Rhodes University

Masters Thesis

PyMORESANE: A Pythonic and CUDA-accelerated implementation of the MORESANE deconvolution algorithm

Author:
Jonathan Kenyon

Supervisor:
Prof. Oleg Smirnov

A thesis submitted to

Rhodes University
Department of Physics and Electronics

in fulfilment of the requirements
for the degree of Master of Science

The financial assistance of the South African SKA Project (SKA SA) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to the SKA SA.

March 2015

Declaration of Authorship

I, Jonathan Kenyon, declare that this thesis titled, 'PyMORESANE: A Pythonic and CUDA-accelerated implementation of the MORESANE deconvolution algorithm' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

RHODES UNIVERSITY

Faculty of Science
Department of Physics and Electronics

Master of Science

PyMORESANE: A Pythonic and CUDA-accelerated implementation of the MORESANE deconvolution algorithm

by Jonathan Kenyon

The inadequacies of the current generation of deconvolution algorithms are rapidly becoming apparent as new, more sensitive radio interferometers are constructed. In light of these inadequacies, there is renewed interest in the field of deconvolution. Many new algorithms are being developed using the mathematical framework of compressed sensing. One such technique, MORESANE, has recently been shown to be a powerful tool for the recovery of faint diffuse emission from synthetic and simulated data. However, the original implementation is not well-suited to large problem sizes due to its computational complexity. Additionally, its use of proprietary software prevents it from being freely distributed and used. This has motivated the development of a freely available Python implementation, PyMORESANE. This thesis describes the implementation of PyMORESANE as well as its subsequent augmentation with MPU and GPGPU code. These additions accelerate the algorithm and thus make it competitive with its legacy counterparts. The acceleration of the algorithm is verified by means of benchmarking tests for varying image size and complexity. Additionally, PyMORESANE is shown to work not only on synthetic data, but on real observational data. This verification means that the MORESANE algorithm, and consequently the PyMORESANE implementation, can be added to the current arsenal of deconvolution tools.

Acknowledgements

No body of work can be completed in a vacuum. Fortunately, my supervisor, Oleg Smirnov, was there to point me in the correct direction. For this, I offer him my sincerest thanks.

I must also extend my gratitude to Arwa Dabbech and her supervisor, Chiara Ferrari, without whom I would never have had the opportunity to produce this thesis. I am also indebted to them for their help and hospitality during my research visit to France.

Perhaps most importantly, I extend my thanks to Ronel Groenewald, whose helpfulness and knowledge of university procedure has helped me on multiple occasions.

I would like to reiterate how thankful I am for the assistance of the SKA SA. Their contribution made this possible.

I must acknowledge the contribution of Steven Gunn and Sunil Patel in providing the LaTeX template on which this document is based.

Finally, I would like to extend my thanks to my family, whose support was crucial during the tougher moments of this project.

Contents

List of Figures

List of Tables

Notation

$\mathbf{x}$	Vector
$\mathbf{x_i}$	i^{th} entry of vector $\mathbf{x}$
$\mathbf{M}$	Matrix
$\mathbf{M_{i,j}}$	Entry of matrix $\mathbf{M}$ at (i,j)
$\mathbf{M}^T$	Matrix transpose of $\mathbf{M}$
$\mathbf{M}^{-1}$	Matrix inverse of $\mathbf{M}$
$\mathcal{F}(\cdot)$	Fourier transform
$\mathcal{F}^{-1}(\cdot)$	Inverse Fourier transform
$\|\mathbf{x}\|_p$	l^p-norm of vector $\mathbf{x}$

Chapter 1

Introduction

The necessity of deconvolution in the context of radio astronomy has a long history. As such, there have been numerous attempts, some more successful than others, to construct efficient deconvolution algorithms. Recently, great improvements have been made and the development of compressed sensing and its associated techniques has produced a whole new family of algorithms.

One such algorithm, MORESANE (MOdel REconstruction by Synthesis-ANalysis Estimators), has recently been shown to be remarkably capable at the restoration of faint diffuse emission (see [1] and [2]). While the details of its operation are the subject of subsequent chapters, it is important to note that one of its critical weaknesses is its computational complexity. That is, for images of sizes between 2048-by-2048 and 8192-by-8192 pixels, the computation time may be completely unfeasible. Additionally there is no real upper bound on problem size, and new interferometers with large fields of view and high spatial resolution will require even larger images. Addressing this weakness in conjunction with a desire to turn the algorithm into a freely available tool motivated the development of PyMORESANE.

PyMORESANE itself an implementation of the MORESANE algorithm coded in Python. There are arguments against the use of Python in high-performance computing, particularly due to the difficulties introduced by the global interpreter lock and just-in-time compilation. These both limit the maximum achievable performance. However, it remains both remarkably convenient and very familiar to most astronomers. Additionally, due to its popularity and support, it can be adapted to incorporate a wide range of functionality, some of which overcomes its inherent drawbacks.

In order to combat the computational complexity of the algorithm, the aim is to incorporate GPGPU (General-Purpose computing on Graphics Processing Units) functionality

into PyMORESANE. GPUs offer massive parallel computing power. Thus, when dealing with large array-based calculations they are incredibly efficient. This efficiency does not reduce the computational complexity of the algorithm, but means that far more complex calculations may be performed in a comparatively small amount of time.

Subsequent to the inclusion of GPU functionality, a secondary aim is the verification of the algorithms' efficacy both on synthetic data and real observational data. This verification is intended to establish whether or not MORESANE survives the transition from the ideal synthetic case to the less perfect real-world data.

The details of MORESANE as well as its competitors, both old and new, appear in chapter 2. Once the theoretical framework has been established, the details of the actual implementation appear in chapter 3.

The results are grouped into three distinct chapters. The first, chapter 4, presents the results of accelerating the algorithm. This establishes both where the bottlenecks were and how they have been dealt with by the inclusion of GPU code. Chapter 5 then presents the results of applying PyMORESANE to synthetic data. This serves as a check between PyMORESANE and the original implementation. The final results chapter, chapter 6, demonstrates the efficacy of PyMORESANE when applied to real data. MORESANE was previously untested on real observational data.

The final chapter concludes with the findings of the thesis and suggests the further improvements and additional directions in which the algorithm may develop.

Chapter 2

Context and Theory

This chapter explores the necessity of deconvolution and its myriad approaches, old and new. The parallel capabilities of graphics hardware is also briefly discussed. Note, all colour-map images (radio images) in this and subsequent chapters have scales in units of Jansky ($1\text{Jy} = 10^{-26}\text{W.m}^{-2}\text{.Hz}^{-1}$). This corresponds to the spectral flux density.

2.1 Interferometry

In order to fully explain deconvolution in the context of radio astronomy, it is important to understand interferometry and how it necessitates deconvolution. This brief explanation will follow the RIME (Radio Interferometer Measurement Equation) formalism, as presented first in [3] and later elaborated upon in [4].

Single dish radio telescopes, in general, have very poor angular resolution. This is due to the fact that angular resolution (θ) is directly proportional to wavelength (λ) and inversely proportional to aperture diameter (D), as show in equation 2.1 [5].

$$\theta \approx \frac{\lambda}{D} \tag{2.1}$$

Radio waves have wavelengths several orders of magnitude greater than visible light. Thus, for single-dish radio telescopes, the only way to improve angular resolution is to increase the aperture diameter by building bigger dishes. However, the dish size rapidly becomes impractical. In order to overcome this limitation, the study of radio interferometry and aperture synthesis was born.

Physically, a radio interferometer consists of two or more radio telescopes. It is analogous to its optical counterpart - by examining the interference pattern between electromagnetic waves at spatially separated points, additional information about the source of the emission can be extracted. In particular, an interferometer may be used to synthesise a far larger aperture than would be feasible for a single dish radio telescope; an interferometer has the angular resolution of a single dish with a diameter equal to the greatest separation between the interferometer's constituent telescopes.

The smallest possible radio interferometer consists of only two dishes. This pair is referred to as a baseline. A large interferometer, consisting of many dishes, can be treated as a number of two-element interferometers or baselines. Each dish measures a voltage which corresponds to the received radio emission. For each unique baseline, these voltages are correlated to produce a single complex value at a given instant. This value is termed a visibility. It is convenient at this point to introduce the following relationship [6]:

$$V_\nu(u,v,w) = \iint A(l,m) I_\nu(l,m) e^{-2\pi i(ul+vm+wn)} \frac{dldm}{n} \tag{2.2}$$

In the above equation, $V_\nu(u,v,w)$ is the measured visibilities, $I_\nu(l,m)$ is the true brightness distribution of the sky, and the exponential term contains the phase information. $A(l,m)$ is the primary beam pattern which is treated as unity for the remainder of this discussion, as it does not contribute to the formulation of the problem.

Before proceeding, the above coordinate systems require some explanation. The (u,v,w) coordinates, measured in units of wavelength, give position in the Earth's frame. The w-axis is chosen to be perpendicular to the central point (phase-centre) of the field which is under observation. The u and v axes then correspond to East and North respectively in the plane which is normal to w at the surface of the Earth. This is commonly referred to as the uv-plane.

The coordinates of a point in the sky are given by (l,m,n). They are the direction cosines of a point relative to the u, v and w axes. A more thorough, pictographic explanation is available in [7].

For various technical reasons - see [8] - it is desirable to have zero fringe frequency at the phase centre. Incorporating this into equation 2.2 results in the subtly modified form of equation 2.3.

$$V_\nu(u,v,w) = \iint I_\nu(l,m) e^{-2\pi i(ul+vm+w(n-1))} \frac{dldm}{n} \tag{2.3}$$

Equation 2.3 is very similar to a two dimensional Fourier transform between (l,m) and (u,v). It is only the presence of the w-term which prevents it from being precisely that. Fortunately, this term may be omitted under certain conditions [8].

The first such condition is true only for co-planar interferometers; the baselines have no w-component and thus the term in w disappears. The second such condition is referred to as the small-field approximation. The approximation is valid when $|l|$ and $|m|$ are sufficiently small for equation 2.5 to hold, noting that the direction cosines l, m and n are related by equation 2.4:

$$l^2 + m^2 + n^2 = 1 \tag{2.4}$$

$$(n-1)w = (\sqrt{1-l^2-m^2} - 1)w \approx -\frac{1}{2}(l^2+m^2)w \approx 0 \tag{2.5}$$

In the second of the above cases, the following form of 2.2 is applicable, noting here that $n \approx 1$:

$$V_\nu(u,v) = \iint I_\nu(l,m) e^{-2\pi i(ul+vm)} dldm \tag{2.6}$$

Equation 2.6 shows that the visibilities are, in fact, just the two dimensional Fourier transform of the sky brightness distribution. It is this remarkable property that allows radio astronomers to construct images of the sky from the measured visibilities; the above equation may be inverted to give equation 2.7.

$$I_\nu(l,m) = \iint V_\nu(u,v) e^{2\pi i(ul+vm)} dudv \tag{2.7}$$

This suggests that recovering the sky brightness distribution is as simple as performing an inverse Fourier transform on the measured visibilities. Unfortunately, as previously stated, equation 2.6, and consequently equation 2.7, is an idealisation. If $V_\nu(u,v)$ was known at every point in the uv-plane, then $I_\nu(l,m)$ could be perfectly calculated. However, an interferometer only measures visibilities at finite number of discrete points in the uv-plane.

These points are described by a sampling function $S(u,v)$, which is non-zero only at the sampled points and their negatives. This "two-for-one" property arises from the fact that the sky is real; its Fourier transform is Hermitian, so sampling one point in

the transform also gives information about a second point. Specifically, the visibility at point $(-u, -v)$ is equal to the complex conjugate of the visibility at the point (u, v) [8] .

For notational simplicity, $\mathcal{F}$ will be used to notate a two-dimensional Fourier transform and $\mathcal{F}^{-1}$ shall denote its inverse. Equation 2.6 and 2.7 may be recast as 2.8 and 2.9 respectively.

$$V_\nu(u, v) = \mathcal{F}(I_\nu(l, m)) \tag{2.8}$$

$$I_\nu(l, m) = \mathcal{F}^{-1}(V_\nu(u, v)) \tag{2.9}$$

The actual visibilities $V_\nu^{'}(u, v)$ measured by an interferometer are $S(u, v)$ multiplied by $V_\nu(u, v)$, which can be expressed as follows:

$$V_\nu^{'}(u, v) = S(u, v)V_\nu(u, v) \tag{2.10}$$

By replacing $V_\nu(u, v)$ with $V_\nu^{'}(u, v)$ in equation 2.9, a new expression for the modified sky brightness distribution, $I_\nu^{'}(l, m)$, is obtained in equation 2.11. By invoking the properties of the Fourier transform, this equation may be further manipulated to give equation 2.12 which expresses $I_\nu^{'}(l, m)$ as the convolution of the true sky brightness distribution, $I_\nu(l, m)$, with the Fourier transform of the sampling function. This quantity is referred to as the point spread function (PSF) or dirty beam and is notated as $P(l, m)$.

$$\begin{aligned} I_\nu^{'}(u, v) &= \mathcal{F}^{-1}(V_\nu^{'}(u, v)) && (2.11) \\ &= \mathcal{F}^{-1}(S(u, v)V_\nu(u, v)) \\ &= \mathcal{F}^{-1}(S(u, v)) * \mathcal{F}^{-1}(V_\nu(u, v)) \\ &= P(l, m) * I_\nu(l, m) && (2.12) \end{aligned}$$

Conceptually, equation 2.12 shows that the information captured by an interferometer cannot be used to perfectly reconstruct the true sky brightness distribution. Instead, when constructing an image from the visibilities a "dirty" result - henceforth referred to as the dirty image - is obtained; the sky has been corrupted by convolution with the PSF. The desire to reverse the effects of this convolution makes deconvolution a necessary procedure.

The difficulty with deconvolution arises from the uncertainty which the convolution with the PSF introduces. As information is not available at every point in the uv-plane, a

dirty image does not correspond to a unique sky. There is a continuum of skies which could produce equivalent visibilities. It is the aim of deconvolution to produce an image which best represents the reality. In order to do this, prior assumptions have to be made and the missing data interpolated or inferred.

Before discussing deconvolution any further, there are a few aspects of interferometry which are commonly employed to improve the reconstruction of the sky brightness distribution. Due to the relationship between the sampling function and the PSF, improving the number of sampled points or uv-coverage also improves the PSF. This in turn reduces the negative effects of convolution.

The simplest approach to improving the sampling function is to add additional telescopes to the interferometer. Each additional baseline corresponds to an additional pair of sampled points but, as with increasing the size of a single dish, this approach rapidly becomes impractical and cost-prohibitive.

The second, ubiquitous, approach is to exploit the Earth's rotation. As the earth rotates, each baseline measures the visibility at a slightly different pair of points in the uv-plane. Thus, by taking measurements at regular time intervals, a series of visibilities and their associated sampling functions can be obtained.

The visibilities from every time interval can be combined when making an image and the total sampling function becomes the summation of the sampling functions of each time interval. This procedure is known as Earth-rotation aperture synthesis [9]. It dramatically improves the sampling function and, consequently, the PSF. However, even with the best sampling function possible, it is impossible to completely remove the PSF, reaffirming the need for an efficient means of deconvolution.

Examples of the sampling function, PSF and a simulated synthetic dirty image for the KAT7 array appear in figures 2.1, 2.2, and 2.3.

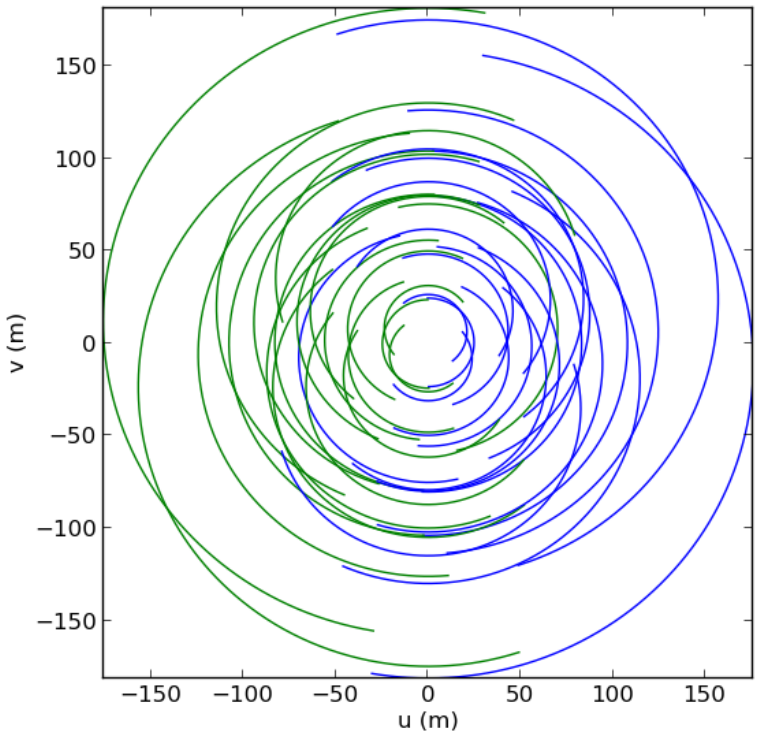

FIGURE 2.1: Coverage of the uv-plane for a simulated KAT7 observation.

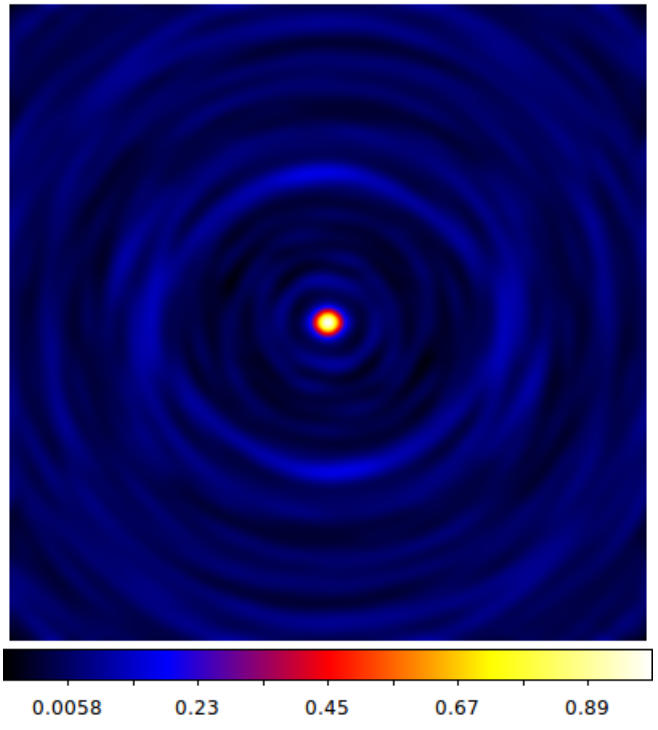

FIGURE 2.2: PSF of KAT7 for the uv-coverage in figure 2.1.

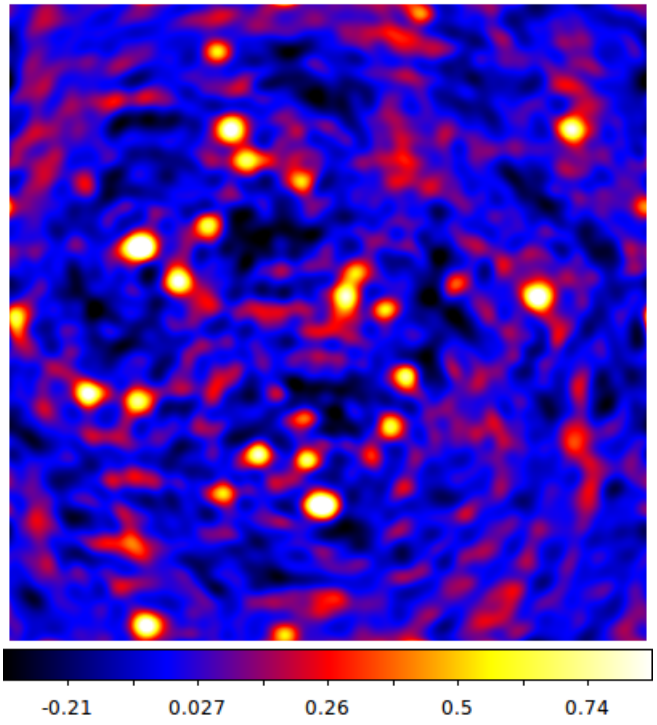

FIGURE 2.3: Dirty image of a simulated KAT7 observation.

2.2 Deconvolution

Deconvolution is ultimately an inverse problem - given the results of an operation, the aim is to recover the input to that operation. For the purposes of this explanation, deconvolution will be treated in the image plane. To clarify, rather than treating the true sky brightness distribution and visibilities as the input and output of the convolution, the images thereof will be used instead. Thus, the goal of deconvolution is to reclaim an image of the true sky brightness distribution from the dirty image constructed from the visibilities. This can be more elegantly expressed in the notation of linear algebra. Note that a noise term is included as data obtained from an interferometer contains noise. In image space, the noise is both additive Gaussian and correlated due the incomplete sampling of the uv-plane [10]. The forward operation may be written as follows:

$$\mathbf{y} = \mathbf{P}\mathbf{x} + \mathbf{n} \tag{2.13}$$

In equation 2.13, $\mathbf{y}$ is a vector containing the pixel values of the dirty image, $\mathbf{x}$ is a vector containing the pixel values of the true sky image, $\mathbf{n}$ is a vector containing the noise per pixel, and $\mathbf{P}$ is a matrix which, when applied to $\mathbf{x}$, amounts to a convolution with the PSF. This is the image-space equivalent of equation 2.12.

However, while equation 2.13 is very intuitive in one dimension, it is less so for two. In two dimensions, $\mathbf{x}$ and $\mathbf{y}$ are vectors constructed by stacking the columns of the input and output images respectively and $\mathbf{P}$ is a large, difficult to construct matrix.

This notation is an improvement over that presented in section 2.1, as expressing the reverse operation is much simpler. Deconvolution may be naively expressed as the matrix inverse of $\mathbf{P}$ applied to $\mathbf{y}$.

Unfortunately, it is not possible to perform this operation, even if $\mathbf{P}$ is known exactly. In general $\mathbf{P}$ is under-determined; it is rank deficient (has zero eigenvalues) and consequently it is non-invertible. It is for this reason that inverse problems are one of the most exhaustively studied problems in mathematics. Solving inverse problems usually amounts to an optimisation problem in which an objective function (usually the l^2 norm) is minimised.

This minimisation problem in non-trivial in radio images - the presence of noise further complicates matters. Thus, additional criteria, such as regularisation constraints, are imposed to emphasise certain features of the recovered model image. Section 2.3.4 explores this notion more thoroughly.

Deconvolution is not restricted to the radio regime; optical instruments suffer from similar problems. However, as most optical instruments are not interferometric in nature, their PSFs tend to be better behaved and thus do not impact the entire image. This does not reduce the necessity of deconvolution as fine structure can be blurred by instrumental effects such as diffraction. Whilst a discussion of optical techniques is not within the purview of this thesis, it is interesting to note the parallels between the techniques of optical and radio astronomy.

2.3 Current Techniques

Due to the presence of convolution in all interferometric measurements, several approaches to deconvolution have already been explored. This section offers a brief overview of some of the most well-known techniques.

