

RHODES UNIVERSITY

Where leaders learn

DESIGN PATTERNS AND SOFTWARE TECHNIQUES FOR LARGE-SCALE, OPEN AND REPRODUCIBLE DATA REDUCTION

by

Gijs Jan Molenaar

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Supervised by

Prof. Oleg Smirnov

2021

Contents

Abstract v

Declaration of Authorship vii

Acknowledgements viii

Preface x

Publications xi

Source Code xii

1 History of data reduction pipelines 1
- 1.1 Introduction . . . 2
- 1.2 1930s . . . 2
- 1.3 1940s . . . 2
- 1.4 1950s . . . 5
- 1.5 1960s . . . 5
- 1.6 1970s . . . 7
  - Zero-generation calibration to first-generation calibration . . . 8
  - Astronomical Image Processing System . . . 10
  - CANDID . . . 10
  - CLEAN . . . 11
- 1.7 1980s . . . 11
  - Self-calibration and the second generation of calibration algorithms . . . 13
  - GIPSY . . . 13
  - DWARF . . . 14
  - IRAF . . . 15
  - Miriad . . . 15
- 1.8 Meanwhile in South Africa . . . 15
- 1.9 1990s . . . 17
  - AIPS++ . . . 17
  - NEWSTAR . . . 18
  - The Measurement Set version 1 . . . 18
  - The Measurement equation . . . 19
  - DIFMAP . . . 19
- 1.10 2000s . . . 19
  - The Measurement Set version 2 . . . 19

MeqTrees, third-generation calibration algorithms . . . 19
Obit . . . 19
CASA . . . 20
Casacore . . . 20
1.11 2010s . . . 21
Cuisine . . . 21
Docker . . . 21
The ALMA pipeline . . . 21
Common Workflow Language . . . 22
DDFacet and killMS . . . 22
Kliko . . . 22
Stimela . . . 23
CARACal (formerly known as MeerKATHI) . . . 23
Default pre-processing pipeline . . . 23
The LOFAR two-metre survey pipeline . . . 24
1.12 Overview of events . . . 25
1.13 Discussion . . . 27

2 Fundamentals **29**
2.1 Electromagnetic radiation . . . 30
2.2 Interferometry . . . 33
Aperture synthesis . . . 34
The (u, v, w) coordinate system . . . 36
The radio interferometer measurement equation . . . 37
2.3 Image reconstruction . . . 40
2.4 Calibration . . . 42
Reference calibration . . . 42
Self-calibration . . . 44

3 KERN **46**
3.1 Introduction . . . 47
3.2 The target platform . . . 47
3.3 Other packaging methods . . . 48
Anaconda . . . 48
Python and pip . . . 48
Collaboration with Debian . . . 49
3.4 Usage . . . 50
3.5 Notable packages . . . 50
Casacore . . . 50
Casacore data . . . 51
MeqTrees . . . 51
CASA . . . 51
AIPS . . . 52
LOFAR . . . 52
Pulsar software . . . 53
Unversioned packages . . . 53
3.6 Containerisation . . . 53
Docker . . . 53

Singularity . . . 53
3.7 Project structure . . . 54
The release cycle . . . 54
Technical structure . . . 54
3.8 Recommended usage . . . 55
3.9 Usage numbers . . . 55
3.10 Conclusions . . . 56

4 Kliko **57**
4.1 Introduction . . . 58
Software in science . . . 58
Software containerisation with Docker . . . 58
4.2 The Kliko specification . . . 60
The Kliko image . . . 60
Expected run-time behaviour . . . 60
Flavours of Kliko images . . . 61
The `/kliko.yml` schema . . . 61
The `/parameters.json` file . . . 62
4.3 Running Kliko containers . . . 63
Running a container manually . . . 63
Inside the Kliko container . . . 64
Kliko-run . . . 64
4.4 Chaining containers . . . 65
4.5 Example of usage of Kliko . . . 66
VerMeerKAT . . . 66
RODRIGUES . . . 67
4.6 Software availability . . . 70
4.7 Discussions and prospects . . . 70
Limitations . . . 70
Future work . . . 71
4.8 Conclusions . . . 71

5 CWL and Buis **73**
5.1 Introduction . . . 74
5.2 The CommonWL standard . . . 74
CommandLineTool class file . . . 75
Job file . . . 75
Workflow class file . . . 75
Runners . . . 76
5.3 Buis – the web-based frontend for CommonWL runners . . . 76
Functional design . . . 76
Technical design . . . 78
Usage . . . 79
5.4 Use case example: a 1GC pipeline . . . 80
5.5 Discussion . . . 83

6 Vacuum Cleaner **85**
6.1 Introduction . . . 86
6.2 Radio interferometric imaging . . . 86
6.3 Method . . . 87
Network architecture . . . 88
Objective function . . . 89
Implementation . . . 90
Training . . . 90
6.4 The simulation . . . 91
6.5 The results . . . 93
Scoring . . . 93
Evaluation . . . 93
Computational performance . . . 94
6.6 Conclusion and discussion . . . 96
6.7 Future work . . . 97

7 Conclusions **98**

A KERN packages **101**

B Kliko specification and example **109**
B.1 An example of a kliko.yml file . . . 110
B.2 The Kliko validation specification . . . 111

Abstract

The preparation for the construction of the Square Kilometre Array, and the introduction of its operational precursors, such as LOFAR and MeerKAT, mark the beginning of an exciting era for astronomy. Impressive new data containing valuable science just waiting for discovery is already being generated, and these devices will produce far more data than has ever been collected before. However, with every new data instrument, the data rates grow to unprecedented quantities of data, requiring novel new data-processing tools. In addition, creating science grade data from the raw data still requires significant expert knowledge for processing this data. The software used is often developed by a scientist who lacks proper training in software development skills, resulting in the software not progressing beyond a prototype stage in quality.

In the first chapter, we explore various organisational and technical approaches to address these issues by providing a historical overview of the development of radio-astronomy pipelines since the inception of the field in the 1940s. In that, the steps required to create a radio image are investigated. We used the lessons-learned to identify patterns in the challenges experienced, and the solutions created to address these over the years. The second chapter describes the mathematical foundations that are essential for radio imaging. In the third chapter, we discuss the production of the KERN Linux distribution, which is a set of software packages containing most radio astronomy software currently in use. Considerable effort was put into making sure that the contained software installs appropriately, all items next to one other on the same system. Where required and possible, bugs and portability fixes were solved and reported with the upstream maintainers. The KERN project also has a website, and issue tracker, where users can report bugs and maintainers can coordinate the packaging effort and new releases. The software packages can be used inside Docker and Singularity containers, enabling the installation of these packages on a wide variety of platforms.

In the fourth and fifth chapters, we discuss methods and frameworks for combining the available data reduction tools into recomposable pipelines and introduce the Kliko specification and software. This framework was created to enable end-user astronomers to chain and containerise operations of software in KERN packages. Next, we discuss the Common Workflow Language (CommonWL), a similar but more advanced and mature pipeline framework invented by bio-informatics scientists. CommonWL is supported by a wide range of tools already; among other schedulers, visualisers and editors. Consequently, when a pipeline is made with CommonWL, it can be deployed and manipulated with a wide range of tools.

In the final chapter, we attempt something unconventional, applying a generative adversarial network based on deep learning techniques to perform the task of sky brightness reconstruction. Since deep learning methods often require a large number of training samples, we constructed a CommonWL simulation pipeline for creating dirty images and

corresponding sky models. This simulated dataset has been made publicly available as the ASTRODECONV2019 dataset. It is shown that this method is useful to perform the restoration and matches the performance of a single clean cycle. In addition, we incorporated domain knowledge by adding the point spread function to the network and by utilising a custom loss function during training. Although it was not possible to improve the cleaning performance of commonly used existing tools, the computational time performance of the approach looks very promising. We suggest that a smaller scope should be the starting point for further studies and optimising of the training of the neural network could produce the desired results.

Declaration of Authorship

I, Gijs Molenaar, declare that this thesis titled, 'Design patterns and software techniques for large-scale, open and reproducible data reduction' 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: _____

Acknowledgements

During this study, I spoke to many people in the field about the history of radio astronomy pipelines. I found it fascinating, but also challenging, to give a proper overview of the essential subjects, while not getting lost in the volumes of information and the details. I hope I was still able to give a decent overview of the field. However, I have to admit that it feels as if the history chapter does not do justice to the greatness of this subject itself and I am sorry if I neglected to mention essential names or milestones that I may have forgotten.

First and foremost, I want to thank my supervisor, Oleg Smirnov, for his overall guidance and wise words and advice, which always seem to be given at the perfectly timed moment to keep my productivity up. I want to thank Landman Bester and Sphesihle Makhathini for working together with me on chapters five and six; I would not have been able to complete these without their invaluable input.

The incomplete and randomised list of people who contributed to the history chapter of this thesis is:

- Oleg Smirnov, Rhodes University and SARAO
- Elizabeth Waldram, University of Cambridge
- Bernie Fanaroff, SKA South Africa
- Jan Noordam, ASTRON
- Tim Pearson, CIT
- Ger van Diepen, ASTRON
- Wim Brouw, Kapteyn Institute
- Bob Sault, ATNF
- Gareth Hunt, NRAO
- Bill Cotton, NRAO
- Eric Greisen, NRAO
- Jeff Kern, NRAO
- Brian Glendenning, NRAO
- Rick Perley, NRAO

- Peter Teuben, University of Maryland
- Andre Offringa, ASTRON
- Cyril Tasse, GEPI, Observatoire de Paris, CNRS
- Justin Jonas, Rhodes University and SARAO
- Sarah White, Rhodes University

I want to thank these people who took the time to answer my e-mails containing numerous questions. My thesis is finished, and I hope it is a testament to these people listed above and to the fantastic field that radio astronomy is.

Last, but by far not least, I want to thank and praise my fiancée, best friend and future wife, Megan Galloway, who I bored to death with reviewing this thesis. Without her invaluable input, comments and corrections, this document would not have reached its current quality.

Preface

My PhD journey started when I was invited to South Africa by my supervisor, Oleg Smirnov, to help him create packages for the tools used in aperture synthesis. While working at SKA South Africa (now SARAO) in Cape Town, I decided that I wanted to contribute to this field of science and progress my career as a research engineer by furthering my studies through a PhD.

When Oleg and I sat down to determine the subject and direction of my research, he suggested I should focus on data reduction pipelines. I was unaware then, that this is a very active and exciting topic, which offers the opportunity to have an impact on the field. I started my work by creating KERN packages of existing software. I find this part of my work to be the least scientific, as it is pure software engineering, but ironically this work seems to have the highest impact, as it is used by scientists all around the world. This fills me with great pride. I hope the other chapters in this thesis will be contributing to the start of a worldwide collaboration on constructing cross-platform data reduction pipelines.

Since I started working on my thesis, science and the software industry have become even more closely linked. Even a company such as Microsoft, once the detractor of free software, became one of the most prominent open-source companies in the world. I believe there is only one way forward, and that is to grow the openness of the radio astronomy software development process and encourage collaboration around the world.

Relating to this, over the last few years the scientific world has changed slightly. The appreciation for non-scientists working in science is growing, and the community has coined a name for these professionals: the research engineer. James Watson, the molecular biologist, famously advised to try to avoid being the smartest person in the room. I think being a research engineer is the best possible way of doing this, as it allows you to be close to, and learn from, the most intelligent and inspiring people in the world. I am grateful for the opportunity to do so during my research and hope to be able to continue doing so throughout my career.

Publications

Over the course of this research project, the author was involved with the following publications:

- 'Kern', Astronomy and Computing 24 (Molenaar and Smirnov, 2018)
- 'Kliko; The scientific compute container format' (Molenaar, Makhathini, et al., 2018)
- 'The LOFAR transients pipeline' (Swinbank et al., 2015)
- 'LOFAR MSSS: detection of a low-frequency radio transient in 400 h of monitoring of the North Celestial Pole' (A. J. Stewart et al., 2015)
- 'Pulsar polarisation below 200 MHz: Average profiles and propagation effectsd' (Noutsos et al., 2015)
- 'Low-radio-frequency eclipses of the redback pulsar J2215+ 5135 observed in the image plane with LOFAR' (Broderick et al., 2016)
- 'PySE: Software for extracting sources from radio images' (Carbone et al., 2018)
- 'AARTFAAC flux density calibration and Northern hemisphere catalogue at 60 MHz' (Kuiack et al., 2018)

Source Code

New software projects

In the course of this research project, the software programs listed hereafter were produced. These are all open-source and available for use free of charge. The source code for these programmes should be considered an integral part of this work.

KERN

KERN is a Linux distribution of radio astronomy tools. KERN is further discussed in Chapter 3.
website: https://kernsuite.info/

Kliko

Kliko is the scientific compute container specification and implementation. Kliko is discussed in Chapter 4.
website: https://github.com/gijzelaerr/kliko

RODRIGUES

RODRIGUES is a web-frontend for Kliko, which is also discussed in Chapter 4. website: https://github.com/ska-sa/rodrigues/

Buis

Buis is a web-frontend for CWL runners. The software is discussed in Chapter 5.
website: https://github.com/gijzelaerr/buis/

Vacuum Cleaner

Vacuum Cleaner is a deep learning (TensorFlow) based image brightness reconstruction program for radio astronomy images The project architecture and performance are discussed in Chapter 6.
website: https://github.com/gijzelaerr/vacuum-cleaner

Contributions to existing projects

In addition, the author was involved with the following open-source software projects.

Common Workflow Language

The Common Workflow Language plays a vital role in the second half of this thesis. The author was in close dialogue with the upstream maintainers, fixing and reporting bugs, testing new features and last but not least, contributing modifications to the CWL specification crucial for the adaptation of the library in the radio astronomy field. This feature is enabled with the *InplaceUpdateRequirement* flag[1]. The Common Workflow Language is further discussed in Chapter 5.

website: https://github.com/common-workflow-language

Casacore and Python-Casacore

The astronomy packages to which the author contributed the most time by far are Casacore and Python-Casacore. Casacore is a suite of C++ libraries for radio astronomy data processing, while Python-Casacore is the Python wrapper for these libraries. Casacore contains the core libraries of the old AIPS++/CASA package. For both projects, the author operated as a release manager and coordinated new releases, including feature management. The author moved the projects to Github, introduced containerised continuous integration infrastructure, and constructed binary wheels for Python-Casacore.

