RHODES UNIVERSITY
Where leaders learn

Department of Physics & Electronics

Finite Precision Arithmetic in Polyphase Filterbank Implementations

A thesis submitted in fulfilment of the requirement for the degree of

Master of Science in Physics & Electronics

of

Rhodes University

by

Talon Myburgh

Supervised by:
Professor Justin Jonas

Abstract

The MeerKAT is the most sensitive radio telescope in its class, and it is important that systematic effects do not limit the dynamic range of the instrument, preventing this sensitivity from being harnessed for deep integrations. During commissioning, spurious artefacts were noted in the MeerKAT passband and the root cause was attributed to systematic errors in the digital signal path. Finite precision arithmetic used by the *Polyphase Filterbank* (PFB) was one of the main factors contributing to the spurious responses, together with bugs in the firmware. This thesis describes a software PFB simulator that was built to mimic the MeerKAT PFB and allow investigation into the origin and mitigation of the effects seen on the telescope. This simulator was used to investigate the effects in signal integrity of various rounding techniques, overflow strategies and dual polarisation processing in the PFB. Using the simulator to investigate a number of different signal levels, bit-width and algorithmic scenarios, it gave insight into how the periodic dips occurring in the MeerKAT passband were the result of the implementation using an inappropriate rounding strategy. It further indicated how to select the best strategy for preventing overflow while maintaining high quantization efficiency in the FFT.

This practice of simulating the design behaviour in the PFB independently of the tools used to design the DSP firmware, is a step towards an end-to-end simulation of the MeerKAT system (or any radio telescope using finite precision digital signal processing systems). This would be useful for design, diagnostics, signal analysis and prototyping of the overall instrument.

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree and that it represents my own work. I know the meaning of plagiarism and declare that all of the work in the thesis, save for that which is properly acknowledged, is my own.

Talon Myburgh

Supervised by:
Professor Justin Jonas

Acknowledgements

I would like to thank:

My mother and father for their support and care throughout these two years of my Masters.

Andrew Martens for answering all of my questions regarding the MeerKAT F-Engine. I further thank him for his assistance in developing the Simulink design files used to test the simulator.

Ludwig Schwardt for his help in the early development of the Python simulator and especially his assistance with the developed Python fixed point number system.

Dr Jason Manley and the SARAO DSP team for giving me a desk in their office and always being open to answering any of my questions.

Wesley New for teaching me how to get up and running with the CASPER tool-flow and for his help in setting my machine up with Vivado and Simulink.

Marcel Gouws for sharing how his Python F-Engine simulator was developed and providing the idea for the shift-scheme versus quantisation efficiency test as part of my research.

Francois Kapp for setting me up with the SARAO engineering crowd.

The CASPER community for being so interested in my research, inviting me to their many conferences around the world and teaching me so much more than I could have ever learned if I were doing this research outside their community.

Professor Oleg Smirnov and the RATT team from Rhodes University for funding my entire Masters degree, keeping me in their midst and being interested in my work despite my research move towards engineering.

This project garnered so much interest from so many professionals that listing them all here would not be possible. However, my biggest thanks is reserved for last. Before being my supervisor, Professor Justin Jonas was my DSP and Radio Astronomy lecturer. It was in his classes that I developed such an interest for this research field. This degree took many twists and turns which predominantly resulted in incredible opportunities. It was a privilege to have a two-year-long dialogue with Professor Jonas about all things MeerKAT and DSP. Thank you for having me as your student and for the time you took to carefully supervise my work.

Contents

.

List of Figures

CHAPTER 1

Introduction

The South African MeerKAT radio telescope is a precursor to the Square Kilometre Array (SKA) telescope and is to be integerated into SKA Phase 1 [23]. The MeerKAT is the most sensitive radio telescope in its class and it is essential that internal systematic effects do not limit its ability to detect faint radio sources. Special effort has been made to ensure that the telescope site has minimal *Radio Frequency Interference* (RFI) by mitigating both internal and external emission. The digitiser and receiver are built to pass accurate, high dynamic range data to the digital correlator/beamformer. Despite the high precision of the overall instrument, slight systematic inaccuracies were noted in the MeerKAT passband and were attributed to errors in the digital signal path. These were discovered to be as a result of the finite precision arithmetic used by the *Polyphase Filterbank* (PFB) (a key component of the correlator/beamformer), which prompted this research.

Two different effects were noticed during MeerKAT commissioning:

1. Small narrow-band dips in the receiver passband, occuring at regular frequency intervals.
2. Spurious bandpass distortions that were attributed to a loss in signal path dynamic range.

During the course of this investigation it was found that:

1. The dips in the spectrum noise floor returned by the MeerKAT F-Engine were caused by the rounding scheme in use. This is a *least significant bit* (LSB) effect that only becomes prominent with long time integrations.
2. The dynamic range of the signal path is sensitive to the input signal level, the spectral shape of the signal and the strategy used to prevent arithmetic overflow caused by narrow-band RFI, in the signal processing chain. Overflow is a *most significant bit* (MSB) effect that corrupts the PFB output since it leaks between channels.

Initially, these effects were to be tested by direct simulation of MeerKAT's PFB design in Simulink. It was realised however, that a far more valuable test would be conducted on an independent platform that was designed to mimic the MeerKAT PFB. Not only would this test the function of MeerKAT's PFB, but it could be designed to make analysis and visualisation of signal flow through the PFB possible. This lead to the development of two Python simulators (one floating point and one fixed point) that were used for the majority of the tests conducted for this thesis.

Besides testing varying rounding and shift schemes, these simulators allow for quick testing of how these effects vary for different input signals and their levels as well as PFB designs. This is mostly done by using the floating (which serves as the ideal) and fixed point (which introduces finite precision effects) simulators in conjunction and comparing the results for the same input. This way, one can discriminate between numerical effects and actual *digital signal processing* (DSP) bugs.

This practice of simulating the design behaviour in the PFB independently is something that could be extended to the entire MeerKAT data path. Should the designed simulator function to within a certain accuracy, it could serve in an end-to-end simulation of the MeerKAT system which would be incredibly useful for diagnostics, signal analysis and prototyping.

CHAPTER 2

Radio Interferometry

This Chapter starts by introducing the principles of observational radio astronomy with a focus on the practice of radio interferometry. From this, the fundamental interferometric measurement quantity, the visibility, is introduced. Thereafter, we address instrumentation (specifically that used by MeerKAT) and work to show the reader, that the correlator is fundamental to the measurement of the visibility through the use of the two processes: frequency channelisation and correlation.

Leaving radio astronomy momentarily, the concept of finite precision arithmetic is introduced. This is done by first covering the theory of binary number systems and then the practices used in mitigating inherent quantisation error in digital data. Having introduced the X and F-Engines that perform correlation and frequency channelisation respectively in MeerKAT, we isolate the F-Engine as being more sensitive to numerical inaccuracy due to coherent systematic effects.

While both beamforming and cross-correlation operations make use of the F-Engine, the science goals for the beamformer are not as sensitive to F-Engine error as the imaging process is. For this reason, this thesis investigates the effects of finite precision error introduced by MeerKAT's F-Engine when performing auto- and cross-correlation.

2.1 Radio Astronomy

The four quantities that fully describe the polarisation state of an electromagnetic wave are know as the Stokes parameters:

$$\begin{aligned} &I(\theta,\phi,\nu,t) \\ &Q(\theta,\phi,\nu,t) \\ &U(\theta,\phi,\nu,t) \\ &V(\theta,\phi,\nu,t) \end{aligned} \tag{2.1}$$

where ϕ and θ are sky coordinates, ν the telescope observing bandwidth and t the time.

Since these parameters may be used to characterise the radio sky, polarisation is a very important measurement despite some astrophysical radio emissions being inherently un-polarised [1]. While some radio telescopes sense left and right circular polarisations, MeerKAT uses an *ortho-mode transducer* (OMT) to sense the two orthogonal linear electric field components:

$$\begin{aligned} E_x &= e_x(t)\cos(\omega t + \delta_x) \\ E_y &= e_y(t)\cos(\omega t + \delta_y) \end{aligned} \tag{2.2}$$

These may be manipulated to gain the four Stokes parameters I, Q, U and V as:

$$\begin{aligned} I &= \langle E_x E_x^* + E_y E_y^* \rangle \\ Q &= \langle E_x E_x^* - E_y E_y^* \rangle \\ U &= \langle E_x E_y^* + E_y E_y^* \rangle \\ V &= i\langle E_x E_y^* - E_y E_x^* \rangle \end{aligned} \tag{2.3}$$

where (*) represents complex conjugation and $\langle ... \rangle$ a time averaging. I is a measure of the waves total power, the linearly polarized components are represented by Q and U and the circularly polarized component by V [1].

For a single radio dish (depicted in figure 2.1), the antenna temperature is given by:

$$T_A = \frac{\int\int T_B(\theta,\psi)P_n(\theta,\psi)\sin\theta d\theta d\psi}{\int\int P_n(\theta,\psi)\sin\theta d\theta d\psi} \ [K] \tag{2.4}$$

where P_n is the normalised power polar pattern of the antenna and

$$T_B = \frac{c^2 I_\nu}{2k\nu^2} \ [K] \tag{2.5}$$

is the sky brightness temperature. The antenna temperature for a single polarisation component is defined to be:

$$T_A = \frac{P_A}{k\Delta\nu} \ [K] \tag{2.6}$$

where P_A is the power at the appropriate OMT terminal.

The relationship shown in equation 2.4 implies that the observed antenna temperature is a convolution of the sky brightness temperature with the beam pattern [2].

The angle between the half power points that specifies the angular width of the main beam of the antenna, is referred to as the *half-power beamwidth* (HPBW) and is approximately given by the relation:

$$\theta_{HPBW} \approx \frac{\lambda}{D} \text{ [rad]} \tag{2.7}$$

where λ is the observing wavelength and D the diameter of the observing aperture [3]. Given the relation in equation 2.7, the beam size dictates the angular resolution of the telescope. Higher angular resolution (smaller θ_{HPBW}) for a given observing wavelength depends on the size of the dish/aperture. Practical constraints limit the size of single dish apertures and as such, the theory of radio interferometry was introduced, and with it, much higher angular resolution.

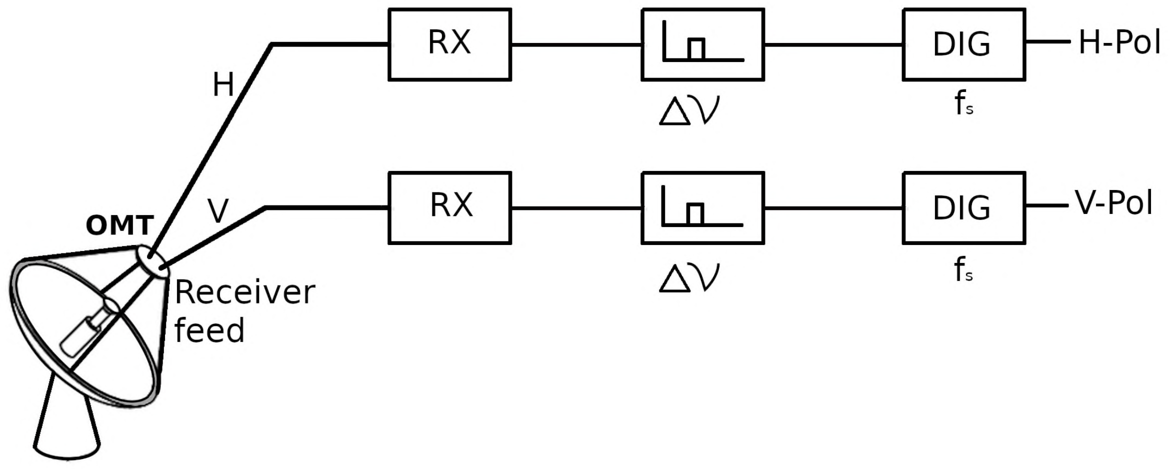

Figure 2.1: The MeerKAT receiver feed on each dish is sensitive to the two linear polarisation components H and V. Each flows through a separate receiver RX, that contains a cryo-cooled low-noise amplifier. Then, each passes through a band-pass filter (depending on the receiver band under observation) and finally the two orthogonal polarisation signals are sampled at the sampling frequency f_s and digitised.

2.2 Direct Digitization

For a band-limited signal, the Nyquist theorem states that no information is lost by the sampling process, if the sampling frequency is high enough. Sampling a time-domain signal causes aliasing in the frequency domain, which can cause detrimental effects if the aliased spectral components overlap. For a baseband (low-pass) rectangular spectrum with upper cut-off frequency ν_c, the full spectrum width is $2\nu_c$ (with negative frequencies included). Samples in time of at most $\frac{1}{2\nu_c}$ fully specify the function. This critical rate frequency of $2\nu_c$ is known as the *Nyquist rate* [4].

In some systems, frequency down-conversion is performed by using a mixer and local oscillator at frequency ν_{LO}. This is referred to as heterodyning. MeerKAT is a non-heterodyne system and instead uses aliasing to its advantage. Every frequency interval $\frac{f_s}{2}$ (where f_s is the sampling frequency) starting at DC, is considered a Nyquist zone. By sampling in the 2nd Nyquist zone as MeerKAT does, high frequency *radio frequency* (RF) signals are aliased into the first Nyquist zone (see figure 2.2) [5]. This technique requires a band-pass Nyquist filter to prevent spectral leakage, where the Nyquist criterion is that $\frac{f_s}{2} > \Delta\nu$ where $\Delta\nu$ is the bandwidth of the filter. In this way, the original RF signal is converted to a baseband signal, as would be the case for a heterodyne system.

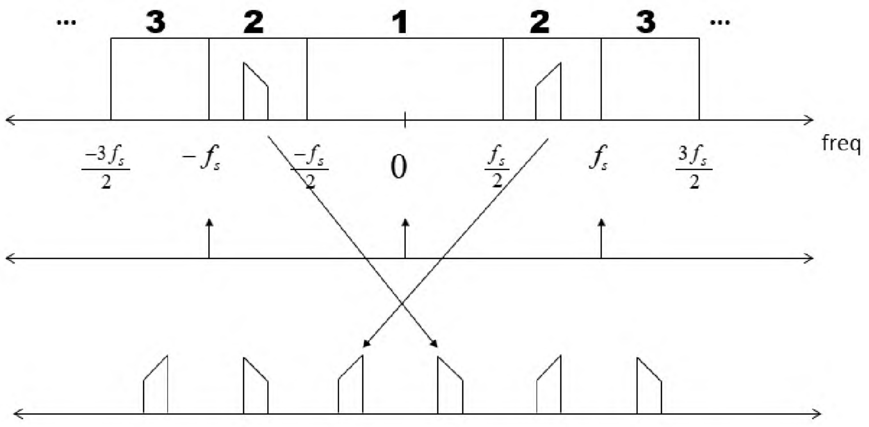

Figure 2.2: Diagram illustrating the aliasing of negative and positive frequencies (row 1) into the first Nyquist zone (row 3) by using second zone Nyquist sampling (see delta functions in row 2). f_s is the sampling frequency and zones are defined in increments of $f_s/2$ from DC. Copywrite: Dan Boschen, 2019.

2.3 Analytic Signals

A receiver outputs a real voltage $V_R(t)$ for each polarisation and therefore the digitiser records associated real digital signals. Later calculations of correlation products require complex-valued voltages $V_1(t)$ and $V_2(t)$. Therefore, the real signal produced by the digitiser must be transformed to a complex signal whilst not losing/adding any additional information.

A real voltage signal may be represented by the standard Fourier synthesis equation:

$$V^{(r)}(t) = \int_{-\infty}^{\infty} v(\nu) e^{-2\pi i \nu t}\, d\nu \tag{2.8}$$

Because $V^{(r)}(t)$ is real, its spectrum is Hermitian, i.e. $v(-\nu) = v^{\star}(\nu)$ and hence it has redundancy. Discarding the negative frequency components, we can define a new complex-valued voltage function

$$\begin{aligned} V(t) &= 2\int_{0}^{\infty} v(\nu) e^{-2\pi i \nu t}\, d\nu \\ &= \int_{-\infty}^{\infty} v(\nu) e^{-2\pi i \nu t}\, d\nu - i\int_{-\infty}^{\infty} i\,\mathrm{sgn}(\nu) v(\nu) e^{-2\pi i \nu t}\, d\nu \\ &= V^{(r)}(t) + iV^{(i)}(t) \end{aligned} \tag{2.9}$$

where $V^{(i)}(t) = -\int_{-\infty}^{\infty} i\,\mathrm{sgn}(\nu) v(\nu) e^{-2\pi i \nu t}\, d\nu$ is the Hilbert transform of $V^{(r)}(t)$.

Because $v(\nu)$ is Hermitian, it is easy to show that $i\,\mathrm{sgn}(\nu)v(\nu)$ is also Hermitian and hence, $V^{(i)}(t)$ is real valued.

It can be shown that:

$$V^{(i)}(t) = \frac{1}{\pi}\int_{-\infty}^{\infty} \frac{V^{(r)}(\tau)}{t-\tau}\, d\tau \tag{2.10}$$

where $V^{(i)}(t)$ and $V^{(r)}(t)$ are both real-valued. In this way, $V(t)$ can be represented by two real-valued voltages [6].

Examining equation 2.10, it can be seen that the Hilbert transform corresponds to a $-90°$ phase shift of the original signal. In physical electronic circuitry, the generation of an analytic signal may be done as indicated by figure 2.3.

The generation of a complex-valued analytic signal is a natural consequence of the action of the Fourier Transform in the F-Engine since a *digital signal Fourier transform* (DFT) can be considered to be and I/Q mixer.

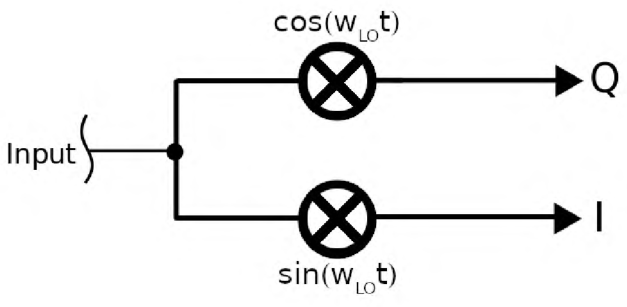

Figure 2.3: Hilbert transformer used to convert a real input signal to an analytic signal with real *in-phase* part I and imaginary *quadrature phase* part Q. For heterodyne systems, the $-90°$ phase shift is accomplished using quadrature local oscillator signals and an I/Q mixer which mixes with sinusoids $-90°$ out of phase.

2.4 The van Cittert-Zernike Theorem

The practice of radio interferometry is based on the van Cittert-Zernike theorem, a result from classical optics.

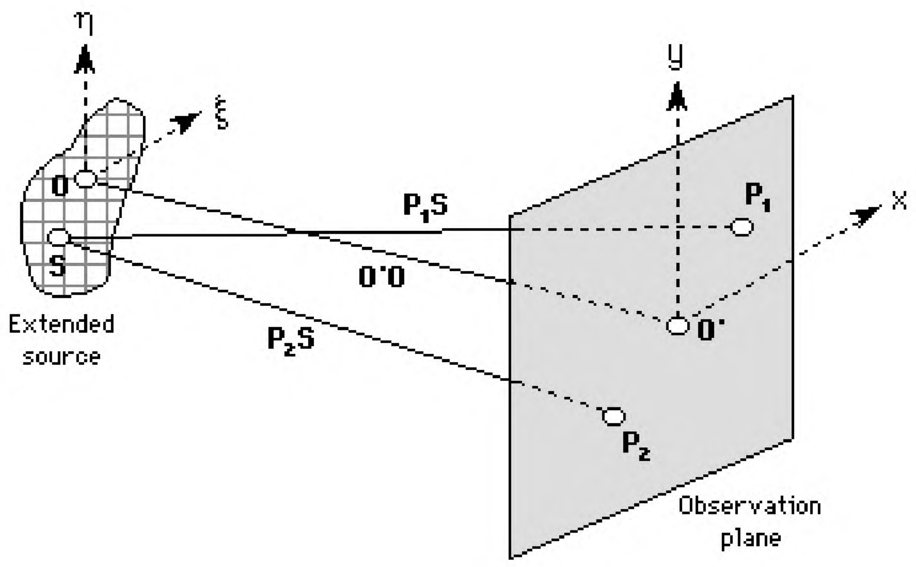

Figure 2.4: Illustrating the van Cittert-Zernike theorem. Sourced from: `http://people.seas.harvard.edu/~jones/ap216/lectures/ls_3/ls3_u6A/ls3_unit6A.html`

Consider the geometry of figure 2.4 showing an extended radio source $S(\zeta, \eta)$ and two receptors P_1 and P_2 in a plane parallel to the source plane. The van Cittert-Zernike theorem states that:

$$J(P_1, P_2) = \int_{\sigma} I(s) \frac{e^{i\bar{k}(R_1 - R2)}}{R_1 R_2} dS \tag{2.11}$$

where $\bar{k} = 2\pi\bar{\nu}/c$ is the wave number in a vacuum and $J(P_1, P_2)$ is the mutual coherence of the waves $E_1(t)$ and $E_2(t)$ detected at P_1 and P_2 [7, Ch. 10]. This mutual coherence is also calculated as:

$$J(P_1, P_2) = \langle E_1(t), E_2(t)^* \rangle \tag{2.12}$$

where $\langle \dots \rangle$ is a time average. In general, the quantity in equation 2.12 is complex-valued.

Under the Fraunhofer approximation [1], using direction cosines as sky coordinates (l, m) and $u = \frac{x}{\lambda}$, $v = \frac{y}{\lambda}$, equation 2.11 reduces to

$$J(P_1, P_2) = \frac{e^{i\psi} \int \int_\sigma I(l,m) e^{-i2\pi(ul+vm)} \, \mathrm{d}l \, \mathrm{d}m}{\int \int_\sigma I(l,m) \, \mathrm{d}l \, \mathrm{d}m} \tag{2.13}$$

where ψ is a phase factor that depends on the origin chosen for the (l, m) coordinate system.

Equation 2.13 shows that under the required assumptions, the mutual coherence is equal to the normalised Fourier transform of the intensity function of the source at a single point corresponding to the relative positions of P_1 and P_2 [7, Ch. 10]. By moving P_1 and P_2 around in the (x, y) plane, the Fourier transform of the image can be sampled to arbitrary resolution. This allows the source image to be derived via an inverse Fourier transform.

In practice, the radio interferometer records the complex measure

$$V_{12} = \langle V_1(t) V_2^*(t) \rangle \tag{2.14}$$

where $V_1(t)$ and $V_2(t)$ are the voltages measured at the two receptors corresponding to the wave E-fields $E_1(t)$ and $E_2(t)$. Equation2.14 is called the visibility and the following section details this further.

[1] $R > D^2/\lambda$, where R is the distance to the source, D, the largest aperture in the array and λ, the wavelength of the measured light

2.5 Simple Interferometer

The van Citter-Zernike theorem provides a basis for a method of using an array of radio antennas observing simultaneously, to simulate a very large and incompletely-filled aperture. This is known as radio interferometry [8]. Modern radio astronomy interferometers operate over a frequency range of 10MHz to 1THz. Radio astronomy has the benefit of being able to sample amplitudes and phases of waves rather than dealing with quantum phenomena. This is as a result of the frequencies (ν) under observation being such that $h\nu << kT$ [2]. Hence, one receives ample photons from sources $> 10\,\mathrm{K}$ [3]. To simultaneously measure as many Fourier components of the data as possible, a large number of diverse baselines are needed to sufficiently sample the visibility plane and hence be able to reconstruct an image. For this reason, radio interferometers employ arrays of antennas so that at any instant there are $N(N-1)/2$ baselines.

Single dish instruments have an angular resolution $\approx \lambda/D$ (see equation 2.7). With interferometry, the angular resolution is estimated by λ/B where B is the largest baseline (spacing) between two elements in an array. Hence, observing with radio interferometers enables the imaging of complex source morphologies with angular resolution scales $\approx 10^{-4}$ arc-seconds. Such angular resolution measures are significantly higher than that achievable with optical/near-infrared astronomy [3].

For the analysis that follows it is sufficient to consider a single baseline defined by two arbitrary receptors forming a two element interferometer. Consider the two-element interferometer shown in figure 2.5. The correlator block implements equation 2.14. Working under the assumption that the radio emission received by any two points in space is received as a plane wave (due to the large distance between the source and observer), a time delay (τ_g) will occur between the two measured signals. This is known as the geometric time delay and is given by

$$\tau_g = \frac{\vec{B}_{ij} \cdot \hat{s}}{c}, \tag{2.15}$$

where $\vec{B}_{ij}$ is the baseline vector (pointing from dish i to dish j) and $\hat{s} = (l, m, \sqrt{1 - l^2 - m^2})$ is the source unit vector with l and m being sky coordinates represented as direction cosines [9].

For a given frequency, it is convenient to describe the baseline in a new coordinate system defined by:

$$(u, v, w) = \frac{\vec{B}}{\lambda} \tag{2.16}$$

These dimensionless coordinates are measured in spatial frequencies and describe baselines in the uv plane. For the purpose of this analysis, we will assume a co-planar array, and hence assume $w = 0$. This is consistent with the assumptions made for the van Citter-Zernike theorem (see section 2.4). Noting the time delay experienced between the two dishes, we may ascribe a phase difference between the two dishes as:

$$\phi = -2\pi i \tau_g \nu = -2\pi i \frac{\vec{B} \cdot \hat{s}}{\lambda} \tag{2.17}$$

[2]where h is the Planck constant, k is the Boltzmann constant and T is the characteristic temperature of objects being observed in Kelvin.

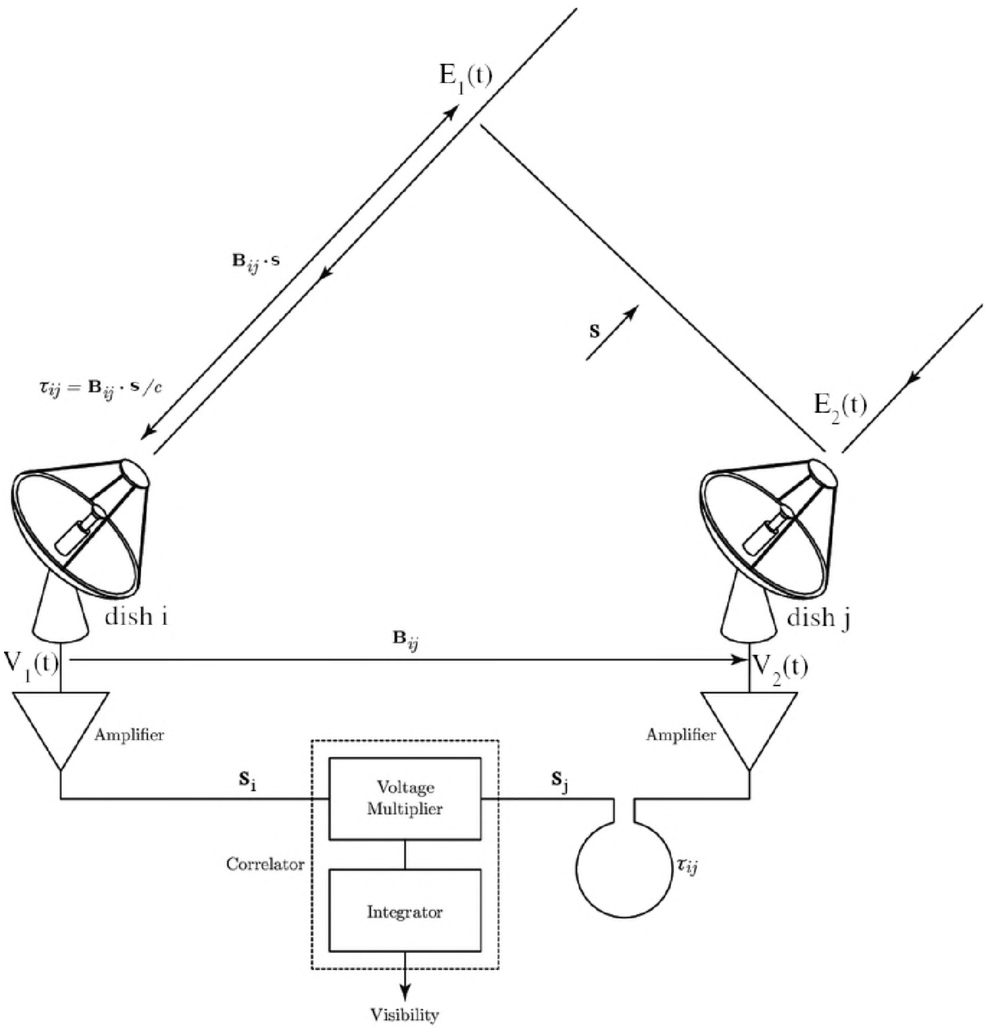

Figure 2.5: Two element interferometer with dishes i and j separated by baseline $\vec{B_{ij}}$ both pointing in direction of source vector $\hat{s}$. The correlator enclosed in the dotted line box, shows the production of a visibility from the two signals s_i and s_j. Diagram sourced from: `https://www.researchgate.net/figure/Two-element-interferometer_fig2_51963705` on 13/02/2019.

The projection of the baseline vector ($\vec{B}$) on the source direction unit vector ($\hat{s}$) may be expanded to

$$\frac{\vec{B} \cdot \hat{s}}{\lambda} = ul + vm$$

rendering equation 2.17 as

$$\phi = -2\pi i(ul + vm) \tag{2.18}$$

From equation 2.13:

$$\begin{aligned} V_{ij}(\nu) &\propto e^{i\phi} \int\int I(l,m) e^{-i2\pi(ul+vm)}\, d\sigma \\ &\propto e^{i\phi} \int\int I(l,m) e^{-i2\pi \frac{\vec{B}\cdot\vec{s}}{\lambda}}\, d\sigma \\ &\propto e^{i\phi} \int\int I(l,m) e^{-i2\pi\tau_g\nu}\, d\sigma \end{aligned} \tag{2.19}$$

The telescope component used to compute the visibility in equation 2.19 is known as the correlator [10]. If the delay $\tau_{ij} = \tau_g$ for a given reference direction (phase centre [3], indicated by $\hat{s}$ in figure 2.5), the output from the correlator is proportional to the visibility given in equation 2.19. Therefore,

$$\langle s_i, s_j^* \rangle \propto V_{ij}(\nu) \propto \int\int I(l,m) e^{-i2\pi\Delta\tau_g\nu}\, d\sigma \tag{2.20}$$

where $\langle s_i, s_j^* \rangle$ is the cross-correlation of signals s_i and s_j (shown in figure 2.5) and $\Delta\tau_g = \tau_g - \tau_{ij}$ is the residual delay for directions away from the phase centre.

2.5.1 Finite Bandwidths and Delay Tracking

Equation 2.20 for quasi monochromatic interferometers does not allow for finite bandwidths which are necessary for practical interferometers. For a constant source brightness and interferometer response across a small frequency range $\Delta\nu$ centred on ν_c, equation 2.20 generalises to:

$$V_{ij}(\nu) = \int \left[(\Delta\nu^{-1}) \int_{\nu_c - \Delta\nu/2}^{\nu_c + \Delta\nu/2} I_\nu(l,m) e^{-2\pi i \Delta\tau_g \nu}\, d\nu \right] d\sigma$$

The integral in square brackets is the Fourier transform of a rectangle function and therefore:

$$V_{ij}(\nu) = \int I_\nu(l,m) \mathrm{sinc}(\Delta\nu\Delta\tau_g) e^{-2\pi i \tau_{ij}\nu_c}\, d\sigma \tag{2.21}$$

Equation 2.21 indicates how the fringe amplitude for a single baseline is attenuated by the factor $\mathrm{sinc}(\Delta\nu\tau_{ij})$. Eliminating this attenuation for any phase centre ($\hat{s}$), is done by introducing a compensating delay $\tau_0 = \tau_{ij}$ in the signal path of the reference antenna as shown by figure 2.5. With varying phase centre, τ_0 must be continuously re-calculated to track τ_{ij} within a tolerance $|\tau_0 - \tau_{ij}| << (\Delta\nu)^{-1}$ [3]. This is usually done with digital electronics and is further explained in subsections 2.7.3.1 and 2.7.3.2.

This delay tracking only applies to the phase centre. For positions away from the phase centre, there is a residual geometric delay $\Delta\tau_{ij}$. Hence, to allow wide-field imaging, $\Delta\nu$ needs to be small to ensure $\Delta\tau_{ij} << \Delta\nu^{-1}$.

Following this, for wideband, wide-field interferometry, the signal needs to be channelised to reduce $\Delta\nu$ to avoid decorrelation caused by the severe attenuation introduced by the sinc factor [3].

[3]Where $l = 0, m = 0$.

2.6 Wideband Correlators

Mathematically, a correlation is defined as:

$$V_1(\tau) \star V_2(\tau) = \int_{-\infty}^{\infty} V_1^*(t-\tau)V_2(t)dt \tag{2.22}$$

where $V_1^*(t)$ denotes the complex conjugate of the voltage at receiver 1. It is related to the convolution function by

$$V_1(t) * V_2^*(-t) = V_1(t) \star V_2(t) \tag{2.23}$$

If, for figure 2.5, we assume that both dishes deliver the same voltage $V(t)$ to the correlator and that one lags the other by time delay $\tau_g - t$, then:

$$\begin{aligned} s_j &= V(t) \\ s_i &= V(\tau_g - t) \end{aligned}$$

Should the correlator be a 'lag' correlator and should it integrate for $2T$ seconds, it will produce the un-normalised auto-correlation:

$$s_j \star s_i = \lim_{T\to 0} \frac{1}{2T} \int_{-T}^{T} V(t)V(t-\tau_g)\,dt \tag{2.24}$$

If the correlation is required in the frequency domain, the Fourier transform of $s_j \star s_i$ from equation 2.24 is taken. This is known as an XF[4] or spectroscopic lag correlator.

The squared amplitude of a frequency spectrum is known as the power density spectrum, and the power spectrum of a signal is the Fourier transform of the auto-correlation function of that signal. This is known as the Wiener–Khinchin relation (see equation 2.25) and requires that the input signals are deterministic or statistical in nature [4].

$$\begin{aligned} &|H(\nu)|^2 = \int_{-\infty}^{\infty} r(\tau)e^{-i2\pi\nu\tau}\,d\tau \\ &\text{where} \\ &r(\tau) = \int_{-\infty}^{\infty} |H(\nu)|^2 e^{i2\pi\nu\tau}\,d\nu \end{aligned} \tag{2.25}$$

where $H(\nu)$ is the amplitude (voltage) response, and hence $|H(\nu)|^2$ is the power spectrum of the signal input to the correlator [4].