2.3.1 CLEAN

The CLEAN procedure is the original deconvolution algorithm, presented by J. A. Högbom in [11]. It remains one of the most successful and powerful algorithms for deconvolution. This is particularly note-worthy as it is elegant in its simplicity.

CLEAN's approach to deconvolution is reliant on the fact that the PSF has a known structure. A point source will produce an impulse response in the image corresponding to the PSF and an extended source will produce a superposition of such responses. Thus, the PSF can be regarded as being correlated to the dirty image. This correlation is inherent - the point of maximum correlation is the maximum point in the image. Simply put, the brighter a source, the greater its correlation with the PSF.

CLEAN exploits this correlation to remove the PSF. It selects the pixel with the maximum intensity as being a source. It then takes the source pixel and adds some fraction of it to a new, model image of the sky. It simultaneously subtracts the same fraction of the PSF, scaled and centred at the location of the source pixel, from the dirty image. This process is iterative; on each iteration the dirty image, commonly referred to as the residual, is updated in the same way until a stopping criterion is reached. The stopping criterion is usually based on the noise level of the residual.

The net effect of CLEAN is the removal of the contribution of the brightest sources from the dirty image and the creation of a model image containing the true sky, uncontaminated by the PSF.

In general, the model produced by CLEAN is convolved with a Gaussian restoring beam (often called the clean beam) which usually corresponds to the central lobe of the PSF. This is then added to the final residual. It is this combination of residual and convolved CLEAN components - referred to as the restored image - which presents the best version of the information hidden in the original dirty image.

The original CLEAN algorithm proposed by Högbom has been improved upon by several other authors. The most notable improvements include those of Clark in [12] and Schwab in [13].

Clark's improvements to the algorithm were designed to accelerate it. At the time, the processing power and memory available for computation was far more limited. Clark identified the successive subtraction of the PSF from the dirty image as being the same as convolving the model image with the PSF and subtracting the result from the dirty image. This realisation motivated him to split the CLEAN algorithm into a major and minor loop.

Clark's minor loop makes use of a so-called beam patch - the PSF's central region. The beam patch is used to identify a limiting flux for the dirty image. All points below the flux of the brightest pixel's largest side-lobe outside the beam patch are not considered during the current major loop iteration.

The remainder of the minor loop behaves much like Högbom's CLEAN. It identifies the brightest pixel in its thresholded dirty map and removes some fraction of the beam patch at its location. The flux removed from the brightest pixel is placed into the model image. This is done iteratively until no components lie above the original threshold. Conceptually, this can be regarded as localised deconvolution - the minor loop does not attempt to remove the effects of the PSF in its entirety. Instead, it simply creates a model of all the source pixels above the flux limit by removing the PSF's brightest contributions.

The major loop completes the CLEAN operation by convolving the model image produced by the minor loop with the entire PSF. The result is then subtracted from the dirty image. This can be considered as the aggregation of several iterations of Högbom's original implementation.

Each major loop uses the updated dirty image to redetermine the flux threshold and the major loop iterates until such time as a stopping criterion is reached. This criterion is often heuristic and determines how close to the noise the algorithm will clean. As with Högbom's CLEAN, the final model image is usually convolved with a Gaussian restoring beam and added to the residual.

Computationally, Clark's CLEAN is a substantial improvement. Using the beam patch massively reduces the problem size in the minor loop and the convolution is done in a single step on each major loop iteration. The convolution is performed using an FFT (Fast Fourier Transform), which scales well with problem size ($O(N\log_2(N))$).

Schwab adapted Clark's CLEAN [13] to make use of the visibilities in the major loop. The model image is used to compute a set of visibilities which are subtracted from the measurement set. A new image is then computed, which becomes the input to the subsequent major loop.

This approach has several advantages over its predecessors. Both Högbom and Clark's implementations are limited to the central quadrant of the dirty image. This is due to the fact that, for an image and PSF of the same size, convolution with the PSF produces edge effects outside of the central region. By removing the model from the visibilities, the PSF is effectively removed everywhere, allowing the algorithm to operate on the whole dirty image. This approach also reduces the errors introduced by convolution with the PSF, allowing the algorithm to clean more deeply and with fewer false detections.

Unfortunately, while CLEAN is still a remarkable algorithm, it is not without its flaws. In particular, there is an implicit assumption that all sources, point or otherwise, can be modelled as a discrete number of delta functions. This assumption arises from the selection of one pixel on each iteration. As the sky is principally filled with point sources, this is often irrelevant. However, it means that CLEAN does not produce an accurate model when applied to diffuse emission. At best, it will distribute delta functions over the source's extent. At worst, particularly when the diffuse emission is close to the noise level, the removal of the first delta function from the diffuse emission will render the remainder of the emission unrecoverable.

2.3.2 Multiscale CLEAN

As mentioned in section 2.3.1, the CLEAN algorithm does not model diffuse emission realistically. Thus, in order to mitigate this weakness, multi-scale CLEAN, as proposed in [14], has been developed. It improves substantially on the basic CLEAN approach whilst maintaining its predecessors simplicity.

The multi-scale approach is relatively easy to explain. Where the basic CLEAN algorithm limits itself to using delta functions to model sources, multi-scale CLEAN combines both delta functions and extended components. These extended components are chosen to be tapered, truncated parabolas which are similar to Gaussians, but avoid the difficulties the long, non-zero tails of Gaussians introduce. The quantity and extent of these components is a user-specified parameter.

In order to make use of the extended components, multi-scale CLEAN constructs multiple versions of the dirty image. Each version is smoothed by convolution with the extended component of interest. The result of this smoothing operation is an image cube, with each image plane corresponding to a particular resolution or scale. A similar smoothing procedure is applied to the PSF. Thus, for every plane in the image cube, there is a smoothed version of the PSF; the PSF at that scale.

The algorithm then proceeds in a fashion analogous to the basic CLEAN algorithm. The maximum pixel across all scales is found, subject to the application of a scaling parameter. The scaling parameter is somewhat arbitrary, but in practice is related to the ratio of the current scale to the maximum scale. However, the parameter is vital, as it biases the algorithm towards smaller scales. This prevents point sources couched in diffuse emission from being modelled with large scale components. Failure to do this often results in undesirable negative values around point sources in the residual.

Once the maximum pixel has been found, the model is updated with some fraction of the corresponding multi-scale component. The PSF associated with that scale is then subtracted from the residual at the location of the maximum pixel. The other planes in the image cube are also updated by exploiting the relationship between the various scales - the PSF for the current maximum scale is convolved with the component of each other scale. This essentially gives the maximum component as it would appear in the other scales.

This process is repeated until such time as a predefined threshold is reached, or the maximum number of components is identified. The final step, as with CLEAN, is the convolution of the model with a restoring beam and the addition of the residuals to form the restored image.

Multi-scale CLEAN has been shown (see [14]) to be more effective at recovering diffuse emission than conventional CLEAN. However, it too has its flaws. In particular, the various component scales are user-specified - there is no way to know in advance which scales should be chosen to produce the best results. This free-parameter is undesirable as it precludes automation. Additionally, the scaling or biasing factor introduced when determining the maximum across all the scales is wholly arbitrary. A relationship which works was derived empirically, but it is unlikely to be the only or the best solution.

One final thing to note is that multi-scale CLEAN does restrict the morphology of the resulting model image. Even though the extended components can model diffuse sources, ultimately the algorithm still searches for the maximum single pixel across scales. Even if this pixel appears in a high-scale plane, it is ultimately still a blurred delta function which means that the components which will appear in the model will all be circular for a parabolic component approach.

2.3.3 Bayesian Approaches

Recent work has proposed the use of Bayesian inference to construct a model of the true sky. These approaches are particularly appealing as they are the most statistically accurate of the current techniques and quantify the error in the reconstructed image. Prior to these Bayesian approaches, such uncertainties were beyond the conventional algorithms.

Presenting the Bayesian framework is beyond the scope of this work, but it is nevertheless interesting to gloss over its current presence in the literature. The MEM (Maximum Entropy Method) [15] is a precursor to the more recent Bayesian approaches. It attempts to find a model sky by maximising the relative entropy between the dirty image and the model sky. However, this alone is insufficient and so the objective function which is ultimately maximised contains measures of the relative entropy, the χ^2 value and the total power of the expected MEM image. This approach works remarkably well, particularly on diffuse emission. However, it is more computationally intensive than CLEAN and is very dependent on the correct specification of its various constraints. Additionally, it has been know to incorrectly estimate source flux. One final thing to note is that this version of the MEM algorithm, unlike the original presented in [16], is not truly Bayesian due to the inclusion of total power in the objective function. The total power term is a form of regularisation.

Regularisation refers to the inclusion of some additional factor when determining the solution to an ill-posed problem. This factor can be regarded as a penalty which induces desirable behaviour in a solution.

A more recent approach appears in [17]. Known as RESOLVE, this approach is more explicitly Bayesian. That is, given data d and a true signal s, RESOLVE attempts to construct what is known as the posterior distribution which is written as $\mathcal{P}(s|d)$. This can be though of as the probability of signal s given data d. Bayesian methods, however, require a prior. That is, they require additional information in the form of a model which is thought to represent the probability distribution of the signal s.

RESOLVE makes use of a prior which focusses on diffuse, extended emission and achieves high-quality reconstruction of such emission, particularly for low-noise data. However, as with all Bayesian methods, the computational cost is large. This is due to fact that Bayesian methods have to find the most probable solution in a very large parameter space. RESOLVE does provide a good uncertainty estimator but it is unclear whether or not it is suited to real data.

Another Bayesian approach is Gibbs Sampling, as introduced in [18]. It is similar to RESOLVE, although it uses a different prior. What makes this approach particularly interesting is that is can be automated. The data drives the solution procedure, so there is no need for user input. With the data rate of the next-generation interferometers, automation will be vital. The presented results are also quite impressive. Once again, it is important to note that the test cases were relatively small and it remains unclear whether the approach is suited to real data. The current implementation does feature multiprocessing support, though no timing results are available.

2.3.4 Compressed Sensing and Sparsity

Compressed sensing refers to a family of techniques which are employed in signal processing, as explored in [19]. Fundamentally, they address the problem of signal reconstruction in the presence of an under-determined linear system. Conceptually, this can be regarded as an approach which allows the best reconstruction of a signal from the minimum number of measurements.

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{2.14}$$

Equation 2.14 is linear measurement equation where $\mathbf{y}$ is M-dimensional vector containing samples or measurements, $\mathbf{x}$ is an N-dimensional vector corresponding to the signal of interest, and A is the M-by-N measurement matrix which describes how the signal is sampled. Usually, in order to perfectly reconstruct the signal, it is necessary to have M and N equal. Compressed sensing has shown [19] that M may be substantially less than N provided that the signal $\mathbf{x}$ is sufficiently sparse.

This notion of sparsity describes a signal which, in some representation, consists mainly of zeroes. This is not to be confused with matrix sparsity which is a measure of how many entries of a matrix are zero.

While many signals may not be strictly sparse (containing mainly zero coefficients), they are often compressible. A signal is compressible if, in some basis, the majority of its coefficients are close to zero but not necessarily zero. That is, for a signal $\mathbf{x}$ of size $(N, 1)$, equation 2.15 holds.

$$\mathbf{x} = \mathbf{S}\boldsymbol{\gamma} \tag{2.15}$$

In the above equation, $\mathbf{x}$ is as previously defined, $\mathbf{S}$ is a dictionary of size (N, M) and $\boldsymbol{\gamma}$ is a sparse vector of size $(M, 1)$. This is known as synthesis. Conceptually, this means

that the linear combination of only a few columns (often referred to as atoms) of the full dictionary $\mathbf{S}$ are needed to synthesise $\mathbf{x}$. $\boldsymbol{\gamma}$ picks out the atoms in question. In general, the dimensions of the dictionary $\mathbf{S}$ are such that $N > M$.

The second, slightly different, approach is known as analysis. In this case sparsity is derived from an operator, say $\mathbf{A^T}$, such that the product $\mathbf{A^T x}$ is sparse. Conceptually, this differs from the synthesis approach as, unlike $\boldsymbol{\gamma}$, neither $\mathbf{A^T}$ or $\mathbf{x}$ need be sparse. From this perspective, analysis can be regarded as the correlation between a dictionary $\mathbf{A}$ of size (N, M) and the signal vector $\mathbf{x}$, the result of which is sparse, as only a few atoms of $\mathbf{A}$ will be well correlated with $\mathbf{x}$.

The notions of synthesis and analysis are simply different means by which a signal can be represented as sparse and they are in fact identical when an orthonormal basis is used. This is vital, as compressed sensing relies on the fact that the sparsest solution to an under-determined system is the best one. This can be applied to solutions in the form of a regularisation factor; a sparsity measure can be used to regularise the solution of a linear system. This encourages the solution towards sparsity.

Several new deconvolution algorithms make use of sparsity in some basis. This includes both LOFAR-CS [20], a CS-based deconvolution algorithm for the LOFAR array, and the PURIFY library [21] which includes routines implementing the SARA (Sparsity Averaging Reweighted Analysis) imaging algorithm.

PURIFY/SARA makes use of sparsity by exploiting wavelet decompositions. Unique to SARA is the concatenation of both wavelet dictionaries and the Dirac basis. These are used to extablish the average sparisty of an image across multiple representations. The resulting sparsified coefficients are used to reconstruct the signal of interest using the SDMM (simultaneous-direction method of multipliers). PURIFY/SARA works directly on the visibilities, which improves signal reconstruction and has been shown to produce excellent results on synthetic data. Unfortunately, it is currently untested on real data.

LOFAR-CS is of more interest in the context of this thesis, due to the similarities between it and MORESANE. In particular, they both make use of the same wavelet decomposition and noise estimator. However, LOFAR-CS makes use of FISTA (Fast Iterative Shrinking Threshold Algorithm), whereas MORESANE uses a conjugate gradient method. LOFAR-CS also has several similarities with PURIFY/SARA, as they are both examples of standard constrained/unconstrained problems often seen in compressed sensing. LOFAR-CS, like PURIFY/SARA, works directly on the visibilities. LOFAR-CS has been shown to produce superior results to more traditional deconvolution algorithms, and has been tested in a more realistic setting than PURIFY/SARA. Information on the computational performance of the algorithm is not readily available.

2.4 MORESANE

MORESANE (MOdel REconstruction by Synthesis-ANalysis Estimators) is recently developed algorithm proposed by Dabbech et al. [2]. This section attempts to explain MORESANE as simply as possible. However, a rigorous mathematical approach appears in [2].

2.4.1 Compressed Sensing and Sparsity in MORESANE

Armed with the introductory outline of sparsity as presented in section 2.3.4 it is possible to further elaborate on sparsity in the context of MORESANE and the deconvolution problem.

Returning to the notation of section 2.2, with $\mathbf{y} = \mathbf{Px}$ providing a model of the dirty image ($\mathbf{y}$) and its relation to the true sky ($\mathbf{x}$) by convolution with the PSF ($\mathbf{P}$), it is possible to write the form of the solutions for both synthesis and analysis.

$$\mathbf{x_S} = \mathbf{S} \cdot \{\arg\min_{\boldsymbol{\gamma}} \frac{1}{2} \|\mathbf{PS}\boldsymbol{\gamma} - \mathbf{y}\|^2 + \mu_p \|\boldsymbol{\gamma}\|_p^p\} \tag{2.16}$$

The synthesis equation (2.16) describes a solution using a synthesis dictionary, as presented in equation 2.15. To clarify, $\mathbf{x_S}$ corresponds to the synthesis solution, $\|\cdot\|_p$ is the l^p norm, and μ_p is a tuning parameter which adjusts the weighting of regularisation factor, $\|\boldsymbol{\gamma}\|_p^p$. The remaining symbols retain their previous definitions.

The first term of the expression to be minimised is not wholly unfamiliar, as it is ultimately a measure of the distance between the measured values, $\mathbf{y}$, and the values predicted by the model, $\mathbf{x} = \mathbf{S}\boldsymbol{\gamma}$. This is referred to as the data fidelity term.

The second term of the minimisation expression is a measure of the sparsity of the solution. p corresponds to the degree of the norm. The most obvious choice for a norm which is indicative of a system's sparsity is the l^0 pseudo-norm, which is simply a count of the non-zero entries in the given matrix. However, the l^0 pseudo-norm is non-convex; the resulting optimisation problem is not readily solvable. As a result, it is usually replaced with the l^1 norm which is convex and remains an excellent measure of sparsity. The l^1 norm is a summation of all the elements in a given matrix. The additional factor of μ_p weights the sparsity measure and determines the degree of sparsity required.

The whole of the minimisation expression is minimised with respect to $\boldsymbol{\gamma}$. Conceptually, this describes a search procedure for an optimal vector $\boldsymbol{\gamma}$ which minimises the objective

(minimisation) function. The final product with the dictionary $\mathbf{S}$ is necessary to return from a sparse representation to the actual solution vector $\mathbf{x_S}$.

$$\mathbf{x_A} = \{\arg\min_{\mathbf{x}} \frac{1}{2} \|\mathbf{Px} - \mathbf{y}\|^2 + \mu_p \left\|\mathbf{A^T x}\right\|_p^p\} \tag{2.17}$$

The analysis equation (2.17) is very similar to its synthesis counterpart, with the analysis solution vector represented by $\mathbf{x_A}$. The factor μ_p and $\|\cdot\|_p$ retain their previous definitions.

The principle difference between the approaches appears in the regularisation factor, $\left\|\mathbf{A^T x}\right\|_p^p$, although its function remains the same; it is a regularisation factor which promotes sparsity. However, unlike the synthesis case for which $\boldsymbol{\gamma}$ contained all the sparsity information, the objective function must be minimised with respect to $\mathbf{x}$ such that $\mathbf{A^T x}$ is sufficiently sparse.

2.4.2 The Isotropic Undecimated Wavelet Transform

The IUWT (Isotropic Undecimated Wavelet Transform) is used to construct the analysis dictionary for the MORESANE algorithm. A full mathematical explanation of the IUWT is presented in [22], and an elaboration on its use in MORESANE appears in [23].

First, a brief introduction to wavelet transforms in general is necessary ([24] is an authoritative reference). Conceptually, they are similar to a Fourier transform, as they transform a signal from one basis to another. However, unlike a Fourier transform, which decomposes signals into sines and cosines which are localised only in Fourier space, wavelet transforms make use of functions which are localised to some degree in both real and Fourier space. This is vital when the signal in question is non-stationary and the location (whether in time or space) of the components of the signal are important. The general form of a wavelet transform appears in equation 2.18.

$$Wf(u,s) = \langle f, \psi_{u,s} \rangle = \int_{-\infty}^{+\infty} f(t) \frac{1}{\sqrt{s}} \psi^* \Big(\frac{t-u}{s} \Big) dt \tag{2.18}$$

This is simply the inner product (correlation) between a function $f(t)$ and some wavelet function $\psi_{u,s}(t)$ where u and s correspond to translation and dilatation of ψ respectively.

There are a multitude of different wavelet transforms, each one corresponding to a different choice for ψ. Writing an analytic function for ψ may be difficult, as it is

determined by a series of filter banks. Suffice it to say that the IUWT and its associated ψ is particularly well suited to astrophysical applications.

The first reason for this is its isotropy, as many astronomical objects are isotropic. The second is that it is simple to implement and the transform itself is relatively fast. The third is that, due to the lack of decimation (sub-sampling), it is translation invariant and highly redundant. Thus, morphological features can be accurately modelled.

Returning to the problem which MORESANE is attempting to solve, it is possible to write the following:

$$\boldsymbol{\alpha} = \mathbf{A^T y} \tag{2.19}$$

In the above equation, $\mathbf{y}$ is the vectorised dirty image, $\mathbf{A^T}$ is an operator with $\mathbf{A}$ corresponding to the IUWT decomposition and $\boldsymbol{\alpha}$ is the matrix containing the analysis (wavelet) coefficients. This, as described in subsection 2.3.4, is an analysis operation which produces a coefficient set showing the correlation between $\mathbf{A}$ and $\mathbf{y}$.

The IUWT decomposes $\mathbf{y}$ into a set of $(J+1)N$ coefficients where J is the number of wavelet scales and N is the number of pixels in the dirty image. Here, scale refers to the extent of the wavelet with which the signal is correlated. Low scale values correspond to high-frequency components, which in turn correspond to the fine details of the image, such as point sources and edges. High scale values correspond to low-frequency components which tend to correlate well with extended emission. Thus, $\boldsymbol{\alpha}$ may be more formally written as $\boldsymbol{\alpha} = [\mathbf{w}_1^T, ..., \mathbf{w}_J^T, \mathbf{c}_J^T]^T$ where each $\mathbf{w}_j$ is the detail coefficients at scale j and $\mathbf{c}_J$ is the smoothest approximation of the original image.

The IUWT is also well suited to the application as it allows for perfect reconstruction of a decomposed signal. Thus, the following equation holds:

$$\mathbf{y} = \mathbf{S}\boldsymbol{\alpha} \tag{2.20}$$

In the above equation, $\mathbf{y}$ is the dirty image, $\mathbf{S}$ is a synthesis dictionary which corresponds to the analysis dictionary $\mathbf{A}$ and $\boldsymbol{\alpha}$ is as defined above. This is a very important property, as otherwise it would not be possible to return from wavelet space to the original signal.

The IUWT can be implemented using the "à trous" algorithm (translates as "with holes") [24]. This approach allows for the decomposition to be rapidly determined in accordance with equations 2.21 and 2.22, with $\mathbf{c}_0 = \mathbf{y}$.

$$\mathbf{c}_{j+1}[k] = \sum_m \mathbf{h}[m]\mathbf{c}_j[k + m2^j] = (\mathbf{h}^{(j)} * \mathbf{c}_j)[k] \tag{2.21}$$

$$\mathbf{w}_{j+1}[k] = \sum_m \mathbf{g}[m]\mathbf{c}_j[k + m2^j] = (\mathbf{g}^{(j)} * \mathbf{c}_j)[k] \tag{2.22}$$

In the left-hand equality of the equations above, $\mathbf{c}_j$ and $\mathbf{w}_j$, both of which are indexed by k, correspond to the approximation and detail coefficients at scale j. $\mathbf{h}$ and $\mathbf{g}$ are the filters corresponding to the IUWT which are indexed by m. These equations are equivalent, as shown in the right-hand equality, to a convolution between the filters $\mathbf{h}^{(j)}$ or $\mathbf{g}^{(j)}$ with the approximation coefficients $\mathbf{c}_j$ where the filters are dilated for increasing j. That is, the filters have a finite number of non-zero entries which at each scale are separated by $2^j - 1$ zeroes. It is this property which makes the "à trous" rapid, as the convolution may be reduced to a sum over the non-zero products of the filter with the approximation coefficients.

The corresponding reconstruction algorithm has the following form:

$$\mathbf{c}_j[k] = (\tilde{\mathbf{h}}^{(j)} * \mathbf{c}_{j+1})[k] + (\tilde{\mathbf{g}}^{(j)} * \mathbf{w}_{j+1})[k] \tag{2.23}$$

All symbols retain their previous definitions; the only change appears in $\tilde{\mathbf{h}}^{(j)}$ and $\tilde{\mathbf{g}}^{(j)}$ which are the filters used for reconstructing the signal.