Website for Casacore: https://github.com/casacore/casacore

Website for Python-Casacore: https://github.com/casacore/python-casacore

[1] https://www.commonwl.org/v1.1/CommandLineTool.html#InplaceUpdateRequirement

Chapter 1

History of data reduction pipelines

1.1 Introduction

To understand why making data reduction pipelines for aperture synthesis is still a challenging problem requiring active research, it is necessary and exciting to look at the history of aperture synthesis and the attempts that have been made to create one-for-all data reduction pipelines.

Since radio astronomy is a relatively young field, the current radio astronomy generation is fortunate in that many legendary people who have defined the path of this field are still alive. This introduction gives a summary of the events, insofar as it is possible. This summary includes the often accidental discovery of phenomena or computational behaviour, the invention of algorithms and the birth of essential software projects, which are often still in use today. It is easy to get distracted by the many interesting facts and factoids about this young science, but the focus of this thesis is aperture synthesis and software pipelines. Meanwhile, an attempt is made to learn from success and failure stories and extract valuable lessons for future data reduction work.

Much of this chapter is based on personal (or e-mail) interviews with the various players in the field conducted by the author. In the interest of flow, the author omits the ubiquitous (private communication) citation where such interviews are cited. Private communication should be assumed unless a citation is explicitly given. Please note that the level of detail in what follows is not always consistent from subject to subject. Some interviewees were able to provide rich detail and anecdotes, while sources for other subjects remained sparse. We have taken the editorial decision to present the available content as is.

1.2 1930s

A historical overview of radio astronomy is incomplete without mentioning the field's founding father, Karl Guthe Jansky. In 1931, at the age of 26, while working for Bell Labs investigating sources of static that might cause interference to radio voice transmissions, Jansky discovered radiation in the radio spectrum coming from the Milky Way. This discovery was the first time this kind of radiation was measured (see Figure 1.1). Jansky (1933) published a paper titled 'Electrical disturbances apparently of extraterrestrial origin'. Unfortunately for Jansky, the astronomy community mostly ignored this paper. It was only after his death in 1950 that astronomers started looking more closely at his findings.

Fortunately, at least one person did not ignore Jansky's results. In 1938 a young man named Grote Reber made the first purpose-built parabolic radio telescope in his backyard to continue the research of Jansky. In 1940 he published a paper in the Astrophysical Journal and remained the only radio astronomer for nearly a decade. His radio sky map, finished in 1941, first showed the existence of the bright radio sources Cygnus A and Cassiopeia A (Baade and Minkowski, 1954).

1.3 1940s

Meanwhile, in 1944, the Dutch scientist Hendrik van de Hulst predicted that neutral hydrogen should produce electromagnetic radiation as a result of energy state change. The radiation was predicted to have a frequency of $f \approx 1420$ MHz or a wavelength of

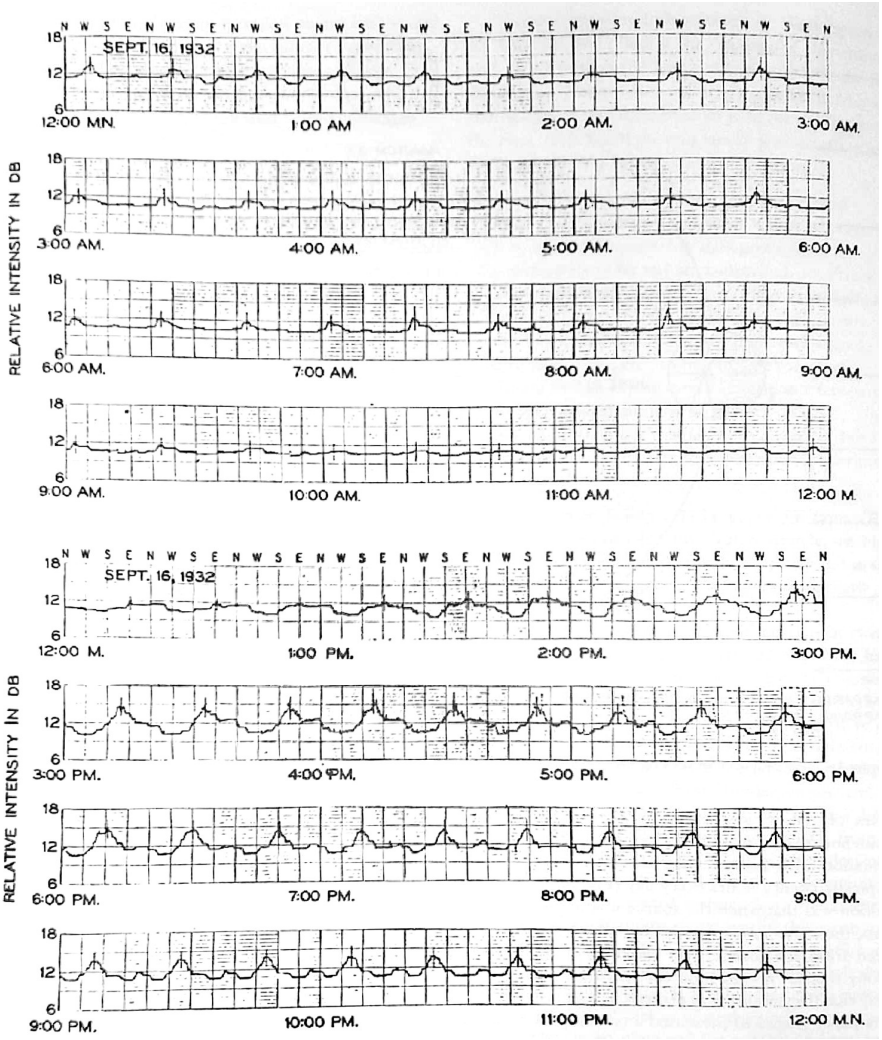


Figure 1.1: Jansky's strip chart recording of intensity changes, one of the few documents remaining from this work. Image taken from Sullivan (2009)

$\lambda \approx 0.21$ m. This phenomenon became known as the hydrogen line, 21-centimetre line, or HI line. Since hydrogen is the most abundant element in the universe, van de Hulst expected the sky to be saturated with this signal. Furthermore, electromagnetic radiation in this frequency range can pass through the earth's atmosphere with little interference, and therefore should be observable from the ground. Van de Hulst realised this, and together with his supervisor, Jan Oort, successfully lobbied for the development of radio telescope technology in the Netherlands.

Oort was one of the first astronomers to realise the potential of radio astronomy and foresaw the engineering challenges this novel field would encounter. In subsequent years, Oort would eventually build up a network of both astronomers and engineers, gather funds and lay the foundations for making the Netherlands one of the top players in the field.

Around the same time, the Second World War in Europe ended. The upcoming reconstruction era became the perfect birthplace for the radio astronomy field in the Netherlands specifically, since it was easier to argue for funds for radio telescopes than

other telescopes. Radio telescopes can be built and operated at sea level, meaning the money would be spent inside the Netherlands, which would benefit the reconstruction of the country. In addition, the field would go on to benefit from the radar technology developed during the war all around the world.

Radio astronomy has benefited from radar technology, since they are technically closely related. Radar is based on broadcasting a radio signal and detecting its reflection from metallic objects of interest at a considerable distance, while astronomy observes natural radio waves from outer space. Radar technology developed during the war became of great use for radio astronomy. A prime example inspired by cutting edge radar technology is the telescope constructed in 1946 on the cliff at Dover Heights by The Commonwealth Scientific and Industrial Research Organisation (CSIRO) in Australia to measure the interference between the direct waves and waves reflected by the sea. This *cliff interferometer* was built to locate the origin of the solar radio emission. The idea of a cliff interferometer came from *multiple-path interference* already used in ship-mounted radar in the war, which was used to improve positional estimates. With the additional resolution resulting from employing interferometry, Ruby Payne Scott discovered that radio waves from the sun were related to sunspots.

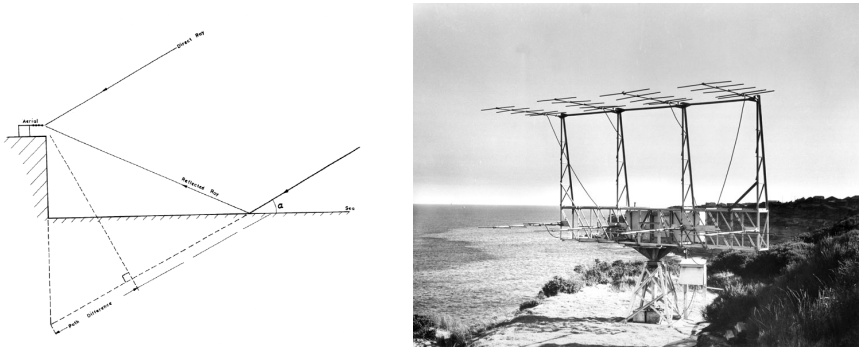


Figure 1.2: The sea cliff interferometer

The Owens Valley Radio Observatory (OVRO), which would later become an important player in the field, was founded by the California Institute of Technology (Caltech) in the late 1940s.

At Cambridge, Ryle and Vonberg (1946) were the first astronomers to apply interferometry to astronomical measurements at radio wavelengths successfully. Aperture synthesis was born.

Meanwhile, in the Netherlands, the Netherlands Institute for Radio Astronomy (ASTRON) was founded in 1949, initially named the 'Foundation for Radio Radiation from the Sun and Milky Way' (SRZM).

The first British electronic computer, named Edsac (for Electronic Delayed Storage Automatic Computer), came into service in 1949 (Edsac will be further elaborated on below). Computers began to play a vital role in radio astronomy, and the development of these fields over the next decades are closely linked. The memory unit of Edsac was based on primitive mercury tubes carrying acoustic signals, and contained a total of 3800 valves.

1.4 1950s

At the beginning of the 1950s, Jansky passed away due to a heart condition, sadly missing most of the impact of his findings.

IBM created the *Fortran* computer programming language for scientific and engineering applications. This language has been used extensively in radio astronomy, and is still frequently used to this day, especially in scientific computations.

Meanwhile, Martin Ryle published a paper that described the use of the earth's rotation to improve the (u, v) coverage (this principle is explained further in the fundamentals, Chapter 2) of an observation (Ryle, F. G. Smith, et al., 1950). It would take a decade before this method could be put to the test, simply because computers were not powerful enough to perform computations on the required scale.

Around the same time in the USA, the predicted 21 cm (1420 MHz) hydrogen line was first detected by Ewen and Purcell (1951). Both had experience with radar technology and they decided to use a horn-shaped dish in an attempt to detect the hydrogen line - with success. Dutch astronomers detected the line around six weeks later, and both teams published their findings simultaneously.

The National Radio Astronomy Observatory (NRAO) was founded in 1956 in Green Bank, West Virginia, USA. Given the strength of the US economy, and the overlap with related fields involved in the Cold War, this group was well-funded. Native English-speaking countries such as Australia and the UK could benefit greatly from this fast progress. Some other countries also benefited from the US slipstream: the Dutch, with their early participation in the field and excellent English language skills, were assured of a position among the top radio astronomy countries.

In 1958, the Edsac II electronic computer replaced the Edsac I computer in Cambridge. In this design, the memory was based on small ferrite rings, which could be magnetised in one or two directions for boolean values, and contained about 6 000 valves.

In 1958, two unnamed 27-metre telescopes were constructed at OVRO. Eric Greisen, who has been interviewed for this chapter, would later use these to study interferometry.

1.5 1960s

Both the One-Mile Telescope in Cambridge, UK and the two-dish version of the Green Bank Interferometer (GBI) in West Virginia, USA were completed in 1964. In 1967, GBI was extended to a three-dish configuration. The One-Mile Telescope was the first telescope to utilise the earth's rotation to improve (u, v) coverage.

When Elizabeth Waldram – who joined the Cambridge group in 1960 – was asked in an interview for the Cavendish magazine (February 2014) about early radio astronomy software and pipelines, she stated that it was Ann Neville[1] (Figure 1.3) who, as a research student, implemented Ryle's idea to utilise the earth's rotation. Waldram continues: 'It proved a tremendous success: a region of diameter 8° about the North Celestial Pole (NCP, Figure 1.3) was mapped with a resolution of 4.5 arcmin and with some eight times the sensitivity of earlier surveys'. Neville was responsible for the huge programming task of organising the raw data and programming the Fourier transform. Waldram very modestly overlooks her own crucial contribution, which was, arguably, the invention of convolutional gridding. Gridding is the process of converting irregularly sampled visibility

[1] née Neville, subsequently known by her married name, Gower.

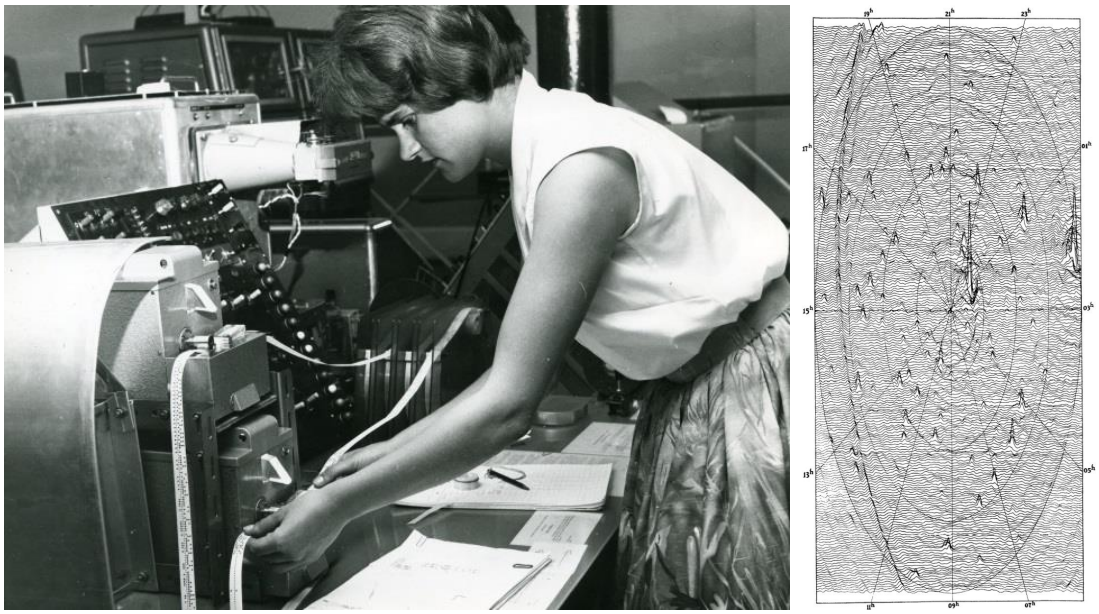


Figure 1.3: On the left, Neville feeding paper tapes from observations for the North Pole Survey into Edsac II. On the right, a radio map of the North Celestial Pole region. This was the deepest image of the sky at the time (Ryle and Neville, 1962)

data onto a regular grid prior to the Fourier transform, and it remains a crucial piece of every interferometric imaging package to this day.



