RHODES UNIVERSITY

MASTERS THESIS

Accelerated Implementations of the RIME for DDE Calibration and Source Modelling

---

Author:
Joshua VAN STADEN

Supervisors:
Dr. Landman Bester
Dr. Simon Perkins
Prof. Oleg Smirnov

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Centre for Radio Astronomy Techniques and Technologies (RATT)
Department of Physics and Electronics

February 1, 2021

RHODES UNIVERSITY

Abstract

Faculty of Science
Department of Physics and Electronics

Master of Science

Accelerated Implementations of the RIME for DDE Calibration and Source Modelling

by Joshua VAN STADEN

Second- and third-generation calibration methods filter out subtle effects in interferometer data, and therefore yield significantly higher dynamic ranges. The basis of these calibration techniques relies on building a model of the sky and corrupting it with models of the effects acting on the sources. The sensitivities of modern instruments call for more elaborate models to capture the level of detail that is required to achieve accurate calibration. This thesis implements two types of models to be used in for second- and third-generation calibration. The first model implemented is shapelets, which can be used to model radio source morphologies directly in *uv* space. The second model implemented is Zernike polynomials, which can be used to represent the primary beam of the antenna. We implement these models in the CODEX-AFRICANUS package and provide a set of unit tests for each model. Additionally, we compare our implementations against other methods of representing these objects and instrumental effects, namely NIFTY-GRIDDER against shapelets and a FITS-interpolation method against the Zernike polynomials. We find that to achieve sufficient accuracy, our implementation of the shapelet model has a higher runtime to that of the NIFTY-GRIDDER. However, the NIFTY-GRIDDER cannot simulate a component-based sky model while the shapelet model can. Additionally, the shapelet model is fully parametric, which allows for integration into a parameterised solver. We find that, while having a smaller memory footprint, our Zernike model has a greater computational complexity than that of the FITS-interpolated method. However, we find that the Zernike implementation has floating-point accuracy in its modelling, while the FITS-interpolated model loses some accuracy through the discretisation of the beam.

Acknowledgements

Writing a thesis is not a task that is done by a single person. In some way or another, many people have given some form of input to this thesis. I dedicate this section to expressing my gratitude to all who had a hand in my work, be it directly or indirectly.

First and foremost, I thank my supervisors Dr. Landman Bester, Dr. Simon Perkins and Prof. Oleg Smirnov for all the academic advice, guidance and funding throughout my degree. This would have not been possible without your help.

Secondly, I thank the wonderful members of the RATT research group for their general advice and interesting conversations throughout my time in Masters. A special mention to Ulrich Mbou Sob for his assistance in helping me grasp some concepts in radio interferometry and his feedback on my work. Of course, the RATT research group is not complete without the administrative staff. A special thank you to Ronel Groenewald and Zizipo Lusizi for administrating all of my travels and funding.

To my family in Durban, a special thank you goes to you. Despite the distance between us, you still work hard to support me. Your presence is felt even here in Grahamstown.

A very special thank you goes to my partner and best friend Katherine Gillam, who has seen me develop throughout my Masters degree. You have been a large base of support through some very challenging parts of the past 3 years. You have been an ear to listen to my problems and a voice to give me advice. I think that you have heard enough of me talking (sometimes to you and sometimes to myself) about DDEs and various polynomials to last a lifetime. Thank you for being such a wonderful person.

A very special thank you to my Grahamstown family. Over the years, you guys have helped to make Grahamstown my home. You have selflessly accepted me into your lives and offered a support base when I needed it the most. I am grateful to have wonderful people like you in my life.

This work is based upon research supported by the South African Research Chairs Initiative of the Department of Science and Technology and National Research Foundation.

Contents

List of Abbreviations

DCT	**D**iscrete **C**osine **T**ransform
PSF	**P**oint **S**pread **F**unction
RIME	**R**adio **I**nterferometer **M**easurement **E**quation
1GC	**1**st **G**eneration **C**alibration
2GC	**2**nd **G**eneration **C**alibration
3GC	**3**rd **G**eneration **C**alibration
JIT	**J**ust **I**n **T**ime
LLVM	**L**ow **L**evel **V**irtual **M**achine
DFT	**D**iscrete **F**ourier **T**ransform
FFT	**F**ast **F**ourier **T**ransform
FWHM	**F**ull **W**idth at **H**alf **M**aximum
HWHM	**H**alf **W**idth at **H**alf **M**aximum
GIL	**G**lobal **I**nterpreter **L**ock

Chapter 1

Introduction

1.1 Introduction

Studying astronomical signals in the radio band has yielded significant discoveries, but higher resolution and sensitivity is needed for further discoveries to take place. The resolution of a single element telescope is given as $\theta \sim \frac{\lambda}{D}$, where λ is the wavelength of the observed signal and D is the diameter of the aperture (Thompson, Moran, and Swenson Jr, 2008). By minimizing θ, we say that the telescope has a higher resolution as it is able to resolve smaller objects. One can therefore improve the resolution of the single element telescope by increasing the diameter of the aperture; however, given the large values of λ in the radio regime, the efficacy of this is limited. Interferometry, which is the process of correlating the outputs of multiple elements to synthesize a single element with a larger aperture, is a solution to the issue as mentioned above (Thompson, Moran, and Swenson Jr, 2008). For a two-element interferometer, the response function is given as the distance between the elements. The resolution of an interferometer is therefore $\theta \sim \frac{\lambda}{B}$, where B is the length of the longest baseline in the interferometer (Thompson, Moran, and Swenson Jr, 2008).

While interferometers can achieve high-resolution images, the signals emitted by the sources are corrupted by specific effects in their paths. These effects are associated with the electronics of the telescope, as well as atmospheric effects such as the ionosphere (Thompson, Moran, and Swenson Jr, 2008). Calibration is performed on the data to remove these corruptions and attempt to recover the correct image. It aims to determine the effects acting on the signal and applies the inverse of these effects.

This chapter begins by introducing interferometry and defines the complex visibility function. It follows by defining the radio interferometer measurement equation (RIME), which is a mathematical formalism for modelling antenna and propagation effects. Finally, it explains the process of calibrating for these effects and obtaining higher dynamic ranges through the process of self-calibration (selfcal).

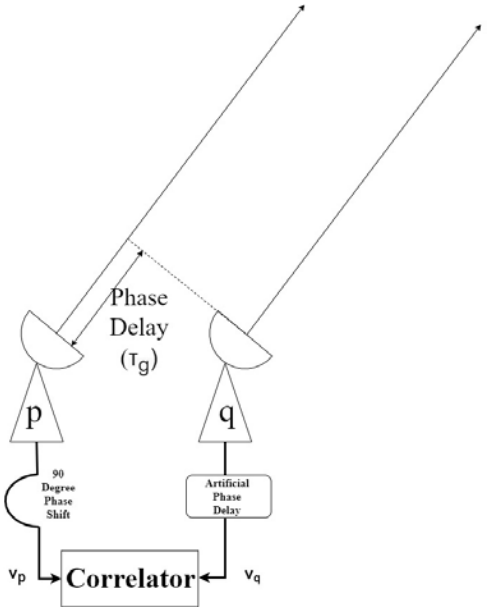


Figure 1.1: A simple, two-element correlating interferometer.

1.2 Interferometry

This section largely follows the explanation of interferometry provided by Thompson, Moran, and Swenson Jr, 2008. The receiving elements (such as dishes) in an interferometer consisting of more than two elements are paired into baselines. As more elements are included in an interferometer, it yields further information for a given observation, the nature of which will be discussed further in this section.

Figure 1.1 illustrates a single baseline, comprised of two elements p and q. The output of each element is transmitted to a correlator, which multiplies the two signals. The voltage output by the correlator, C, is related to the voltages output by the antennas, v_p and v_q by

$$C = \langle v_p v_q \rangle, \tag{1.1}$$

where $\langle \cdot \rangle$ denotes an averaging in time and frequency.

In Equation (1.1), the v_q term, as it is represented in Figure 1.1, equates to an amplitude (A_q) modulated by a cosine function of the angular frequency of the incident wave ($\omega = \frac{2\pi}{\lambda}$) and time ($t$). Therefore,

$$v_q = A_q \cos(\omega t).$$

The voltage measured by antenna p is similar to that measured by antenna q. However, as illustrated in Figure 1.1, there exists a geometric phase delay τ_g of the

signal to antenna p due to the angle between the source and baseline vectors. Therefore, its voltage will amount to $A_p \cos[\omega(t-\tau_g)]$. Equation (1.1) is therefore modulated by the product of these cosine functions

$$C = \langle A_{pq}^2 \cos(\omega t)\cos[\omega(t-\tau_g)]\rangle = \langle \frac{A_{pq}^2}{2}[\cos(\omega\tau_g)+\cos(2\omega t-\omega\tau_g)]\rangle. \tag{1.2}$$

Applying the $\langle\cdot\rangle$ operator over an amount of time sufficiently larger than the period of the wave causes $\cos(2\omega t-\omega\tau_g)$ to approximate to 0. Averaging Equation (1.2) over a time scale sufficiently larger than the period of the wave therefore yields

$$C = \frac{A_{pq}^2}{2}\cos(\omega\tau_g). \tag{1.3}$$

Correlating the outputs of antennas p and q therefore yields $\frac{A_{pq}^2}{2}$ modulated by a fringe pattern $\cos(\omega\tau)$, which is a function of the angular frequency and the phase delay of the source. As the source moves across the fringe pattern, the sensitivity of the telescope to the source will change.

Equation (1.3) samples the even component of the source brightness, without sampling the odd component. A phase delay of 90 degrees is inserted into the signal path from antenna p to the correlator to sample the odd component of the sky brightness. We refer to this as phase-switching. Inserting this artificial phase delay yields

$$C = \frac{A_{pq}^2}{2}\cos(\frac{\pi}{2}-\omega\tau_g) = \frac{A_{pq}^2}{2}\sin(\omega\tau_g). \tag{1.4}$$

We now define C_{cos} as the correlator response given in Equation (1.3). C_{sin} can be defined as the correlator response with a 90-degree phase shift introduced into the signal path

$$C_{sin} = \frac{A_{pq}^2}{2}\sin(\omega\tau_g). \tag{1.5}$$

With the sin component of the visibility, we can now define the complex visibility function $\mathbf{V}$ for a single baseline

$$\mathbf{V} = C_{cos} - \imath C_{sin} = \frac{A_{pq}^2}{2}[\cos(\omega\tau_g) - \imath\sin(\omega\tau_g)] = \frac{A_{pq}^2}{2}e^{-\imath\omega\tau_g} \tag{1.6}$$

In interferometry, we aim to measure the sky brightness $\mathbf{I}$. In this case, the sky brightness is measured as $\frac{A_{pq}^2}{2}$. Substituting $\mathbf{I} = \frac{A_{pq}^2}{2}$ into Equation (1.6), one gets

$$\mathbf{V} = \mathbf{I}e^{-\imath\omega\tau_g}. \tag{1.7}$$

As further sources contribute to the observation, their brightnesses add linearly to the complex visibility. In the continuous case, this gives

$$\mathbf{V} = \int\int_{sky} \mathbf{I}(\mathbf{s})e^{-\imath\omega\tau_g}d\Omega, \tag{1.8}$$

where $d\Omega$ denotes an infinitesimal area on the unit sphere, and $\mathbf{s}$ denotes a unit vector showing the direction of the source. The phase delay, τ_g is defined as $\mathbf{b} \cdot \mathbf{s}$, where $\mathbf{b}$ denotes the baseline vector. We define our xyz coordinate system with the z coordinate pointing in the direction of the center of the target field, which we define as $\mathbf{s}_0$. We define (l, m, n) as the three components of the $\mathbf{s}$ vector, where $n = \sqrt{1 - l^2 - m^2}$ because $\mathbf{s}$ is a unit vector. We also define (u, v, w) as the three components of the baseline vector $\mathbf{b}$, where w lies in the direction of $\mathbf{s}_0$. Because we wish to describe the uvw coordinates in terms of wavelengths, we therefore say that $(u, v, w) = \frac{\mathbf{b}}{\lambda}$ gives us uvw coordinates in units of wavelengths. Because $\omega = \frac{2\pi}{\lambda}$, the $e^{-\imath\omega\tau_g}$ equates to $e^{2\pi\imath(ul+vm+wn)}$. Substituting this into Equation (1.8), we get

$$\mathbf{V}(u, v) = \int\int_{sky} \mathbf{I}(l, m) e^{-2\pi\imath(ul+vm+wn)} d\Omega. \tag{1.9}$$

Towards $\mathbf{s}_0$, the complex exponent in Equation (1.9) becomes equal to $e^{-2\pi\imath wn}$ which is variable in time and frequency. This is undesirable, as the correlator averages in time and frequency, which causes a loss of amplitude in a phase-variable signal. To minimize this loss in the direction $\mathbf{s}_0$, the correlator can insert an artificial phase delay corresponding to $\mathbf{b} \cdot \mathbf{s}_0$. This results in

$$\mathbf{V}(u, v) = \int\int_{sky} \mathbf{I}(l, m) e^{-2\pi\imath[ul+vm+w(n-1)]} \frac{dldm}{n}. \tag{1.10}$$

As a consequence of this, the phase remains at 0 at the phase centre.

Equation (1.10) amounts to a 2D Fourier transform of the sky in the absence of the w-term. This can occur for a coplanar radio interferometer ($w = 0$) or for observations with a very narrow field of view ($n \approx 1$). Many algorithms exist to eliminate this term and transform Equation (1.10) into a 2D Fourier transform. These algorithms include w-stacking and w-projection (Pratley, Johnston-Hollitt, and McEwen, 2019). With the assumption that Equation (1.10) is a 2D Fourier transform or with w corrections made, we say that each baseline vector samples its corresponding frequency in Fourier space.

When making measurements of the sky brightness, the antenna has a limited field of view. The power collected by an antenna can be expressed as (Thompson, Moran, and Swenson Jr, 2008)

$$\mathbf{P} = \tilde{\mathbf{A}}_\nu(\theta, \phi)\mathbf{I}_\nu(\theta, \phi)\Delta\Omega\Delta\nu, \tag{1.11}$$

for radial coordinates (θ, ϕ) and over a solid angle Ω. The aperture function of the antenna is expressed as $\tilde{\mathbf{A}}$. This aperture function is circular for an ideal dish with no imperfections. The Fourier transform of the dish aperture function is given as a jinc function which gives rise to the primary beam (Thompson, Moran, and Swenson Jr, 2008). In practical applications, the primary beam will have slight variations from the jinc function that arise from aperture imperfections. Incorporating the primary

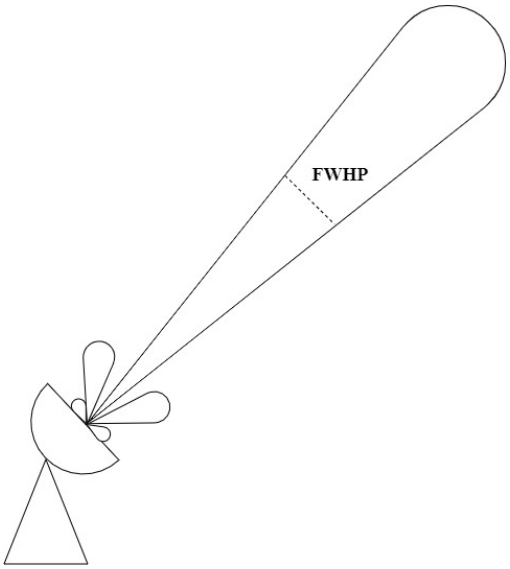


Figure 1.2: The primary beam of a single element in a radio interferometer.

beam $\mathbf{A}_\nu(l,m)$ into Equation (1.10), we get (Thompson, Moran, and Swenson Jr, 2008)

$$\mathbf{V}(u,v) = \int\int_{sky} \mathbf{A}_\nu(l,m)\mathbf{I}(l,m)e^{-2\pi\imath[ul+vm+w(n-1)]}\frac{dldm}{n}. \tag{1.12}$$

The primary beam has a multiplicative effect in the image domain. As illustrated in Figure 1.2, it consists of a main lobe at the pointing centre, as well as smaller side lobes around the edges. The main lobe is characterized by its Full Width at Half Maximum (FWHM) which is defined as the diameter of the beam at half its power. To read the full brightness of a source, one must shift the center of the primary beam to that source. This is simply done by steering the antenna physically towards that source.

Due to the limited sampling of the uv plane by the limited number of baselines, the measured visibilities amount to the true Fourier transform of the sky $\mathbf{V}$ multiplied with the sampling function $\mathbf{S}$ in Fourier space. According to the convolution theorem,[1] this amounts to a convolution of the true image with the point spread function $\mathbf{PSF} = \mathcal{F}\{\mathbf{S}\}$ to produce the dirty image. Note that we use $\mathcal{F}(f)$ to denote the Fourier transform of f.

One can attain a better PSF by making observations at fixed time intervals. Due to the rotation of the Earth relative to the source, this results in the antennas sampling more spatial frequencies (Ryle, 1962). This is referred to as Earth-rotation synthesis and forms an improved PSF with smaller sidelobes.

The intrinsic source brightness varies with frequency. This variation is generally smooth across frequency and follows a power law (Thompson, Moran, and Swenson

[1] The convolution theorem states that a multiplication in Fourier space amounts to a convolution in image space, and vice versa.

Jr, 2008). This power law is given as

$$\mathbf{I}_\nu = \mathbf{I}_{\nu_0}(\frac{\nu}{\nu_0})^\alpha, \tag{1.13}$$

where ν_0 is the reference frequency and α denotes the spectral index. The spectral index of the source is a product of the properties of that source. Certain properties can be inferred from the spectral index.

1.3 Visibility Modelling

1.3.1 The Measurement Equation

The radio interferometer measurement equation (RIME, or measurement equation) is a mathematical formalism describing the effects acting on the observed visibilities from the signal's point of origin to its measurement at the correlator (Hamaker, Bregman, and Sault, 1996; Smirnov, 2011). This formalism is a tool used in modelling sources and instrumental effects for calibration, the significance of which will be explored in Section 1.4.2.

For this section, we will follow the formalism of (Smirnov, 2011). Let us consider a quasi-monochromatic signal originating from a source in the sky. Using the same orthonormal basis described in Section 1.2 for our xyz coordinates, we describe the signal $\mathbf{e}$ as a complex-valued column vector

$$\mathbf{e} = \begin{pmatrix} e_x \\ e_y \end{pmatrix}. \tag{1.14}$$

In the absence of any instrumental effects, the measured voltage at each station becomes the value of the electric field itself. We can therefore describe the voltage measured at each aperture $\mathbf{v}$ as an equivalent to the electric field

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}. \tag{1.15}$$

The voltages output from antennas p and q are averaged over a small time and frequency interval and input to the correlator. The correlator then outputs four values: the voltages at each feed from antenna p multiplied with the conjugate of the voltages from each feed at antenna q. We can define this as the 2×2 visibility matrix, which represents the visibilities after all corruptions have affected the source

$$\mathbf{V}_{pq} = 2 \begin{pmatrix} \langle v_{px} v_{qx}^* \rangle & \langle v_{px} v_{qy}^* \rangle \\ \langle v_{py} v_{qx}^* \rangle & \langle v_{py} v_{qy}^* \rangle \end{pmatrix}. \tag{1.16}$$

The factor of 2 is a useful convention, see Smirnov, 2011 for details.

As mentioned before, the voltages output by the apertures are directly proportional to the electric field vector in the absence of any effects. In fact, disregarding the

phase delay, the output of the correlator without any effects acting upon the signal is referred to as the brightness matrix. This is a 2×2 matrix describing the polarization of the source as measured with each correlation. From the brightness matrix, one can determine the Stokes parameters (McMaster, 1954). The brightness matrix is related to the Stokes parameters through

$$\mathbf{B} = 2 \begin{pmatrix} \langle e_{px} e_{qx}^* \rangle & \langle e_{px} e_{qy}^* \rangle \\ \langle e_{py} e_{qx}^* \rangle & \langle e_{py} e_{qy}^* \rangle \end{pmatrix} = \begin{pmatrix} I + Q & U + \imath V \\ U - \imath V & I - Q \end{pmatrix}. \tag{1.17}$$

The visibility $\mathbf{V}$ measured at antennas p and q after corruptions is given in RIME terms as

$$\mathbf{V}_{pq} = \mathbf{G}_p \left(\int \int \mathbf{E}_p(l,m) \mathbf{K}_p(l,m) \mathbf{B}(l,m) \mathbf{K}_q^H(l,m) \mathbf{E}_q^H(l,m) dl dm \right) \mathbf{G}_q^H. \tag{1.18}$$

In Equation (1.18), the effects acting on $\mathbf{B}$ are broken down into direction-dependent effects (DDEs) $\mathbf{E}$ and direction-independent effects (DIEs) $\mathbf{G}$. Each of these matrices can be expanded into Jones chains, which is typically the case if multiple DDEs and DIEs act on the source.

Equation (1.18) also has the phase delay matrix $\mathbf{K}$, which equates to a scalar matrix of the value $e^{-\imath\omega\tau}$. This matrix combines with the brightness matrix $\mathbf{B}$ to form the source coherency (denoted as $\mathbf{X}$)

$$\mathbf{X}_{pq} = \mathbf{K}_p \mathbf{B} \mathbf{K}_q^H. \tag{1.19}$$

The source coherency is, therefore, the Fourier transform of the brightness matrix. This can be thought of as the visibility at all 4 correlations of a single point in the sky. It is more complicated to speak of the source coherency matrix of extended sources when there are non-trivial DDEs affecting the observation. The reason for this is that the DDEs cannot be pulled out of the integral in (1.18). One way to overcome this is to assume that the DDE is constant across the extent of the source. This will only be an accurate approximation when the source is small compared to the typical length scales over which the DDE varies. This is the regime we will confine ourselves to in this thesis. We can then talk about the source coherency matrix of extended sources as the Fourier transform of some pre-determined shape function scaled by the total flux of the source. This is further elaborated on in § 3.

The RIME given in Equation (1.18) describes a continuous sky brightness distribution where a source lying in a given direction (l, m) is corrupted by a DDE in that specific direction and then a DIE. This representation is suitable for describing a continuous sky. However, to actually process interferometric data on a computer, we inevitably have to work with discretisations of (1.18). For our purposes we will find it convenient to discretise the sky into a number of discrete sources which combine linearly to give the total brightness matrix. Furthermore, we assume that these

sources are not affected by DIEs so that the **G** term falls away. Additionally, we assume that the only DDE acting on the source is the primary beam (also denoted as **E**). Assuming that the primary beam remains constant across the extent of a single source, but can vary from source to source, we end up with the following discrete form of the RIME

$$\mathbf{V}_{pq} = \sum_s \mathbf{E}_p(l_s, m_s)\mathbf{X}_{pq}\mathbf{E}_q^H(l_s, m_s). \tag{1.20}$$

This thesis makes use of this RIME to generate visibilities from the implemented models.

