original Data	114	-	114	-
	Mode 1	114	0 %	59	58.77 %
	Mode 2	114	0 %	47	58.77 %
	Mode 3	114	0 %	38	66.67 %
FRB170922	Original Data	865	-	641	-
	Mode 1	878	1.50 %	655	-2.18 %
	Mode 2	890	2.89 %	391	39.00 %
	Mode 3	890	2.89 %	369	42.43 %
P1858-0814	Original Data	188	-	8339	-
	Mode 1	349	-13.29 %	349	95.81 %
	Mode 2	217	15.43 %	8303	0.43 %
	Mode 3	114	-14.89 %	114	98.63%
M31	Original Data	377	-	43	-
	Mode 1	377	0 %	16	62.79 %
	Mode 2	377	0 %	27	37.21 %
	Mode 3	377	0 %	11	74.42 %

Table 5.1

The results in this table show the impact of the FF-Flagger's three different modes on the FRB detection and false trigger rate for various different observations. Each observation is approximately 30 minutes long. All FRBs and false triggers with a signal-to-noise ratio below 8 are ignored for these results. All of the signal is simulated except in the Crab pulsar file.

5.3 Discussion and Further Optimisations.

While the FF-Flagger is an effective RFI-mitigation pipeline there is still room for improvement. In this section some of these opportunities are discussed along with the thinking behind the current version.

The upgrade the FF-Flagger most needs is the capacity to process multiple telescope-dish beams. This should be simple to implement and would enable the FF-Flagger to be used on the full 12-dish data stream at WSRT.

As shown in section 4.4, the SIR operator is effective, particularly at increasing the detected signal-to-noise ratio of FRBs. The SIR operator requires a mask but simple constant dilation could be an effective alternative that doesn't require a mask. For this thesis a dilation operation added to pointwise edge-thresholding was experimented with but this severely damaged the performance even if the dilation was only one-dimensional along the efficient memory-access axis. For this reason constant dilation was excluded from the final FF-Flagger but, given the success of the SIR operator, some other form of morphological detection could be included in a later version of the FF-Flagger.

pointwise edge-thresholding would also be more effective if it could be executed across time samples as well as across frequency channels. The current implementation only operates along frequency channels because this is memory-access efficient. Efficiently executing pointwise edge-thresholding across time samples would require an efficient in-place GPU transpose.

As mentioned in section 1.1.3 some RFI is narrowband but long in time. As discussed in section 1.2.4 this type of RFI is uncommon at WSRT but it does still occur. Currently the FF-Flagger does nothing to mitigate this type of RFI. Point-

wise edge-thresholding across time samples would help to mitigate this type of RFI. Alternatively, a sigma cut could be done on the mean intensities of frequency channels. The issue with this is that the mean intensities of frequency channels at WSRT has a pseudo-sinusoidal pattern across frequency channels that reduces the effectiveness of a sigma cut. One solution to this is to subtract down-sampled frequency-channel means before the sigma cut to remove the pseudo-sinusoidal structure. The problem with all of these solutions is that they are computationally costly and the type of RFI they mitigate is not particularly common. Therefore they were excluded from this version of the FF-Flagger.

It was also show in section 4.1 that the trimmed mean is in many cases a better measure of centrality for the time-sample sigma cut than the mean. We can use pigeonhole sorting to efficiently compute the percentile required for the trimmed mean but this is only efficient if the time-sample data is along the memory-access efficient axis. This would also require an efficient in-place GPU transpose.

There are also opportunities within the FF-Flagger to increase the performance. For example, many of the kernels have multiple interacting parameters than effect the computational performance and therefore automated performance tuning would be helpful. Also kernels could be found within the FF-Flagger that could be further optimised by using a performance model like the roof-line model.

6. CONCLUSIONS

The goal of this thesis was to develop a software solution to the problem of RFI for the FRB-detection effort at WSRT. The proposed solution was a GPU-accelerated RFI-mitigation pipeline called FF-Flagger based upon a novel thresholding algorithm called point edge-thresholding.

The efficacy of the FF-Flagger was measured by its impact on the signal-to-noise ratio of the data-processing pipeline. Signal was measured as the number of FRBs detected and noise was measured as the number of false triggers. Improving the signal-to-noise ratio had to be done within the computational constraints, being real-time performance and minimal additional memory.