Figure 1.4: On the left, one of the original 35 mm negatives used to construct the map in Figure 1.3. On the right, Waldram in 2019 (with Steve Gull and James Kent, photo by Smirnov.)

All these computations were performed on the Edsac II computer at the University Mathematical Laboratory of Cambridge, which was quite a challenge in itself. The computer was controlled by paper tape and all programming was done in machine language. The central compute unit was based on thermionic valves with limited life, so the machine would fail on average every few hours. Fires were frequent. Meanwhile there was an intense competition for computing among various fields - genetics, physics, number

theory, economy and molecular biology all wished to crunch numbers on this computer.

This was, perhaps, also the birth of computer-based visualisation. The Neville-Ryle map in Figure 1.3 was produced by plotting small sections of the countour plot on a cathode-ray tube attached to the Edsac II, photographing the display, and then combining the 48 individual 35 mm photographs. The plotting software was also developed by Waldren.

Figure 1.4 is a photograph of one of the original 35 mm negatives.

Around 1963 the Edsac II was slowly replaced by the Titan computer built by the Belgian computer manufacturer, Ferranti. At the time the Titan was a technological wonder, with 32 K of 38-bit words of memory, compared to the 1 K of 40-bit words of memory in Edsac II. Nonetheless, the large quantities of data that had to be processed were pushing the machine to its limits, and low-level machine code optimisation involving manipulation of bit patterns was required. Waldram writes in an article in Cavendish magazine that thanks to the help of David Wheeler from the Cambridge Mathematics Laboratory the group got access to the new Fast Fourier Transform (FFT) algorithm, years before it was officially published. Still, it took an hour or more of machine time to map 10 square degrees of sky. [2]

In 1966 Greisen joined Caltech as a graduate student. He used a 'package' of Fortran programs to process the data from the two 90-foot CalTech telescopes. Greisen had to study this package to make sure that an observational strategy would not run into computation issues. That led to him becoming the expert on that software, to eliminate several bugs in it, and eventually to rebuild the code to handle a three-element array.

Meanwhile, the PDP-10 mainframe computer family, also known as the DEC-10, was taken into production by Digital Equipment Corporation (DEC) in 1966. This computer would play a significant role in the digital processing of the radio astronomy data and would be used at all institutes around the world. The PDP-10 architecture is based on a 36-bit word length, which is surprising by modern standards.

Construction of the Westerbork Synthesis Radio Telescope (WSRT) in the Netherlands began in 1966. The telescope was initially conceived as the Benelux Cross Antenna Project and was to be built close the Belgian border. However, Belgium dropped out of the project, and the decision was taken to scale down and relocate the project to Westerbork.

1.6 1970s

DEC launched the PDP-11, a series of 16-bit *minicomputers*. There is no clear definition of a minicomputer. However, the New York Times suggests this definition: 'costing less than $ 25 000 ($ 163 500 in 2018), has some type of input-output devices such as a teleprinter, memory of about 4 000 words, and with circuitry capable of performing

[2] To transform the radio astronomy data from the visibility space to the image space, the Discrete Fourier Transform (DFT) is used. Calculating the DFT requires N^2 arithmetic operations, making it a very costly algorithm. Fortunately, in 1965, Cooley and Tukey (re-)discovered a much faster method to compute the same results (Cooley and Tukey, 1965). They named this algorithm the Fast Fourier Transform (FFT), which only requires $N \log N$ arithmetic operations and is today still the preferred way for calculating the DFT. The word rediscovered is used, since Gauss actually wrote down similar techniques to calculate the FFT around 1805. These writings went mostly unnoticed for more than a hundred years until rediscovery (Heideman et al., 1984).

calculations under the control of stored programs written in some form of higher-level computer languages such as Fortran or Basic' (W. D. Smith, 1970).

About 600 000 PDP-11s were sold, making it one of the most successful minicomputers ever produced. Several innovative features in its instruction set and an additional set of general-purpose registers made the system more comfortable to use than previous models and inspired the design of now popular microprocessor architectures such as the Intel x86. The PDP-11 would dominate data-processing in radio astronomy for the following period.

Queen Juliana of the Netherlands officially opened the WSRT in 1970. WSRT consists of 14 dish antennas, each with a diameter of 25 metres. The array is placed on an east-west line, with the most extended baseline spanning three kilometres. Ten of the dishes are located at a regular interval of 144 metres on a straight line. Nowadays, this configuration would be considered inefficient, since it contains multiple redundant baselines that observe the same (u, v) point. At the time, computers were not powerful enough to be able to handle calibration algorithms with too many baselines, so the configuration made sense, and the original plan called for correlating only the fixed-movable pairs of dishes.

Tim Pearson, who would later create the famous PGPlot and the Caltech Package, started his career in 1972 at Cambridge, where he worked with the One-Mile Telescope. There he started working on data reduction/mapping programs in Fortran.

Ryle and Antony Hewish received a Nobel prize for physics in 1974, which was the first Nobel prize that recognised astronomical research. Hewish received the award for his role in the discovery of pulsars. This award is not without controversy, since it was his PhD student, Jocelyn Bell, who discovered the first pulsar. In 2018 Bell was awarded the $ 3 million Breakthrough Prize for her discovery.

Around 1975, Jan Noordam joined ASTRON, and Ger van Diepen joined the same institute in 1978. At that point, WSRT data was supposed to be reduced in Leiden on IBM computers. Van Diepen joined a team of four, under supervision of Harten, with the team goal being to bring the processing, and specifically the calibration, to ASTRON. This was based on software written initially by Wim Brouw. A PDP-11 running the RSX11M operating system was used, but later also VAX-780 running VMS, a computer based on the PDP-11 but using a 32-bit architecture. These machines were also programmed in Fortran, with some manual assembler programming. Although all imaging and calibration was supposed to be performed in Leiden, Johan Hamaker (ASTRON) had been secretly working on a calibration method called *kneading* (Hamaker, 1979). Kneading is a systematic approach to adjust the instrumental phase and gaining parameters to suppress error patterns in a synthesis map. The results can be seen in Figure 1.5. Hamaker stored his data in an improvised 'uv-data' format, which was later used by Noordam to develop redundant spacing calibration (RSC).

Zero-generation calibration to first-generation calibration

The 1970s marked what was, in hindsight, a crucial development: the emergence of antenna-based calibration. The thought process leading to this has been remarkably difficult to pin down in the literature. However, one may safely assume that early radio interferometers were calibrated in what is physically the most straightforward and obvious way, i.e. by treating each baseline as an independent measurement device, and estimating the *baseline-based* gain from observations of a reference source. As correlator

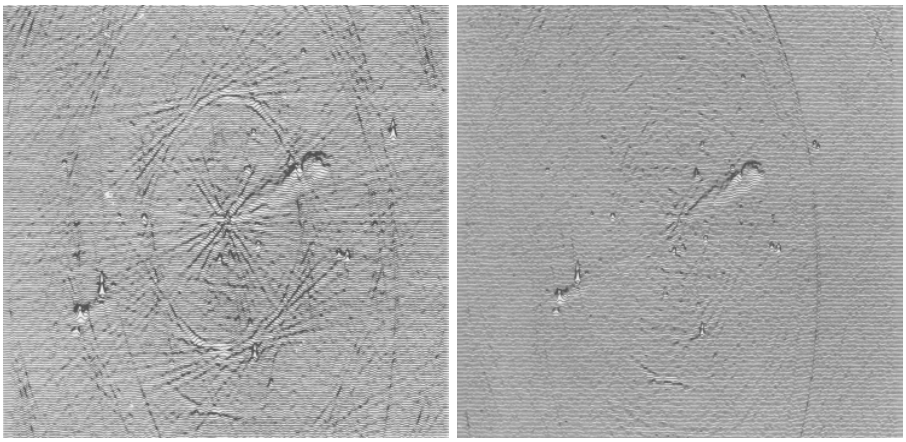


Figure 1.5: Map of 3C236 made from four 12-hour observations. The figure on the left is a dirty map with the bright centre point source subtracted, the one on the right shows the image after kneading was applied. Images taken from Hamaker (1979)

technology improved (thus reducing the influence of the correlator errors on the output), it was realised that the response of a given baseline was actually a product of two *antenna-based* gains (see, e.g. the radio interferometer measurement equation (RIME), Eq. 2.50, for a modern treatment of this.) For a three-element interferometer, this offers no real advantages (three baseline gains are refactored into three-antenna gains), but for larger numbers of elements, this quickly leads to a significant reduction in the number of unknowns needing to be estimated during calibration – a line of reasoning that eventually resulted in the idea of self-cal (see the 1980s).

According to Rick Perley, the antenna-based calibration approach was originally pioneered by Barry Clark on the Green Bank three-element interferometer. Clark would later fully adopt this method for the very large array (VLA) from the very start. By contrast, Michiel Brentjens confirms (Perley, priv. comm.) that the WSRT initially employed a baseline-based approach. (Since only the fixed-movable antenna pairs were to be correlated, an antenna-based approach would not have offered obvious advantages anyway.) Early VLA calibration linearised the antenna-based calibration equations by taking the logarithms of the visibilities. This simplified the calculations but did not treat the measurement noise in a statistically correct way. Finally, in 1975, D'Addario introduced a least-squares approach (VLA Memo #119[3]), which still underpins antenna-based calibration to this day.

In the three generations of calibration nomenclature introduced by Noordam and Smirnov (2010), this historical distinction is overlooked. The authors propose the term *first-generation calibration* (1GC) to refer to calibration using reference sources. In modern parlance, 1GC is most often used specifically in reference to *antenna-based* calibration using reference sources. In the interest of historical accuracy, we propose the term zero-generation calibration (0GC) for the older *baseline-based* approach.

[3]https://library.nrao.edu/public/memos/vla/sci/VLAS_119.pdf

Astronomical Image Processing System

Around this time, the first predecessor of modern data reduction packages started emerging. In 1978, first steps were taken to create the Astronomical Image Processing System (AIPS) in Charlottesville, Virginia. One of the target platforms for AIPS was the VAX 11/780.

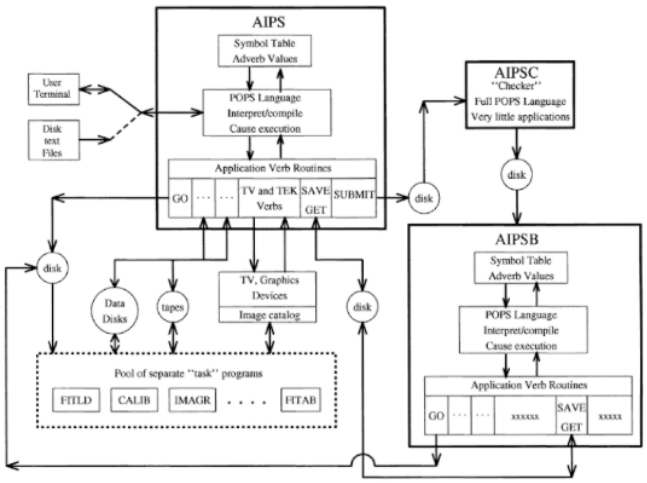


Figure 1.6: AIPS from the user's point of view. Image taken from Greisen (2003)

Another crucial development was the emerging of data standards. Until that time, all institutes used their own storage format, making it hard to exchange data. The Flexible Image Transport System (FITS) standard was created by Wells and Greisen (1979). This standard was an attempt to serve all astronomers in all frequencies and was tape storage and punch card friendly. Unfortunately, the first FITS version did not support all radio interferometric feature requirements. Initially, the standard only supported basic images and visibilities, while the AIPS developers envisioned a more versatile format that would support clean component tables, antenna tables, and more. This led to the formalisation of the generalised extension format for FITS (Harten et al., 1988), which enabled the AIPS developers to extend the storage format to their needs, eventually resulting in the UVFITS format. Thanks to the popularity of AIPS, UVFITS remains in use to this day.

Greisen states: 'the initial VLA data-processing software was a translation of the GBI software. Then on a dare from the programmers at the VLA, Schwab and Greisen wrote a new package in 7 days in PL/1 which did all sorts of calibration, including basic self-cal (not called that then), and imaging and map-plane Clean.'

CANDID

Bill Cotton (NRAO) stated that in 1974, Bob Hjellming developed the first data-processing software for the VLA. All VLA data calibration was performed on a DEC-10 running the TOPS10 operating system and was written in the 'SAIL' programming language. SAIL stands for 'Stanford Artificial Intelligence Language' and is a modified version of ALGOL. ALGOL itself is short for 'algorithmic language', an imperative programming language developed in the mid 1950s. The code to do the calibration was called CANDID

(Command and Algorithm Notation for Data Inundation Device). Here, 'inundation device' refers to the VLA, since it was the first device ever to generate this much data. This was 1970s slang for 'Big Data' (thus proving that neither Big Data nor humorous and somewhat contrived acronyms are a modern phenomenon). CANDID included a control language with the vision of doing pipelined analysis. However, the implementation was much too slow. It was replaced by a suite of programs that used the same underlying disk format, but was completely independent. The identity of the author of this rewrite, unfortunately, is unknown. These were loosely known as the 'DEC-10 package'. They were usually executed manually, although a few did use the TOPS10 command language to string jobs together.