1.4 Calibration and Imaging

Calibration is the process of solving for and correcting the corruptions (a.k.a. gains) described in Section 1.3.1. The purpose of calibration is to determine antenna gains **G** which minimize

$$min_{\mathbf{G}}||\mathbf{D} - \mathbf{GMG}^H||, \tag{1.21}$$

where **D** denotes the observed data and **M** denotes the model visibilities. The inverse of the gain solutions can then be applied to **D** to produce what is known as the corrected data.

This section will describe the process of calibrating for these gains. It will begin by describing how the first-order gain terms are calibrated for, with the help of calibrator sources. It follows by describing self-calibration which eliminates more subtle effects and achieves higher dynamic ranges in the images for more precise measurements.

1.4.1 First Generation Calibration

First-generation calibration (1GC) is the first step in calibrating data from radio measurements. It uses calibrator sources in the sky of known flux density and spectrum to compute antenna gains (Thompson, Moran, and Swenson Jr, 2008). A typical observation is sectioned into separate calibrator and target scans. The visibilities obtained during calibrator scans are used in combination with the known source properties to compute the gains (or Jones matrices) affecting the observation. These gains are then transferred to the target field by interpolating across scan boundaries and applying the inverse of the interpolated gains to form the corrected data.

1GC aims to eliminate dominant effects in the observation data. Firstly, it solves for the flux scale. This is so that a certain response in the interferometer will translate to a known flux density measurement. Secondly, the response of an antenna changes as a function of frequency and this response, referred to as the bandpass, has to be calibrated for during 1GC. To calibrate for the bandpass, one requires a

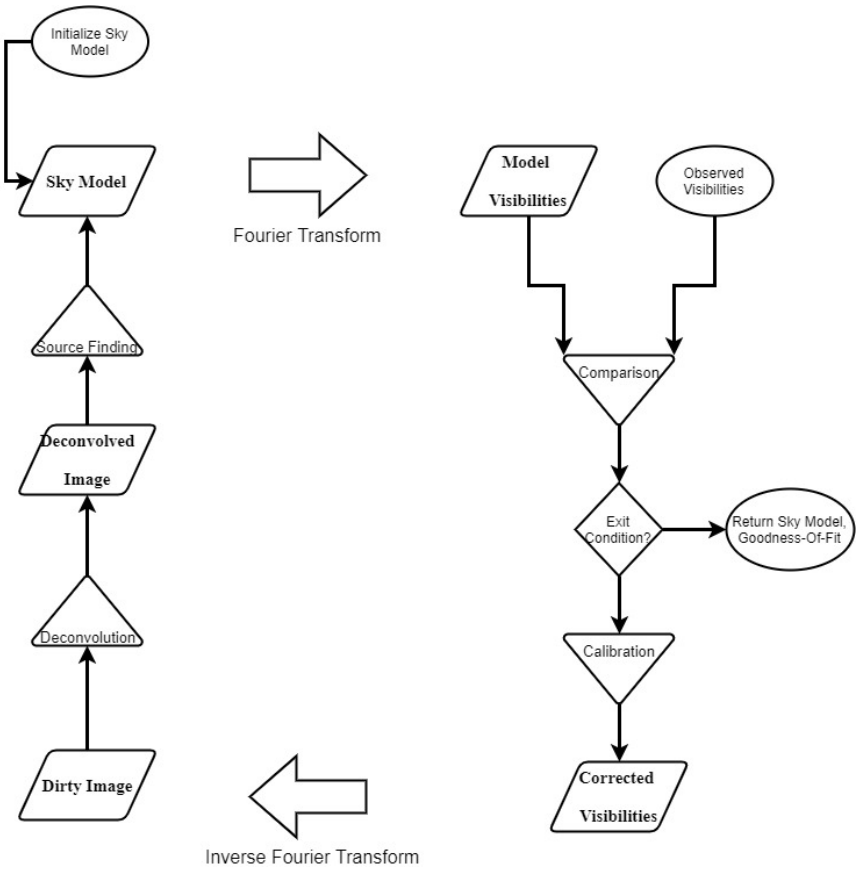


Figure 1.3: The process of selfcal. Figure adapted from (Sob et al., 2017).

point-like calibrator source with sufficient brightness and a known frequency spectrum. If no point-like sources are available for calibration, one with a well-modelled morphology will suffice. In addition to the flux scale and bandpass one has to solve for possible delay errors in the correlator.

These effects all have to be calibrated for before we can even consider the target field of the observation. Typically the data only starts resembling that of the target field once these effects have been corrected for. However, these are still just first order corrections and additional corruptions are inevitable. However, unlike the calibrator scans, we do not have access to the true model visibilities, $\mathbf{M}$, during target scans.

1.4.2 Second Generation Calibration

After the interferometric data is calibrated using 1GC, higher dynamic ranges can be achieved by using second-generation calibration (2GC) (Cornwell and Wilkinson, 1981), also known as direction independent self-calibration (selfcal). 2GC is an iterative process which starts by deriving an approximate model image using the data obtained after 1GC and then alternates between calibration and imaging in an attempt to refine both the calibration solutions and the image of the target field.

Figure 1.3 illustrates the process of self-calibration. The first iteration of the process begins with calibrated data **D** from 1GC. From this data, imaging and deconvolution produces an image of the target field. A model is generated from this image and degridded into model visibilities **M**. Similar to 1GC, antenna gains **G** are then determined using the model visibilities. The inverse of these gains are applied to the data **D** and the next iteration of 2GC begins with more accurate data.

Because the sky model used is a single image, 2GC cannot correct for direction-dependent corruptions on sources within the sky model. Hence, 2GC can only correct for DIEs. While this increases the dynamic range of the image, further improvements on the dynamic range can be achieved by correcting for the more subtle DDEs.

1.4.3 Third Generation Calibration

Modern interferometers are characterised by their high sensitivities, wide bandwidths and larger fields of view, all of which make the problem of DDEs more apparent. These effects include the primary beam (see Section 1.2), which will be simulated in this thesis. DDEs, like DIEs, can be calibrated for using a selfcal-like procedure. This further increases the obtainable dynamic range in the image. The process of using selfcal to correct for DDEs is termed third-generation calibration (3GC). In RIME terms, the DDEs are modelled as $\mathbf{E}_p$ and $\mathbf{E}_q^H$ in Equation (1.20).

Other than the types of gains solved for, the other main difference between 2GC and 3GC is in the sky model used. While 2GC uses a single image to represent the sky, 3GC models the sky as a sum of separate individual sources. This is due to the fact that each source is corrupted by the DDE differently based on its position. Source-finders are used to build up a component-based sky model from which visibilities can be generated. Software packages such as Montblanc (Perkins et al., 2015), Meqtrees (Noordam and Smirnov, 2010) and Codex-Africanus [2] are packages that implement these models. These are, however, restricted to modelling Gaussians and point sources which obviously cannot capture more complicated source morphologies. This leads to possible incorrect or incomplete models which can bias the calibration procedure and lead to subtle artefacts in the final image, such as ghost sources (Grobler et al., 2014; Nunhokee, Grobler, and Smirnov, 2015). .

Generally, a heuristic-based approach is adopted in modelling DDEs, as modelling each physical effect undergone by the source can become intractable. A particularly important DDE is the primary beam, since it affects virtually all observations. Depending on the instrument, this particular DDE can be approximated with various models. One example of such a model is the Westerbork Synthesis Radio Telescope (WSRT) $\cos^3$ model. This model is calculated as

$$E(l,m) = \cos^3(C\nu\sqrt{l^2+m^2}), \tag{1.22}$$

[2] https://github.com/ska-sa/codex-africanus

where C is roughly a constant (for a given bandwidth) that varies slightly with frequency. This model is valid only above the 10% level of the beam.

The MeerKAT antenna primary beam has no such simple approximation. Various methods exist to model this beam, such as the Eidos software package (Asad et al., 2019)[3] which uses Zernike polynomials (see Chapter 4). Most software packages for modelling antenna gains do not contain a specific model for the MeerKAT primary beam. The MONTBLANC and MEQTREES packages do implement a generic primary beam modelling capability where the beam is represented by 3-dimensional (xy plus frequency) cubes.

1.5 Problem Statement

As discussed in Section 1.4.3, calibrating data with an incomplete sky model leads to subtle effects such as source suppression and ghost sources. Current software packages support point source models and Gaussian models for extended sources. While one is able to amalgamate various combinations of these models, it does not always provide a realistic representation of the source, particularly in the case of complicated source morphologies. This project aims to provide an implementation of a more realistic galaxy morphology model into a current software package. Additionally, we aim to provide an accelerated implementation of the MeerKAT primary beam. Overall, this provides a component-based sky model for complex sources, as well as the ability to corrupt them for higher accuracy.

The goals of this thesis are to

- Provide shapelet expansion functionality in CODEX-AFRICANUS.
- Provide Zernike polynomial expansion functionality in CODEX-AFRICANUS.
- Provide a proof-of-concept of these two implementations.

1.6 Thesis Outline

This chapter has given an outline on interferometers and how calibration works with radio interferometers. Chapter 2 gives an outline on the principles of the forms of acceleration we use, and a description of the software environment in which we implement our models.

We follow in Chapters 3 and 4, which describe the two basis functions implemented viz. shapelets and Zernike polynomials. This includes a description on testing done on the implementations and experiments run on them. These experiments compare the implementations with other forms of source and beam modelling.

In Chapter 5, we conclude our findings, discussing the viability of these implementations in a software package, and future improvements on the implementations of the basis functions.

[3] https://github.com/ratt-ru/eidos

Chapter 2

Computing Environment

2.1 Introduction

The aim of this thesis is to create accelerated implementations of models for the purpose of DDE calibration. To accelerate these models, we must find the optimal tools and technologies available for this purpose.

Our goal for this software matches the goals of CODEX-AFRICANUS. This means that we aim to provide an implementation of these basis functions that is easily accessible to most radio astronomers. To make their code highly accessible, the creators of CODEX-AFRICANUS have written their software package in PYTHON.

Because PYTHON suffers from an interpreter overhead and has a Global Interpreter Lock (GIL) on its scripts, we make use of a JIT compiler. This releases the GIL and runs highly optimized low-level bytecode. We assume that our software will be running on a modern multi-core computer. For processing larger measurement sets, it may be run on a distributed compute cluster. As a result, we can take advantage of parallelism to accelerate our code.

Section 2.2 introduces the concept of compute graphs, which is an underpinning concept of one of the frameworks we use to accelerate our implementation. Section 2.3 presents the tools and technologies used in the model implementation. Section 2.4 presents the testing framework used for verifying the implementations and section 2.5 concludes with an explanation on examining computational complexity.

2.2 Compute Graphs

Our implementation makes use of computational graphs. A computational graph is a directed graph that models operators and variables as nodes and data-flows as edges (Rocklin, 2015). Figure 2.1 illustrates a simple computational graph that computes $5(x + yz)$. It begins by computing $u = yz$. Once u has been computed, it will compute $v = x + u$. It concludes by outputting $f = 5v$. Each node in a computational graph represents an operator, with inputs and outputs.

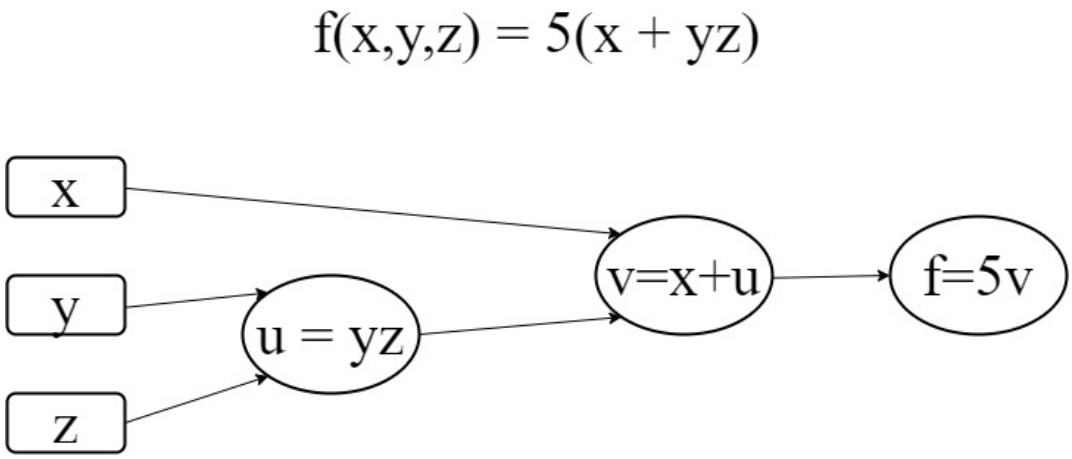


Figure 2.1: An example computational graph, computing $5(x + yz)$.

2.2.1 Compute Graphs in Software

Many popular software frameworks model computational tasks using compute graphs. Notable frameworks include PyTorch (Paszke et al., 2019), Theano (The Theano Development Team et al., 2016), TensorFlow (Abadi et al., 2015) and Dask (Dask Development Team, 2016). This is particularly useful in the field of machine learning where the underlying structure of a neural network can be represented by a graph. Other than deep learning frameworks, compute graphs are used in software for radio astronomy. The first software package for radio astronomy that made use of compute graphs is MeqTrees (Noordam and Smirnov, 2010). Much like the work in this thesis, MeqTrees uses compute graphs to represent and compute measurement equations for the purposes of visibility simulation.

Frameworks like Dask allow the user to break computation up into smaller subsections which can be submitted by a scheduler to multiple threads, cores and nodes, allowing the computational load to spread efficiently. This enables the user to fully exercise every core in a CPU and every node in a cluster. By sharing the computation across multiple nodes, the framework can solve problems that are out-of-core, where the total memory requirements exceed the size of the RAM in a single node.

Additionally, computational graphs can be evaluated lazily. This means that the evaluation of a variable is deferred until that particular value is used in the code. Evaluating code lazily can introduce some benefits, such as the ability to create calculable infinite data structures and avoiding error conditions and unnecessarily large computations where they are not needed.

2.3 The Model Architecture

2.3.1 Codex-Africanus

We implement the shapelet (Chapter 3) and Zernike (Chapter 4) models into CODEX-AFRICANUS,[1] still under development at the time of writing. The aim of this package is to provide a well-documented and accelerated implementation of common algorithms for calibrating and imaging radio interferometric data.

For readability and rapid development, CODEX-AFRICANUS is implemented in PYTHON (Python Software Foundation, 2018). This allows the software package to be accessible to most radio astronomers as PYTHON is easily learned. CODEX-AFRICANUS is accelerated using a combination of two frameworks: NUMBA (Lam, S. K. and Pitrou, A. and Seibert, 2015) and DASK (Dask Development Team, 2016). NUMBA provides a JIT compilation of each individual algorithm and DASK builds a compute graph to combine multiple accelerated algorithms.

2.3.2 Numba

NUMBA (Lam, S. K. and Pitrou, A. and Seibert, 2015) is a JIT compiler for PYTHON. This JIT compiler is built on LLVM. Functions JITted by NUMBA are not affected by PYTHON's GIL, thereby freeing the PYTHON interpreter to execute PYTHON code on multiple threads. This allows the NUMBA function to fully utilise the available CPU cores. The NUMBA JIT compiler exists as a function decorator. A typical JITted function will be of the form

```
@numba.jit(out_type(in_type1, in_type2), options)
def my_function(param1, param2):
```

This statically types the inputs and output of the function (and therefore all data types within the function). Explicitly declaring types in NUMBA is optional and, if the types aren't specified, a separate compilation is created for each permutation of input type that is inferred from the data.

2.3.3 Dask

CODEX-AFRICANUS uses DASK to build compute graphs, with each node in the graph being a NUMBA-JITted function. DASK is able to create chunks of NUMPY arrays from one large array and submit parallel computations per chunk to the scheduler. The scheduler will then perform these computations on multiple cores or even nodes. Additionally, one can access measurement sets using DASK-MS[2] which provides a Data Access Layer.

DASK is written using a simple and easy-to-grasp syntax. An example of the syntax used in DASK can be seen below.

[1] https://github.com/ska-sa/codex-africanus/pull/99
[2] https://github.com/ska-sa/dask-ms

```
import dask.array as da
import numpy as np

N = 100
c = N // 5

list_one = da.arange(N, chunks=c)
list_two = da.arange(N, chunks=c)
list_product = list_one * list_two
list_sum = list_one + list_product

print(list_sum.compute())
```

This code snippet creates 5 chunks of data, with each chunk being a NUMPY array of 20 elements. As a result, DASK will submit 5 copies of the computational graph to the scheduler.

The DASK scheduler handles multi-threading automatically. This removes a large amount of responsibility from the developer, while still keeping the code readable. This scheduler keeps the memory footprint small by applying a Last-In-First-Out policy. This ensures that the most recently-created tasks get a higher priority and prevents thread starvation whereby a thread is denied access to shared resources due to other threads over-utilising these resources.

Figure 2.2 illustrates the compute graph associated with the above code snippet. The code snippet generates 5 copies of the compute graph, with each copy having a unique identifying number.

Because DASK is evaluated lazily, no values are computed until the COMPUTE() method is called. As a result, if one wanted to inspect the *list_product* value, one would simply have to call the COMPUTE() method on the *list_product* variable which stores the computational graph to calculate *list_product*.

2.4 Testing

When developing software, it is important for testing to be performed on the implementations. Testing allows one to specify conditions for their software to be considered functional.

For this thesis, we simply perform unit testing on our implementations. Unit testing tests code at the smallest level of functionality. In unit testing, we wish to determine if our implementation outputs what we expect it to output.

While one can technically create their own manual tests, there are many benefits of making use of a testing tool such as PYTEST, NUNIT, etc. Testing frameworks such as PYTEST and NUNIT facilitate the rapid creation of unit tests. Test case development in these tools are created rapidly and are more readable. Creating test cases in these frameworks confirm that new added features work while still ensuring that previous features are functional. These frameworks include the option to

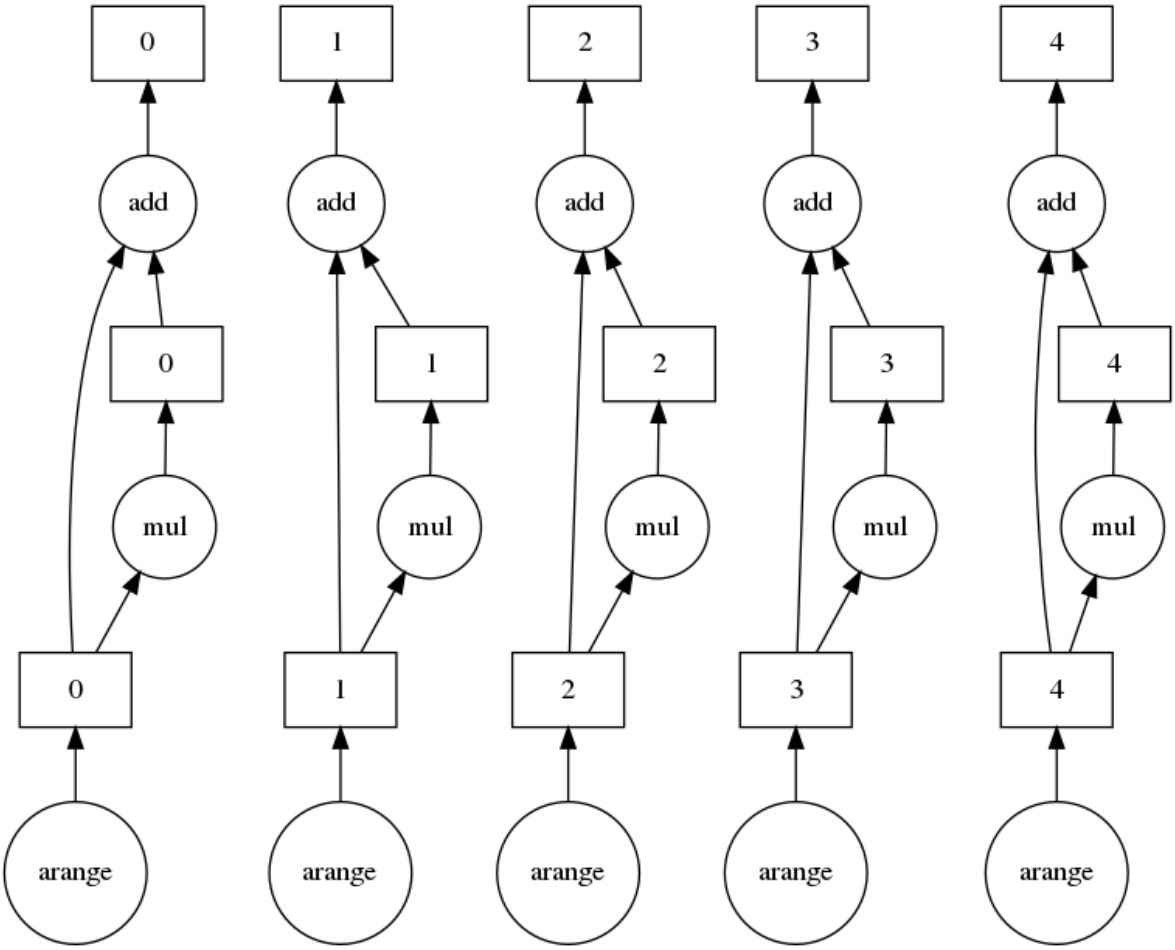


Figure 2.2: An example compute graph generated by DASK.

create setup and teardown functions which respectively creates and destroys test data, allowing this data to be used by multiple test cases before being destroyed.

Unit tests are run on continuous integration (CI) servers. When the developer uploads a change to the repository, the CI server runs the test cases to ensure that existing functionality is not broken.

2.4.1 Pytest

There are different frameworks with which to perform unit testing, depending on requirements. Some software frameworks require mocks of data types to be created to test certain functionality. These mocks exist to create an instance of an object without affecting various aspects of the system. This is useful in many business contexts. For example, in a retail system that allows users to purchase items from the system, one does not wish to go through the full procedure of creating a new purchase, as this might create charges and affect stock lists. One might simply wish to create a purchase object in the system.

Our implementation exists as an isolated aspect of CODEX-AFRICANUS. Because we simply wish to implement various models for imaging and calibration, this does not affect data in a database, nor does it have any negative side-effects on an already existing system. As a result, PYTEST fills all the needs we have on our implementation. For the sake of testing, we simply require evaluating our respective polynomials on lm or uv grids.

PYTEST is an easy-to-learn testing framework. Syntax for marking a function in the script as a test simply requires that the name of the function begins with TEST_. Options for running the test can be added by adding a function decorator. It also has support for test fixtures, which are consistent pieces of data with which to test. Like most PYTHON packages, there is also a large community of support for using PYTEST, and it is open-source.

2.5 Computational Complexity

In determining the value of applying an algorithm to a particular problem and comparing it to other algorithms, one must look at how the resource cost of that algorithm scales with the problem size. This is referred to as the computational complexity of the algorithm. The computational resources that are analyzed in this manner are the CPU cycles (the time complexity) and the amount of storage in memory (the space complexity) (Wilf, 2002). Each complexity has a specific set of programming constructs that determines how it scales.

The scaling of the time complexity is determined by the increase in runtime introduced by an increase in a specific input. To determine this increase, one must look at the loops and recursive calls in their algorithm. If a loop in an algorithm depends on the size of a specific input, that algorithm will scale with that input. Recursive

calls can cause varying forms of scaling within a program, depending on the input of the recursive call.