The FF-Flagger operates on the assumptions that, compared to signal, RFI is shorter in time, not dispersed and generally less smooth. The results in this thesis show that by using these assumptions the FF-Flagger is able to mitigate a lot of the RFI at WSRT and increase the signal-to-noise ratio of the FRB-detection effort on various different WSRT observations. The component algorithms of the FF-Flagger were compared to the default RFI-mitigation algorithms at WSRT which make up the AOFlagger and were shown to perform considerably better for the application of transient detection at WSRT.

The details of a GPU-accelerated linear time-complexity implementation of the FF-Flagger compatible with Amber were also discussed in this thesis. It was shown that this implementation does adhere to the tenth-of-a-second real-time requirements and requires only 6.3 megabytes of additional memory per GPU.

The thesis includes a version of the FF-Flagger implemented in Amber but the performance of the pipeline in conjunction with Amber has not yet been tested. All of the astronomical results were done using Heimdall because the final version of Amber is still being developed. Two upgrades are needed before the pipeline is ready for use. First, the kernels need to be updated to handle multiple on-sky beams. Second, a method for online tuning of the parameters is required to ensure optimal performance.

The efficacy of the FF-Flagger was demonstrated using data observed at WSRT. The only requirement of the pipeline specific to WSRT is that the data can be sorted using pigeonhole sorting, which requires it to be low-bit integers. Given that this requirement is met by most radio telescopes there is reason to believe that the FF-Flagger will be effective at any one of them, although this claim still has to be proven.

This thesis has conceived and delivered a working implementation of the FF-Flagger that is shown to be a viable way to mitigate RFI at WSRT because it significantly increases the signal-to-noise ratio of transient detection and achieves this within the computational constraints of the data processing pipeline at WSRT.

LIST OF REFERENCES

LIST OF REFERENCES

[1] E. Petroff, E. Barr, A. Jameson, E. Keane, M. Bailes, M. Kramer, V. Morello, D. Tabbara, and W. van Straten, "Frbcat: the fast radio burst catalogue," *Publications of the Astronomical Society of Australia*, vol. 33, 2016.

[2] E. M. Purcell and D. J. Morin, *Electricity and magnetism*. Cambridge University Press, 2013.

[3] G.-A. Valley, "Basics of radio astronomy," 1998.

[4] H. Hirabayashi, H. Hirosawa, H. Kobayashi, Y. Murata, Y. Asaki, I. M. Avruch, P. G. Edwards, E. B. Fomalont, T. Ichikawa, T. Kii, *et al.*, "The vlbi space observatory programme and the radio-astronomical satellite halca," *Publications of the Astronomical Society of Japan*, vol. 52, no. 6, pp. 955–965, 2000.

[5] A. A. Abdo, M. Ackermann, M. Ajello, A. Allafort, E. Antolini, W. Atwood, M. Axelsson, L. Baldini, J. Ballet, G. Barbiellini, *et al.*, "Fermi large area telescope first source catalog," *The Astrophysical Journal Supplement Series*, vol. 188, no. 2, p. 405, 2010.

[6] R. Nan, D. Li, C. Jin, Q. Wang, L. Zhu, W. Zhu, H. Zhang, Y. Yue, and L. Qian, "The five-hundred-meter aperture spherical radio telescope (fast) project," *International Journal of Modern Physics D*, vol. 20, no. 06, pp. 989–1024, 2011.

[7] M. á. van Haarlem, M. Wise, A. Gunst, G. Heald, J. McKean, J. Hessels, A. De Bruyn, R. Nijboer, J. Swinbank, R. Fallows, *et al.*, "Lofar: The low-frequency array," *Astronomy & Astrophysics*, vol. 556, p. A2, 2013.

[8] J. M. Cordes, "Coherent radio emission from pulsars," *Space Science Reviews*, vol. 24, no. 4, pp. 567–600, 1979.

[9] D. R. Lorimer and M. Kramer, *Handbook of pulsar astronomy*, vol. 4. Cambridge university press, 2005.

[10] J. Katz, "Fast radio burstsa brief review: Some questions, fewer answers," *Modern Physics Letters A*, vol. 31, no. 14, p. 1630013, 2016.

[11] E. Gibney, "Why ultra-powerful radio bursts are the most perplexing mystery in astronomy," Jul 2016.