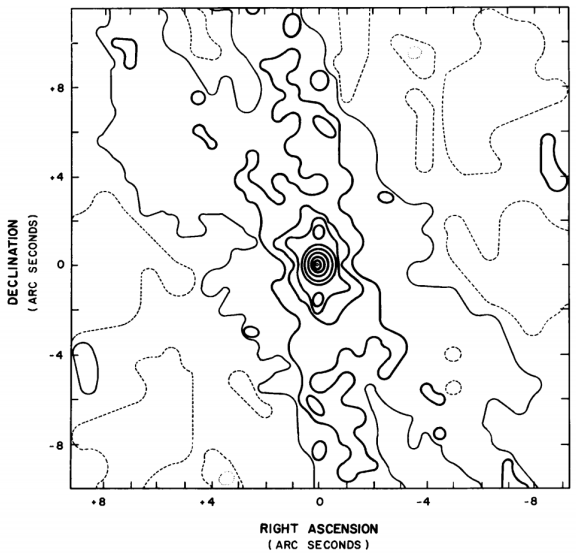


Figure 1.7: Response to a point-source, from the VLA user manual, April 1977: 'Synthesised beam of the 1977 VLA. Declination 40°, hour angle coverage −6 to +6 hours. Contour levels are at intervals of 10% of the peak response. 10% contour is shown faint, zero contour is omitted, negative contours are dashed.'

CLEAN

In the same year, Högbom (1974) published a paper describing CLEAN, the legendary algorithm still in use today for removing convolution artifacts in radio astronomy images (deconvolution). The algorithm assumes that the sky consists of discrete point sources. It proceeds by finding the peak value in the image, and subtracting the point spread function (PSF) at that position, multiplied by a fraction of the peak value (given by the *loop gain* parameter, typically $\lesssim 0.1$). This process is iteratively repeated until the peak value becomes smaller than a specified threshold, or a set number of iterations is reached.

1.7 1980s

During the 1980s, work began on a software pipeline to produce images directly from the DEC-10 computer. Clark worked on a pipeline named ISIS, which took data directly

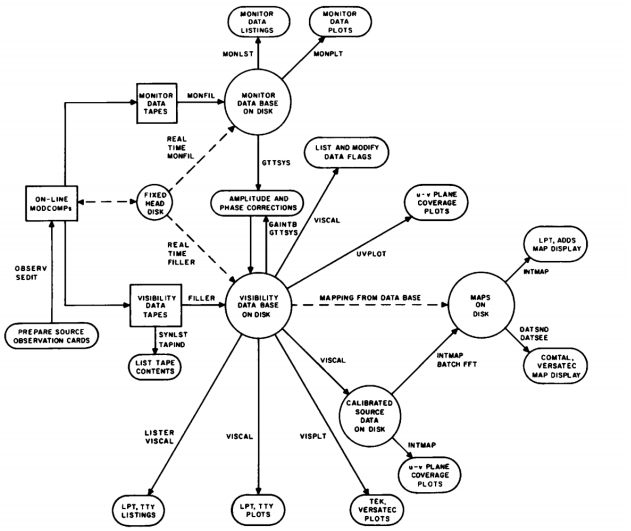


Figure 1.8: Offline processing, from the VLA user manual, April 1977: 'Circles represent data storage on disk, squares represent magnetic tapes, ovals represent things to be accomplished, and arrows are associated with program names to indicate what data is accessed to accomplish what purpose with what program. A dashed arrow indicates features that will be implemented later in 1977.'

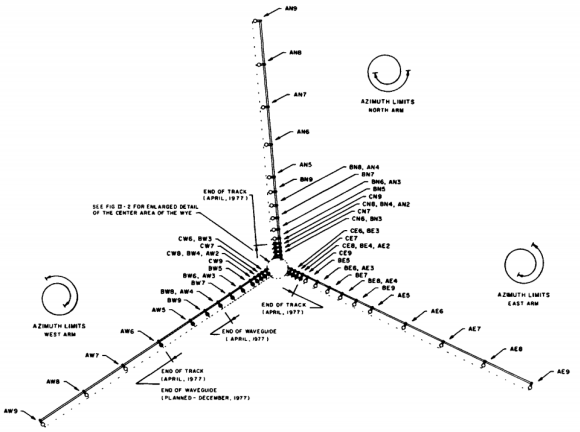


Figure 1.9: VLA layout, from the VLA user manual, April 1977

from the DEC-10 and processed it using a Convex array of vector processors controlled by PDP-11s. According to Gareth Hunt, this was not successful for a variety of reasons: 'First, the DEC-10 had a word length of 36-bits (this was built before the world standardised on 8-bit bytes) with its own floating-point format. The PDP-11s did have 8-bit bytes, but the data formats had to be translated to its floating-point format. The array processors had yet another word length; these led to data conversion issues. Second, with a variety of computer hardware, error handling was inconsistent. Thus the whole system

was not stable enough'.

Brouw was at the VLA in 1980/1981 where he worked on a special array processor with large RAM, which was used in between two PDP 11/45s, raw data on one side and calibrated images on the other. Maps of up to a resolution of 2 000 by 2 000 pixels were produced. Meanwhile, in 1980 the construction of all the 27 dishes of the Very Large Array (VLA) was completed. At the same time, the first public version of AIPS was released (Wells, 1985).

In 1989 Hunt joined the AIPS group and worked with the system developed by Greisen and Cotton. By that time, AIPS also included synthesis calibration, which was required to support very long baseline interferometry (VLBI). Many of the tasks in AIPS developed around that time are still used for the VLA.

Self-calibration and the second generation of calibration algorithms

With the advent of digital correlators and the emergence of antenna-based calibration (see discussion on 0GC and 1GC above), astronomers slowly came to the realisation that their calibration problem was overconstrained. When a digital (and thus, presumably, error-free) correlator is used, it can be assumed that the instrumental gain of a baseline is the product of the complex gains of two antennas. For example, for WSRT, this implies that the measurements made by 91 baselines share only 14 independent errors that require calibration. This reduction in free parameters gave rise to the ability to *self-calibrate* the telescope during an observation. This gave rise to the era of self-cal, or second-generation calibration (2GC), explained in more detail in Subsection 2.4.

The essentials of self-cal (in the guise of *hybrid mapping*) were described by Cornwell and Wilkinson (1981). As often seen in history, some discoveries are arrived at almost simultaneously because 'the time is right'. Around that time, Noordam and Ger de Bruyn devised a way to exploit the redundant configuration of WSRT for calibration. They named this technique RSC (Noordam and De Bruyn, 1982), see above. RSC exploits the fact that two equal baselines observe the same visibility values. In practice, these values differ, which is caused by instrumental effects. Similar stories are told by Greisen, and Frazer Owen, who both applied self-cal-like techniques before this name was current. Owen claims that he 'showed a 4-antenna VLA self-cal to the trustees well before AIPS could do this'. In a review Pearson and Readhead (1984) summarised this and related techniques, and coined the term *self-calibration*.

GIPSY

In 1983 the Kapteyn Institute in Groningen, the Netherlands, released GIPSY, the Groningen Image Processing SYstem (Van der Hulst et al., 1992). The main objectives of the design of GIPSY were to 'have the ease of adding application programs, to have a flexible, self-descriptive, a multidimensional data structure, to make use of the de facto-standard X11 environment for graphics and image display and to code the system in ANSI C and the applications in either ANSI C or standard Fortran 77.' GIPSY was initially developed to process WSRT data but was later extended to process other instrument data as well.

DWARF

In early 1980, ASTRON launched the Dwingeloo-Westerbork Astronomical Reduction Facility (DWARF) (Hamaker, Harten, et al., 1985). Few astronomers had experience with indirect imaging or were familiar with the complex operations required to process data from WSRT. For this reason, it was decided that the WSRT data reduction would be executed by experts, using programs written mostly by Brouw. Data would be received at ASTRON, but then moved by train, on tape, to Leiden, where the maps would be produced. Below are the motivations quoted from the DWARF paper:

> A good environment should minimise the efforts required on the parts of the user and of the programmer to work with it. It should provide in the simplest possible way for all operations except those specific to individual applications. Such operations include:
>
> 1. Program-user interaction: The interface must cater for all modes of operation ranging from straight batch to highly interactive. It must free the user from all redundant typing. As much as possible, user input should be checked immediately for validity.
> 2. Creation and administration of data files.
> 3. I/O on a variety of data files, including those created by other systems.
> 4. Error reporting.
> 5. Record-keeping should minimise the need for manual administration.
> 6. Simultaneous execution from one terminal of more than one program, either interactively or in batch mode.
> 7. Programming: The interface for application programs should be as simple as possible, in terms of number of subroutine calls and number of arguments per call, as well as in terms of special declarations (e.g. of control arrays) to be made on behalf of the system. Not only does all this make programming per se easier; as a consequence it also promotes easy exchange of programs with other systems.
>
> We found all environments existing or planned a few years ago unsatisfactory in one or several of the above areas. In particular the user, data I/O and program interfaces fell short of what was wanted and judged to be feasible.

Noordam's theory is that Ron Ekers, then a professor at the University of Groningen, was frustrated by the limited involvement and freedom for the astronomer in the processing of radio data. While the astronomer had the option to supply various parameters, the operators would apply during image production, the process would sometimes take weeks. These delays made it very hard to adjust the parameters iteratively. This would later be of influence in the formation and design of AIPS and the surrounding infrastructure. AIPS gave astronomers the opportunity and freedom to build and run pipelines with manually modified parameters. We feel this is a very important sentiment, and its influence is clear to see in the development of radio astronomy software since then. Modern software often tries to give the users this freedom.

IRAF

In 1986, The 'Image Reduction and Analysis Facility (IRAF)' package was created. Although this package is mostly oriented towards optical astronomy, parts of it have been inherited by packages such as AIPS++ and Common Astronomy Software Applications (CASA).

Miriad

In 1988, the Berkeley Illinois Maryland Association (BIMA) had a debate with experts from the AIPS, GIPSY and IRAF camps, and decided that the existing tools were not flexible enough. Thus, Miriad was born. This software aimed at being a 'full service' radio interferometry data-reduction package. Miriad was initially designed for compact array millimetre radio astronomy but was later also used at lower frequencies. Despite its age, Miriad is still being developed and used. One of the difficulties with Miriad is a lack of versioning and a public and open software development process. The source code and binaries are published on the CSIRO FTP server, but apart from the upload timestamp, it is not trivial to work out the latest changes. This makes it hard to monitor the development process, track changes, and make Debian packages, as is described in Chapter 3. In 1995, a retrospective view paper was published (Sault et al., 1995), which gives an exciting view of the design considerations of Miriad.

1.8 Meanwhile in South Africa

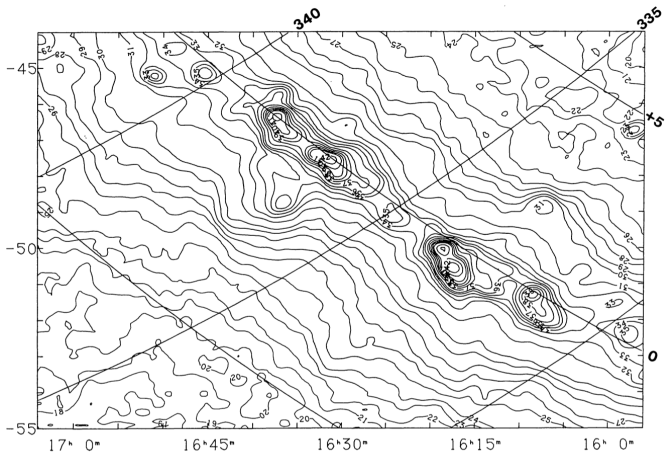


Figure 1.10: Contour map with a grid of galactic coordinates superimposed, image taken from Jonas et al. (1985)

Given the institution hosting this research, it would be amiss not include some South African context. Rhodes University was the first university in South Africa to establish a radio astronomy effort, with Jack Gledhill's work in the area of solar system astronomy in the 1950s. He was followed by such names as Eddie Baart, Graeme Poole, Gerhard de Jager and Gerhard Verschuur. Justin Jonas, effectively the father of the MeerKAT telescope, and now chief technologist at the South African Radio Astronomy Observatory

(as well as remaining a professor at Rhodes), has been with the university since his undergraduate studies in the late 1970s.

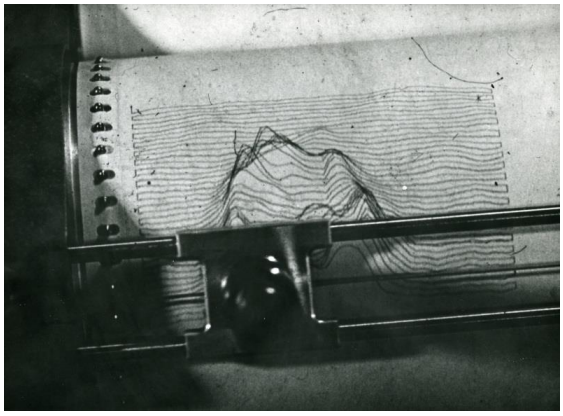


Figure 1.11: Photograph of the first plot made by the Edsac II computer of the radio image of the supernova remnant Cassiopeia A, observed by the One-Mile Telescope. The plot was made using a Calcomp drum plotter - one of the first commercial 'computer graphics' peripherals.

After reading an early draft of this work, Justin Jonas contributed the following account, which is reproduced verbatim and in full:

> [Fig 1.11] shows a Calcomp drum plotter in action – one of the first commercial 'computer graphics' peripherals. We inherited a large one from the Rhodes Computer Centre when the ICL mainframe was decommissioned. I interfaced it to the 16-bit parallel port on our VAX 11/730 using an 'open-source' Centronics printer driver for VMS, a Motorola 6800 evaluation board with some assembler code (this was the basis for my Honours computer interfacing course for many years, and I still have the 6800 board somewhere), and some random electronics to drive the plotter stepper motors and pen solenoid. I see now that what I reinvented then is called Bresenham's line algorithm (Bresenham, 1965).
>
> I wrote a number of plotting libraries that provided rudimentary plotting functions, and some higher-level functions like contour plotting. I also wrote emulators for some popular plotting libraries of the era - such as HPGL. Everybody was doing this re-invention - before the Internet there was little opportunity to share code, particularly in Apartheid era in South Africa.
>
> We used this system to plot early contour maps from the SKYMAP 2.3 GHz survey, but it was infuriatingly slow - a single map would take 8 hours to plot. And the liquid-ink drafting pen would often give up the ghost half-way through, requiring ultrasonic cleaning of the pen and a restart. Ball-point pens were more reliable, but the lines were too fine and "Astronomy and Astrophysics" would not accept the plots for publication.
>
> The figures for Jonas et al. (1985) (see Fig. 1.10 for an example) took more than a month to plot, and longer still for the maps in my MSc thesis. Students have it too easy today!