MORSEANE makes use of the second generation IUWT which has associated filters ($\mathbf{h}$, $\mathbf{g}$, $\tilde{\mathbf{h}}$, $\tilde{\mathbf{g}}$). A rigorous explanation of the choice filter banks will not be presented here, though it is thoroughly covered in [22]. However, it is useful to note that for this version of the IUWT, $\mathbf{h} = \tilde{\mathbf{h}}$, $\mathbf{g} = \delta - \mathbf{h} * \mathbf{h}$ and $\tilde{\mathbf{g}} = \delta$, where the Dirac function $\delta_{k,l} = 1$ if $k, l = 0$ and is zero otherwise. Incorporating this knowledge into equations 2.21 and 2.22 yields a simplified decomposition scheme which can be easily implemented.

$$\mathbf{c}_{j+1}[k] = (\mathbf{h}^{(j)} * \mathbf{c}_j)[k] \tag{2.24}$$

$$\mathbf{w}_{j+1}[k] = \mathbf{c}_j[k] - (\mathbf{h}^{(j)} * \mathbf{h}^{(j)} * \mathbf{c}_j)[k] = \mathbf{c}_j[k] - (\mathbf{h}^{(j)} * \mathbf{c}_{j+1})[k] \tag{2.25}$$

Likewise, the reconstruction scheme of equation 2.23 is also simplified:

$$\mathbf{c}_j[k] = (\mathbf{h}^{(j)} * \mathbf{c}_{j+1})[k] + \mathbf{w}_{j+1}[k] \tag{2.26}$$

The discussion thus far has only dealt with the one dimensional case of the "à trous" algorithm. However, as shown in [24], the extension to two dimensions is fairly trivial, particularly for the case of separable filters such as in the IUWT.

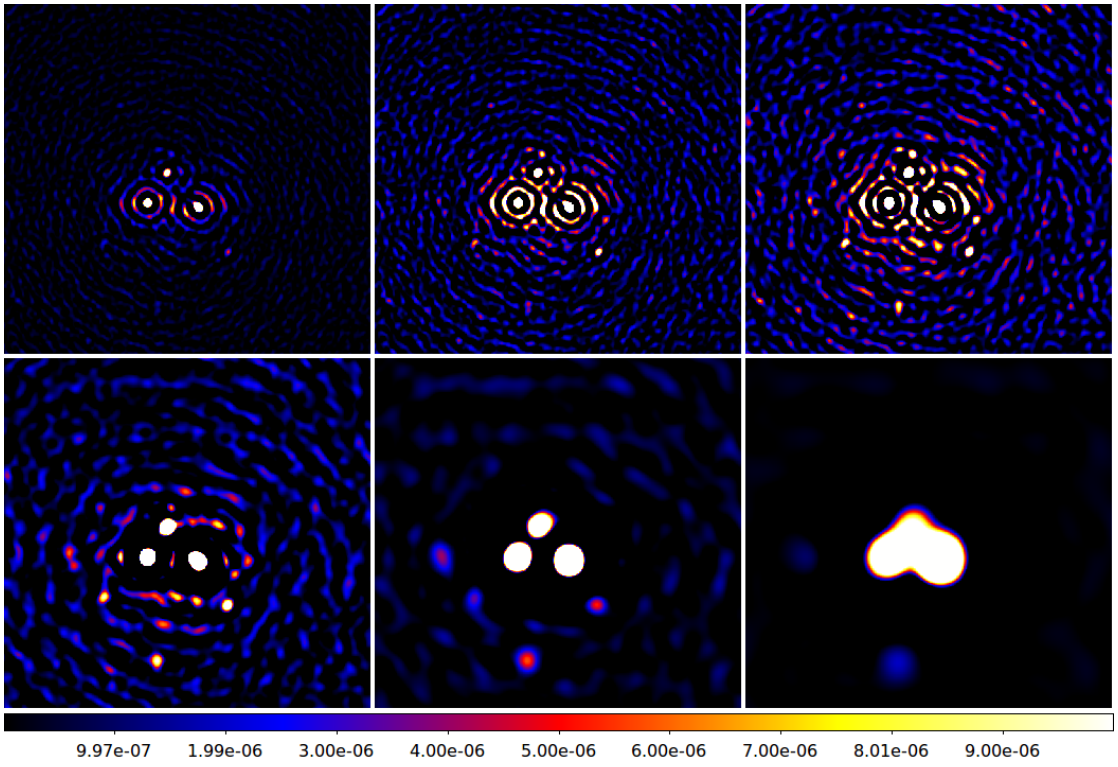

FIGURE 2.4: Six plane IUWT second-generation decomposition of synthetic data. The images are in order of ascending scale from top left to bottom right.

An example of the IUWT decomposition appears above. The smoothest approximation of the initial image, $\mathbf{c}_j$, has been omitted as it is of little interest. As is clear in the image, the compact sources (smallest details) appear in the low scale planes, whereas extended emission appears at the high scales. To be clear, the high scale planes in this case are not necessarily only diffuse emission - the point sources are also smeared out. Thus, for very bright point sources couched in diffuse emission, it is necessary to remove the brightest compact sources before the signature of the diffuse emission becomes clear.

2.4.3 The Algorithm

Thus armed with the knowledge of the previous sections, it is possible to outline the MORESANE algorithm [2]. The details of its implementation will be the subject of the following chapter, but an understanding the of the aim and operation of the algorithm is beneficial. It is important to note that, like CLEAN, the algorithm is iterative.

The starting point of the algorithm is the IUWT (see subsection 2.4.2). The dirty input image is decomposed up to a specified scale. This yields an array of analysis coefficients from which the maximum analysis coefficient and its associated scale are determined. The wavelet coefficients at scales greater than the maximum are discarded for the iteration.

Due to the redundancy of the transform, there too many (non-zero) coefficients in the decomposition to be useful. Thus, a thesholding procedure is used to remove the smallest coefficients.

MORESANE makes use of the MAD (Median Absolute Deviation) [25] estimator when determining the noise level at a given scale. This scale dependent estimation is necessary as noise is more prominent at the lower scales. A multiplicative constant is usually attached to the MAD estimator and the result is used as the threshold or detection limiting value.

MORESANE currently uses a hard threshold; all coefficients below the threshold are set to zero and those above are unaltered. The result of the thresholding procedure, the de-noised analysis coefficients, consist of islands of analysis coefficients which lie above the threshold value. The following procedure, termed object identification, is carried out these values.

Object identification consists of a masking procedure, followed by a second thresholding operation and then a pursuit of scale-connected structures. For clarity, structures are connected, non-zero components at a given scale, and objects are connected structures, where connectivity is determined by whether or not the structures overlap. MORESANE identifies structures at all scales using a simple test for connected components. However, not all structures correspond to real sources, and a second thresholding procedure is required.

The thresholding procedure is based on the scale containing the maximum coefficient. A user-specified tuning parameter, τ, is used to limit which structures are considered significant for a given iteration. Thus, objects at the maximum scale which contain a coefficient within some percentage, given by τ, of the maximum coefficient, are regarded as significant. All the insignificant structures are zeroed out. A similar procedure

is applied on the lower scales too, however the threshold is based on the maximum coefficient at that scale, rather than the overall maximum.

The remaining structures then undergo a process which connects them to the structures at lower scales. This process is simple, as structures are said to be connected if they overlap. Thus, if a structure at scale j has non-zero coefficients above it at scale $j-1$, the structures are connected and form an object. This process connects structures from the maximum scale up, and any structures which remain unconnected to the maximum scale are also removed.

The result of the object identification is a binary mask which determines where the coefficients of interest are. Multiplication of this mask with the analysis coefficients yields a sparse version of the analysis coefficients. On each iteration only a relatively low number of objects will be found, so there will be relatively few non-zero coefficients.

This sparse set of coefficients, which corresponds to the significant objects (the sources in the input image) is then used to determine the true sky which corresponds to those coefficients. This is done by means of a conjugate gradient descent procedure. While the mathematics behind the conjugate gradient descent method will not be presented here, it is pertinent to express its objective.

$$\mathbf{x}_{sig} = \arg\min_{\hat{\mathbf{x}}} \left\| \boldsymbol{\alpha}_{sig} - \mathbf{M}\mathbf{A}^T\mathbf{P}\hat{\mathbf{x}} \right\|_2^2 \text{ s.t. } \hat{\mathbf{x}} \geq 0 \tag{2.27}$$

In the above equation, $\boldsymbol{\alpha}_{sig}$ corresponds to the identified significant analysis coefficients, $\hat{\mathbf{x}}$ is an approximation of the sky which would give rise to those coefficients following a convolution with the PSF, $\mathbf{P}$, an IUWT decomposition, $\mathbf{A}^T$, and a multiplication with the significant object mask, $\mathbf{M}$. The result of minimising with respect to $\hat{\mathbf{x}}$, $\mathbf{x}_{sig}$, is the closest approximation to the sky consisting of only the significant objects.

Only a fraction of the partial sky model, $\mathbf{x}_{sig}$, is added to the true sky model. The true sky model may then be convolved with the PSF and subtracted from the dirty image. The process, as mentioned at the beginning of the subsection, is iterative. Thus, on each iteration the true sky model is updated and the result is used to further deconvolve the dirty image.

The SNR (signal-to-noise ratio) is used to asses whether the algorithm is successfully approximating the objects of interest. The algorithm iterates until such time as a tolerance or detection limit is reached, or until the algorithm can no longer identify any significant objects.

In practice, instead of running the algorithm once by specifying an initial number of decomposition scales, it is run iteratively for increasing scales. Thus, MORESANE attempts to identify sources at small scales before moving on to the diffuse emission.

The pseudo-code for the algorithms is given below. Note, however, that PyMORESANE has substantially more options than presented here, though the basic functionality remains the same. This pseudo-code, whilst nearly identical to the original presented in [2], has been adapted in an attempt to simplify it and also to make comparison with the operation of PyMORESANE simpler.

The symbols presented in the pseudo-code correspond to the quantities already described in this section. Those which have been omitted are briefly described here. The indices k and n refer to scale. Thus, the weighting vector $\mathbf{w}_k$, contains the weights for the various scales. These weights serve to ensure that the maximum is identified at the correct scale. A thresholding operation, $T_x(\cdot)$, thresholds $(\cdot)$ with respect to x. That is, coefficients in $(\cdot)$ less than x are set to zero. K refers to the maximum scale of the IUWT decomposition. $\mathbf{r}$ is the residual. On the first iteration, the residual and the dirty image are equivalent. As the algorithm iterates the residual replaces the original dirty image - it is what remains after the deconvolution step.

Algorithm 1 MORESANE

Inputs: $\mathbf{y}, \mathbf{P}$
$J \leftarrow 1$
$J_{max} \leftarrow \log_2(\text{size}(\mathbf{y})) - 1$
$\mathbf{r}_0 \leftarrow \mathbf{y}$
$\mathbf{x}_{model} \leftarrow \mathbf{0}$
while $J < J_{max}$ **do**
  $\mathbf{x}_J \leftarrow$ **Algorithm 2**
  if $\mathbf{x}_J = \mathbf{0}$ **then**
    return $\mathbf{x}_{model}$, $\mathbf{r}_J$
  end if
  $\mathbf{x}_{model} \leftarrow \mathbf{x}_{model} + \mathbf{x}_J$
  $\mathbf{r}_J \leftarrow \mathbf{y} - \mathbf{P} * \mathbf{x}_{model}$
  $J \leftarrow J + 1$
end while
return $\mathbf{x}_{model}$, $\mathbf{r}_J$

Algorithm 2 Sky estimation - major loop.

Inputs: $\mathbf{r}, \mathbf{P}, \gamma, N_{itr}$
$i \leftarrow 0$
$\mathbf{x}_{(i=0)} \leftarrow \mathbf{0}$
$\mathbf{r}_{(i=0)} \leftarrow \mathbf{r}$
while $\boldsymbol{\alpha}_i \neq \mathbf{0}$ **and** $i < N_{itr}$ **do**
  $\mathbf{M}_{sig}, \boldsymbol{\alpha}_{sig} \leftarrow$ **Algorithm 3**.
  Calculate $\mathbf{x}_{sig}$ using the Conjugate Gradient Descent algorithm - minor loop.
  $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \gamma \mathbf{x}_{sig}$
  $\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \gamma \mathbf{P} * \mathbf{x}_{sig}$
  $i \leftarrow i + 1$
end while
return $\mathbf{x}_i$

Algorithm 3 Object extraction.

Inputs: $\mathbf{r}, \mathbf{P}, K, \tau$
$\mathbf{M} \leftarrow \mathbf{0}$
$\boldsymbol{\alpha} \leftarrow \mathbf{A}_K^T \mathbf{r}$
$\boldsymbol{\alpha} \leftarrow T_{MAD}(\boldsymbol{\alpha})$
$\mathbf{w}_k \leftarrow \left\| \mathbf{A}_k^T \mathbf{P} \right\|_2$
$\alpha_{max} \leftarrow \max(\frac{\boldsymbol{\alpha}_k}{\mathbf{w}_k})$
$k_{max} \leftarrow k_{\alpha_{max}}$
for $n = k_{max}, k_{max-1}, \ldots, 0$ **do**
  $\alpha_{max} \leftarrow \max(\boldsymbol{\alpha}_n)$
  Remove structures from $\boldsymbol{\alpha}_n$ containing no coefficients greater than $\tau \alpha_{max}$
  if $n \neq k_{max}$ **then**
    Remove structures from $\boldsymbol{\alpha}_n$ for which $\boldsymbol{\alpha}_{n+1} = 0$
  end if
  $\mathbf{M}_n \leftarrow 1$ where $\boldsymbol{\alpha}_n > 0$
end for
return $\mathbf{M}, \mathbf{M}\boldsymbol{\alpha}$

2.5 CPUs, GPUs, CUDA, and PyCUDA

As previously mentioned, the next generation of radio interferometers will produce tremendous amounts of data. As a result, the rate at which the data can be processed into a form which is suitable for extracting science is of vital importance. Thus, any new algorithm needs to perform at least relatively quickly. This section discusses, albeit very briefly and without excessive detail, how hardware can be used to accelerate an algorithm. This can be considered a theoretical preview into how PyMORESANE has been accelerated.

In general, when producing computer code, the first goal is a working CPU-based (Central Processing Unit) implementation. A very basic, single-core CPU performs instructions, whether they are arithmetic or logical, sequentially. This is a bit of an oversimplification, as modern CPUs are far more sophisticated than can be described here. However, for the purpose of understanding the importance of parallelism, this simplification has no ramifications.

Thus, when computing something like $(a+b)+1$, a CPU does each operation separately and stores the intermediate results. To a user, an operation this simple will appear instant - even though the operations are performed one after the other, the answer will be calculated so quickly that it won't take a perceptible amount of time. However, if the previous expression was instead performed over and over again, there would be a noticeable delay before the result became available. This is referred to as a compute-bound process. The CPU utilisation will be at one hundred percent and the computation will take time. CPU-bound processes can naively be thought of a process that would complete faster if the CPU was faster.

The first alternative to compute-bound processes are I/O-bound processes. These processes are limited by the speed at which the data can be accessed. Typically, this will be a problem if large quantities of data are being read from disk. The data can only be copied into memory at a certain speed.

The second alternative to compute-bound processes are memory-bound processes. These processes are limited by either the quantity or speed of the memory involved. A frequently given example is the multiplication of large matrices as each entry has to be fetched from memory, operated upon and then stored in memory again. The speed at which a CPU can access memory is finite.

The principle problem encountered in MORESANE was with compute-bound processes. Working with large images means that there are several million pixels to deal with, even when performing a simple addition. This problem motivated the search for alternate

approaches to mitigating the effects of the compute-bound tasks which were slowing down the algorithm.

The first approach when faced with a such a problem is to investigate multiprocessing - using multiple CPU cores to perform more operations simultaneously, thus reducing the time taken. However, while such an approach seems simple, it can also bring with it a host of other problems (see Chapter 3).

Fortunately, there is an alternative to CPU-based multiprocessing - the GPU (Graphics Processing Unit). Modern GPUs are becoming more and more focussed on GPGPU programming and optimisation [26]. This turns a tool which was focussed primarily on graphics based calculations into a general purpose computing resource. The result is a massively parallel environment, which is conducive to computations on large problem sizes.

There are two primary GPU manufacturers, however, the following discussion will focus on Nvidia GPUs and their associated language - CUDA - which is an extension to C.

The most modern of Nvidia's GPUs are very complicated pieces of hardware. However, it is possible to explain their operation relatively simply. High-end GPUs have multiple SMs (Streaming Multiprocessors) each of which is made up SPs (Streaming Processors). This essentially allows them to be used as massively multi-core CPUs. There is some important jargon below which provides insight into GPU operation and its hierarchy.

When a function, usually referred to as a kernel, is executed on a GPU, it is executed on a grid. This grid is made up of a number of blocks, which are in turn made up of threads [27]. This hierarchy may seem confusing at first, but it provides a solid platform on which to build.

The lowest level of the hierarchy is the thread. Each thread can be thought of as a single tiny program or the body of a loop; one thread operates in much the same way as a single core CPU would. However, the latest Nvidia GPUs can handle several thousand such threads simultaneously; threads are grouped into blocks, the size of which is based on the GPU in question, in order to make manipulating them simpler.

Each thread block is assigned to a single SM and the threads within the block are executed concurrently. Blocks are scheduled in warps, which consist of 32 threads. Instructions are performed per warp, which gives rise to a very important property of GPU optimisation - warp divergence. If threads within a thread block would follow different execution paths, as a result of an if statement for example, in reality all the threads follow both execution paths. While this does not alter the results - the threads

executing in the wrong path are disabled - it can have a large impact on performance if the divergent execution paths are long.

Blocks are logically ordered into a grid. Thus, when a given SM concludes execution on its current thread block, the GPU allocates the next block in the grid to the now-free SM. An SM can actually process multiple thread blocks simultaneously if they have sufficient available resources.

The latest Nvidia Tesla GPUs have up to 15 SMs and an associated 2880 CUDA cores. Each SM can handle 64 warps, which amounts to 2048 concurrent threads. This level of parallel processing power can be used to produce a massive speed-up in certain applications.

Fortunately for the development of PyMORESANE, a Python wrapper for CUDA routines exists - PyCUDA [28]. This handy module removes a great deal of the technical difficulties involved when producing code for the GPU and allows for CUDA C code to be written directly into Python.

Kernels written in C are compiled at runtime and cached so that subsequent uses or calls do not require additional compilation. Thus, while PyCUDA is not quite as fast as code written in C/C++, it is convenient and still offers the substantial speed-up available on GPUs.

Chapter 3

Implementation Details

The following chapter discusses the details of implementing the MORESANE algorithm in Python and its subsequent acceleration. It is presented as a series of sections, each of which explains the implementation of a component of the overall algorithm. The final section details the manner in which the components fit together. PyMORESANE is freely available from GitHub (https://github.com/ratt-ru/PyMORESANE) and should be regarded as a digital appendix to this section. The details presented here will not be exhaustive, but should be reviewed in conjunction with the code itself; it is thoroughly commented.

Before embarking on an explanation of the individual components, it is first necessary to make some comments about the overall layout of PyMORESANE. The program is broken up into modules, each of which contains a set of specific tools. This is a particularly useful property of Python as it allows for logical chunking of functionality. Thus, the IUWT decomposition, recomposition and the "á trous" algorithm they require appear in a separate module from that which handles source extraction.

3.1 Implementing the IUWT

The IUWT is one of the fundamental building blocks of the MORESANE algorithm and, due to the iterative nature of the algorithm, it is invoked many times. Additionally, although the computational complexity of the IUWT may be reduced by the "á trous" algorithm, when operating on large N-by-N arrays it is still a relatively time-consuming process. As problem sizes are restricted to being powers of two in order to ensure that the FFT operates at maximum efficiency, doubling the array dimensions quadruples the number of mathematical operations required to arrive at the solution.

With the above factors in mind, the first step in developing PyMORESANE was a rudimentary CPU-based implementation of the IUWT using fairly standard Python modules. The original MATLAB implementation provided an excellent starting point. However, the original was implemented using explicit loops. While this approach works and is relatively fast in MATLAB, it necessitates the evaluation of multiple if statements in order to handle edge effects.

Edge effects arise from the convolution of the finite length IUWT filter-banks with the signal. When the filter is applied sufficiently close to the edges of the signal, it is necessary to apply some sort of border condition. In MORESANE, the edges are treated by mirroring the signal data. This approach can produce unwanted effects and does require additional computation.

Fortunately, NumPy [29], a Python module which handles array-based mathematics, is particularly well suited to the problem. It allows for arrays to be indexed with negative values, such that items may be accessed relative to the final element in a dimension as opposed to the first. Additionally, it is very simple to slice arrays using this indexing. These factors make dealing with edge effects substantially simpler, as pieces of the signal may be treated in different ways without needing to assess whether an individual entry falls on the edge.

To this end, the "á trous" algorithm has been re-implemented using NumPy. This performs what amounts to a convolution of the signal and is an important piece of the overall IUWT. The implementation requires a two-dimensional input array, a scale and an appropriate filter. The scale, as explained in subsection 2.4.2, determines the number of zeroes between the non-zero entries of the filter. However, it is not necessary to explicitly dilate the filter, as doing so would massively increase the computational cost. The beauty of the "á trous" algorithm is that since the filter has only a finite number of non-zero entries, convolution may be performed by summation over only the non-zero elements in the filter.

In practice, the one-dimensional IUWT filter-bank has five non-zero elements. In two-dimensions, due to the separability of the filter, the one-dimensional filter can be applied first along one axis, and then the other. Thus, in the absence of edge effects, the implementation consists of selecting each entry in the input and summing the products of the filter with the input. The resut is stored at the position of the selected entry. The incorporation of mirroring at the edges does not massively alter the computational cost but does introduce an additional degree of complexity.

This complexity arises from the need to slice the input into two sections for each product - one section is unaffected by edge effects while the other takes the mirroring into account.

The position at which the input is sliced is determined by the scale and each product corresponds to one element of the filter. Thus, there are five products per element, four of which need to account for edge effects. The central element of the filter can never fall outside the input.

In order to obtain a set of detail coefficients for a given scale, the decomposition requires two successive applications of the "á trous" algorithm. The detail coefficients are obtained from the difference between the two results. To clarify, the detail coefficients are obtained from the difference in the approximation coefficients at two successive scales. Additionally, determination of the detail coefficients at a given scale is dependant on the approximation coefficients ($\mathbf{c}_j$) of all the scales preceding it.

Therefore, an additional option has been incorporated into the IUWT: scale-adjustment. This option omits the calculation of detail coefficients which are not of interest. To elaborate, if certain scales are to be ignored during an iteration, there is no reason to calculate more than their approximation coefficients. This means that only one application of the "á trous" algorithm is required at those scales.

The above description corresponds to the *ser_iuwt_decomposition* and *ser_a_trous* functions in the code. However, after implementing these functions, it was of interest to see whether or not their performance could be improved. Some rudimentary tests showed that for larger problems and higher scale decompositions (e.g. 4096-by-4096 pixel image, 8 scales), a single decomposition was taking several seconds to execute. Several hundred decompositions are required over the course of MORESANE, and as such the IUWT decomposition becomes a computational bottle-neck.