However, more applicable to interferometry is the relation produced when cross-correlating two differing waveforms:

[4]Where X is the multiply and integrate stage, and F the Fourier transform stage. Its ordering dictates which operation is done first.

$$\begin{aligned}
&s_j = V_1(t) \\
&s_i = V_2(\tau_g - t) \\
&\text{where} \\
&V_1(t) \neq V_2(t)
\end{aligned}$$

Again, should a lag correlator, that integrates for $2T$ seconds be used, its response for a given baseline would now be:

$$s_j \star s_i = \lim_{T \to 0} \frac{1}{2T} \int_{-T}^{T} V_1^*(t-\tau) V_2(t) dt \tag{2.26}$$

In practice, this integration time in equation 2.26 is a few seconds or minutes, but it is long compared to both the period and reciprocal bandwidth of both waveforms [4].

A further result is explored where if,

$$\begin{aligned}
V_1(t) &= \int_{-\infty}^{\infty} \mathscr{V}_1(\nu) e^{-2\pi i \nu t} d\nu \\
V_2(t) &= \int_{-\infty}^{\infty} \mathscr{V}_2(\nu) e^{-2\pi i \nu t} d\nu
\end{aligned}$$

where $\mathscr{V}$ denotes the Fourier transform of V, then

$$\begin{aligned}
V_1(t) \star V_2(t) &= \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} \mathscr{V}_1^*(\nu) e^{-2\pi i \nu t}\, d\nu \int_{-\infty}^{\infty} \mathscr{V}_2(\nu') e^{-2\pi i \nu' (t+\tau)}\, d\nu' \right] d\tau \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathscr{V}_1^*(\nu) \mathscr{V}_2(\nu') e^{-2\pi i \tau (\nu' - \nu)} e^{-2\pi i \nu' t}\, d\tau\, d\nu\, d\nu' \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathscr{V}_1^*(\nu) \mathscr{V}_2(\nu') e^{-2\pi i \nu'} \left[\int_{-\infty}^{\infty} e^{-2\pi i \tau (\nu' - \nu)}\, d\tau \right] d\nu\, d\nu' \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathscr{V}_1^*(\nu) \mathscr{V}_2(\nu') e^{-2\pi i \nu' t} \delta(\nu' - \nu)\, d\nu'\, d\nu \\
&= \int_{-\infty}^{\infty} \mathscr{V}_1^*(\nu) \mathscr{V}_2(\nu) e^{-2\pi i \nu t}\, d\nu \\
&= \mathscr{F}^{-1} [\mathscr{V}_1^*(\nu) \mathscr{V}_2(\nu)]
\end{aligned} \tag{2.27}$$

The result of equation 2.27 is known as the correlation theorem [11].

By theorem 2.27, the following relation is observed:

$$\mathscr{F}[V_1(t) \star V_2(t)] = \mathscr{V}_1^*(\nu) \times \mathscr{V}_2(\nu) \tag{2.28}$$

The left-hand side of equation 2.28 is an XF or spectroscopic lag correlator, while the right-hand side is what is known as an FX correlator.

2.7 Correlator Design

The function of a correlator is to perform the auto or cross-correlation of voltage signals from a single or pairs of antennas. Within an array of N dishes, there are $N(N-1)/2$ pairs, and as such, cross-correlation generates more output data streams than input, but generally at a lower data rate [11].

Calculating spectral visibilities $\mathscr{V}_{i,j}(\nu_k)$ involves a Fourier transform stage 'F' and cross-correlation stage 'X' (see section 2.6). The relative ordering of these stages results in two correlator variants: *XF* and *FX* correlators [11].

2.7.1 XF correlator

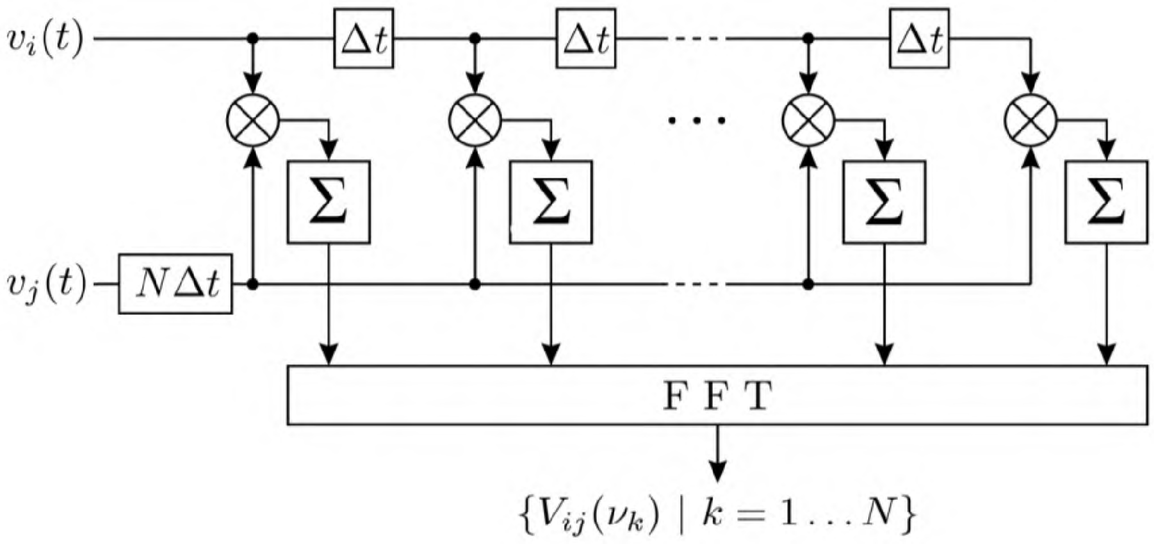

Figure 2.6: XF correlator design. Sourced from `http://www.atnf.csiro.au/research/radio-school/2011/talks/RAS-correlators.pdf`

This correlator first cross-correlates the signals, then Fourier transforms the result (since spectral data is required by scientists). This operation is a direct implementation of equation 2.26 with a Fourier transform applied to the result. Figure 2.6 shows this operation as a spectroscopic 'lag' correlation circuit. One signal is conjugated and delayed relative to the other, before multiplying the two signals for differing values of $\Delta t/\tau$. The result of each product is then integrated for some amount of time and presented to a point on an N-point[5] DFT, where the 'F' stage is applied [10].

Early correlators used the XF architecture because the bit-growth provided by the accumulators allowed for very low precision input digital data (i.e. low resolution *analogue to digital converters* (ADCs) and multipliers), and the FFT only had to be calculated once per integration time. This suited the limitations of the available hardware, i.e. ADCs and compute.

Given linear processing, equation 2.28 indicates that the two architectures are equivalent. In practice, however, the XF correlator is often not RFI robust due to non-linear effects. These

[5]'Point' is the term used to refer to the size or length of the Fourier transform.

effects manifest due to the XF correlator not localising RFI to a single channel and allowing it to corrupt the whole output spectrum. Furthermore, the XF correlator does not scale well with many dishes. With the advancements in technology that enabled higher resolution ADCs and compute logic as well as faster DFT algorithms, the FX correlator architecture became largely adopted.

2.7.2 FX correlator

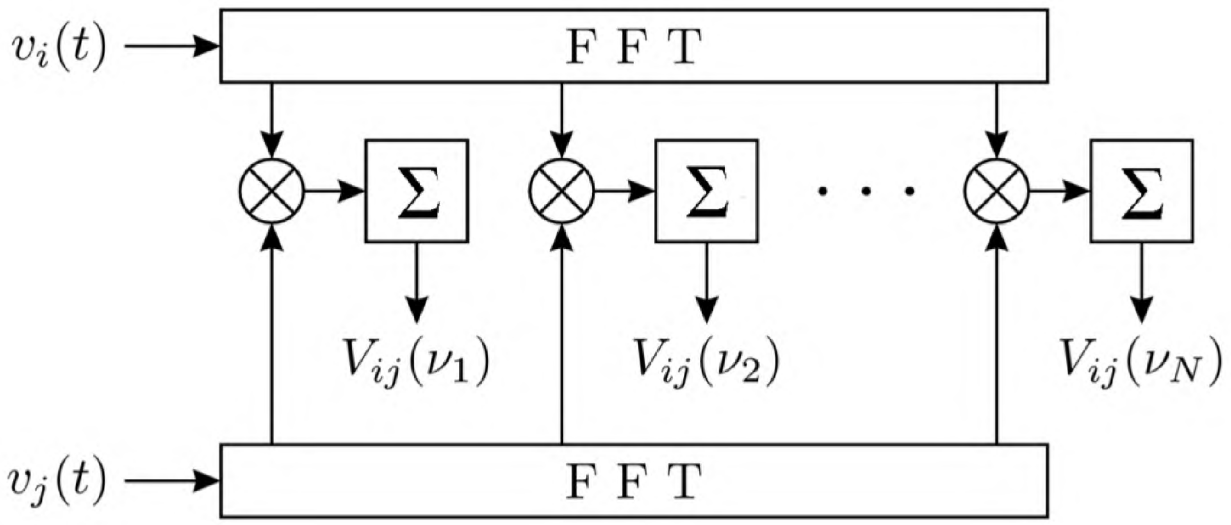

Figure 2.7: FX correlator design. Sourced from `http://www.atnf.csiro.au/research/radio-school/2011/talks/RAS-correlators.pdf`

By equation 2.28, one is able to form a correlation in the frequency domain by multiplying the Fourier transform and conjugate of $V_1(t)$ with the Fourier transform of $V_2(t)$. Contrary to the XF correlator, here the 'F' stage is applied first and then the 'X'. Figure 2.7 represents this. A desirable feature of this architecture, is that the output will inherently produce spectral data (unlike the XF correlator with must be post Fourier transformed). Furthermore, in an FX correlator all that is required for the 'X' stage, is to multiply the spectrum from a single antenna channel-wise with another antennas spectrum [11]. In large arrays, this property makes FX correlators scale better than XF correlators. Furthermore, narrow-band RFI is localised to its channel, enabling scientists to flag it and make use of the remaining channels for science. Since MeerKAT makes use of an FX correlator, this thesis will only cover its operation in depth.

Figure 2.8 shows context as to how the FX correlator is situated for a given baseline in MeerKAT. Ignoring the receiver and digitiser (see figure 2.1), from each antenna, two digital streams are output (H and V polarisations). The antenna will have a coarse delay inserted into its path to correct for the geometric delay relative to a reference antenna. Both feeds are then passed into an F-Engine which performs the F-stage of the FX correlator. Thereafter, a fine delay is performed on the signal to further correct for the geometric delay. Finally, the multiply-accumulate (MAC) operation is performed in the X-Engine. Unlike a standard MAC function, this one will output four values $X_1X_2^*$, $X_1Y_2^*$, $X_2^*Y_1$ and $Y_1Y_2^*$ which are used in the recovering of the Stokes parameters I, Q, U and V (see equation 2.3). Each product is complex because they are the cross-product between two complex spectra.

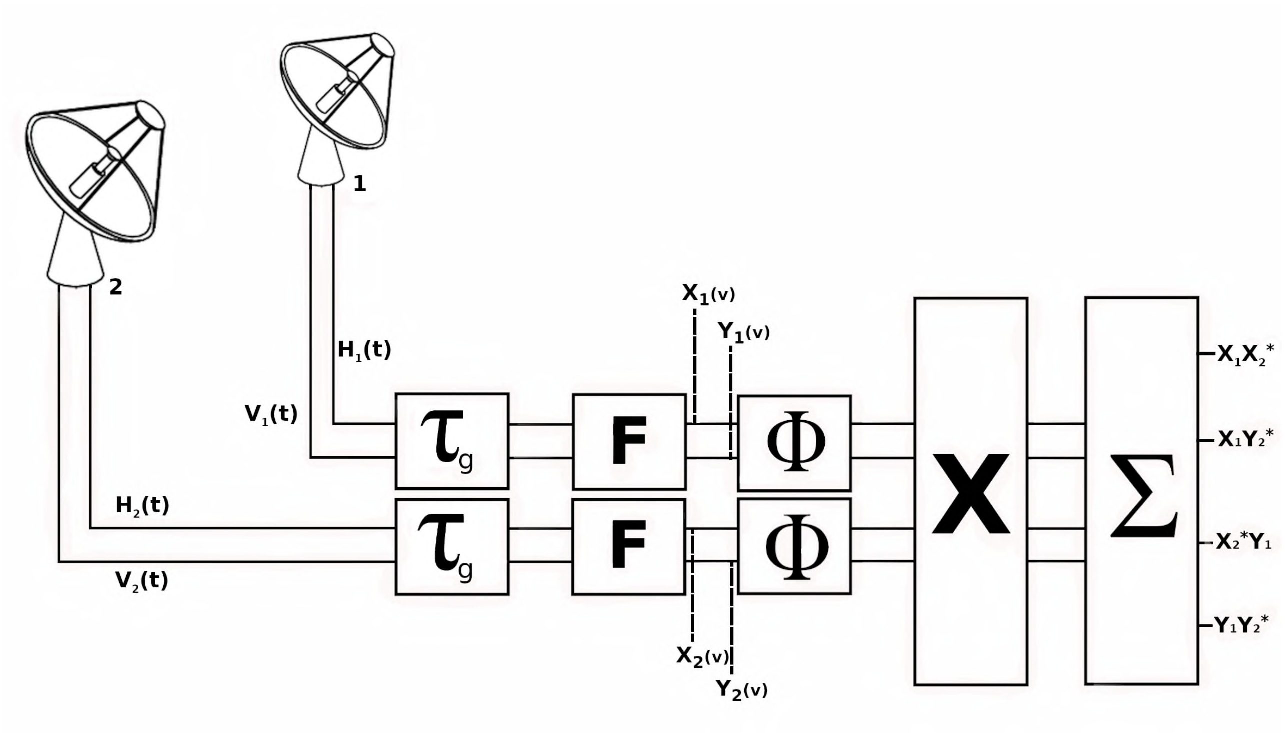

Figure 2.8: Digitised dual polarisation feeds $H(t)$ and $V(t)$ leave dishes 1 and 2. They are then coarse-delayed and channelised by the F-Engine to produce complex signals $X(\nu)$ and $Y(\nu)$. Following this, the signals are fine-delayed and then multiplied channel-wise and accumulated to produce the products necessary for the recovery of the stokes parameters in the frequency domain.

The input signals to the FFT are real, but the output spectral signals are complex-valued and Hermitian. By ignoring the redundant negative frequency channels, the required complex-valued analytic signals are obtained [4].

2.7.3 Digital Delay Tracking

In the previous section, the process of correlation is described analytically. While the conversion of equations 2.26 and 2.28 to electronic circuitry is shown in subsections 2.7.1 and 2.7.2, the digital correlator process must calculate a corresponding τ_0 to correct for the geometric time delay τ_{ij} (see section 2.5.1). This is done in two stages. First, coarse-delay tracking and then fine-delay tracking. The delay applied in each of these stages is done according to the following equation:

$$\tau_0 = N \times T_s + f \times T_s \tag{2.29}$$

where $0 \leq f < 1$.

The coarse-delay operation computes a value for N and exacts delay $N \times T_s$, while the fine-delay operation computes a value for $f \in [0, 1)$ and exacts delay $f \times T_s$. Recalling section 2.5, the required delay is a function of the antenna position and phase centre of the observation [4].

2.7.3.1 Coarse-delay tracking

Coarse-delay tracking involves buffering up readings at the reference antenna to correct for the geometric delay at the per-digital sample scale. Following equation 2.29, this means determining a value for N, where N is the number of digital samples to buffer. This is a time-domain based delay and hence is situated prior to the F-stage of the FX correlator as shown in figure 2.8.

2.7.3.2 Fine-delay tracking

Fine-delay tracking computes a value for f in equation 2.29, where f is the gradient of a linear phase slope. This delay correction is implemented in the frequency domain (post F-Engine see figure 2.8) by multiplying the complex spectral output of the F-Engine with a computed linear phase slope. In order to avoid wrap-around, it is imperative that the coarse-delay correction has reduced the delay to be contained within 1 sample i.e. fine-delay correction only corrects for the geometric delay at an intra-digital sample scale.

2.8 The X-Engine

After passing through the receiver, digitiser, coarse-delay correction, F-Engine and fine-delay correction, the X-Engine accepts two streams X and Y (representing the spectral result for each polarisation) from each antenna as show by figure 2.8.

Essentially, these four frequency domain complex signals are multiplied pair-wise and integrated in the X-Engine to produce the following matrices:

$$\begin{bmatrix} X_1X_1^* & X_1Y_1^* \\ Y_1Y_1^* & Y_1X_1^* \\ X_2X_2^* & X_2Y_2^* \\ Y_2Y_2^* & Y_2X_2^* \end{bmatrix} \text{ and } \begin{bmatrix} X_1X_2^* & X_1Y_2^* \\ Y_1Y_2^* & Y_1X_2^* \\ X_1^*X_2 & X_1^*Y_2 \\ Y_1^*Y_2 & Y_1^*X_2 \end{bmatrix} \tag{2.30}$$

The matrix on the left in equation 2.30 is the auto-correlation (see section 2.6), while the matrix on the right is the cross-correlation. The auto-correlation matrix is used for bandpass calibration [4]. Of the cross-correlation matrix, the first two rows can be shown to be the conjugate of the bottom two rows. As such, only the results from the first two rows need be outputted for the cross-correlation as shown by figure 2.8.

2.9 Polyphase Filterbank

Section 2.6 showed that for the 'FX' architecture, the 'F' stage performs a Fourier Transform, while the 'X' stage simply multiplies channel-wise and integrates. Channelisation by just an FFT is problematic because of the implied sinc function shape of the spectral channels. This causes spectral leakage between channels and scalloping loss [1]. Windowing of the time-domain data is needed to obtain a more rectangular spectral channel. In practice, this is mitigated by using a polyphase *Finite Impulse Response* (FIR) filter, followed by a DFT. This architecture (detailed by figure 2.9) forms a PFB [1], which is an efficient use of multi-rate DSP techniques. For this reason, many radio telescopes use a PFB for the 'F' stage to perform the Fourier Transform. The MeerKAT is one of them.

Given its complexity over the 'X' stage, the 'F' stage is more likely to produce mathematical finite precision error. This thesis aims to document/investigate the F-Engines' operation in detail.

The PFB is situated in the F-Engine after the digitiser and coarse-delay tracking in the signal chain (see figure 2.8). It receives samples of 2 time domain signals (2 polarisations) with coarse-delay corrections to account for different geometrical delays for each antenna. This digital signal is then passed to a commutator which distributes each value to a FIR register in the PFB FIR sequentially. The output of each FIR register is then passed into a point on the DFT (see figure 2.9). The following two subsections document the mathematical procedures performed within the PFB FIR and DFT.

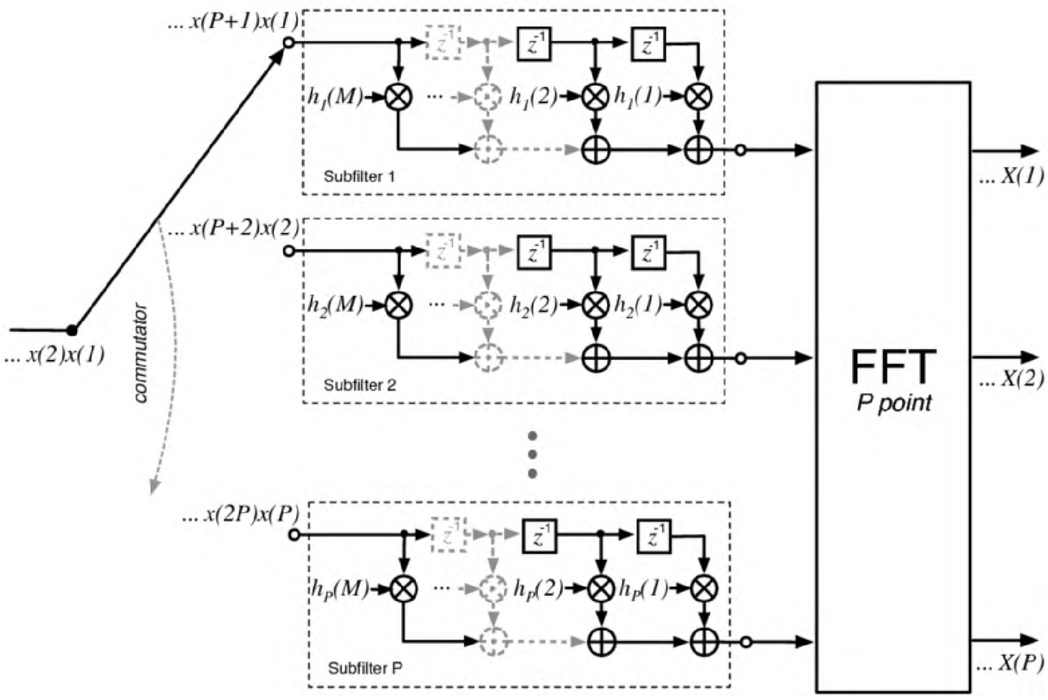

Figure 2.9: A PFB is formed by coupling a polyphase FIR filterbank with a DFT. Sourced from [1].

2.9.1 The FIR register operation

A FIR filter acts as a moving window average for an input $x(n)$ and computes the sum:

$$y(n) = \sum_{m=0}^{M-1} h(m)x(n-m) \tag{2.31}$$

where $h(m)$ is a set of M coefficients used for weighting. These coefficients are derived from the time-domain window function. We refer to the value M as the number of *taps*.

Figure 2.9 shows P streaming FIR filters encapsulated in dashed-line boxes, which act to decompose the input signal into P phases and down-sample the signal by $\downarrow P$. This technique (mathematically shown by equation 2.32) is known as polyphase decomposition [1].

$$y(n') = \sum_{p=0}^{P-1} \sum_{m=0}^{M-1} h_p(m)x_p(n'-m) \tag{2.32}$$

The data flow that occurs within the PFB FIR is illustrated by figure 2.10.

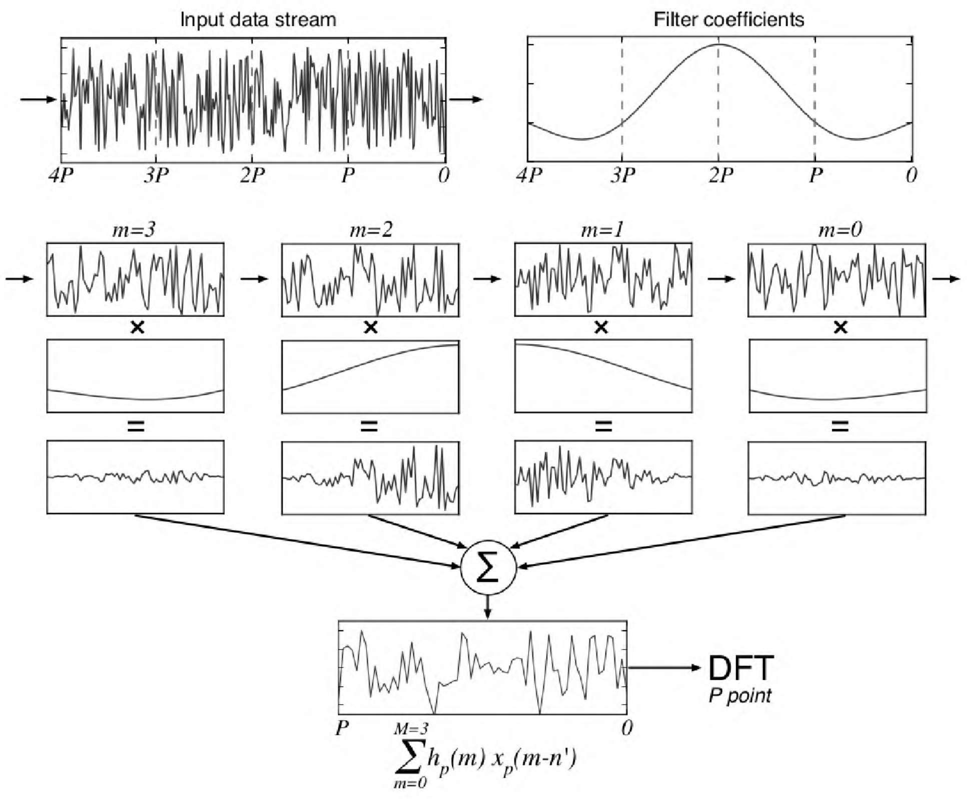

Figure 2.10: Graphical representation of the signal flow within a polyphase filterbank. Here, P = 64 and M = 4 polyphase taps. Data is read in segments of length P until M × P samples populate the FIR taps. The data and filter coefficients which are distributed across the M × P taps are multiplied together and summed over taps per FIR filter. After this, a P-point DFT is computed and another P input samples distributed in by the commutator.
Sourced from [1]

The commutator splits the input into P branches, feeding a different 'phase' of the signal to each of the polyphase sub-filter. The commutator in essence acts to apply a z^{-p} delay on each branch before a $\downarrow P$ downsampling.

The FIR filterbank buffers the values from the commutator until all taps (M) of each FIR filter (P FIRs) are filled i.e. buffering $M \times P$ values.

A full window (typically hann or hamming) of length $M \times P$ coefficients is pre-computed and stored in memory at build time (in the case of a *Field Programmable Gate Array* (FPGA) these values are stored in BRAM, see section 3.1).

The values contained within the FIR filterbank are multiplied by these window coefficients and summed along each FIR filter to present to a P-point DFT the signal $y(n')$ shown by equation 2.32 which is P values long [1].

2.9.2 The DFT operation

The DFT used in MeerKAT's F-Engine is a time decimated (DIT), Radix-2, natural-order-in, *Fast Fourier Transform* (FFT).

The basic DFT equation is given by equation 2.33,

$$X[k] = \sum_{n=o}^{N-1} x[n] W_N^k \, , \, 0 \leq k \leq N-1 \tag{2.33}$$

where N is the length of the signal being Fourier transformed and the coefficients $W_N^k = e^{-j2\pi nk/N}$ are known as the *twiddle factors*. This algorithm has an order of complexity $O(N^2)$, which has been reduced by a family of algorithms known as the FFT [12].

For even N, one can decimate the input time signal $x[n]$ such that $f[n] = x[2n]$ and $g[n] = x[2n+1]$, and rewrite equation 2.33 as

$$\begin{aligned} X[k] &= \sum_{n=0}^{(N/2)-1} f[n] W_{N/2}^{2nk} + \sum_{n=0}^{(N/2)-1} g[n] W_{N/2}^{(2n+1)k} \\ &= \sum_{n=0}^{(N/2)-1} f[n] (W_N^2)^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} g[n] (W_N^2)^{nk} \end{aligned} \tag{2.34}$$

Since $W_N^2 = W_{N/2}$, equation 2.34 may be rewritten as

$$\begin{aligned} X[k] &= \sum_{n=0}^{(N/2)-1} f[n] (W_{N/2})^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} g[n] (W_{N/2})^{nk} \\ &= F[k] + W_N^k G[k] \end{aligned} \tag{2.35}$$

where $F[k]$ and $G[k]$ are the Fourier transforms of $f[n]$ and $g[n]$ respectively [12].

Each of the sums in equation 2.35 is an $(N/2)$-point DFT as a result of the symmetric properties of the DFT [13]. Continuing to expand the DFT, one eventually reaches N/2 terms of 2-point DFT's which is the Radix-2 algorithm [12].

Figure 2.11 gives a graphical representation of an 8-point, Radix-2, natural-order-in, DIT FFT.

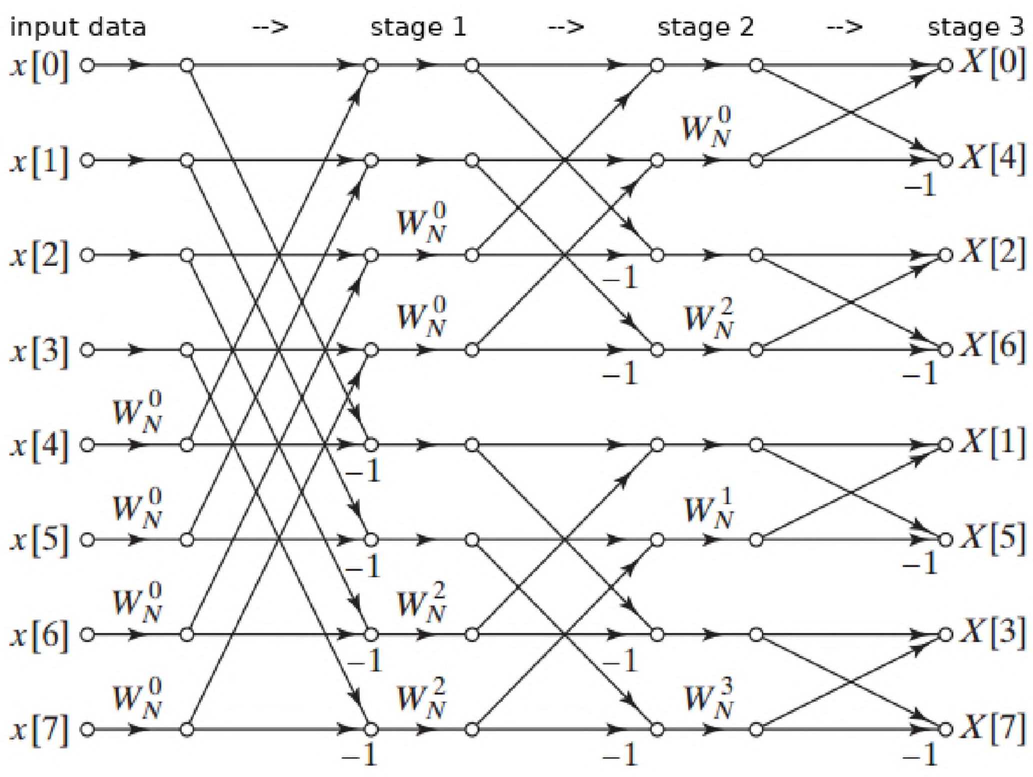

Figure 2.11: Decimation-in-time, 8-point FFT with natural order in and bit-reversed order out. Text at the top of the image indicates the ordering of the stages in the diagram. Post the third stage, data will undergo a bit-reversal transformation to return the output in natural order. Sourced from [13]

While other algorithms exist for the computing of the FFT, the Radix 2 decimation-in-time algorithm is used as it more suited to *buffered* architectures [12]. Buffering is the technique used in the MeerKAT FFT implementation, whereby the signals are buffered up and processed in parallel as opposed to *streaming* the signals through the FFT structure. The algorithm implemented in figure 2.11 is comprised of the three essential operations described below.

2.9.2.1 Bit-reversal:

There are two variants of the DIT FFT: natural-order-in and natural-order-out. Depending on which, one either bit-reverses the FFT input or output data. MeerKAT makes use of a post-bit-reversing (or a natural-order-in) structure as shown by figure 2.11. The ordering follows a ruling that the binary representation of the index of the sample is bit-reversed (i.e. $100 \rightarrow 001$, $110 \rightarrow 011$, $010 \rightarrow 010$ etc). The output in figure 2.11 is in bit-reversed order, and as such needs to be bit-reversed again in order to have the $X[n]$ values in natural order [12].

2.9.2.2 Butterfly:

Figure 2.12 shows the 2-point DFT structure (known as a *butterfly*) that the Radix-2 algorithm is based on. $f_m(j)$ is multiplied by the complex factor W_N^k before being added/subtracted to/from $f_m(i)$ [13].

Each Radix-2 butterfly performs one subtraction, one addition and one multiplication. Given that $f_m(i)$ and $f_m(j)$ are usually complex, this means that complex multipliers and adders must be used.

Complex addition/subtraction is largely simple in that one sums/subtracts the imaginary and real components respectively. For an N-point FFT, $N \log_2 N = 24$ complex additions are required.

Complex multiplication however involves 4 individual multiplications and 2 additions. In general, the Radix-2 architecture uses $N/2$ butterflies per stage for $\log_2(N)$ stages, requiring a total of $N/2 \log_2(N)$ butterflies. This bounds the maximum number of multiplications to $N/2 \log_2(N)$ bearing in mind that some of them involve twiddle factors of unity (which means passing the signal through or negating it) [14].

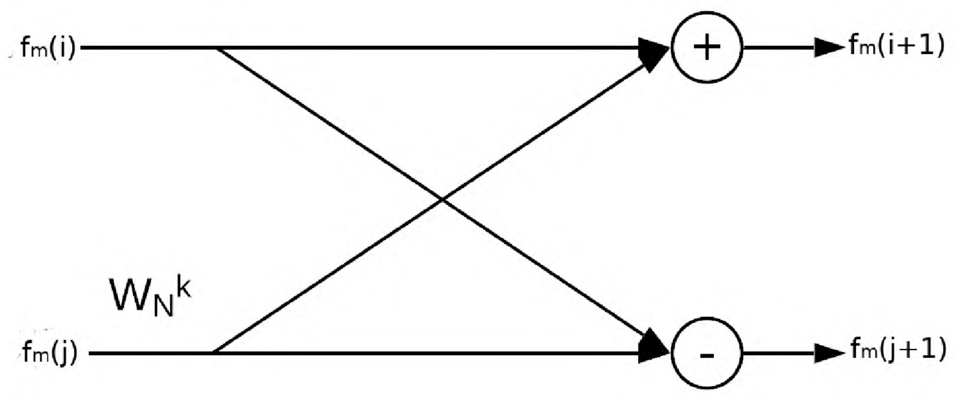

Figure 2.12: Signals $f_m(i)$ and $f_m(j)$ being passed into a two-point DFT (known as a butterfly). W_N^k is multiplied with signal $f_m(j)$ prior to subtraction.

2.9.2.3 Twiddle factor:

The twiddle factor by which one of the inputs to the butterfly is multiplied, is calculated using the value of N and k and depends on the stage of the algorithm as shown by figure 2.11 [12]. Since these twiddle factors are constant per FFT algorithm and size, they need not be calculated on the fly. If these twiddle factors are pre-computed at design time and stored in look-up tables, this offers a computational speed-up. For the natural-order-in DIT FFT, these twiddle factors need to be stored (or accessed) in bit-reversed order.