The spatial complexity informs on how the primary memory requirements of the system scales with a larger input space. To determine this, one must examine the inputs and outputs of the algorithm, as well as the auxiliary memory requirements. Additionally, recursive calls can cause scaling in stack memory requirements, which may cause an overflow if left unchecked.

For analysing computational complexity, we use asymptotic notation (Bachmann, 1894; Landau, 1909). This notation has 3 forms: the $\mathcal{O}$ (big-O), Θ (big-Theta) and Ω (big-Omega) notations (Bachmann, 1894; Landau, 1909; Wilf, 2002). For $f(x)$ and $g(x)$

- $f(x) = \mathcal{O}(g(x))$ if $\exists C, x_0$ such that $|f(x)| < Cg(x) \forall x > x_0$,
- $f(x) = \Theta(g(x))$ if $\exists c_1 > 0, c2 > 0$ such that $c_1 g(x) < f(x) < c_2 g(x) \forall x$, and
- $f(x) = \Omega(g(x))$ if $\exists C > 0$ and a sequence $x_1, x_2, x_3, \ldots$ tending to infinity, such that $\forall j |f(x_j)| > Cg(x_j)$.

For this thesis, we analyze the computational complexity in our algorithms through the $\mathcal{O}$ notation as this represents the upper limit of the runtime of our algorithm. Characteristically, the $\mathcal{O}$ notation does not consider constants (Wilf, 2002). This is due to the fact that, in this notation, we measure how an algorithm scales with the scaling of the input. Additionally, $\mathcal{O}$ notation drops non-dominant terms as we are measuring the upper limit of the runtime (Wilf, 2002).

2.6 Conclusion

While statically typed languages offer a far superior runtime to dynamically typed languages, they are difficult to develop in due to their strict type constraints and longer compile times. Dynamically typed languages trade fast execution speed for rapid development and dynamic typing. JIT compilation offers a medium between the two as code may be written in a dynamically typed language and compiled into an executable block of code.

An efficient language, however, is not sufficient to accelerate code on modern hardware systems. Modern computers are multi-core, which requires multi-threading to achieve any acceleration. For large-scale computations, compute clusters exist to distribute a large workload among many nodes. In terms of frameworks, one can use DASK to perform multi-threading and distributed computations. This framework makes use of a lazily-evaluated compute graph and a specialized scheduler to evenly distribute the workload.

Our implementation is created using PYTHON with a NUMBA JIT compiler and DASK to perform parallel and distributed computations. We implement our models into CODEX-AFRICANUS, which is a suite of common algorithms for radio astronomy. To perform unit testing on our implementation, we use PYTEST.

Chapter 3

Modelling Galaxy Morphologies using Shapelets

3.1 Chapter Outline

Refregier, 2003 proposed a method for decomposing a function into a series of basis functions describing smaller shape components termed *shapelets*. These shapelets are particularly useful for describing galaxy morphologies. Additionally, they remain invariant under Fourier transform. This allows for fast and direct modelling in Fourier space.

This chapter presents an implementation of the shapelet model for galaxy morphologies. In Section 3.2, we begin by introducing the shapelet basis functions. We follow by detailing the implementation of the shapelets in Section 3.3 and the unit tests run on them. We present a proof-of-concept of the implementation in Section 3.4 by using it to simulate NGC 6251 (Waggett, Warner, and Baldwin, 1977), a radio galaxy with a single jet. This includes a description of the simulated observation of NGC 6251. Finally, we present a comparison of its utility against that of NIFTY-GRIDDER (Reinecke, Steininger, and Selig, 2018) in Section 3.5.

3.2 Gauss-Hermite Shapelets

A model image, henceforth referred to as $\mathbf{I}(l, m)$, can be decomposed into a set of shapelet coefficients and expanded as (Refregier, 2003)

$$\mathbf{I}(l, m) = \sum_{n_1, n_2}^{N_{max}} \mathbf{f}_{n_1, n_2} B_{n_1}(l; \beta_l) B_{n_2}(m; \beta_m), \tag{3.1}$$

where β is the characteristic extent of the source. The basis functions are summed over n_1 and n_2 which denote the order of the given polynomial along the l and m axes, respectively. Thus, the coefficient describing the polynomial at order (n_1, n_2) is $\mathbf{f}_{n_1, n_2}$. The order of the shapelet is taken to be the sum of n_1 and n_2. We truncate the expansion at some maximum order N_{max}. This creates a triangle of coefficients. We

discuss this further in Section 3.4.3. The basis function B_n is given as

$$B_n(x;\beta) = \frac{H_n(\frac{x}{\beta})e^{-\frac{x^2}{2\beta^2}}}{\sqrt{2^n \pi^{\frac{1}{2}} n! \beta}}, \tag{3.2}$$

where H_n is a Hermite polynomial of order n.

The Hermite polynomial is defined by the contour integral (Arfken and Weber, 1999)

$$H_n(x) = \frac{n!}{2\pi\imath} \oint e^{-t^2+2tx} t^{-n-1} dt, \tag{3.3}$$

where the contour encircles the origin and traverses in a counter-clockwise direction. The integral results in polynomials of the form

$$\begin{aligned}
H_0(x) &= 1 \\
H_1(x) &= 2x \\
H_2(x) &= 4x^2 - 2 \\
H_3(x) &= 8x^3 - 12x \\
H_4(x) &= 16x^4 - 48x^2 + 12 \\
H_5(x) &= 32x^5 - 160x^3 + 120x \\
H_6(x) &= 64x^6 - 480x^4 + 720x^2 - 120 \\
H_7(x) &= 128x^7 - 1344x^5 + 3360x^3 - 1680x \\
H_8(x) &= 256x^8 - 3584x^6 + 13440x^4 - 133440x^2 + 1680.
\end{aligned}$$

The Hermite polynomials have a recurrence relation of

$$H_n(x) = 2xH_{n-1}(x) - 2nH_{n-2}(x), \tag{3.4}$$

which can be easily implemented.

The parameters required for the expansion outlined in Equation (3.1) consist of the two β parameters (one for the l dimension and one for the m dimension) as well as the matrix of coefficients. Each coefficient in the matrix can be determined as

$$\mathbf{f}_{n_1,n_2} = \int \int_{-\infty}^{\infty} \mathbf{I}(l,m) B_{n_1}(l;\beta_l) B_{n_2}(m;\beta_m) dl dm. \tag{3.5}$$

The first 16 basis sets up to $n_1 = 4$ and $n_2 = 4$ can be seen in Figure 3.1. These shapelets can be described as perturbations around an elliptical Gaussian (Refregier, 2003). The zero-order shapelet basis function is a 2D elliptical Gaussian with zero position angle.

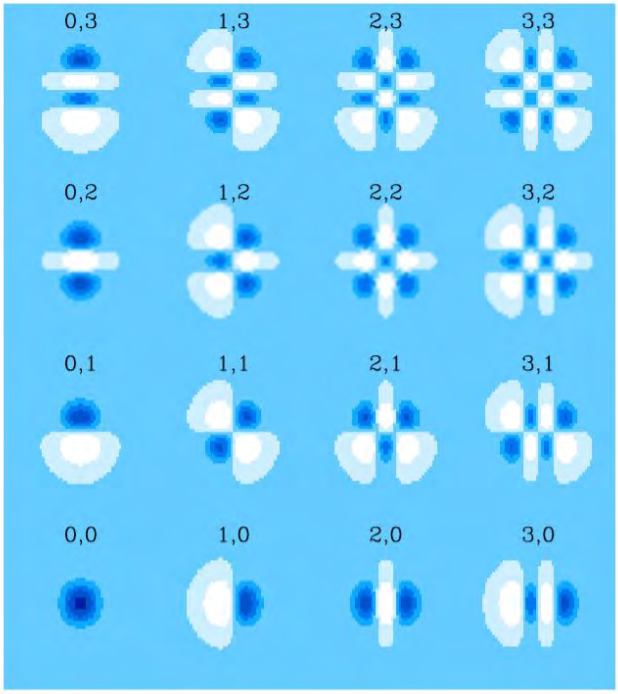


Figure 3.1: Shapelets ranging from order $0 \leq n_1, n_2 \leq 3$ (taken from Refregier, 2003).

3.3 Shapelet Implementation

This section outlines our implementation of shapelet basis functions for radio interferometric imaging and calibration. It begins with a description of the RIME in terms of the shapelet basis functions. Following this, we outline the unit tests performed on the implementation. Note that, in this work, we do not attempt to implement the decomposition (3.5). Instead, we focus on an efficient way to compute visibilities given shapelet coeffcients.

3.3.1 From Shapelets to Visibilities

Shapelets are invariant under Fourier transform, to a scaling factor (Refregier, 2003). We express the Fourier transform of a function as $\mathcal{F}\{\cdot\}$. In particular, the Fourier transform of a model expressed with shapelets in a single dimension is given as

$$\mathcal{F}\{B_n\}(u;\beta) = \imath^n B_n(u;\beta^{-1}). \tag{3.6}$$

Because the shapelet model is separable in l and m, the Fourier transform of a 2D model expressed using shapelets is

$$\mathcal{F}\{\mathbf{I}\}(u,v;\beta_l,\beta_m) = \sum_{n_1,n_2}^{N_{max}} \mathbf{f}_{n_1,n_2} \mathcal{F}\{B_{n_1}\}(u;\beta_l)\mathcal{F}\{B_{n_2}\}(v;\beta_m). \tag{3.7}$$

Therefore, according to Refregier, 2003, the analytic expression of this is

$$\mathcal{F}\{f\}(u,v;\beta_l,\beta_m) = \sum_{n_1,n_2}^{N_{max}} \imath^{n_1+n_2} \mathbf{f}_{n_1,n_2} B_{n_1}(u;\beta_l^{-1}) B_{n_2}(v;\beta_m^{-1}). \tag{3.8}$$

The invariance of shapelets under Fourier transform allows for fast and direct modelling of galaxy morphologies in visibility space. In conjunction with the RIME, we use this analytic expression to generate model visibilities.

While it is possible to model each of the four Stokes parameters with separate sets of shapelet coefficients, we only consider the Stokes I in this chapter. With this in mind, the continuous RIME for a set of sources corrupted by a primary beam according to Equation (1.18) is given as

$$\mathbf{V}_{pq}(u,v,w) = \int\int \mathbf{E}_p \mathbf{I} \mathbf{E}_q^H \mathbf{K}_{pq} \frac{dldm}{n}, \tag{3.9}$$

where $\mathbf{K}_{pq} = \exp\left(-2\pi\imath\frac{\nu}{c}[u_{pq}l + v_{pq}m + w_{pq}(n-1)]\right)$ denotes the scalar phase delay and $\mathbf{I}$ denotes the model image. The model image consists of a set of sources and each source is modelled as a set of basis functions multiplied by their respective coefficients i.e. $\mathbf{I} = \sum_s f_s(l-l_s, m-m_s)$ where l_s and m_s denote the location of a source labelled by s and the individual sources are modelled as

$$f_s = \sum_{n_1,n_2}^{N_{max}} \mathbf{f}_{n_1,n_2,s} B_{n_1}(l-l_s;\beta_l) B_{n_2}(m-m_s;\beta_m). \tag{3.10}$$

Substituting this into Equation (3.9), we get

$$\mathbf{V}_{pq}(u,v,w) = \sum_s \int\int \mathbf{E}_p(l,m) f_s(l-l_s, m-m_s) \mathbf{E}_q^H(l,m) \mathbf{K}_{pq} \frac{dldm}{n_s}. \tag{3.11}$$

For sources that are relatively small, we can assume the DDE to be constant across the extent of the source. As a result, the $\mathbf{E}$ terms remain constant and can move out of the integral. Applying this assumption, we get

$$\mathbf{V}_{pq}(u,v,w) = \sum_s \mathbf{E}_{p,s} \left(\int\int f_s(l-l_s, m-m_s) \mathbf{K}_{pq} \frac{dldm}{n_s} \right) \mathbf{E}_{q,s}^H, \tag{3.12}$$

where $E_{p,s} = E_p(l_s, m_s)$. Since the visibilities corresponding to each source can be computed independently, we can, without loss of generality, consider a single source at a time. Thus, for each source, we define $l' = l - l_s$ and $m' = m - m_s$. It follows that $l = l' + l_s$ and $m = m' + m_s$ and, since l_s and m_s remain constant, they can be separated from the integral. Additionally, we assume n_s to remain constant across the source (for a relatively small source), thereby allowing it to be removed from the integral as well. Subsequently, our $\mathbf{K}_{pq}$ term can be separated into two components viz. one that varies with the integral and one that remains constant. Reflecting this,

we get

$$\mathbf{V}_{pq}(u,v,w) = \sum_s \mathbf{E}_p W_s \left(\int \int f_s(l',m') \mathbf{K}'_{pq} dl' dm' \right) \mathbf{E}_q^H, \tag{3.13}$$

where $\mathbf{K}'_{pq} = \exp\left(-2\pi\imath \frac{\nu}{c}[ul' + vm']\right)$ is the portion of the $\mathbf{K}_{pq}$ term that varies with the integral. The portion of the $\mathbf{K}_{pq}$ term that remains constant across the integral is placed into W_s, which is referred to as the w-term and calculated as

$$W_s = \frac{e^{-2\pi\imath \frac{\nu}{c}[ul_s + vm_s + w(n_s - 1)]}}{n_s}. \tag{3.14}$$

Equation (3.13) provides a form of generating visibilities using the shapelet model by multiplying the w-term with $\int \int f(l',m') e^{-2\pi\imath \frac{\nu}{c}[ul' + vm']} dl' dm'$, which is the Fourier transform of the shapelet model where the coordinates in harmonic space are given by $\frac{\nu}{c}(u,v)$. These Fourier transforms can be evaluated analytically using Equation (3.8). The implementation of the shapelet model can be found in Appendix A. At the time of writing, this implementation exists as a pull request into the files of CODEX-AFRICANUS[1]. From this point forward, our implementation of the shapelet model existing in the files of CODEX-AFRICANUS will be referred to as the Shapelet modified Codex Africanus (ShaCA).

3.3.2 Testing

Testing was done dimension-by-dimension. Firstly, the 1-D case was tested in the Fourier domain. According to Refregier, 2003, shapelets in the 1D case amount to

$$f(l;\beta) = \sum_n^{N_{max}} \mathbf{f}_n \frac{H_n(\frac{l}{\beta}) e^{-\frac{l^2}{2\beta^2}}}{\sqrt{2^n \pi^{\frac{1}{2}} n! \beta}}, \tag{3.15}$$

and their Fourier counterparts amount to

$$\mathcal{F}\{f\}(u;\beta) = \sum_n^{N_{max}} \mathbf{f}_n \frac{H_n(u\beta) e^{-\frac{\beta^2 u^2}{2}}}{\sqrt{2^n \pi^{\frac{1}{2}} n! \beta^{-1}}} \imath^n. \tag{3.16}$$

The 1D unit test consisted of generating a regular l-grid in the image domain and populating it with the shapelet simulation in image space. We follow by generating a regular u-grid in the frequency domain and populating it with the analytic Fourier transform of these shapelets. Taking the Fast Fourier Transform (FFT) of the image domain shapelets[2], we should see a match to Equation (3.16). The unit test for this comparison is detailed in Appendix B.

[1] https://github.com/JoshVStaden/codex-africanus/pull/4

[2] Note that artefacts arising from the periodic boundary conditions of the FFT can be avoided either by padding the image or by making sure that the extent of the source is small compared to the size of the image.

The next unit test run on the implementation compares the shapelet image generated by our implementation to that of SHAPELETS[3], a software package that generates images using shapelets. Given the same coefficients and the same values of β_l and β_m, we should see the same values output by each respective implementation. The unit test comparing ShaCA against the SHAPELETS package is detailed in Appendix C.

The next unit test run on the implementation extends from the 1D case we performed in the Fourier domain to the 2D case. The Fourier transform of the shapelet model can be seen in Equation (3.8). The ground truth for this particular unit test is found by taking the Fast Fourier Transform (FFT) of the shapelets in the image domain. Second, we evaluate the analytic expression detailed in Equation (3.8) on a regular grid corresponding to the frequencies of the FFT. If the two forms of Fourier transform match, the test passes. This unit test is outlined in Appendix D.

We conclude testing on the shapelets by testing the w-term outlined in Equation (3.14). The w-term corrects for the phase of the model for sources off-phase-center. As discussed in Section 3.2, the zero-order shapelet equates to a Gaussian. Therefore, a zero-order shapelet placed at a distance from the phase center will produce equal visibilities to a Gaussian placed at the same location. Since there are existing packages to predict the visibilities of a Gaussian model component, we can test our implementation of the w-term against them. In particular, the Gaussian predict is also implemented in CODEX-AFRICANUS, which has in turn been verified against the MeqTrees[4] (Noordam and Smirnov, 2010) and Montblanc[5] (Perkins et al., 2015) packages.

From (3.8) we see that the zero order shapelet (i.e. $n_1 = n_2 = 0$) takes the form

$$f(\tilde{u}, \tilde{v}) = \frac{\mathbf{f}_{0,0}\sqrt{\beta_u \beta_v}}{\sqrt{\pi}} \exp\left(-\frac{1}{2}\beta_u^2 \tilde{u}^2 - \frac{1}{2}\beta_v^2 \tilde{v}^2\right), \tag{3.17}$$

where it should be kept in mind that, because of the frequency scaling of the harmonic coordinates, we have that $\tilde{u} = \frac{\nu}{c}u$ and $\tilde{v} = \frac{\nu}{c}v$. Now, for a zero position angle, the Gaussian model in CODEX-AFRICANUS is given by

$$f(\tilde{u}, \tilde{v}) = \exp\left(\frac{-2\pi^2}{\left(2\sqrt{2\log(2)}\right)^2}\left(e_{maj}^2 \tilde{u}^2 + e_{min}^2 \tilde{v}^2\right)\right) \tag{3.18}$$

for extent parameters e_{maj} and e_{min} describing the extent of the source as measured by the v and u axes respectively. For historical reasons the definition of a Gaussian source component in CODEX-AFRICANUS does not follow the usual parametrisation of Gaussian in terms of the standard deviation (or variance). Instead, the extent parameters correspond to the FWHM of the Gaussian. Thus, to convert to

[3] https://github.com/griffinfoster/shapelets
[4] https://github.com/ska-sa/meqtrees
[5] https://github.com/ska-sa/montblanc

the more standard parametrisation, we need to scale $\sigma_u = e_{min}/(2\sqrt{2\log 2})$ and $\sigma_v = e_{maj}/(2\sqrt{2\log 2})$. In this case, the above formula can be written as

$$f(\tilde{u},\tilde{v}) = \exp\left(-2\pi^2\left(\sigma_u^2\tilde{u}^2 + \sigma_v^2\tilde{v}^2\right)\right). \tag{3.19}$$

Assuming $\sigma_u = \sigma_v = \sigma$ and $\beta_l = \beta_m = \beta$, to get an equivalent σ value, one must input a β of

$$\beta = 2\pi\sigma. \tag{3.20}$$

If we assume $e_{maj} = e_{min} = e_m$, this can be expressed in terms of e_m as

$$\beta = \frac{\pi e_m}{\sqrt{2\log(2)}}. \tag{3.21}$$

To test the w-term, we make use of the the included PREDICT script[6] in CODEX-AFRICANUS. This script generates visibilities from a TIGGER[7] sky model and outputs them into the MODEL_DATA column in a measurement set.

We use the predict script to generate visibilities corresponding to a Gaussian source with a diameter of 10″. We place this source at an arbitrary position relative to phase center, in this case, at right ascension and declination 5° each. Subsequently, we use the inputted sky model to generate the equivalent Gaussian as specified in (3.21). This test can be found in Appendix E.

3.4 Example: Simulating a Galaxy Modelled with Shapelets

This section details a simulation of NGC 6251 using shapelets. We incorporate Equation (3.13) into the CODEX-AFRICANUS RIME. We then follow by using this modified RIME to generate a measurement set from a shapelet model of a galaxy. We begin with a description of the tools and technologies used in the simulation, and follow with a description of the sky model format we use to simulate the source.

To simulate such a source, we need

- A sky model (in this case, the shapelets),
- the CODEX-AFRICANUS RIME API,
- a measurement set, and
- an imaging tool.

We use the MEASUREMENT SET data format specified by CASA (McMullin, J. P. and Waters, B. and Schiebel, D. and Young, W. and Golap, 2007) to store interferometric data. Read and write operations on this format is supported through the

[6] https://github.com/ska-sa/codex-africanus/blob/master/africanus/rime/examples/predict.py
[7] https://github.com/ska-sa/tigger-lsm

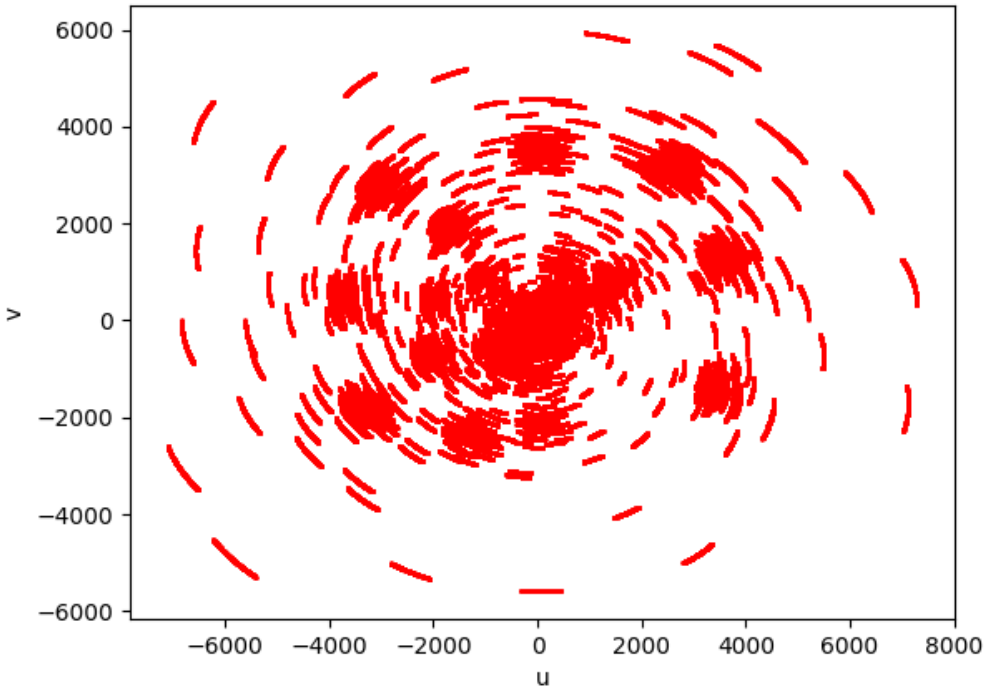


Figure 3.2: The *uv* coverage of the simulated MeerKAT observation.

PYRAP library and, more recently, through the DASK-MS library. The DASK-MS library provides a Data Access Layer to DASK. This provides DASK with the ability to perform distributed reads and computations.