It was with this in mind that the second approach to the IUWT decomposition was implemented. These functions - *mp_iuwt_decomposition* and *mp_a_trous* in the code - were adapted to make use of the Python multiprocessing module.

Adapting code for multiple CPUs or CPU cores is not a straightforward process. However, due to the aforementioned separability of the IUWT filter-bank, when using the "á trous" algorithm each element depends only on elements which share its row and column. Additionally, all calculations along one axis are performed prior to computations along the second axis.

Thus, a logical approach is to break the input into strips; first along one axis and then the other. Each CPU can then be allocated a strip on which it can perform the necessary computations. Once all the calculations for one axis have been completed, the intermediate result can be assembled and split up again along the second axis. There is, however, a problem with this approach.

The problem stems from the splitting of the input. This operation is relatively time-consuming, but recombining both the intermediary and final results is even more so. In order to mitigate this issue, a slightly different approach has been adopted, which makes use of shared memory to prevent the difficulties introduced by genuinely slicing the input.

Thus, before slicing the input and allocating it to the CPUs, it is first instantiated as a shared memory object. This means that multiple processes can access it simultaneously and all such processes can see its current contents. Thus, slicing no longer requires an explicit memory copy. Instead, each CPU operates only on its allocated piece of shared memory. This makes reassembling the output unnecessary, as the result will be correctly stored in shared memory.

This approach has been successful. However, it has several downsides. In particular, spawning the additional processes proves to be a relatively expensive task. Thus, the multiprocessing approach is of limited use for small- and mid-sized problems. However, as will be investigated in the results chapter, it may be more viable for very large problems.

Noting both the successes and failures of implementing the IUWT decomposition and "á trous" algorithm using CPU-based multiprocessing, a GPU-based solution was the next obvious step. Given no prior knowledge of GPU computing, implementing the above is by no means trivial, and the implementation has been through several evolutions. The code - *gpu_iuwt_decomposition* and *gpu_a_trous* - differs vastly from either the serial or multiprocessing approaches.

The first major difference is the additional work that goes into ensuring that the input arrays are of the correct data type. GPUs and PyCUDA are far more stringent regarding both type and size. In this respect, PyCUDA differs vastly from ordinary Python. For the purposes of PyMORESANE, all GPU calculations were performed at 32-bit precision. While 64-bit is preferable for accuracy, GPUs are far less efficient for 64-bit floats than for 32-bit floats. Thus, in order to guarantee a speed-up, the all inputs were cast as 32-bit floats.

Another major consideration when employing GPUs is the expense of memory copies. In order to get the maximum speed-up using GPUs, data needs to be moved to the GPU's RAM. In fact, this is usually the most prohibitive aspect of using GPUs. If the memory copies are too frequent or too large, more time is wasted on moving the data than is saved by the processing power of the GPU. Many of the revisions to the GPU implementation have been motivated by a desire to reduce the number and frequency of memory copies.

In order to make the IUWT as lightweight as possible, there is a degree of obfuscation in the code. Due to the manner in which GPUs operate, the row and column convolutions which are used by the "á trous" algorithm had to be separated. This is due to the fact that every column convolution must be performed before every row convolution or vice-versa. Once a GPU kernel is launched, there is no guarantee of which threads will finish first. Thus, it is necessary to have two distinct kernels which can be executed successively, rather than one.

In fact, the "á trous" code for the GPU implementation does not actually perform any calculations. Instead, in order to preserve the implementation structure present in the single-core and multiprocessing code, it contains the kernels corresponding to the row and column convolutions. Thus, the GPU implementation performs the convolutions in the body of the IUWT code, rather than running the "á trous" function directly.

A side effect of this approach is the need for an array in which the intermediate values can be stored. Fortunately, such an array can be created on the GPU, which is considerably less expensive than creating an empty array and transferring it. However, as the IUWT requires the results of two successive convolutions, additional arrays are also necessary to store those outputs. To this end, several arrays are created and reused to reduce the memory requirements of the algorithm. This does, however, make the code a little less readable.

Whilst improving the most computationally expensive portion of the IUWT for GPUs - the "á trous" algorithm - it became apparent that the entirety of the IUWT could be performed on the GPU. This realisation has repercussions which will be discussed in section 3.4. In the short term, however, the realisation motivated further work on the IUWT decomposition. The result is code which will accept either GPU arrays (PyCUDA array which efficiently handles memory copies) or NumPy arrays from main memory and performs the IUWT decomposition upon them.

Once the initial array is on the GPU, the result is computed entirely on the GPU, employing an additional kernel for storing the detail coefficients.

The final GPU implementation has been found to be substantially faster than either the single-core or multiprocessing cases, and retains all of their features, including the ability to disregard uninteresting detail coefficients by means of a scale-adjustment parameter.

The discussion thus far has been concerned primarily with the IUWT decomposition. The IUWT recomposition has been implemented in an analogous fashion, starting from a serial approach and building up towards an efficient GPU implementation. The "á trous" algorithm is also employed in the recomposition algorithm and the implementations already described remain perfectly applicable.

A lengthy explanation of the IUWT recomposition implementation is not informative. It is simply the addition of the detail coefficients at each scale convolved with the appropriate filter-bank. In this case, as mentioned in the theory sections, the filter bank is the same for the decomposition and recomposition, hence the use of the "á trous" algorithm in the reconstruction.

The single-core and multiprocessing IUWT recomposition implementations suffer from the same bottle-necks and problems as their decomposition counterparts. However, once again the GPU implementation proves to be efficient and, following the same evolution as the decomposition, it ultimately occurs entirely on the GPU subsequent to the initial memory copy. Results appear in chapter 4.

The IUWT recomposition incorporates the same scale-adjusting behaviour as its counterpart, which allows the reconstruction of signals using only the detail coefficients belonging to the scales of interest.

One additional feature, unique to GPU-based implementations is the ability to leave their output on the GPU and return a handle to the GPU array rather than the explicit values which could be stored in main memory. This option, has further significance in section 3.4.

The final version of the IUWT implementations can be found in the iuwt.py module at `https://github.com/ratt-ru/PyMORESANE/blob/master/iuwt.py`.

3.2 Implementing Object Extraction

Object extraction is a multi-stage process, the aim of which is the construction of a set of significant coefficients. The first step in pursuit of these coefficients is an IUWT decomposition of a dirty image. The decomposition is stored as a three-dimensional array, for which the third dimension corresponds to the scale of the decomposition. As explained in section 2.4.3, object extraction consists of a de-noising operation, followed by a search for significant structures and, subsequently, objects.

The de-noising operation is performed in the main body of the PyMORESANE code by a function called *threshold* from the IUWT toolbox module. This module contains the majority of the non-IUWT, non-FFT functions required by PyMORESANE.

The thresholding operation is trivial to implement. Given a three-dimensional input, the implementation simply iterates over the number of scales, calculates the MAD estimator for each scale and then zeros out the values below the MAD estimator multiplied by

a specified sigma level. This sigma level adjusts how close to the estimated noise the algorithm will attempt to detect structures.

Little was done to alter this thresholding procedure. However, it is interesting to note that the thresholding operation is by no means lightweight or efficient. Specifically, computing the MAD estimator is expensive. Is based on a median value, and computation of a median value amounts to a sorting procedure. On large problem sizes with high scale counts each scale might require sorting, by means of an example, 16777216 pixels (for a 4096-by-4096 pixel image) at up to 10 or possibly more scales. A simple test shows that even sorting a 16777216 entry list takes over a second on ordinary hardware. Multiplied by the number of scales or an even larger problem size, this can be prohibitive.

However, whilst GPU sorting algorithms exist, they are not easily available in Python and are not included in the basic PyCUDA functionality. As a result, the calculation of the MAD estimator and the subsequent thresholding procedure have been left unaccelerated.

The thresholding procedure takes place outside of the actual object extraction function as some additional operations are performed prior to the invocation of *source_extraction*.

The source extraction (equivalent to object extraction) procedure was recursive in the original MATLAB implementation, as was the initial Python implementation. However, it scaled very poorly with the number of objects in the image. Another approach has been devised which differs in some respects but which has been shown empirically to give identical results but at a far lower computational cost.

The basic source extraction algorithm only requires the thresholded coefficient set and a tolerance value, τ, as explained in section 2.4.3. Inside the source extraction code, the first step is the determination of connected pixels (structures) at each scale. For this purpose, the ndimage module from the SciPy package [29] was exceptionally convenient. It has inbuilt functionality for constructing label images.

A label image is constructed by determining which adjacent pixels are non-zero and then assigning an integer index to the structures discovered in this fashion. Thus each integer up to the number of structures identifies a single group of connected pixels.

The subsequent step is a second thresholding procedure which removes all of the coefficients below some tolerance τ of the maximum coefficient at the current scale. The remaining coefficients belong to the structures of interest.

However, the aim is the recovery of all coefficients belonging to the significant structures rather than only those left by the second thresholding procedure. This is achieved by finding the unique indices of the structures left after the second thresholding procedure.

This list of unique coefficients is used, in conjunction with the original labelled array to create a binary mask for the coefficients of interest.

The process is performed iteratively, starting from the largest scale. This is necessary as there is an additional consideration. Ultimately, the goal is the extraction of objects or sources which are comprised of connected structures. In order to determine the vertical (inter-scale) connections between structures, at each iteration after the first, the significant structures are both thresholded and multiplied by the mask for structures at the previous (higher) scale. This removes the need for a recursive procedure as connections are formed to lower scales only if they have significant coefficients which fall inside the boundaries of the structures at the higher scale.

The function returns the final binary mask for significant sources as well as the the product of the mask with the detail coefficients - the coefficients belonging to the sources.

The source extraction procedure has an erratic execution time due to its dependence on the number of significant structures in the wavelet coefficients. For large images of complex fields, the number of structures may be prohibitively high. Fortunately, the problem is also readily adapted to the GPU.

The approach is virtually identical to the CPU case as the initial masking and thresholding procedures are still performed by the CPU. However, there is the usual additional setup and type-casting necessary for the GPU.

The GPU is used, following the determination of the unique labels belonging to the significant structures, to construct the binary mask for the significant structures at each scale. To this end, the implementation iterates over the unique labels and the GPU is used to locate the structures corresponding to the labels. Structures thus located appear in the binary mask.

Unfortunately, due to the computational cost incurred in transferring data to the GPU, the GPU implementation is not substantially faster for low object counts. This is exacerbated by the fact that the number of computations per data transfer is low, and the implementation becomes memory-bound rather than compute-bound. However, it scales much better with object count than the CPU implementation.

Additional functionality is included in order to retain a copy of the final binary mask on the GPU. This is used to accelerate the main body of PyMORESANE.

The object extraction implementations form part of the iuwt_toolbox.py module (https://github.com/ratt-ru/PyMORESANE/blob/master/iuwt_toolbox.py).

3.3 Implementing the FFT and Convolution

The FFT is used frequently during the MORESANE algorithm, particularly for performing the large convolutions which are necessary to deconvolve the image. While neither the FFT nor convolution are unique to MORESANE, they are both computationally expensive operations. Due to their impact on performance, several steps have been taken to accelerate them.

For the basic CPU-based implementation of PyMORESANE, both the FFT and convolution are trivial to implement, as they are both built-in NumPy functions. These implementations are by no means poor, however they are ill-suited to large problem sizes. In order to mitigate this problem and accelerate the algorithm, a GPU-based solution has been devised.

In truth, this was the first portion of PyMORESANE to be accelerated by GPU. Implementing the FFT from scratch is in itself a considerable task. Fortunately, the CUDA SciKit [30] Python package contains GPU implementations of the FFT which can be used from Python. These FFT functions require more setup than the conventional NumPy FFT, but do perform substantially faster.

In order to make using the GPU FFT as simple as possible, an additional module has been written to handle all FFT functionality. The most basic functions, *gpu_r2c_fft* and *gpu_c2r_ifft*, which correspond to the forward real-to-complex transform and the inverse complex-to-real transform simply perform the necessary setup before invoking the CUDA code via the SciKit interface.

A note on setup is necessary here, as it is illustrative of the behaviour of the GPU. In order to carry out the FFT, the output array needs to be pre-allocated on the GPU. Thus, prior to calling the FFT code, the size of the output has to be determined. As the input is strictly real, the real-to-complex transform can be used. This is slightly faster than the alternative complex-to-complex transform, as the Fourier transform of a real-valued signal is hermitian. Thus, only half the resulting values and the constant term needs to be stored. This reduces the problem size, which of course increases its speed. The same is true of the inverse transform; given an array with only half the size of the original image, it is necessary to preallocate a full size array to accommodate the result of the inverse transform.

Although the FFT is of importance in its own right, its principal use for for carrying out convolution by performing a multiplication in the Fourier domain as opposed to the prohibitively expensive task of performing explicit convolution. In order to achieve this, a special convolution function - *fft_convolve* in the code - has been written.

This convolution function incorporates both CPU and GPU functionality, allowing either mode to be used interchangeably with the specification of some optional parameters. This is crucial, as otherwise it would not be possible to run PyMORESANE on a variety of platforms. Additionally, the function allows for both linear and circular convolution. The distinction here is in how edge effects are handled.

In the case of linear convolution, which is more computationally expensive, the inputs are padded so that the edge effects do not appear within the final result. Circular convolution does not pad the inputs, which can lead to odd behaviour along image borders.

Many implementations of the Fourier transform produce output with the quadrants shifted. In order turn the shifted output into what is expected, most implementations make use of a fftshift operation, which swaps the offending quadrants around. There is no inbuilt method for this in the CUDA SciKit. Thus, in the initial implementation, the NumPy fftshift method was used to correct the output. However, this operation is relatively expensive and requires that the array be transferred back to main memory.

An additional problem had to be addressed when making use of padded arrays. When the final answer is obtained from the FFT, only the central region is of interest. As such, it is desirable to store only the region of interest in main memory. However, memory copies have to be performed on contiguous data: that is, the all elements of the data have to be adjacent in memory. Slicing an array violates this property, and thus any array which has been sliced cannot be retrieved from the GPU.

The above motivated the development of additional GPU kernels to handle both the fftshift operation and the second problem introduced by slicing the output.

The *fft_shift* function in the code is relatively rudimentary. Given an input array, it determines the length of the array axes, and consequently identifies the quadrants. The elements of the diagonally opposite quadrants are then swapped in order to produce the correct output.

The *contiguous_slice* function slices the central region out of a GPU array, but also proceeds to make it contiguous. This is achieved by creating a new array on the GPU and copying the elements of interest directly into the new array. The new array is allocated in a contiguous memory block, and thus copying to it element by element preserves its contiguity. This ensures that the result can be copied from the GPU and into main memory.

All the functionality required for performing the tranforms and convolution appear in the iuwt_convolution.py module (https://github.com/ratt-ru/PyMORESANE/blob/master/iuwt_convolution.py).

3.4 Implementing PyMORESANE

The following section details the implementation of the main body of PyMORESANE. The previous sections all form part of this larger whole. Figure 3.1 provides an overview of the operation of PyMORESANE. The assosicated code appears in pymoresane.py (https://github.com/ratt-ru/PyMORESANE/blob/master/pymoresane.py).

3.4.1 Setup

The main body of PyMORESANE is couched in object oriented environment. This means that a custom *FitsImage* class has been implemented to hold the dirty image and the PSF, and the MORESANE algorithm is simply a method of a *FitsImage* object.

The object-based approach does offer some useful features. In particular, having attributes (object level variables) means that once the object is instantiated, it is very simple for its associated methods to modify those attributes. Thus, attributes can be seen as global variables with respect to the object. This is helpful, as it provides a simple method for storing results without having to explicitly handle return statements.

Several attributes are associated with *FitsImage* objects. However, most are simply used for handling the FITS (Flexible Image Transport System) format in which the input images are stored. To this end, the PyFITS module [31] is the go-to tool. It allows for FITS files to be handled with ease. Thus, many of the attributes are portions of the FITS file, such as the data or its associated descriptive header. The outputs are also instantiated as attributes of the object to ensure that all results are stored once the *moresane* method has been called.

It should be noted that *moresane* accepts many parameters, all of which are explained within the help function. This help function forms part of the *pymoresane_parser* module which handles command line input to algorithm. Additionally, many of the parameters are optional, and the defaults will be fine in most cases. However, all GPU functionality is controlled by means of these parameters and is disabled by default.

Additional setup is performed inside the *moresane* method, including the assignment of default values which are dependant on the non-optional input (dirty image, PSF, output location).

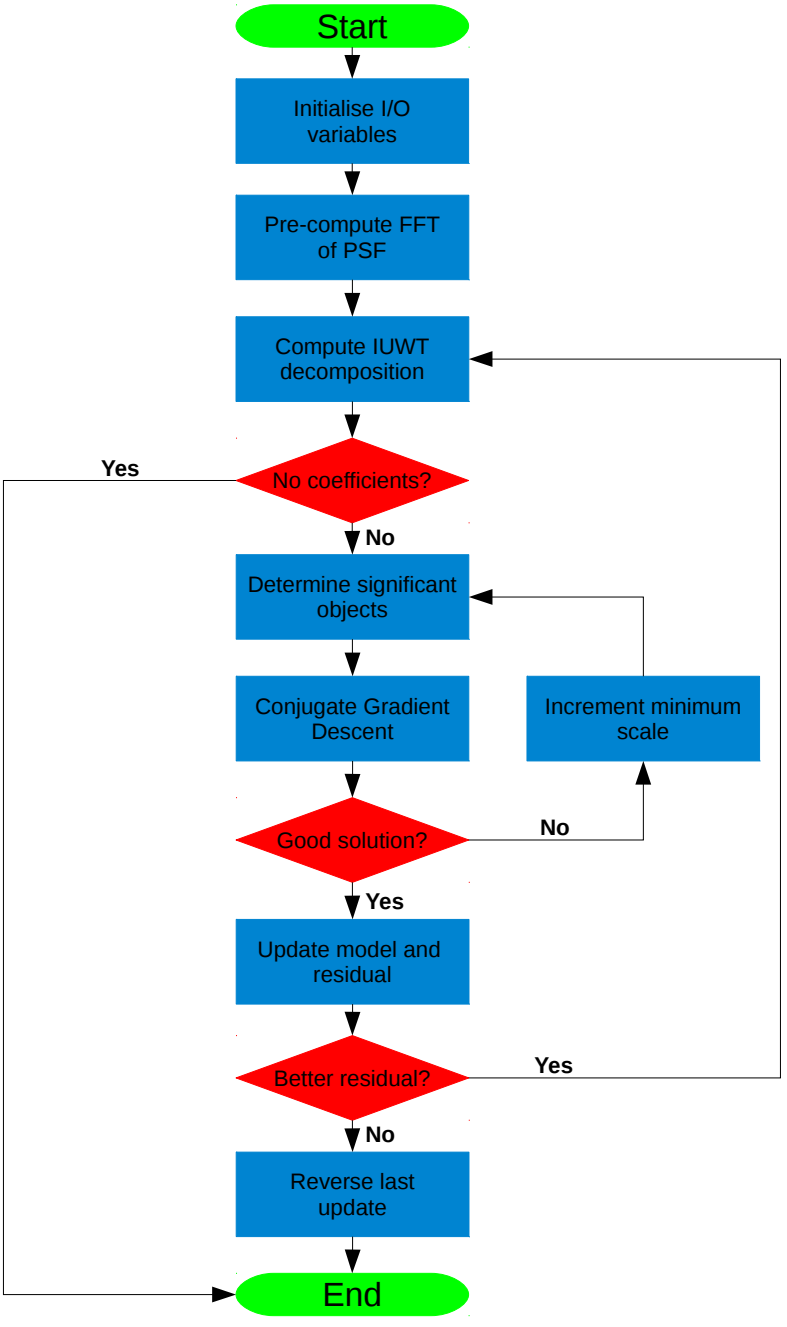

FIGURE 3.1: Simplified flowchart which details the operation of PyMORESANE.

An important inclusion in PyMORESANE is a user parameter which allows specification of a subregion of the dirty image which is to be deconvolved. PyMORESANE constructs a Python slice object which corresponds to the region of interest, and uses it to select the relevant area of the dirty image. Deconvolution may thus be restricted to a limited area. This is helpful when running the algorithm on systems with limited RAM, or when the signal of interest is known to be very localised within the image.

One piece of the additional setup is of vital importance to the acceleration of PyMORESANE; the pre-computation of the Fourier transform of the PSF. This quantity does not vary and as it is used regularly throughout the algorithm. Thus, there is no need to recalculate it and large amount of time can be saved. However, PyMORESANE goes one step further and exploits the GPU in its pre-computation.

Given the specification of some optional parameters, not only does the pre-computation take into account whether or not the PSF should be padded, but also determines whether the transform should be computed and stored on the GPU. Whilst this naturally means a portion of the GPU's memory is perpetually occupied by the result, it massively accelerates the convolution step of the algorithm as no additional memory copies or calculation are required for the PSF.

In fact, PyMORESANE improves on the original MATLAB implementation even further by allowing the use of a PSF which is double the size of the dirty image. This completely removes the convolution artefacts which usually accrue around the edges of deconvolved images. These artefacts are the products of incomplete convolution.

Unfortunately, doubling the dimensions of the PSF increases the computational expense of calculating its transform. However, it does mean that PyMORESANE may deconvolve full dirty images, rather than restricting itself to the central quadrant.

The final piece of setup is the pre-computation of the weighting factors for each scale, based on the IUWT decomposition of the PSF. These weighting factors are used to ensure that the maximum detail coefficient is correctly located. Prior to weighting, it is not possible to objectively locate the maximum.

3.4.2 The Major Loop - Part 1

The major loop of PyMORESANE consists of a while loop which iterates until such time as a stopping criterion has been reached. There are several criteria for the major loop, two of which are based on user-specified values; the maximum number of permissible iterations and the accuracy of the result. The final criterion terminates the while loop if analysis of the detail coefficients produces no structures.

The purpose of the main loop can be broken down into steps, the first of which is the IUWT decomposition of the dirty image, using the previously described implementation (section 3.1). This decomposition produces the set of detail coefficients which are of interest for the current iteration. The de-noising procedure, as mentioned in section 3.2, is carried out immediately after the calculation of the detail coefficients.

Additional edge suppression functionality, unique to PyMORESANE, is also included in the initial stages of the major loop. Edge suppression has been implemented in two ways. The first allows user specification of a number of pixels to ignore at each edge. This parameter allows users to tune how close to the edge of the dirty image they wish the algorithm to deconvolve. As a result, even in the absence of a double PSF, edge effects can be reduced.

The second edge suppression technique zeroes out all coefficients in the decomposition which are subject to edge effects. Thus, low scales are barely affected, but high scales will lose many coefficients. This is necessary as the mirroring border condition can create fictitious diffuse emission at the image edges. This is exacerbated, in the absence of a double-size PSF, by border artefacts from the convolution.

Both edge suppression techniques have been implemented by multiplying the de-noised coefficients by a binary mask. The binary mask is constructed based on either the static offset value or the calculated value for edge-corrupted coefficients. Both techniques have been shown to reduce the number of false detections in images for which a double-size PSF is unavailable.

Regardless of whether or not the de-noised coefficients have edge suppression applied, the location of the maximum coefficient is determined following a weighting procedure of the maxima at each scale. If the maximum coefficient is found at a scale lower than the total number of scales, scales above that of the maximum coefficient are ignored for the current iteration.