> When dot-matrix printers arrived I wrote a VAX assembler programme that emulated the functionality of the low-level Calcomp primitives, and modified this for laser printers when they appeared. I made a multi-pole switch to allow use of all three plotter options from the single VAX Centronics port.
>
> We ran Starlink on the VAX - thought it might have been mentioned, but I guess its use was largely limited to the UK and former colonies. Many of today's radio observatory directors were VAX/VMS/Starlink system administrators back in the day (Michael Garrett, Phil Diamond, etc). We ran our own SKYMAP single-dish analysis pipeline on top of the Starlink platform - this had many authors over time, starting with Pete Mountfort. We had the Fortran source code for Glynn Haslam's NOD 2 package, which was used for all of the Bonn surveys (Haslam 408 MHz, Reich 2700 MHz, etc), but it was notoriously impenetrable, and we used a different raster scanning pattern for SKYMAP. Starlink faded away with the advent of Linux on Intel '86 desktops, and I migrated our pipeline to Linux. I also discovered Pearson's PGPLOT, and all of my plotting legacy was binned! A real privilege to be working with Tim today in the C-BASS consortium.

1.9 1990s

The AIPS package moved to a *version control* system for the first time. Version control refers to methods for tracking and keeping a history of changes to text files. Nowadays, it is standard practice for software engineers to use revision control software to maintain documentation and source code and configuration files. As developers create software, it is common for multiple versions of the same software to be used in different situations while simultaneously receiving bug fixes and new features. These bugs and features of the software are frequently only present in specific versions. Therefore, to locate and fix bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs.

At the most primitive level, developers could retain multiple copies of the program's different versions and label them appropriately. Until roughly this time, this was how astronomers developed AIPS, and, presumably, other radio software packages. While this method can work, it is inefficient and error-prone, as many near-identical copies of the program have to be maintained. This approach requires a lot of self-discipline on the part of developers.

To automate this process and reduce human error, software to automate some or all of the revision control process was created. This software ensures that the majority of management of version control steps is hidden from the developer. Many version control software solutions have been introduced, and their popularity and usage have varied over time. The de-facto standard in software engineering is currently git, but mercurial, subversion and CVS, to name a few, are still used.

AIPS++

In 1988, Paul Vanden Bout, the director of the NRAO, called together an independent review panel: the Software Advisory Group (SWAG). This review panel later evolved into the international AIPS++ consortium (Croes, 1993). The group was chaired by

Tim Cornwell (CSIRO), who declined to be interviewed for this work. Other members included Noordam (ASTRON), Hunt (NRAO), Geoff Croes (DRAO), and Ray Norris (CSIRO). The goal was to set up a new universal software infrastructure that would serve the needs of all radio astronomy institutes around the world. AIPS++ started off using C++ and *Glish*. Glish is a scripting language developed by Vern Paxson, now a professor of computer science at the University of Berkeley. Unfortunately, this language never developed the required momentum to survive. AIPS++ eventually stopped using it, replacing it with a Python interface.

The consortium was initially an effort by the most significant radio astronomy institutes around the globe: the already mentioned ASTRON and NRAO, but also the Australia Telescope National Facility (ATNF now CSIRO), the Jodrell Bank Observatory (JBO), the MERLIN/VLBI National Facility (MERLIN/VLBI), BIMA and Tata Institute of Fundamental Research in India (TIFR). Although all these institutes managed their telescope(s), some did not write their processing software, but used AIPS or Miriad.

NEWSTAR

Brouw, at that time working at ASTRON, tried to get modifications into AIPS to improve support for WSRT. However, according to him, this code kept on disappearing for 'unknown reasons'. These disappearances could be blamed on the then missing, but now widely available and used, software project management tools such as Git and contribution workflows such as pull requests (Dabbish et al., 2012).

Around 1990, Brouw started working on the famous NEWSTAR package for processing WSRT data. In 1992, Brouw tried to get NEWSTAR working at CSIRO in Australia, but this turned out to be less trivial than initially thought. It was hard, at that time at least, to keep software universal enough to support multiple types of radio telescopes. He says that this attempt was eventually a failure.

The Measurement Set version 1

In October 1996, the Measurement Set (MS) version 1[4] standard was finalised. Although the astronomy field already had a storage standard (FITS), this format was not suitable for storing the massive tables with visibilities used in radio astronomy. A measurement set, or casacore table, is similar to a relational database table with the addition that table cells can contain n-dimensional arrays. It also has a rich SQL-like query language named Table Query Language (TaQL). Each table consists of numbered rows and named columns. A column can hold scalar values or arrays of any dimensionality and shape. Furthermore, the table and each column can hold a set of keywords, which, for example, can be used to define the units. A table is nestable; thus, a keyword's value can be a keyword set in itself. This MS format is flexible enough to contain a huge variety of shapes and sizes of data, and is widely used to transport data between programs and astronomical institutes.

[4]https://casacore.github.io/casacore-notes/191.pdf

The Measurement equation

Two orthogonal receptors dipoles are required to observe the polarisation of an electromagnetic signal. To describe these waves, the receptor's signals are correlated, resulting in four complex numbers, commonly referred to as XX, XY, YX, and YY (or RR, RL, LR, LL if circularly polarised receptors are used). Until that time, it was not well understood how to model and process the polarised characteristics of electromagnetic waves. Separate expressions and calculations were performed for each correlation combination, and each had to use approximations to make the calculations feasible. That changed when Hamaker realised that using the Kronecker matrix product would result in elegant expression without the need for approximations. In 1995, the measurement equation was introduced to the world of aperture synthesis imaging by Hamaker, Bregman, et al. (1996a). The mathematical foundations of the measurement equation are discussed in depth in Chapter 2.

DIFMAP

In 1994 Shepherd et al. (1994) created DIFMAP, written in C. DIFMAP is described as an 'interactive program for synthesis imaging' and performs all the interferometry calibration-loop steps in an interactive matter. Initially, it was a wrapper around all the steps in the Caltech VLBI programs (CITVLB)[5]. Pearson writes, 'Difmap has had a long life and is still used in some VLBI experiments'.

1.10 2000s

The Measurement Set version 2

A new version of the MS was created in early 2000[6]. The changes were mostly of a data-structuring nature (schematical). Van Diepen adds that changes were made in the underlying table code, specifically the `MultiFile` option, to merge multiple tables into one file. In addition, many improvements to the table query language (TaQL) were added.

MeqTrees, third-generation calibration algorithms

Around 2000 Noordam and Smirnov (2010) started working on MeqTrees. The original idea can be traced to 'MNS-trees' (Noordam, 1997). MeqTrees was arguably the first software package to fully utilise the powerful simplifications made possible by the measurement equation. It was also the first software to take direction-dependent effects (DDE) into account. The incorporation of DDE treatment gave rise to the third-generation calibration era (3GC).

Obit

Somewhere at the beginning of the 21st century, Cotton (2008) started Obit, a 'development environment for astronomical algorithms'. Cotton writes that Obit branched away from AIPS over time, and the code has a very different design from AIPS, although it

[5] https://www.cv.nrao.edu/adass/adassVI/shepherdm.html
[6] https://casa.nrao.edu/Memos/229.html

can invoke AIPS if needed. Greisen writes that this project is more research-oriented, while AIPS remains oriented to serving the majority of NRAO's telescope users. The limitations of Fortran77 drove Cotton's motivation to create Obit. Dropping Fortran77 support enabled the usage of third-party libraries for features such as threading and FFTs. Furthermore, the fork enabled Cotton to make serious breaking changes in a major production package. Obit also has a Python interface called Parseltongue (Kettenis et al., 2006) which interfaces to AIPS and is maintained by JIVE.

According to Cotton (2019, priv. comm.), Obit has a small but active user community: the (USA) Naval Research Laboratory's commensal 1-metre system (VLITE) on the VLA uses Obit for the astronomical calibration/imaging. Menton's MPIfR group is using Obit for the continuum imaging of their EVLA Galactic plane C-band survey. Lastly, the SARAO SDP group is using Obit in its MeerKAT calibration pipeline for spectroscopic data.

Cotton states that he intends to continue using and developing Obit for the foreseeable future, but there is no real long-term vision. The work on Obit is 'tolerated by NRAO but not exactly encouraged'.

CASA

In 2006 the AIPS++ consortium was disbanded. At the same time, NRAO needed a modern data-processing system to support the new ALMA telescope. Thus, AIPS++ was forked and CASA was born (McMullin et al., 2007a).

CASA is distributed as a self-contained and monolithic package providing tools oriented to data post-processing for modern radio telescopes. This monolithic nature makes CASA easy to get started with, but more awkward for the more advanced user. The package is bundled with its own Python interpreter, making it hard and often impossible to integrate with other software packages. Fortunately, NRAO recently changed the described architecture, and in December 2019 NRAO released a new CASA 6. This release is based on Python 3 and moves away from the monolithic design. This means CASA is now distributed as a binary wheel and can be installed in an existing Python environment, next to other radio astronomy utilities. Not all components, specifically some graphical user-interface parts, have been ported to the new design yet, so this release is for the time being oriented towards developers and power users.

Casacore

The core of the AIPS++ libraries is still being maintained and developed by a subset of the original consortium members and is now known as *Casacore*. A Boost-Python based Python wrapper named *python-casacore* also exists, which has mainly been developed by ATNF and ASTRON to replace Glish. Python-Casacore is not used by CASA. The project internally maintains an alternative set of Python bindings (CASApy). The most important component of Casacore is the measurement set accessor module.

1.11 2010s

Cuisine

In 2010, Renting (ASTRON) created Cuisine[7], a Python-based pipeline framework for building data reduction pipelines. Although LOFAR and the Transient Pipeline (TraP) (Swinbank et al., 2015) used the software for a while, there appear to be no projects remaining using this framework.

Docker

In 2013 a company named dotCloud released Docker, an OS-level virtualisation platform to deliver software in packages called containers. Although virtualisation was not new, the use of the recently introduced process isolation features to the Linux kernel made the product popular and accelerated adaptation. Docker caused an interdisciplinary shift in deployment strategies, from scientific pipelines to websites. Docker will be discussed in more detail in Chapter 4.

The ALMA pipeline

The Atacama Large Millimeter/submillimeter Array (ALMA) became operational in October 2011. The initial ALMA configuration consisted of 66 high-precision antennas and observed at wavelengths of 9.6 to 0.3 millimetres (31 to 1000 GHz). Of these 66 antennas, 54 antennas are 12 metres in diameter, and the remaining 12 antennas are 7 metres in diameter. ALMA was initially a split responsibility collaboration between the NRAO and European Southern Observatory (ESO). This collaboration was later extended with Japanese, Taiwanese, and Chilean partners. ALMA is the most expensive ground-based astronomical project, costing between 1.4 and 1.5 billion US dollars. This amount dates from approximately 2013, and has not been corrected for inflation.

In October 2014 the first ALMA pipeline based on CASA was released. Since this release, the CASA software itself is bundled with the full ALMA pipeline. The ALMA pipeline is renowned and praised in the radio astronomy community as being one of the first fully automated data-processing pipelines. Glendenning, Head of Data Management and Software Department at NRAO, cites multiple reasons for its success. First of all, the location of the telescope is ideal, high up and far away from interference caused by humans. Also, the observation scope is more limited than that of many other radio astronomy observatories, the field of view is small, and the dynamic range is generally low. However, implementing the data-processing took time; the pipeline was not a success from the start. It was only after five years that the pipeline effort started to be successful. Glendenning also says that NRAO sees the pipeline development primarily as a science operations organisational problem, and less as a software problem. There are currently less than five full-time equivalent (FTE) units working on the ALMA pipeline software, whereas there are more than 20 non-software FTE units involved. These positions include heuristics development, quality assurance evaluation and queue management.

Jeff Kern, the Project Director for Science Ready Data Products at NRAO, underlines this. He says that it took a while to get everyone to understand what is involved in really defining a heuristic. If it works on one data set, only a fraction of the work towards

[7] https://www.astron.nl/~renting/cuisine.html

creating a functional pipeline for another dataset is done. Kern also claims that the ALMA pipeline is successful because it constrains the problem space. ALMA is very prescriptive about how users may operate the telescope and limits the freedom to apply smart ideas. These restrictions are important for the pipeline. Data needs to be well structured, regular (the pattern should be the same for all observations), and of high quality. This change was a culture change for many of the radio astronomers. Kern explains: 'There was a period where the operations group would throw an extra antenna on a long baseline into the array because more data is always better, and it cannot hurt. However, this made the pipeline almost unusable for these data sets (there was an objection to just throwing the extra antenna away), so they ended up going through a manual process until the culture changed. I still hear occasional grumblings about how ALMA prevents people from doing clever things in the Observation tool (Bridger et al., 2012), I think that is part of why the pipeline succeeded.'

Meanwhile, around 2014 CASA developers switched back to using the upstream Casacore again, improving the collaboration and reducing duplicate efforts between ASTRON, NRAO, SARAO and other involved parties.

Common Workflow Language

In 2014 the Common Workflow Language (CWL) project, which is covered in Chapter 5, published its first release. CWL can be seen as the HTML of pipelines, enabling pipeline creators to define a pipeline in an abstract domain-specific language. In June 2019 the 1.1 version of the specification was released, including improvements to the handling of writable directories, which are important for directory-based datasets such as the MS used in radio astronomy, to which the present work contributed. CWL is a wrapper around the command line interface of existing software, and does not require any modifications to the original software, making it ideal for wrapping legacy software and chaining these together in high-level abstractions.

DDFacet and killMS

Around 2015 Tasse et al. (2018) started realising their 3CG ideas into a new product called DDFacet. DDFacet is a wide-band wide-field spectral deconvolution framework based on image plane faceting, that takes into account generic DDE and is based on the RIME. The development was primarily driven by the need to address the ionospheric effects for the LOFAR telescope. The purpose of faceting is to approximate a wider field of view with multiple small narrow-field images. The companion killMS (Smirnov and Tasse, 2015; Tasse, 2014a; Tasse, 2014b) package performs direction-dependent calibration on individual tessels (tessels are collections of facets), while DDFacet can apply these solutions during imaging.