We initialize an empty measurement set with MAKEMS,[8] and populate it using the RIME API in CODEX-AFRICANUS. The specifications of the measurement set are represented in a simple key-value format. We detail these specifications in Appendix F.

3.4.1 The Observation

The measurement set simulates 64 MeerKAT antennas observing the sky for a period of 80 minutes with a 10 second integration time. This resulted in 480 time steps in the observation. The antennas observed only within the 1085 MHz frequency with a bandwidth of 1 MHz. The center of the observation lay at right ascension 0° and declination -30°. The resulting *uv* tracks can be seen in Figure 3.2 and the corresponding PSF in Figure 3.3.

[8] https://github.com/ska-sa/makems

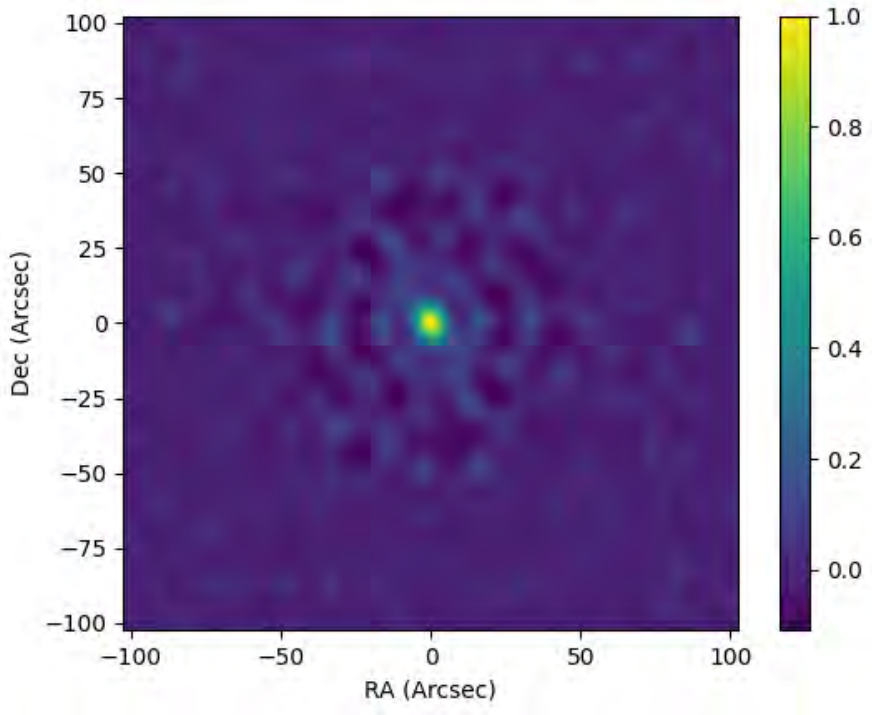


Figure 3.3: The PSF of the simulated MeerKAT observation.

3.4.2 Imaging and Deconvolution

We use WSCLEAN's single scale clean algorithm to recover a deconvolved image from the simulated visibilities. With WSCLEAN, one can specify the desired resolution and pixel scale. One can also specify the desired weighting scheme of the visibility samples. While one might choose either natural weighting to optimize instrument sensitivity or uniform weighting to optimize resolution, we have chosen Briggs (Briggs, 1995) weighting. This weighting scheme varies between natural and uniform weighting as a function of a single, real-valued robustness parameter. A robustness parameter valued at -2.0 gives a weighting that is close to uniform, and a robustness parameter of 2.0 gives a weighting scheme close to natural. We choose a robustness value of 0, as this gives a good tradeoff between sensitivity and resolution.

The command used to deconvolve and image the interferometric data is given as

```
1 wsclean -j 8 -weight briggs 0.0 -channels-out 1 -niter 100000 -nmiter 15
    ↪  -auto-threshold 3 -mgain 0.85 -padding 2.0 -size <pixels> <pixels
    ↪ > -scale <scale> -data-column MODEL_DATA /path/to/MeasurementSet.
    ↪ MS/
```

where the pixel and scale values are set accordingly.

3.4.3 The Sky Model

The format chosen to specify our sky model is the TIGGER-LSM[9] format. For a set of sources, this format allows us to specify the

- source name,
- source position,
- spectral index,
- reference frequency,
- Stokes parameters, and
- shape parameters.

Shape parameters may differ between models, and the parameters given infers the model. For example, an elliptical Gaussian would require e_{maj} and e_{min} parameters so including these parameters in the TIGGER-LSM file without specifying shapelet coefficients would infer an elliptical Gaussian. The e_{maj} and e_{min} parameters can also be used to describe the extent of a shapelet. To convert these parameters to their respective β values, one must use the conversion specified in Equation (3.21). The only extra parameter needed by a shapelet source model is the coefficient matrix. If the TIGGER-LSM parser scans the SHAPELET_COEFFS keyword in the format line, it will infer a shapelet model. An example of a shapelet modelled in this format would be

```
1 #format: name ra_d dec_d i spi freq0 emaj_s emin_s shapelet_coeffs
2 J0 0.00 -30.00 10.00 -0.700000 1085000000 33.333 80.0
    ↪ 1.9988189101699798123, ...
```

This particular code snippet describes a single source labelled J0, at right ascension 0.00 degrees and declination -30.00 degrees. The source is not polarized and has a total flux density of 10 Jy at a frequency of 1085 MHz, which scales with frequency according to a power law. The source has a length of 80″ and a width of 33.333″. The shape of the source can be described with a set of comma-separated coefficients. For multiple shapelet coefficients in a comma-separated list c_n for $n = 0, 1, 2, 3, 4, ...$, the coefficients will be read into a 2-dimensional grid as

$$\begin{matrix} c_0 & c_2 & c_5 & c_9 \\ c_3 & c_7 & & \\ c_6 & & & \end{matrix}$$

Because we truncate our shapelet expansion at a fixed order N_{max}, this results in a triangular matrix. The coefficient grid for NGC 6251 extracted from the SHAPELETS

[9] https://github.com/ska-sa/tigger-lsm

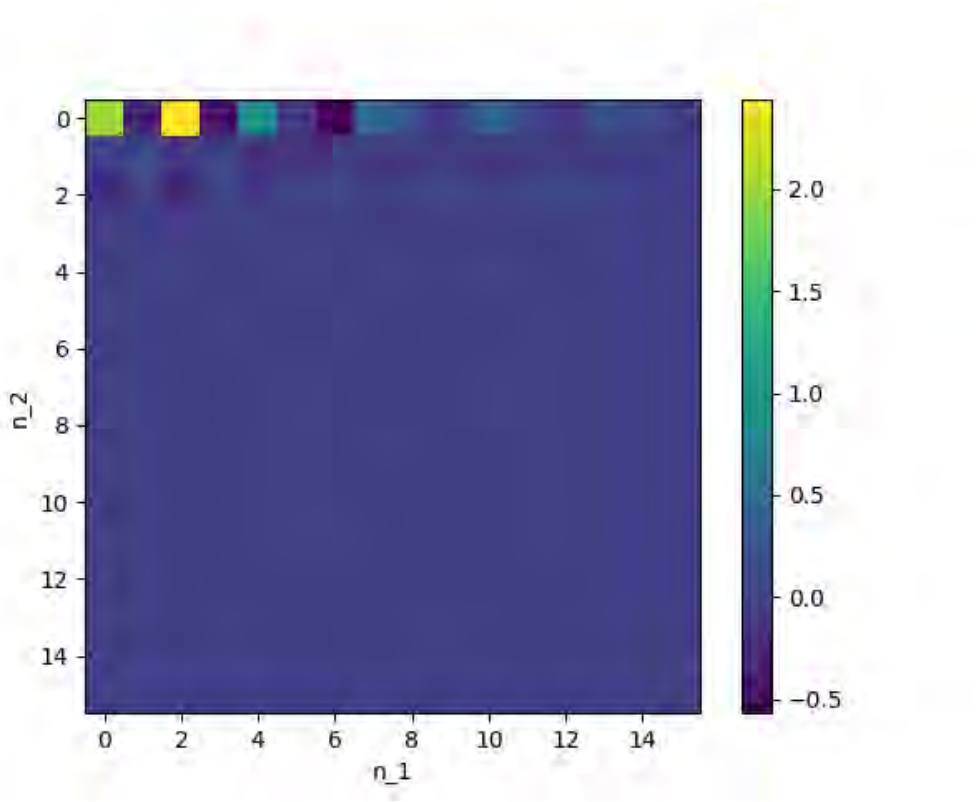


Figure 3.4: The shapelet coefficient matrix of NGC 6251. Coefficients extracted from the SHAPELETS software package.

software package[10] can be seen in Figure 3.4. The predict script that generates visibilities from this sky model and populates the measurement set is detailed in Appendix G. An expansion of the coefficients in Figure 3.4 on a regular grid can be seen in Figure 3.5, and the dirty image obtained with WSCLEAN can be seen in Figure 3.6. We make the assumption that the central pixel of the image was aligned with the center of the galaxy during shapelet decomposition.

3.5 Comparison to Nifty Gridder

This section presents a comparison of the utility of ShaCA to that of the NIFTY-GRIDDER (Reinecke, Steininger, and Selig, 2018).

The NIFTY-GRIDDER is a recently developed software package that grids visibilities into an image and degrids an image into a set of visibilities according to the van Cittert-Zernike theorem (Reinecke, Steininger, and Selig, 2018; Thompson, Moran, and Swenson Jr, 2008). These operations can be performed to an arbitrary accuracy to a maximum of around 10^{-12} (Reinecke, Steininger, and Selig, 2018), making the NIFTY-GRIDDER a suitable reference implementation to test our shapelets against.

[10] https://github.com/griffinfoster/shapelets

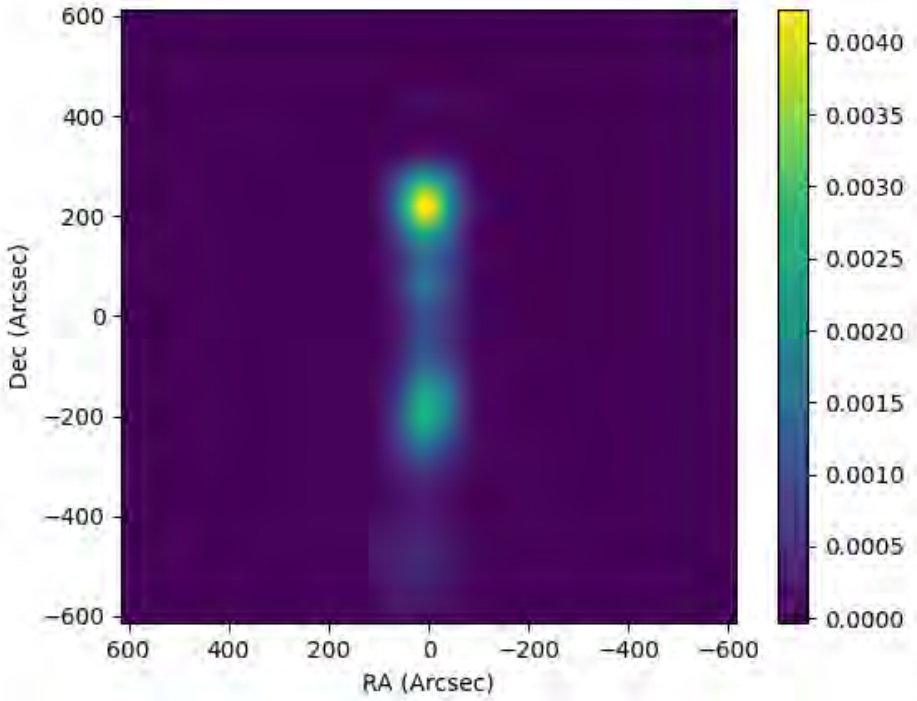


Figure 3.5: NGC 6251 modelled directly in the image domain using the 136 shapelets from the SHAPELET software package.

From the outset, we expect the NIFTY-GRIDDER to have better runtimes and scaling than ShaCA. We argue, however, that the NIFTY-GRIDDER has difficulty in accounting for DDEs while ShaCA does not. This section therefore gives an indication of the cost in runtime of using ShaCA to account for DDEs over the NIFTY-GRIDDER.

We begin by detailing the design of the experiment. The computational complexity of the two implementations are analyzed after the description of the experiment design. We analyze the results of this experiments with two metrics: the RMS error between the models and the runtimes of the two implementations.

3.5.1 Experiment Design

We use the shapelet model given in Section 3.4.3 as a form of testing. For ShaCA, we vary the number of shapelet components by simply taking the n largest components and varying n. We run this up to 70 components. Because the runtime of the NIFTY-GRIDDER does not change with respect to n, we simply de-grid the full image and compare that time across all n. The image of NGC 6251 used consists of 1024×1024 pixels, which is de-gridded to a precision of 10^{-10}. We run both the NIFTY-GRIDDER and ShaCA with 16 threads.

For storing the simulated visibility data, we use the measurement set specified in Section 3.4.1; however, we use a different number of integration times. Our first measurement set is generated from a simulated observation with an integration time

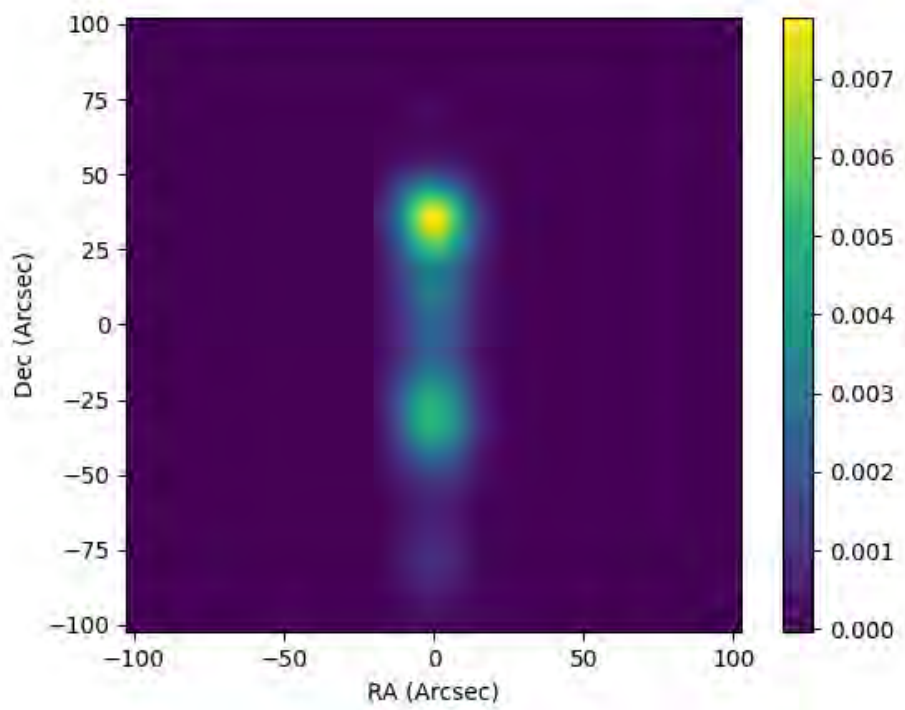


Figure 3.6: The restored image after modelling visibilities corresponding to an observation of NGC 6251 and cleaning them using WSCLEAN.

of 10 seconds. The second and third observations have the same integration times, but take place over two hours and three hours, respectively.

3.5.2 Model Complexities

The time complexity of an FFT is $\mathbf{P}\log_2(\mathbf{P})$ for an image consisting of $\mathbf{P}$ pixels. If the support of the gridding kernel is denoted as $\mathbf{K}$ and the number of w-stacks is denoted as $\mathbf{W}$, the time complexity of the NIFTY-GRIDDER is $\mathcal{O}(\mathbf{KRF} + \mathbf{WP}\log_2(\mathbf{P}))$, where $\mathbf{R}$ denotes the number of rows and $\mathbf{F}$ denotes the number of frequency channels that are being combined into a single image.

The space complexity of an FFT scales similarly to its time complexity. Therefore, for a computation involving degridding an image consisting of $\mathbf{P}$ pixels to $\mathbf{R}$ visibilities and $\mathbf{F}$ frequency channels, the space complexity of the NIFTY-GRIDDER is $\mathcal{O}(\mathbf{RF} + \mathbf{P}\log(\mathbf{P}))$ where we note that the gridder has been designed so that the space complexity does not scale with the number of w-stacks (Arras et al., 2019).

ShaCA makes $\mathbf{RFSN}$ iterations through the data, where $\mathbf{R}$ and $\mathbf{F}$ has the same meaning as above and $\mathbf{S}$ is the number of sources with $\mathbf{N}$ shapelet coefficients. The time complexity of ShaCA is therefore $\mathcal{O}(\mathbf{RFSN})$.

As well as the time complexity of the algorithm, we want to analyze the space complexity of the implementation. ShaCA takes 6 inputs, namely

- COORDS of shape ($\mathbf{R}$, 3),
- FREQUENCY of shape ($\mathbf{F}$,),
- COEFFS of shape ($\mathbf{S}$, $\mathbf{N}_1$, $\mathbf{N}_2$),
- BETA of shape ($\mathbf{S}$, 2),
- DELTA_LM of shape (2,), and
- LM of shape ($\mathbf{S}$,2),

where we denote the two maximum shapelet orders as $\mathbf{N}_1$ and $\mathbf{N}_2$ to give a total number of $\frac{\mathbf{N}_1\mathbf{N}_2}{2} = \mathbf{N}$ coefficients. The output shape of the implementation is ($\mathbf{R}$, $\mathbf{F}$, $\mathbf{S}$). The code simply performs a series of arithmetic operations on each element in its input data and writes to its output data. While there is a recursive call in the code that computes Hermite polynomials, this recursive call scales linearly with $\mathbf{N}$. Because there is no additional memory scaling, the space complexity of the implementation, like its time complexity, is $\mathcal{O}(\mathbf{RFSN})$.

3.5.3 Model RMS

We compute RMS error between the shapelet model S and the nifty model N as

$$RMS = \sqrt{\frac{\sum_{r=0}^{R}(\mathbf{S}(r) - \mathbf{N}(r))^*(\mathbf{S}(r) - \mathbf{N}(r))}{R}}, \tag{3.22}$$

where $S(r)$ denotes the r^{th} visibility for model S and S and N have a total of R visibilities.

Figure 3.7 illustrates the RMS error between the visibilities generated by the nifty and shapelet models. This varies as the number of shapelet components increase. Note that components are added to the model in descending order of their magnitudes. Additionally, for comparison, we show the RMS of the full shapelet model. Figures 3.7a, 3.7b and 3.7c demonstrate this result for the one, two and three hour observations, respectively.

The full-component shapelet model gives an RMS error of roughly 4×10^{-8}, 9×10^{-9} and 2×10^{-9} for the one, two and three hour observations, respectively. This roughly matches the accuracy of the NIFTY-GRIDDER which was set to 10^{-10}.[11] For all 3 graphs, we see the RMS of the partial shapelet models closely matching the RMS of the full shapelet model at roughly 60 components. At 60 components, the models give RMS errors of roughly the same as that output from the full-component shapelets.

Interestingly, we also see that the shapelet model with very few coefficients actually provides a better fit to NGC 6251's morphology than the full shapelet model. We can attribute this to the fact that NGC 6251 has a relatively simplistic morphology and therefore relatively few components are needed to get sufficient accuracy. To exploit this feature in practice, however, we would have to know which basis functions to use for the fit at the outset. Also note that, for galaxies with more complex morphologies, it is likely that many more components would be required to model them with sufficient accuracy. Figure 3.8 illustrates two other galaxies decomposed into shapelet coefficients, NGC 0383 and NGC 1365 (Bastien, Oozeer, and Somanah, 2016). The coefficients for NGC 0383 are compact, with most of them existing within the first few components. This is reflective of the simple morphology of the galaxy. However, the coefficient grid for NGC 1365 is more spread out, and subsequently it can be seen that more components are needed to model this galaxy. In both cases, the galaxies were modelled with 256 components in total.

3.5.4 Model Timings

Figure 3.9 illustrates the results of timing the two implementations against each other. The horizontal lines denote the timings of the NIFTY-GRIDDER which, as previously mentioned, does not vary with n. It does, however, increase its runtime with the number of visibilities generated. The NIFTY-GRIDDER soon outperforms ShaCA at roughly 30 components. While this would give sufficient accuracy for the current comparison, it is not clear how many components would be required for galaxies with more complex morphologies.

[11] Note that this factor is the rms error between the actual visibilities computed using the Discrete Fourier Transform (DFT) and those of the nifty griddder. We are not guaranteed that each visibility is accurate up to this factor which might explain the above discrepancy.

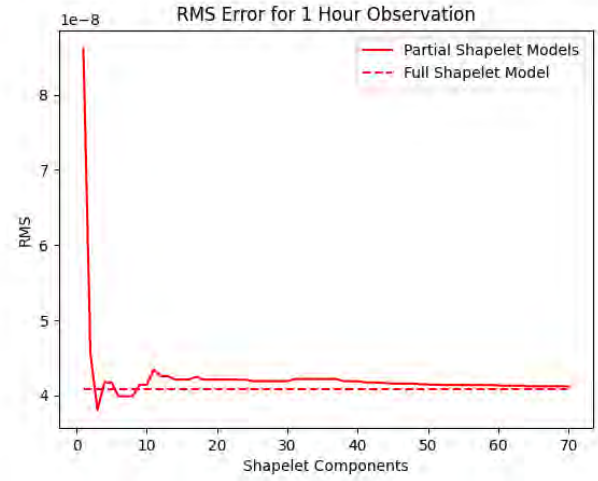
While the NIFTY implementation has a faster runtime and does not scale with source model complexity, it is not able to apply non-trivial DDEs (i.e. DDEs that can vary with time, frequency and antenna) to sources. This is largely due to the fact that it degrids the same image to all baselines assuming a common gridding kernel. Because the shapelets compute models on a source-by-source basis, and we have an analytic expression for the visibilities, it is able to apply gains to individual sources, given that they are small enough so that our above approximation holds. Thus, ShaCA could be used for observations involving multiple bright compact sources with DDEs corrupting them.

3.6 Conclusion

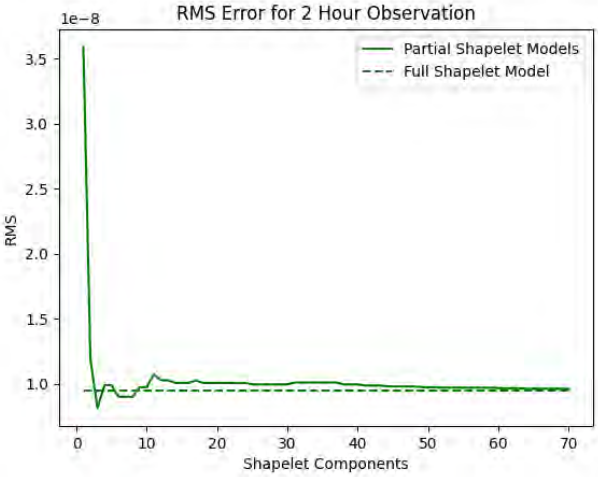
This chapter has presented an implementation of a form of galaxy morphology modelling using Gauss-Hermite basis functions, also referred to as shapelets. This model is fully parametric and offers a simple analytic expression under Fourier transform, allowing for direct modelling in the Fourier domain.