An analogous procedure has been implemented for determining the scale-adjustment parameter as described in section 3.1. However, instead of locating and removing scales above the maximum, the scale-adjust parameter is determined from the scales below the maximum. A scale, and all scales below it, is ignored if it is found to have no non-zero components after the de-noising operation.

Thus, only coefficients which lie at scales between the scale-adjustment value and the scale of the maximum coefficient are considered in the object extraction procedure. In fact, it is at this point in the implementation that the *object_extraction* function from the *iuwt_toolbox* module is used, as described in section 3.2.

It is at this juncture in the implementation that the minor loop occurs. However, prior to its execution, the output from the object extraction procedure, the significant coefficients, is recomposed into a single image using the IUWT recomposition. The result is important in the minor loop.

3.4.3 The Minor Loop

In order to preserve the logical flow of this discussion, the minor loop must be discussed prior to concluding the discussion of the major loop.

The minor loop corresponds to the conjugate gradient descent method, as mentioned in section 2.4.3. This portion of the algorithm is responsible for constructing an approximation of the true sky brightness distribution by minimising the objective function in equation 2.27.

The initial implementation was identical to that of the original MATLAB version. Thus, starting from the recomposition of the significant objects in the dirty image, the method attempts to solve for the sky brightness iteratively. This is implemented in accordance with generic pseudo-code in [32]. The only major difference is the addition of a step to ensure the positivity of the solution.

In order to achieve this, during each iteration the computed approximation has all negative values removed, if such values exist. Then the descent direction is recomputed such that the modified non-negative solution is consistent.

A line-by-line discussion of the conjugate gradient descent method is unnecessary, as it is a fairly common numeric solution method. However, one of the most crucial improvements of PyMORESANE actually appears in the conjugate gradient descent implementation.

Several of the previous implementation sections, particularly sections 3.1, 3.2 and 3.3, have alluded to the fact that PyMORESANE includes functionality to retain the results of many of the GPU-based implementations on the GPU itself, rather than returning the results to main memory. All this functionality has been building towards this portion of the minor loop.

The conjugate gradient descent method, and in particular the successive application of the above operations, is the single most computationally expensive portion of PyMORESANE as it is an expensive iterative procedure that is nested inside a loop. The GPU implementations massively reduce that cost.

Specifically, the conjugate gradient descent method requires a convolution between the approximate solution and the PSF, an IUWT decomposition, a multiplication by the significant objects mask and finally an IUWT recomposition. All of these functions have GPU implementations and all of them may retain their results on the GPU.

This allows the above sequence of operations to be performed entirely on the GPU. In the event that PyMORESANE is run in this mode, only after the IUWT recomposition in the result returned to main memory. This massively reduces the number of memory copies required, thus accelerating PyMORESANE even further.

As the conjugate gradient descent method is iterative, it iterates until such time as one of multiple stopping criteria is reached. These criteria have been implemented in accordance with the original. They are based on the SNR between sources extracted from the dirty image and the associated sources derived from the approximation of the sky brightness. A simple convenience function for computing the SNR has been implemented, however it is of little interest.

Depending on the value of the SNR, the minor loop determines whether the approximation has been successful, whether it needs to continue iterating or whether it is necessary to try again. In the event that the SNR becomes either too high too quickly or too low, the major loop will be rerun using one fewer of the low scales. This often ensures better behaviour, as noise and false detections tend to exist at the lower scales.

If the SNR begins decreasing, but prior to the decrease it was sufficiently high, or exceeds an upper limit, the approximation is deemed successful and the minor loop is terminated.

3.4.4 The Major Loop - Part 2

Once the minor loop has terminated, PyMORESANE returns to the major loop, the second half of which is responsible for updating the attributes of the *FitsImage* object.

To this end, the model image is updated with some percentage of the sky model approximated at the current iteration. This is analogous to the loop gain of the traditional CLEAN algorithm. The percentage which is added on each iteration is a user-specified parameter. Once the model has been updated, it is possible to perform the actual deconvolution step.

In order to achieve this, the model image is convolved with the PSF. The result is then subtracted from the original dirty image to produce the residual. As PyMORESANE iterates, the model image grows more complete and as such the residual improves.

Some additional stopping criteria are handled towards the end of the main loop. In particular, if the minor loop doesn't manage to produce an accurate approximation at any scale, PyMORESANE terminates. Additionally, although the SNR is a decent measure of the accuracy of the model, it doesn't necessarily mean that the residual will improve after the deconvolution step.

Thus, the major loop also determines whether or not the standard deviation of the residual is decreasing. In the event that deconvolving the contribution of the clean image from the dirty image does not improve the standard deviation of the residual, PyMORESANE reverts the last update to both the model image and the residual.

These updates are the last step in the original implementation of MORESANE, bar the subsequent saving of the resulting residual and model images as separate FITS files.

3.4.5 Updating PyMORESANE

Whilst PyMORESANE was in development, the original algorithm was updated, thus altering its functionality. PyMORESANE retains the original version but also includes the changes in the second method to the *FitsImage* class - *moresane_by_scale*.

This approach is slightly different as instead of running the algorithm once, allowing all scales up to some user specified value to be considered, MORESANE is applied iteratively. Each iteration allows the inclusion of one additional high scale.

Implementing this after developing the basic functionality of PyMORESANE was relatively simple. The *moresane_by_scale* method simply invokes the *moresane* method and adjusts the relevant scale parameters. Only a few additional changes to the original *moresane* method were required.

The first was the inclusion of a flag as an attribute of a *FitsImage* class. This flag allows the *moresane* method to terminate the execution of *moresane_by_scale* in the event that a single iteration of *moresane* does not achieve anything.

The second change is very minor but entails altering the manner in which the *moresane* method updates the model and residual. This is both necessary and important as the the result from each iteration of the *moresane* method must be available at subsequent iterations. Additionally, it is vital that iterations which perform no changes do not overwrite the existing data.

3.4.6 Additional Features

Some additional features which are not part of the algorithm itself but which are nevertheless important to PyMORESANE are detailed here.

Firstly, PyMORESANE does have a logger, thus allowing it to save its terminal output to a log file. The log level may be set from the command line. The output contains useful values and will show errors in the event of unexpected operation.

More important that the logger is the *restore* method. This method constructs and saves the restored image by making use of the *beam_fit* module. The module contains functions which fit an elliptical Gaussian to the central region of the PSF. The resulting Gaussian is convolved with the model image and added to the residual to produce the restored image. This image is then saved in its own FITS file with the addition of the so-called clean-beam parameters to its header.

Chapter 4

Results: Acceleration

This chapter presents the results of accelerating the various pieces of the MORESANE algorithm. The initial sections show the improvements to the individual implementations while the final section deals with the acceleration of PyMORESANE as whole. For the purposes of benchmarking the performance of the algorithm, the code was executed on a server running two Intel Xeon E5-2690 CPUs (8 physical cores per CPU, 16 threads per CPU) at 2.90GHz, 512GB of DDR3-1333 RAM and an Nvidia Tesla K10 GPU. The results were obtained using cProfile which is a low-overhead C extension for profiling Python code.

A brief introduction on the use of PyMORESANE appears in Appendix A.

4.1 The IUWT

The various implementations of the IUWT decomposition have been tested on random data of varying size. Each array was populated with radomly positioned delta functions. For the purpose of this test, the input array dimensions were chosen to be increasing powers of two; the case for which the implementation was designed. Additionally, all decompositions were performed up to eight scales.

In order to obtain relatively accurate results, each implementation was run ten times for each data size. The average of the execution time was calculated from the resulting values. Whilst averaging over more results would have been better, it was not practical. In particular, the single-core implementation is very slow for large problem sizes and, in order to retain consistency, the sample size was restricted so that results could be obtained in a reasonable amount of time.

Implementation	Image Dimensions (N,N)	Average Execution Time (s)
Single-core	(512,512)	0.1092 ± 0.0057
	(1024,1024)	0.5713 ± 0.0117
	(2048,2048)	2.7129 ± 0.0199
	(4096,4096)	19.1231 ± 0.4860
	(8192,8192)	73.3205 ± 2.0039
Multi-core (4 cores)	(512,512)	0.3004 ± 0.0082
	(1024,1024)	0.8171 ± 0.1320
	(2048,2048)	3.1909 ± 0.5309
	(4096,4096)	13.5226 ± 2.4594
	(8192,8192)	41.0192 ± 3.7574
GPU	(512,512)	0.0627 ± 0.1582
	(1024,1024)	0.0424 ± 0.0010
	(2048,2048)	0.1667 ± 0.0048
	(4096,4096)	0.6702 ± 0.0170
	(8192,8192)	2.5965 ± 0.0339

TABLE 4.1: Comparison of the average execution times with problem size for the various IUWT decomposition implementations.

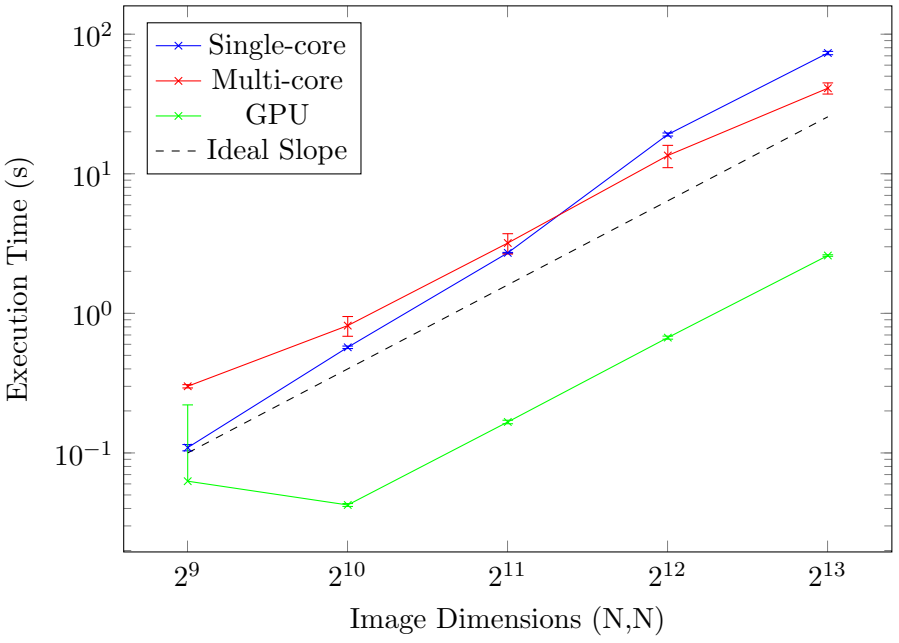

FIGURE 4.1: Plot of average execution time against data dimensions for the various IUWT decomposition implementations. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

The standard deviations of the execution times have been included, although they are relatively low in almost all cases. This in itself is an important result as there is not a great deal of scatter associated with the implementations. One overarching point is that the multi-processing implementations tend to have a larger scatter due to their dependence on CPU scheduling.

Table 4.1 shows the numeric results of profiling the IUWT decomposition and figure 4.1 is a plot of the same information. Both the table and the figure provide some interesting insight into the behaviour of the various implementations.

The first and most obvious improvement is the massive speed increase of the GPU implementation relative to both the single- and multi-core implementations. It is, however, important to the note that there is no reference implementation - the single-core implementation is not perfectly optimised. Regardless, the improvement in the GPU implementation is sufficiently large - more than an order of magnitude at all but the smallest problem size - that even perfectly optimised single-core CPU code would be unlikely to outperform it.

The single-core implementation is relatively slow, particularly for large problem sizes. For smaller problem sizes, the execution times scale as expected - quadrupling the problem size by doubling the data dimensions quadruples the execution time. However, between problems of dimensions $(2048, 2048)$ and $(4096, 4096)$, there is a discrepancy in the scaling and a corresponding increase in execution time. This suggests that once the problem exceeds a certain size, additional overhead is incurred. This is likely to be the result of fetching values from RAM.

The multi-core implementation has the most variation of the implementations. Initially, it is slower than the single-core implementation. This is the result of the overhead accrued in spawning the additional processes, which seems to be large in Python. However, there is a point at which the overhead is outweighed by the improvement in calculation speed. This point occurs between problems of dimensions $(2048, 2048)$ and $(4096, 4096)$, after which the multi-core implementation is faster than the single-core implementation.

The scaling of the GPU implementation is somewhat strange. It is difficult to draw a comparison for the smaller problem sizes as the execution time in dominated by overhead. However, for problems of size $(1024, 1024)$ and larger the GPU has near-perfect one-to-one scaling; quadrupling the problem size quadruples the execution time. This is ideal and matches up with expected gradient presented in figure 4.1. The wavelet transform should scale as $\mathcal{O}(N)$ for a problem of N pixels. Careful inspection of the data does reveal that, for the largest tested problem size, the scaling of the multi-core implementation seems to improve.

Returning to the GPU implementation, there are a few features which are of interest. The first is the erratic behaviour for the smallest problem size. The exact cause of this behaviour is unclear as, even with overhead, a smaller problem size is not expected to be slower than a larger one. It is, however, highly likely that this is a peculiarity of the GPU, which is poorly suited to small problem sizes. As one of the aims of PyMORESANE is adapting MORESANE to work on larger images, issues at the low-end of the problem sizes are largely irrelevant. It is, however, interesting that the GPU implementation is still the fastest, even for the smallest problem sizes. This may not be true for small problems in general, but certainly is for sufficiently high scales.

The final feature of interest is the scaling of the GPU implementation. The data points for problems of size $(1024, 1024)$ and larger fall on a straight line; the execution time scales precisely as expected. This is particularly impressive for the large problem sizes, at which the GPU implementation is approximately 28 times faster than the single-core implementation and approximately 16 times faster than the multi-core implementation.

The results for the IUWT recomposition implementations are presented in table 4.2 and figure 4.2 respectively. Many of the features of the recomposition are the same as their decomposition counterparts, including the crossover point of the single-core and multi-core implementations and the various comments on scaling.

Overall execution times are lower because the recomposition operation is more light-weight and only requires one application of the "á trous" code. This also explains why the GPU implementation is not as much faster as in the decomposition case as it gets less done per memory copy.

Additionally, the GPU implementation does not have stable scaling for the $(1024, 1024)$ problem size, and the data only reflects a straight line from $(2048, 2048)$ onwards. However, the same reasoning as for the decomposition applies - the large problem sizes are more important.

For the largest tested problem size, the GPU is approximately 14 times faster than the single-core case and approximately 6 times faster than the multi-core implementation.

Implementation	Image Dimensions (N,N)	Average Execution Time (s)
Single-core	(512,512)	0.0883 ± 0.0476
	(1024,1024)	0.3048 ± 0.0284
	(2048,2048)	1.3694 ± 0.0925
	(4096,4096)	8.4700 ± 0.0426
	(8192,8192)	32.8955 ± 0.7885
Multi-core (4 cores)	(512,512)	0.2941 ± 0.0292
	(1024,1024)	0.5609 ± 0.0018
	(2048,2048)	1.6328 ± 0.1828
	(4096,4096)	6.1068 ± 0.2200
	(8192,8192)	15.4529 ± 0.8306
GPU	(512,512)	0.0575 ± 0.1098
	(1024,1024)	0.0657 ± 0.0218
	(2048,2048)	0.1503 ± 0.0024
	(4096,4096)	0.5932 ± 0.0017
	(8192,8192)	2.3781 ± 0.0370

TABLE 4.2: Comparison of the average execution times with problem size for the various IUWT recomposition implementations.

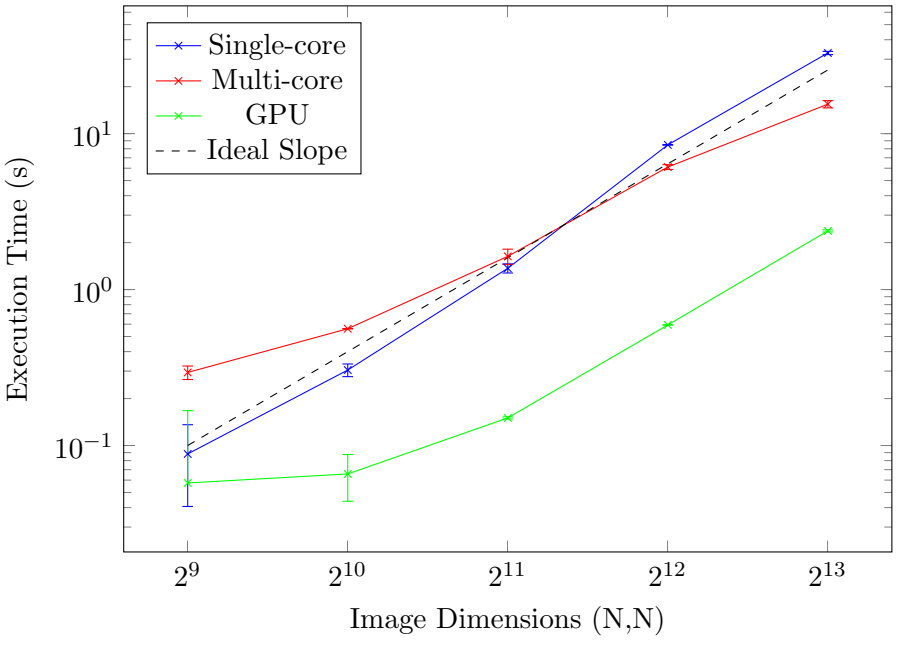

FIGURE 4.2: Plot of average execution time against data dimensions for the various IUWT recomposition implementations. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

4.2 Object Extraction

The object extraction procedure has been tested both by varying the data dimensions and by varying the number of objects in the data. This was achieved by populating arrays with increasing numbers of randomly poitioned delta functions. Unfortunately, the results presented here cannot give a complete picture of the improvement in the object extraction code as it has been altered substantially from the original. Thus, both implementations are a great deal faster than the original recursive approach.

The first set of results, which appear in table 4.3 and figure 4.3, were obtained by varying the number of objects in the data. In order to accomplish this in a realistic way, delta functions were added to an empty array. The IUWT was applied to the resulting array in order to obtain a four scale decomposition. Thus, each delta function corresponds to one object, or set of connected wavelet coefficients, in the decomposition. No restrictions were placed on the delta functions, so highly populated data may have overlapping structures. This is realistic as it emulates reality - sources may have overlapping signatures.

In order to obtain a decent estimate of the execution time for the object extraction procedure, it was run ten times per object count and the average of the individual execution times was computed. Both the CPU (single-core) and GPU implementations were profiled.

Both implementations are relatively fast. However, is worth noting the GPU implementation is faster for all object counts and has substantially better scalability. This is evident in the relatively slow increase in execution time with object count.

The second set of results, which are included in table 4.4 and figure 4.4, show the variation of execution time with data dimensions for the object extraction procedure. These results were obtained in a similar fashion to those for the IUWT decomposition and recomposition. However, the object count was fixed to ten to ensure that behaviour was consistent across data dimensions.

This particular set of results requires very little explanation. The GPU implementation is faster than the single-core implementation by approximately a factor of two. Additionally, both implementations scale closely with the problem size for data dimensions of $(1024, 1024)$ and above. The slight discrepancy for the $(512, 512)$ case is a result of the constant overhead incurred regardless of problem size, both for the CPU and GPU code.

Implementation	Number of Objects (n)	Average Execution Time (s)
CPU	5	0.7964 ± 0.1308
	10	0.9593 ± 0.0036
	15	1.2714 ± 0.0029
	20	1.4712 ± 0.0050
	25	1.7310 ± 0.0023
	30	1.9673 ± 0.0019
	35	2.1891 ± 0.0038
	40	2.3995 ± 0.0029
	45	2.6340 ± 0.0088
	50	2.8934 ± 0.0026
GPU	5	0.4585 ± 0.0028
	10	0.4566 ± 0.0039
	15	0.4764 ± 0.0043
	20	0.4842 ± 0.0051
	25	0.4902 ± 0.0015
	30	0.4982 ± 0.0016
	35	0.5251 ± 0.0025
	40	0.5349 ± 0.0029
	45	0.5323 ± 0.0032
	50	0.5530 ± 0.0027

TABLE 4.3: Comparison of the average execution times with object count for the object extraction implementations.

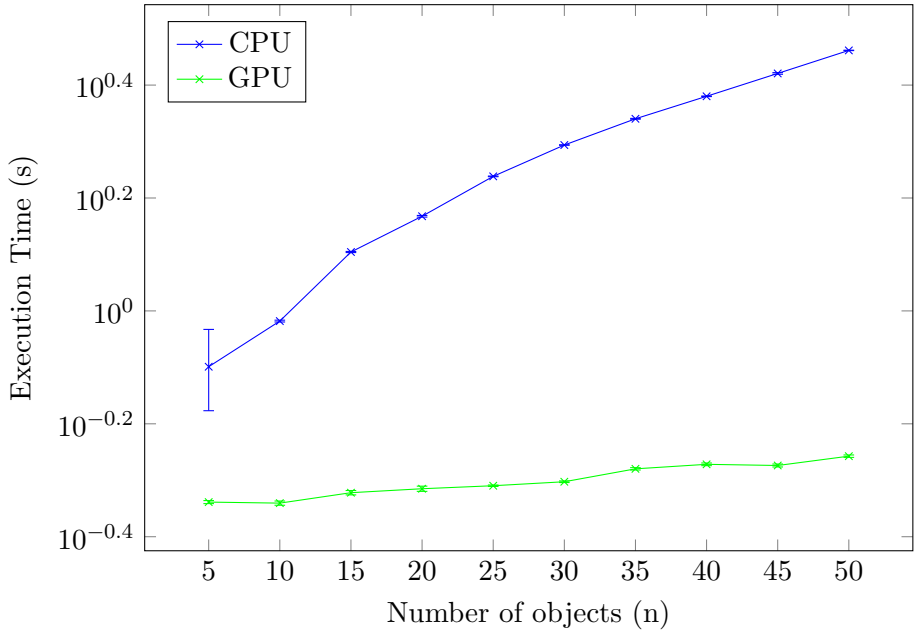

FIGURE 4.3: Plot of execution time against object count for the object extraction implementations. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

Implementation	Image Dimensions (N,N)	Average Execution Time (s)
CPU	(512,512)	0.1020 ± 0.0480
	(1024,1024)	0.2331 ± 0.0034
	(2048,2048)	0.9725 ± 0.0018
	(4096,4096)	4.4024 ± 0.0159
	(8192,8192)	18.9054 ± 0.0519
GPU	(512,512)	0.0526 ± 0.0293
	(1024,1024)	0.1188 ± 0.0046
	(2048,2048)	0.4990 ± 0.0012
	(4096,4096)	2.4511 ± 0.0013
	(8192,8192)	9.7733 ± 0.0106

TABLE 4.4: Comparison of the average execution times with data dimensions for the object extraction implementations.