2.9.3 Dual Polarisation Processing

A single complex Fourier transform can simultaneously compute the spectrum for two real signals $g(t)$ and $h(t)$ by invoking the following Fourier transform properties [15]:

1. The Fourier transform of a real signal is guaranteed to be Hermitian.

2. A purely real signal has a complex spectrum composed of a symmetric real and asymmetric imaginary part.
3. A purely imaginary signal has a complex spectrum composed of a asymmetric real and symmetric imaginary part.
4. Fourier transforms obey the linearity principle i.e $\mathscr{F}[ag(t)+bh(t)] = a\mathscr{F}[g(t)]+b\mathscr{F}[h(t)]$ where $a, b \in \mathscr{R}$.

So, if we combine $g(t)$ and $h(t)$ so that they form the complex signal:

$$x(t) = g(t) + ih(t)$$

then their spectra ($X(k)$) will be the sum of their individual transforms. By invoking points 2 and 3, one may produce the *split operation* (equation 2.36), and use it to extract the spectra: $H(k)$ and $G(k)$, which correspond to each of the real signals h(t) and g(t) [16].

$$\begin{aligned} G(k) &= \frac{1}{2}[X(k) + X^*(N-k)] \\ & \qquad\qquad\qquad\qquad k = 0, 1, \ldots, N-1 \\ H(k) &= \frac{1}{2i}[X(k) - X^*(N-k)] \end{aligned} \tag{2.36}$$

Since both polarisations H and V have the same geometric delay τ_g, MeerKAT is able to use operation 2.36 to compute the spectra for both polarisations coming from a single antenna simultaneously.

2.10 Number Systems

The representation of real numbers in hardware has an affect on the accuracy, precision and speed with which mathematical operations are computed. The choice of which number system to use is often hardware limited and only of concern to high performance or high accuracy projects (nuclear research, deep learning, finance etc.).

Before proceeding, it is necessary to define a quantitative comparison technique for a number system. We define *decimal accuracy* as $-\log_{10}(|\log_{10}(x/y)|)$ where x and y are the correct and computed value respectively [17]. *Precision* is the number of bits given to representing a number (i.e. the more bits, the higher the precision). *Dynamic range* is the ratio between the largest to smallest value a number can assume given a certain number of bits. Measured in decibels, the dynamic range is calculated as $20\log_{10}(2^{n_{bits}})$ for fixed point binary representations, while numbers with scaling (which increases dynamic range) measure dynamic range differently [1].

Digitally, integer numbers are typically recorded as two's complement binary at the machine level. This enables the representation of signed/unsigned numbers and re-use of hardware for performing addition and subtraction [18]. Two's complement signed and unsigned numbers have a range of

$$\left[\frac{-2^{(bits-1)}}{2}, \frac{2^{(bits-1)}}{2}\right)$$

and

$$\left[0, 2^{(bits)}\right)$$

respectively, where the signed range is asymmetric as there is one more negative number than positive representable [18].

2.10.1 Fixed Point numbers

Fixed point numbers typically store real numbers in signed/unsigned binary, with the use of a 'binary point' [18]. This binary point acts as a divider between the integer and fractional parts.

A signed fixed point number is stored as a signed integer according to equation 2.37.

$$(-1)^{\mathrm{s}}(\text{integer bits})2^{(\text{fraction bits})} \tag{2.37}$$

A fixed point number is parametrised by its total number of bits and number of fraction bits. The notation used in this thesis will be '$\langle w, f\rangle$' where w is the full bit length, f the fraction length and $w - f$ is the integer length. By equation 2.37, the storing of a rational number is done by scaling the number and storing it as an integer (remembering its scaling). For example, storing 0.245771 in a $\langle 16, 10\rangle$ fixed point number with an infinite-rounding scheme means performing the following operation:

$$\text{fixednum}\langle 16, 10\rangle = (\text{int})\,(0.245771 \times 2^{10}) = (\text{int})\,251.669504 = 252$$

where '(int)' denotes the casting of a rational number to an integer while invoking the specified rounding scheme.

Turning fixednum back into a rational number means dividing it by 2^{10}. This effectively reverses the scaling and returns 0.24609375. Since fixednum only has 10 fractional bits available, the lack of precision resulted in a decimal accuracy of ≈ 3.244165. The better utilisation of bits to increase precision and dynamic range is the primary motivation for floating point numbers.

2.10.2 Floating Point numbers

Floating point numbers store a normalised signed or unsigned rational number. The format used for floating point numbers is:

$$(-1)^{\mathrm{s}}(1.\mathrm{f})\beta^{\mathrm{E}} \tag{2.38}$$

where 1.f is the normalised significand, β the base and E the exponent. In the early 1980's the IEEE produced a binary standard known as the 754 standard for floating point (see latest revision [19]). This means $\beta = 2$.

The 754 floating point standard (referred to as floats from here on) dictates how the full bit length should be split with regards to fraction and exponent bit length. The following example shows the conversion of a rational number to a 'half-precision' (16 bit) float. This standard sets aside one bit for the sign, five bits for the exponent and ten bits for the significand [19]. The following example shows how 0.245771 would be stored in floating point

$$\text{floatnum in binary}\langle 16\rangle = \text{0b0011001111011101}$$

where the leading 0 indicates the number is positive, the five following bits indicate an E = 12 and the last 10 bits indicate a significand = 989. Floating point numbers make use of a bias $(2^{k-1} - 1)$ where k is the bits allocated to the exponent. Biasing enables the exponents to be 'signed' so as to represent both small and large values while allowing for numbers to be compared lexicographically.

Adding a '1.' to the significand and dividing by the scaling (2^{E}) will give back floatnum = 0.2457. This yields a decimal accuracy of ≈ 3.901425, which is significantly higher than that achieved with the fixed point representation.

2.10.3 Posit numbers

Posit numbers are a direct drop-in replacement for the float and offer a larger dynamic range, higher accuracy, better closure, bitwise identical results across systems, simpler hardware and simpler exception handling [17]. The posit format is

$$(-1)^{\mathrm{s}}\mathrm{useed}^{k}2^{\mathrm{es}}1.\mathrm{f} \tag{2.39}$$

where $\mathrm{useed} = 2^{2^{\mathrm{es}}}$. What this translates to is a composite scaling factor of $\mathrm{SF} = (2^{\mathrm{es}})^{k} + \mathrm{es}$.

Qualitatively, posits have no 'NaN' (not-a-number) representations, no separate $\pm\infty$, no 'negative zero' and equality and ordering of posits is done in the same way as for binary integers. This translates to very simple circuitry for posit manipulation, unlike for floats [17].

Figure 2.13 shows the decimal accuracy of posits vs floats. As can be seen, close to 0, posits have a higher decimal accuracy, and tapered precision. Floats on the other hand have gradual underflow, and asymmetric precision. Since numbers in use are typically near 0, the tapered precision offered by posits is preferable [17].

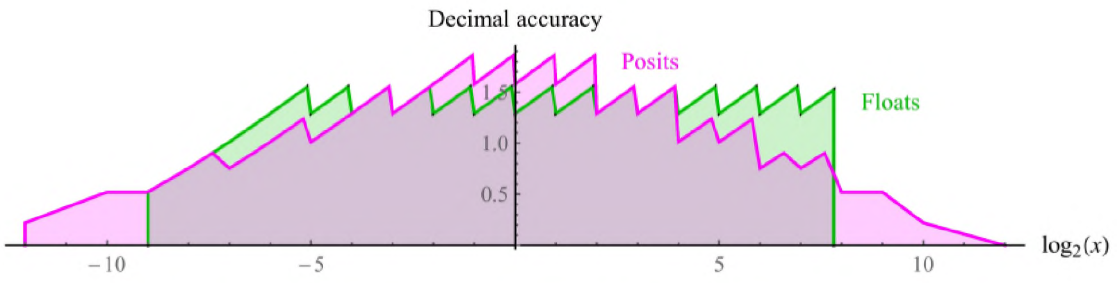

Figure 2.13: Decimal accuracy comparison of floats and posits. Sourced from [17]

If again we were to store the value 0.245771 in a 16-bit posit number we would have:

$$\text{positnum in binary}\langle 16\rangle = 0\text{b}0001111110111011$$

and the value returned would be 0.24578857. This yields a decimal accuracy of ≈ 4.508 which, in turn, is significantly higher than that achieved with floats.

Since posits are a relatively new creation (published in 2017), they lack the supported hardware and software to be used in any correlator structures as yet. Perhaps, for future SKA implementations these numbers will be considered.

2.11 Quantisation Schemes and Finite Precision Arithmetic

Finite precision number systems require a set conditions for how a real number is represented within its domain. This involves developing procedures for exception handling, boundary conditions, precision and arithmetic. While every number system has its own set of rules, this section will explicitly look at those used by fixed point binary since this is what is used in the MeerKAT correlator.

Within the F-Engine, the only arithmetic operations used are that of addition and multiplication (see section 2.6), both of which induce bit growth as shown by the below subsections 2.11.1 and 2.11.2.

2.11.1 Addition

Consider fixed point numbers $A = \langle 10, 6\rangle$ and $B = \langle 10, 6\rangle$ (recall notation from section 2.10). Computing $C = A + B$ will mean $C = \langle 11, 6\rangle$, since for every addition, one must account for a carry in the integer length. Note that one may only add together two fixed point numbers of the same fraction length (i.e. same scaling) [20]. As long as overflow does not occur, the addition of fixed point numbers will not cause inaccuracy in representing the summation. However, since overflow may occur, dynamic range constraints must be considered in system development [21].

While some methods exist for rectifying overflow after it has been detected, they are not reliable techniques because of the non-linearity of the processing following overflow [21]. The methods discussed in subsection 2.11.4 will look at the general practice in choosing scale factors to prevent overflow and maintain the largest possible signal-to-roundoff noise ratio.

2.11.2 Multiplication

Again, considering $A = \langle 10, 6\rangle \in \mathbb{R}$ and $B = \langle 10, 6\rangle \in \mathbb{R}$, computing $C = A \times B$ renders $C = \langle 20, 12\rangle \in \mathbb{R}$. Furthermore, for $(A, B) \in \mathbb{C}$, $C = A \times B$ means $C = \langle 21, 12\rangle \in \mathbb{C}$. This is as a result of complex multiplication invoking an inherent addition. Multiplication cannot cause overflow if both numbers are properly scaled. Where quantisation will occur however, is in re-quantising the result [21]. Subsection 2.11.3 will study the typical rounding operations applied when re-quantising the result and the side affects thereof.

2.11.3 Rounding

The finite representation of a real number will often require that the precision of the number is reduced to fit within a given length of digits (or in the case of computers, bits). When selecting a round-off scheme, one must consider how it affects the accuracy and the cost of implementation [18].

Consider x and $y \in \mathscr{R}$ and let $F()$ be the machine representation when rounding. The following machine representations must then be satisfied [18]:

1. $F(x) \leq F(y)$ whenever $x \leq y$.
2. If $x \in F$ then $F(x) = x$.
3. If F_1 and F_2 are consecutive numbers in F such that $F_1 \leq x \leq F_2$, then either $F(x) = F_1$ or $F(x) = F_2$.

The three rounding options most in use are truncation, bankers or even-rounding and infinite ($\pm\infty$) rounding.

Truncation (round-to-zero) is the simplest scheme in which the extra digits are removed with no change to the remaining digits. While this method is fast and does not require any additional hardware, its numerical performance is poor. The value 2.99 would be truncated to 2.

Infinite rounding (round-to-nearest ∞) is a more accurate scheme than truncation and adds 0.5 (if positive) or subtracts 0.5 (if negative) to the value before truncating. The maximum precision error is approximately half that of the truncation error, but the operation does require a full addition be performed. 2.5 would be rounded to 3, while -2.5 would round to -3. This scheme will introduce a bias since we consistently round away from zero.

Finally, Bankers rounding (round to nearest even) avoids the bias introduced by infinite rounding. For positive values, it adds 0.5 and truncates only if the significand is odd. Else it will subtract 0.5 and truncate. The reverse applies for negative numbers. 2.5 would then round to 2, while 3.5 would round to 4. In this way, half the time the value is rounded away from zero and half the time towards zero. This scheme requires a full addition and checking of the significands *least significant bit* (LSB) to see whether it is odd or even. Again, the maximum error is approximately half that of the truncation error and it reduces the infinite rounding bias.

While the 'round-to-nearest' schemes have a better numerical performance than truncation, their main disadvantage is that they require a complete add operation since the carry may propagate across the entire significand. Look-up tables holding the rounded results are a possible solution to this [18].

2.11.4 Overflow and Underflow

While in the previous subsection 2.11.3 we addressed the issue of storing values with a precision exceeding that representable by a number system, this section deals with the treatment of values that exceed the minimum and maximum values of the number type.

Underflow is simply when the number is too small to be represented by the number system and is nearly always just represented by the number 0.

Overflow is the opposite. For numbers too large to be represented by the number system, two options are usually employed.

The first is to wrap the value from +max to -max and visa versa for signed numbers and from max to 0 for unsigned numbers. Binary arithmetic does this wrap automatically i.e. for a 4-bit unsigned system, $0b1111 + 0b0001 = 0b10000 \rightarrow 0b0000$. Hence, wrapping requires no additional compute and is used where overflow is unexpected (since wrapping is unatural) and speed is of the essence.

The second is to saturate. For the same 4-bit unsigned system, $0b1111 + 0b0001 = 0b10000 \rightarrow 0b1111$. This is a more natural approach (as 4 is closer to 5 than 0 is), but requires additional compute to compare the result to the max and min of the system before setting the value to either the max or min.

A popular means to prevent overflow is to free up the *most significant bit* (MSB) of both operands before the arithmetic procedure. This is done by applying a right-shift operation. An example of this is when performing $0b0011 + 0b1111$, we would first right-shift each before summing: $0b0001 + 0b0111 = 0b1000$. Later when transferring to a number system of a higher dynamic range, the right-shift would be recalled and reversed with a left-shift to get the result 0b10000. This is not equal to 0b10010 but is closer than simply saturating the result to 0b1111 or wrapping it to 0b0001.

Recall that division by two corresponds to a single bit right-shift of a fixed point binary number and as such right-shifting by N bits corresponds to division by 2^N. The reverse applies for left-shift where one is multiplying by two.

2.11.5 Exception Handling

Fixed point number systems have no special bit patterns reserved for exceptions like NaN or Inf in floating point. For every N bit fixed point number, all possible bit patterns are reserved for a valid number [22]. However, since fixed point number systems are usually deterministic, mathematical anomalies are mostly avoidable.

CHAPTER 3

The MeerKAT F-Engine

The MeerKAT is a 64 dish interferometer with an F-Engine design built using the CASPER architecture and tool-flow.

This Chapter introduces CASPER, the DSP blocks: *pfb_fir_generic* and *fft_wideband_real* that are used in the building of the PFB, the CASPER supported hardware on which the design is implemented and finally the finite precision considerations made in building the F-Engine.

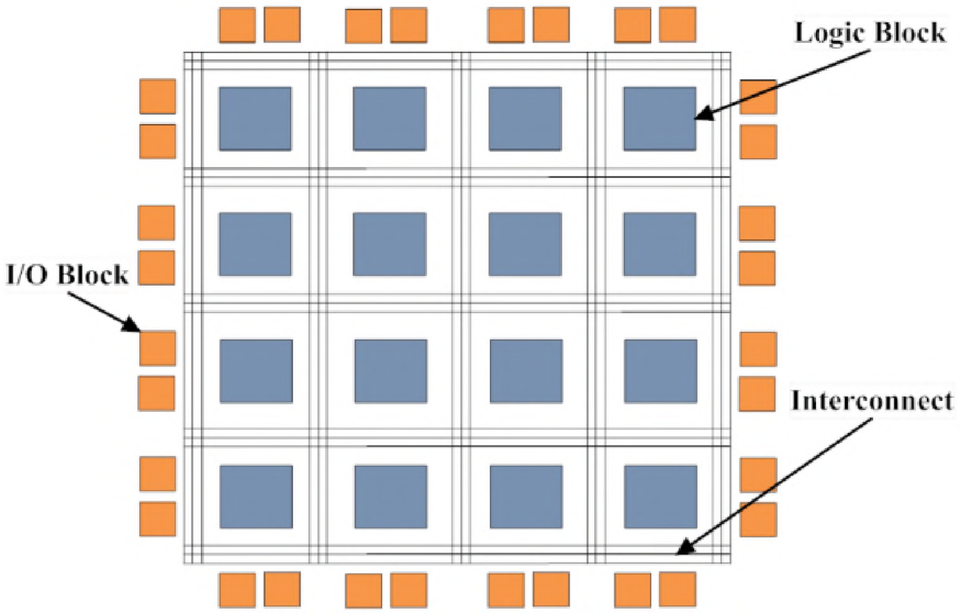

Figure 3.1: Generic fabric of an FPGA detailing the placement of Logic Blocks, Interconnects and I/O. Sourced [25]

3.1 Field Programmable Gate Arrays

High clock-rate, large *input/output* (I/O) applications require silicon solutions that are not always realisable on Graphical Processing Units or Central Processing Units [24]. Radio telescope correlators are one such application that often require many high speed interfaces and customisable high speed DSP. For this, FPGAs are perfectly suited.

The fine-grained reconfigurable fabric used in an FPGA is made of logic blocks, I/O blocks, interconnects and several integrated circuits (hard-cores) for DSP, encryption, arithmetic, memory etc. Logic blocks are cells comprised of *look-up tables* (LUT's) and serve to provide one-to-one mapping from inputs to outputs. Figure 3.1 details a generic fabric layout for an FPGA.

Defining the layout of the FPGA fabric is done largely through the use of *Hardware Description Languages* (HDL) like Verilog and VHDL. FPGA vendors often provide a tool-chain for mapping the design to the fabric as well as IP cores that contain pre-built designs for serial interfaces, FFTs, MAC functions etc. Xilinx, the manufacturer of the FPGA used in the MeerKAT correlator, provides the ISE and Vivado tool-chains. The tool-chain maps a design to the FPGA fabric by several steps: logical synthesis, translation, technology mapping and placement and routing. Following these steps, the tool-chain will return a bit-stream that may be loaded onto the FPGA for physical operation [25].

MATLAB provides the tool Simulink that lets a user draw out a design by connecting up pre-built 'blocks' that contain IP cores or HDL to speed up design. An example of this is shown in figure 3.2 where the CASPER PFB FIR has been connected up to the CASPER FFT to create a PFB. At build-time Simulink calls on system generator (a feature of the Xilinx toolflow) to begin the process of mapping the Simulink design to the FPGA fabric.

Some aspects of FPGA development are relatively simple, while other issues that might be dealt with at a software level like finite precision arithmetic, wordlength optimisation, on-chip routing delays (latencies) and memory interfacing are harder [24].

On-board memory (BRAM), I/O and hard-cores are limited. This means that for large builds like the correlator, special considerations for memory utilisation (like storing FFT twiddle factors in BRAM), I/O interfaces (like ethernet connections) and hard-core usage (like multipliers for the FFT butterflies) must be made else the design will not fit on the FPGA chip. Furthermore, the builds must usually meet timing constraints for reconfiguration and processing.

All these considerations were made when building the correlator designs for MeerKAT.

3.2 CASPER

CASPER is a collaboration that aims to minimise time-to-science with its open source hardware, software and tool-flow. The community builds and refines DSP tools that are commonly used throughout radio astronomy. Coupled with their supported open-source hardware platforms, CASPER hopes to enable the scientist to develop a complex back-end to their radio telescope [26].

Designs and protocols for FIR filters, FFTs, ADCs, Ethernet etc. are written in HDL or built up from Simulink blocks by FPGA engineers and packaged in the development suite *mlib_devel*, which is maintained by CASPER. Then, in conjunction with MATLAB & Simulink, a user is able to lay these designs and connect them in the manner required for their system. An example of this is shown in figure 3.2.

Having the design for their system, a user parses this Simulink file to Xilinx's Vivido or ISE system generator for the production of a .fpg file which contains the bitstream and metadata required for the programming of the FPGA on the target board.

Finally, by way of the *casper_fgpa* python package, a user may upload the .fpg file and program it to a CASPER supported hardware platform. Once programmed, the casper_fpga package wraps the firmware on the board and allows you to read/write to its software registers.

3.3 CASPER's Polyphase Filterbank

The primary CASPER DSP blocks that combine to form the PFB used in MeerKAT's 1k[1], 4k and 32k F-Engines are the *pfb_fir_generic* and *fft_wideband_real* blocks contained within the CASPER DSP Blockset library. Figure 3.2 shows a rudimentary PFB design that uses these blocks in Simulink.

The FIR block is customisable at design time to scale in size, tap length, windowing type, overflow handling, bit-widths, rounding scheme and bin widths. It furthermore offers options on how to build the design into FPGA fabric in terms of latency and coefficient storage.

The FFT block lets the user decide point size, number of simultaneous inputs, bit-widths of the input data, output data and twiddle factors, rounding schemes and overflow handling. It similarly offers the user some control on how the design is built into the FPGA fabric by giving control over BRAM usage, latencies and DSP core usage.

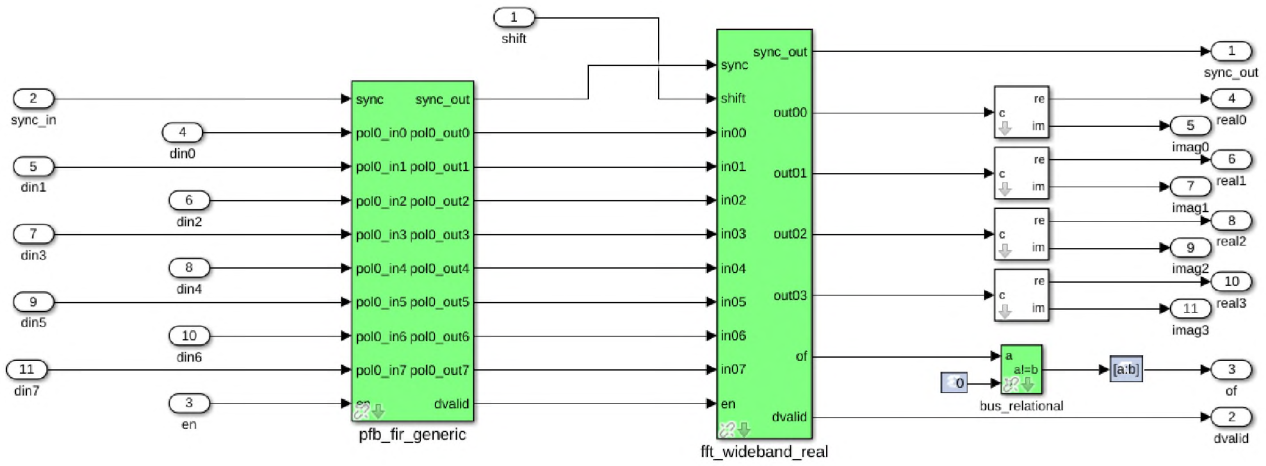

Figure 3.2: Simulink layout of *pfb_fir_generic* and *fft_wideband_real*. 8 real simultaneous inputs are accepted by the FIR and FFT blocks and 4 simultaneous complex outputs are supplied by the FFT block. The FFT block also accepts an integer from its shift port that will dictate the shift scheme of the FFT. These 4 complex outputs are split into 4 real and 4 imaginary signals and passed out.

[1]Note, the size mentioned dictates the spectral channel size. For example, a 1k FFT has 2^{11} inputs and 2^{10} output spectra (since half the output channels are discarded - see section 2.7).

3.4 SKARAB

In practice, correlators are implemented in hardware for speed and timing consistency. The typical approach is to either use dedicated circuitry, or, as with MeerKAT, to implement the correlator on an FPGA [11].

The FPGA hardware boards used in MeerKAT's F- and X-Engines are known as The *Square Kilometre Array Reconfigurable Application Board* (SKARAB). This board supports an FPGA, 4 high speed serial transceivers, Ethernet interfaces, Hybrid Memory Cubes and ADC mezzanine cards. For MeerKAT, the ADC mezzanines are not used because the digitiser is situated on the antenna. With no onboard Central Processing Unit (CPU), the SKARAB requires a Microblaze soft-core CPU be programmed to the FPGA fabric [27]. The onboard FPGA is a Xilinx Virtex 7 FPGA and it is to this chip that all F-Engine and X-Engine designs are programmed. A schematic layout of the board is seen in figure 3.3.

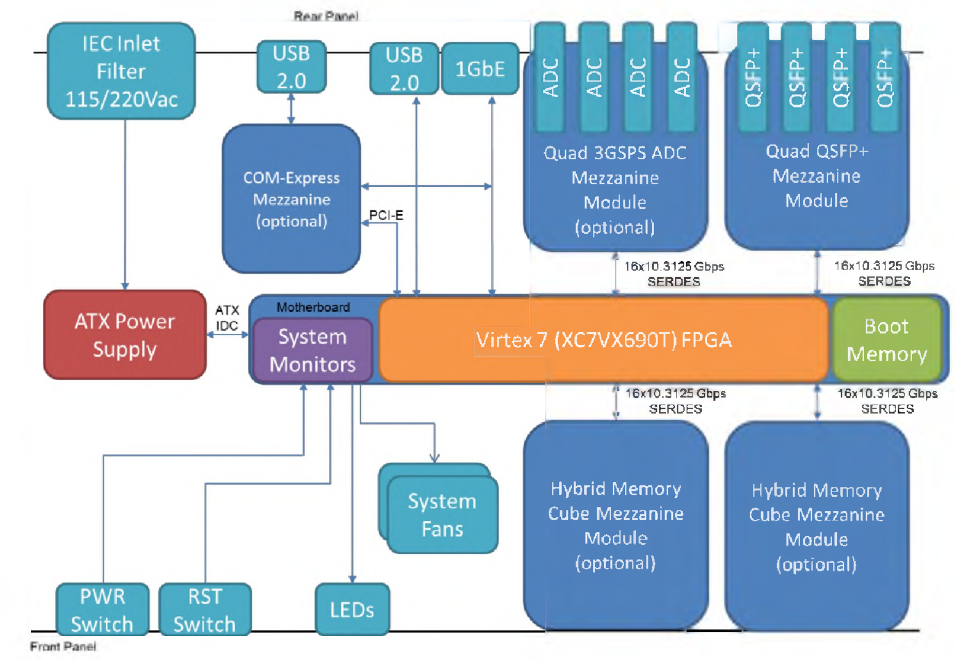

Figure 3.3: Onboard layout of SKARAB. Sourced [27]

3.5 Arithmetic procedures in the MeerKAT F-Engine

FPGAs use digital logic, constraining the data processed to be represented by either fixed or floating point numbers. As mentioned, both number systems have their own arithmetic procedures (see section 2.10).

MeerKAT's FX correlator uses signed, 2's compliment, fixed-point numbers, since these numbers allow for faster arithmetic and more deterministic results. As mentioned in section 2.11, the two primary arithmetic procedures used in its operation are addition and multiplication.

For the X-Engine, a channel-wise multiplication and accumulation is performed (see 2.6). The multiplication will induce bit-growth, but no overflow (see 2.11.2). Re-quantising the product will cause a loss in precision, but it is localised to the channel in which the multiplication was performed. The accumulation will also introduce bit-growth and in order to avoid overflow, the values are accumulated using larger bit-width fixed point numbers (typically 32-bit). If overflow does occur, the channel will saturate but not leak into neighbouring channels. The loss in re-quantising the product and effects of saturation in the accumulation are fairly well understood and mostly unavoidable without altering the number system or bit-widths. For this reason, the focus of this thesis is on the effects of finite precision arithmetic in the F-Engine and more specifically, the Polyphase Filterbank.

The MeerKAT F-Engine studied within this thesis was an $\langle 18, 17\rangle$ bit fixed-point CASPER design. Therefore, numbers represented in this system lie in the bounds $[-1, 1)$. Three F-Engine variations are in action, namely a 1k, 4k and 32k. $\langle 10, 9\rangle$ fixed-point data is fed into the F-Engine from the 10-bit ADC in the Digitizer (D-Engine). The ADC has an *effective number of bits* value of 7.6 and *root mean square* (RMS) of 17 counts. This corresponds to 4-bits being toggled 50% of the time on a cold sky (no RFI). Signals are set at this level as a trade between quantisation efficiency and headroom for RFI [5]. These 10 bits are parsed into the upper 10 bits of an $\langle 18, 17\rangle$ fixed-point number for processing by the PFB. This leaves the lower 8-bits free for growing precision during multiplication. The trade-off is that one needs to scale in the FFT immediately since no head-room is left for overflow. This 18-bit precision is maintained through the FIR and FFT. Below, we discuss the concerns and considerations in both the FIR and FFT processing.

3.5.1 Quantisation in the CASPER FIR Filterbank

As seen in figure 2.9, the FIR stage of the F-Engine performs a multiply-accumulate operation pointwise on a decimated time signal. In filtering, two forms of quantisation are of concern. First, is in coefficient representation and second is in finite arithmetic operations [21].

The windowing coefficients are generated at build-time according to equation 3.1

$$W = (H(P \times M) \times \mathrm{sinc}(\mathrm{f_{width}} \times ((P \times M)/(P) - M/2))) \tag{3.1}$$

where $H()$ is the windowing function (*Hann, Hamming, Bartlett etc...*), P is the size of the FFT, M is the tap size and $\mathrm{f_{width}}$ is the scaling of the bin width (where 1 is normal). These values are then stored in the FPGAs' BRAM and indexed as needed during its operation.

A plot of the windowing coefficients generated for a 4k, 8-tap PFB FIR using Hann windowing and $\mathrm{f_{width}} = 1$ is shown in figure 3.4.

Coefficients are usually stored as the same type as the input data. Since the coefficient values don't exceed the bounds $[-1, 1)$ (see section 3.5) and 17-bits represent the values with sufficient precision, there is no real concern of inaccurate representation.

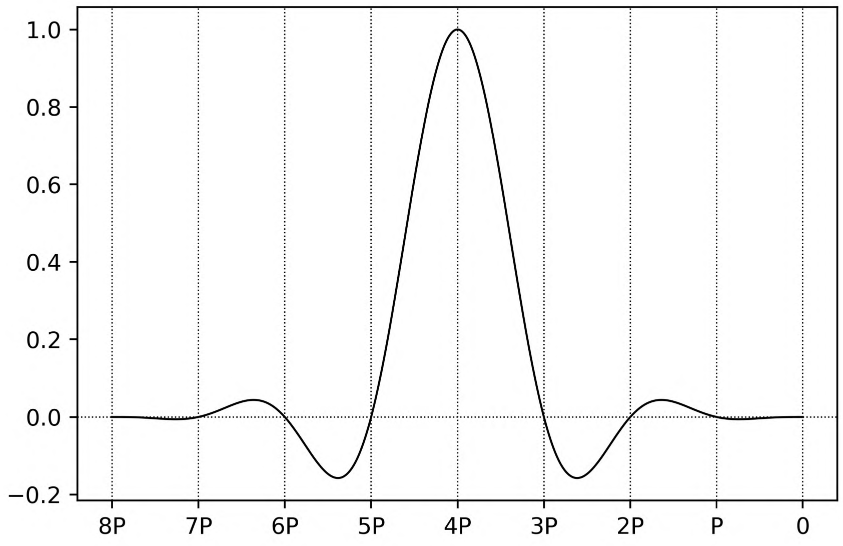

Figure 3.4: Plot of 8-tap, 4k F-Engine PFB FIR window coefficients using Hann windowing with $\mathrm{f_{width}} = 1$. P is FFT-length $= 2^{13}$. The boundaries at intervals of P indicate the coefficient values applied across taps $1 \rightarrow 8$.

These coefficients shown in figure 3.4 are multiplied with the real and imaginary parts of input data individually. No overflow would occur from the product and the extra precision is sliced away. While this may introduce some round-off noise, the primary concern is in the accumulation that occurs thereafter.

Typically, the worst case bit-growth when accumulating N numbers is $log_2(N)$. So, in this PFB FIR instance, where the tap length is 8, the largest bit-growth that could occur would be 3. If as an extreme, the input data were a series of -1's (the value of largest magnitude representable by the system), the data stored in the PFB FIR would be equal to the inverse of the window coefficient values (shown in figure 3.4). Recalling that $\mathrm{f_{width}} = 1$ normalises the coefficients, given this input, the maximum value the summation could be is 1. Furthermore, if one takes the absolute sum (to account for an input of values varying between ≈ 1 and -1), one would see the largest this value could be is < 2 (since the number system cannot represent $+1$). This indicates that for the 8-tap PFB FIR the maximum bit-growth that could occur is 1-bit as opposed to 3.

This prediction can be made at design-time and is calculated for each FIR as shown below:

$$\mathrm{fir}_{\mathrm{scale}} = \mathrm{nextpow2}(\max(\Sigma_{i=0}^{8}(|\mathrm{coeffs}_i|)))$$

where nextpow2() determines the next power of 2 greater than its input, max() returns the maximum value in a vector and the summation is over the FIR for its taps $0 \rightarrow 8$. For a full PFB FIR, this sum is performed for each FIR and from the absolute maximum of all those sums, a value for $\mathrm{fir}_{\mathrm{scale}}$ is chosen.

This scale value is then applied to the output of the 8-tap PFB FIR before restoring the data to an $\langle 18, 17 \rangle$ number and passing it on to a point on the FFT.

The decision to right shift down after the accumulation as opposed to before (as this would prevent overflow), is made by considering that saturation in any of the FIRs cannot leak into any of its neighbours (as with the FFT see subsection 3.5.2). Therefore, it is more beneficial to let the values in the FIR have access to full dynamic range when accumulating despite the possibility of overflow.

3.5.2 Quantisation in the FFT

Building an FFT in hardware means considering the effects of representing the data and twiddle factors with finite precision. This includes the round-off noise in compensating for the bit-growth caused by multiplication, scaling the data to prevent overflows caused by addition and inaccurate transformation due to incorrect finite representation of the twiddle factors W_N^k [21].