Furthermore, the chapter presents unit tests on our implementation. The first of these unit tests proves the correctness of our model in frequency space in one dimension. The second unit test begins testing in two dimensions in image space and the third test extends this into the Fourier domain. The final test examines the w-term component which is necessary to accurately model sources far from phase center.

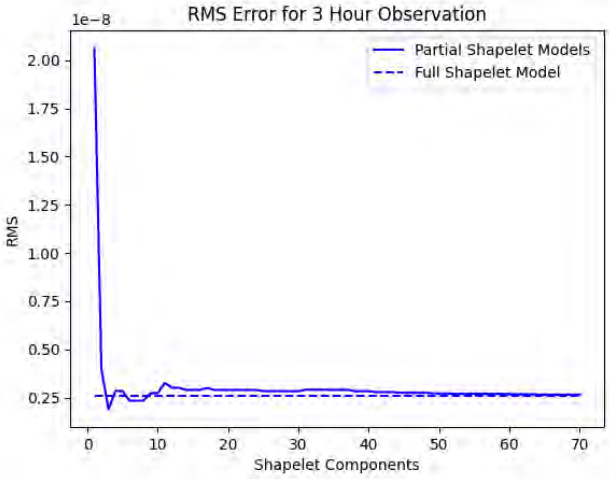
We use a model of NGC 6251 to simulate visibilities as a proof-of-concept of the implemented shapelets. Using this simulation, we compare ShaCA to that of NIFTY-GRIDDER. We find that the NIFTY-GRIDDER runs faster than our implementation. However, the benefits of ShaCA over the NIFTY-GRIDDER is that, as we explain in the next chapter, it is possible to apply DDEs to individual sources. Additionally, the implementation is fully parametric, which allows support for parameterised solvers.



(a) The RMS error for the 1 hour observation.



(b) The RMS error for the 2 hour observation.



(c) The RMS error for the 3 hour observation.

Figure 3.7: The RMS error between the shapelet and NIFTY implementations for (a) the 1 hour observation, (b) the 2 hour observation and (c) the 3 hour observation.

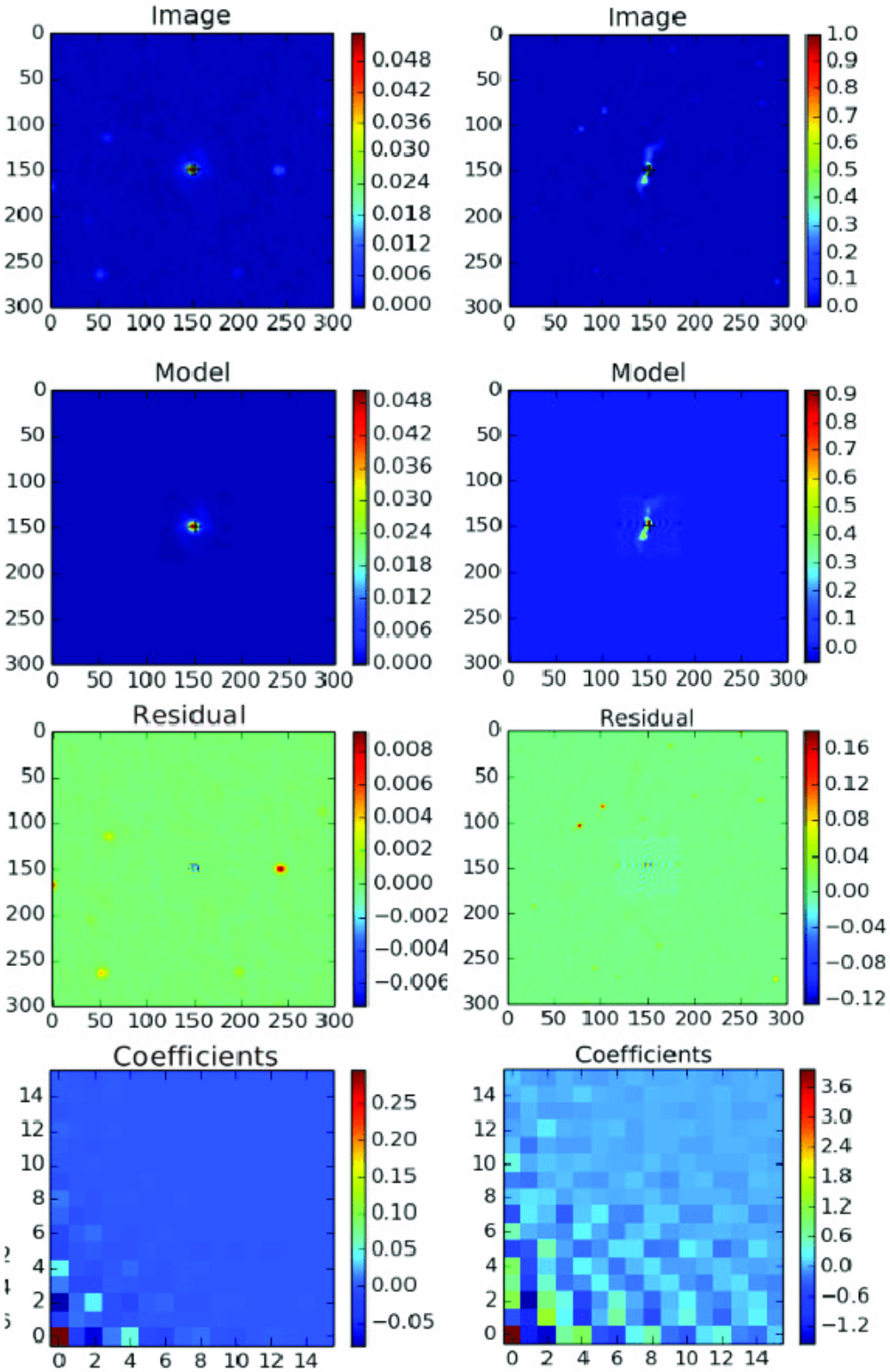


Figure 3.8: The decomposition of NGC 0383 (left) and NGC 1365 (right) into shapelet coefficients. Taken from (Bastien, Oozeer, and Somanah, 2016).

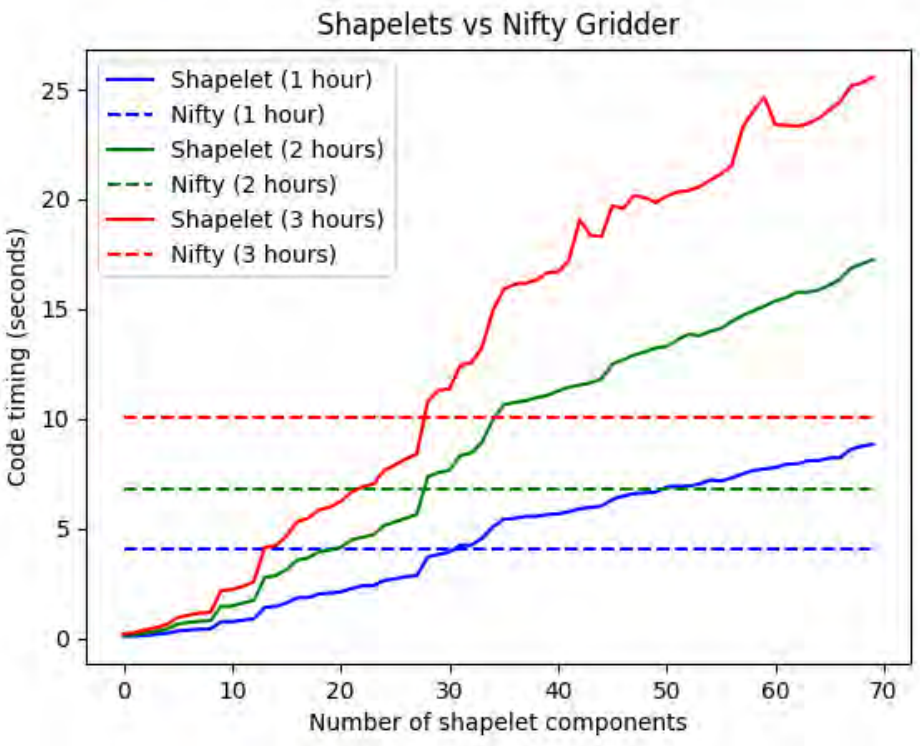


Figure 3.9: The runtime of ShaCA as it varies with the number of shapelet components as compared to the runtime of the NIFTY-GRIDDER. These runtimes also vary with the number of rows.

Chapter 4

Modelling Primary Beam Patterns using Zernike Polynomials

4.1 Chapter Outline

Zernike polynomials are polynomials used in the field of optics to describe aberrations around a unit sphere (Zernike, 1934; Born and Wolf, 1964). In the field of radio interferometry, these polynomials can be used to model the antenna primary beam (Iheanetu, 2019).

This chapter presents an accelerated implementation of Zernike polynomials for inclusion into CODEX-AFRICANUS. We begin by defining the polynomials mathematically and follow by detailing the implementation. After detailing the implementation, we provide a benchmark of our code against a multi-threaded NUMPY implementation of the Zernike polynomials. We test the accuracy of the beam's corruption by measuring the RMS error between our Zernike implementation and the currently implemented form of DDE modelling in CODEX-AFRICANUS viz. by supplying and interpolating a FITS beam image. It is important to note that, while Chapter 3 only considers the Stokes I, this chapter investigates the beam at all four correlations. As a result, we can examine the leakage introduced by the beam model. Following the testing procedure on the accuracy of the beam, we provide a proof-of-concept by imaging two sets of sources corrupted by the primary beam. Firstly, we image two point sources corrupted by the beam. This gives us an indication of the effects of the primary beam on the total flux density of the corrupted sources in the final image. Secondly, we combine the source and beam modelling formalism developed in this thesis to produce primary beam corrupted visibilities of NGC 6251.

4.2 Zernike Polynomials Definition

For linear feeds, we define the E-Jones term as a 2×2 matrix, with each component containing its own set of coefficients

$$\mathbf{E} = \begin{pmatrix} e_{xx}(l,m) & e_{xy}(l,m) \\ e_{yx}(l,m) & e_{yy}(l,m) \end{pmatrix}. \tag{4.1}$$

Each element can be modelled as a Zernike polynomial with coefficients describing the beam eg.

$$e_{xx}(l,m) = \sum_{n,m} c^m_{xx,n} Z^m_n(r,\theta), \tag{4.2}$$

where r and θ describe the radial conversion of the lm coordinates. The Zernike polynomial term, $Z^m_n(r,\theta)$ can be written as (Zernike, 1934)

$$Z^m_n(r,\theta) \pm \imath Z^{-m}_n(r,\theta) = R^m_n(r) e^{\pm \imath m\theta}, \tag{4.3}$$

therefore giving the following set of equations:

$$Z^m_n(r,\theta) = R^m_n(r)\cos(m\theta)$$

for $m \geq 0$ and

$$Z^{-m}_n(r,\theta) = R^m_n(r)\sin(m\theta)$$

for $m < 0$.

The radial function $R^m_n(r,\theta)$ is given by (Zernike, 1934)

$$R^m_n(r,\theta) = \sum_{k=0}^{\frac{n-m}{2}} \frac{(-1)^k (n-k)!}{k![\frac{1}{2}(n+m)-k]![\frac{1}{2}(n-m)-k]!} r^{n-2k}.$$

Table 4.1: Noll indexing scheme for Zernike polynomials.

	m								
	-4	-3	-2	-1	0	1	2	3	4
n									
0					j=0				
1				j=1		j=2			
2			j=3		j=4		j=5		
3		j=6		j=7		j=8		j=9	
4	j=10		j=11		j=12		j=13		j=14

Indexing for the Zernike polynomials requires two parameters: the order of the polynomial n and the angular frequency m. Noll, 1976 presents a single-number

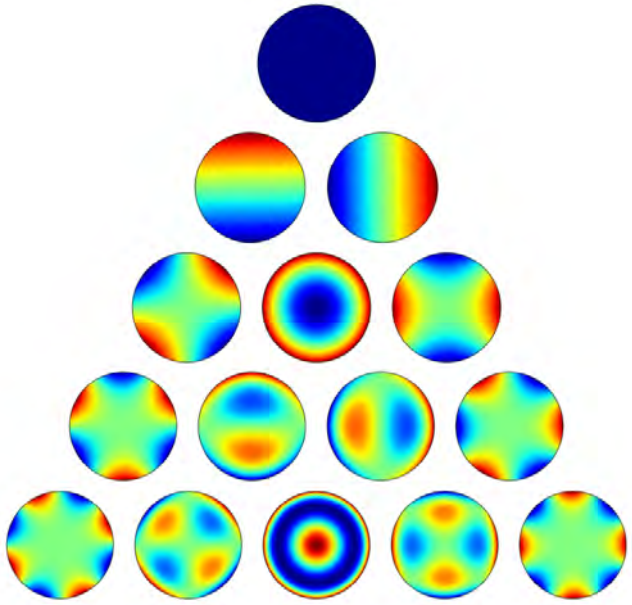


Figure 4.1: Zernike polynomials up to the first 5 orders of n. Taken from (Cook, 1976).

indexing system for Zernike polynomials, referred to as the Noll index of the polynomial. The number is typically represented as j and is calculated as shown in Table 4.1. The Zernike polynomials are even when $m \geq 0$ and odd when $m < 0$.

Figure 4.1 shows these Zernike polynomials up to 5 orders. The first three orders ($n = 0, 1, 2$) are referred to as low order aberrations and orders 3 and up are known as higher-order aberrations (Zernike, 1934).

4.3 Implementing the MeerKAT Primary Beam Model

Zernike polynomials give a fully parametric basis for modelling the antenna primary beam. This section begins by outlining the modelling of the primary beam within the RIME. It follows with a structure for storing the beam coefficients. Further, it follows with an explanation of the beam model used in EIDOS (Asad et al., 2019) and an illustration of the beam.

4.3.1 Implementation

The discrete RIME for a set of sources corrupted by the E-Jones term is given as

$$\mathbf{V}_{pq} = \sum_{s} \mathbf{E}_{ps} \mathbf{X}_{pqs} \mathbf{E}_{qs}^{H}, \tag{4.4}$$

where $\mathbf{X}_{pqs}$ denotes the per-source coherency matrix. $\mathbf{X}_{pqs}$ amounts to the bracketed integral in Equation (3.12).

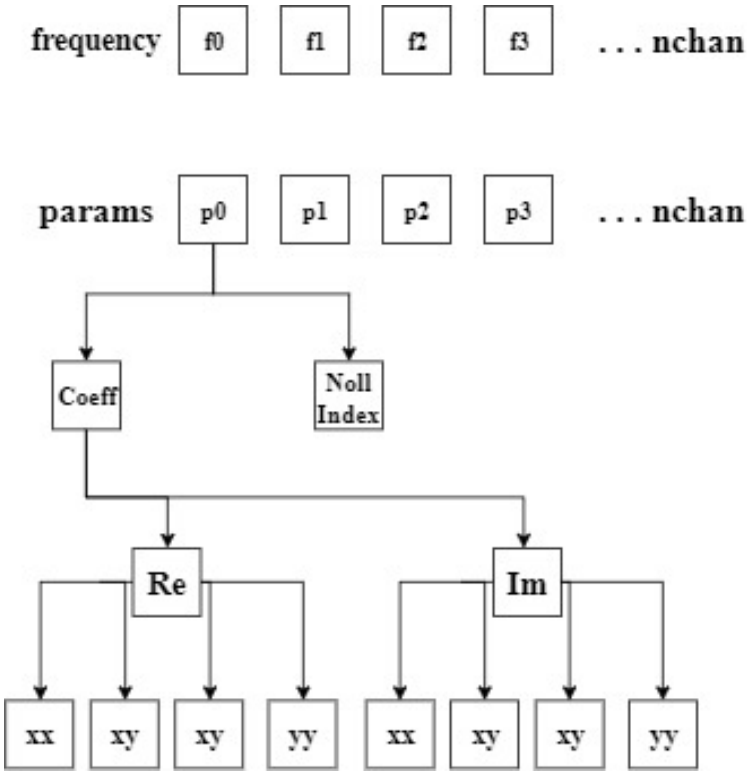


Figure 4.2: Layout of beam model structure. Frequencies in the frequency array are index-matched to parameters in the params array. Each element in the param array yields a *coeffs* array and a Noll index array. Each of these yields Noll indices and coefficients with real and imaginary parts describing each of the four Jones elements in Equation (4.1).

The E-Jones has a dependence on direction. Additionally, each term in Equation (4.4) is a 2×2 Jones matrix and therefore has 4 correlations. In the case of the coherency matrix, each correlation describes the polarization state of the source. The Jones matrices around the coherency matrix describe how the polarization state of the source is modified by that particular effect.

The primary beam varies in frequency and differs per-antenna. As a result, a different set of coefficients must be described for each of these factors. We wish to store our beam model in the NUMPY compressed array format. A separate beam model must be stored for the real and imaginary components of the beam. To select the beam for all antennas at a specific frequency, we also wish to store a frequency map with the beam model. Figure 4.2 illustrates this beam layout. It must be noted that, despite the fact that the implementation is able to expand coefficients per-antenna, our file format does not contain different coefficients per-antenna. This is because the beam model only uses a single set of coefficients to describe the common beam across all antennas before known per-antenna effects are taken into account. These effects, which we describe below, are applied on the fly for each antenna separately.

Because the antenna is susceptible to pointing errors, antenna scaling, frequency scaling and parallactic angle rotation during the course of the observation, this has

a direct effect on the beam. These effects manifest as simple coordinate transforms or a scaling of the amplitude of the beam. As a result, we add inputs to correct for these into the beam simulation. This is similarly implemented in the beam cube in CODEX-AFRICANUS.[1]

4.3.2 Beam Model

Our implementation relies on the beam modelling strategy of (Asad et al., 2019) which is made available through the EIDOS[2] software package. EIDOS fits Zernike polynomials to holographic measurements of the primary beam. In particular, it models the MeerKAT primary beam at L-band. As in the previous chapter, our implementation is not geared towards the acceleration of the fitting procedure. Rather, we take the coefficients produced by EIDOS as given and attempt to accelerate the implementation of a primary beam pattern described by these coefficients into the RIME. Since this computation typically has to be performed multiple times, it is important to have a fast method to evaluate the Zernike models.

EIDOS uses a Discrete Cosine Transform (DCT) to denoise and compress Zernike coefficients along the frequency axis. We will not delve any further into these details here. Instead, we simply work with the coefficients obtained when taking the inverse DCT of the compressed coefficient file produced by EIDOS. These coefficients are stored at each frequency channel. In total, 8001 channels were modelled, ranging from 870 MHz to 1670 MHz. Every channel has a total of 160 coefficients. These coefficients are used per correlation, making 40 coefficients per correlation. 20 of those 40 are used for the real component of the beam and the remaining 20 are used for the imaginary component.

Figure 4.3 illustrates the real parts of the four Jones elements of the simulated beam constructed at 1085 MHz using CODEX-AFRICANUS. We verify our implementation of the Zernike model by comparing each of the four correlations to the values computed by EIDOS directly. Appendix H shows the implementation of the Zernike polynomials into CODEX-AFRICANUS. Appendix I illustrates one of the unit tests for testing the beam.

Figure 4.4 illustrates the beam power across the field of view at 1085 MHz. The beam power is calculated as

$$\frac{e_{xx}e^*_{xx} + e_{yy}e^*_{yy}}{2}, \tag{4.5}$$

and represents the degree to which the beam attenuates the Stokes I of the corrupted source. It reaches 0.5 at 39.6 arcminutes from the center.

[1] https://codex-africanus.readthedocs.io/en/latest/rime-api.html#africanus.rime.beam_cube_dde
[2] https://github.com/ratt-ru/eidos

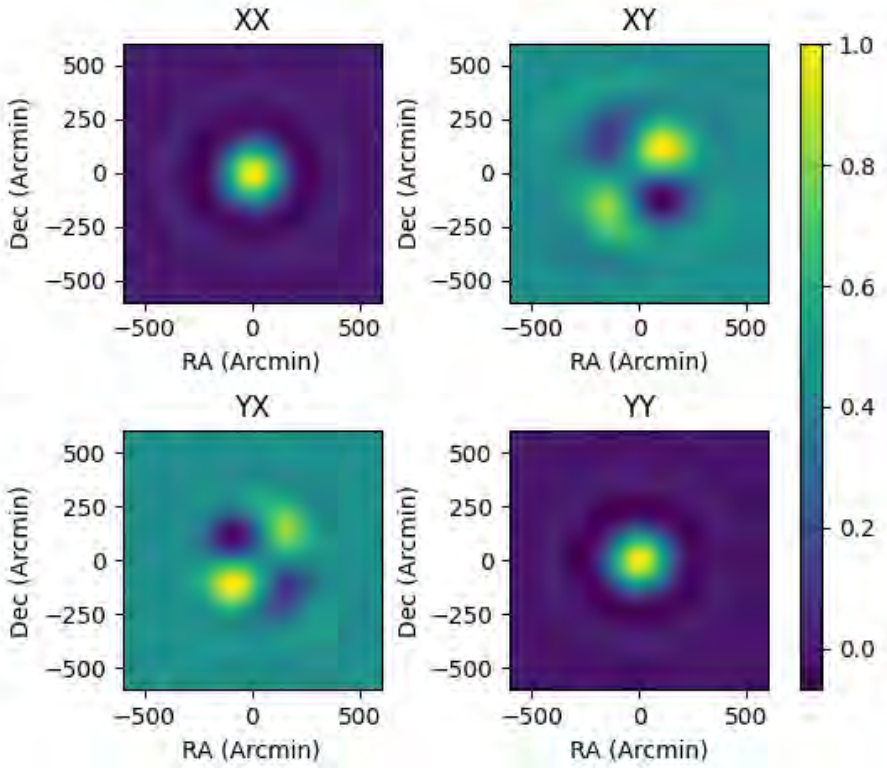


Figure 4.3: The MeerKAT primary beam simulated at 1085 MHz at all 4 correlations using the Zernike implementation.

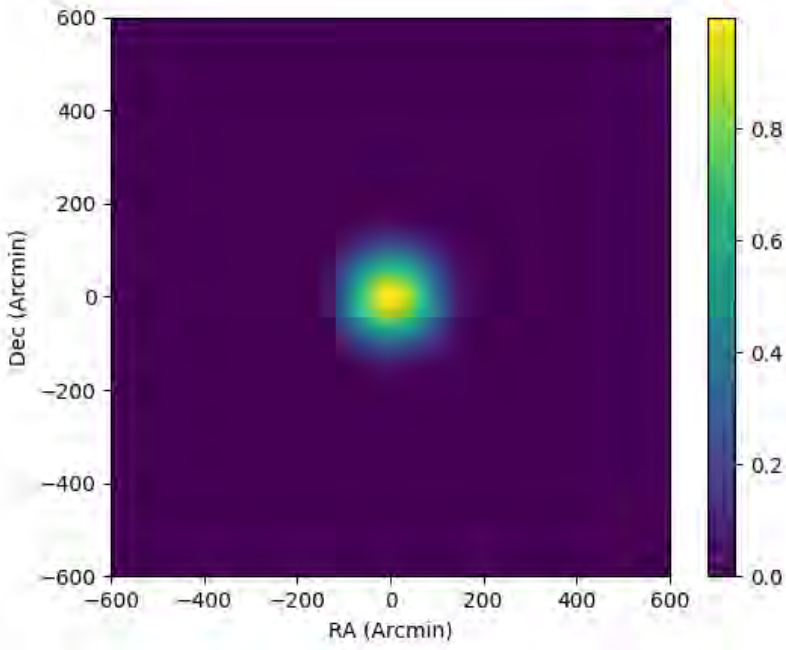


Figure 4.4: The computed power of the simulated MeerKAT primary beam.