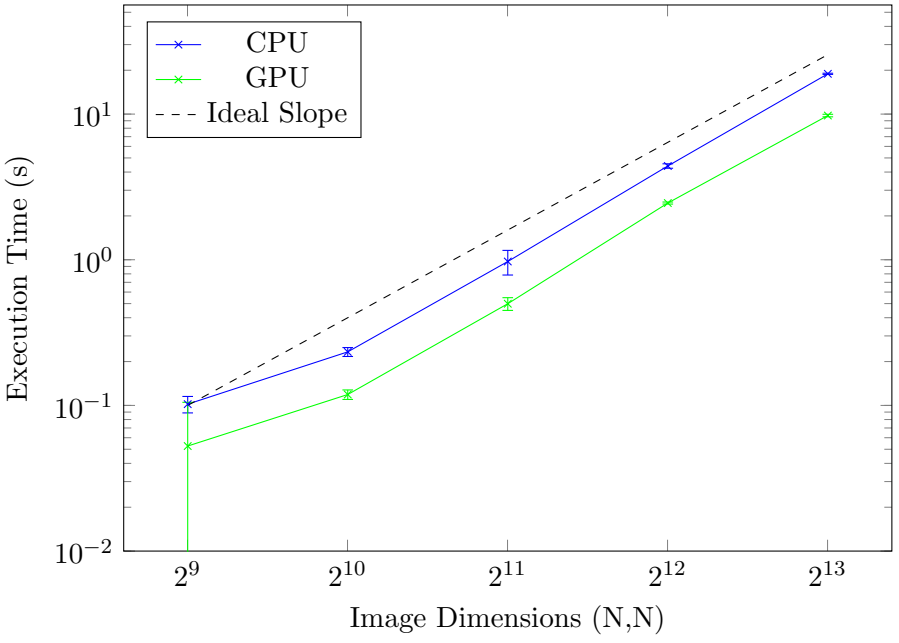

FIGURE 4.4: Plot of execution time against data dimensions for the object extraction implementations. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

4.3 Convolution and the FFT

For the sake of completeness, the results of performing the FFTs and convolution on both the CPU and the GPU are presented here. The FFT and IFFT used for the GPU implementation form part of the CUDA Scikit, and thus are not original code. However, setting up and executing the transforms is an important task, and one which accelerates PyMORESANE a great deal. The results appear in table 4.5 and figure 4.5.

The table presents the data explicitly with exact timings for each problem size. The data was obtained by timing each implementation twenty times per problem size and taking the average of the results. The input to the convolution was the PSF for the real observation used in chapter 6 and an array filled with a random distribution of point sources.

Several features are of interest in the figure. The first is the fact that the CPU implementation exceeds the GPU for the smallest data dimensions tested. This, however, is not particularly surprising, as convolution requires several memory copies on top of the actual computation. As such, for relatively small problem sizes, the expense of moving the data around far outweighs the gains made in the computation of the multiple FFTs and IFFTs. However, on larger problem sizes, the GPU comes into its own and is a full order of magnitude faster.

In terms of scaling, the behaviour of both implementations is somewhat erratic for small problem sizes. For the GPU, this is likely for the same reasons as discussed above; memory copies. They are less efficient for small amounts of data. The CPU is less erratic but the cause is also less certain. However, the variation is not sufficient to warrant concern as only the data for $(1024, 1024)$ seems inconsistent and may be a product of randomness in the execution. This is supported by the larger error bar.

The larger problem sizes reveal the more stable scaling properties of both implementations. For the CPU-based implementation, for problems of size $(2048, 2048)$ and greater, the scaling becomes one-to-one, as seen by the slope of the line from that point onwards. The GPU implementation does not present ideal scaling for small problem sizes. However, it does appear to exhibit the desireable one-to-one property for the largest problem sizes. This makes sense, as the operation of both implementations is the same. The GPU implementation is expected to continue scaling in the same fashion for even larger problems and remain at a least a full order of magnitude faster than CPU-based convolution.

Implementation	Image Dimensions (N,N)	Average Execution Time (s)
CPU	(512,512)	0.0625 ± 0.0132
	(1024,1024)	0.3444 ± 0.0164
	(2048,2048)	0.8109 ± 0.1870
	(4096,4096)	3.0488 ± 0.1543
	(8192,8192)	12.4459 ± 0.1785
GPU	(512,512)	0.2120 ± 0.0524
	(1024,1024)	0.2285 ± 0.0089
	(2048,2048)	0.1510 ± 0.0500
	(4096,4096)	0.3232 ± 0.0396
	(8192,8192)	1.0875 ± 0.1951

TABLE 4.5: Comparison of the average execution times with data dimensions for the convolution implementations.

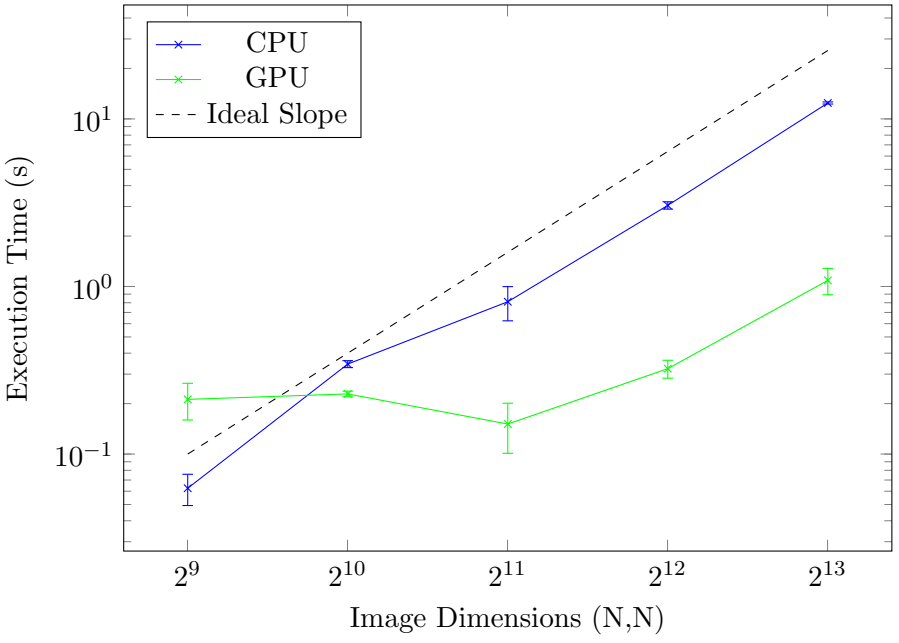

FIGURE 4.5: Plot of execution time against data dimensions for the convolution implementations. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

4.4 PyMORESANE

The overall acceleration of MORESANE, in the form of PyMORESANE, was the primary goal of this project. The following section presents the results pertinent to determining the degree of success in this regard. The subsequent chapters will present examples of running PyMORESANE on both real and synthetic data.

The results of this section were obtained by running PyMORESANE in both CPU mode (no multi-processing) and GPU mode. Both approaches were applied to the same data, and the region of interest (deconvolution region) was varied. The approaches were timed once for each problem size. The results appear in table 4.6 and figure 4.6. The data used for this test was the same as the real data used in chapter 6.

The table gives the numeric total execution times for each run of the two approaches. With the exception of the smallest problem size, the GPU-accelerated implementation is vastly superior to the CPU implementation. Even for the second smallest problems size, the GPU is around three times faster. For the largest problem size tested, the GPU is more than an order of magnitude faster. This is a striking example of the benefits offered by GPU-based implementations.

Figure 4.6 serves to emphasise the degree of improvement in the GPU-accelerated implementation over the basic CPU approach. It clearly shows that, while the implementations have virtually the same execution speed for data of dimensions $(512, 512)$, thereafter they are divergent - the GPU implementation is more than merely faster, it also scales considerably better with problem size. Thus, the factor of ten speed up seen in the (4096.4096) case might increase even further as the problem size is increased. However, hardware does impose limitations. In particular, the amount of RAM on an individual GPU places a limit on the maximum problem size. Additionally, CUDA currently does not allow allocation of arrays which are individually larger than two gigabytes in size.

There is additional evidence of the previously mentioned improvement concealed in the slope of the two curves. For the CPU case, each execution time is substantially more than quadruple than that of previous problem size. Thus, the one-to-one scaling previously seen in the individual sections seems absent here. This is unfortunate as it means that the CPU implementation rapidly becomes prohibitively slow for large problems. This loss of perfect scaling is due to overhead in the main body of the PyMORESANE code, which varies greatly with problem size. In particular, for large problems, operations such as sorting or multiplying large arrays become very expensive.

However, the GPU implementation exhibits completely different behaviour, and the data for each problem size reveals that the GPU implementation scales at substantially better than one-to-one with problem size. This is remarkable, and ensures that the GPU implementation remains feasible on far larger problems than the CPU. The overhead costs are mitigated for the GPU implementation, as a large number of the expensive multiplications are performed on the GPU, which almost removes their contribution.

In order to provide a clearer picture of how the improvement of the individual pieces of the implementation contributed to the overall acceleration of PyMORESANE, the tests were also performed using a profiler. The profiler provided a detailed breakdown of the time spent in each function inside the implementation. For the purposes of visualising the results, the bar graphs which appear in figure 4.7 and 4.8 were created.

Each bar graph shows the overall execution time of PyMORESANE, in addition to the individual contributions of the IUWT decomposition and recomposition (combined), convolution, and the remaining operations. These remaining operations include the object extraction procedure. The times are plotted on a logarithmic axis which unfortunately hides the fact that the individual timings add up to the total. However, it is necessary to allow comparison between problems of such vastly differing sizes.

The first of the bar graphs, figure 4.7, shows the breakdown for the CPU-based implementation. The plot reveals quite clearly that the majority of the CPU-based implementation is spent in computing the the IUWT decomposition and recomposition. The convolution contributes substantially less than that for all problem sizes. The remaining functionality of PyMORESANE does not even contribute as much as the convolutions.

One feature of interest is the way in which the problem scales. Each group of bars is nearly a perfectly scaled version of the others. The only exception is for the smallest problem size, for which the IUWT decompositions and recompositions are faster. This is simply because the problem is sufficiently tiny to avoid additional memory access overhead.

The second bar graph, figure 4.8, corresponds to the GPU-accelerated implementation. It has substantially more interesting features. The first and most readily apparent feature is the massive drop in the execution time of the IUWT decompositions and recompositions. Its execution time is roughly an order of magnitude below that of the total, revealing how much less of an impact it has in the GPU case.

The second noticeable feature is that there is that the contributions of the individual functions vary from problem size to problem size. In particular, the convolution operation grows more and more efficient whilst the IUWT operations are slightly slower on the larger problems. However, for the largest problems it is actually the remaining

functionality, that which doesn't belong to either the IUWT or the convolutions, that takes the most time. This suggests that further optimisation may be possible.

Further investigation of the profiler output reveals that certain operations which seemed minor when determining bottlenecks in the CPU implementation are actually quite problematic once the GPU-acceleration has mitigated the effects of the IUWT and convolution. In particular, computation of the median becomes very expensive when the data dimensions N are large as the data then contains N^2 elements to sort.

Implementation	Image Dimensions (N,N)	Execution Time (h:m:s)
CPU	(512,512)	00:01:57
	(1024,1024)	00:29:34
	(2048,2048)	02:43:08
	(4096,4096)	16:12:19
GPU	(512,512)	00:02:08
	(1024,1024)	00:10:02
	(2048,2048)	00:28:05
	(4096,4096)	01:17:06

TABLE 4.6: Comparison of the execution times with data dimensions for PyMORESANE in CPU and GPU mode.

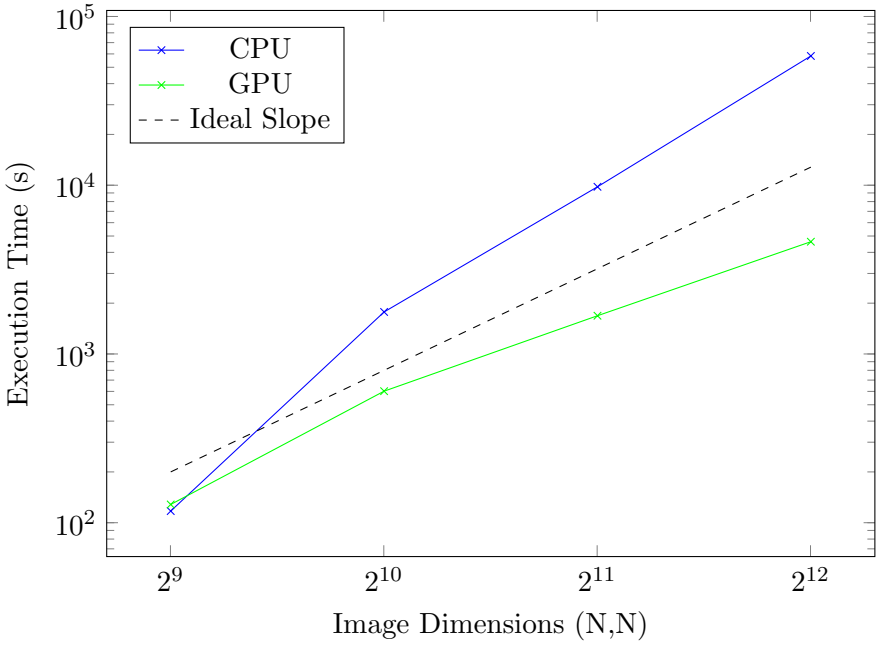

FIGURE 4.6: Plot showing the execution times of PyMORESANE in CPU and GPU mode with varying data dimensions. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

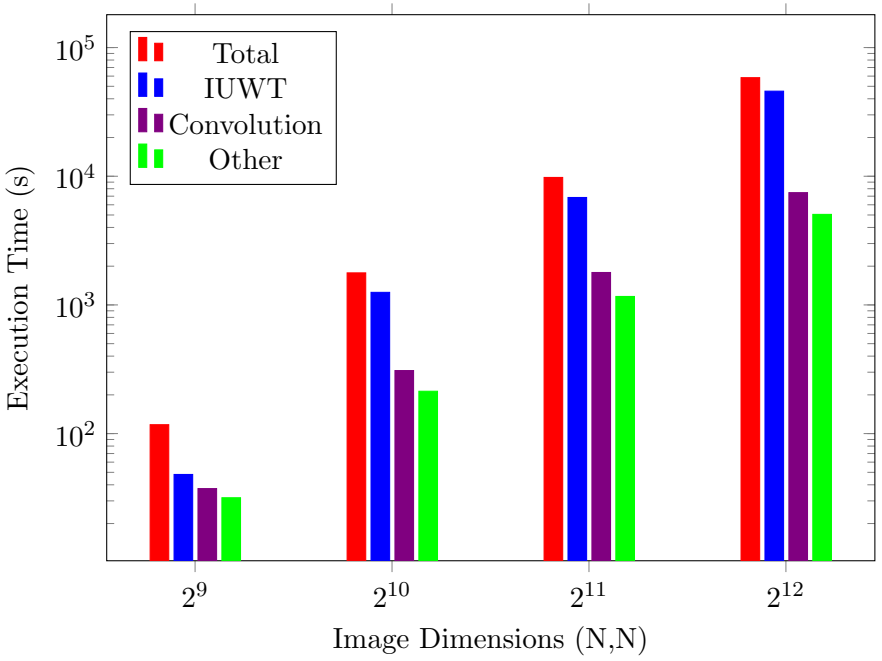

FIGURE 4.7: Histogram showing the execution times for the component functions of PyMORESANE in CPU mode. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

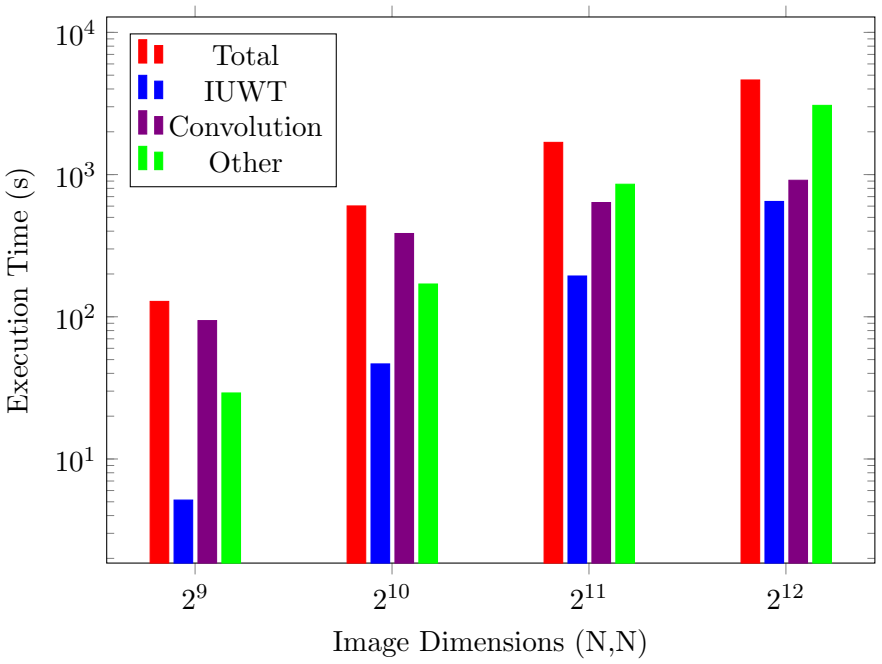

FIGURE 4.8: Histogram showing the execution times for the component functions of PyMORESANE in GPU mode. The average execution times have been plotted on a logarithmic scale to allow comparison across orders of magnitude.

Chapter 5

Results - Synthetic Data

This chapter presents and analyses the results of running PyMORESANE on synthetic data. A far more exhaustive comparison appears in [2], but this serves a check of PyMORESANE's functionality. The following results are merely a representative example showing that PyMORESANE produces comparable output to that obtained in [2]. That is, for the same synthetic field, similar reconstruction quality is obtained.

The test data in question is a simulation of a field containing both a faint, diffuse radio halo and bright, compact sources. The input model appears in figure 5.1. The simulation was performed for the JVLA (Jansky Very Large Array) in A configuration at a frequency of 1.4GHz. The integration time was chosen as 60 seconds with a total observation time of 8 hours. The details of the original field simulation appear in [1].

Imaging was performed using Briggs weighting with a robustness of -2 (approximately uniform weighting) and a cell size of 0.4 arcseconds. The resulting dirty image and its associated PSF appear in figures 5.2 and 5.3 respectively. The size of the dirty image is (2048,2048) pixels.

These images were used as the input to PyMORESANE. For comparison, both Cotton-Schwab CLEAN (CS-CLEAN) and multi-scale CLEAN (MS-CLEAN) were run on the same data. The implementations used for comparison were those of lwimager. Both of these algorithms make use of the visibilities to avoid the errors introduced by edge effects in image-space only deconvolution.

Objective comparison of the various algorithms is challenging as both CS-CLEAN and MS-CLEAN do not evaluate the residual or noise in the same way as MORESANE. However, for the simulation, the noise level was known. Thus, the limiting flux was set to be four times the standard deviation of the noise across all the deconvolution algorithms.

The number of iterations for CS-CLEAN and MS-CLEAN were made arbitrarily high to ensure that the threshold was reached.

As MS-CLEAN restricts itself to modelling only the inner quadrant of the dirty image, PyMORESANE was also confined to the same region to ensure that the problem was of the same size and complexity. Additionally, all metrics were calculated for the same region. This advantages PyMORESANE slightly, as it prevents any remaining edge effects from corrupting the results.

The scales required by MS-CLEAN were chosen as [0,1,2,4,8,16,32,64,128]. There was no particular motivation for this choice other than a sort of parallel between these scales and those used by PyMORESANE.

The results of running the various deconvolution algorithms are presented in figures 5.4, 5.5 and 5.6. Each set of figures corresponds to one of the interesting outputs of the deconvolution for each algorithm.

Importantly, these results are untuned; no parameters were chosen to advantage or disadvantage a particular algorithm. The defaults of each algorithm were made consistent and only the limiting flux was adjusted.

The model images are immediately interesting, simply because they differ so greatly from algorithm to algorithm. The CS-CLEAN model is as expected, with the diffuse emission being poorly approximated by a collection of delta functions. This means the model provides very little information about the extended emission, and convolution with a restoring beam is necessary to get an idea of the source structure. This is not the case for either MS-CLEAN or MORESANE, both of which recover substantially more realistic models.

The MS-CLEAN model is a better representation of the truth and does resemble the input sky. However, as is obvious from the figure, there is a substantial number of artefacts. Some of these are spurious detections whilst others, in particular the negative regions, reflect the absence of a positivity constraint in MS-CLEAN. Thus, negative components may be added to the model.

Another, less obvious, feature is the way in which the morphology of the model is affected by the manner in which MS-CLEAN operates. This is evident in the distinct roundness of the features in the model. This is the result of using extended components which are inherently round.

PyMORESANE produces a model superior to both CS-CLEAN and MS-CLEAN. There is sufficient proof of this in the model image itself. The presence of a positivity constraint ensures that the model is smooth. Whilst there are are a few spurious detections, they

are minor and strictly positive. Additionally, the morphology of the source is very well modelled. The distinct roundness evident in MS-CLEAN is absent and the feature on the left of the figure is much closer to that of the original image.

The residual images give further information about the performance of the algorithms. The failings of CS-CLEAN are readily apparent - a large amount of the diffuse emission is left in the residual. This is due to the fact the CS-CLEAN is implicitly restricted to using only a delta function basis, which is not well-adapted to the recovery of extended emission. This well-known failing of CLEAN, and one of the driving forces behind the search for better deconvolution techniques.

In stark contrast to CS-CLEAN, MS-CLEAN does a good job of recovering the diffuse emission. In fact, the residual is distinctly noise-like. The only problem is the presence of morphologically identifiable features at the level of the noise. These correspond to the brightest compact sources in the field. This problem is likely the result of the slightly arbitrary weighting scheme applied when identifying the scale of the maximum pixel. In this case, the scale of the compact sources seems to have been overestimated at some point. This is supported by the bowls formed around the structures left in the residual; oversized components are deconvolved removing both a portion of the actual source and the region around it.

PyMORESANE succeeds where MS-CLEAN fails and its residual appears to be completely noise-like. There is no obvious structure in the residual and only very close inspection may reveal the location of the compact sources, but their flux is at the noise level. This is quite an achievement, and is indicative of MORESANE's success as a deconvolution algorithm.

The restored images are presented here although they do not provide as clear an indication of the performance of the algorithms. CS-CLEAN clearly retains PSF structure which is visible in the structured dark regions on the upper right and lower left of the image. The restored maps for MS-CLEAN and PyMORESANE are basically visually indistinguishable, due to the fact that the improvements to the model concern features close to the noise level. As the noise is added to the restored image, many of the improvements are masked. Quantification of the restored images' dynamic ranges reveals the slightly superior performance of PyMORESANE. These numerical results appear in table 5.1 along with the time taken for each algorithm to run and the RMS (root mean square) of the residuals.

PyMORESANE was used in GPU mode - naturally, as was shown in chapter 4, non-GPU operation would have been prohibitively slow. To this end, PyMORESANE has a comparable execution time to its strongest competitor in this setting. Whilst it is a factor

of two slower than MS-CLEAN, that is not so large are to prevent its consideration as a mainstream deconvolution algorithm. Of course, CS-CLEAN is far faster than either MS-CLEAN or PyMORESANE, but offers correspondingly worse results.

The dynamic range - ratio of the brightest pixel in the restored image to the standard deviation of the residual - is indicative of how successful a deconvolution procedure is. However, it is not a perfect metric as it can be biased. Regardless, in this instance it does agree with the visual results.