Kliko

At the beginning of 2016, unaware of the existence of CommonWL, the author of this thesis started working on a container-based pipeline abstraction framework, which is discussed in detail in Chapter 4. Since over time it became clear Kliko has a significant overlap in design and functionality with CommonWL, while the latter has a solid

adaptation base, it was decided to discontinue development and maintenance of Kliko. [8]

Stimela

Meanwhile, Sphesihle Makhathini (Rhodes University) was inspired by Kliko and decided to create a similar product called Stimela[9]. Stimela is quite similar to Kliko and inherits various design decisions. The most significant difference between Stimela and Kliko is that the former is less strict about handling IO purity and dealing with functional side-effects. Kliko is, in that sense, similar to CommonWL, which by default only allows modifications on copies of the intermediate data products, ensuring that the original data is not affected. That makes Stimela more flexible when data consistency and reproducibility are less of a worry, or when the data volumes are so massive that storing copies is not feasible. Moreover, Stimela scripts can be transpiled to CommonWL projects, a feature used in Section 5.4. Stimela is oriented to radio astronomy only and is bundled with scripts for various tasks (e.g., CASA, MeqTrees, AOflagger, SoFiA, etc.) from various data reduction packages.

CARACal (formerly known as MeerKATHI)

CARACal (Józsa et al., 2020a) is a relatively new effort (its first public release was in May 2020) to make a pipeline for radio interferometry continuum and spectral line data in full polarisation. It is based on Stimela. While mostly in use with MeerKAT data, as the original name suggests, the pipeline is actually evolving to become telescope-independent, and a few astronomers (though mostly from the core development team) have already applied it to data from VLA, LOFAR, ASKAP and uGMRT. Since CARACal is based on Stimela, it runs on any platform supporting containerisation technology. CARACal is actively developed, mostly by astronomers at SARAO, Rhodes University, and Osservatorio Astronomico di Cagliari INAF (OAC). It offers support for all modern imaging and calibration tools, including 3GC.

Default pre-processing pipeline

The default pre-processing pipeline (DPPP or DP3) (van Diepen and Dijkema, 2018) is a streaming processing pipeline for radio interferometric data, mostly used for preprocessing data coming from the LOFAR telescope. DPPP's functionality includes averaging, flagging and various kinds of calibration techniques. DPPP goes through visibilities in temporal order and can perform standard operations such as averaging, phase-shifting and flagging misbehaving stations. The data is not written to disk between the steps in a pipeline, making this tool suitable for operations where I/O dominates. DPPP has a plug-in architecture; other computing steps can be provided by loading a shared library. AOFlagger (Offringa, 2010) is one of the packages made available as a plug-in

[8] Initially, the author was disappointed about the waste of time spent on developing a product that already existed. In addition, he was disappointed in himself for not having put more effort into researching existing solution. However, over time he became proud of the similarities between Kliko and CommonWL, feeling reinforced about various design decisions taken while creating Kliko. In the end, the CommonWL team took well-considered decisions, which resulted in a cleaner design. Also, the CommonWL community is much more matured, leading to the decision to stall the Kliko development, and focus all efforts on CommonWL.

[9] https://github.com/SpheMakh/Stimela

to DPPP using the described plug-in mechanism. The framework also contains a bridge that enables loading Python steps. Consequently, and contrary to CWL, a DPPP plug-in requires significantly more effort to create. Where a CWL step would only involve the creation of a YAML text file, a DPPP plug-in requires writing and compiling a C(++) intermediate layer. These are skills not every astronomer or even programmer possesses. On the other hand, the memory-based transfer of intermediate data product makes DPPP much more suitable for high volumes of data, which are typical in radio astronomy.

The LOFAR two-metre survey pipeline

The LOFAR two-metre sky survey (LoTSS) (Shimwell et al., 2017) is an ongoing high-resolution, high-sensitivity survey of the northern sky in the low $120 - 168$ MHz range. The survey is currently about 20% complete. The total raw data volume of this survey will hold an impressive amount of about 50 PetaBytes, and a total of 13 000 hours of observation time is required. The LoTSS data is stored in the LOFAR long-term archive (LTA)[10], which is distributed over three sites in Europe.

The processing consists of the two steps, a direction-independent effect (DIE) pre-processing step and a DDE step. The DIE calibration step (Van Weeren et al., 2016) makes use of the DPPP in combination with BlackBoard self-cal (Pandey et al., 2009) for averaging and calibration. The DIE pipeline has been mostly automated (Mechev et al., 2018), and the processing is performed on the SURFsara grid facilities. To facilitate the processing, custom tools have been created, which are based on PiCaS[11]. PiCaS is an in-house developed CouchDB-based token pool server for heterogeneous compute infrastructure. Unfortunately, the data is spread out over the three LTA sites, while only one site has a direct fast link to the SURFsara grid facilities. Transporting the data between sites is slow and expensive, so it is more cost- and time-effective to process the data at the storage site. Unfortunately, the DIE pipeline has been tailored for the SURFsara grid environment. Consequently, the current processing software needs to be modified to run on the infrastructure available locally at the storage sites. The DDE calibration pipeline has been handcrafted with Python[12] and makes use of KillMS and DDFacet. This pipeline is manually run on a local cluster.

[10] https://lta.lofar.eu/
[11] http://doc.grid.surfsara.nl/en/latest/Pages/Practices/picas/picas_overview.html
[12] https://github.com/mhardcastle/ddf-pipeline

1.12 Overview of events

Table 1.1: A brief timeline of radio astronomy and data reduction pipelines

Year	Who / Where	Event
1931	Karl Jansky	Discovers radiation in the radio spectrum coming from the Milky Way.
1938	Grote Reber	Produces first radio map of Cygnus A and Cassiopeia A.
1944	Hendrik van de Hulst	Predicts HI line.
1946	Martin Ryle & Derek Vonberg, Cambridge	First to apply interferometry to astronomical measurements at radio wavelengths.
1950	Martin Ryle, Cambridge	Describes the use of the Earth's rotation to improve the (u, v) coverage.
1951	Harold Ewen & Edward Purcell, Harvard University	HI line first detected.
1956	NRAO	The National Radio Astronomy Observatory is founded.
1966	WSRT	Construction of the Westerbork Synthesis Radio Telescope begins.
1970	WSRT	Westerbork telescope construction completed.
1973	NRAO	Construction of the VLA begins.
1974	Robert Hjellming, NRAO	CANDID, the first data-processing software for the VLA is created.
1974	Jan Högbom	The CLEAN algorithm is formulated.
1975	NRAO	The first VLA antenna is put into place.
1978	Charlottesville, Virginia	Development of the AIPS software project begins.
1979	Ron Harten, ASTRON	FITS format is created.
1979	Johan Hamaker, ASTRON	Introduces Kneading.
1980	NRAO	VLA is officially inaugurated.
1980	ASTRON	DWARF project is initiated.
1980	NRAO	ISIS pipeline for VLA is created.
1981	Tim Cornwell & Peter Wilkinson	Self-cal is developed, ushering in the 2GC era.
1983	Kapteyn Institute	GIPSY software for processing WSRT data is created.
1985	NRAO	First public release of AIPS.
1986	NOAO	The 'Image Reduction and Analysis Facility (IRAF)' package is created.
1988	BIMA	The MIRIAD package is created.
1988		The SWAG, later AIPS++ consortium is formed.
1990	NRAO	The AIPS software package moves to a version control system.
1990	Wim Brouw, ASTRON	The NEWSTAR software project is created.
1994	Martin Shepherd	DIFMAP is created, oriented towards Very Large Baseline Interferometry specifically.
1995	Johan Hamaker & Jaap Bregman & Bob Sault	The Measurement Equation is developed.
1996		Measurement Set format version 1 is formalised.

Continued on next page

Table 1.1 – continued from previous page

Year	Who / Where	Event
2000	Jan Noordam & Oleg Smirnov, ASTRON	The incorporation of DDE treatment in calibration gives rise to the 3GC era.
2006		AIPS++ consortium is disbanded, AIPS++ software forked into CASA.
2008	Bill Cotton, NRAO	OBIT is forked from AIPS, enabling more research-oriented development.
2009	ASTRON	The default pre-processing pipeline (DP3), a streaming processing pipeline for radio interferometric data, is created.
2011	NRAO	The Atacama Large Millimeter/submillimeter Array (ALMA) becomes operational.
2012	ASTRON	The LOFAR telescope is officially inaugurated.
2013		Docker is released, introducing containerised deployment to a wider audience.
2014	NRAO	The first ALMA pipeline based on CASA is released.
2014	NRAO	CASA switches back to using casacore.
2014		The Common Workflow Language (CWL) project is initiated.
2015	Cyril Tasse & Oleg Smirnov, OPM & SKA-SA	DDFacet and killMS are introduced.
2016	Gijs Molenaar, SKA-SA	The Kliko containerised pipeline construction framework is created.
2016	Sphesihle Makhathini, SKA-SA	The Stimela framework and Caracal pipeline is created.
2018	SARAO	The MeerKAT telescope is officially inaugurated.

1.13 Discussion

> Programming is the process of sequential framing and solution of problems.
>
> Mikhail Donskoy

The supervisor of this thesis first heard this profundity (or perhaps platitude) from the developer of the inaugural world computer chess champion (Berliner, 1978), who presumably knew a thing or two about solving problems. Looking at the history of radio astronomy software, no problem seems too difficult. From Gower battling a combustible Edsac II, to Tasse struggling with an uncooperative ionosphere, there is a golden thread running through the field of talented individuals and teams coming up with solutions to seemingly insurmountable problems, and dealing with experiments, surveys and telescopes successfully. On the other hand, while these solutions are never (or hardly ever) secret, only certain kinds of solutions seem to be successfully shared. The author himself, being a software engineer brought up in the era and ethos of open-source software, is naturally interested in the process of solving *other* people's problems through the sharing and interoperability of software.

It is interesting to see where this process has succeeded and failed. Firstly, one needs to discount cases where sharing of software is not intended or attempted because the software was never meant to solve other people's problems in the first place. There is no historical record of the early Cambridge software being used anywhere else, but presumably, no other group had an Edsac II to run it on anyway. To take a more recent example, the LoTSS pipeline is highly survey-specific: it is built, maintained and operated by a core team, it is not being used with other LOFAR data or by external users, nor is it in the team's remit to support this (that said, one should recognise that a significant part of the LoTSS development has fed back into its constituent packages, in particular DDFacet and killMS, which *are* widely used). We will only consider user-facing software – and, in particular, user-facing pipelines. User-facing software in this context refers to software that is being used by astronomers, and not for the lower-level (and often telescope-specific) operations of the telescope.

Many other instances are documented here where software *was* built to be user-facing. Imagine a two-dimensional phase space with use cases (i.e. telescopes and observational regimes) loosely lined up along the X-axis and individual processing tasks along the Y-axis. A package such as WSClean or DDFacet would have a footprint that is very broad but shallow: successfully used across many telescopes, but restricted to one specific task – imaging. NEWSTAR would be very tall and thin: only supporting WSRT data, but implementing most tasks. Monolithic packages such as CASA and AIPS and Miriad would form large, patchy clouds: supporting many telescopes and most tasks, but with the occasional yawning gap in functionality. A complete pipeline would be a tall, unbroken column stretching from the top to the bottom. At present, there is only one successful example of this – the CASA-based ALMA pipeline – a very thin and tall column in the phase space.

The ALMA example is instructive. As discussed above, success was achieved by applying a ruthlessly narrow focus, with a large and relatively centralised team, including over 20 non-software FTEs. Although the resulting software itself is fully open-source,

the development process is nearly the polar opposite of the open-source 'bazaar' model. Perhaps this is the only way to write successful pipelines, and more general pipelines are impossible?

At the same time, if one shoud pick any telescope/use case along the X-axis, and draw an imaginary vertical line up, more often than not one would be drawing a line through an unbroken sequence of packages providing all the required tasks. In fact, many users can perform virtually all their processing without leaving the confines of CASA (and earlier, AIPS), just not in a pipelined and automatic way. Many tasks will also have multiple packages available for the job. Thanks to standards such as FITS and the MS, these packages are likely to speak the same formats. Moreover, one would also find that the sequence of tasks is the same for many use cases. All the functionality is there, so it seems a little paradoxical that there have been so few successful user-facing pipelines.

One aspect of the problem is that radio interferometry itself has been described as death by a million papercuts. Every telescope and every observation is slightly different in annoying little detail. Kern and Glendenning's remarks on the ALMA pipeline allude to some of this difficulty and show that an army of 20 FTEs is one way of developing the necessary pipeline heuristics to stave off such a death.

A second aspect is that the mere process of software installation (not even considering interoperability) has often been death by *another* million papercuts. Software design and engineering are easy to get into but hard to do well, and authors of astronomy software tend to be astronomers, rather than professional software engineers. As a result, widely used packages have different and confusing build systems and a tangled (and sometimes contradictory) web of dependencies. Just getting all the software components built and running on one machine has, in the past, been a major job. Reproducing the same setup on a different system would often start from square zero. Moreover, all of this has to happen before one can even starts thinking about pipeline heuristics!

It is this second aspect of the problem to which the author of the present work hopes to make a modest contribution. If the existing software pieces were easier to put together, perhaps more astronomers' time would be available to solve the truly interesting parts of the pipeline problem[13].

[13]One should note that, of the recent software efforts described above, CARACal is the first attempt at a user-facing pipeline that extensively uses the technologies and ideas developed in the course of this work. It will be interesting to see whether it proves any more successful than previous efforts!

Chapter 2

Fundamentals

The aim of this chapter is to introduce some of the key concepts and the notation encountered in radio astronomy. These aspects are addressed informally in the chapter and we refer to Hamaker, Bregman, et al. (1996a), Smirnov (2011a), and Thompson et al. (2017) for full details.

2.1 Electromagnetic radiation

Astronomy is the scientific field that studies celestial objects and phenomena. Radio astronomy focuses on objects that emit electromagnetic energy at lower (radio) frequencies. As radio waves cannot be observed with the human eye, it was only in the 20th century that it was realised that there is a wealth of study material in the lower frequency electromagnetic emission originating from outside the atmosphere. It is not surprising it took humans so long to discover these emissions. Before we can construct an image of the radio sky, a tremendous amount of engineering, computation, mathematical insight and precision is required.