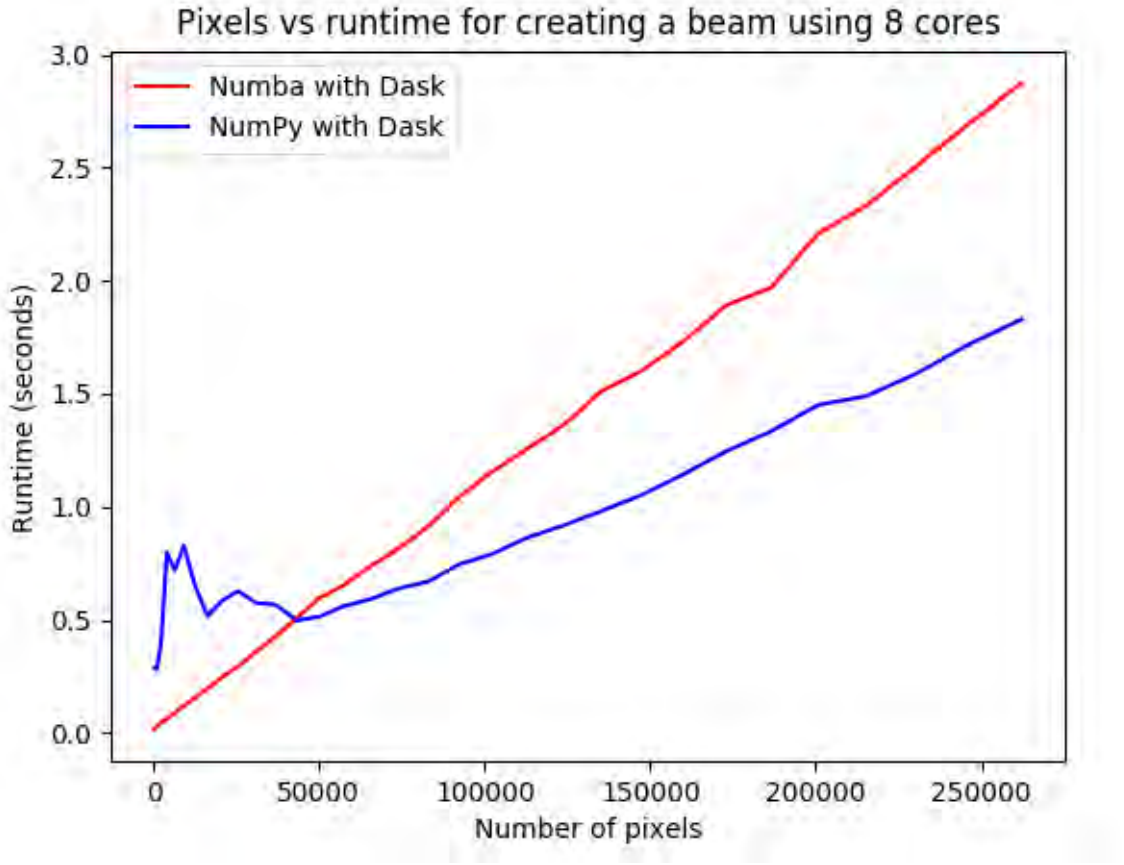


Figure 4.5: A comparison of the NUMBA implementation of the beam and the NUMPY version of the beam as the number of pixels increase.

4.3.3 Benchmarking

For fast modelling of the primary beam, we wish to create an accelerated model. To compare the degree to which our implementation of the Zernike polynomials are accelerated, we benchmark our code against a similar implementation using NUMPY in place of NUMBA. To benchmark this, we reconstruct the MeerKAT primary beam on a regular over a single frequency. For the first test, we vary the number of pixels in the image and assume 8 CPU cores. For the second test, we vary the number of cores and assume a default grid of 256×256 pixels. In this implementation, the MeerKAT primary beam was reconstructed using 20 Zernike polynomials each for the real and imaginary components.

In both cases, we use a similar chunking strategy. We chunk according to the *source* dimension. Because the beam is evaluated on a grid, the *source* dimension will be the largest, consisting of the total number of pixels.

We compare the runtimes of the two implementations with respect to two factors: the number of pixels in the image and the number of cores used. Figure 4.5 illustrates the runtime of the implementations as the beam increases with pixels, and similarly

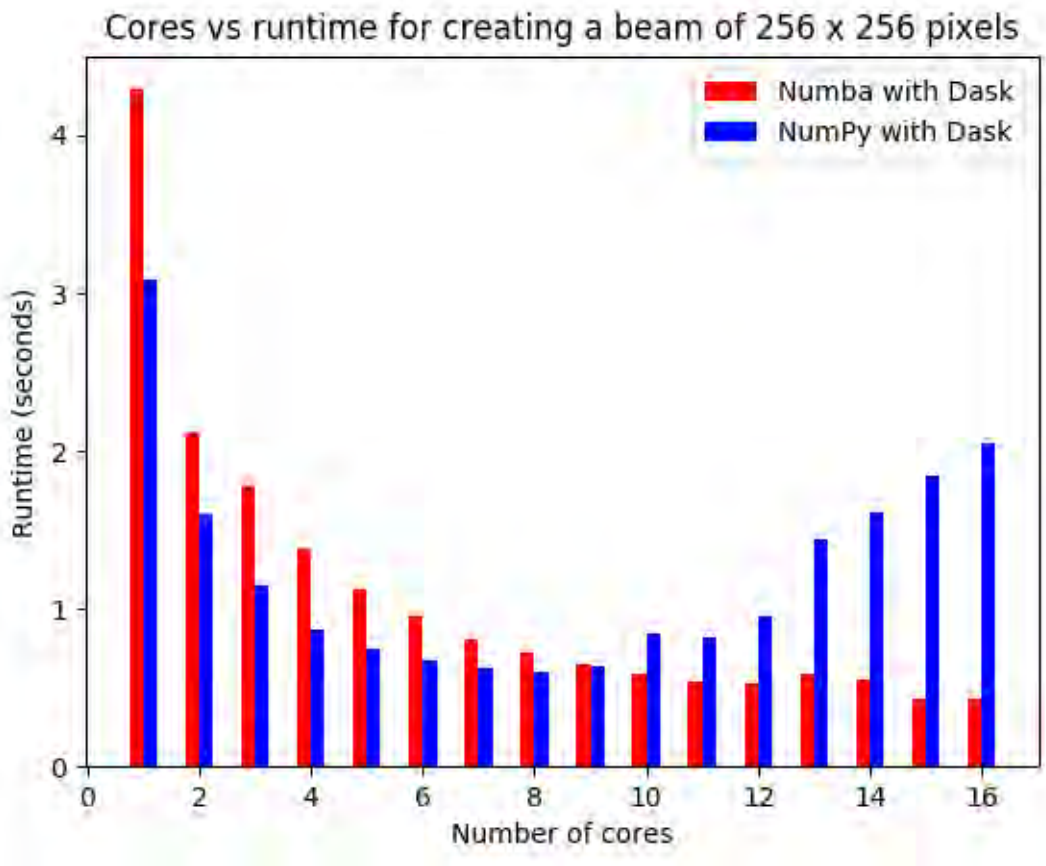


Figure 4.6: A comparison of the NUMBA implementation of the beam and the NUMPY version of the beam as the number of CPU cores increase.

Figure 4.6 for an increasing number of cores.

At around 128 pixels in the image, the NUMBA implementation runs better than the NUMPY implementation on 8 cores. This can be attributed to the fact that NUMBA runs similarly to highly optimized C code. Therefore smaller inputs will be processed faster than NUMPY, as NUMPY will have more overheads than NUMBA. However, the NUMPY implementation soon overtakes the NUMBA implementations. As inputs become larger, the overheads affecting NUMPY become smaller in comparison. NUMPY is also vectorized, and so it is optimized for larger operations on a single core. This is in comparison to NUMBA, which simply iterates through the array.

When running on a single core, the NUMPY implementation outperforms the NUMBA implementation. Again, this can be attributed to the vectorized nature of NUMPY. However, as more cores are added, we see NUMBA outperform NUMPY. Presumably, this is due to the fact that each core is doing less, and therefore the NUMPY overheads cause a longer runtime. From these graphs, we can make two conclusions: the NUMPY implementation performs poorly when underworked, and the NUMBA implementation performs poorly when overworked.

4.4 The Accuracy of the Zernike Implementation

This section verifies the accuracy of the Zernike polynomial implementation against the current form of evaluating generic DDEs in CODEX-AFRICANUS. This form of evaluation involves the specification of a DDE as a FITS cube with two spatial dimensions and one frequency dimension. In this case, the specified DDE is the MeerKAT primary beam. To evaluate the beam, one must specify the extent of the beam, a map of frequencies and the desired $(l, m, frequency)$ coordinate. This method linearly interpolates the beam values along the spatial dimensions and along the frequency axis. From this point forward, we will refer to this method as the FITS-interpolated method. Additionally, we provide a comparison of the computational complexities of the two implementations.

To test the accuracy of the Zernike model, we test the various implementations' ability to corrupt an elliptical Gaussian extended source. This source is simply modelled on a regular u-grid. We investigate their respective effects on the visibility amplitudes at each correlation, assuming a linear feed.

4.4.1 Model Complexity

The FITS-interpolated method begins by generating an updated array of frequency interpolation data. This data includes the lower frequency coordinate, scaling data and the lower weight. To achieve this, the script must locate each target frequency in the beam frequency map. Because the beam frequency map is sorted from lowest to highest, the script can make use of a binary search tree to locate this. If we denote

the range of frequencies in the beam frequency map as **B**, a binary search results in a scaling of $\mathcal{O}(\mathbf{F}\log_2(\mathbf{B}))$ for **F** target frequencies.

The FITS-interpolated implementation corrects for the same antenna errors as mentioned in Section 4.3.1. This adds computational complexity for every time interval **T** and antenna **A**. During the spatial interpolation, the beam indices are located by comparing the lm coordinates with the extent of the beam. If the lm coordinates are found to be between two beam indices, the interpolation occurs linearly between the beam values at those indices. This adds no time complexity to the script. The beam is evaluated for every source **S** at every target frequency for every correlation **C**. Factoring in the fact that each antenna error must be corrected for, the FITS-interpolated beam scales with $\mathcal{O}(\mathbf{TASFC} + \mathbf{F}\log_2(\mathbf{B}))$.

The Zernike implementation, like the FITS-interpolated implementation, must iterate through each time interval, antenna, frequency and source. The Zernike implementation must additionally iterate through each Zernike basis function, and so the time complexity scales with the number of polynomials **P**. Therefore, the time complexity of the Zernike implementation is $\mathcal{O}(\mathbf{TAFSCP})$. Typically, the number of Zernike polynomials used to model the primary beam will not need to exceed 15 as including more coefficients adds no significant improvement on the error in the case of a single frequency (Iheanetu, 2019).

The difference in time complexity between the FITS-interpolated and Zernike models lie in the fact that the FITS-interpolated model performs an extra frequency interpolation at the beginning of the run with a scaling of $\mathcal{O}(\mathbf{F}\log(\mathbf{B}))$, whereas the Zernike model must evaluate a maximum of **P** polynomials at each iteration of the code. While the Zernike implementation should be interpolating the coefficients along the frequency axis step before expanding them, we do not implement this. This is due to the fact that the coefficients are available on a very fine frequency grid. Regardless, the Zernike implementation has a higher time complexity than the FITS-interpolation method. This is due to the fact that the Zernike implementation has multiple coefficients to evaluate.

The inputs required for the FITS-interpolation approach include the beam cube at each correlation, the extent of the beam which is of fixed size, the frequency map and the desired lm coordinates and frequencies. Additionally, for computing antenna errors, it also includes inputs for the parallactic angles, pointing errors and antenna scaling. The dimensions of these inputs gives a scaling in memory requirements of $\mathcal{O}(\mathbf{NBC} + \mathbf{S} + \mathbf{TAF})$.

The Zernike implementation has similar inputs and memory requirements as the FITS-interpolated approach. The difference in their memory requirements lie in the fact that the FITS-interpolated approach requires a beam cube to evaluate the DDE, whereas the Zernike implementation requires Zernike coefficients and Noll indices to evaluate the DDE. However, instead of scaling with **NBC**, the Zernike model scales with **AFCP**. Therefore, the memory complexity of the Zernike implementation is $\mathcal{O}(\mathbf{AFCP} + \mathbf{S} + \mathbf{TAF})$. While relatively few Zernike coefficients are needed to

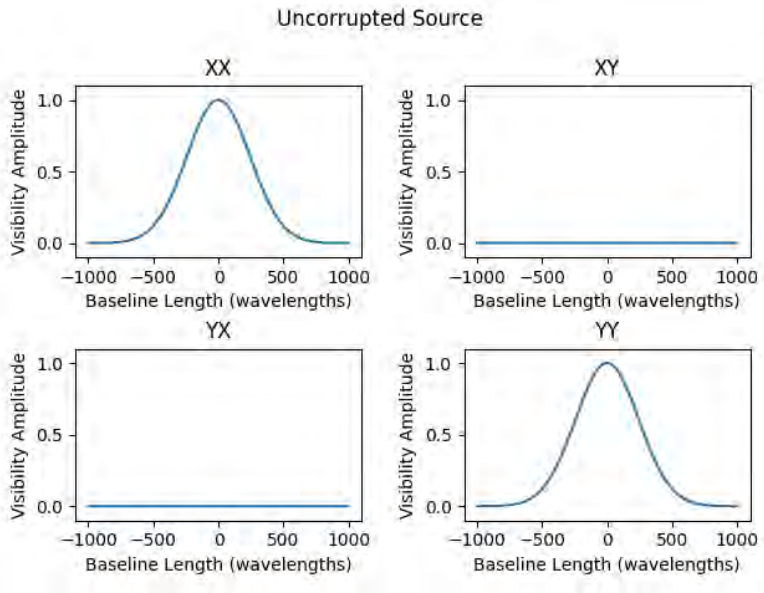


Figure 4.7: Visibility amplitude vs baseline length plot for an uncorrupted and unpolarized Gaussian source at all 4 correlations.

sufficiently describe the beam at floating-point accuracy, the FITS-interpolated model requires a high-resolution beam to give sufficient accuracy. The Zernike implementation therefore leaves a smaller footprint on memory.

4.4.2 Model Accuracy

We generate a sky model corresponding to an unpolarized elliptical Gaussian source located at the half width at half-maximum (HWHM) of the beam and corrupt it. The extent of the elliptical Gaussian source measures 100 arcseconds in both directions, making it circular. We simulate this source at 1085 MHz with a total flux density of 1 Jy. The uncorrupted source can be seen in Figure 4.7. To keep the Zernike polynomial and shapelet implementations separate, we simply use the included Gaussian model in CODEX-AFRICANUS.

Figure 4.8 illustrates the results of the test on the FITS-interpolated beam, whereas Figure 4.9 illustrates the results of the test for the Zernike simulation. The models agree to a respective RMS within the 10^{-5} order of magnitude for the parallel-hand correlations and the 10^{-6} order of magnitude for the cross-hand correlations.

The primary beam introduces polarization leakage (i.e. instrumental polarization) into the visibility data. Following Equation (1.17), we extract the Stokes parameters for the corrupted source. At the HWHM of the beam, the corrupted source is roughly 0.45% Q-polarized, 2% U-polarized and 0.19% V-polarized.

The error introduced between evaluating the Zernike simulation and the FITS-interpolation method is caused by the discretisation of the beam when it is converted to a FITS format. The linear interpolation along the frequency and spatial axes is not as accurate as evaluating the Zernike polynomials directly. The accuracy of the linear

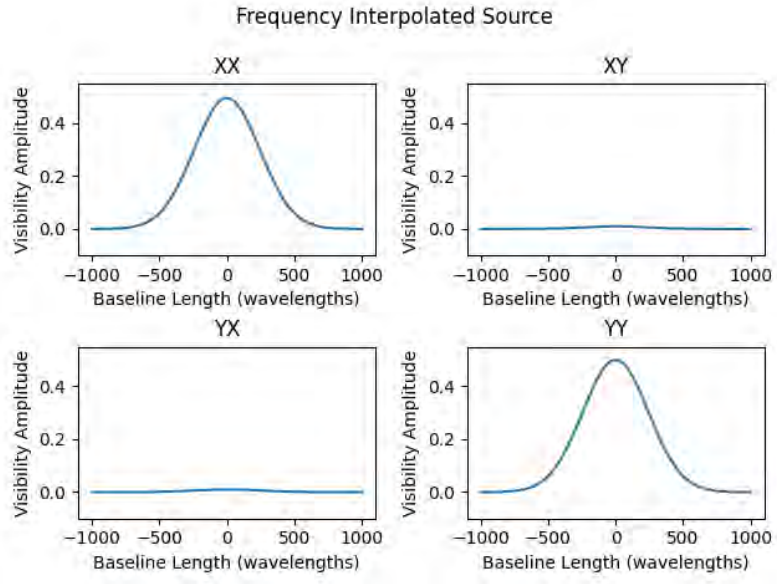


Figure 4.8: Visibility amplitude vs baseline length plot for an unpolarized Gaussian source corrupted with the FITS-interpolated beam at all 4 correlations.

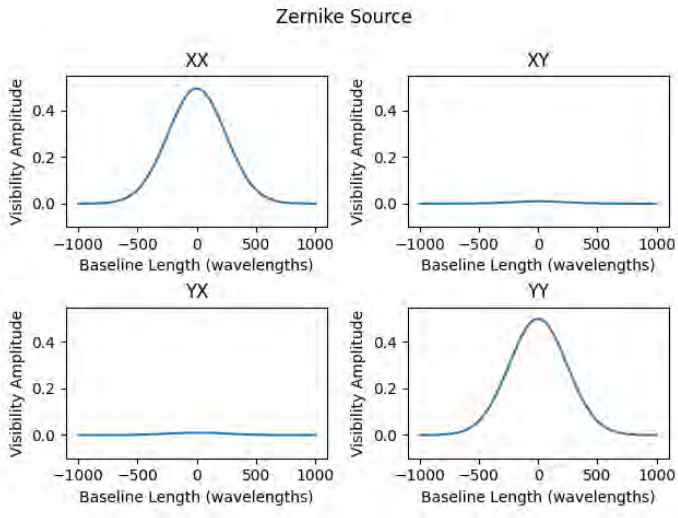


Figure 4.9: Visibility amplitude vs baseline length plot for an unpolarized Gaussian source corrupted with the Zernike beam at all 4 correlations.

interpolation can be increased with the resolution of the beam, but this comes at a large cost in memory usage as **N**, the number of pixels in the beam, can grow large fairly quickly.

4.5 Proof-of-Concept Simulation

This section provides a proof-of-concept simulation where we corrupt component-based sky models with the primary beam. The measurement set used for these experiments are specified in Section 3.4.1. We first wish to investigate the effects on the total flux density by corrupting two point sources. We then wish to corrupt the shapelet model of NGC 6251 simulated in Section 3.4.3 with the primary beam. The images are then recovered using WSCLEAN as specified in Section 3.4.2.

4.5.1 The Effect of the Primary Beam on the Final Image

We wish to investigate the effects of the beam on the total flux density of the source. To do this, we simulate two point sources: one at phase center and one at the HWHM of the beam. Each of these point sources are simulated with a total flux density of 1 Jy. We expect to see the source at phase center maintain its full flux density, while the flux density of the source at the HWHM should be roughly 0.5. As mentioned in Section 4.3.2, the HWHM of the beam is roughly 39.6 arcminutes.

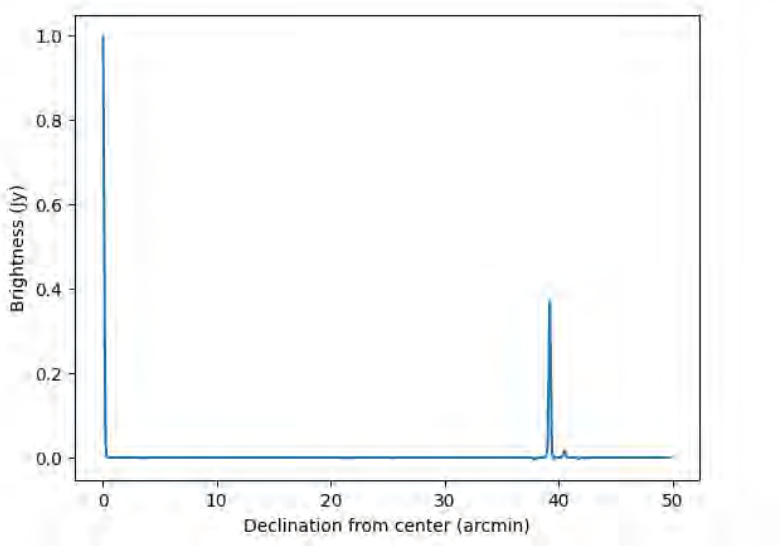


Figure 4.10: Two point sources corrupted by a primary beam. The first source was placed at phase center, and the second point source was placed at 39.6 arcminutes from phase center.

Figure 4.10 illustrates a cross-section of the image from the source at phase center to the source at the HWHM. As expected, the total flux density of the source placed at the HWHM of the beam is about half of that at phase center.

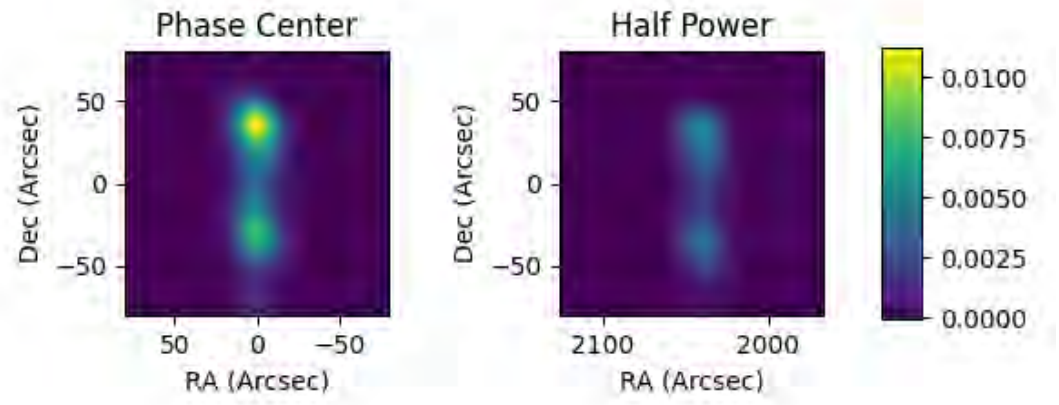


Figure 4.11: The restored images of NGC 6251 corrupted with the simulated MeerKAT primary beam.