The RMS of the residual is a measure of how much signal is left in the residual. Again, this is not a perfect metric, and it can be misleading. What is interesting in this instance is that all the algorithms produce a residual with the same RMS, even though they are visually different. This emphasises the difficulty in quantifying the success of a deconvolution algorithm.

Algorithm	Execution Time (h:m:s)	Dynamic Range	Residual RMS (Jy)
CS-CLEAN	00:00:27	186.41	0.000002
MS-CLEAN	00:07:25	191.64	0.000002
PyMORESANE	00:15:02	212.12	0.000002

TABLE 5.1: Comparison of the algorithm execution times, dynamic ranges and residual RMS as applied on synthetic data. PyMORESANE was run using GPU functionality.

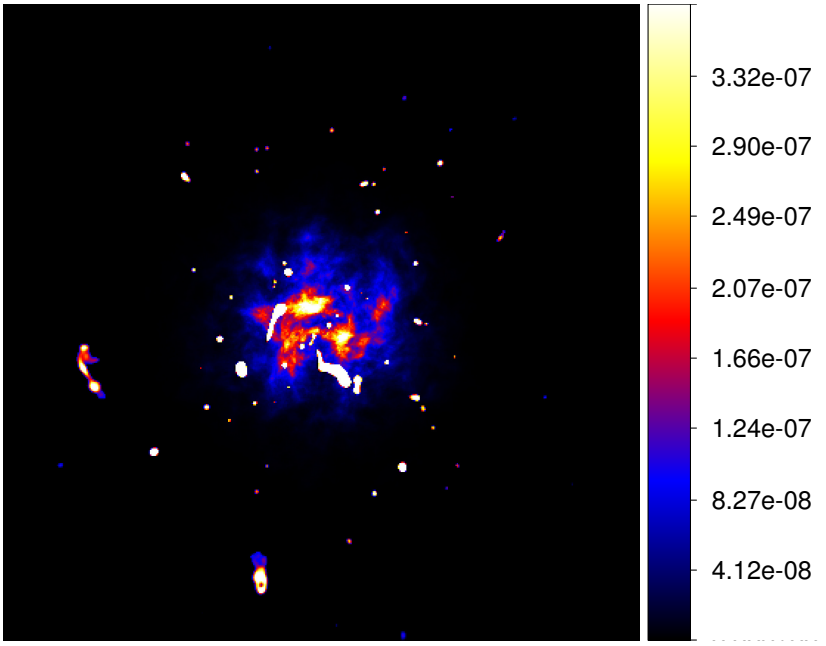

FIGURE 5.1: The sky model on which the simulation was based.

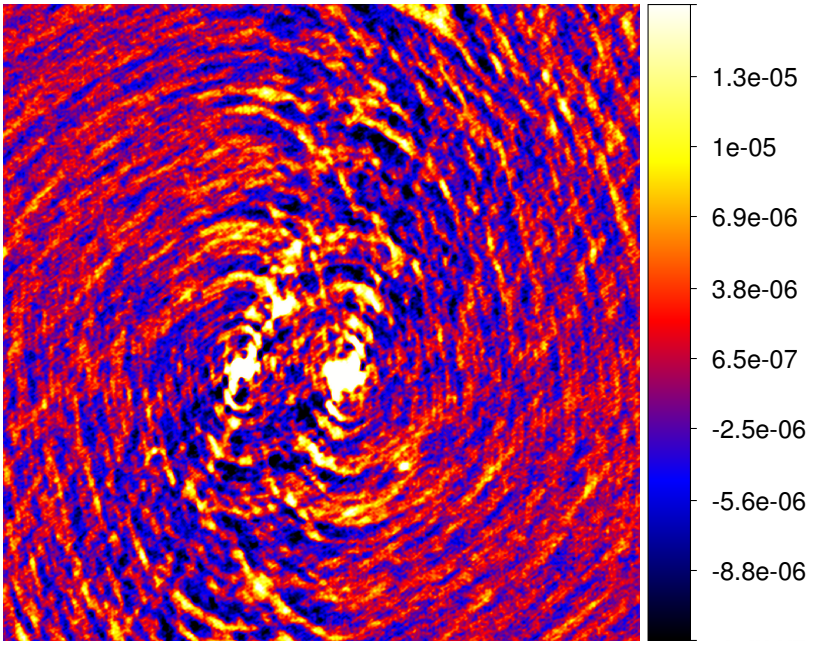

FIGURE 5.2: The dirty image derived from the sythetic data of a radio halo. No diffuse emission is visible.

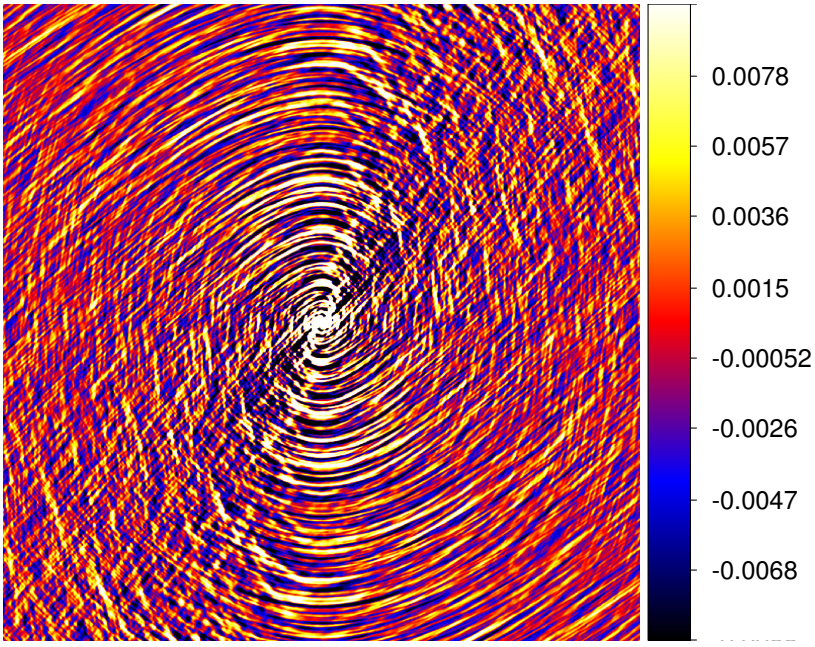

FIGURE 5.3: The PSF corresponding the sythetic data.

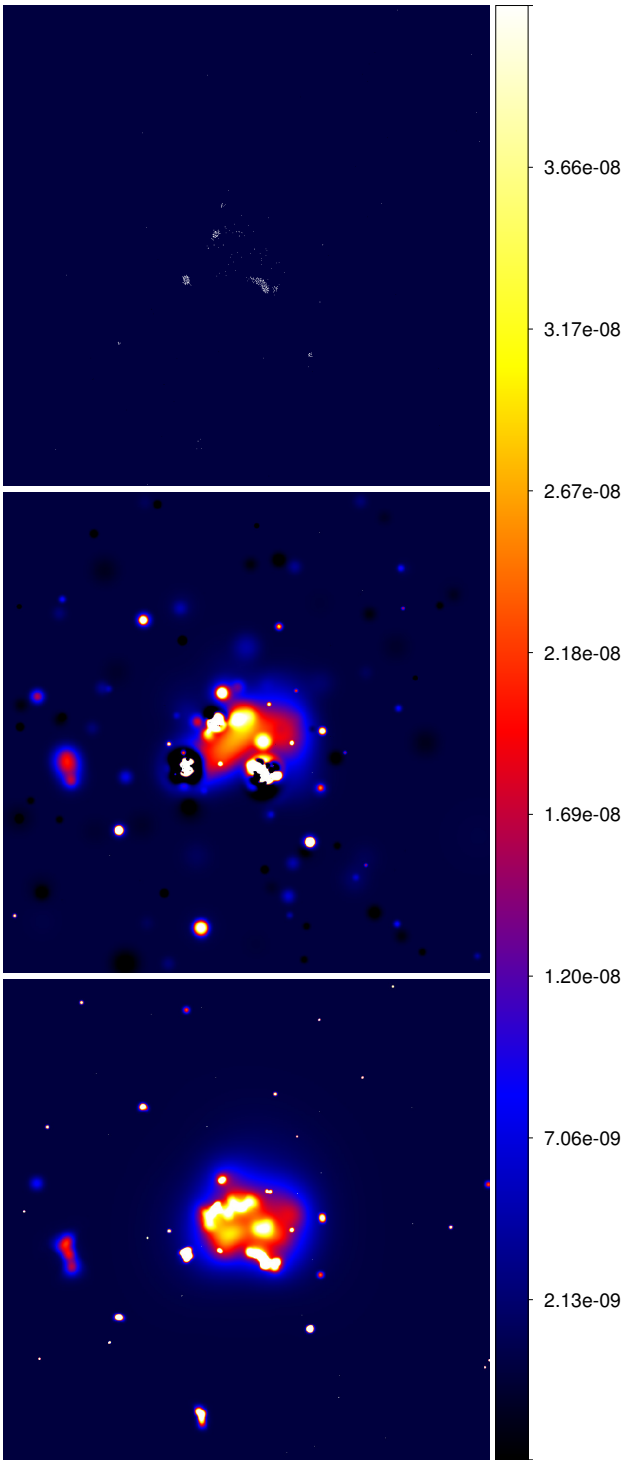

FIGURE 5.4: The models produced by CS-CLEAN, MS-CLEAN and PyMORESANE respectively.

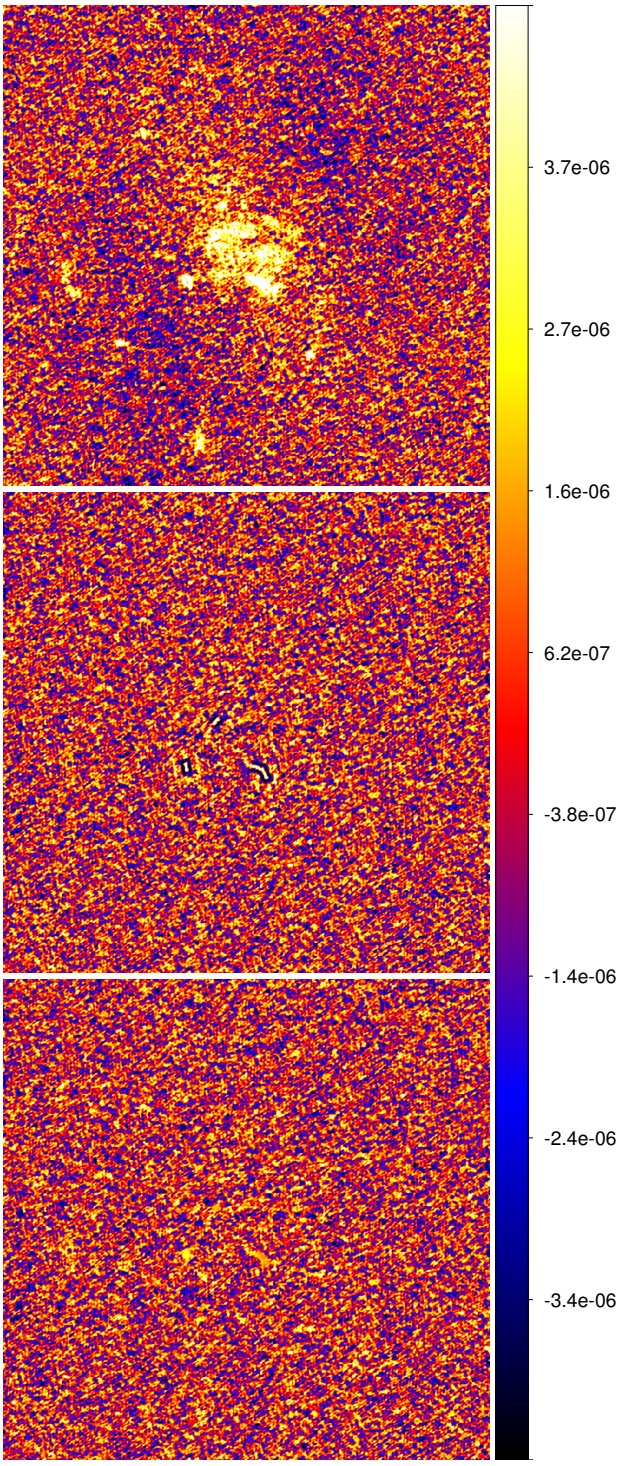

FIGURE 5.5: The residuals produced by CS-CLEAN, MS-CLEAN and PyMORESANE respectively.

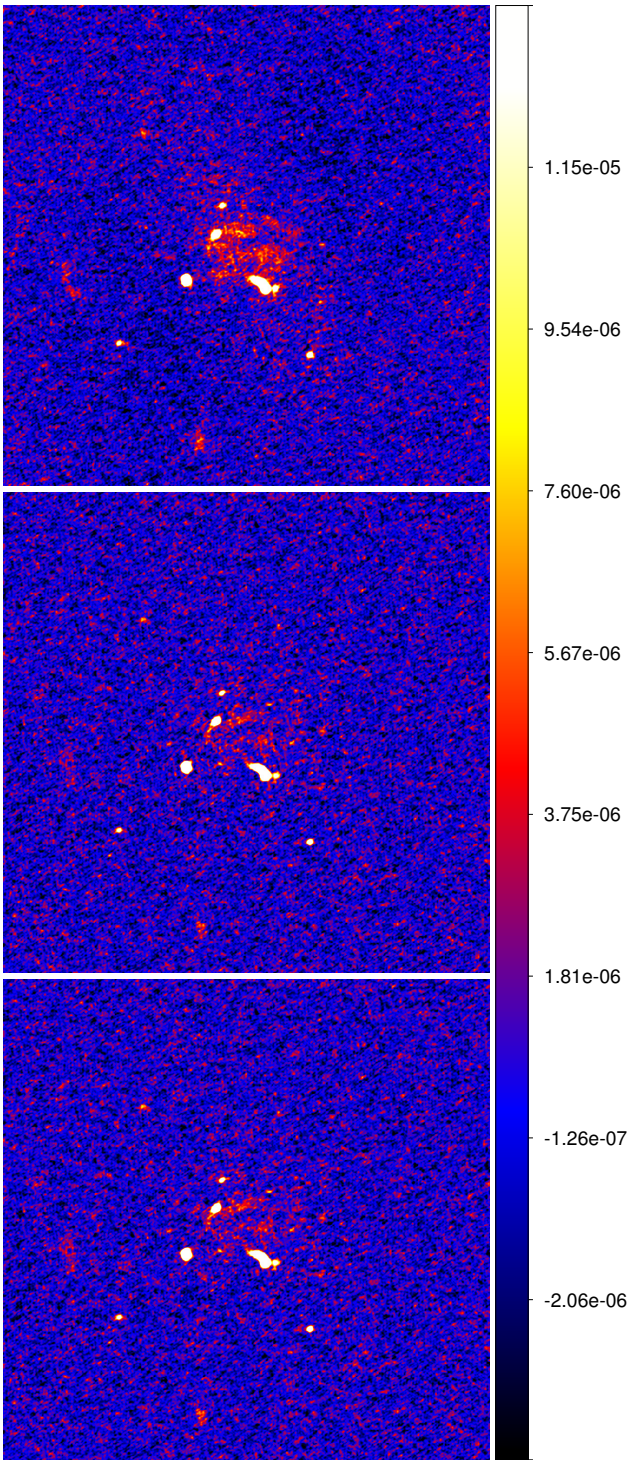

FIGURE 5.6: The restored maps produced by CS-CLEAN, MS-CLEAN and PyMORE-SANE respectively.

Chapter 6

Results - Real Data

The following chapter presents and analyses the results of running PyMORESANE on real data. The data in question is an ATCA (Australia Telescope Compact Array) observation of a field containing diffuse radio emission at frequencies between 1.96Ghz and 2.20Ghz. The integration time was 5.27 seconds and the total observation time was 04:36:38. The dirty image of the field and its associated PSF appear in figures 6.1 and 6.2 respectively. The size of the dirty image is (4096,4096) pixels. The data was provided by C. Trigilio (INAF-OACT, Catania, Italy) and F. Cavallaro (UNICT/CASS, Catania, Italy).

As in the case for synthetic data, both CS-CLEAN and MS-CLEAN were run for comparison with PyMORESANE. However, unlike in the synthetic data case, the noise level in the real image was unknown and could only be estimated. This is beyond the basic functionality of either MS-CLEAN or CS-CLEAN. In order to circumvent this problem, PyMORESANE was run using only the default parameters. The resulting residual was used to estimate a limiting flux for the other algorithms.

Unfortunately, whilst the above approach did yield comparable results in the case of CS-CLEAN, a large flaw in MS-CLEAN became apparent. That is, the slightly arbitrary way in which the various scales are weighted in MS-CLEAN causes erratic behaviour for this particular field, where the diffuse emission is of comparable intensity to that of the point sources. The only way to produce results in this instance was to tune the MS-CLEAN parameters. Ultimately, this resulted in upping the limiting flux threshold a great deal to around 0.04Jy as opposed to the 0.002Jy used for CS-CLEAN. This massive discrepancy works in MORESANE's favour, as it reveals a set of circumstances in which MS-CLEAN does not work but MORESANE does.

To clarify the above, both PyMORESANE and CS-CLEAN were run using their default parameters, bar the threshold value in the CS-CLEAN case. Only MS-CLEAN was altered so that its results were at the very least comparable to those of the other algorithms.

The results of the deconvolution are presented in figures 6.3, 6.5 and 6.6. Each set of figures corresponds to one of the interesting outputs of the deconvolution for each algorithm.

The model images follow the same trend as in the synthetic data case. Naturally, CS-CLEAN's model is non-physical in appearance due to the previously mentioned restriction to a delta function basis. MS-CLEAN's model is limited by the depth to which it was allowed to clean, though the diffuse emission is reasonably well modelled. It is PyMORESANE for which the results are truly impressive. Both the diffuse emission and the bright compact source embedded in it are well modelled. Additionally, a classical looking radio galaxy is beautifully modelled in the lower left quadrant of the image. A zoomed version of this source appears in figure 6.4. It is worth noting that the PyMORESANE model does not appear to be corrupted by false detections at this flux threshold.

The residual images are far more interesting in this case due to the major visual differences between them. CS-CLEAN does remarkably well given its simplicity, although it clearly struggles so close to the noise. This is evident in the negative values which appear within signature of the diffuse emission. This is roughly the best that CS-CLEAN can do, as even at this flux threshold, various artefacts began corrupting the residual.

MS-CLEAN produces a rather poor residual. Once again, this is due to the high flux threshold. Even at this level, it is clear that the compact source couched in the diffuse emission is being poorly approximated, and a negative bowl has formed around it. Allowing the deconvolution to proceed to the same flux threshold as CS-CLEAN yielded a map dominated by this negative hole. The majority of the field has not been deconvolved.

PyMORESANE has a very structured residual, and the diffuse emission is still relatively apparent. However, this emission is all at or below the limiting flux value. Allowing the algorithm to go deeper into the noise would naturally recover more of this emission, but incur additional false detections. Regardless, the residual is still fairly good, as it has not been marred by the overestimation of sources. Close inspection reveals a poorly deconvolved source in the top left of the image. This is likely due to calibration effects - the data used was not fully calibrated. Additionally, some direction dependent effects

are still present. However, it may be possible to improve PyMORESANE to be more capable of handling of such problems.

Whilst the various problems faced by the algorithms have already been discussed, in the restored images it is clear that MORESANE produces a better result. The deconvolved emission is quite clear and, unlike CS-CLEAN, there are fewer visible discrepancies around the diffuse emission.

The metrics and execution times of the algorithms appear in table 6.1. Here the metrics are a little confusing, as they suggest that CS-CLEAN in outperforming PyMORESANE in terms of accuracy, even though the results are visually better for PyMORESANE. However, as previously discussed, these metrics are not flawless, and sometimes overestimation can introduce bias. Regardless, PyMORESANE and CS-CLEAN are sufficiently similar to claim that they both do a good job, even though PyMORESANE seems to model the diffuse emission more accurately. Unfortunately, there is no simple metric for morphological recovery.

In terms of execution speed, CS-CLEAN is the obvious leader. MS-CLEAN appears to outperform PyMORESANE in this regard, but the value is misleading; the higher flux threshold used for MS-CLEAN also means it does not require as many iterations. Whilst attempting to produce adequate results for MS-CLEAN, a test with a threshold of 0.002Jy took longer to complete using MS-CLEAN than it did with PyMORESANE in GPU mode.

Algorithm	Execution Time (h:m:s)	Dynamic Range	Residual RMS (Jy)
CS-CLEAN	00:01:48	165.43	0.000420
MS-CLEAN	00:11:50	110.95	0.000621
PyMORESANE	00:25:43	165.14	0.000449

TABLE 6.1: Comparison of the algorithm execution times, dynamic ranges and residual RMS as applied on real data. PyMORESANE was run using GPU functionality.

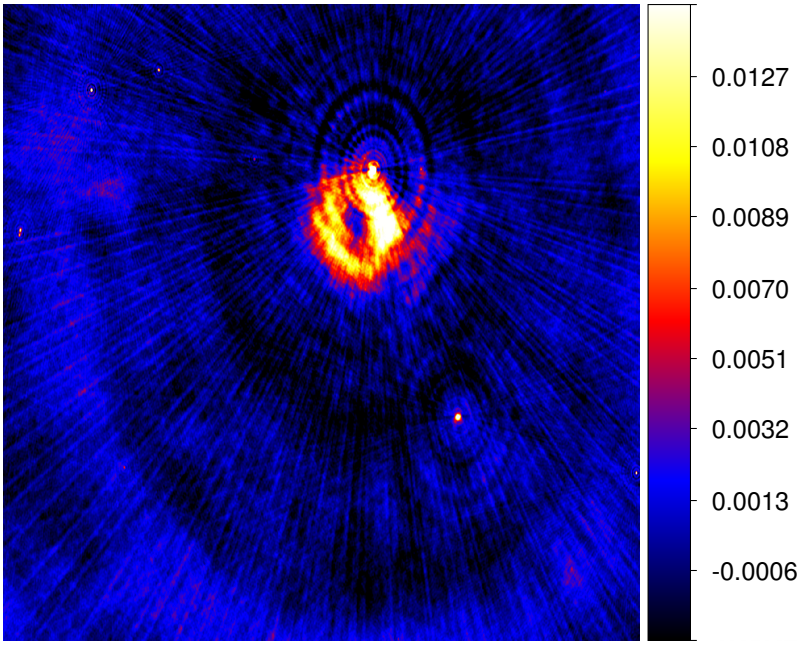

FIGURE 6.1: The dirty image derived from the observational data. Imaging was performed using multi-frequency synthesis and uniform weighting.