Radio telescopes can observe celestial objects from within the earth's atmosphere between the frequencies of approximately 10 Mhz (30 m) and 1 THz (0.3 mm). At higher frequencies, we enter the infrared domain. The absorption of infrared radio waves by molecules such as water vapor and CO_2 makes the atmosphere opaque to telescopes operating at frequencies above approximately 1 THz. Optical astronomy, for example, is more affected by atmospheric absorption than most telescopes operating in the radio window. As a result, optical telescopes need to be constructed at higher altitudes where the atmosphere is less distorting. On the other hand, at much lower frequencies, the ionosphere is troublesome, since it contains free electrons that distort radiation coming from outside the atmosphere. Fortunately, within the 10 Mhz - 1 THz window, observations are less sensitive to atmospheric distortions. This means that, for example, a useful radio telescope such as LOFAR can also be built at sea level in a country like the Netherlands, while a useful optical telescope cannot, or not without major limitations. It is also significantly cheaper to construct and easier to operate than a telescope in high-altitude, remote locations. In addition, the window of frequencies that is relatively unobstructed by the atmosphere is much wider for radio telescopes.

To understand how a radio map is created, it is crucial to understand what a radio telescope measures. Astronomical observations all aim to infer the physical properties of celestial objects.

Consider the situation in Fig. 2.1, which depicts the observation of radio emission from an astronomical source. This can be described as:

$$\mathrm{d}P = L_\nu \cos(\theta)\mathrm{d}\Omega\mathrm{d}A\mathrm{d}\nu, \tag{2.1}$$

where $\mathrm{d}P$ is the power (W) intercepted by the surface area $\mathrm{d}A$ (m^2) of the source element $\mathrm{d}\Omega$ (sr) in the bandwidth $\mathrm{d}\nu$ (Hz). Note that these equations are only valid if the source of emission is sufficiently far away that the emission arrives at the receivers as plane waves.

The specific brightness L_ν is then defined as:

$$L_\nu = \frac{\mathrm{d}P}{\cos(\theta)\mathrm{d}\Omega\mathrm{d}A\mathrm{d}\nu}, \tag{2.2}$$

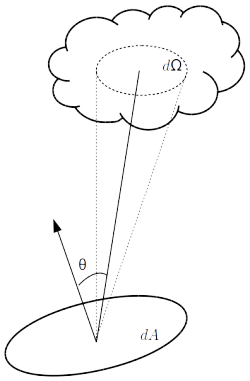


Figure 2.1: Relationship between solid angle $\mathrm{d}\Omega$, surface area $\mathrm{d}A$ and θ, the angle between the line normal to $\mathrm{d}A$ and the centre of $\mathrm{d}\Omega$. Image taken from Rohlfs and Wilson (2013).

Emission	Frequency	Description
1.6 KJy	1.4 GHz	Cygnus A, the closest extragalactic radio source
2 KJy	10 GHz	The Milky Way
1.6 MJy	1.4 GHz	The sun
4 MJy	10 GHz	The sun
110 MJy	1.8 GHz	Radio-frequency interference from a GSM telephone transmitting 0.5 W at a distance of 1 km

Table 2.1: A list of sources, their emission at the indicated frequency, as in Kraus (1986).

in units of $\mathrm{W\,m^{-2}\,Hz^{-1}\,sr^{-1}}$. Integrating over the solid angle we obtain the flux density (S_ν):

$$S_\nu = \int_\Omega L_\nu \cos(\theta) \mathrm{d}\Omega \tag{2.3}$$

in units of $\mathrm{W\,m^{-2}\,Hz^{-1}}$. Since radio signals from astronomic objects are usually extremely weak, radio astronomers commonly measure the flux density in units of Jansky (Jy) defined as

$$1\,\mathrm{Jy} \;=\; 10^{-26}\,\mathrm{W\,m^{-2}\,Hz^{-1}}. \tag{2.4}$$

Table 2.1 lists some well-known sources and their flux densities. The last row in the table clearly shows that artificial sources can be orders of magnitude stronger than natural sources. Such unwanted interference is commonly referred to as radio frequency interference (RFI) and has to be removed from the data, a topic that we will not elaborate on further in this thesis.

Astronomers often refer to the *brightness temperature* of a source even when the radiation is not of thermal origin (e.g. synchrotron radiation) (Thompson et al., 2017).

This assumes that the emitter is a black body operating in the regime where Rayleigh-Jeans approximation holds, i.e. $h\nu \ll k_b T$, where h is the Planck constant, k_b is the Boltzmann constant and T is the temperature of the black body. In this regime, the brightness (specific intensity) is:

$$L_\nu = \frac{2\nu^2 k_b T}{c^2}, \tag{2.5}$$

where it should be noted that L_ν does not necessarily have a thermal origin. The source temperature T_{src} is defined as the brightness temperature associated with the power received by the telescope from the source under observation:

$$\begin{aligned} P_\nu &= \int_A \int_\Omega L_\nu dA d\Omega & (2.6)\\ &= \int_A \int_\Omega \frac{2kT_{src}}{\lambda^2} dA d\Omega & (2.7)\\ &= 2kT_{src}. & (2.8) \end{aligned}$$

The last step can be made using the antenna theorem (Siegman, 1966), which states that $dAd\Omega = \lambda^2$. Once a link has been made between the power P_ν received by the antenna at a given frequency, the source temperature T_{src}, and the observed intensity L_ν, we can proceed with the conversion between flux density and brightness temperature for an unresolved source:

$$\begin{aligned} P_\nu &= L_\nu A_e \Omega_a & (2.9)\\ 2kT_A &= S_\nu A_e & (2.10)\\ T_A &= \left(\frac{A_e}{2k}\right) S_\nu, & (2.11)\\ & & (2.12) \end{aligned}$$

where A_e is the 'effective area'. A_e is described as $A_e = \eta_a A_p$, where η_a is the aperture efficiency and A_p is the projected area of the telescope. The conversion between K and Jy, also known as the 'forward gain' of an antenna, now becomes:

$$K/Jy = \frac{A_e}{2k}. \tag{2.13}$$

In effect, brightness temperature is just another measure to describe the brightness of a source. The source temperature T_{src} describes the energy received from the source we are interested in; the system temperature T_{sys} describes the actual power received as a function of both the sky (T_{sky}) and the receivers (T_{Rx}):

$$T_{sys} = T_{sky} + T_{Rx}. \tag{2.14}$$

The dominant component is the receiver temperature, T_{Rx}, which comes from the thermal noise from the receiver electronics. Depending on the observation frequency, receivers can be cooled to reduce T_{Rx}. T_{sky} describes the power generated by everything in the sky that we do not want to observe. This includes background sources, water vapor in the atmosphere, galactic backgrounds and more. In order to detect the target source,

we need $T_{src} > T_{rms}$, where the observation-dependent quantity T_{rms} is the noise in our measurement of the observation-independent quantity T_{sys}:

$$T_{rms} = \frac{T_{sys}}{\sqrt{N}}, \tag{2.15}$$

where N is the number of independent data points. For a telescope operating in the radio regime, the number of independent samples is equal to $\Delta\nu \cdot \tau$, where $\Delta\nu$ is the bandwidth (in Hz) and τ is the integration time in seconds (Thompson et al., 2017). This gives us:

$$T_{rms} = \frac{T_{sys}}{\sqrt{\tau\Delta\nu}}. \tag{2.16}$$

Now we can write down the *the radiometer equation*, an expression for the signal-to-noise ratio:

$$\frac{S}{N} = \frac{T_{src}}{T_{rms}} = \frac{T_{src}}{T_{sys}}\sqrt{\tau\Delta\nu}. \tag{2.17}$$

A typical value for $\Delta\nu$ is 10 MHz and for τ it is 1 sec, which makes $\sqrt{\tau\Delta\nu} \sim 3 \times 10^3$. T_{sys} is usually between $40 - 200K$. Now, the *system equivalent flux density* (SEFD) is the flux density equivalent of T_{sys}:

$$\text{SEFD} = \frac{T_{sys}}{(K/Jy)} = \frac{T_{sys}}{A_e/2k} = \frac{2kT_{sys}}{A_e}. \tag{2.18}$$

The SEFD is useful for comparing the sensitivity of two different systems, as it contains both T_{sys} and A_e. Furthermore, the sensitivity calculation is simplified, since if the flux of the observed source and the SEFD are both known, then the required integration time to make a given S/N detection is calculated as follows:

$$\frac{S}{N} = \frac{S_\nu(Jy)}{\text{SEFD}}\sqrt{\tau\Delta\nu}. \tag{2.19}$$

If we substitute for S in the temperature-based radiometer equation the following expression for the RMS variations in flux density $S_{\nu,rms}$ is produced:

$$S_{\nu,rms} = \frac{\text{SEFD}}{\sqrt{\tau\Delta\nu}}. \tag{2.20}$$

In the temperature-based radiometer equation, signal-to-noise will go up with increased $\Delta\nu$ and τ, while in the flux-density-based radiometer equation, RMS flux density variations will go down with increased $\Delta\nu$ and τ.

The use of temperature in previous equations is convenient because it provides the means to relate signals to the actual power received by a telescope. Before going into these details, we next consider how radio signal are measured in practice.

2.2 Interferometry

The diffraction limit places an upper bound on the maximum amount of detail that can be distinguished when making electromagnetic observations. This limit, the so-called angular resolution θ (in radians) of a telescope, is given approximately by:

$$\theta \approx 1.22\frac{\lambda}{D}, \tag{2.21}$$

where λ is the observed wavelength and D is the diameter of the aperture. The 1.22 coefficient is based on the intensity of diffracted light from a circular aperture (in the Fraunhofer limit), which is derived using a Bessel function of the first kind, of order one. The constant may be different for different aperture shapes, but the $\frac{\lambda}{D}$ relationship remains the same.

To distinguish two points, they need to be separated by an angle larger than θ. For example, at an optical wavelength of 500 nm, this means a three-metre dish would be able to distinguish sources separated by approximately 2×10^{-7} rad (≈ 0.05 arcseconds). To reach the same resolution when observing neutral hydrogen H1 emission ($\lambda \approx 0.021\,\mathrm{m}$) we would need a dish with a diameter of more than 100 kilometers. The radio telescope with the largest diameter ever constructed is the 500-metre aperture spherical telescope (FAST) (Peng et al., 2001). A telescope of this scale is already extremely complex to build, comes at great financial costs and has a significant negative ecological impact. Building larger telescopes on earth is impractical and eventually becomes impossible as we near the structural limits imposed by material science. Fortunately, it is possible to synthesise a larger telescope from multiple smaller telescopes using a technique called interferometry.

Aperture synthesis

A radio interferometer is an array of radio antennas that are used simultaneously to simulate a discretely-sampled single telescope of very large aperture. The maximum angular resolution of such a telescope depends on the distance between the two antennas with the largest separation. Each pair of antennas in the array is called a baseline. The number of unique baselines N_b for an array of N antennas is $N_b = (N^2 - N)/2$, excluding auto-correlations. The angular resolution of an interferometer is defined by:

$$\theta \approx 1.22\frac{\lambda}{B_{\max}}, \tag{2.22}$$

where $B_{\max}$ is the maximum baseline.

Let us consider the simplified case where we have only two antennas p and q, depicted in Fig. 2.2. The antennas are at positions $\mathbf{r}_p$ and $\mathbf{r}_q$ respectively. An astronomical source emits emission $\mathbf{e}$ from direction $\mathbf{s}$, which arrives at the two antennas separated by a separation vector or baseline

$$\mathbf{b} = \mathbf{r}_p - \mathbf{r}_q. \tag{2.23}$$

As mentioned before, the source is sufficiently distant so we can safely assume the emission arrives at the antennas as plane waves. This means when a wavefront arrives at $\mathbf{r}_q$ at time t, the same wavefront will arrive at $\mathbf{r}_p$ at time $t + \tau_g$. The geometric delay, τ_g, resulting from the difference in path length traversed by the signal, can be computed with

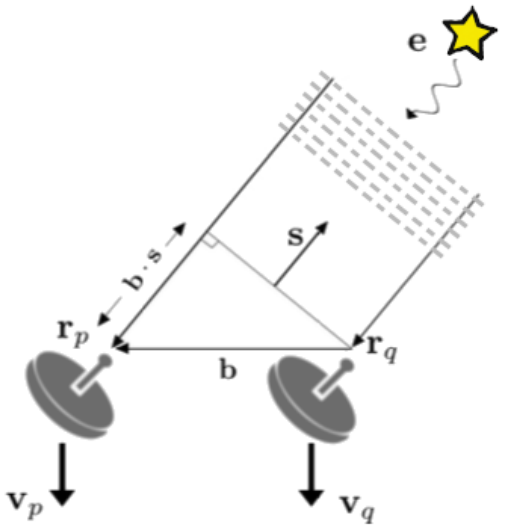


Figure 2.2: A basic interferometer with two antennas

$$\tau_g = \mathbf{b} \cdot \mathbf{s}/c, \tag{2.24}$$

where c is the speed of light in a vacuum (2.99×10^8 ms^{-1}) and $\mathbf{s}$ is a unit vector directed at the source. Thus it is possible to compensate for the difference in path length traversed by the signal by artificially retarding the signal at $\mathbf{r}_q$ by τ_g. To see how this is done we need to first look more closely at what is actually measured by an interferometer.

Interferometers actually only have access to the voltage induced across the feeds of its receivers. The signal, an electric field $\boldsymbol{e}$, is an incident plane wave and so can be described by a 2 dimensional complex vector (see Hamaker, Bregman, et al. (1996b) and Smirnov (2011a) for example). When using an orthonormal xyz coordinate system with z along the direction of propagating, $\boldsymbol{e}$ can be represented as

$$\boldsymbol{e} = \begin{bmatrix} e_x \\ e_y \end{bmatrix}, \tag{2.25}$$

where each of e_x and e_y are complex numbers representing the amplitude and phase of the incident wave in these respective directions. The voltage, $\boldsymbol{v}$, induced across the feeds will be related to $\boldsymbol{e}$ via a 2×2 (R. C. Jones, 1941), $\boldsymbol{J}$, so that

$$\boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \boldsymbol{J}\boldsymbol{e}. \tag{2.26}$$

For the sake of simplicity here, we consider only a single component of this vector $v = v_x$. Now, we can write the voltage at antennas p and q, v_p and v_q respectively, as

$$v_p(\boldsymbol{r}_p, \nu, t_p) = \epsilon_p(\boldsymbol{s}, \nu, t_p) e^{-\imath\omega t_p}, \tag{2.27}$$

and

$$v_q(\boldsymbol{r}_q, \nu, t_q) = \epsilon_q(\boldsymbol{s}, \nu, t_q) e^{-\imath\omega t_q}, \tag{2.28}$$

where ϵ is the amplitude, $\phi = \omega t$ is the phase of a plane wave with angular frequency ω. If we now form the correlation between v_p and v_q^* we find

$$v_{pq} = \langle v_p, v_q^* \rangle = \langle \epsilon_p \epsilon_q \rangle e^{-\imath\omega\tau_g}, \tag{2.29}$$

where angle brackets denote averaging over time, a superscript * denotes the complex conjugate and we have used the fact that $t_q = t_p - \tau_g$. Thus it is evident that, given sufficient information about the Jones matrices J_p and J_q, it is possible to recover information about the signal from the quantity v_{pq}. This is the essence behind aperture synthesis. By placing multiple antennas at different locations we can measure correlations between the induced voltages across distances much larger than the diameter of a single dish. If the feeds are ideal (i.e. $J_p = J_q = I$) and the source amplitude does not change over time (i.e. $\epsilon_p = \epsilon_q = A$), we simply recover

$$v_{pq} = A^2 e^{-\imath\omega\tau_g}, \tag{2.30}$$

which is a complex sinusoid as a function of τ_g, which depends on the relative positions between antennas p and q. This is the simplest form of a visibility function. It describes what an ideal pair of antennas located $\boldsymbol{b}$ apart would measure if there was only a single source in direction $\boldsymbol{s}$. We now need a convenient coordinate system in which to express $\boldsymbol{b}$ and $\boldsymbol{s}$.