If $x(n)$ is an N-point sequence and $X(k)$ the discrete Fourier transform thereof, Parseval's theorem states that:

$$\sum_{n=0}^{N-1} x^2(n) = \frac{1}{N}\sum_{k=0}^{N-1} |X(k)|^2 \tag{3.2}$$

Equation 3.2 indicates that the mean square value of the transform $X(k)$ is N times the mean square value of $x(n)$. By equation 3.2, it can be shown that for a purely white noise signal, the channel amplitude growth is $\sqrt{N}$, i.e. $\frac{1}{2}\log_2(N)$ bits. For a mono-chromatic tone, the growth for the channel it is placed in is N, i.e. $\log_2(N)$ bits. This significant magnitude increase for mono-chromatic tone inputs, indicates the need to avoid overflow by introducing scaling procedures in fixed point arithmetic [21].

Bit growth can also be examined at the level of an individual butterfly. Consider figure 2.12, $f_m(i)$ and $f_m(j)$ are the inputs to the DIT butterfly with twiddle factor coefficient W_N^k. The outputs, $f_{m+1}(i)$ and $f_{m+1}(j)$, are then

$$\begin{aligned} f_{m+1}(i) &= f_m(i) + W_N^k \times f_m(j) \\ f_{m+1}(j) &= f_m(i) - W_N^k \times f_m(j) \end{aligned} \tag{3.3}$$

By equation 3.3, it can be shown that the maximum modulus of the complex numbers is non-decreasing from stage to stage such that:

$$\begin{aligned}\max\{|f_m(i)|, |f_m(j)|\} &\leq \max\{|f_{m+1}(i)|, |f_{m+1}(j)|\} \\ &\leq 2\ \max\{|f_m(i)|, |f_m(j)|\}\end{aligned} \tag{3.4}$$

Equation 3.4 shows that the magnitude of the signal level grows at a rate less than or equal to one bit per stage (i.e. a factor of 2) [21].

To avoid overflow caused by the growth detailed above, three scaling techniques are mentioned by [21]:

1. Shift right by one bit every stage. That way, if $|f_0(i)| < \frac{1}{2}$ for all i and the data is right shifted once after each stage (except the last) there will be no overflows.
2. Control the sequence such that $|f_m(i)| < \frac{1}{2}$ for all i. At each stage, $f_m(i)$ is computed and if its absolute value exceeds $\frac{1}{2}$, the entire array is right-shifted once.
3. Test for overflow. Scale the initial sequence such that $|Re\{f_0(i)\}| < 1$ and $|Im\{f_0(i)\}| < 1$ rather than one half like techniques 1 and 2. If an overflow occurs during a butterfly operation, the entire sequence at that stage is right-shifted by one bit beginning with the butterfly at which the overflow occurred. In this way more than one overflow may occur per stage, but no more than two.

Technique 1 is the fastest and simplest to program, but can be largely inaccurate due to the loss in precision for inputs where aggressive[2] scaling is not required. Technique 2 is time-consuming and still slightly (though less than technique one) inaccurate since the sequences are always scaled to be less than one-half (i.e. half the dynamic range is never used). The third technique is the most accurate, but is exhaustive since it requires the re-processing of the sequence each time an overflow is detected [21].

The CASPER complex FFT makes use of technique 1 and allows for the passing of an integer that, when translated to a binary sequence, indicates at each stage whether to right shift or not. For example, passing the integer 1935 to a 1k FFT which has 11 stages, translates to binary as 0b11110001111. This scheme dictates that a right-shift be applied on stages $1 \rightarrow 4$, no right-shift on stages $5 \rightarrow 7$ and more right-shifts on stages $8 \rightarrow 11$. This mapping of right-shifts to integer is non-obvious. A higher integer value does not necessarily indicate more right-shifting i.e. the integer 7 implies 3 right-shifts, while the integer 8 implies a single right-shift.

Important to note however, is that while the magnitude of the signal may not grow by more than 2 from stage to stage (as shown by equation 3.4), the individual imaginary and real components may. An example of this can be seen for $W_N^k = 0.707\ldots{-0.707}\mathrm{j}\ldots$ and $f_m(i) = f_m(j) = -1-1\mathrm{j}$,

[2]The use of the term *aggressive scaling/shifting* in this thesis will indicate a scheme whereby one right shifts the data at every stage of the FFT.

then:

$$f_{m+1}(i) = -2.414\ldots - 1\mathrm{j}$$
$$\text{and}$$
$$f_{m+1}(j) = 0.4142\ldots - 1\mathrm{j}$$

Measuring growth for $f_m(i)$ as $f_{m+1}(i)/f_m(i)$ and magnitude growth as $|f_{m+1}(i)|/|f_m(i)|$ shows that the output magnitude of $f_{m+1}(i) = 2.613\ldots$, which implies a magnitude growth of $1.848\ldots$ and therefore a magnitude bit-growth of $\log_2(1.848\ldots) = 0.886\ldots$ bits. This magnitude bit-growth is less than one, and as such a single right-shift is sufficient scaling. However, the real components' growth for $f_m(i)$ is $2.414\ldots$ which corresponds to a bit-growth of $\log_2(2.414\ldots) = 1.272\ldots$ bits. This growth is greater than one bit and as such a single right-shift is not sufficient scaling. This could indicate that scaling the output by two once per stage is not sufficient in avoiding overflow and that under certain circumstances, scaling by four (right-shifting twice) is required to prevent the individual real and imaginary components from overflowing. However, since the CASPER FFT does not allow for alternative scaling, this issue is not further addressed in this thesis.

CHAPTER 4

Implementing and testing the Python PFB Simulator

Studying MeerKAT's F-Engine and more specifically the CASPER DSP blocks used in its development, required an understanding of the CASPER tool-flow and the behaviour of its DSP. As such, the foundation of this project was learning how to create a design in Simulink, simulate its behaviour, synthesise it and deploy it to CASPER supported hardware. From there, research was done into how the tool-flow is implemented, what the differing hardware platforms offer and how they are used in radio astronomy.

However, in looking to test the finite precision effects present in the MeerKAT PFB, it became apparent that analysing the system from the Simulink framework was time consuming, difficult and required software licenses for both Simulink and Vivado. Furthermore, it was deemed important to study these effects through an independent framework. Therefore, in order to conduct this research more effectively, a full fixed and floating-point PFB simulator was developed in Python3. This provided the benefits of:

- Assisting/Attesting in/to the understanding of the system under study.
- Simplifying debugging since there is more control over the input vectors, operation of the simulator and no real requirement of a SKARAB board once the simulator is proven to be functioning correctly.
- Speeding up the prototyping of PFB designs and testing the effects of varying shift, rounding and overflow schemes for a variety of inputs.
- Being written in Python means that for many who read the code, the operation of the CASPER PFB will be easier to understand.

4.1 Floating point simulator

The floating point implementation was developed first, primarily, since NumPy (a popular mathematical package in Python) natively supports floating point.

A Radix-2, natural order in, DIT FFT that may be staged in its operation was developed. Staging is a feature of the floating and fixed-point simulators that will save a copy of the data at each stage of the FFT. This allows for a unique analysis (since this is not done in the CASPER FFT) of the signal flow through the FFT, which is especially useful when analysing where and why error was introduced in the result.

This FFT accepts a data vector that is a power of two in length and a bit-reversed array of twiddle factors (see subsection 2.9.2). Typically, FFT algorithms are implemented recursively in software, but for this research, an iterative design was used since it was a requirement to identically mimic the action of the CASPER FFT design.

Algorithm 1 from [28], details the iterative Radix-2 FFT structure (in pseudocode) used. Further functions to perform **bitrev** were developed as well as functions to generate the twiddle factors in vector w, but, given that they were more trivial by comparison, the reader is left to inspect the code in the appendix to see how they operate. The actual code does contain modifications to algorithm 1 to have it be more parallel in its operation and hence better utilise the strength of NumPy arrays.

Algorithm 1 In-place iterative Radix 2, natural order in, DIT FFT

```
Require: $N$ is of power 2
 1: Input: Real or Complex vector $data$ (length $N$) and vector $w$ containing $N/2$ bit-reversed
    twiddle factors
 2: Output: Complex vector (length $N$)
 3: $PairsInGroup := N/2$
 4: $NumOfGroups := 1$
 5: $Distance := N/2$
 6: while $NumOfGroups < N$ do
 7:    for $K = 0$ to $NumOfGroups - 1$ do
 8:       $Jfirst = 2 \times K \times PairsInGroup$
 9:       $JLast = Jfirst + PairsInGroup - 1$
10:       $Jtwiddle = K$
11:       $W = w[Jtwiddle]$
12:       for $J = Jfirst$ to $Jlast$ do
13:          $Temp = W \times data[J + Distance]$
14:          $data[J + Distance] = data[J] - Temp$
15:          $data[J] = data[J] + Temp$
16:       end for
17:    end for
18:    $PairsInGroup = PairsInGroup/2$
19:    $NumOfGroups = NumOfGroups \times 2$
20:    $Distance = Distance/2$
21: end while
22: $data =$bitrev$(data)$
23: return data
```

Extending from this, the full PFB object was built. This object depends on the FFT length, tap length, FIR windowing coefficients, whether to process a complex or real vector (for dual polarisation processing, see subsection 2.9.3), whether to stage the FFT, bin-width scaling $\mathrm{f}_{\mathrm{width}}$ (see subsection 3.5.1) and whether to integrate/accumulate the outputs. Figure 4.1 shows a high-level design flow of the PFB object.

Inputs accepted to the PFB are a real or complex vector, while outputs are either a single complex vector (given a real input), or two complex vectors that are the spectrum of the real and imaginary parts, given a complex input. If the PFB is told not to accumulate the outputs, the output will be a two-dimensional matrix of N by the number of spectra.

Should staging have been chosen, a two-dimensional matrix of $log_2(N)+2$ by N will be supplied that contains the input data, the data at each stage of the FFT and the bit-reversed output data. Again, should the PFB be told not to accumulate, the output will be a three-dimensional matrix of N by the number of spectra by the number of stages.

4.2 Fixed point Simulator

Extending from the floating point implementation, this software object looks to mimic the quantisation effects experienced in the CASPER PFB by using the same number system: fixed point numbers. This meant studying how the MeerKAT implementation handled bit-growth and copying this for the simulator. So while it has the same parameters and input and output formats as the floating point implementation, it has additional parameters for handling the processing of fixed point numbers.

Since NumPy did not support fixed point numbers, a fixed point number system was built. It is declared for a specific bit-length, fractional bit-length and rounding scheme and allows for the user to pass it real numbers (double precision floats) for conversion to fixed point numbers following the procedure outlined in subsection 2.10.1. The result of this procedure is stored as 64-bit integer value and constrained by the fractional bit-width (that limits precision) and integer bit-width (that limits dynamic range). Inbuilt into the object are functions governing how arithmetic is performed, mimicking a fixed point like way (see section2.11). The numbers were also designed to allow for several rounding schemes and made to saturate rather than wrap when overflowing[1].

Extending from the fixed point number class is a complex fixed point number class. This class declares a real and imaginary fixed point number and similarly accepts a double precision complex number for conversion. Arithmetic is altered to do complex arithmetic whilst using the underlying fixed point arithmetic, rounding and overflow behaviour.

These number systems were built with a NumPy back-end for ease of use and to try have the floating and fixed point simulators be comparable. This meant that the prominent differences in operation would be owing to precision and overflow handling (where these effects in the floating-point simulator are negligible in this thesis).

Obeying the bit growth handling from section 3.5, the FIR and FFTs were implemented using fixed point numbers. Alterations to the FFT algorithm 1 are shown below in algorithm 2, where now, steps are included to control bit-growth and shift.

Lines 15 and 16 of algorithm 2 control the fractional bit-growth incurred through multiplication. $W.fracbits$ is the number of bits allocated to the fractional part of the twiddle factors, while the $normalise()$ function clips the data to lie within its dynamic range (which is bit-width dependant). The right-shift applied in line 15 slices away this fractional bit-growth. This correction will produce quantisation noise that is dependant on the rounding scheme used by the numbers as well as their precision. Lines 21 - 24 treat the effects of overflow in the FFT by right-shifting if for stage i, $swreg$ is a 1. While a fraction of the quantisation noise introduced here will be owed to the rounding scheme used, should this if-statement be by-passed and overflow occurs in the following iteration, the quantisation noise caused by saturation and channel leakage will be dominant.

[1]since wrapping is only used when not anticipating overflow - and we look to test the effects of overflow: see subsection 2.11.4.

Algorithm 2 In-place iterative Radix 2, natural order in, DIT fixed-point, FFT

```
Require: $N$ is of power 2
 1: Input: Real or Complex vector $data$ (length $N$), vector $w$ containing $N/2$ bit-reversed
    twiddle factors and shiftregister swreg.
 2: Output: Complex vector (length $N$)
 3: $PairsInGroup := N/2$
 4: $NumOfGroups := 1$
 5: $Distance := N/2$
 6: $i = 0$
 7: while $NumOfGroups < N$ do
 8:   for $K = 0$ to $NumOfGroups - 1$ do
 9:     $Jfirst = 2 \times K \times PairsInGroup$
10:     $JLast = Jfirst + PairsInGroup - 1$
11:     $Jtwiddle = K$
12:     $W = w[Jtwiddle]$
13:     for $J = Jfirst$ to $Jlast$ do
14:       $Temp = W \times data[J + Distance]$
15:       $Temp >> W.fracbits$
16:       normalise($Temp$)
17:       $data[J + Distance] = data[J] - Temp$
18:       $data[J] = data[J] + Temp$
19:     end for
20:   end for
21:   if $swreg[i] == 1$ then
22:     $data >> 1$
23:   end if
24:   normalise($data$)
25:   $PairsInGroup = PairsInGroup/2$
26:   $NumOfGroups = NumOfGroups \times 2$
27:   $Distance = Distance/2$
28:   $i = i + 1$
29: end while
30: $data =$ bitrev($data$)
31: return data
```

Prior to the fixed point FFT, quantisation control is done in the FIR to remove the bit-growth incurred by the summation and multiplication as described in subsection 3.5.1.

4.3 The CASPER PFB design

The above simulators are only valuable insofar as they mimic the behaviour of the CASPER PFB. The operation of these simulators were tested against a 4k CASPER PFB (as used within the MeerKAT F-Engine) shown in figure 4.2. While originally these designs were implemented on SKARAB hardware and data was passed to and from it using software registers in the FPGA fabric, this was prone to error in data conversions and collecting data correctly.

Supplying input to the PFB design shown in figure 4.2 is done by connecting the input signal to the cwg80 bus. cwg80 is an eight-value-wide bus accepting eight 18-bit values simultaneously from the input signal. Collection of the output is done by recording the eight 18-bit values that are simultaneously present on the pfb0 to pfb7 lines. The process of testing, was then to insert various Simulink generated signals into cwg80, simulate the full design for some time, save the input signals to file and save the corresponding outputs to file. When comparing with the software simulators, this made it simpler to ensure that the inputs and outputs to both PFB designs were the same.

The same was done for just the 4k FFT alone (see figure 4.3), since the finite precision effects that this research aims to investigate occur in the FFT process. Therefore, it was more important that the simulator FFTs mimicked the CASPER FFT than the simulator PFBs mimicking the CASPER PFB. As such, testing began with the FFT and would only progress to the PFB once the software simulated FFTs comfortably mimicked the CASPER FFT.

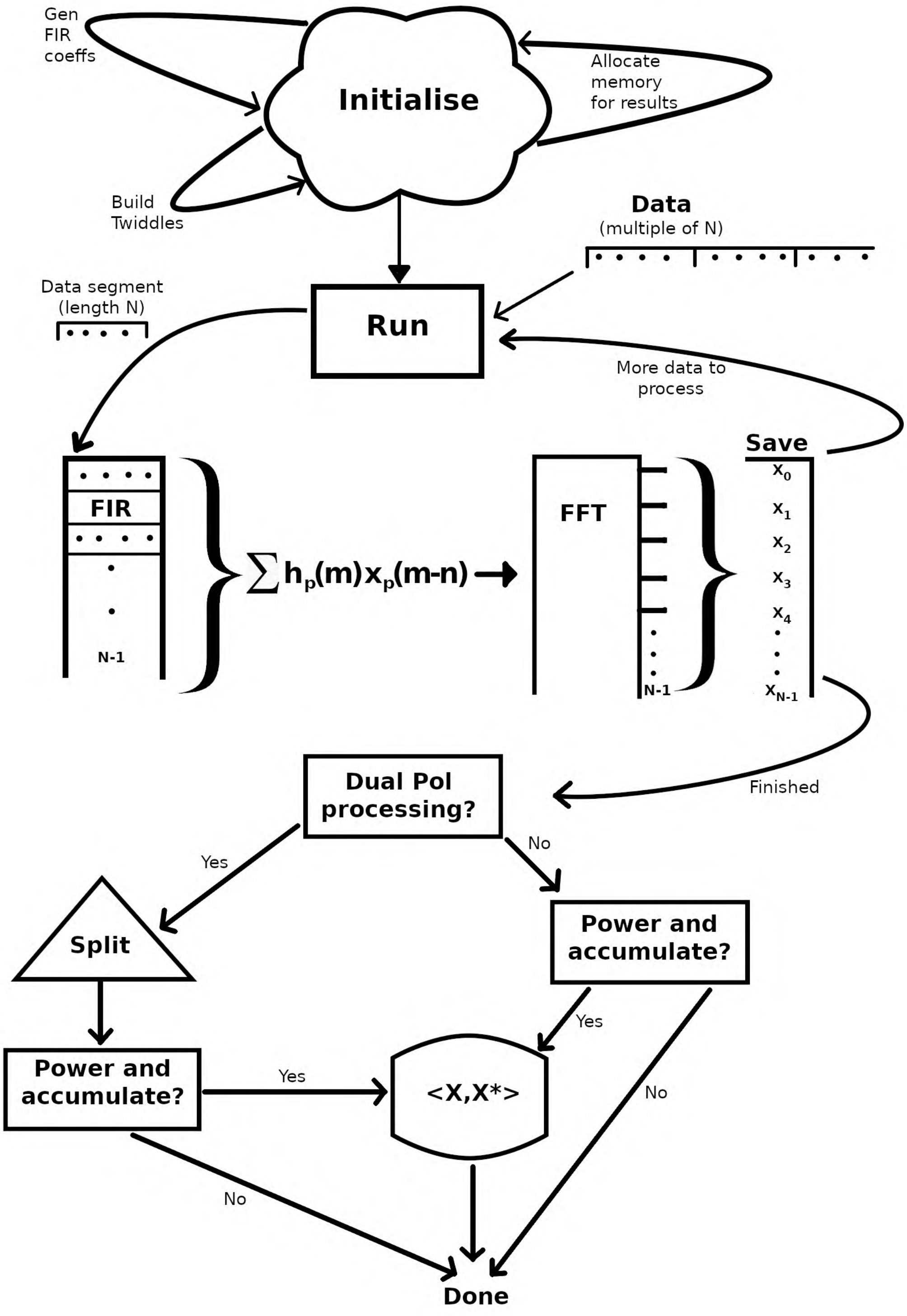

Figure 4.1: Flow chart detailing the operation of the PFB simulator. On initialisation, parameters regarding the size of FFT, taps etc. are passed to the PFB and used in the generation of twiddle factors, fir windowing coefficients and variable space for outputs. Given data and instructed to 'run', the PFB object will segment the data into portions of length N. Each segment is then loaded into the first taps of the PFB FIR, which will then return the FIR output $\sum h_p(m)x_p(m-n)$. This output is passed on to the N-length Radix-2 FFT. The output spectra are then stored, and if there are more data segments to process, the cycle repeats. Else, the complex FFT output data is split if dual polarisations were processed and summed and accumulated if specified.

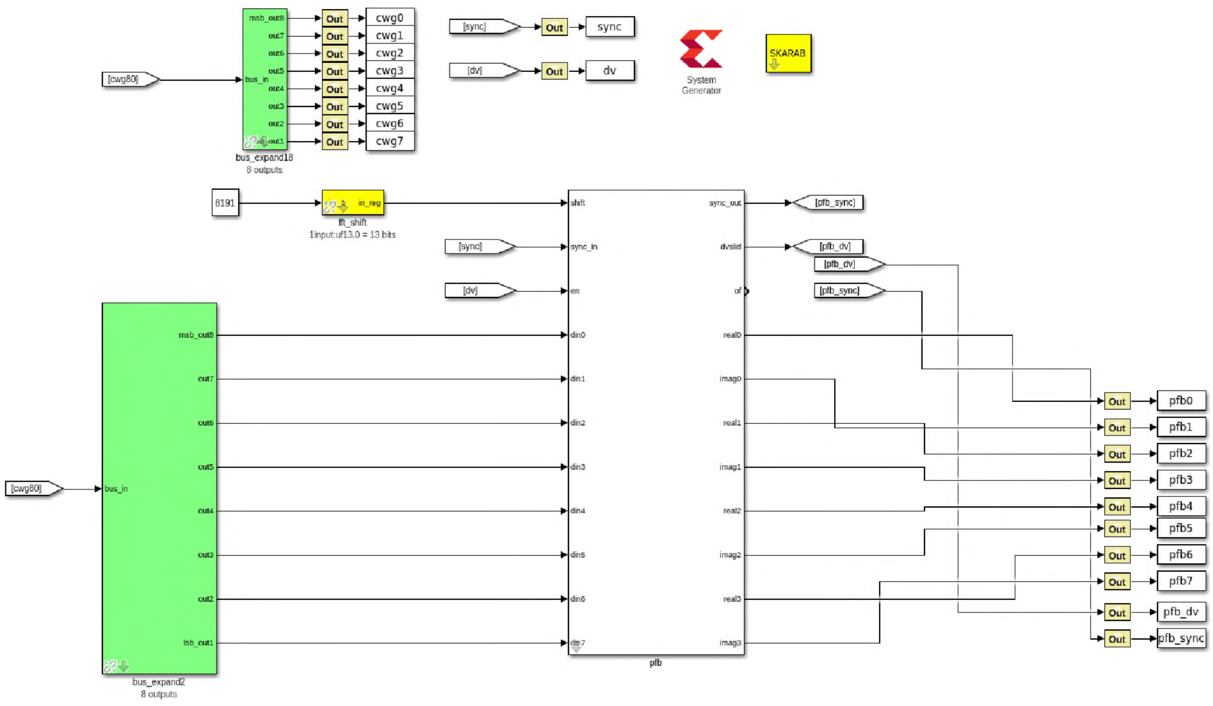

Figure 4.2: The Simulink design of the 4k MeerKAT PFB used for testing. The cwg80 terminals connect to the input source which outputs 8 samples of the signal simultaneously. The upper terminal passes these samples to workspace variables cwg0 - cwg8, while the bottom terminal expands the signals and passes them to the CASPER PFB block. This block accepts a shift register, a data valid and sync signal for timing. The PFB outputs 8 values simultaneously which are saved to workspace variables pfb0 - pfb8. After simulation, a MATLAB script is run to extract the data from all the workspace variables and re-order them into a single input vector and real and imaginary output vector. The system generation blocks are there for build-time to build for a SKARAB.

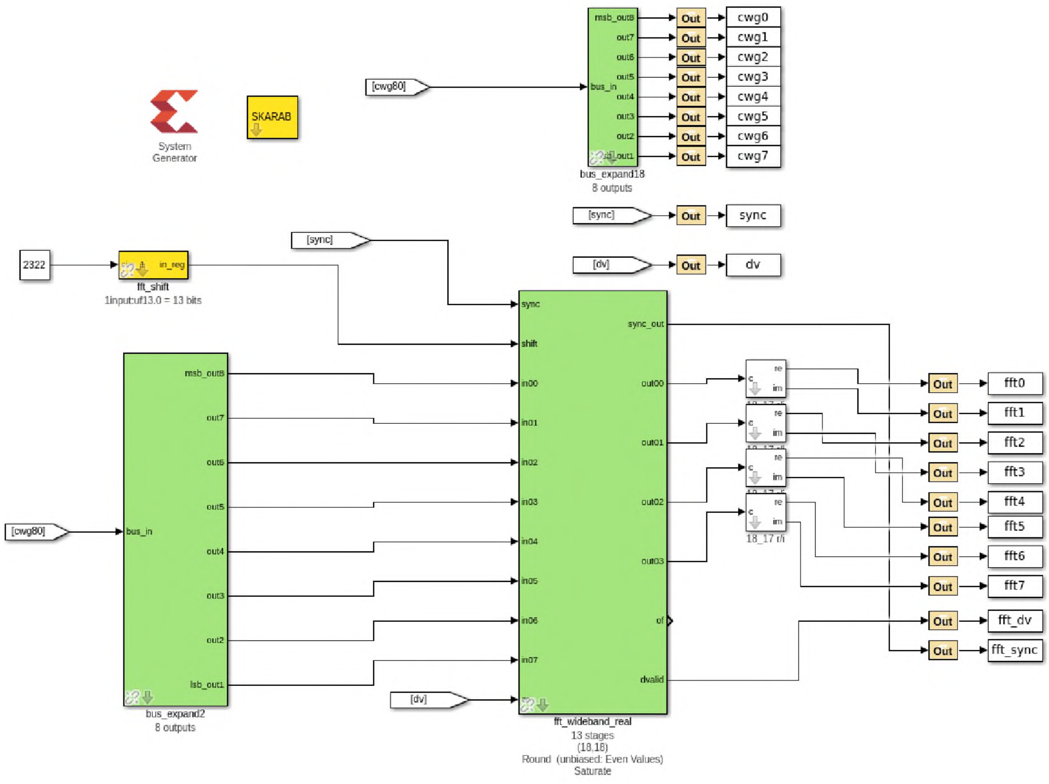

Figure 4.3: The simulink design of the 4k MeerKAT FFT used for testing. The cwg80 terminals connect to the input source which outputs 8 samples of the signal simultaneously. The upper terminal passes these samples to workspace variables cwg0 - cwg8, while the bottom terminal expands the signals and passes them to the CASPER FFT block. This block accepts a shift register, a data valid and sync signal for timing. The FFT outputs 8 values simultaneously which are saved to workspace variables fft0 - fft8. After simulation, a MATLAB script is run to extract the data from all the workspace variables and re-order them into a single input vector and real and imaginary output vector. The system generation blocks are there for build-time to build for a SKARAB.

CHAPTER 5

Results and Discussion

This chapter reports the results of running the various simulations and tests for a variety of scenarios in order to test the simulators and to investigate the systematic effects observed in the MeerKAT data.

In this thesis, the SNR is measured as:

$$SNR = 20 \times \log_{10}\left(\frac{(1/\sqrt{2}) \times (\max(\text{tone})^2)}{(\text{mean}((\text{noise})^2))}\right) \tag{5.1}$$

where max() selects the maximum value of the input vector and mean() calculates the mean of an input vector.

For fixed precision integer signals, such as the digital representation of voltages provided by an ADC, it is convenient to introduce a decibel measure that characterizes the peak amplitude of the signal relative to the maximum amplitude allowed by the number of bits used to represent the number (e.g. the resolution of an ADC). This measure is *decibels relative to full-scale*, abbreviated by dBFS and defined by:

$$dBFS = 20 \times \log_{10}\left(\frac{\max(|\text{data}|)}{(2^{\text{bits}-1})}\right) \tag{5.2}$$

where “data” is the digital signal and $2^{nbits-1}$ is the maximum allowed amplitude span.

5.1 CASPER PFB versus Python Simulator PFB

The primary test is to show that the developed fixed point Python PFB simulator correctly mimics the behaviour of the CASPER PFB. To do this, we test a 4k, 8-tap, Hann filtered, even-rounding PFB design in Simulink (see section 4.3) against a 4k, 8-tap, Hann filtered, even-rounding PFB design in the fixed and floating point Python simulator. This is done in two parts: first, test the FFTs alone and then the full PFB.

The fixed point PFBs and FFTs (CASPER and Python simulator) receive an $\langle 18, 17\rangle$ fixed point number input. They then both process at that same precision and output an $\langle 18, 17\rangle$ complex fixed point number (the CASPER outputs the real and imaginary separately - see section 4.3). The input signal levels for the *white* gaussian noise is chosen so that the lower 4-bits of the input data is occupied more than 50% of the time (see section 3.5). Input tone levels are set so that the amplitude takes 16-bits of the available 18, signifying strong RFI but not so strong that overflow is caused in the first butterfly.

First, a pure tone is loaded into the 4k FFT Simulink design and simulated. The output is collected and saved to disc. The same pure tone is then processed in a 4k Python fixed and floating point FFT simulator and its results are saved to disc. Both CASPER and the simulator fixed point FFT use aggressive shifting and even rounding. The absolute values of the results of the CASPER, and simulator fixed and floating point FFTs are compared in figure 5.1, while the real and imaginary components of the CASPER and simulator fixed point FFT are inspected separately in figure 5.6.

The same pure tone input is then again processed through the CASPER 4k FFT and fixed point simulator 4k FFT, but this time, a non-aggressive scheme (0b101010101010) is used. A comparison between their results is shown in figure 5.3, while the data from each stage of the fixed point FFT simulator for this same run are shown in figure 5.4.

Next, the same pure tone but now with added gaussian noise is used as an input to all the 4k FFTs (simulated float, fixed and CASPER FFT). The input SNR is $\approx$ 19dB and the outputs are recorded. The absolute output of each FFT is shown in figure 5.5.

The final FFT test checks the 4k FFTs response to an impulse input, where the simulator fixed point FFT and the CASPER FFT use a shift-scheme of 0 (no shifting). The real and imaginary components of each FFT are compared in figure 5.9.

Having tested the 4k FFTs for varying input and shift-register configurations, the 4k PFBs are then compared. To begin, the same pure tone from the first FFT test is loaded into each 4k PFB. Each of the 4k PFBs use Hann windowing and 8 taps. Both fixed point PFBs (the CASPER and Python simulated one) use even rounding and aggressive shifting (integer value of 8191). Their absolute results are shown in figure 5.7.

Gaussian noise is then added to the same pure tone so that the signal has an $SNR \approx$ 19dB, and the same PFBs (with their configurations maintained from the previous test) are tested. These absolute results are shown in figure 5.8.

Finally, an impulse signal is input to each PFB where now the fixed point PFBs (CASPER and the simulated one) use a shift-scheme of 0. All other configuration parameters are unchanged from the previous test. These even and imaginary results are compared in figure 5.9.

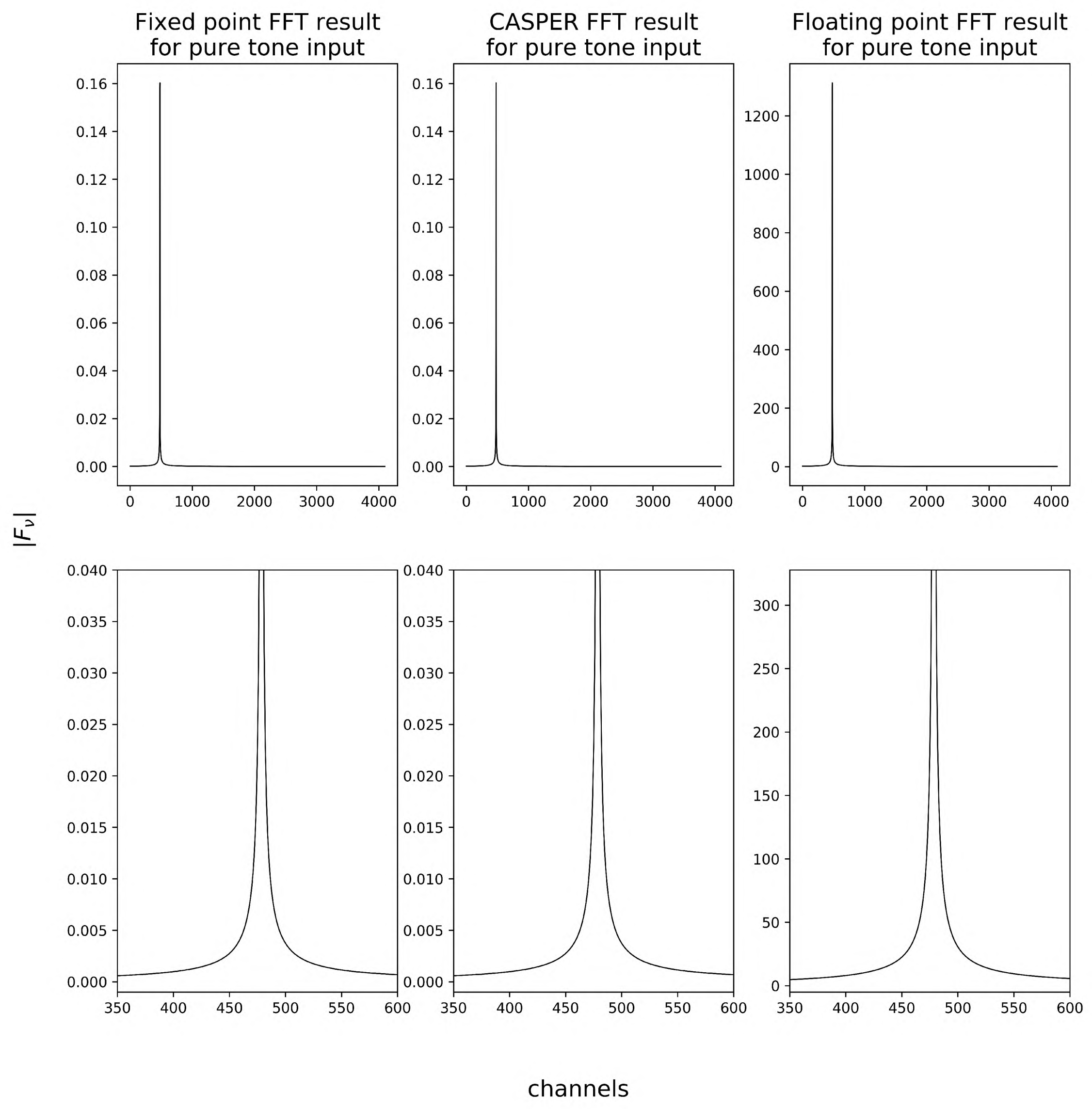

Figure 5.1: Top row is the magnitude of the FFT output for a pure tone input for the three different FFT implementations. The input tone had an amplitude of $\frac{1}{2}$. The delta spike sits in channels 478 and 479 for all FFTs. CASPER and the Simulated fixed point FFTs are 2^{13} times smaller in magnitude than the Simulated floating point FFT owing to the scaling of 8191 (0b1111111111111) applied in the CASPER and fixed point FFTs. The bottom row is a magnification of both the vertical and horizontal scale in order to note details in the spike produced.