4.5.2 Corrupting NGC 6251 with the MeerKAT Primary Beam

This experiment simulates two copies of NGC 6251: one at phase center and one at the HWHM of the beam. We expect to see the source at phase center maintain its full brightness and the source at HWHM to maintain half of its brightness.

To extend the script accordingly, we add the function specified in Appendix J. This function outputs a DDE which can be used to corrupt the shapelet model.

Figure 4.11 illustrates the image obtained from this observation. While the source at phase center maintained its full brightness, the source at the HWHM maintained half of its brightness. This matches the expected behaviour of the observation and, together with the test outlined above, verifies that our implementation works as expected.

4.6 Conclusion

This chapter has presented an implementation of Zernike polynomials into CODEX-AFRICANUS. These polynomials offer a form of modelling the primary beam. We use the Zernike model for the MeerKAT primary antenna supplied by Asad et al., 2019 as a form of testing the implementation.

To test the accuracy of the beam, we compare our implementation to the method of evaluating the beam using a FITS beam model. This method evaluates the beam at a specific coordinate by inputting a FITS beam cube, and evaluates it at a specific frequency through interpolation. We analyze the computational complexity of the two methods of beam evaluation, and find that the Zernike implementation has a higher time complexity, but a lower memory complexity. The lower memory requirement is due to the fact that the Zernike implementation does not need to hold high resolution beams in memory. We find that the two beams agree to an RMS within the 10^{-5}

order of magnitude for the parallel-hand correlations and 10^{-6} for the cross-hand correlations. We attribute this RMS to inaccuracies introduced through interpolation predominantly in the spatial dimension. We conclude that the Zernike implementation can give a more accurate beam while keeping memory requirements low.

In Section 4.5.1, we illustrate the corruption of a set of point sources with one at phase center and one at the HWHM of the beam. This is achieved through a simulation of the visibilities of these sources, similar to the one detailed in Section 3.4.3. We find that, at the HWHM of the beam, the total flux density is halved. This matches the expected behaviour of the source at the HWHM.

Finally, as a proof-of-concept, we simulate two copies of NGC 6251 as presented in Section 3.4.3, place one at phase center and one at the HWHM of the beam. We then corrupt these simulated sources with the simulated beam. This demonstrates that a galaxy with a complex morphology can be decomposed into shapelet components and these components can be corrupted with a primary beam. While decomposing this galaxy into shapelets can give an accurate simulation of the galaxy, the accuracy can be improved by corrupting it with the Zernike-simulated primary beam.

Chapter 5

Conclusion

Direction-dependent calibration is an important aspect of self-calibration and forms the basis of 3GC. The discrepancies between 2GC and 3GC lie in the type of effect calibrated for and the form of modelling used. While 2GC calibrates for effects that can be assumed constant across the field of view, 3GC calibrates for effects that vary with direction. Additionally, 2GC can make use of a single image to use as its sky model, whereas the sky model used in 3GC must be component-based so that each source can be corrupted differently depending on its position within the target field.

Most calibration packages like MONTBLANC, MEQTREES and CODEX-AFRICANUS implement component-based sky models, but these sky models only allow for point source models and simple extended Gaussian sources. This only allows for very simple morphological modelling which can result in subtle artefacts (see Grobler et al., 2014; Sob et al., 2017 for example).

This thesis has presented an implementation of a component-based sky model that is able to simulate extended galaxies with complex morphologies. This sky model is based on decomposing the galaxy morphology into a set of shapelets. Because the shapelets remain invariant under Fourier transform, this allows for a simple and useful form for modelling their visibilities. Additionally, we provide an accelerated implementation of Zernike polynomials which are used in modelling many optical effects and are useful in modelling antenna primary beams.

We implement the shapelet and Zernike polynomial models in CODEX-AFRICANUS, a software package with the aim of providing a fully documented implementation of algorithms for radio astronomy. To accelerate our implementation, we use NUMBA as a JIT compiler while building compute graphs with DASK. This allows us to specify rules for chunking the data and submitting these chunks to the DASK scheduler, which in turn submits them to various cores or nodes.

For the implementation of the shapelet basis functions, we test our implementation against that of the SHAPELETS software package to ensure that the correct values are being output. Additionally, we test the implemented DFT of the shapelets by computing the visibilities on a regular grid and comparing them to the FFT of the shapelet model expanded in image space. Finally, we test the w-term that arises from wide-field modelling of the shapelets by comparing the visibilities of a zero-order shapelet placed off phase center to that of an identical Gaussian computed

with one of the reference implementations. For the implementation of the Zernike polynomials, we simply test the values output by our implementation to that of EIDOS.

After verification of the various models, we test their viability for practical application by comparing them to various reference implementations. For the shapelets, we compare our implemented DFT with NIFTY-GRIDDER. This software package is able to grid and de-grid between images and visibilities. For the purpose of this thesis, we simply de-grid an image onto a set of *uvw* coordinates. This is done to an arbitrary accuracy set by the user. To achieve sufficient accuracy, we find that the runtime of the NIFTY-GRIDDER outperforms the shapelet implementation. This is due to the fact that the runtime of the shapelet implementation scales with the number of shapelet components. However, since the NIFTY-GRIDDER is unable to correctly account for non-trivial DDEs, this implementation is not viable for 3GC. We therefore conclude by stating that a sacrifice in runtime is expected when simulating a component-based sky model for 3GC.

The Zernike beam model was tested by corrupting a Gaussian source, a point source and the NGC 6251 galaxy modelled with shapelets. The Gaussian source was corrupted by simulating an unpolarized Gaussian on a regular *uv*-grid located at the half-power point of the beam. The source was corrupted and each correlation examined. We found that the corruption roughly halved the intensity of the Gaussian source. We also found that the EIDOS beam model contained slight inequalities at each of the correlations of the simulated beam. These inequalities are intentional and are able to simulate the slight polarization leakage in the MeerKAT primary beam.

For the point source test, we simulate two point sources: one at phase center and one at the half-power point of the beam. We find that the total flux density of the point source in the final image at the half-power point is roughly halved while the one at phase center remains the same. We repeat this test for the NGC 6251 test, replacing point sources with the simulated NGC 6251. We see that, from this implementation, we are able to simulate a set of sources with complex morphologies and corrupt these sources with an accurately simulated primary beam.

5.1 Future Work

While this thesis has provided certain building blocks to implement the forward model from image to visibility, further work can be done on the basis functions. Firstly, while our implementation expand a set of already-existing coefficients, it does not implement a form of decomposing an image into coefficients. Additionally, the forward model implemented in this thesis can be incorporated as part of a larger full calibration or Bayesian modelling framework.

Appendix A

Shapelet Model Implementation

```
import numba
import numpy as np
from africanus.constants import c as lightspeed
from africanus.constants import minus_two_pi_over_c

e = 2.71828182845904523536028747135 27
square_root_of_pi = 1.77245385091


@numba.jit(nogil=True, nopython=True, cache=True)
def hermite(n, x):
    if n == 0:
        return 1
    elif n == 1:
        return 2 * x
    else:
        return 2 * x * hermite(n - 1, x) - 2 * (n - 1) * hermite(n - 2,
            ↪ x)


@numba.jit(numba.uint64(numba.int32), nogil=True, nopython=True, cache=
    ↪ True)
def factorial(n):
    if n <= 1:
        return 1
    ans = 1
    for i in range(1, n):
        ans = ans * i
    return ans * n


@numba.jit(nogil=True, nopython=True, cache=True)
def basis_function(n, x, beta, fourier=False):
    if fourier:
        beta = 1.0 / beta
    basis_component = 1.0 / np.sqrt(2.0**n * np.sqrt(np.pi)
                                    * factorial(n) * beta)
    exponential_component = hermite(n, x / beta) * np.exp(-x**2 /
                                                          (2.0*beta**2))

```

```
    if fourier:
        return 1.0j**n * basis_component * exponential_component
    else:
        return basis_component * exponential_component


@numba.jit(nogil=True, nopython=True, cache=True)
def phase_steer_and_w_correct(uvw, lm_source_center, frequency):
    l0, m0 = lm_source_center
    n0 = np.sqrt(1.0 - l0**2 - m0**2)
    u, v, w = uvw
    real_phase = minus_two_pi_over_c * frequency * (u * l0 + v * m0 +
                                                    w * (n0 - 1))
    return np.exp(1.0j * real_phase)


@numba.jit(nogil=True, nopython=True, cache=True)
def shapelet(coords, frequency, coeffs, beta, fourier=True,
             dtype=np.complex128):
    """
    shapelet: outputs visibilities corresponding to that of a shapelet
        ↪ at phase center
    Inputs:
        coords: coordinates in (u,v) space with shape (nrow, 3)
        frequency: frequency values with shape (nchan,)
        coeffs: shapelet coefficients with shape, where coeffs[3, 4] =
            ↪ coeffs_l[3] * coeffs_m[4] (nsrc, nmax1, nmax2)
        beta: characteristic shapelet size with shape (nsrc, 2)
    Returns:
        out_shapelets: Shapelet with shape (nrow, nchan, nsrc)
    """
    nrow = coords.shape[0]
    nsrc = coeffs.shape[0]
    nchan = frequency.shape[0]
    out_shapelets = np.empty((nrow, nchan, nsrc), dtype=np.complex128)
    for row in range(nrow):
        u, v, w = coords[row, :]
        for chan in range(nchan):
            fu = u * frequency[chan] / lightspeed
            fv = v * frequency[chan] / lightspeed
            for src in range(nsrc):
                nmax1, nmax2 = coeffs[src, :, :].shape
                beta_u, beta_v = beta[src, :]
                if beta_u == 0 or beta_v == 0:
                    out_shapelets[row, chan, src] = 1
                    continue
                tmp_shapelet = 0+0j
                for n1 in range(nmax1):
                    for n2 in range(nmax2):
                        tmp_shapelet += 0 if coeffs[src][n1, n2] == 0 \
```

```
87                                                  else coeffs[src][n1, n2] *
                                                        ↪ basis_function(n1, fu,
                                                        ↪  beta_u, fourier) *
                                                        ↪ basis_function(n2, fv,
                                                        ↪  beta_v, fourier)
88                  out_shapelets[row, chan, src] = tmp_shapelet
89      return out_shapelets
90
91
92  @numba.jit(nogil=True, nopython=True, cache=True)
93  def shapelet_with_w_term(coords, frequency, coeffs, beta, lm,
94                           dtype=np.complex128):
95      """
96      shapelet: outputs visibilities corresponding to that of a shapelet
97      Inputs:
98          coords: coordinates in (u,v) space with shape (nrow, 3)
99          frequency: frequency values with shape (nchan,)
100         coeffs: shapelet coefficients with shape, where coeffs[3, 4] =
                ↪ coeffs_l[3] * coeffs_m[4] (nsrc, nmax1, nmax2)
101         beta: characteristic shapelet size with shape (nsrc, 2)
102         lm: source center coordinates of shape (nsource, 2)
103     Returns:
104         out_shapelets: Shapelet with shape (nrow, nchan, nsrc)
105     """
106     nrow = coords.shape[0]
107     nsrc = coeffs.shape[0]
108     nchan = frequency.shape[0]
109     out_shapelets = np.empty((nrow, nchan, nsrc), dtype=np.complex128)
110     for row in range(nrow):
111         u, v, w = coords[row, :]
112         for chan in range(nchan):
113             fu = u * frequency[chan] / lightspeed
114             fv = v * frequency[chan] / lightspeed
115             for src in range(nsrc):
116                 nmax1, nmax2 = coeffs[src, :, :].shape
117                 beta_u, beta_v = beta[src, :]
118                 l, m = lm[src, :]
119                 if beta_u == 0 or beta_v == 0:
120                     out_shapelets[row, chan, src] = 1
121                     continue
122                 tmp_shapelet = 0 + 0j
123                 for n1 in range(nmax1):
124                     for n2 in range(nmax2):
125                         tmp_shapelet += 0 if coeffs[src][n1, n2] == 0 \
126                                           else coeffs[src][n1, n2] *
                                                ↪ basis_function(n1, fu,
                                                ↪ beta_u, True) *
                                                ↪ basis_function(n2, fv,
                                                ↪ beta_v, True)
127                 w_term = phase_steer_and_w_correct((u, v, w), (l, m),
                        ↪ frequency[chan])
128                 out_shapelets[row, chan, src] = tmp_shapelet * w_term
```

```
129     return out_shapelets
130
131
132 def shapelet_1d(u, coeffs, fourier, delta_x=1, beta=1.0):
133     """
134     The one dimensional shapelet. Default is to return the
135     dimensionless version.
136     Parameters
137     ----------
138     u : :class:`numpy.ndarray`
139         Array of coordinates at which to evaluate the shapelet
140         of shape (nrow)
141     coeffs : :class:`numpy.ndarray`
142         Array of shapelet coefficients of shape (ncoeff)
143     fourier : bool
144         Whether to evaluate the shapelet in Fourier space
145         or in signal space
146     beta : float, optional
147         The scale parameter for the shapelet. If fourier is
148         true the scale is 1/beta
149     Returns
150     -------
151     out : :class:`numpy.ndarray`
152         The shapelet evaluated at u of shape (nrow)
153     """
154     nrow = u.size
155     if fourier:
156         if delta_x is None:
157             raise ValueError("You have to pass in a value for delta_x in
                ↪  Fourier mode")
158         out = np.zeros(nrow, dtype=np.complex128)
159     else:
160         out = np.zeros(nrow, dtype=np.float64)
161     for row, ui in enumerate(u):
162         for n, c in enumerate(coeffs):
163             if fourier:
164                 out[row] += c * basis_function(n, 2 * np.pi * ui, beta,
                    ↪ fourier=fourier) * np.sqrt(2 * np.pi) / delta_x
165             else:
166                 out[row] += c * basis_function(n, ui, beta, fourier=
                    ↪ fourier)
167     return out
168
169
170 def shapelet_2d(u, v, coeffs_l, fourier, delta_x=None, delta_y=None,
    ↪ beta=[1.0, 1.0]):
171     nrow_u = u.size
172     nrow_v = v.size
173     if fourier:
174         if delta_x is None or delta_y is None:
175             raise ValueError("You have to pass in a value for delta_x
                ↪ and delta_y in Fourier mode")
```

```
176         out = np.zeros((nrow_u, nrow_v), dtype=np.complex128)
177     else:
178         out = np.zeros((nrow_u, nrow_v), dtype=np.float64)
179     for i, ui in enumerate(u):
180         for j, vj in enumerate(v):
181             for n1 in range(coeffs_l.shape[0]):
182                 for n2 in range(coeffs_l.shape[1]):
183                     c = coeffs_l[n1, n2]
184                     if fourier:
185                         out[i, j] += c * basis_function(n1, 2 * np.pi *
                                ↪ ui, beta[0], fourier=fourier) \
186                                 * basis_function(n2, 2 * np.pi * vj,
                                    ↪  beta[1], fourier=fourier) *
                                    ↪ (2*np.pi)/(delta_x * delta_y)
187                     else:
188                         out[i, j] += c * basis_function(n1, ui, beta[0],
                                ↪  fourier=fourier) \
189                                 * basis_function(n2, vj, beta[1],
                                    ↪ fourier=fourier)
190     return out
```

Appendix B

One Dimensional Shapelet Unit Test

```
def test_1d_shapelet():
  # set signal space coords
  beta = 1.0
  npix = 513
  coeffs = np.ones(1, dtype=np.float64)
  l_min = -15.0 * beta
  l_max = 15.0 * beta
  delta_l = (l_max - l_min)/(npix-1)
  if npix % 2:
    l = l_min + np.arange(npix) * delta_l
  else:
    l = l_min + np.arange(-0.5, npix-0.5) * delta_l
  img_shape = shapelet_1d(l, coeffs, False, beta=beta)

  # get Fourier space coords and take fft
  u = Fs(np.fft.fftfreq(npix, d=delta_l))
  fft_shape = Fs(fft(iFs(img_shape)))

  # get uv space
  uv_shape = shapelet_1d(u, coeffs, True, delta_x=delta_l, beta=beta)

  assert np.allclose(uv_shape, fft_shape)
```

Appendix C

Two Dimensional Shapelet Unit Test

```
def test_2d_shapelet():
  # Define all respective values for nrow, ncoeff, etc
  beta = [.01, .01]
  nchan = 1
  ncoeffs = [1, 1]
  nsrc = 1

  # Define the range of uv values
  u_range = [-3 * np.sqrt(2) * (beta[0] ** (-1)), 3 * np.sqrt(2) * (beta
      ↪ [0] ** (-1))]
  v_range = [-3 * np.sqrt(2) * (beta[1] ** (-1)), 3 * np.sqrt(2) * (beta
      ↪ [1] ** (-1))]

  # Create an lm grid from the regular uv grid
  max_u = u_range[1]
  max_v = v_range[1]
  delta_x = 1/(2 * max_u) if max_u > max_v else 1/(2 * max_v)
  x_range = [-3 * np.sqrt(2) * beta[0], 3 * np.sqrt(2) * beta[0]]
  y_range = [-3 * np.sqrt(2) * beta[1], 3 * np.sqrt(2) * beta[1]]
  npix_x = int((x_range[1] - x_range[0]) / delta_x)
  npix_y = int((y_range[1] - y_range[0]) / delta_x)
  l_vals = np.linspace(x_range[0], x_range[1], npix_x)
  m_vals = np.linspace(y_range[0], y_range[1], npix_y)
  ll, mm = np.meshgrid(l_vals, m_vals)
  lm = np.vstack((ll.flatten(), mm.flatten())).T
  nrow = lm.shape[0]

  # Create input arrays
  img_coords = np.zeros((nrow, 3))
  img_coeffs = np.random.randn(nsrc, ncoeffs[0], ncoeffs[1])
  img_beta = np.zeros((nsrc, 2))
  frequency = np.empty((nchan), dtype=np.float)

  # Assign values to input arrays
  img_coords[:, :2], img_coords[:, 2] = lm[:,:], 0
  img_beta[0, :] = beta[:]
  frequency[:] = 1
```

```
36  img_coeffs[:, :, :] = 1
37
38  # Create output arrays
39  gf_shapelets = np.zeros((nrow), dtype=np.float)
40
41  ca_shapelets = shapelet_1d(img_coords[:, 0], img_coeffs[0, 0, :],
        ↪ False, beta=img_beta[0, 0]) \
42    * shapelet_1d(img_coords[:, 1], img_coeffs[0, 0, :], False, beta=
          ↪ img_beta[0, 1])
43
44  for n1 in range(ncoeffs[0]):
45    for n2 in range(ncoeffs[1]):
46      c = img_coeffs[0, n1, n2]
47      sl_dimensional_basis = sl.dimBasis2d(n1, n2, beta=beta)
48      shapelets_basis_func = sl.computeBasis2d(sl_dimensional_basis,
            ↪ img_coords[:, 0], img_coords[:, 1])
49      gf_shapelets[:] += c * shapelets_basis_func[:]
50  assert np.allclose(gf_shapelets, ca_shapelets)
```

Appendix D

FFT Shapelet Unit Test

```
def test_fourier_space_shapelets():
  # set overall scale
  beta_l = 1.0
  beta_m = 1.0

  # only taking the zeroth order with
  ncoeffs_l = 1
  ncoeffs_m = 1
  coeffs_l = np.ones((ncoeffs_l, ncoeffs_m), dtype=np.float64)

  # Define the range of lm values (these give 3 standard deviations for
      ↪ the 0th order shapelet in image space)
  scale_fact = 10.0
  l_min = -3 * np.sqrt(2) * beta_l * scale_fact
  l_max = 3 * np.sqrt(2) * beta_l * scale_fact
  m_min = -3 * np.sqrt(2) * beta_m * scale_fact
  m_max = 3 * np.sqrt(2) * beta_m * scale_fact

  # set number of pixels
  npix = 257

  # create image space coordinate grid
  delta_l = (l_max - l_min)/(npix-1)
  delta_m = (m_max - m_min)/(npix-1)
  lvals = l_min + np.arange(npix) * delta_l
  mvals = m_min + np.arange(npix) * delta_m
  assert lvals[-1] == l_max
  assert mvals[-1] == m_max

  img_space_shape = shapelet_2d(lvals, mvals, coeffs_l, False, beta=[
      ↪ beta_l, beta_m])#shapelet_1d(lm[:,0], coeffs_l[0,:], False, beta
      ↪ =beta_l) * shapelet_1d(lm[:,1], coeffs_m[0,:], False, beta=
      ↪ beta_m)

  # next take FFT
  fft_shapelet = Fs(fft2(iFs(img_space_shape.reshape(npix,npix))))

  # get freq space coords
  freq = Fs(np.fft.fftfreq(npix, d=(delta_l)))

```

```
37  # Call the shapelet implementation
38  coeffs_l = coeffs_l.reshape(coeffs_l.shape + (1,)) # We only have a
        ↪ single shape parameter, so we simply add another dimension onto
        ↪ it
39  uv_space_shapelet = shapelet_2d(freq, freq, coeffs_l, True, delta_x=
        ↪ delta_l, delta_y=delta_m, beta=[beta_l, beta_m])
40
41  assert np.allclose(fft_shapelet, uv_space_shapelet)
```

Appendix E

W-Term Test

```
import numpy as np
from pyrap.tables import table
from africanus.model.shape.dask import shapelet_with_w_term
from africanus.rime.dask import predict_vis
from africanus.coordinates import radec_to_lm
import Tigger
import dask.array as da

# Load sky model
source = Tigger.load("./sky-model.txt").sources[0]

# Get coordinates, convert to lm
source_extent = np.array([[source.shape.ex, source.shape.ey]])
phase_dir = np.array([0, -30]) * np.pi / 180
source_radec = np.array([[source.pos.ra, source.pos.dec]])
source_lm = radec_to_lm(source_radec, phase_dir)

# Get MS data
ms = table("./TestPowerBeam.MS_p0")
uvw = ms.getcol("UVW")
frequency = table("./TestPowerBeam.MS_p0/SPECTRAL_WINDOW/").getcol("
    ↪ REF_FREQUENCY")
row_chunks = uvw.shape[0] // 64

# Initialize
coeffs = np.ones((1, 1, 1))
beta = source_extent * np.pi / (np.sqrt(2.0 * np.log(2.0)))
coeffs = coeffs * (np.sqrt(np.pi) / beta[0, 0])

# Allocate Dask Arrays
uvw = da.from_array(uvw, chunks=(row_chunks, 3))
frequency = da.from_array(frequency)
coeffs = da.from_array(coeffs)
beta = da.from_array(beta)
source_lm = da.from_array(source_lm)

# Get shapelets
s_fn = shapelet_with_w_term(uvw, frequency, coeffs, beta, source_lm).
    ↪ compute()