[12] B. Marcote, Z. Paragi, J. Hessels, A. Keimpema, H. van Langevelde, Y. Huang, C. Bassa, S. Bogdanov, G. Bower, S. Burke-Spolaor, *et al.*, "The repeating fast radio burst frb 121102 as seen on milliarcsecond angular scales," *The Astrophysical Journal Letters*, vol. 834, no. 2, p. L8, 2017.

[13] D. Lorimer, M. Bailes, M. McLaughlin, D. Narkevic, and F. Crawford, "A bright millisecond radio burst of extragalactic origin," *Science*, vol. 318, no. 5851, pp. 777–780, 2007.

[14] M. Verheijen, T. Oosterloo, W. Van Cappellen, L. Bakker, M. Ivashina, and J. van der Hulst, "Apertif, a focal plane array for the wsrt," in *AIP Conference Proceedings*, vol. 1035, pp. 265–271, AIP, 2008.

[15] J. van Leeuwen, "Arts–the apertif radio transient system," in *The Third Hot-wiring the Transient Universe Workshop*, pp. 79–79, 2014.

[16] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. Lazio, "The square kilometre array," *Proceedings of the IEEE*, vol. 97, no. 8, pp. 1482–1496, 2009.

[17] A. Sclocco, J. Van Leeuwen, H. E. Bal, and R. V. Van Nieuwpoort, "A real-time radio transient pipeline for arts," in *Signal and Information Processing (GlobalSIP), 2015 IEEE Global Conference on*, pp. 468–472, IEEE, 2015.

[18] L. Connor and J. van Leeuwen, "Applying deep learning to fast radio burst classification," *arXiv preprint arXiv:1803.03084*, 2018.

[19] J. Taylor, N. Denman, K. Bandura, P. Berger, K. Masui, A. Renard, I. Tretyakov, and K. Vanderlinde, "Spectral kurtosis based rfi mitigation for chime," *arXiv preprint arXiv:1808.10365*, 2018.

[20] A. Offringa, A. de Bruyn, S. Zaroubi, and M. Biehl, "A lofar rfi detection pipeline and its first results," *arXiv preprint arXiv:1007.2089*, 2010.

[21] J. Akeret, C. Chang, A. Lucchi, and A. Refregier, "Radio frequency interference mitigation using deep convolutional neural networks," *Astronomy and computing*, vol. 18, pp. 35–39, 2017.

[22] A. Offringa, J. Van De Gronde, and J. Roerdink, "A morphological algorithm for improving radio-frequency interference detection," *Astronomy & Astrophysics*, vol. 539, p. A95, 2012.

[23] M. Kesteven, R. Manchester, A. Brown, and G. Hampson, "Rfi mitigation for pulsar observations," *Proceedings of Science*, vol. 23, pp. 877–885, 2010.

[24] F. Briggs, J. Bell, and M. Kesteven, "Removing radio interference from contaminated astronomical spectra using an independent reference signal and closure relations," *The Astronomical Journal*, vol. 120, no. 6, p. 3351, 2000.

[25] G. Hellbourg, R. Weber, C. Capdessus, and A.-J. Boonstra, "Oblique projection beamforming for rfi mitigation in radio astronomy," in *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, pp. 93–96, IEEE, 2012.

[26] N. Niamsuwan, J. T. Johnson, and S. W. Ellingson, "Examination of a simple pulse-blanking technique for radio frequency interference mitigation," *Radio Science*, vol. 40, no. 5, 2005.

[27] A. R. Offringa, "Algorithms for radio interference detection and removal," 2012.

[28] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial intelligence review*, vol. 22, no. 2, pp. 85–126, 2004.

[29] F. E. Grubbs, "Procedures for detecting outlying observations in samples," *Technometrics*, vol. 11, no. 1, pp. 1–21, 1969.

[30] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," *Journal of Experimental Social Psychology*, vol. 49, no. 4, pp. 764–766, 2013.

[31] B. Winkel, J. Kerp, and S. Stanko, "Rfi detection by automated feature extraction and statistical analysis," *Astronomische Nachrichten: Astronomical Notes*, vol. 328, no. 1, pp. 68–79, 2007.

[32] H. Mark, "Optimizing parallel reduction in cuda," *NVIDIA CUDA SDK*, vol. 2, 2008.

[33] L. Connor, "Arts-analysis," *GitHub Repository:liamconnor/arts-analysis*, 2018.