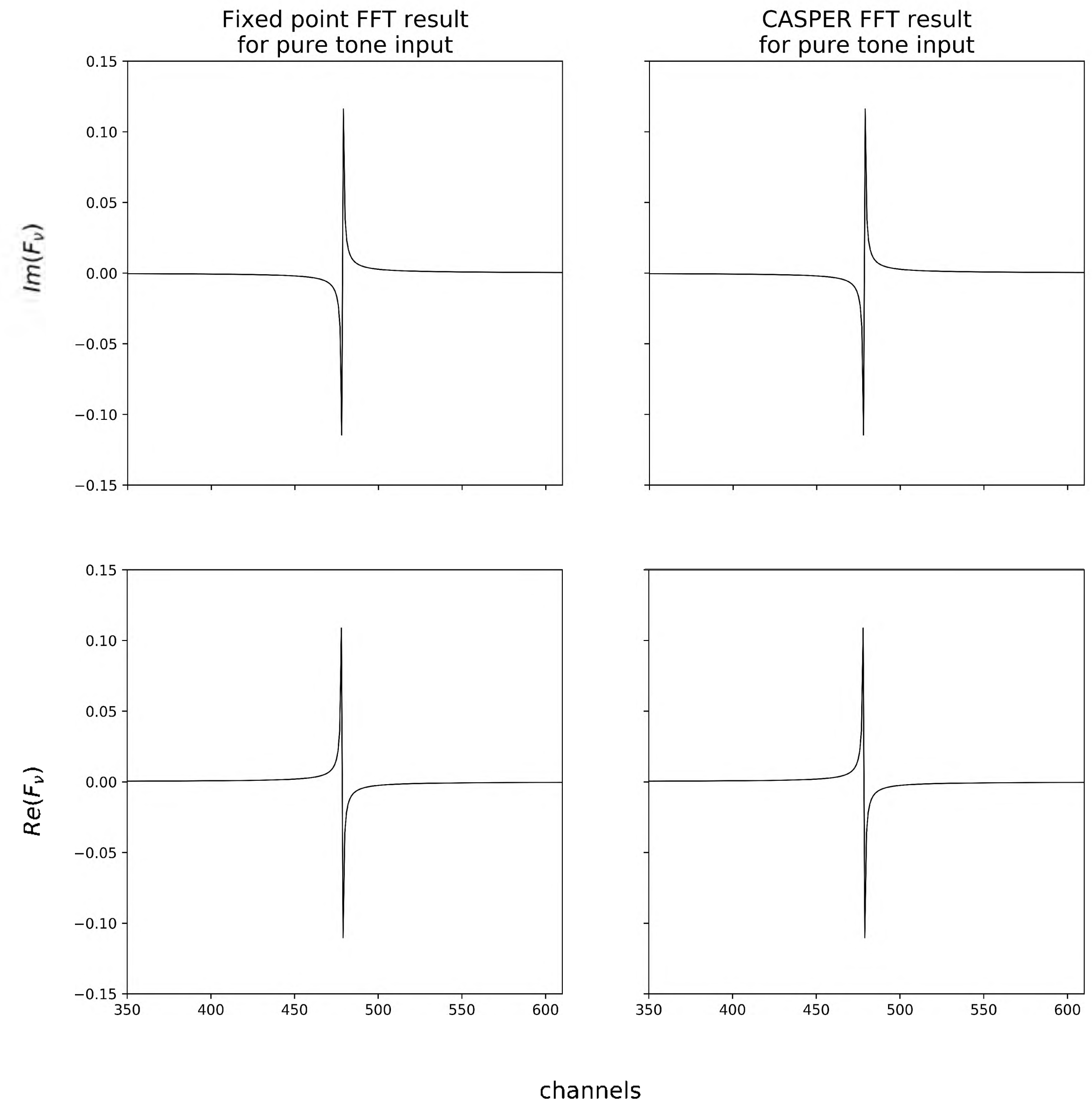

Figure 5.2: Real and imaginary components of the two fixed point FFT outputs for a pure tone input. The tone had an amplitude of $\frac{1}{2}$. Both FFTs used a scaling of 8191 (0b1111111111111). The top row displays the imaginary components, while the bottom displays the real.

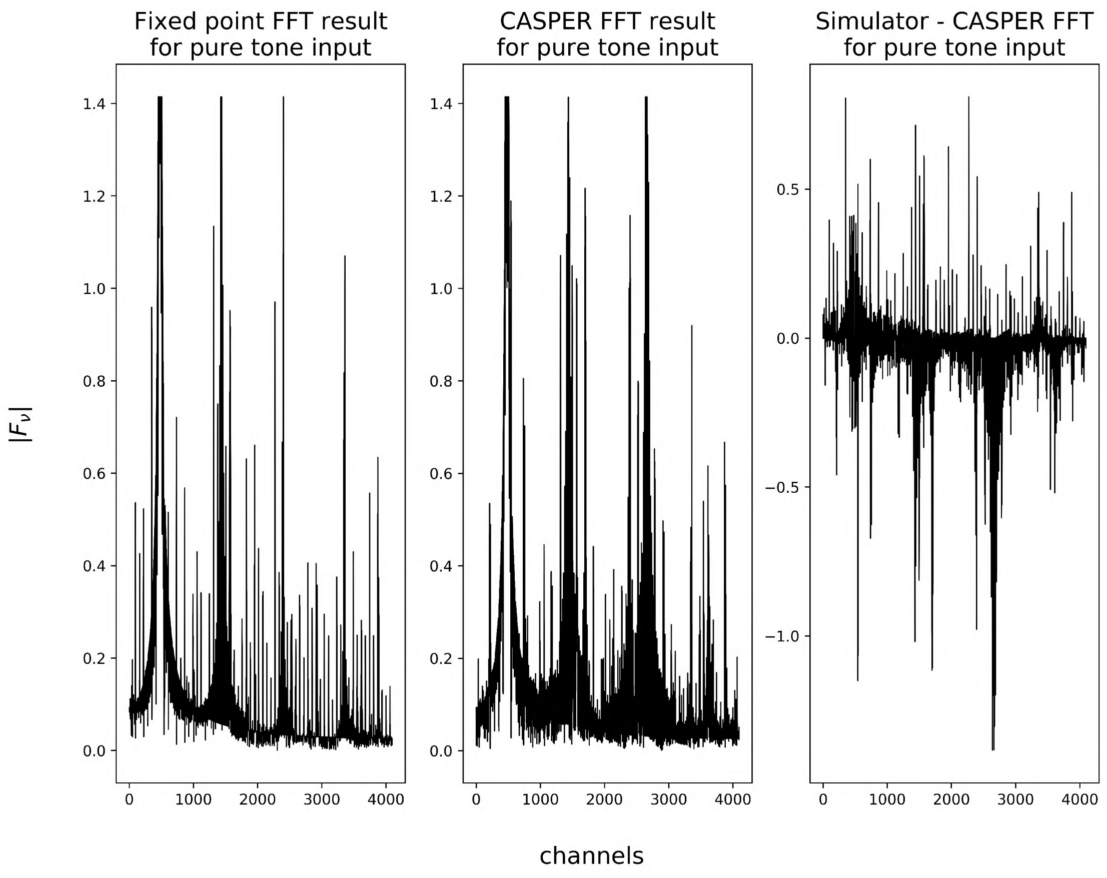

Figure 5.3: Pure tone input result for simulator fixed point and CASPER FFTs. This tone is the same as used in figure 5.1, but here, the FFTs overflow and saturate during operation due to a lack of shifting. The shift scheme used here is 2322 (0b100100010010). This shows a similarity in the overflow behaviour of both FFTs. The plot on the right shows the difference of the Python simulator FFT against the CASPER FFT.

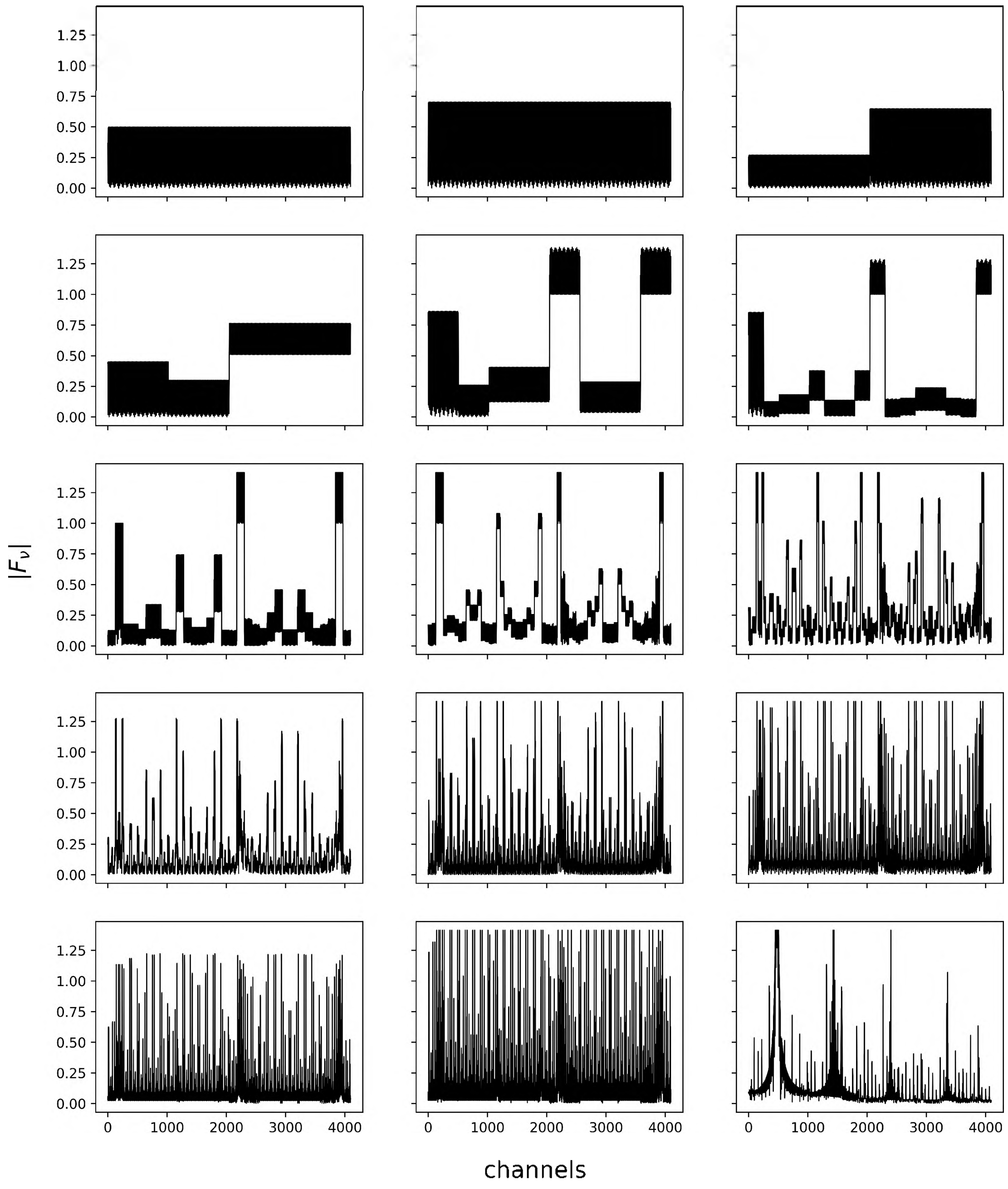

Figure 5.4: Sequential stages of the fixed point FFT processing a pure tone input shown in figure 5.3. This exhibits the functionality of the fixed point simulator and indicates where in the process overflow is occurring. Relating to figure 2.11 which details the location of stages of an 8-point FFT, here we display the result of each stage as data travels through a 8192-point FFT. The stages are read from top left to bottom right starting with the output of the PFB FIR, then the 13 stages of the FFT and ending with the bit-reversal of the output in stage 13. The root cause of spectral leakage can be investigated using this feature of the fixed point simulator.

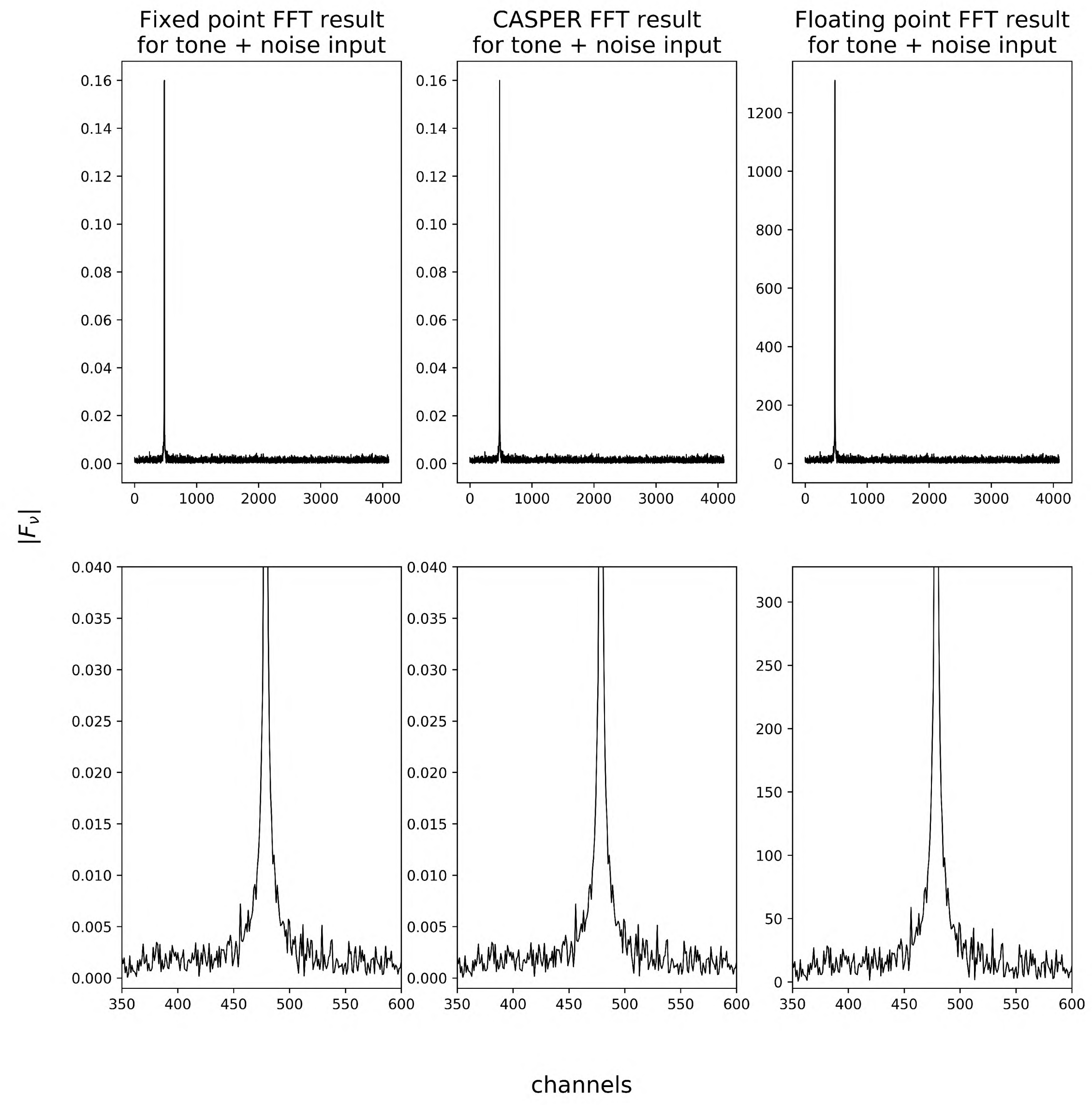

Figure 5.5: Tone with gaussian noise FFT result for the three different FFT implementations. The input signal has an SNR of 18.97dB. The delta spike sits in channels 478 and 479 for all FFTs. CASPER and the fixed point FFTs are 2^{13} times smaller in magnitude than the floating point FFT owing to the scaling of 8191 (0b1111111111111) applied in the CASPER and fixed point FFTs. The bottom row is a magnification of both the vertical and horizontal scale in order to note details in the noise and spike.

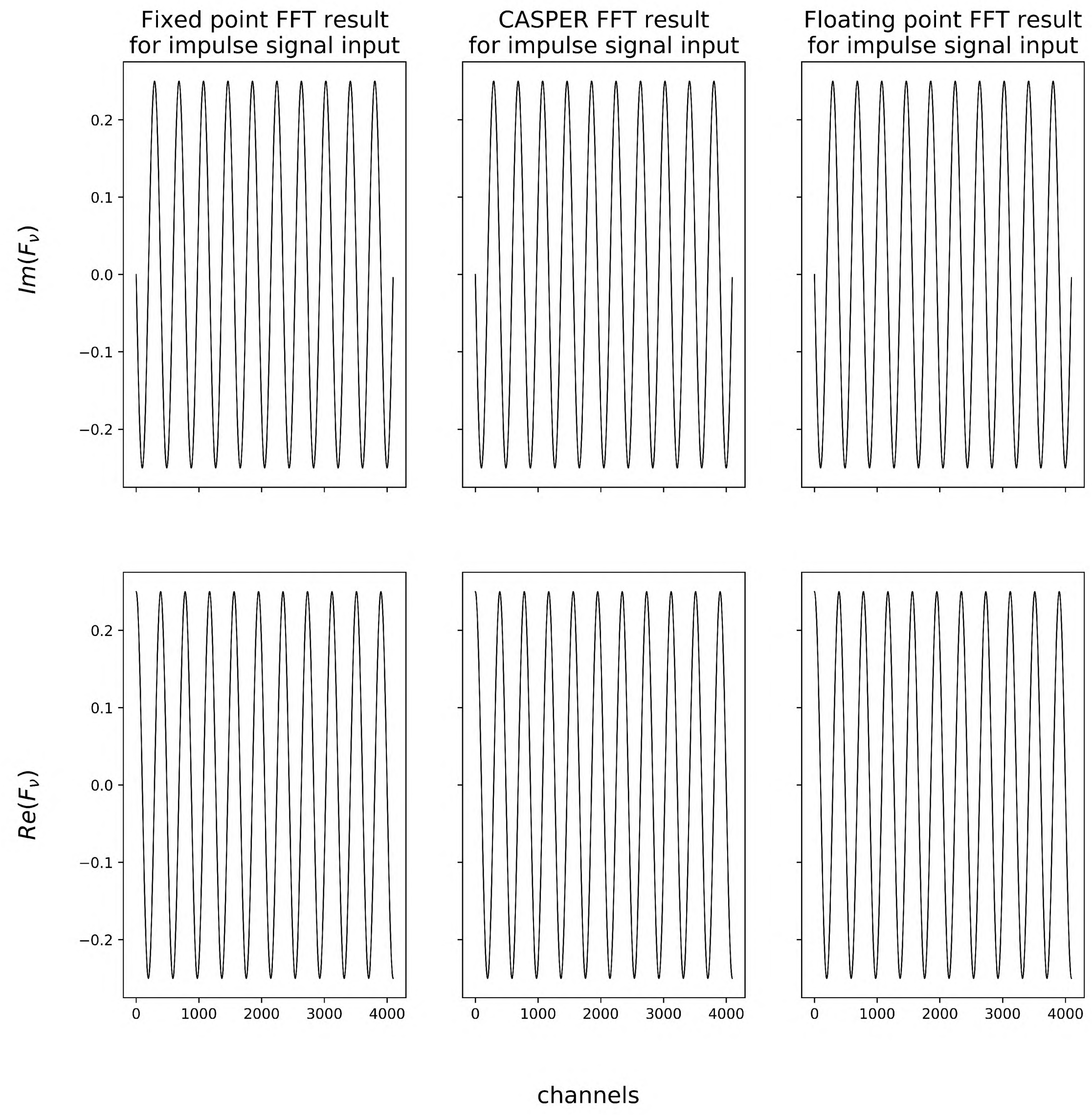

Figure 5.6: Impulse input result for the three different FFT implementations. Top row displays the imaginary components of each FFT output, whilst the bottom row displays the real components of each FFT output. The shift scheme used here is 0 (0b0000000000000) and as such, the fixed point and CASPER FFT are the same size as the floating point. This test shows that the various FFTs' outputs don't differ in phase.

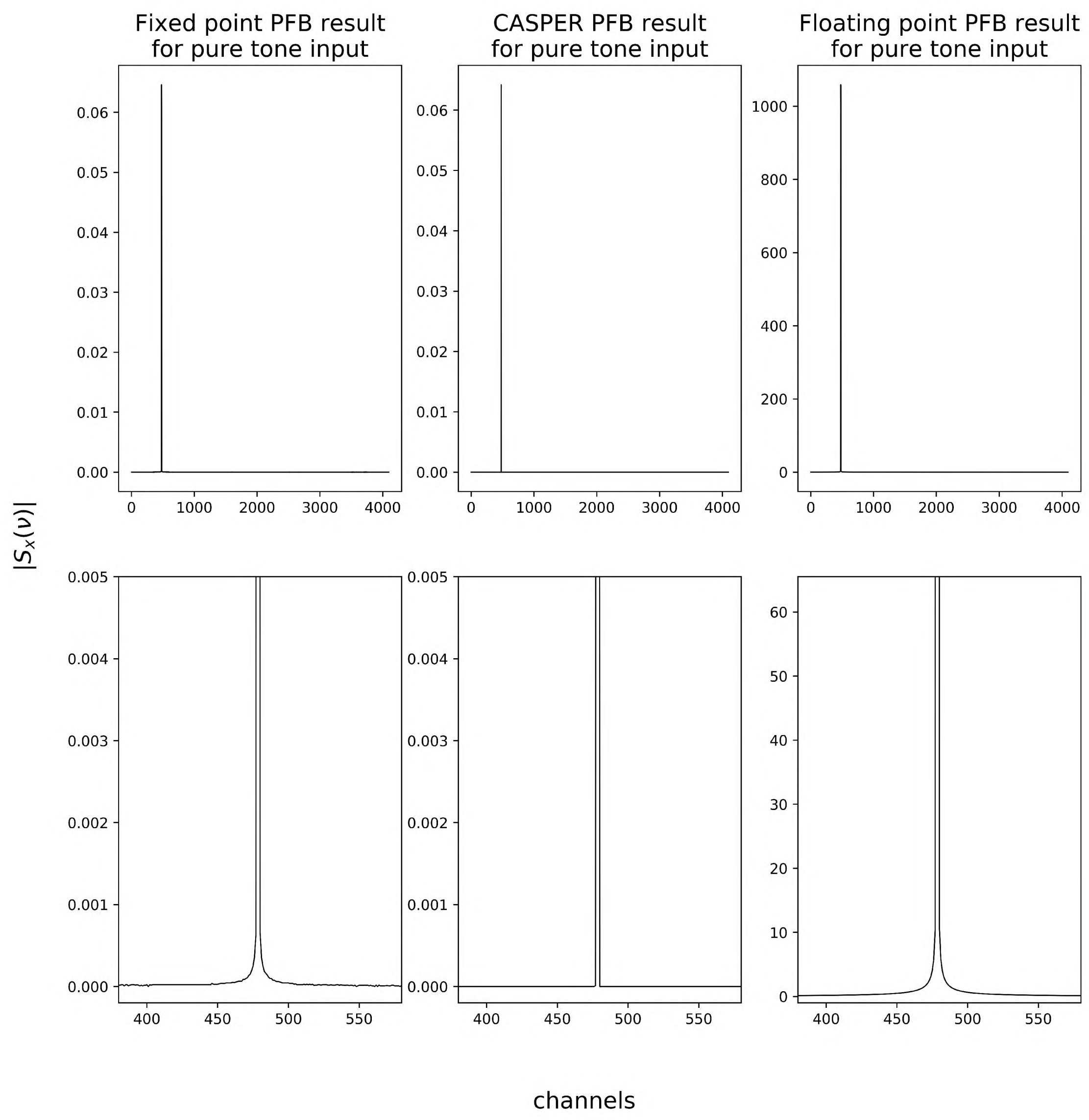

Figure 5.7: Top row is the magnitude of the three different PFB implementation results for a pure tone input. The bottom row is a magnification of both the vertical and horizontal scale in order to note details in the spike produced. The input tone had an amplitude of $\frac{1}{2}$. The delta spike sits in channels 478 and 479 for all PFBs. CASPER and the Simulated fixed point PFBs are 2^{14} times smaller in magnitude than the Simulated floating point PFB owing to the scaling of 8191 (0b1111111111111) in the FFT and the scaling by 2 in the FIR of the CASPER and fixed point PFBs.

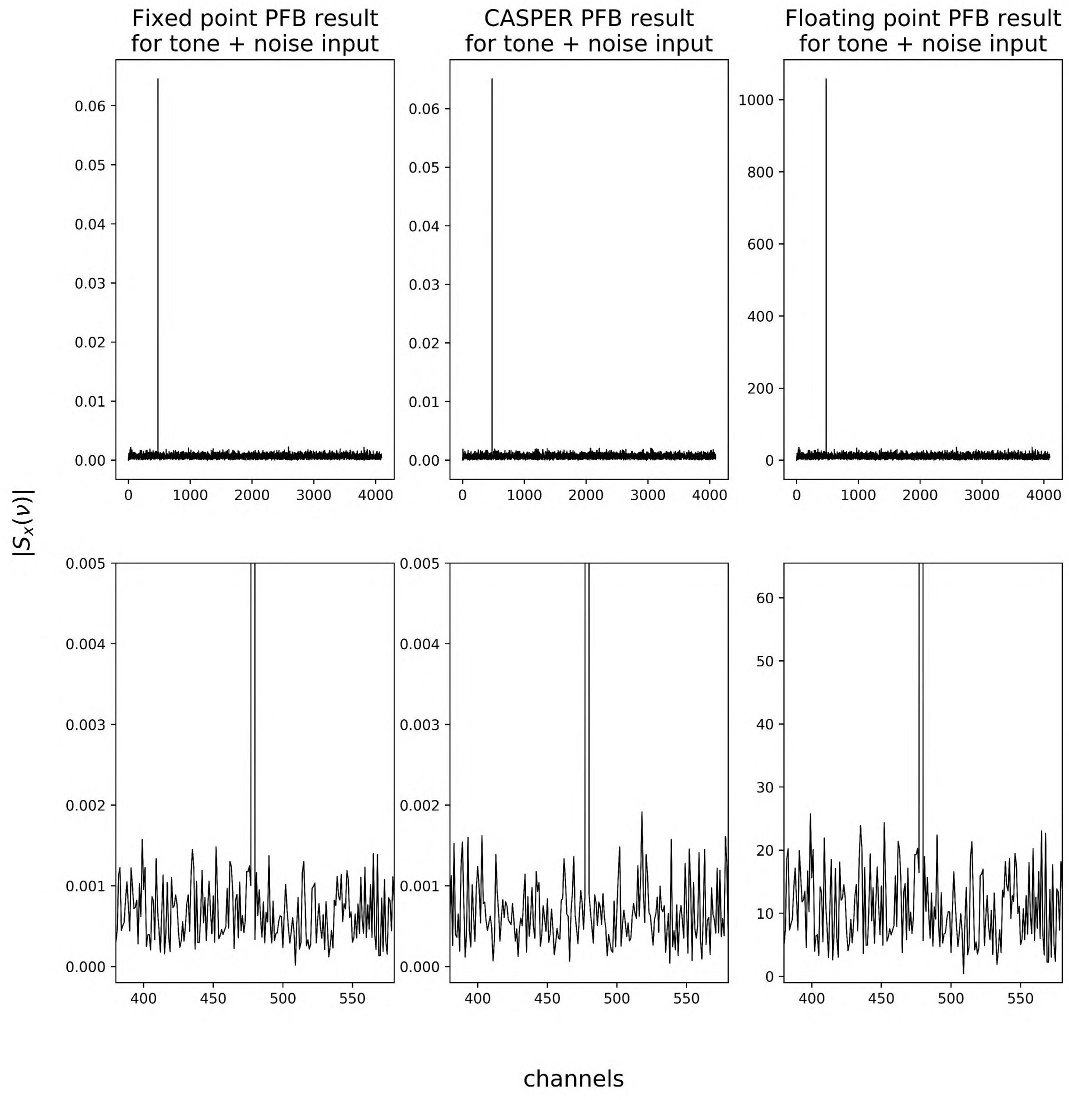

Figure 5.8: Top row is the magnitude of the three different PFB implementation results for a noisy tone input. The bottom row is a magnification of both the vertical and horizontal scale in order to note details in the noise and spike produced. The input tone had an amplitude of $\frac{1}{2}$ and the input an SNR of 19.06dB. The delta spike sits in channels 478 and 479 for all PFBs. CASPER and the Simulated fixed point PFBs are 2^{14} times smaller in magnitude than the Simulated floating point PFB owing to the scaling of 8191 (0b1111111111111) in the FFT and the scaling by 2 in the FIR of the CASPER and fixed point PFBs.

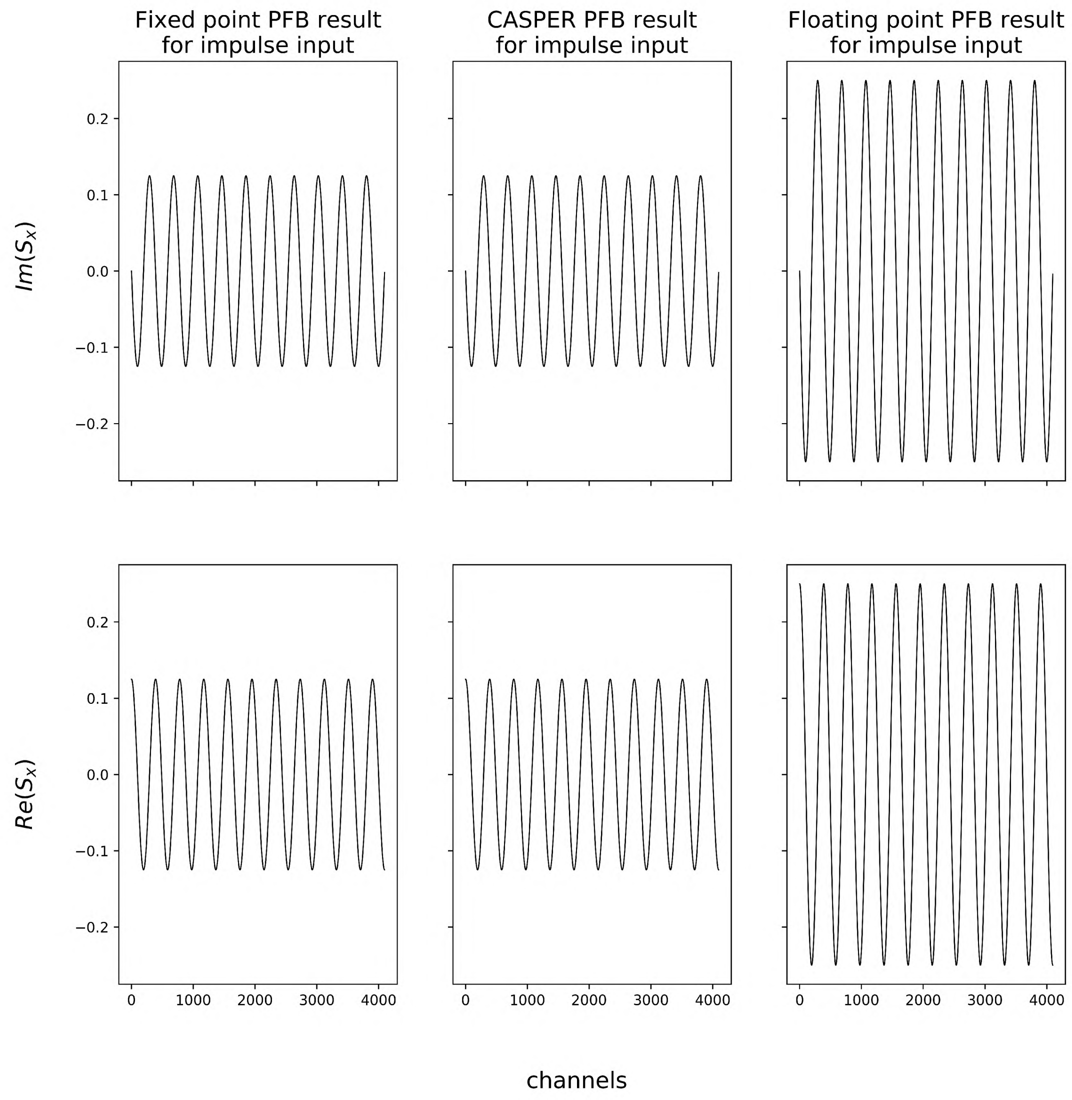

Figure 5.9: Impulse input result for 3 PFBs. Top row displays the imaginary components of each PFB output, whilst the bottom row displays the real components of each PFB output. The shift scheme used here is 0 (0b000000000000) and as such, the fixed point and CASPER PFB are half the size of the floating point PFB owing to the scaling of 2 in the fixed and CASPER FIRs. This test shows that the various PFBs' outputs don't differ in phase.

Discussion

It appears that the Simulated fixed point FFT mimics the CASPER fixed point FFT well. Furthermore, as confirmation that both fixed point FFTs are producing a correct result, they match with the floating point implementation when shifting aggressively (except for in magnitude as a result of said shifting). This is indicated by figures 5.1, 5.2 and 5.5. Figure 5.6 further confirms this, since the results produced are in-phase when given a impulse as input (else we would expect to see a differing frequency between the sinusoidal results).

Where the fixed point FFTs differ notably is in figure 5.3. This was of concern since this thesis in part looks to address the issues of overflow in the CASPER PFB. This difference proved very hard to fix and is likely the result of negligible differences in simulators and data conversions. Overall, the characteristic '4 humps' and the saturating '3 peaks' do indicate a comfortable mimicry of the overflow to be expected in the CASPER fixed point FFT. Figure 5.4 shows the staged output of the fixed point Python simulator for the overflow case in figure 5.3. It further shows that overflow starts in stage 3 and from there propagates.