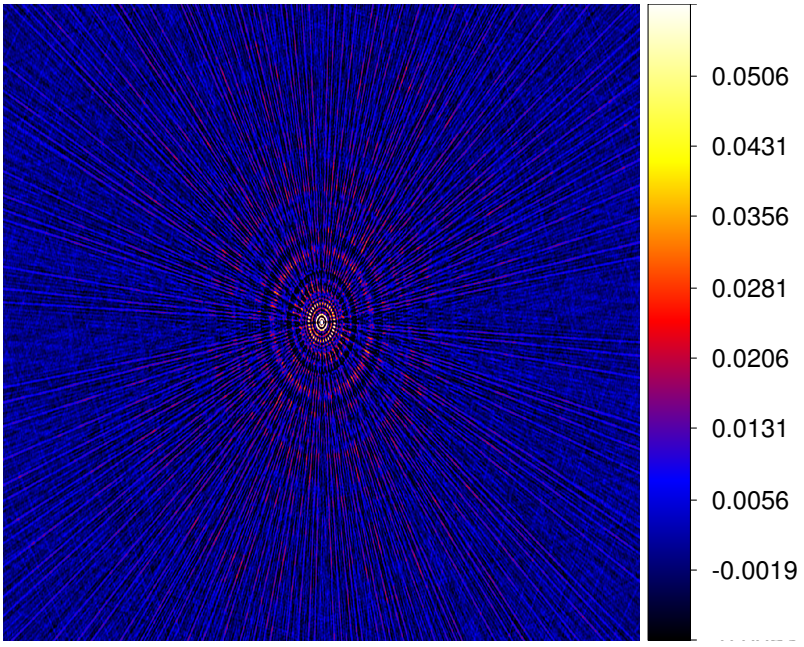

FIGURE 6.2: The PSF assosciated with the above dirty image.

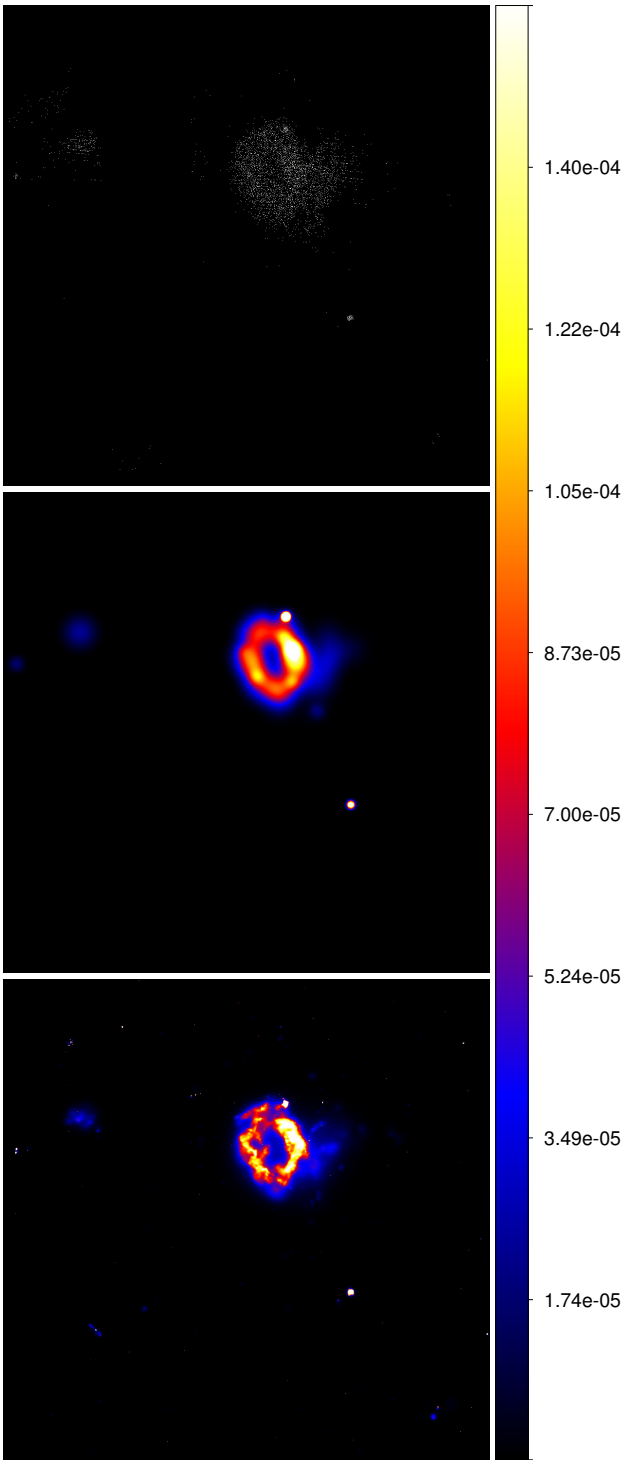

FIGURE 6.3: The models produced by CS-CLEAN, MS-CLEAN and PyMORESANE respectively.

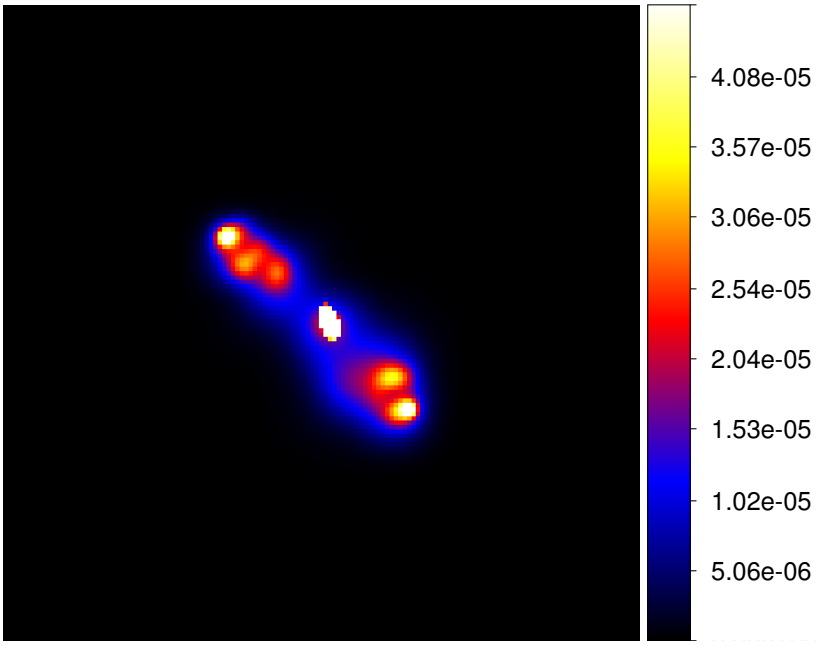

FIGURE 6.4: A zoomed image of the well-recovered radio galaxy.

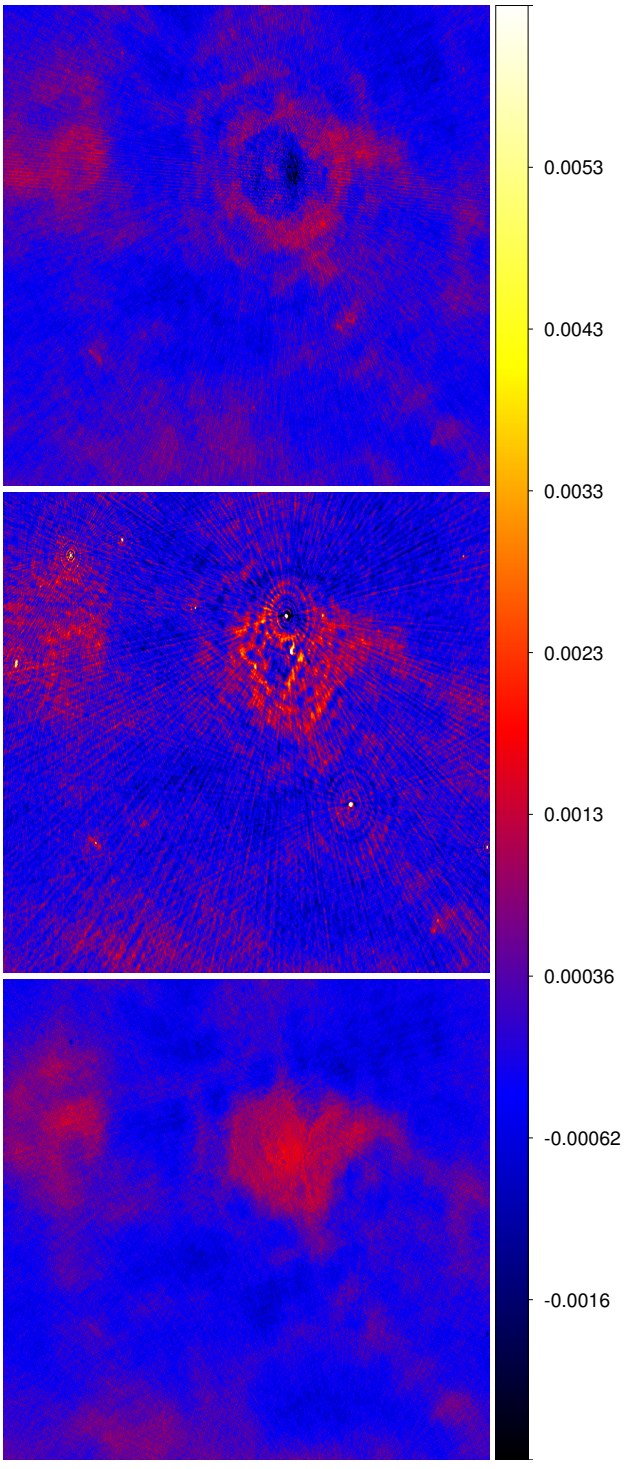

FIGURE 6.5: The residuals produced by CS-CLEAN, MS-CLEAN and PyMORESANE respectively.

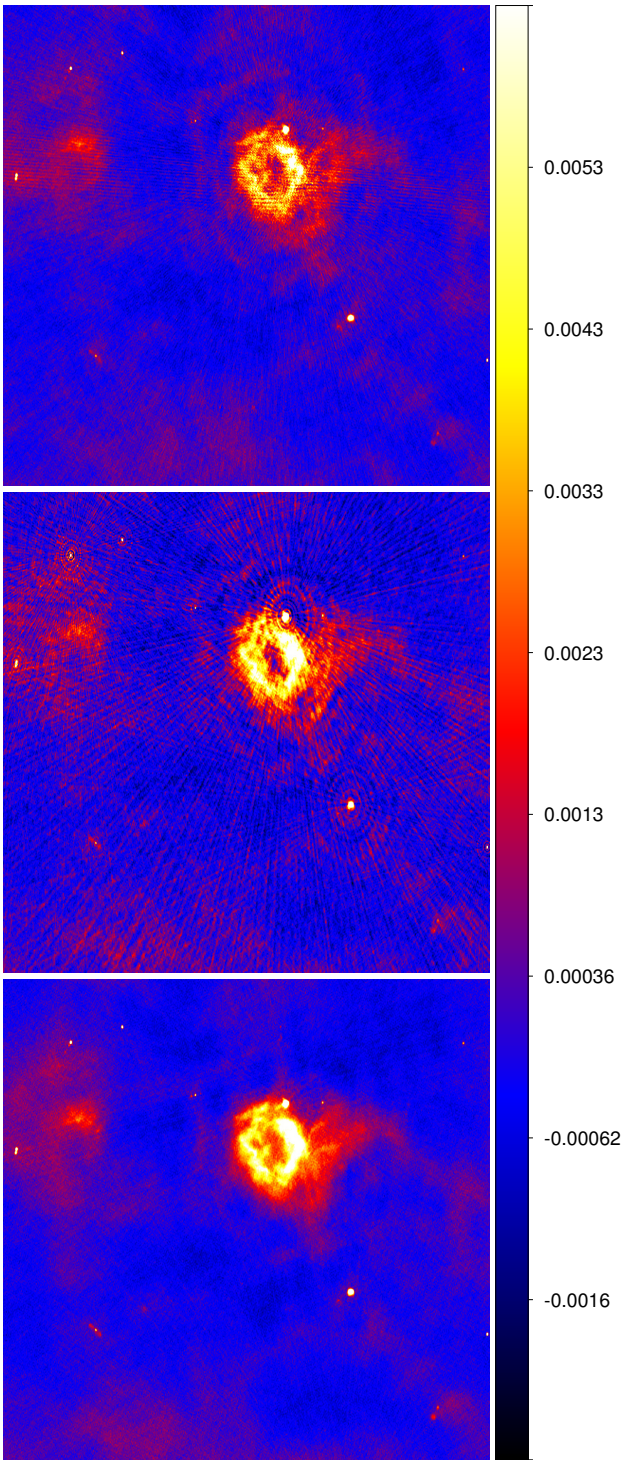

FIGURE 6.6: The restored maps produced by CS-CLEAN, MS-CLEAN and PyMORE-SANE respectively.

Chapter 7

Conclusion

MORESANE, as discussed in chapter 2, has been shown to be a novel, powerful deconvolution technique. However, it is a very computationally expensive algorithm. The newest and next-generation interferometers will produce larger, more detailed images than ever before. This requires any new deconvolution approach to be not only accurate and sensitive to complex morphologies but also computationally feasible.

Fortunately, with the advent of GPGPU programming, massive parallel environments mean that it is possible to perform more complex calculations in far less time. Thus, algorithms which can be adapted to the hardware environment are far more likely to become successors to CLEAN and its ilk.

As with any tool, an implementation's success is based on its use. Thus, it is important that new algorithms do not require proprietary software and are freely available. The original implementation of MORESANE was hampered by its use of proprietary software which consequently motivated the development of PyMORESANE.

PyMORESANE, the implementation details of which appear in chapter 3, is a Pythonic implementation of MORESANE. By making use of freely available Python packages to exploit the GPU, it has been accelerated to the point that its execution time is comparable with the current-generation multi-scale algorithm, MS-CLEAN. The results supporting this appear in chapter 4.

Additionally, PyMORESANE has been shown in chapter 5 and 6 to be successful not only in reproducing MORESANE's results on synthetic data but also in working on real observational data. This is a triumph for both MORESANE and PyMORESANE as it verifies that the algorithm still functions in the absence of the ideal conditions associated with synthetic data. This is important as many of MORESANE's competitors are untested on real data.

A further advantage of PyMORESANE is that it functions well without having its parameters fine tuned to the data in question. This means it can be run without user input; allowing it to be used in automated applications such as data reduction.

The successes of PyMORESANE in recovering diffuse emission in large images will allow for more in-depth study of related astrophysical phenomena than is currently possible with existing techniques such as CLEAN.

Whilst it is true that there are further improvements and optimisations to be made, PyMORESANE is already a fully functional piece of software which can be used by radio astronomers with relatively little difficulty. Such improvements would need to be directed equally at the mathematics of the original algorithm and at the code of PyMORESANE in order to speed up the algorithm further.

There are many directions in which both the algorithm, and consequently PyMORESANE, may develop. These directions include the use of a major loop which converts the model image to visibilities and subtracts it from the residual visibilities in the uv-plane, much like CS-CLEAN (https://github.com/ratt-ru/PyMORESANE/tree/vissub - currently in development). Other possible developments include an extension to multi-frequency image cubes and deconvolution of images with spatially varying PSFs.

In summary, PyMORESANE is a successfully implemented and accelerated version of a next-generation deconvolution algorithm. It is freely available and can easily be added to the arsenal of deconvolution tools used by radio astronomers. It is well suited to the deconvolution of extended emission, but is also just as capable of recovering point sources. This makes it ideal for observations of real, complex fields with unknown morphology and ensures that it will find use in the hands of the astronomy community at large.

Appendix A

PyMORESANE: Instructions

The following appendix serves as a very brief introduction to the use of PyMORESANE. Whilst it may seem that there are many parameters, most of them are related to optimisation, and very few of them will need to be changed for normal usage. Note, the PyMORESANE help command will present all these options in the terminal.

Following installation, and assuming that PyMORESANE has been aliased as runsane, the standard input is as follows:

runsane dirty psf outputname

These are the non-optional parameters which are specified by position - order matters. The first positional argument, **dirty**, should be the name and address of the dirty .fits image in question. The second, **psf**, is the same as dirty but for the PSF assosciated with the dirty image. The final positional argument, **outputname**, is a string on which the names of the output will be based.

There are many optional arguments, all of which can be exposed with the **runsane --help** command. The most important optional parameters are as follows, and take the format of **--argumentname ARGUMENTVALUE**:

Argument	Long Argument	Functionality
-ep	--enforcepositivity	Forces output model to be positive. Yields a smoother, more realistic model at the expense of computation time. Note, this is boolean and does not accept a value.
-sbr	--subregion	Selects the central N-by-N pixels to deconvolve. Restrict this to powers of 2 for optimal functionality.
-sl	--sigmalevel	Specifies how close to the estimated noise the algorithm will deconvolve. This is a multiplier with a default value of 4 - deconvolves to 4 sigma.
-lg	--loopgain	The gain factor analogous to that of CLEAN.
-tol	--tolerance	The tolerance factor for object extration. May usually be left as the default.

The arguments explained thus far may all alter the results and are also the only paramters which may be tuned. The remaining paramters, omitted here, but documented in the help function, are pricipally flags for enhanced operation modes. These include enabling and disabling GPU functionality, changing the way in which convolution is performed, and incorporating edge suppression.

A basic test problem would be to take an image of size $(1024, 1024)$, along with its PSF, and run the following command:

runsane dirty.fits psf.fits test_output --enforcepositivity

This command will work in most cases without any further input or tuning and concludes these basic instructions. For more advanced functionality, refer to the help functions or the code at (https://github.com/ratt-ru/PyMORESANE).

Bibliography

[1] C. Ferrari et al. Non-thermal emission from galaxy clusters: feasibility study with SKA1. In *Advancing Astrophysics with the SKA (AASKA14)*, 2014. URL http://arxiv.org/abs/1412.5801.

[2] A. Dabbech et al. MORESANE: MOdel REconstruction by Synthesis-ANalysis Estimators. *Submitted to A&A*, 2014. URL http://arxiv.org/abs/1412.5387.

[3] J. P. Hamaker, J. D. Bregman, and R. J. Sault. Understanding radio polarimetry. I. Mathematical foundations. *Astronomy and Astrophysics Supplement Series*, 117 (1), May 1996. URL http://dx.doi.org/10.1051/aas:1996146.

[4] O. M. Smirnov. Revisiting the radio interferometer measurement equation. I. A full-sky Jones formalism. *Astronomy and Astrophysics*, 527(A106), February 2011. URL http://dx.doi.org/10.1051/0004-6361/201016082.

[5] J. J. Condon and S. M. Ransom. Essential Radio Astronomy. URL http://www.cv.nrao.edu/course/astr534/ERA.shtml. Accessed: 2015-03-13.

[6] M. Born and E. Wolf. *Principles of optics*. Cambridge University Press, seventh edition, 1999. ISBN 978-0-52-164222-4.

[7] B. G. Clark. Coherence in Radio Astronomy. In *Synthesis Imaging in Radio Astronomy II*, volume 180 of *ASP Conference Series*, pages 1–10, 1999.

[8] A. Richard Thompson. Fundamentals of Radio Astronomy. In *Synthesis Imaging in Radio Astronomy II*, volume 180 of *ASP Conference Series*, pages 11–36, 1999.

[9] E. B. Formalont. Earth-rotation aperture synthesis. *Proceeding of the IEEE*, 61(9), September 1973. URL http://dx.doi.org/10.1109/PROC.1973.9247.

[10] V. Radhakrishnan. Noise and Interferometry. In *Synthesis Imaging in Radio Astronomy II*, volume 180 of *ASP Conference Series*, pages 671–688, 1999.

[11] J. A. Högbom. Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines. *Astronomy and Astrophysics Supplement*, 15, June 1974. URL http://articles.adsabs.harvard.edu/full/1974A%26AS...15..417H.

[12] B. G. Clark. An efficient implementation of the algorithm 'CLEAN'. *Astronomy and Astrophysics*, 89(3), September 1980. URL http://articles.adsabs.harvard.edu/full/1980A%26A....89..377C.

[13] F. R. Schwab. Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry. *Astronomical Journal*, 89, July 1984. URL http://articles.adsabs.harvard.edu/full/1984AJ.....89.1076S.

[14] T. J. Cornwell. Multi-Scale CLEAN deconvolution of radio synthesis images. *IEEE Journal of Selected Topics in Signal Processing*, 2(5), 2008. URL http://arxiv.org/abs/0806.2228.

[15] T. J. Cornwell and K. F. Evans. A simple maximum entropy deconvolution algorithm. *Astronomy and Astrophysics*, 143(1), February 1985. URL http://adsabs.harvard.edu/abs/1985A%26A...143...77C.

[16] J. Skilling and R. K. Bryan. Maximum entropy image reconstruction: general algorithm. *Monthly Notices of the Royal Astronomical Society*, 211(1), May 1984. URL http://mnras.oxfordjournals.org/content/211/1/111.full.pdf+html.

[17] H. Junklewitz et al. RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *Submitted to A&A*, February 2013. URL http://arxiv.org/abs/1311.5282.

[18] P. M. Sutter et al. Probabilistic image reconstruction for radio interferometers. *Monthly Notices of the Royal Astronomical Society*, 438(1), February 2014. URL http://dx.doi.org/10.1093/mnras/stt2244.

[19] E. J. Candès, J. K. Romberg, and T. Tao. Stable Signal Recovery from Incomplete and Inaccurate Measurements. *Communications on Pure and Applied Mathematics*, 59(8), 2006. URL http://arxiv.org/abs/math/0409186.

[20] H. Garsden et al. LOFAR Sparse Image Reconstruction. *Submitted to A&A*, 2014. URL http://arxiv.org/abs/1406.7242.

[21] R. E. Carrillo, J. D. McEwen, and Y. Wiaux. PURIFY: a new approach to radio-interferometric imaging. *Monthly Notices of the Royal Astronomical Society*, 439 (4), February 2014. URL http://dx.doi.org/10.1093/mnras/stu202.

[22] J. L. Starck, L. Fadili, and F. Murtagh. The Undecimated Wavelet Decomposition and its Reconstruciton. *IEEE Transactions on Image Processing*, 16(2), February 2007. URL http://dx.doi.org/10.1109/TIP.2006.887733.

[23] A. Dabbech, D. Mary, and C. Ferrari. Astronomical image deconvolution using sparse priors: An analysis-by-synthesis approach. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2012. URL http://dx.doi.org/10.1109/ICASSP.2012.6288711.

[24] S. Mallat. *A Wavelet Tour of Signal Processing: The Sparse Way*. Academic Press, third edition, 2009. ISBN 978-0-12-374370-1.

[25] T. Pham-Gia and T. L. Hung. The mean and median absolute deviations. *Mathematical and Computer Modelling*, 34(7), October 2001. URL http://www.sciencedirect.com/science/article/pii/S0895717701001091.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2), March 2008. URL http://dx.doi.org/10.1145/1365490.1365500.

[27] NVIDIA Corporation. CUDA C Programming Guide. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3I50O60zL. Accessed: 2014-11-04.

[28] A. Klöckner. PyCUDA. URL http://mathema.tician.de/software/pycuda/. Accessed: 2014-11-04.

[29] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. URL http://www.scipy.org/. Accessed: 2014-11-25.

[30] L. Givon. CUDA SciKit. URL http://scikit-cuda.readthedocs.org/en/latest/index.html. Accessed: 2014-11-13.

[31] J. C. Hsu, P. Barrett, C Hanley, J. Taylor, M. Droettboom, and E. M. Bray. PyFITS. URL https://pythonhosted.org/pyfits/. Accessed: 2014-11-14.

[32] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. URL http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf. Computer Science Notes.