```

```
# Get time index and antenna index
time_idx = da.from_array(np.unique(ms.getcol("TIME"), return_inverse=
    ↪ True)[1], chunks=(row_chunks,))
antenna1 = da.from_array(ms.getcol("ANTENNA1"), chunks=(row_chunks,))
antenna2 = da.from_array(ms.getcol("ANTENNA2"), chunks=(row_chunks,))

# Convert to visibilities
jones = np.zeros((1, s_fn.shape[0], 1) + (2, 2), dtype=s_fn.dtype)
jones[0, :, 0, 0, 0], jones[0, :, 0, 1, 1] = s_fn[:, 0, 0], s_fn[:, 0,
    ↪ 0]
jones = da.from_array(jones, chunks=(1, row_chunks) + jones.shape[2:])
visibilities = predict_vis(time_idx, antenna1, antenna2,
                           None, jones, None, None, None).compute()

# Get Gaussian visibilities from MODEL_DATA and compare
gaussian_vis = ms.getcol("MODEL_DATA")
assert np.allclose(gaussian_vis[:, 0, 0].real, visibilities[:, 0, 0, 0].
    ↪ real)
assert np.allclose(gaussian_vis[:, 0, 0].imag, visibilities[:, 0, 0, 0].
    ↪ imag)
```

Appendix F

Observation Specifications

Measurement Set

```
NParts=1
NBands=1
NFrequencies=1
StartFreq=1085e6
StepFreq=10e6
StartTime=2019/10/3/20:46:00
StepTime=10
NTimes=360
RightAscension=00:00:00.0
Declination=-30deg
TileSizeFreq=8
TileSizeRest=10
WriteAutoCorr=F
AntennaTableName=/path/to/antenna_tables/MeerKAT
MSName=TestPowerBeam.MS
VDSPath=.
```

Appendix G

Generating Visibilities from Shapelets

```
def vis_factory(args, source_type, sky_model,
                ms, ant, field, spw, pol):
    try:
        source = sky_model[source_type]
    except KeyError:
        raise ValueError("Source type '%s' unsupported" % source_type)

    # Select single dataset rows
    corrs = pol.NUM_CORR.data[0]
    frequency = spw.CHAN_FREQ.data[0]
    phase_dir = field.PHASE_DIR.data[0][0]  # row, poly

    lm = radec_to_lm(source.radec, phase_dir)
    uvw = -ms.UVW.data if args.invert_uvw else ms.UVW.data

    # (source, spi, corrs)
    # Apply spectral mode to stokes parameters
    stokes = spectral_model(source.stokes,
                            source.spi,
                            source.ref_freq,
                            frequency,
                            base=[1, 0, 0, 0])

    brightness = convert(stokes, ["I", "Q", "U", "V"],
                         corr_schema(pol))

    # Add any visibility amplitude terms
    s_fn = None
    if source_type == "shapelet":
        s_fn = shapelet_fn(uvw, frequency, source.coeffs, source.beta,
            ↪ lm)
        bl_jones_args = ["shapelet_shape", s_fn]
    elif source_type == "point":
        bl_jones_args = ["phase_delay", phase_delay(lm, uvw, frequency)]

    bl_jones_args.extend(["brightness", brightness])

```

```
    # Unique times and time index for each row chunk
    # The index is not global
    time_idx = np.unique(ms.TIME.data, return_inverse=True)[1]
    utime = np.unique(ms.TIME.data, return_inverse=True)[0]

    jones = baseline_jones_multiply(corrs, *bl_jones_args)

    dde = zernike_factory(args, ant, field, pol, lm,
                          utime, frequency, jones.chunks[1])

    return predict_vis(time_idx, ms.ANTENNA1.data, ms.ANTENNA2.data,
                       dde1_jones=dde, source_coh=jones, dde2_jones=dde)


@requires_optional("dask.array", "Tigger",
                   "daskms", opt_import_error)
def predict(args):
    # Convert source data into dask arrays
    sky_model = parse_sky_model(args.sky_model, args.model_chunks)

    # Get the support tables
    tables = support_tables(args)

    ant_ds = tables["ANTENNA"]
    field_ds = tables["FIELD"]
    ddid_ds = tables["DATA_DESCRIPTION"]
    spw_ds = tables["SPECTRAL_WINDOW"]
    pol_ds = tables["POLARIZATION"]

    # List of write operations
    writes = []

    # Construct a graph for each DATA_DESC_ID
    for xds in xds_from_ms(args.ms,
                           columns=["UVW", "ANTENNA1", "ANTENNA2", "TIME
                               ↪ "],
                           group_cols=["FIELD_ID", "DATA_DESC_ID"],
                           chunks={"row": args.row_chunks}):

        # Perform subtable joins
        ant = ant_ds[0]
        field = field_ds[xds.attrs['FIELD_ID']]
        ddid = ddid_ds[xds.attrs['DATA_DESC_ID']]
        spw = spw_ds[ddid.SPECTRAL_WINDOW_ID.data[0]]
        pol = pol_ds[ddid.POLARIZATION_ID.data[0]]

        # Select single dataset row out
        corrs = pol.NUM_CORR.data[0]

        # Generate visibility expressions for each source type
        source_vis = [vis_factory(args, stype, sky_model,
                                  xds, ant, field, spw, pol)
```

```
88                      for stype in sky_model.keys()]
89
90          # Sum visibilities together
91          vis = sum(source_vis)
92
93          # Reshape (2, 2) correlation to shape (4,)
94          if corrs == 4:
95              vis = vis.reshape(vis.shape[:2] + (4,))
96
97          # Assign visibilities to MODEL_DATA array on the dataset
98          xds = xds.assign(MODEL_DATA=(("row", "chan", "corr"), vis))
99
100         # Create a write to the table
101         write = xds_to_table(xds, args.ms, ["MODEL_DATA"])
102
103         # Add to the list of writes
104         writes.append(write)
105
106     # Submit all graph computations in parallel
107     with ProgressBar():
108         da.compute(writes)
```

Appendix H

Zernike Polynomials Implementation

```python
import numpy as np
from africanus.util.numba import jit


@jit(nogil=True, nopython=True, cache=True)
def fac(x):
    if x < 0:
        raise ValueError("Factorial input is negative.")
    if x == 0:
        return 1
    factorial = 1
    for i in range(1, x + 1):
        factorial *= i
    return factorial


@jit(nogil=True, nopython=True, cache=True)
def pre_fac(k, n, m):
    numerator = (-1.0)**k * fac(n-k)
    denominator = (fac(k) * fac((n+m)/2.0 - k) * fac((n-m)/2.0 - k))
    return numerator / denominator


@jit(nogil=True, nopython=True, cache=True)
def zernike_rad(m, n, rho):
    if (n < 0 or m < 0 or abs(m) > n):
        raise ValueError("m and n values are incorrect.")
    radial_component = 0
    for k in range((n-m)/2+1):
        radial_component += pre_fac(k, n, m) * rho ** (n - 2.0 * k)
    # print(radial_component)
    return radial_component


@jit(nogil=True, nopython=True, cache=True)
def zernike(j, rho, phi):
    # print(rho)
```

```
38      if rho > 1:
39          return 0.0
40      j += 1
41      n = 0
42      j1 = j-1
43      # print(j1, n)
44      while (j1 > n):
45          n += 1
46          j1 -= n
47      m = (-1)**j * ((n % 2) + 2 * int((j1+((n+1) % 2)) / 2.0))
48      # print(m)
49      if (m > 0):
50          return zernike_rad(m, n, rho) * np.cos(m * phi)
51      if (m < 0):
52          return zernike_rad(-m, n, rho) * np.sin(-m * phi)
53      return zernike_rad(0, n, rho)
54
55
56  @jit(nogil=True, nopython=True, cache=True)
57  def _convert_coords(l, m):
58      rho, phi = ((l**2 + m ** 2) ** 0.5), np.arctan2(l, m)
59      return rho, phi
60
61
62  @jit(nogil=True, nopython=True, cache=True)
63  def nb_zernike_dde(coords, coeffs, noll_index, out, parallactic_angles,
64                     frequency_scaling, antenna_scaling, pointing_errors):
65      sources, times, ants, chans, corrs = out.shape
66      npoly = coeffs.shape[-1]
67
68      for s in range(sources):
69          for t in range(times):
70              for a in range(ants):
71                  sin_pa = np.sin(parallactic_angles[t, a])
72                  cos_pa = np.cos(parallactic_angles[t, a])
73
74                  for c in range(chans):
75                      l, m, freq = coords[:, s, t, a, c]
76
77                      l = l * frequency_scaling[c]
78                      m = m * frequency_scaling[c]
79
80                      l += pointing_errors[t, a, c, 0]
81                      m += pointing_errors[t, a, c, 1]
82
83                      vl = l * cos_pa - l * sin_pa
84                      vm = m * sin_pa + m * cos_pa
85
86                      vl *= antenna_scaling[a, c, 0]
87                      vm *= antenna_scaling[a, c, 1]
88
89                      rho, phi = _convert_coords(vl, vm)
```

```
90                      # print("rho, phi,l,m, sqrt(l**2 + m**2) is ", rho,
                            ↪ phi, vl,vm,(l**2 + m**2)**0.5)
91
92                      for co in range(corrs):
93                          zernike_sum = 0
94
95                          for p in range(npoly):
96                              zc = coeffs[a, c, co, p]
97                              zn = noll_index[a, c, co, p]
98                              zernike_sum += zc * zernike(zn, rho, phi)
99                              # print(zn, rho, phi)
100
101                         out[s, t, a, c, co] = zernike_sum
102
103     return out
104
105
106 def zernike_dde(coords, coeffs, noll_index, parallactic_angles,
107                 frequency_scaling, antenna_scaling, pointing_errors):
108     """ Wrapper for :func:`nb_zernike_dde` """
109     _, sources, times, ants, chans = coords.shape
110     # ant, chan, corr_1, ..., corr_n, poly
111     corr_shape = coeffs.shape[2:-1]
112     npoly = coeffs.shape[-1]
113
114     # Flatten correlation dimensions for numba function
115     fcorrs = np.product(corr_shape)
116     ddes = np.empty((sources, times, ants, chans, fcorrs), coeffs.dtype)
117
118     coeffs = coeffs.reshape((ants, chans, fcorrs, npoly))
119     noll_index = noll_index.reshape((ants, chans, fcorrs, npoly))
120
121     result = nb_zernike_dde(coords, coeffs, noll_index, ddes,
122                             parallactic_angles, frequency_scaling,
123                             antenna_scaling, pointing_errors)
124
125     # Reshape to full correlation size
126     return result.reshape((sources, times, ants, chans) + corr_shape)
```

Appendix I

Zernike Polynomial Unit Test

```
def test_zernike_func_xx_corr(coeff_xx, noll_index_xx, eidos_data_xx):
    """ Tests reconstruction of xx correlation against eidos """
    from africanus.rime import zernike_dde
    npix = 17
    nsrc = npix ** 2
    ntime = 1
    na = 1
    nchan = 1
    ncorr = 1
    thresh = 15
    npoly = thresh

    # Linear (l,m) grid
    nx, ny = npix, npix
    grid = (np.indices((nx, ny), dtype=np.float) - nx//2) * 2 / nx
    ll, mm = grid[0], grid[1]

    lm = np.vstack((ll.flatten(), mm.flatten())).T

    # Initializing coords, coeffs, and noll_indices
    coords = np.empty((3, nsrc, ntime, na, nchan), dtype=np.float)
    coeffs = np.empty((na, nchan, ncorr, npoly), dtype=np.complex128)
    noll_indices = np.empty((na, nchan, ncorr, npoly))
    parallactic_angles = np.zeros((ntime, na), dtype=np.float64)
    frequency_scaling = np.ones((nchan,), dtype=np.float64)
    antenna_scaling = np.ones((na, nchan, 2), dtype=np.float64)
    pointing_errors = np.zeros((ntime, na, nchan, 2), dtype=np.float64)

    # Assign Values to coeffs and noll_indices
    coeffs[0, 0, 0, :] = coeff_xx[:thresh]
    noll_indices[0, 0, 0, :] = noll_index_xx[:thresh]

    # I left 0 as all the freq values
    coords[0, 0:nsrc, 0, 0, 0] = lm[0:nsrc, 0]
    coords[1, 0:nsrc, 0, 0, 0] = lm[0:nsrc, 1]
    coords[2, 0:nsrc, 0, 0, 0] = 0

    # Call the function, reshape accordingly, and normalise
    zernike_vals = (zernike_dde(coords, coeffs, noll_indices,
                                parallactic_angles, frequency_scaling,
```

```
41                                      antenna_scaling, pointing_errors)[:, 0,
                                            ↪ 0, 0].reshape((npix, npix)))
42      write_fits(zernike_vals.real, [0], 'test_script_fits_beam.fits')
43      write_fits(eidos_data_xx, [0], "eidos_fits_beam.fits")
44      assert np.allclose(eidos_data_xx, zernike_vals)
```

Appendix J

Simulating a Generic Zernike Beam Model

```
def zernike_factory(args, ant, field, pol, lm, utime, frequency,
    ↪ row_chunks):
    """ Generate a primary beam DDE using Zernike polynomials """
    if not args.zernike:
        return None
    corr_type = tuple(pol.CORR_TYPE.data[0])
    corr_type_set = set(corr_type)
    if corr_type_set.issubset(set([9, 10, 11, 12])):
        pol_type = 'linear'
    elif corr_type_set.issubset(set([5, 6, 7, 8])):
        pol_type = 'circular'
    else:
        raise ValueError("Cannot determine polarisation type "
                         "from correlations %s. Constructing "
                         "a feed rotation matrix will not be "
                         "possible." % (corr_type,))

    # Extract coefficient lengths for beam
    nsrc = lm.shape[0]
    utime = utime.compute()
    ntime = len(utime)
    na = np.max(ant['row'].data) + 1
    nchan = len(frequency)
    npoly = 20

    # Make sure row_chunks and time_chunks are compatible
    n_row_chunks = len(row_chunks)
    time_chunk_size = ntime // n_row_chunks
    time_chunks = None
    if ntime % n_row_chunks != 0:
        time_chunks = ((time_chunk_size,) * (n_row_chunks-1))
    else:
        time_chunks = ((time_chunk_size,) * (n_row_chunks))
    if ntime % n_row_chunks != 0:
        time_chunks = time_chunks + (ntime - sum(time_chunks),)

    # Create inputs to Zernike call
```

```
    zernike_coords = np.zeros((3, nsrc, ntime, na, nchan))
    coeffs_r = np.empty((na, nchan, 2, 2, npoly))
    coeffs_i = np.empty((na, nchan, 2, 2, npoly))
    noll_index_r = np.empty((na, nchan, 2, 2, npoly))
    noll_index_i = np.empty((na, nchan, 2, 2, npoly))
    frequency_scaling = da.from_array(np.ones((nchan,)), chunks=(nchan,)
        ↪ )

    parangles = parallactic_angles(utime, ant.POSITION.data,
                                   field.PHASE_DIR.data[0][0])
    parangles = da.from_array(parangles, chunks=(time_chunks, na))
    feed_rot = feed_rotation(parangles, pol_type)
    dtype = np.result_type(parangles, frequency)

    pointing_errors = da.blockwise(_zero_pes, ("time", "ant", "chan", "
        ↪ comp"),
                                   parangles, ("time", "ant"),
                                   frequency, ("chan",),
                                   dtype, None,
                                   new_axes={"comp": 2},
                                   dtype=dtype)
    antenna_scaling = da.blockwise(_unity_ant_scales, ("ant", "chan", "
        ↪ comp"),
                                   parangles, ("time", "ant"),
                                   frequency, ("chan",),
                                   dtype, None,
                                   new_axes={"comp": 2},
                                   dtype=dtype)

    # Convert coordinates to match a beam with a diameter of 10 degrees
    for src in range(nsrc):
        zernike_coords[0, src, :, :, :] = lm[src, 1] * 180 / np.pi / 5
        zernike_coords[1, src, :, :, :] = lm[src, 0] * 180 / np.pi / 5

    # Load in Zernike coefficients for MeerKAT at L-Band
    coeffs_file = np.load("./zernike_coeffs.npz", allow_pickle=True)
    c_freqs = coeffs_file['freqs']
    ch = [abs(c_freqs-i).argmin() for i in (frequency/1e06)]
    params = coeffs_file['params'][ch, :]

    # Assign coefficients
    for ant in range(na):
        for chan in range(nchan):
            coeffs_r[ant, chan, :, :, :] = params[chan, 0][0, :, :, :]
            coeffs_i[ant, chan, :, :, :] = params[chan, 0][1, :, :, :]
            noll_index_r[ant, chan, :, :, :] = params[chan, 1][0, :, :,
                ↪ :]
            noll_index_i[ant, chan, :, :, :] = params[chan, 1][1, :, :,
                ↪ :]

    # Convert arrays to Dask
    zernike_coords = da.from_array(zernike_coords, chunks=(3, nsrc,
```

```
84                                                          parangles.
                                                                ↪ chunks
                                                                ↪ [0],
85                                                          na, nchan))
86      coeffs_r = da.from_array(coeffs_r)
87      noll_index_r = da.from_array(noll_index_r)
88      coeffs_i = da.from_array(coeffs_i)
89      noll_index_i = da.from_array(noll_index_i)
90
91      # Call Zernike_dde
92      dde_r = zernike_dde(zernike_coords, coeffs_r, noll_index_r,
93                          parangles, frequency_scaling,
94                          antenna_scaling, pointing_errors)
95      dde_i = zernike_dde(zernike_coords, coeffs_i, noll_index_i,
96                          parangles, frequency_scaling,
97                          antenna_scaling, pointing_errors)
98      return da.einsum("stafij,tajk->stafik", (dde_r + 1j * dde_i),
            ↪ feed_rot)
```

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Arfken, G. B. and H. J Weber (1999). *Mathematical methods for physicists*, pp. 712–721.

Arras, Philipp et al. (2019). "NIFTy5: Numerical Information Field Theory v5". In: *Astrophysics Source Code Library*.

Asad, K. M. B. et al. (May 2019). "Primary beam effects of radio astronomy antennas-II: Modelling the MeerKAT L-band beam". In: *Monthly Notices of the Royal Astronomical Society* 485.3, pp. 4107–4121.

Bachmann, Paul (1894). *Analytic Number Theory*.

Bastien, David, Nadeem Oozeer, and Dinesh Somanah (Oct. 2016). "Classifying bent radio galaxies from a mixture of point-like/extended images with Machine Learning". In: pp. 1–2. DOI: 10.1109/RADIO.2016.7772037.

Born, M. and E. Wolf (1964). *Principles of optics*.

Briggs, D. S. (1995). "High fidelity deconvolution of moderately resolved sources". PhD thesis.

Cook, Joshua (1976). "The Zernike polynomials". In: *Journal of Modern Optics* 23.8, pp. 679–680.

Cornwell, T. J. and P. N. Wilkinson (1981). "A new method for making maps with unstable radio interferometers". In: *Monthly Notices of the Royal Astronomical Society* 196.4, pp. 1067–1086.

Dask Development Team (2016). *Dask: Library for dynamic task scheduling*. URL: https://dask.org.

Grobler, T. L. et al. (Apr. 2014). "Calibration artefacts in radio interferometry - I. Ghost sources in Westerbork synthesis radio telescope data". In: *Monthly Notices of the Royal Astronomical Society* 439.4, pp. 4030 –4047.

Hamaker, J. P., J. D. Bregman, and R. J. Sault (1996). "Understanding radio polarimetry. I. Mathematical foundations". In: *Astronomy and Astrophysics Supplement Series* 117.1, pp. 137–147.

Iheanetu, K. (2019). "Modelling and investigating the primary beam effects of reflector antenna arrays". PhD thesis.

Lam, S. K. and Pitrou, A. and Seibert, S. (2015). "Numba: a LLVM-based Python JIT compiler". In: *LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1 –6.

Landau, E (1909). "Handbook of the Study of Distribution of the Prime Numbers". In: *New York Chelsea*.

McMaster, William H. (1954). "Polarization and the stokes parameters". In: *American Journal of Physics* 22.6, pp. 351 –362. URL: https://aapt.scitation.org/doi/pdf/10.1119/1.1933744.

McMullin, J. P. and Waters, B. and Schiebel, D. and Young, W. and Golap, K. (2007). "CASA architectures and applications". In: *Astronomical Data Analysis Software and Systems XVI (ASP Conf. Ser. 376)*. Ed. by R. A. Shaw and F. Hill. San Fransisco, CA, p. 127. URL: https://ui.adsabs.harvard.edu/abs/2007ASPC..376..127M/abstract.

Noll, R. J. (1976). "Zernike polynomials and atmospheric turbulence". In: *JOsA* 66.3, pp. 207–211.

Noordam, J. E. and O. Smirnov (2010). "The MeqTrees software system and its use for third-generation calibration of radio interferometers". In: *Astronomy & Astrophysics* 524, A61.

Nunhokee, C. D., T. L. Grobler, and O. M. Smirnov (2015). "Link between ghost artefacts, source suppression and incomplete calibration sky models". PhD thesis, pp. 22 –45.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Perkins, S. J. et al. (2015). "Montblanc: GPU accelerated radio interferometer measurement equations in support of bayesian inference for radio observations". In: *Astronomy and Computing* 12, pp. 73–85.

Pratley, L., M. Johnston-Hollitt, and J. D. McEwen (2019). *w-stacking w-projection hybrid algorithm for wide-field interferometric imaging: implementation details and improvements*. arXiv: 1903.06555 [astro-ph.IM].

Python Software Foundation (2018). *Python Language Reference*. URL: http://www.python.org/.

Refregier, A. (2003). "Shapelets—I. A method for image analysis". In: *Monthly Notices of the Royal Astronomical Society* 338.1, pp. 35–47.

Reinecke, M., T. Steininger, and M. Selig (2018). *NIFTy - Numerical Information Field TheorY*. URL: https://gitlab.mpcdf.mpg.de/ift/NIFTy.

Rocklin, M. (2015). "Dask: parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th Python in Science Conference*, pp. 130–136.

Ryle, M. (1962). "The new Cambridge radio telescope". In: *Nature* 194, pp. 517–518.

Smirnov, O. (2011). "Revisiting the Radio Interferometer Measurement Equation-I. A Full-Sky Jones Formalism". In: *Astronomy & Astrophysics* 527, A106.

Sob, U. A. M. et al. (2017). "Calibration and imaging with variable radio sources". PhD thesis, p. 18.

The Theano Development Team et al. (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688. arXiv: 1605.02688. URL: http://arxiv.org/abs/1605.02688.

Thompson, A. R., J. M. Moran, and G. W. Swenson Jr (2008). *Interferometry and synthesis in radio astronomy*. John Wiley & Sons.

Waggett, P. C., P. J. Warner, and J. E. Baldwin (1977). "NGC 6251, a very large radio galaxy with an exceptional jet". In: *Monthly Notices of the Royal Astronomical Society*.

Wilf, H. S. (2002). *Algorithms and complexity*, pp. 9–19.

Zernike, F. (1934). "Diffraction theory of the knife-edge test and its improved form, the phase-contrast method". In: *Monthly Notices of the Royal Astronomical Society* 94, pp. 377–384.