The PFB results are noted to differ in both the sharpness of the base of the spike for figure 5.7 and the noise in figure 5.8. The FIRs in each simulator are confirmed to generate their coefficients according to equation 3.1 and scale in the same way. The only differences likely to occur are owed to differences in how complex arithmetic is performed. This is unlikely to cause such a difference however. Rather, it is supposed that the output taken from the CASPER PFB does not correspond with the input vector used to test the other PFBs. While efforts were made to ensure that the input processed by all PFBs was the same through the use of sinks and timing constraints in the Simulink design, it proved very difficult. The CASPER blocks are really intended for streaming processing and not to process discrete vectors and view the corresponding outputs. Hence, proof that the simulated PFB behaves the same as the CASPER instance is given by the approximately equal SNR of the outputs (in figure 5.8) and equal magnitudes of the output spike (in both figures 5.7 and 5.8).

5.2 Shift register regime versus Quantisation Efficiency

A staff member of SARAO, Marcel Gouws, ran a study where for a software PFB simulator of their own, they varied the contents of the FFT shift register when processing a *white* gaussian input and calculated the output quantisation efficiency by comparing it with a floating point equivalent [internal communication]. They calculated the quantisation efficiency using the following equation from [4]:

$$\eta_{shift} = \frac{\langle x, \hat{x}^* \rangle^2}{\langle x, x^* \rangle \langle \hat{x}, \hat{x}^* \rangle} \tag{5.3}$$

Here x denotes the output from the floating point FFT simulation and $\hat{x}$ denotes the output from the fixed point FFT simulation. The operator $\langle a, b \rangle$ denotes the mean $\frac{1}{N}\sum_{i=1}^{N} a_i b_i$ of complex vectors a and b and $()^*$ denotes the complex conjugate.

Here, the aim is to repeat their test and check for a matching result before exploring the quantisation efficiencies attained for a selection of un-tested shift-regimes. While they ran these tests off a full PFB, it was deemed sufficient to run these tests on the FFTs alone, since the FIR is unaffected by the shift-register setup.

The quantisation efficiency is calculated for 1k, 4k and 32k FFT designs for noise inputs of varying dBFS values. For the initial test, each simulated fixed point FFT had their shift scheme varied from no shift to a full aggressive shift.

Input noise vectors of $\approx -15, -20, -25$ and -30dBFS were used since these were the ones Marcel used. Furthermore, for the input to the fixed point FFT simulator, the noise input was an $\langle 18, 17 \rangle$ fixed point number (the format used by the PFB instance studied in this thesis). The fixed point FFT processes at this same precision and outputs a fixed point complex value at this precision. The floating point FFT simulator used double precision float inputs, processed with double precision and outputted a double precision complex value.

The quantisation efficiency results for the 1k, 4k and 32k FFTs are shown in figures 5.10, 5.11 and 5.12 respectively.

Recalling subsection 3.5.2, for a white gaussian noise input, a bit growth of $\frac{1}{2}\log_2(N)$ bits is expected. For the 1k fixed point FFT then, a growth of at most 5.5 bits is expected. For this FFT, we've considered 6 and more right-shifts 'over-shifting' and 5 and less right-shifts 'under-shifting'.

For the 4k fixed point FFT, a growth of at most 6.5 bits is expected. For this FFT, we've consider 7 and more right-shifts 'over-shifting' and 6 and less right-shifts 'under-shifting'.

Finally, for the 32k fixed point FFT, a growth of at most 8 bits is expected. While 8 right-shifts is then critically shifting, we've grouped it with the 'under-shifting' plot, and have 9 right-shifts or more considered as 'over-shifting'.

Figure 5.13 shows the efficiencies produced when testing alternative shift schemes for the 4k

FFTs.

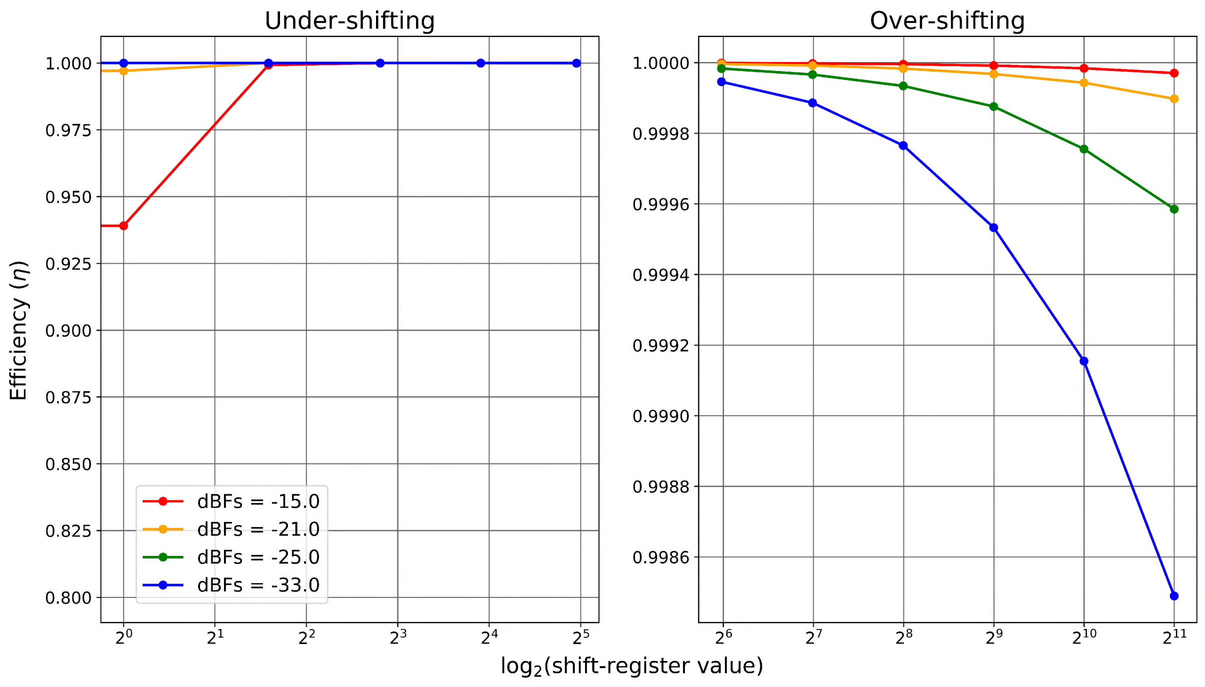

Figure 5.10: Shift-register versus efficiency analysis for fixed point 1k FFT. For varying input signal dBFS levels of -15dBFS, -21dBFS, -25dBFS and -33dBFS several shift register regimes are employed. The plot on the left is for under-shifting (≤ 5 shifts) while the plot on the right is for over-shifting (≥ 6 shifts). Under-shifting uses shift-register values: [0,1,3,7,15,31], while over-shifting uses: [63,127,255,511,1023,2047].

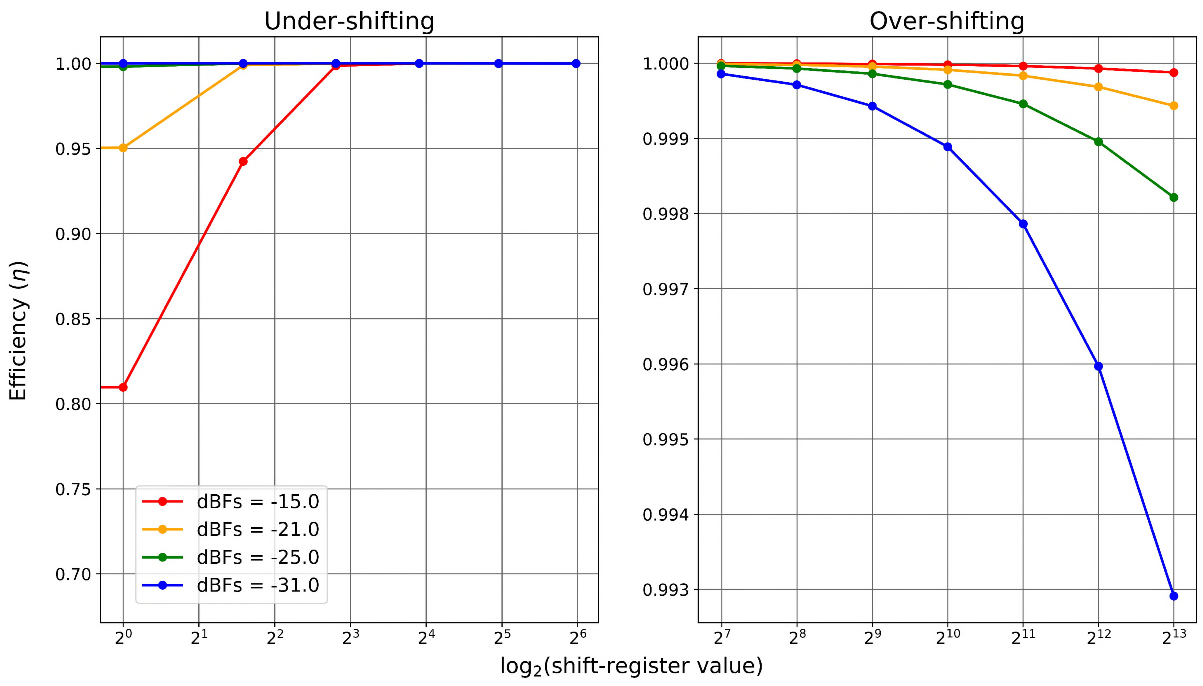

Figure 5.11: Shift-register versus efficiency analysis for fixed point 4k FFT. For varying input signal dBFS levels of -15dBFS, -21dBFS, -25dBFS and -31dBFS several shift register regimes are employed. The plot on the left is for under-shifting (≤ 6 shifts) while the plot on the right is for over-shifting (≥ 7 shifts). Under-shifting uses shift-register values: [0,1,3,7,15,31,63], while over-shifting uses [127,255,511,1023,2047,4095,8191].

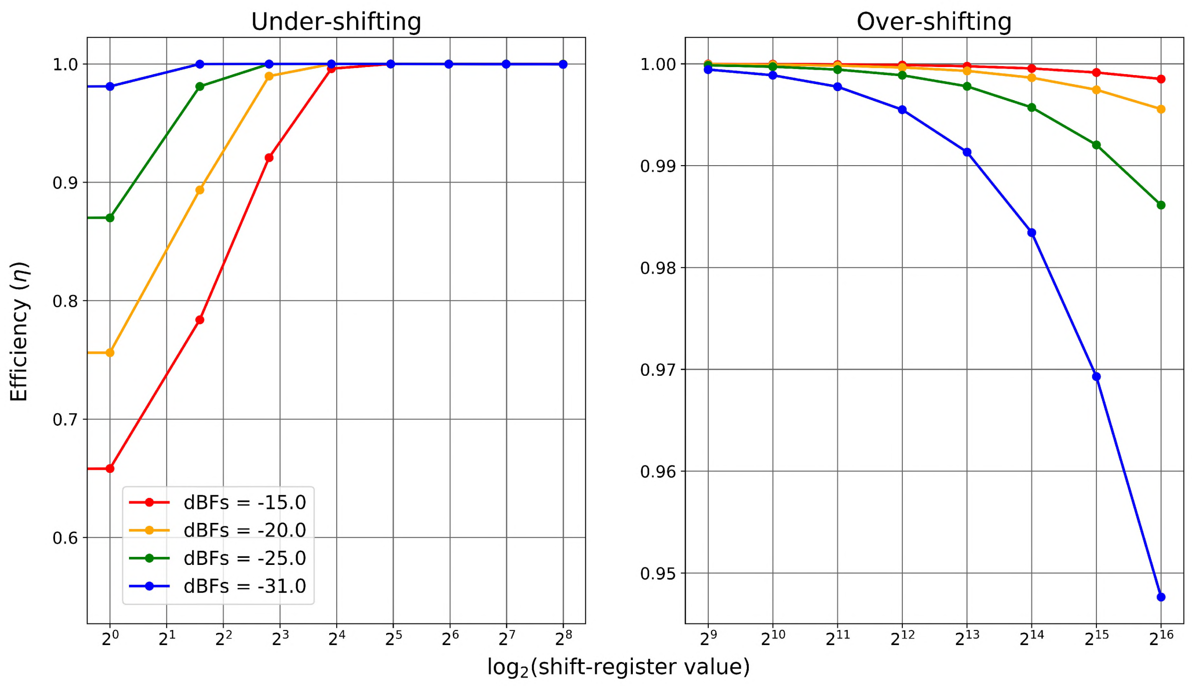

Figure 5.12: Shift-register versus efficiency analysis for fixed point 32k FFT. For varying input signal dBFS levels of −15dBFS, −20dBFS, −25dBFS and −31dBFS several shift register regimes are employed. The plot on the left is for under-shifting ($\leq$ 8 shifts) while the plot on the right is for over-shifting ($\geq$ 9 shifts). Under-shifting uses shift-register values: [0,1,3,7,15,31,63], while over-shifting uses [63,127,255,511,1023,2047,4095,8191].

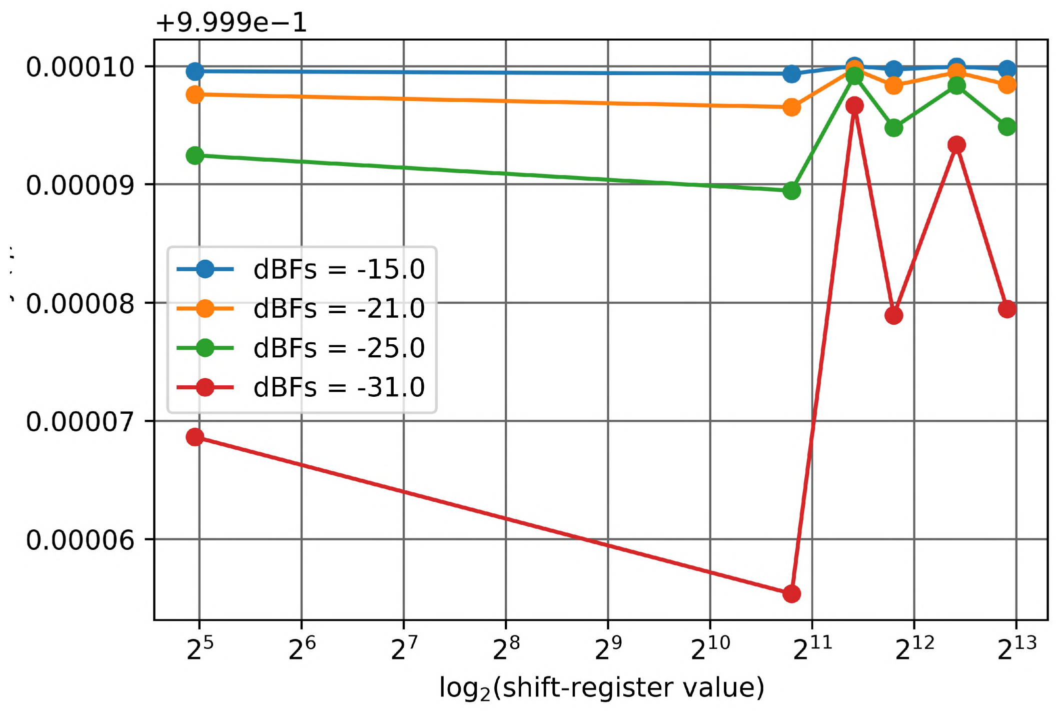

Figure 5.13: Shift-register versus efficiency analysis for fixed point 4k FFT. For varying input signal dBFS levels of −15dBFS, −21dBFS, −25dBFS and −31dBFS several shift register regimes are employed. Differing from figure 5.11 however, rather than test the effect of under and over-shifting, here some mixed shift schemes are tested namely: [1782,2730,3564,5461,7695]

Discussion

The results displayed for the various length FFTs in figures 5.10, 5.11 and 5.12 are in line with the findings Marcel made and with what is expected. Rather than have one contiguous graph, the plots are split into 'under' and 'over'-shifting since the loss in efficiency for under-shifting is substantially worse than for over-shifting and in sharing a y-axes, the effects for over-shifting would not have been very pronounced.

As anticipated, higher dBFS inputs are more efficient with more shifting than lower dBFS and visa versa. The higher the dBFS of the input signal, the more bits it will occupy and as such less precision is lost with over-shifting than lower dBFS inputs. Similarly, larger dBFS are more prone to overflow with under-shifting and are as such less efficient leading to a less efficient result.

Larger FFTs look to gradually perform worse when over-shifting. Comparing the largest shift scheme efficiency result for each FFT, it appears that compared to the 1k FFT, the 4k FFT is ≈ 0.005 less efficient while the 32k FFT ≈ 0.05 less efficient. This is likely due to the input dBFS levels chosen being suited to the 1k FFT and not necessarily to the other FFTs. This is apparent when noting that the efficiency yielded for the $2^{11} - 1$ shift scheme of each FFT is approximately equal. Shifting any more for the same input (as the 4k and 32k do) is bound to gradually deteriorate the quantisation efficiency of the result.

Similarly, larger FFTs look to gradually perform worse when under-shifting. However, unlike with the over-shifting scenario where each FFT has a differing max shift-scheme, here all the FFTs have the same minimum shift-scheme of 0. The reason larger FFTs produce worse efficiencies then, is likely explained by them having more stages i.e. given less shifts, more stages leads to an increased loss of quantisation efficiency due to overflow.

Figure 5.13 is an interesting scenario tested only for the 4k FFT. Unlike the 3 other plots before, here the shifts (apart from shift value 31) are split up and don't all occur contiguously. When comparing the other results with the result for 31 (which was seen as having the highest quantisation efficiency in the previous 4k FFT test), it is apparent that while 4 of them compare better overall, two are vastly more efficient. These two are the scheme 2730 (0b0101010101010) and 5461 (0b1010101010101). Furthermore, shift scheme 2730 has the same number of shifts as shift-scheme 31, suggesting that an alternating shift-scheme performs better for a given number of shifts than having them contiguously placed. Finally, while the previous plots may have alluded to a smooth relation between quantisation efficiency and shift-scheme, this plot re-iterates the point made in subsection 3.5.2 about a non-obvious mapping between shift-scheme and quantisation efficiency/number of shifts.

Recall that over-shifting affects the quantisation efficiency by corrupting the LSB with rounding and that under-shifting affects quantisation efficiency by corrupting the MSB with overflow. This is visible by how the quantisation efficiency loss is far more affected by under-shifting than over-shifting. Hence, it has been generally remarked that an aggressive scheme be used (even at the risk of over-shifting) - especially in the presence of RFI.

5.3 Rounding scheme test

The effects of rounding are not really relevant when looking at a single output since it is the least-precision bits that are affected. Rather, the effects become prevalent when integrating the results.

For this test, nine 4k Python simulator fixed point FFTs are used. There are three 10-bit, three 14-bit and three 18-bit FFTs. Within each bit-width category, each FFT will use a different rounding scheme: even-rounding, infinite-rounding and truncation. Accompanying the fixed point FFT results is a double precision floating point FFT result so as to highlight finite precision error caused in the fixed point processing.

Across each bit-width category, the same input vector is used to ensure that any differences noted in outputs are owed to 'round-off' noise from the varying rounding schemes.

The input used for all FFTs is a single tone with added white gaussian noise. The tone has an amplitude of 1/4 and the noise has a standard deviation of 0.044... This gives an SNR of $\approx$ 30dB. Each FFT produces an output of bit-width equal to its input bit-width (i.e. the 10-bit test takes a 10-bit input, processes at 10-bits and produces a 10-bit output etc.). The shift-scheme used in all fixed point FFTs is 0b111111111111 since the tone is large and any overflow would dominate the effect under study (recalling that rounding is a LSB effect and overflow is a MSB effect).

For each fixed point FFT, thirty three thousand absolute outputs are accumulated as double precision floats (since we are not testing finite precision affects in accumulation) and are referenced against thirty three thousand accumulated floating point FFT absolute outputs.

By comparing the fixed and floating point results, the first test is to check whether there are any differences in the outputs produced based on the differing rounding schemes. The second test will investigate whether an increased precision over 10, 14 and 18-bits mitigates the effects of the differing rounding schemes. The results for the even rounding scheme are shown in figures 5.14 and 5.15. The infinite rounding scheme in figures 5.16 and 5.17. And the truncation in figures 5.18 and 5.19.

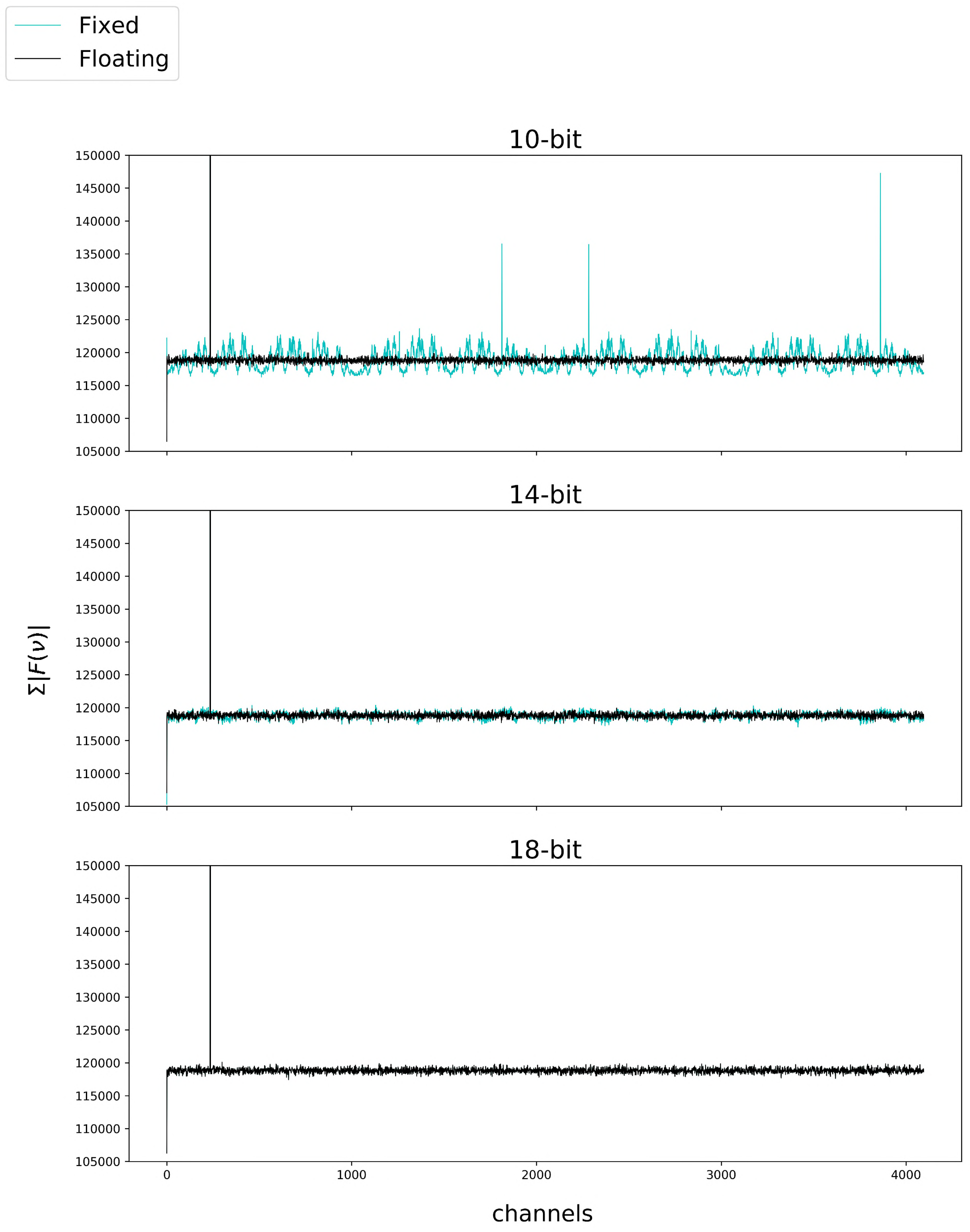

Figure 5.14: Summing 33000 even-rounding fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal.

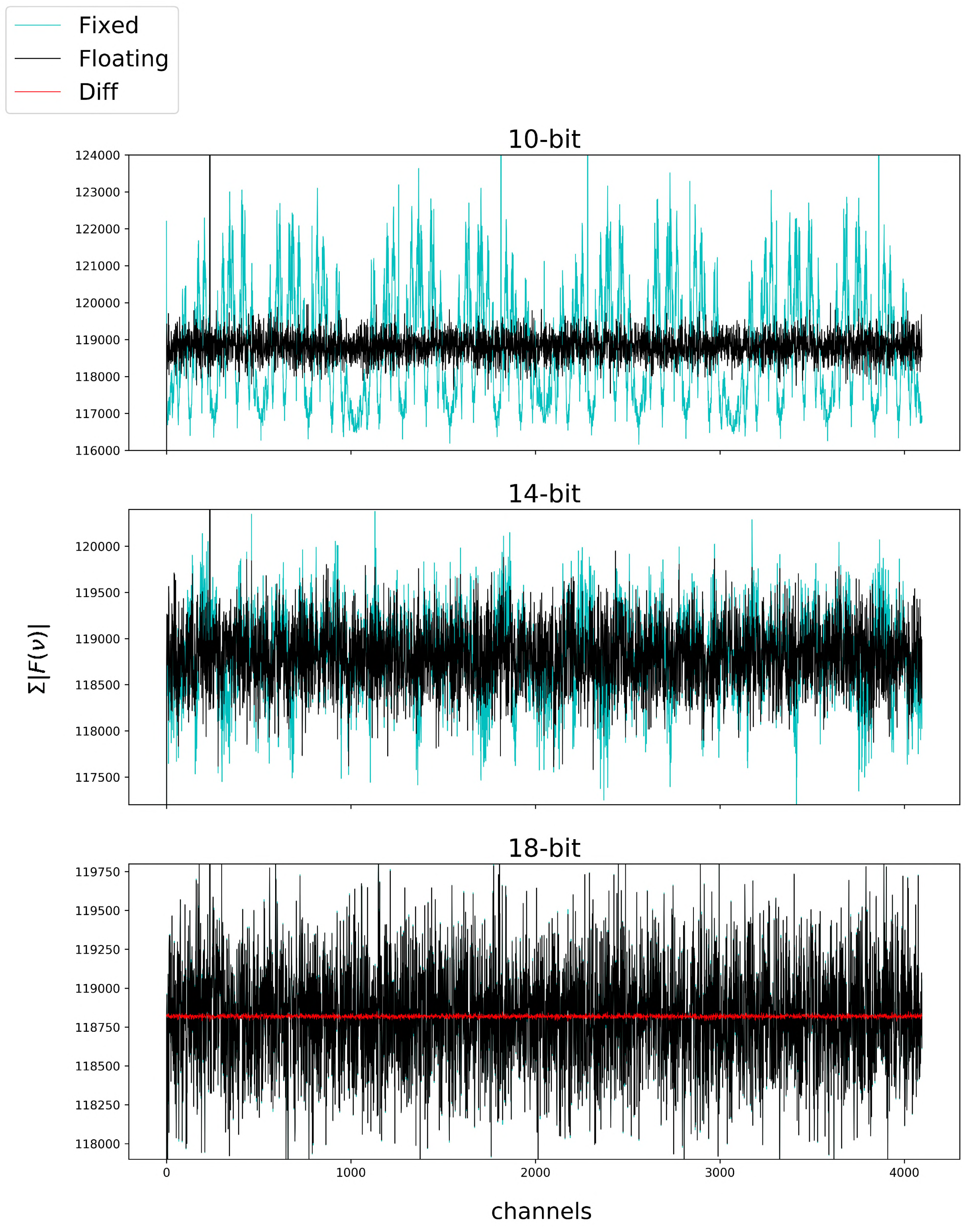

Figure 5.15: Summing 33000 even-rounding fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal. This is a vertical scaling to inspect the noise floor in figure 5.14.

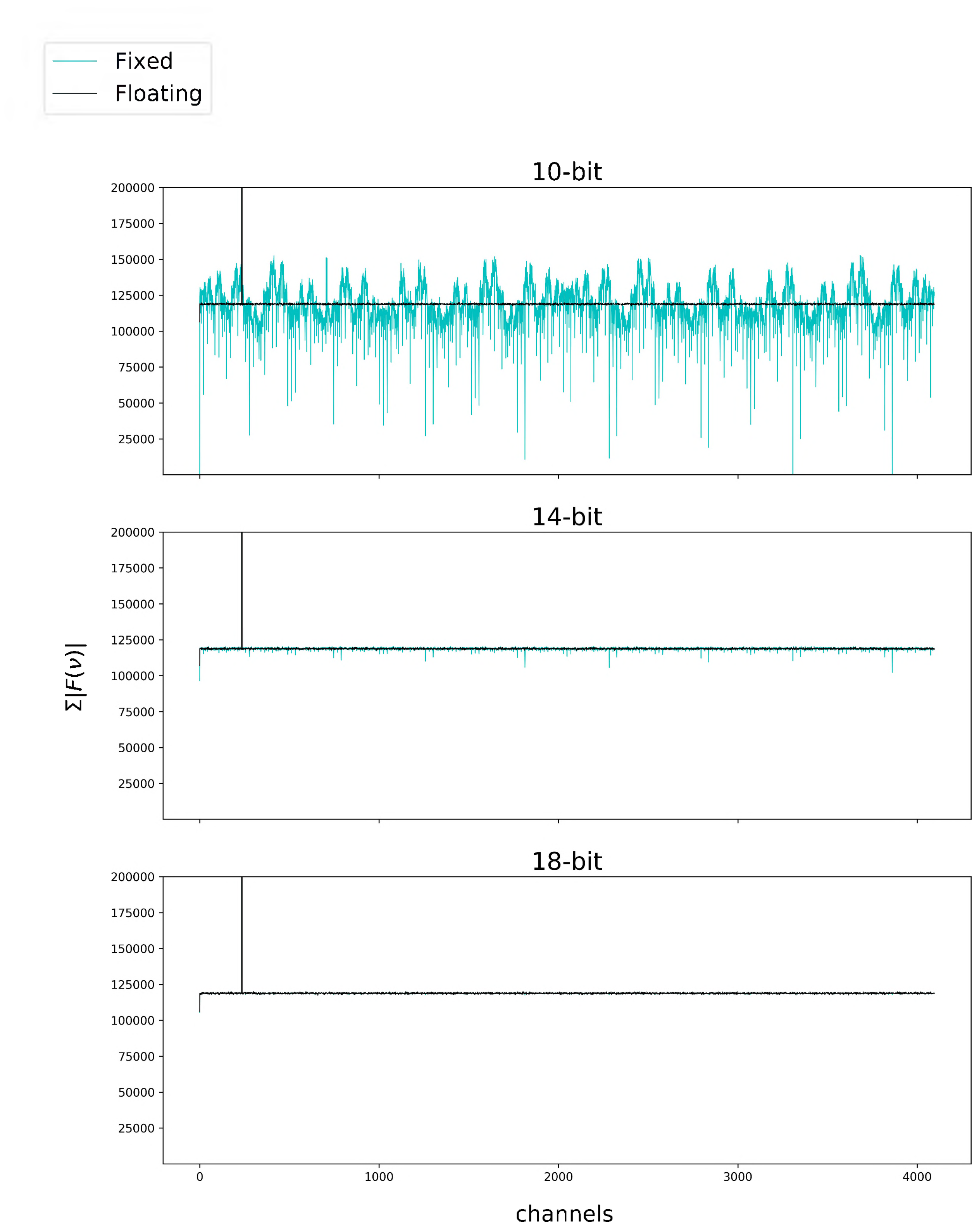

Figure 5.16: Summing 33000 infinite-rounding fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal.

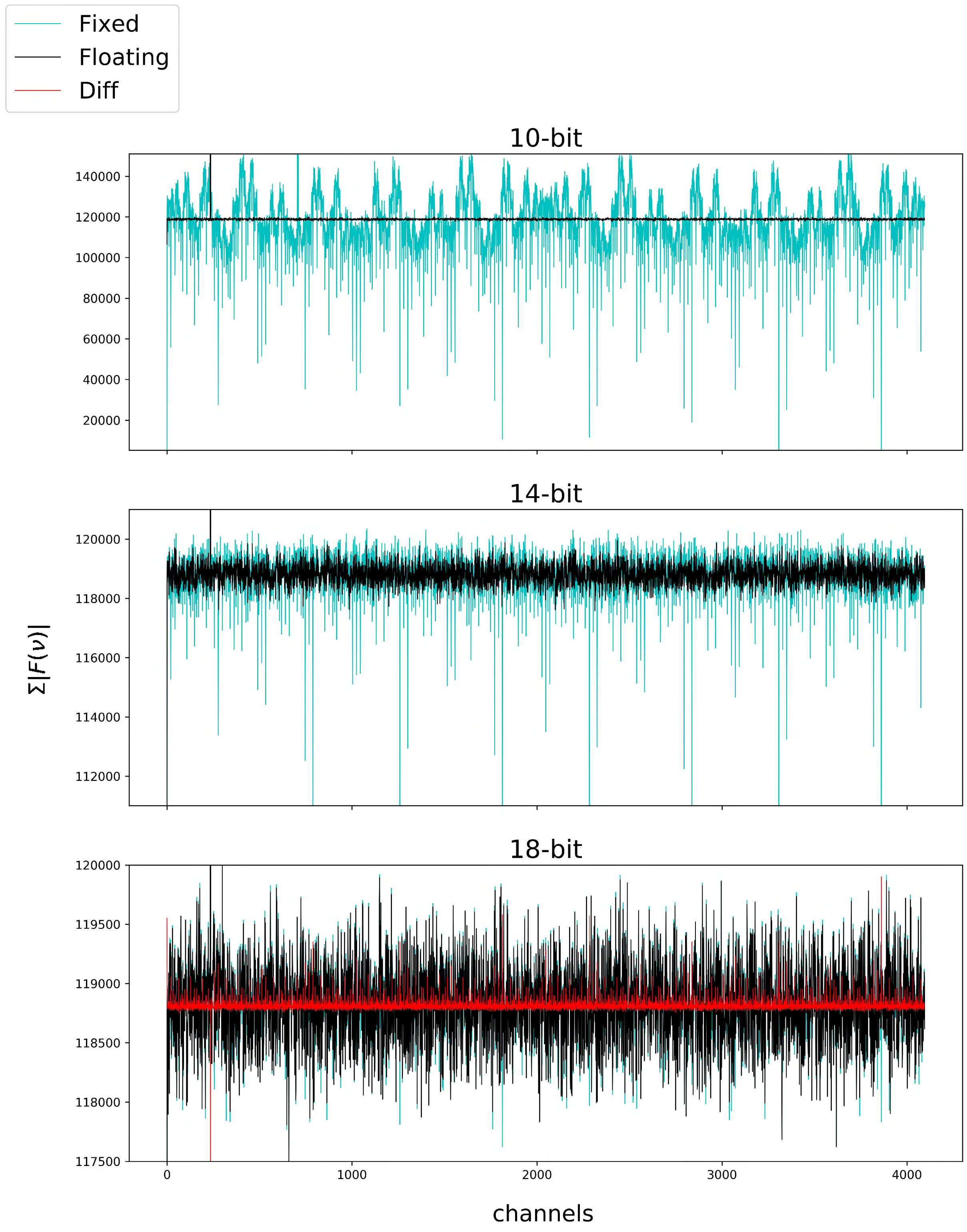

Figure 5.17: Summing 33000 infinite-rounding fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal. This is a vertical scaling to inspect the noise floor in figure 5.16.

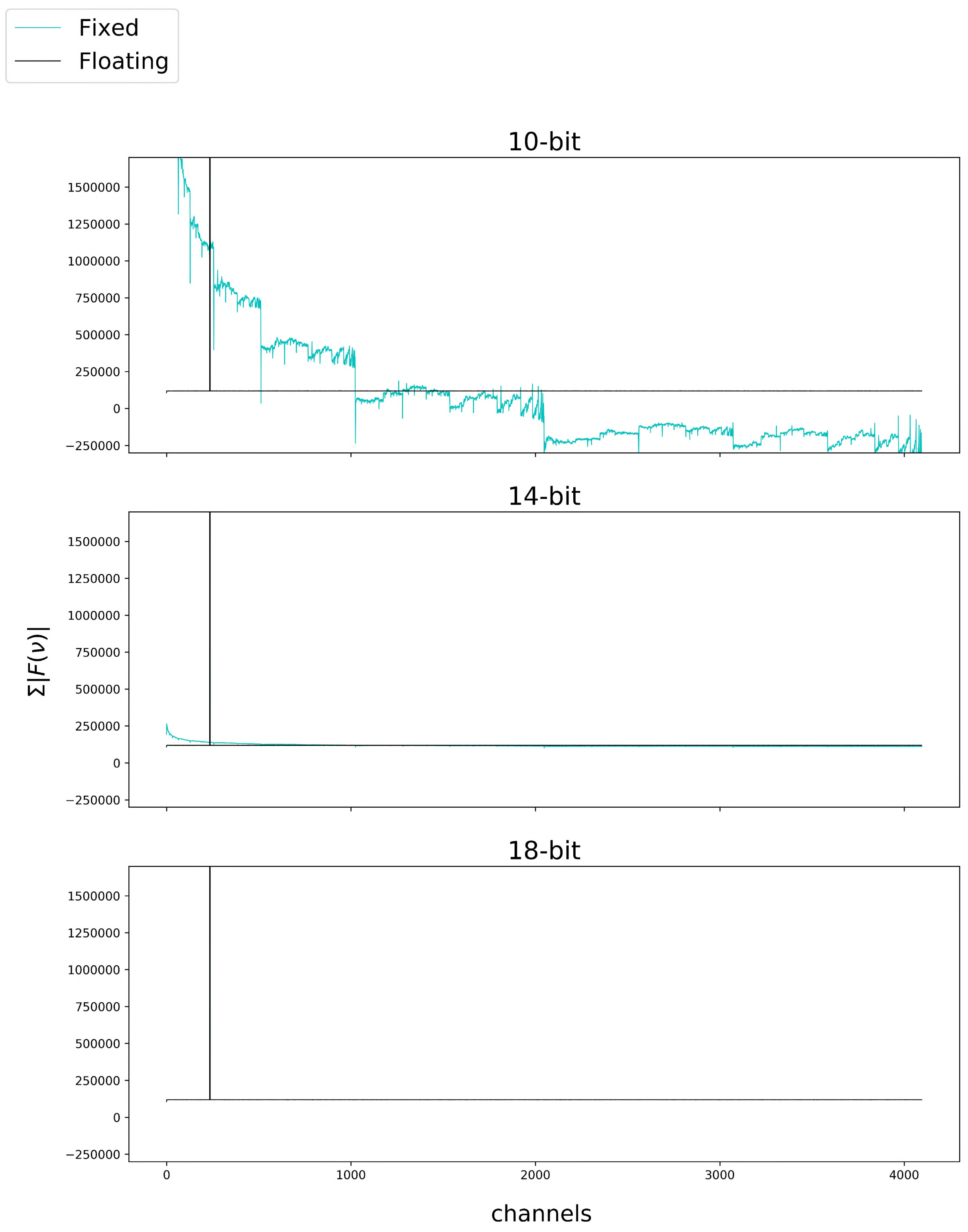

Figure 5.18: Summing 33000 truncating fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal.

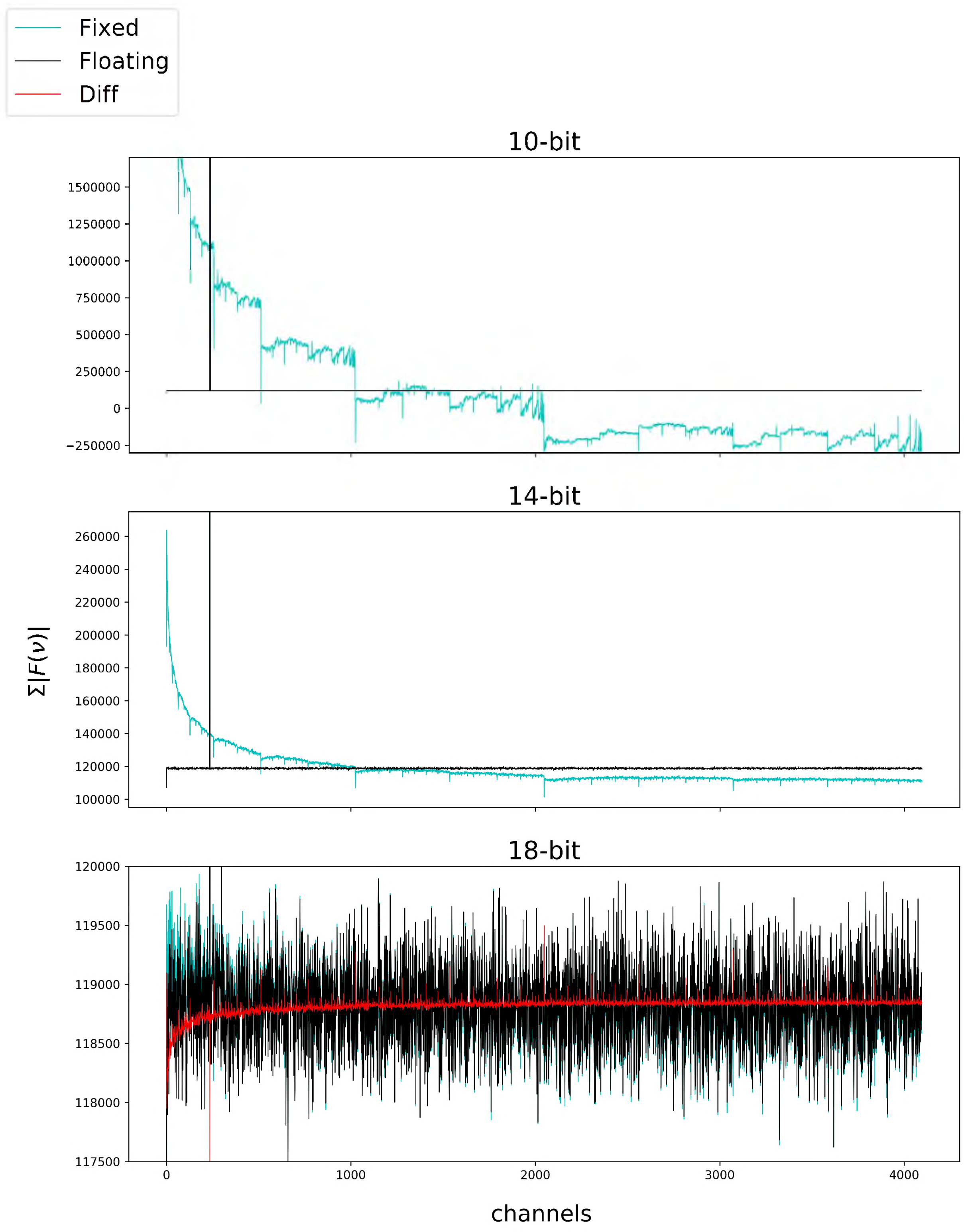

Figure 5.19: Summing 33000 truncating fixed point FFT absolute outputs and referencing them against 33000 summed floating point FFT absolute outputs for an uncorrelated noise with tone input signal. This is a vertical scaling to inspect the noise floor in figure 5.18.

Discussion

Rounding is an error that is directly proportional to the precision of the fixed point number. This is clearly indicated in every rounding scheme case by the substantial drop in quantisation noise with higher bit-widths.

The studying of the results in each rounding scheme reveals unique characteristics in their behaviour.

Beginning with the even-rounding scheme (figures 5.14 and 5.15), at low bit precision (10-bits) two characteristics are observable: the upward spikes induced (mirroring the tone spike in some way) and the symmetric rippling in the noise floor. The 14-bit instance repeats this ripple effect (as shown by the 'clumping' of quantisation noise) but the spikes appear to have been removed. At the 18-bit precision these effects have become largely negligible, although close inspection of the red error does reveal the ripple in the quantisation noise floor.

The infinite-rounding scheme too reveals a symmetric ripple in the quantisation noise floor, but its prominent effect is the downward 'dips' (see figures 5.16 and 5.17). These dips are repeated in spacings approximating the space from the DC channel to the sinusoid spike (and hence are anticipated to occur as the result of the input tone).

The final rounding scheme tested is that of truncation. Given the loss in precision this scheme induces, this was expected to produce the worst result (see subsection 2.11.3). Inspecting figures 5.18 and 5.19 this scheme clearly introduces substantial finite precision error to the spectral floor. Most obvious is the gradual slope toward the DC channel which is prevalent throughout each bit-width. Second to this are the discontinuous 'dips' in the spectrum. These are more severe than the infinite-rounding scheme and occur more frequently around the tone spike.

Ideally, future tests would outline the relation between these rounding characteristics and the input signal. Does the frequency and magnitude of the input tone affect the periodicity and magnitude of the dips and spikes in the quantisation noise? Similarly, do they affect the frequency of the rippling effect seen in the noise floor of the even and infinite rounding schemes? Finally, do they affect the slope surrounding the DC channel of the truncation scheme?

5.4 Dual polarisation test

The above tests use a real input vector i.e. they process only one receiver feed. This is unlike the MeerKAT setup where two receiver feeds are fed into the correlator see section 2.6.

For this test, a 4k, 8-tap, even rounding, Hann, dual-polarisation fixed point PFB is used that accepts two 10-bit complex fixed point inputs and produces two complex fixed point 18-bit outputs.

The first test proves that the dual polarisation splitting works. To do this, two signals, one a *white* noise signal and the other a noisy tone signal, are inserted as the real and imaginary parts into the dual polarisation Python PFB simulator. The output of this is compared with the results of processing the same two inputs through a 4k, 8-tap, Hann dual-polarisation floating point PFB.

The next test checks whether the effects of overflow in one polarisation leak into the other if the tone in the one polarisation is sufficiently strong and overflows. This is done by comparing the fixed and floating point PFB results.

The final test will then check whether in a non-overflow scenario, there is any residue due to rounding.

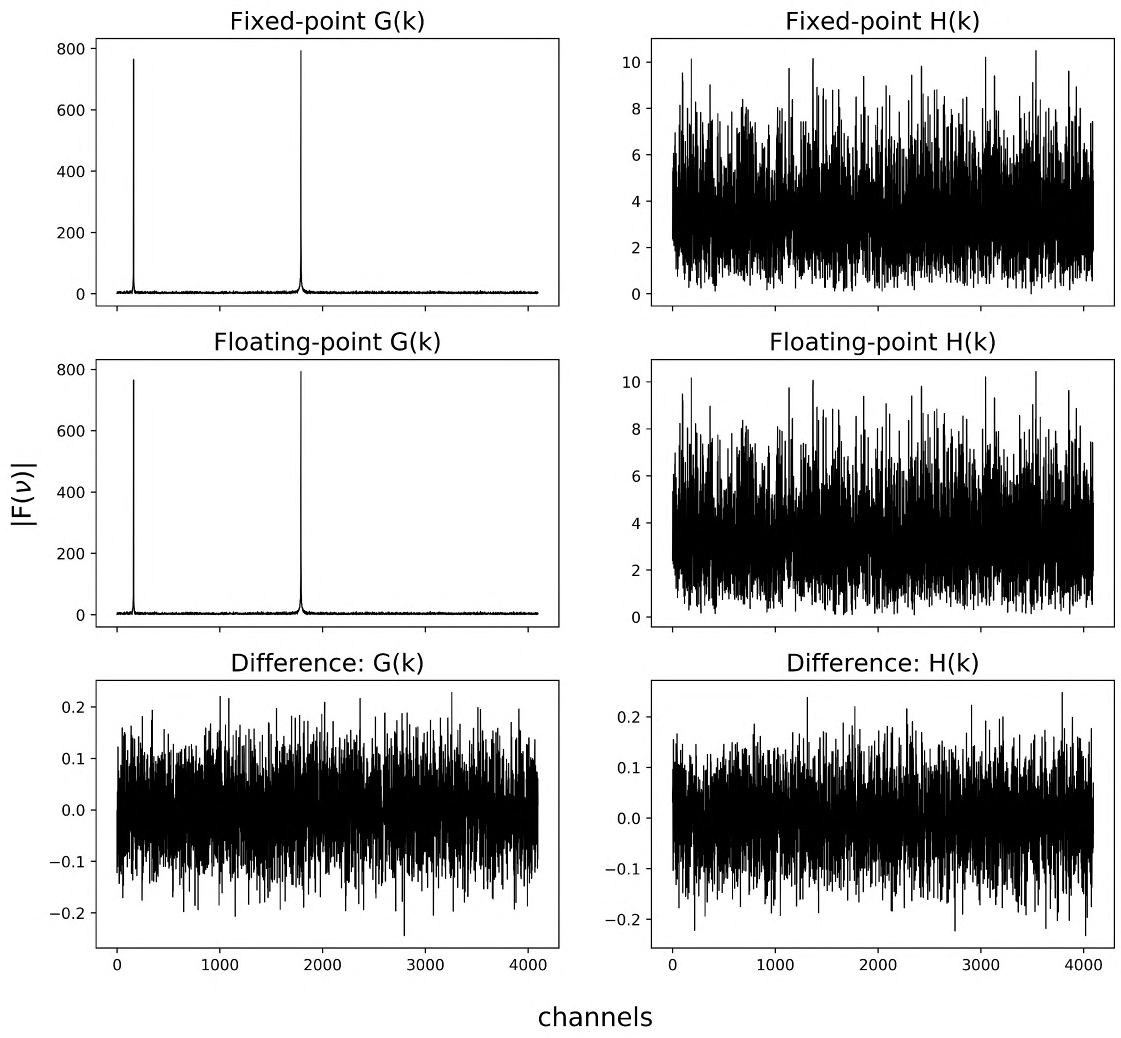

Figure 5.20: Dual polarisation processing of a real noisy two-tone vector of SNR≈ 30dB and an added imaginary noise vector. Here, the fixed point simulator and floating point simulators are compared. The last row indicates the difference |Float PFB| − |Fixed PFB| for both G(k) and H(k). Here an aggressive shift scheme 0b1111111111111 is used to avoid overflow in the polarisation containing the tones.

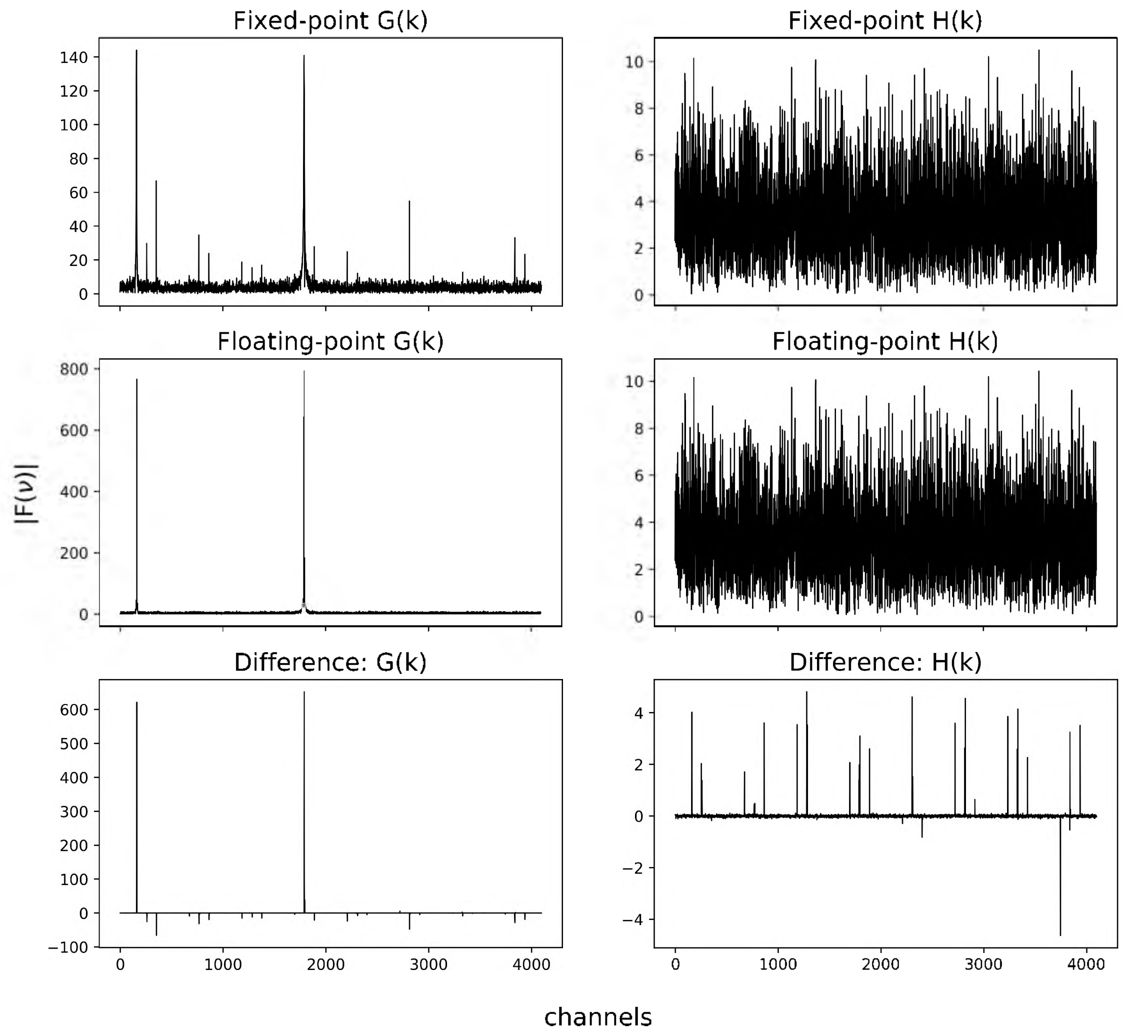

Figure 5.21: Dual polarisation processing of a real noisy two-tone vector of SNR≈ 30dB and an added imaginary noise vector. The last row indicates the difference |Float PFB| − |Fixed PFB| for both G(k) and H(k). Here the shift scheme 0b101010101010 is used with the intention of allowing the polarisation containing the tones to overflow.

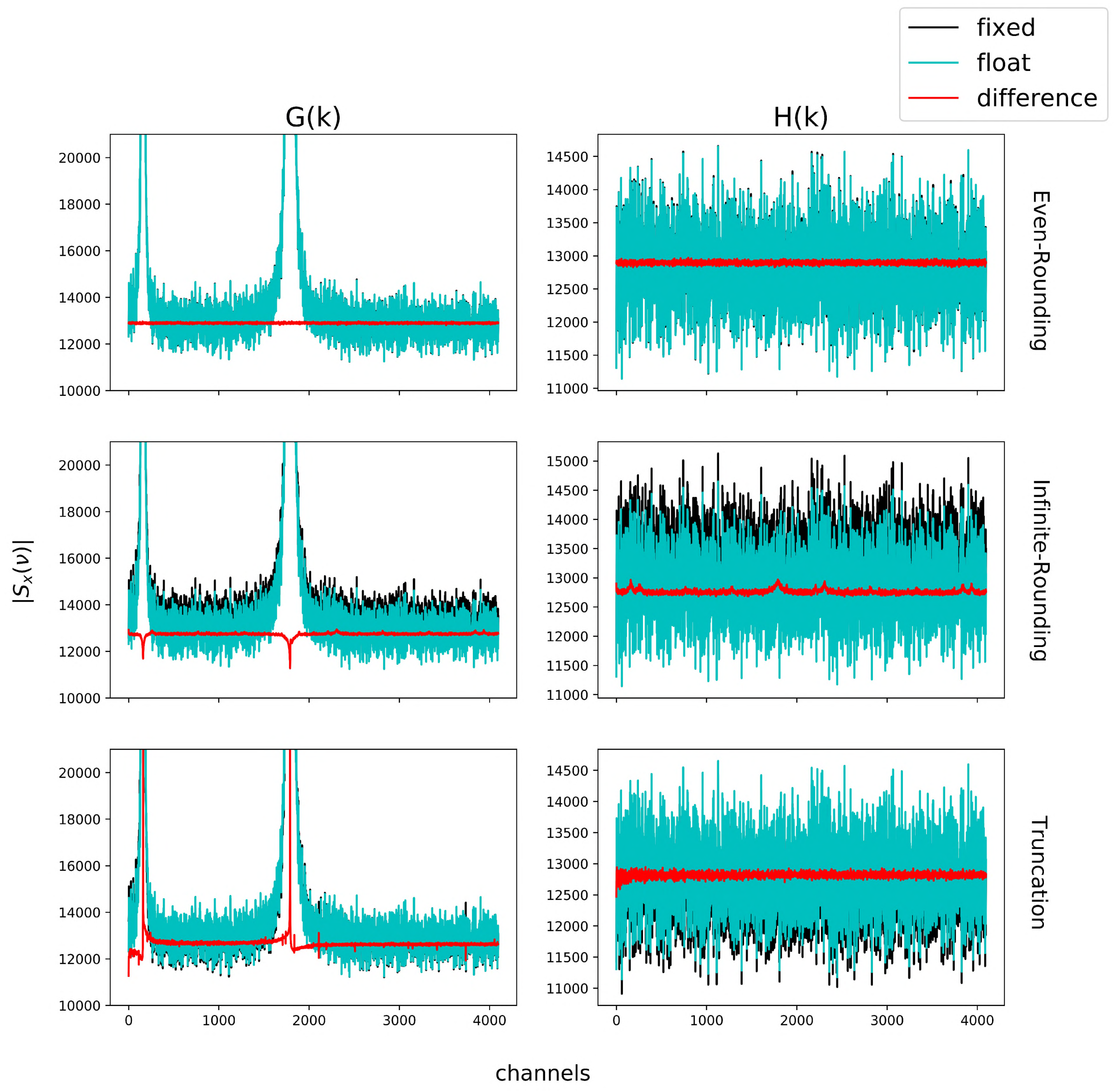

Figure 5.22: One hundred and fifty integrations of dual polarisation results. The input was a real two-tone vector of SNR≈ 30dB with an added imaginary noise vector. Every integration used a new noise vector (for both real and imaginary parts), thereby ensuring the noise was uncorrelated. The first row is the result for G(k) and H(k) when using even-rounding, the second row for infinite-rounding and the final row for when using truncation. The black plot is the floating point result, cyan the fixed point result and red the difference (float - fixed). Both G(k) and H(k) plots are vertically scaled to give a close-up of the noise floor.

Discussion

Figure 5.20 tests the effectiveness of dual polarisation plotting. Recalling that the real part of the signal contained two tones with added noise, whilst the imaginary part contained the noise vector alone, the result produced has accurately split the output transform into its respective transforms. Furthermore, it is to be noted that the quantisation noise for G(k) and H(k) are of the same level (note the absolute difference in the bottom row).

Following that, the results of figure 5.21 indicate a definite spectral leakage between the polarisations. Given that overflow could only occur due to the tones inputted in the real component, any presence of overflow in the fixed point H(k) spectrum must be a result of leakage from the G(k) polarisation. The absolute difference in the H(k) plot shows overflow remnants and so we may conclude that negligence of a single polarisation that overflows, can corrupt the results of the other polarisation when processing two polarisations simultaneously.

The final set of plots show just 150 integrations of the G(k) and H(k) spectra for a real uncorrelated noise input with two tones added to an imaginary uncorrelated noise input. This result was expected to be no different to the earlier rounding tests since both polarisations will use the same rounding when being processed. The even-rounding scheme notably performs well, the infinite-rounding scheme overshoots the floating point result and the truncation result has some serious quantisation error around the spikes. However, the small bumps around channel 150 and channel 1850 seem to indicate that the even-rounding H(k) result has rounding residue from its G(k) result. None of the other schemes appear to have rounding residue leakage, but future tests would integrate for much longer and perhaps reveal such effects.

CHAPTER 6

Conclusion

This thesis work confirmed the speculation that the spurious spectral features seen in the MeerKAT passband were as a result of the rounding scheme adopted. Specifically, since switching from infinite-rounding to even-rounding, the periodic dips that were present in the MeerKAT passband are gone.

This work further confirmed that the shift strategy adopted for the FFT is important in order to set the quantisation efficiency $\eta \approx 1$. It indicated, that while monochromatic tone signals require aggressive shifting, in the event of a white gaussian noise input, non-aggressive strategies should be used. As such, careful treatment of when to shift (i.e. apply all contiguously or alternate etc.) does effect the quantisation efficiency.

In the event that overflow does occur in the FFT, the built fixed point Python simulator provides insight into the effects of overflow, how it propagates and how it can leak between polarisations. It stresses the need to set the shift-register to accommodate the polarisation most likely to overflow, since the other polarisation will not be exempt in the event that it does.

In doing this research, a lot of work went into learning observational radio astronomy and instrumentation. It was not sufficient to just understand the F-Engine but rather, practical research into receivers, digitisers, delay tracking and cross-correlation was required. Furthermore, an understanding of FPGAs was paramount. A lot of work was first done on ROACH 2's (the FPGA board used for KAT7 - a MeerKAT precursor) and then on SKARAB's to understand how they are programmed, interacted with and used for scientific instrumentation. Thereafter, careful study of the CASPER toolflow was done in order to understand the relation between Xilinx and Simulink and how the DSP blocks were created. Given that the Python PFB simulators being developed for this research were only useful insofar as they mimicked the CASPER PFB sufficiently, careful scrutiny of the MeerKAT PFB in Simulink and onboard a SKARAB was done.

This research was very well received by the CASPER community at several conferences, since while it was focussed on studying the effects present in MeerKAT, it is applicable to any other radio instrument using the CASPER PFB. Not only are these results useful to other members of the collaboration, but the simulator is generically implemented to suit whatever fixed and floating point implementation of the CASPER PFB other members might be using. Furthermore, being developed in Python (a language used in the CASPER toolflow) means many members could understand the working of this simulator and perhaps extend its functionality.

Future research into finite precision arithmetic in the PFB would largely involve running the same tests as outlined in this thesis but for different scenarios. Rounding techniques should be tested for longer integrations, larger or smaller bit-widths (MeerKAT uses $\langle 22, 21\rangle$ fixed point numbers now), many tone inputs and pure gaussian noise to see how the quantisation noise differs. Further quantisation efficiency studies should be done for a variety of shift schemes and input signal levels. Finally, a better understanding of overflow in the FFT could be had if a more careful study of the individual stages when overflowing were made.

Research into the DSP behaviour of the PFB (though largely understood) is possible by using the floating point implementation to check for the effect of larger tap sizes, different window functions (this thesis only used Hann, but the simulator allows for others), larger FFTs and for coherent effects when accumulating a large number of outputs.

In conclusion, this work has aimed to document and study the CASPER PFB as it is used in MeerKAT. It is expected that the results, being the simulators produced and the insight into the finite precision effects present in the MeerKAT passband, will serve to educate and inform where necessary, thereby assisting in the building of more accurate radio telescopes.

APPENDIX A

Appendix

This appendix provides the four main Python scripts used for the simulation work done in this thesis. These are the fixpoint.py file that provides the fixed-point number system class required, the pfb_floating.py file that contains the floating-point PFB simulator, the pfb_fixed.py file that contains the fixed-point PFB simulator and pfb_coeff_gen.py file that contains a function used to generate the PFB FIR coefficients and necessary FIR scale factor for both PFB simulators.

All code is available on the github repository: `https://github.com/talonmyburgh/F-Engine_python_sim` where in addition, guides showing how to use the simulators are provided.

Listing A.1: fixpoint.py - The fixed-point Python number system class.

```
"""
Created on Tue May 29 13:45:20 2018
@author: talonmyburgh
"""
##############################IMPORTS#######################################
import numpy as np
import numba as nb
############################################################################

class fixpoint(object):
    """Takes number bits in full, number of fractional bits, minimum and
    maximum number representable, unsigned or signed integer, rounding method
    and overflow method"""
    def __init__(self,bits,fraction, min_int=None, max_int=None,
                 unsigned=False, method = "ROUND"):
        self.method = method
        self.range = 2 ** bits                                      #The
            dynamic range of the number
        self.scale = 2 ** fraction                                  #The
            fractional dynamic range by which the number will be scaled
        self.unsigned = unsigned
        self.__setbnds__(min_int,max_int)                           #Sets
            self.min and self.max of number
        self.data = None

    def __setbnds__(self, min_int=None, max_int=None):
        if min_int is None:                                         # decides
            minimal value
            self.min = 0 if self.unsigned else - self.range // 2
        else:
            self.min = min_int
        if max_int is None:                                         #decides
            maximum value
            self.max = self.range - 1 if self.unsigned else self.range // 2 - 1
        else:
            self.max = max_int

    @property
    def bits(self):                                                 #bits
        property
        return int(np.log2(self.range))

    @bits.setter
```

```
def bits(self,val):
    if(type(val)!=int):
        raise ValueError("'bits' argument must be of type integer")
    else:
        self.range = 2 ** val
        self.__setbnds__()

@property
def fraction(self):                                         #frac
    property
    return int(np.log2(self.scale))

@fraction.setter
def fraction(self,val):
    if(type(val)!=int):
        raise ValueError("'frac' argument must be of type integer")
    else:
        self.scale = 2**val

@property
def unsigned(self):                                         #unsigned
    as property
    return self.min == 0

@unsigned.setter
def unsigned(self,val):
    if(type(val)!=bool):
        raise ValueError("'unsigned' argument must be of type bool")
    else:
        self.min = 0 if val else - self.range // 2
        self.max = self.range - 1 if val else self.range // 2 - 1

@property                                                   #64bit int
    for signed and 64bit uint for unsigned
def FPTYPE(self):
    if(self.unsigned):
        return np.uint64
    else:
        return np.int64

def __repr__(self):                                         #how things
    will be shown when using 'print'
    return 'FP real %s (%d, %d), shape %s' % \
        ('unsigned' if self.unsigned else 'signed',
        self.bits, self.fraction, np.shape(self.data))

def __getitem__(self,key):                                 #method of
    slicing fixpoint arrays
    newfpt = fixpoint(self.bits,self.fraction,unsigned=self.unsigned,
                    method = self.method)
    newfpt.data = self.data.copy()[key]
    return newfpt

def __setitem__(self,key,val):                             #method for
    populating slices of arrays
    self.data[key] = val.data.copy()

def normalise(self):                                       #how to fit
    all data values within the min/max specified
    self.data = np.clip(self.data, self.min, self.max)

def from_float(self, x):                                   #take in
    float values                         #detect overflow method used
    if(self.method =="ROUND"):                             #if we're
        rounding off decimal values bankers style
        self.data = np.clip(np.round(x*self.scale).astype(self.FPTYPE),
                        self.min, self.max)
    elif(self.method =="TRUNCATE"):                        #if we're
        truncating off decimal
        self.data = np.clip(np.trunc(x*self.scale).astype(self.FPTYPE),
                        self.min, self.max)
    elif(self.method == "ROUND_INFTY"):                    #round to
        decimal as round up - much slower but only option now.
        self.data =
            np.clip(self.__roundinfty__(x*self.scale).astype(self.FPTYPE),
                        self.min,self.max)
    else:
        raise ValueError("No recognisable quantisation method specified")

def to_float(self): #for plotting etc
    return (self.data.astype(self.FPTYPE)) / self.scale

def sum(self, *args, **kwargs):                            #rewrite
    the sum method
    res = self.data.sum(*args, **kwargs)                   #use numpy
        sum method
    bits = self.bits + int(np.ceil(np.log2(self.data.size / res.size)))
    result = fixpoint(bits, self.fraction, unsigned=self.unsigned,
                    method = self.method)
    result.data = res
    result.normalise()                                              #clip and
        stuff
    return result

def __mul__(self, w):
    res = self.data * w.data
    result = fixpoint(self.bits + w.bits,
                           self.fraction + w.fraction,
                           unsigned=self.unsigned and w.unsigned,
                           method = self.method)
    result.data = res
    result.normalise()
    return result

def __add__(self, y):
    if(self.scale!=y.scale):
        raise ValueError("Addition performed between two numbers of differing
            scales!")

    res = self.data + y.data
    #adds together, and accounts for carry bit
    result = fixpoint(max(self.bits, y.bits) + 1,
                           max(self.fraction, y.fraction),
                           unsigned=self.unsigned and y.unsigned,
                           method = self.method)
    result.data = res
    result.normalise()
    return result

def __sub__(self, y):
    if(self.scale>y.scale or self.scale<y.scale):
        raise ValueError("Subtraction performed between two numbers of differing
            scales!")

    res = self.data - y.data
    #subtracts together, and accounts for carry bit
    result = fixpoint(max(self.bits, y.bits) + 1,
                           max(self.fraction, y.fraction),
                           unsigned=self.unsigned and y.unsigned,
                           method = self.method)
    result.data = res
    result.normalise()
    return result

def quantise(self, bits, fraction, min_int=None, max_int=None, unsigned=False,
            method="ROUND"):
    result = fixpoint(bits, fraction, min_int, max_int, unsigned,
                      method = method)
    result.from_float(self.to_float())
    return result

def __rshift__(self,steps):                                        #slicing
    and right shifting technique - allows for rounding
    if(self.method == "ROUND"):
        self.data = np.round(self.data/(2**steps)).astype(self.FPTYPE)
    elif(self.method =="ROUND_INFTY"):
        self.data = self.__roundinfty__(self.data/(2**steps)).astype(self.FPTYPE)
    elif(self.method=="TRUNCATE"):
        self.data >>= steps
    else:
        raise ValueError("No recognisable quantisation method specified")
    return self

def __lshift__(self,steps):
    self.data <<= steps
    return self

def copy(self):                                                    #method for
    making a copy of fixpoint type (else get referencing issues)
    tmpfxpt=fixpoint(self.bits,self.fraction,unsigned=self.unsigned,
                     method = self.method,
                     min_int = self.min, max_int = self.max)
    tmpfxpt.data = self.data.copy()
    return tmpfxpt

def power(self):
    return self.data*self.data

"""This method rounds values in an array to +/- infinity"""

@nb.jit
def __roundinfty__(self,array):
    a = array.copy()
    f=np.modf(a)[0]                                                #get
        decimal values from data
    if (a.ndim == 1):                                              #for 1D
```

```
            array
            for i in range(len(array)):
                if((f[i]<0.0 and f[i] <=-0.5) or (f[i]>=0.0 and f[i]<0.5)):
                    a[i]=np.floor(a[i])
                else:
                    a[i]=np.ceil(a[i])
        elif(a.ndim==2):                                                      #for 2D
            array
            for i in range(array.shape[0]):
                for j in range(array.shape[1]):
                    if((f[i,j]<0.0 and f[i,j] <=-0.5) or (f[i,j]>=0.0
                       and f[i,j]<0.5)):
                        a[i,j]=np.floor(a[i,j])
                    else:
                        a[i,j]=np.ceil(a[i,j])
        elif(a.ndim==3):                                                      #for 3D
            array
            for i in range(array.shape[0]):
                for j in range(array.shape[1]):
                    for k in range(array.shape[2]):
                        if((f[i,j,k]<0.0 and f[i,j,k] <=-0.5) or
                           (f[i,j,k]>=0.0 and f[i,j,k]<0.5)):
                            a[i,j,k]=np.floor(a[i,j,k])
                        else:
                            a[i,j,k]=np.ceil(a[i,j,k])
        return a

    __str__ = __repr__                                                        #redundancy
        for print

    """Fixed-point container for complex values which makes use of existing
    fixpoint. Additional parameters here are to specify two fixpoint numbers as
    real and imag, by which cfixpoint will extract all other parameters."""
class cfixpoint(object):

    def __init__(self, bits=None, fraction=None, min_int=None, max_int=None,
                unsigned=False, method = "ROUND", real=None, imag=None):

        if bits is not None:                                                  #if bits
            are supplied (i.e not real and imag)
            self.real = fixpoint(bits, fraction, min_int, max_int, unsigned,
                #declare a real and imag fixpoint
                                method)
            self.imag = fixpoint(bits, fraction, min_int, max_int, unsigned,
                                method)
        elif real is not None:                                              #else use
            real and imag fixpoint supplied
            self.real = real
            self.imag = imag
        else:
            raise ValueError("Must either specify bits/fraction or pass two fixpoint
                numbers to real/imag.")

    @property                                                               #bits
        property
    def bits(self):
        return int(np.log2(self.real.range))

    @bits.setter
    def bits(self,val):
        self.real.bits = val
        self.imag.bits = val

    @property                                                               #fraction
        property
    def fraction(self):
        return int(np.log2(self.real.scale))

    @fraction.setter
    def fraction(self,val):
        self.real.fraction = val
        self.imag.fraction = val

    @property                                                               #range
    def range(self):
        return self.real.range

    @property                                                               #scale
    def scale(self):
        return self.real.scale

    @property                                                               #unsigned
        property
    def unsigned(self):
        return self.real.min == 0

    @unsigned.setter
    def unsigned(self,val):
        self.real.unsigned=val
        self.imag.unsigned=val

    @property                                                    #min
        property
    def min(self):
        return self.real.min + 1j * self.imag.min

    @property
    def max(self):                                               #max
        property
        return self.real.max+ 1j * self.imag.max

    @property
    def data(self):                                              #data held
        in cfixpoint (will be integer)
        return self.real.data + 1j * self.imag.data

    @property
    def method(self):                                            #rounding
        method in use
        return self.real.method

    @method.setter
    def method(self,val):
        self.real.method=val
        self.imag.method=val

    def __repr__(self):                                          #printing
        return 'FP complex %s (%d, %d), shape %s' % \
               ('unsigned' if self.unsigned else 'signed',
                self.bits, self.fraction, np.shape(self.real.data))

    def __getitem__(self,key):                                   #returning
        slices of array
        tmpcfpt = cfixpoint(real = self.real[key],imag = self.imag[key])
        return tmpcfpt

    def __setitem__(self,key,val):                               #setting
        slices of array
        self.real[key] = val.real
        self.imag[key] = val.imag
```

```
def from_complex(self, x):                                          #accepts
    complex array and populates to data with scaling
    self.real.from_float(x.real)
    self.imag.from_float(x.imag)

def to_complex(self):                                               #converts
    data to complex array
    return self.real.to_float() + 1j * self.imag.to_float()

def sum(self, *args, **kwargs):
    result = cfixpoint(real=self.real.sum(*args, **kwargs),
                            imag=self.imag.sum(*args, **kwargs))
    return result

def __mul__(self, w):                                               #complex
    multiplication
    def complex_mult(a, b, c, d):
        """Returns complex product x + jy = (a + jb) * (c + jd)."""
        # Real part x = a*c - b*d
        x = (a*c)-(b*d)
        # Imaginary part y = a*d + b*c
        y = (a*d)+(b*c)
        return x, y
    out_real, out_imag = complex_mult(self.real, self.imag, w.real, w.imag)
    result = cfixpoint(real=out_real, imag=out_imag)
    return result

def __add__(self, y):                                               #complex
    addition
    result = cfixpoint(real=self.real+y.real,
                            imag=self.imag+y.imag)
    return result

def __sub__(self, y):                                               #complex
    subtraction
    result = cfixpoint(real=self.real-y.real,
                            imag=self.imag-y.imag)
    return result

def normalise(self):                                                #normalise
    the real and imag data
    self.real.normalise()
    self.imag.normalise()
#quantise the data to bounds required.
def quantise(self, bits, fraction, min_int=None, max_int=None,
             unsigned=False, method="ROUND"):
    out_real = self.real.quantise(bits, fraction, min_int, max_int,
                                  unsigned, method)
    out_imag = self.imag.quantise(bits, fraction, min_int, max_int,
                                  unsigned, method)
    result = cfixpoint(real=out_real, imag=out_imag)
    return result

def __rshift__(self,steps):                                 #right
    shift data by steps
    self.real >> steps
    self.imag >> steps
    return self

def __lshift__(self,steps):                                 #left shift
    data by steps
    self.real << steps
    self.imag << steps
    return self

def copy(self):                                             #method for
    making a copy of cfixpoint type
    tmpcfxpt = cfixpoint(real = self.real.copy(),imag=self.imag.copy())
    return tmpcfxpt

def conj(self):                                             #returns
    conjugate of cfixpoint
    i_res = self.imag.copy()
    i_res.data = -self.imag.data.copy()
    res = cfixpoint(real=self.real,imag=i_res)
    return res

def power(self):                                            #return
    power as a x a* of cfixpoint
    res = self.copy() * self.conj()
    return res.real

__str__ = __repr__
```

Listing A.2: pfb_floating.py - The floating-point Python PFB simulator class.

```
"""
Created on Thu Aug 16 15:21:43 2018
@author: talonmyburgh
"""
import numpy as np
from pfb_coeff_gen import coeff_gen

# ==============================================================================
# Bit reversal algorithms used for the iterative fft's data re-ordering
# ==============================================================================
"""Arranges chronological values in an array in a bit reversed fashion"""
def bit_rev(a, bits):
    a_copy = a.copy()
    N = 1<<bits
    for i in range(1,bits):
        a >>=1
        a_copy <<=1
        a_copy |= (a[:]&1)
    a_copy[:] &= N-1
    return a_copy

"""Takes an array of length N which must be a power of two"""
def bitrevarray(array,N):
    bits = int(np.log2(N))                                            #number of
       bits to repr numbers in array
    A = np.empty(N,dtype=np.complex64)
    a=np.arange(N)
    A[bit_rev(a,bits)] = array[:]
    return A

# ==============================================================================
# FFT: natural data order in, bit reversed twiddle factors, bit reversed
# order out.
# ==============================================================================
"""Generate array of needed twiddles"""
def make_twiddle(N):
    i=np.arange(N//2)
    arr = np.exp(-2*i*np.pi*1j/N)
    return arr

"""Natural order in DIT FFT that accepts the data, the twiddle factors
(bit reversed) and allows for staging"""
def iterfft_natural_in_DIT(DATA,twid,staged=False):
```

```
data = np.asarray(DATA,dtype = np.complex64)
N = data.shape[0]                                                    #how long
    is data stream

if(staged):
    stgd_data = np.zeros((N,int(np.log2(N))+2),dtype = np.complex64)
    stgd_data[:,0] = data[:]
num_of_groups = 1                                                    #number of
    groups - how many subarrays are there?
distance = N//2                                                      #how far
    between each fft arm?
stg=1                                                                #stage
    counter

while num_of_groups < N:                                             #basically
    iterates through stages
    for k in range(num_of_groups):                                   #iterate
        through each subarray
        jfirst = 2*k*distance                                        #index to
            beginning of a group
        jlast = jfirst + distance - 1                                #first
            index plus offset - used to index whole group
        W=twid[k]
        slc1 = slice(jfirst,jlast+1,1)
        slc2 = slice(jfirst+distance, jlast+1+distance,1)
        tmp = W*data[slc2]
        data[slc2] = data[slc1]-tmp
        data[slc1] = data[slc1]+tmp
    num_of_groups *=2
    distance //=2
    if(staged):                                                      #if we are
        recording stages
        stgd_data[:,stg]=data[:]                                     #log each
            stage data to array
    stg+=1

if(staged):
    stgd_data[:,-1] = bitrevarray(stgd_data[:,-2],N)                 #post
        bit-reordering for last stage - added as extra stage
    return stgd_data
else:
    A=bitrevarray(data,N)                                            #post
        bit-reordering
    return A
```

```
# ==============================================================================
# Floating point PFB implementation making use of the natural order in fft
# like CASPER does.
# ==============================================================================

class FloatPFB(object):
       """This function takes point size, how many taps, what percentage of total
          data to average over,
       to get data from a file or not,what windowing function, whether you're
          running dual polarisations,
       whether you'd like data from a stage, and if so which stage - stage 0 being
          the data in"""
       def __init__(self, N, taps, datasrc = None, w = 'hann',dual = False,
                    staged = False, fwidth=1, chan_acc = False):
           self.N = N                                                   #how many
              points                                               #what averaging
           self.dual = dual                                             #whether
              you're performing dual polarisations or not
           self.reg =np.zeros([N,taps])                                 #our fir
              register size filled with zeros orignally
           self.inputdatadir = None
           self.staged=staged
           self.fwidth = fwidth
           self.chan_acc = chan_acc

           if(datasrc is not None and type(datasrc)==str):              #if input
              data file is specified
              self.inputdatadir = datasrc
              self.outputdatadir = datasrc[:-4]+"out.npy"
              self.inputdata = np.load(datasrc, mmap_mode = 'r')
           else:
              self.inputdata = None

           self.window = coeff_gen(N,taps,w,self.fwidth)[0]             #Get window
              coefficients and scaling
                                                                        #factor to
                                                                           use in
                                                                           FIR
                                                                           registers.
           self.twids = make_twiddle(self.N)
           self.twids = bitrevarray(self.twids, len(self.twids))        #for
              natural order in FFT
```

```
    """Takes data segment (N long) and appends each value to each fir.
    Returns data segment (N long) that is the sum of fircontents*windowcoeffs"""
    def _FIR(self,x):
        self.reg = np.column_stack((x,self.reg))[:,:-1]            #push and
            pop from FIR register array
        X = np.sum(self.reg*self.window,axis=1)                    #filter and
            scale
        return X

    """For dual polarisation processing, we need to split the data after
    FFT and return the individual complex spectra"""
    def _split(self,Y_k):
        R_k = np.real(Y_k)
        R_kflip = R_k.copy()
        R_kflip[1:] = R_kflip[:0:-1]

        I_k = np.imag(Y_k)
        I_kflip = I_k.copy()
        I_kflip[1:] = I_kflip[:0:-1]

        self.G_k = (1/2)*(R_k+1j*I_k+R_kflip-1j*I_kflip)           #declares
            two variables for 2 pols
        self.H_k = (1/2j)*(R_k+1j*I_k-R_kflip+1j*I_kflip)


    """Here we take the power spectrum of the outputs. Chan_acc dictates
    if one must sum over all outputs produced."""
    def _pow(self,X):
        if (self.chan_acc):                                        #if
            accumulation specified
            pwr = X * np.conj(X)
            pwr = np.real(np.sum(pwr,axis=1))
            return pwr
        else:                                                      #if no
            accumulation specified
            pwr = np.real((X * np.conj(X)))
            return pwr

    """Here one parses a data vector to the PFB to run. Note it must be
    numpy array of length N if a data file was not specified before"""
    def run(self,data=None):

        if (data is not None):                                     #if we are
            using an input data array
            self.inputdata = data
        elif(self.inputdata is None):
            raise ValueError ("No input data for PFB specified.")

        size = self.inputdata.size                                  #get length
            of data stream
        stages = size//self.N                                       #how many
            cycles of commutator

        if(self.staged):                                            #if storing
            staged data
            X = np.empty((self.N,stages,int(np.log2(self.N))+2),
                        dtype = np.complex64)
                                                                    #will be
                                                                        tapsize
                                                                        x
                                                                        datalen/point
                                                                        x stages
            for i in range(0,stages):                               #for each
                stage, populate all firs, and run FFT once
                if(i ==0):
                    X[:,i,:] = iterfft_natural_in_DIT(self._FIR(
                            self.inputdata[0:self.N]),self.twids,
                        self.staged)
                else:
                    X[:,i,:] = iterfft_natural_in_DIT(self._FIR(
                            self.inputdata[i*self.N:i*self.N+self.N]),self.twids,
                        self.staged)

        else:                                                       #if storing
            staged data
            X = np.empty((self.N,stages),dtype = np.complex64)
                                                                    #will be
                                                                        tapsize
                                                                        x stages
            for i in range(0,stages):                               #for each
                stage, populate all firs, and run FFT once
                if(i == 0):
                    X[:,i] = iterfft_natural_in_DIT(self._FIR(
                            self.inputdata[0:self.N]),
                        self.twids)
                else:
                    X[:,i] = iterfft_natural_in_DIT(self._FIR(
                            self.inputdata[i*self.N:i*self.N+self.N]),
                        self.twids)

        """Decide on how to manipulate and display output data"""
        if(self.dual and not self.staged):                          #If dual
            processing but not staged
            self._split(X)
            self.G_k_pow = self._pow(self.G_k)
            self.H_k_pow = self._pow(self.H_k)

        elif(not self.dual and self.staged):                        #If single
            pol processing and staged
            self.X_k_stgd = X
            self.X_k_pow = self._pow(X[:,:,-1])
            self.X_k = X[:,:,-1]

        elif(self.dual and self.staged):                            #If dual
            pol and staged
            self.X_k_stgd = X
            self._split(X[:,:,-1])
            self.G_k_pow = self._pow(self.G_k)
            self.H_k_pow = self._pow(self.H_k)

        else:                                                       #If single
            pol and no staging
            self.X_k = X
            self.X_k_pow = self._pow(X)
```

Listing A.3: pfb_fixed.py - The fixed-point Python PFB simulator class.

```
"""
Created on Thu Aug 16 16:13:40 2018
@author: talonmyburgh
"""
import numpy as np
from fixpoint import fixpoint, cfixpoint
from pfb_coeff_gen import coeff_gen

# =============================================================================
# Bit reversal algorithms used for the iterative fft's
# =============================================================================
"""Arranges chronological values in an array in a bit reversed fashion"""
def bit_rev(a, bits):
    a_copy = a.copy()
    N = 1<<bits
    for i in range(1,bits):
        a >>=1
        a_copy <<=1
        a_copy |= (a[:]&1)
    a_copy[:] &= N-1
    return a_copy

"""Takes an array of length N which must be a power of two"""
def bitrevfixarray(array,N):                                          #takes an
    array of length N which must be a power of two
    bits = int(np.log2(N))                                            #how many
       bits it takes to represent all numbers in array
    A = array.copy()
    a = np.arange(N)
    A[bit_rev(a,bits)] = array[:]
    return A

# =============================================================================
# FFT: natural data order in, bit reversed twiddle factors, bit reversed
# order out.
# =============================================================================
def make_fix_twiddle(N,bits,fraction,method="ROUND"):
    twids = cfixpoint(bits,fraction, method = method)
    twids.from_complex(np.exp(-2*np.arange(N//2)*np.pi*1j/N))
    return twids

"""Natural order in DIT FFT that accepts the data, the twiddle factors
(must be bit reversed), a shift register, the bitwidth and fraction
```

```
bit width to process at, the twiddle factor bits and allows for staging"""
def iterffft_natural_DIT(DATA,twid,swreg,bits,fraction,twidfrac,staged=False):

    data=DATA.copy()
    N = data.data.shape[0]                                        #how long
       is data stream

    if(type(swreg)==int):                                 #if integer is parsed
       rather than list
       shiftreg = [int(x) for x in bin(swreg)[2:]]
       if (len(shiftreg)<int(np.log2(N))):
           for i in range(int(np.log2(N))-len(shiftreg)):
               shiftreg.insert(0,0)
    elif(type(swreg)==list and type(swreg[0])==int):      #if list of integers is
       parsed
       shiftreg = swreg
    else:
       raise ValueError('Shiftregister must be type int or binary list of ints')

    if(staged):
       stgd_data = DATA.copy()
       stgd_data.from_complex(np.zeros((N,int(np.log2(N))+2),
                                   dtype = np.complex64))
       stgd_data[:,0] = data[:]
    stages = int(np.log2(N))
    if(len(shiftreg)!=stages and type(shiftreg) is not list):
       raise ValueError("Shift register must be of type list, and its length "
                    +"must be log2(data length)")

    num_of_groups = 1                                             #number of
       groups - how many subarrays are there?
    distance = N//2                                               #how far
       between each fft arm?
    stg=1                                                         #stage
       counter
    while num_of_groups < N:                                      #basically
       iterates through stages
       for k in range(num_of_groups):                             #iterate
           through each subarray
           jfirst = 2*k*distance                                  #index to
               beginning of a group
           jlast = jfirst + distance - 1                          #first
               index plus used to index whole group
           W=twid[k]
```

```
            slc1 = slice(jfirst,jlast+1,1)
            slc2 = slice(jfirst+distance, jlast+1+distance,1)
            tmp = W * data[slc2]
            tmp >> twidfrac                                        #slice off
               lower bit growth from multiply (caused by fraction only)
            tmp.bits =bits
            tmp.fraction=fraction                                  #fraction
               will = (frac1+frac2) - hence right shift by frac2
            tmp.normalise()
            data[slc2] = data[slc1]-tmp
            data[slc1] = data[slc1]+tmp

        if shiftreg.pop():                                         #implement
            FFT shift and then normalise to correct at end of stage
            data>>1
        data.normalise()

        num_of_groups *=2
        distance //=2
        if(staged):                                                #if we are
            recording stages
            stgd_data[:,stg]=data[:]                               #log each
               stage data to array
        stg+=1

    if(staged):
        stgd_data[:,-1] = bitrevfixarray(stgd_data[:,-2],N)        #post
            bit-reordering for last stage - added as extra stage
        return stgd_data
    else:
        return bitrevfixarray(data,N)                              #post
            bit-reordering

# ================================================================================
# Floating point PFB implementation making use of the natural order in fft
# like CASPER does.
# ================================================================================

class FixPFB(object):
        """This function takes point size, how many taps, whether to integrate
        the output or not, what windowing function to use, whether you're
        running dual polarisations, what rounding and overflow scheme to use,
        fwidth and whether to stage."""
        def __init__(self, N, taps, bits, frac,
                     twidbits, twidfrac, swreg,
                     bitsofacc=32, fracofacc=31, unsigned = False,
                     chan_acc =False, datasrc = None, w = 'hann',
                     firmethod="ROUND", fftmethod="ROUND", dual = False,
                     fwidth=1, staged = False):

            """Populate PFB object properties"""
            self.N = N                                                    #how many
               points
            self.chan_acc = chan_acc                                      #if summing
               outputs
            self.dual = dual                                              #whether
               you're processing dual polarisations
            self.taps = taps                                              #how many
               taps
            self.bitsofacc = bitsofacc                                    #how many
               bits to grow to in integration
            self.fracofacc = fracofacc
            self.bits = bits                                       #fft data bitlength
            self.frac = frac
            self.fwidth = fwidth
               #normalising factor for fir window
            if(type(swreg)==int):                                         #if integer
               is parsed rather than list
               self.shiftreg = [int(x) for x in bin(swreg)[2:]]
               if (len(self.shiftreg)<int(np.log2(N))):
                   for i in range(int(np.log2(N))-len(self.shiftreg)):
                       self.shiftreg.insert(0,0)
            elif(type(swreg)==list and type(swreg[0])==int):          #if list of
               integers is parsed
               self.shiftreg = swreg
            else:
               raise ValueError('Shiftregister must be type int or binary list of
                   ints')

            self.unsigned = unsigned                                      #only used
               if data parsed in is in a file
            self.staged = staged                                          #whether to
               record fft stages
            self.twidbits = twidbits                                      #how many
               bits to give twiddle factors
            self.twidfrac = twidfrac
            self.firmethod=firmethod                                      #rounding
```

```
            scheme in firs
        self.fftmethod=fftmethod                                    #rounding
            scheme in fft

        #Define variables to be used:
        self.reg_real = fixpoint(self.bits, self.frac,unsigned = self.unsigned,
                                 method = self.firmethod)
        self.reg_real.from_float(np.zeros([N,taps],dtype = np.int64)) #our fir
            register size filled with zeros orignally
        self.reg_imag = self.reg_real.copy()

        if(datasrc is not None and type(datasrc)==str):             #if input
            data file is specified
            self.inputdata = cfixpoint(self.bits, self.frac,unsigned =
                self.unsigned,
                       method = self.firmethod)
            self.inputdatadir = datasrc
            self.outputdatadir = datasrc[:-4]+"out.npy"
            self.inputdata.from_complex(np.load(datasrc, mmap_mode = 'r'))
        else:
            self.inputdatadir = None

        #the window coefficients for the fir filter
        self.window = fixpoint(self.bits, self.frac,unsigned = self.unsigned,
                               method = self.firmethod)
        tmpcoeff,self.firsc = coeff_gen(self.N,self.taps,w,self.fwidth)
        self.window.from_float(tmpcoeff)

        #the twiddle factors for the natural input fft
        self.twids = make_fix_twiddle(self.N,self.twidbits,twidfrac,
                                      method=self.fftmethod)
        self.twids = bitrevfixarray(self.twids,self.twids.data.size)

    """Takes data segment (N long) and appends each value to each fir.
    Returns data segment (N long) that is the sum of fircontents*window"""
    def _FIR(self,x):
        #push and pop from FIR register array
        self.reg_real.data = np.column_stack(
                (x.real.data,self.reg_real.data))[:,:-1]
        self.reg_imag.data = np.column_stack(
                (x.imag.data,self.reg_imag.data))[:,:-1]

        X_real = self.reg_real*self.window                          #compute
            real and imag products
        X_imag = self.reg_imag*self.window
        prodgrth = X_real.fraction - self.frac                      #-1 since
           the window coeffs have -1 less fraction
        X = cfixpoint(real = X_real.sum(axis=1),imag = X_imag.sum(axis =1))
        X >> prodgrth +self.firsc                                   #remove
           growth
        X.bits = self.bits                                          #normalise
           to correct bit and frac length
        X.fraction = self.frac
        X.normalise()
        X.method = self.fftmethod                                   #adjust so
           that it now uses FFT rounding scheme

        return X                                                    #FIR output

    """In the event that that dual polarisations have been selected, we need to
    split out the data after and return the individual X_k values"""
    def _split(self,Yk):
        #reverse the arrays for the splitting function correctly
        R_k = Yk.real.copy()
        I_k = Yk.imag.copy()

        R_kflip = R_k.copy()
        R_kflip[1:] = R_kflip[:0:-1]

        I_kflip = I_k.copy()
        I_kflip[1:] = I_kflip[:0:-1]

        self.G_k = cfixpoint(real = R_k + R_kflip, imag = I_k - I_kflip)
           #declares two variables for 2 pols
        self.G_k >> 1                                               #for bit
           growth from addition
        self.G_k.bits = self.bits
        self.G_k.normalise()

        self.H_k =cfixpoint(real = I_k + I_kflip, imag = R_kflip - R_k)
        self.H_k >> 1
        self.H_k.bits = self.bits
        self.H_k.normalise()


    """Here we take the power spectrum of the outputs. Chan_acc dictates
    if one must sum over all outputs produced."""
    def _pow(self,X):
        if (self.chan_acc):                                         #if
           accumulation specified
           tmp = X.power()                                         # X times X*
           pwr = X.copy()
           pwr.bits = self.bitsofacc
           pwr.frac=self.fracofacc
           pwr.normalise()                                         #normalise
               multiplication
           pwr.data = np.sum(tmp.data,axis=1)                      #accumulate
           return pwr
        else:                                                       #if no
           accumulation specified
           pwr = X.power()
           pwr.bits = self.bitsofacc
           pwr.frac=self.fracofacc
           pwr.normalise()                                         #normalise
               multiplication
           return pwr

    """Here one parses a data vector to the PFB to run. Note it must be
    cfixpoint type if a data file was not specified before"""
    def run(self,DATA, cont = False):

        if (DATA is not None):                                      #if a data
           vector has been parsed
           if(self.bits != DATA.bits):
               raise ValueError("Input data must match precision specified"
                               +"for input data with bits")
           self.inputdata = DATA
        elif(self.inputdata is None):                               #if no data
           was specified at all
           raise ValueError ("No input data for PFB specified.")

        size = self.inputdata.data.shape[0]                         #get length
           of data stream which should be multiple of N
        data_iter = size//self.N                                      #how many
           cycles of commutator

        X = cfixpoint(self.bits, self.frac,unsigned = self.unsigned,
                    method = self.fftmethod)

        if(self.staged):                                            #if all
           stages need be stored
           X.from_complex(np.empty((self.N,data_iter,int(np.log2(self.N))+2),
                               dtype = np.complex64))           #will be
                                  tapsize x datalen/point x fft stages +2
                                                                #(input and
                                                                   re-ordererd
                                                                   output)
           for i in range(0,data_iter):                         #for each
               data_iter, populate all firs, and run FFT once
               if(i == 0):
                    X[:,i,:] =
                        iterffft_natural_DIT(self._FIR(self.inputdata[0:self.N]),
                    self.twids,self.shiftreg.copy(),self.bits_,self.frac,
                    self.twidfrac,self.staged)
               else:
                    X[:,i,:] = iterffft_natural_DIT(
                            self._FIR(self.inputdata[i*self.N:i*self.N+self.N]),
                            self.twids,self.shiftreg.copy(),self.bits,
                            self.frac, self.twidfrac, self.staged)

        else:                                                    #if stages
           don't need to be stored
           X.from_complex(np.empty((self.N,data_iter),
                               dtype = np.complex64))           #will be
                                  tapsize x datalen/point
           for i in range(0,data_iter):                         #for each
               stage, populate all firs, and run FFT once
               if(i == 0):
                    X[:,i] =
                        iterffft_natural_DIT(self._FIR(self.inputdata[0:self.N]),
                    self.twids,self.shiftreg.copy(),self.bits,self.frac,
                    self.twidfrac, self.staged)

               else:
                    X[:,i] = iterffft_natural_DIT(
                            self._FIR(self.inputdata[i*self.N:i*self.N+self.N]),
                            self.twids,self.shiftreg.copy(),self.bits,
                            self.frac, self.twidfrac, self.staged)

#
        """Decide on how to manipulate and display output data"""
        if(self.dual and not self.staged):                       #If dual
           processing but not staged
           self._split(X)
           self.G_k_pow = self._pow(self.G_k)
           self.H_k_pow = self._pow(self.H_k)
        elif(not self.dual and self.staged):                        #If single
           pol processing and staged
           self.X_k_stgd = X
           self.X_k_pow = self._pow(X[:,:,-1])
           self.X_k = X[:,:,-1]

        elif(self.dual and self.staged):                            #If dual
           pol and staged
           self.X_k_stgd = X
           self._split(X[:,:,-1])
           self.G_k_pow = self._pow(self.G_k)
           self.H_k_pow = self._pow(self.H_k)

        else:                                                       #If single
           pol and no staging
           self.X_k = X
           self.X_k_pow = self._pow(X)
```

Listing A.4: pfb_coeff_gen.py - The FIR coefficient and FIR scale generator.

```
"""
Created on Tue Sep 3 11:41:01 2019
@author: talonmyburgh
"""
import numpy as np

def coeff_gen (N, taps, w='hann', fwidth=1):
    WinDic = {                                                    #dictionary
        of various filter types
    'hann' : np.hanning,
    'hamming' : np.hamming,
    'bartlett': np.bartlett,
    'blackman': np.blackman,
    }
    alltaps = N*taps
    windowval=WinDic[w](alltaps)
    totalcoeffs = (windowval*np.sinc(fwidth*(np.arange(alltaps)/(N) -
        taps/2))).reshape((taps,N)).T
    scalefac = nextpow2(np.max(np.sum(np.abs(totalcoeffs),axis=1)))
    return totalcoeffs ,int(scalefac)

def nextpow2(val):
    i=0
    while(True):
       if(2**i>=val):
           return i
       else:
           i+=1
```

Bibliography

[1] D. C. Price, *Spectrometers and polyphase filterbanks in radio astronomy*, *arXiv preprint arXiv:1607.03579* (2016).

[2] J. Jonas, *Fundamentals of radiation and radio sources*, Department of Phyics and Electronics, Rhodes University, 2017.

[3] J. J. Condon and S. M. Ransom, "Radio telescopes and radiometers." `https://www.cv.nrao.edu/~sransom/web/Ch3.html`. Accessed: 31/01/2019.

[4] A. R. Thompson, J. M. Moran, G. W. Swenson, *et. al.*, *Interferometry and synthesis in radio astronomy.* Wiley New York et al., 1986.

[5] J. Jonas *et. al.*, *The meerkat radio telescope*, in *MeerKAT Science: On the Pathway to the SKA*, vol. 277, p. 001, SISSA Medialab, 2018.

[6] A. V. Oppenheim, A. S. Willsky, and H. Nawab, *Signals and Systems.* Prentice Hall, 2 ed., 1996.

[7] M. Born and E. Wolf, *Principles of Optics.* Cambridge University Press, 7 ed., 1999.

[8] "Radio interferometer." `http://astronomy.swin.edu.au/cosmos/R/Radio+Interferometer`. Accessed: 31/01/2019.

[9] G. Foster, "van cittert-zernike theorem." `http://math_research.uct.ac.za/~siphelo/admin/interferometry/lectures/4-VisibilitySpace/vanCittert-Zernike.pdf`. Accessed: 04/02/2019.

[10] G. B. Taylor, C. L. Carilli, and R. A. Perley, *Synthesis imaging in radio astronomy ii*, in *Synthesis Imaging in Radio Astronomy II*, vol. 180, 1999.

[11] A. Parsons, "Radio astronomy: Tools and techniques." `https://casper.ssl.berkeley.edu/astrobaki/index.php/Radio_Astronomy:_Tools_and_Techniques`. Accessed: 13/02/2019.

[12] A. Sallab, H. Fahmy, and M. Rashwan, *Optimized hardware implementation of fft processor*, .

[13] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing, second edition.* Pearson Education, 1999.

[14] V. Madisetti, *The digital signal processing handbook.* CRC press, 1997.

[15] B. Osgood, *The fourier transform and its applications*, Electrical Engineering Department, Stanford University.

[16] R. Matusiak, *Implementing fast fourier transform algorithms of real-valued sequences with the tms320 dsp platform*, *Application Report SPRA291* (2001).

[17] J. L. Gustafson and I. T. Yonemoto, *Beating floating point at its own game: Posit arithmetic*, *Supercomputing Frontiers and Innovations* **4** (2017), no. 2 71–86.

[18] I. Koren, *Computer arithmetic algorithms.* AK Peters/CRC Press, 2001.

[19] I. Committee *et. al.*, *754–2008 ieee standard for floating-point arithmetic*, *IEEE Computer Society Std* **2008** (2008).

[20] R. Yates, *Fixed-point arithmetic: An introduction*, *Digital Signal Labs* **81** (2009), no. 83 198.

[21] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing.* Prentice-Hall International, 1975.

[22] "Fixmath user's manual." `http://www.nongnu.org/fixmath/doc/`. Accessed: 19/09/2019.

[23] "Skarab." `https://www.ska.ac.za/science-engineering/meerkat/about-meerkat/`. Accessed: 05/11/2019.

[24] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based implementation of signal processing systems.* John Wiley & Sons, 2008.

[25] J. Tarango, "Introduction to fpgas." `http://www.cs.ucr.edu/~jtarango/cs122a_intro_to_fpgas.html`. Accessed: 16/12/2019.

[26] "Casper - about the collaboration." `https://casper.berkeley.edu/index.php/about/`. Accessed: 25/08/2019.

[27] "Skarab." `https://github.com/casper-astro/casper-hardware/blob/master/FPGA_Hosts/SKARAB/README.md`. Accessed: 05/11/2019.

[28] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms.* CRC Press, 1999.