The (u, v, w) coordinate system

In order to relate the visibility function Eq. (2.29) to the properties of the source we first need to introduce a coordinate system. A convenient way to do this is to introduce a Cartesian coordinate system XYZ relative to standard equatorial coordinates (H, δ) where H denotes hour angle and δ declination (Thompson et al., 2017). Using the convention that X points towards $(0^h, 0)$, Y due east towards $(-6^h, 0)$ and Z towards the celestial north pole ($\delta = 0$), we can then define the components of any baseline vector in the XYZ coordinate system, i.e. $\boldsymbol{b} = (b_x, b_y, b_z)$. Now suppose that the interferometer points towards the reference direction $\boldsymbol{s}_0 = (H_0, \delta_0)$. Then, defining (u, v, w) coordinates such that

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} \sin H_0 & \cos H_0 & 0 \\ -\sin\delta_0 \cos H_0 & \sin\delta_0 \cos H_0 & \cos\delta_0 \\ \cos\delta_0 \cos H_0 & -\cos\delta_0 \sin H_0 & \sin\delta_0 \end{bmatrix} \begin{bmatrix} b_x/\lambda \\ b_y/\lambda \\ b_z/\lambda \end{bmatrix}, \tag{2.31}$$

defines a plane relative to $\boldsymbol{s}_0$ where u and v span the plane and w is orthogonal to it. As illustrated in Figure 2.3, we can now specify an arbitrary position on the celestial sphere relative to $\boldsymbol{s}_0$, $\boldsymbol{s}' = \boldsymbol{s} - \boldsymbol{s}_0$ say, using the direction cosines $(l, m, n = \sqrt{1 - l^2 - m^2})$ of (u, v, w) i.e.

$$\boldsymbol{s}' = \boldsymbol{s} - \boldsymbol{s}_0, \quad \text{where} \quad \boldsymbol{s} = \begin{bmatrix} l \\ m \\ n \end{bmatrix}, \quad \boldsymbol{s}_0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.32}$$

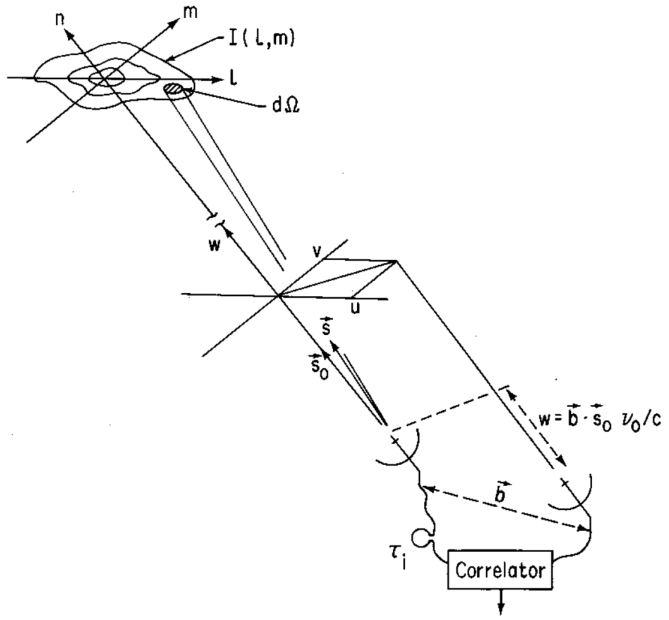


Figure 2.3: The (l, m, n) and (u, v, w) coordinate system used to express source brightness distribution and interferometer baselines, respectively. Image taken from G. B. Taylor et al. (1999).

Then, given the relative coordinates of an arbitrary pair of antennas as $\boldsymbol{b}_{pq} = (u_{pq}, v_{pq}, w_{pq})$, we can rewrite Eq. (2.30) for an arbitrarily located pair of antennas and (single) point in the sky as

$$v(u_{pq}, v_{pq}, w_{pq}) = Ie^{-2\pi\imath\frac{\nu}{c}(u_{pq}l+v_{pq}m+w_{pq}(n-1))}, \tag{2.33}$$

where we have introduced the positive quantity $I = A^2$, a property of the source which we have assumed to be constant in time.

Thus far we have over-simplified the discussion. In practice, antenna feeds are not ideal so it is, in general, not possible to consider only a single component of $\boldsymbol{v}$. In addition, Jones matrices describing transformations of the signal as it goes from source to observer may also vary over time and frequency. In practice, this means the averaging interval in Eq. (2.29) has to be small enough so that the Jones matrices are approximately constant across it. Furthermore, there are actually multiple sources contributing to the visibilities and these have to be combined in a meaningful way. All these effects can be accounted for using the Radio interferometer measurement equation (RIME) (Hamaker, Bregman, et al., 1996b; Smirnov, 2011a).

The radio interferometer measurement equation

We now go back to the more realistic signal propagation model Eq. (2.26). One of the fundamental assumptions underpinning the RIME is that all transformations of the signal along its path of propagation are linear. As a result, multiple transformations of the signal can be described as consecutive applications of Jones matrices in a model of

the form

$$\boldsymbol{v} = \boldsymbol{J}_n \boldsymbol{J}_{n-1} \cdots \boldsymbol{J}_1 \boldsymbol{e} = \boldsymbol{J}\boldsymbol{e}, \tag{2.34}$$

where the numbering corresponds to the order in which the transformations are applied[1]. Next we define the *visibility matrix*, $\boldsymbol{V}_{pq}$, as the average of the outer product between the voltages measured by two spatially separated antennas i.e.

$$\boldsymbol{V}_{pq} = 2\langle \boldsymbol{v}_p, \boldsymbol{v}_q^H \rangle, \tag{2.35}$$

where p and q label antennas, a superscript ^{H} denotes complex conjugate transpose and the introduction of the factor two is for convenience as explained in Smirnov (2011a). Using Eq. (2.34) we can now rewrite Eq. (2.35), as

$$\boldsymbol{V}_{pq} = 2\left\langle \boldsymbol{J}_p \boldsymbol{e} \left(\boldsymbol{J}_q \boldsymbol{e}\right)^H \right\rangle = 2\left\langle \boldsymbol{J}_p \left(\boldsymbol{e}\boldsymbol{e}^H\right) \boldsymbol{J}_q^H \right\rangle, \tag{2.36}$$

where $\boldsymbol{J}_p$ and $\boldsymbol{J}_q$ denote the total Jones matrices of antennas p and q respectively. Assuming that $\boldsymbol{J}_p$ and $\boldsymbol{J}_p$ are constant over the averaging interval, or that the averaging interval is small enough so that they can be considered constant, both can be pulled out of the averaging operator to get

$$\boldsymbol{V}_{pq} = 2\boldsymbol{J}_p \left\langle \boldsymbol{e}\boldsymbol{e}^H \right\rangle \boldsymbol{J}_q^H = 2\boldsymbol{J}_p \begin{bmatrix} \langle e_x e_x^* \rangle & \langle e_x e_y^* \rangle \\ \langle e_y e_x^* \rangle & \langle e_y e_y^* \rangle \end{bmatrix} \boldsymbol{J}_q^H. \tag{2.37}$$

As described in Hamaker, Bregman, et al. (1996b) the quantities in angle brackets can be related to the Stokes parameters (Born and Wolf, 1964). For example, for linear feeds, we can write these in terms of the *brightness matrix* $\boldsymbol{B}$ as

$$\boldsymbol{B} = 2\begin{bmatrix} \langle e_x e_x^* \rangle & \langle e_x e_y^* \rangle \\ \langle e_y e_x^* \rangle & \langle e_y e_y^* \rangle \end{bmatrix} = \begin{bmatrix} I+Q & U+iV \\ U-iV & I-Q \end{bmatrix}. \tag{2.38}$$

This brings us the RIME for a single point source, which can be written as:

$$\boldsymbol{V}_{pq} = \boldsymbol{J}_p \boldsymbol{B} \boldsymbol{J}_q^H. \tag{2.39}$$

Eq. (2.39) describes the relationship between the observed visibilities $\boldsymbol{V}_{pq}$ with the source brightness $\boldsymbol{B}$, and the per-antenna Jones terms $\boldsymbol{J}_p$ and $\boldsymbol{J}_q$. To incorporate multiple discrete sources we can simply sum their contributions i.e.

$$\boldsymbol{V}_{pq} = \sum_s \boldsymbol{J}_{p,s} \boldsymbol{B}_s \boldsymbol{J}_{q,s}^H, \tag{2.40}$$

where s labels the source. Then, the phase difference induced by the geometric delay Eq. (2.24) can be accounted for with a combined Jones matrix of the form

$$\boldsymbol{K}_{pq,s} = e^{-2\pi\imath\frac{\nu}{c}(u_{pq}l_s + v_{pq}m_s + w_{pq}(n_s-1))} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{2.41}$$

[1]It is important to note that the order is essential, since matrix multiplication is not commutative in general.

where, since the individual scalar matrices $\boldsymbol{K}_{p,s}$ and $\boldsymbol{K}_{q,s}$ commute, they can be combined into a single matrix that can be placed anywhere in the chain. This allows us to write Eq. (2.40) as

$$\boldsymbol{V}_{pq} = \sum_s \boldsymbol{J}_{p,s}\boldsymbol{B}_s\boldsymbol{J}_{q,s}^H e^{-2\pi\imath\frac{\nu}{c}(u_{pq}l_s+v_{pq}m_s+w_{pq}(n_s-1))}. \tag{2.42}$$

To incorporate a continuous brightness distribution $B(l,m)$, it is necessary to replace the summation by an integral over the unit sphere, i.e.

$$\boldsymbol{V}_{pq} = \int \boldsymbol{J}_p(l,m)\boldsymbol{B}(l,m)\boldsymbol{J}_q^H(l,m)e^{-2\pi\imath\frac{\nu}{c}(u_{pq}l+v_{pq}m+w_{pq}(n-1))}\frac{dldm}{n}, \tag{2.43}$$

where we have used $d\Omega = \frac{dldm}{n}$ to write the integration over the sphere in terms of l and m. This is the continuous form of the RIME that describes the measurement process of an interferometer. Since we are treating the sky as fixed with the earth rotating under it, the baseline coordinates change with time in a way dictated by Eq. (2.31). As the earth rotates, the baselines trace out ellipses, a process commonly referred to as *earth rotation synthesis*. In addition, the individual Jones matrices can change over time in a non-trivial way. It is by now customary to separate the Jones terms into direction-dependent and direction-independent terms. Recall that the order in the Jones chain needs to be preserved, as matrices do not, in general, commute. Splitting the Jones matrices as

$$\boldsymbol{J}_p(l,m) = \boldsymbol{G}_p\boldsymbol{E}_p(l,m), \tag{2.44}$$

where $\boldsymbol{G}_p$ describes DIEs and $\boldsymbol{E}_p(l,m)$ DDEs, then reflects the fact that DIEs tend to be instrumental effects that happen close to the receiver, whereas DDEs tend to describe transformations happening closer to the source (atmospheric distortions for example). We can then go a step further and write Eq. (2.43) as

$$\boldsymbol{V}_{pq} = \boldsymbol{G}_p\int \boldsymbol{E}_p(l,m)\boldsymbol{B}(l,m)\boldsymbol{E}_q^H(l,m)e^{-2\pi\imath\frac{\nu}{c}(u_{pq}l+v_{pq}m+w_{pq}(n-1))}\frac{dldm}{n}\boldsymbol{G}_q^H. \tag{2.45}$$

We will look at this form of the RIME in more detail when we discuss calibration in Section 2.4. For the moment, things are simplified somewhat further in an attempt to gain deeper insight into the nature of the mapping Eq. (2.45). Firstly, assuming that the DIEs are trivial (i.e. the identity), and that there are no atmospheric effects corrupting the signal, the only Jones terms that remain are the direction-dependent sensitivity patterns of the antennas, also known as the *primary beam* pattern. These will be given by the instantaneous Fourier transforms of the individual dish aperture functions, which we assume to be fixed and known for each antenna in the array. Doing so, we can write down the Mueller form of Eq. (2.45) (Hamaker, Bregman, et al., 1996a) as

$$\boldsymbol{V}_{pq} = \int \boldsymbol{E}_p \otimes \boldsymbol{E}_q^*\mathrm{vec}(\boldsymbol{B})e^{-2\pi\imath\frac{\nu}{c}(u_{pq}l+v_{pq}m+w_{pq}(n-1))}\frac{dldm}{n}\boldsymbol{G}_q^H, \tag{2.46}$$

where $\otimes$ denotes the Kronecker product Murnaghan (1938), $\mathrm{vec}(\cdot)$ vectorises matrices by stacking columns and we have left the dependence on l and m implicit for notational convenience. The 4×4 matrix defined by

$$\boldsymbol{M}_{pq} = \boldsymbol{E}_p \otimes \boldsymbol{E}_q^*, \tag{2.47}$$

is known as the Mueller matrix (R. C. Jones, 1947). The diagonal terms of this matrix can, roughly speaking, be related to the instrumental response to the individual Stokes parameters. For an ideal unpolarised interferometer observing an unpolarised source, only the first and last diagonal elements will be non-zero and will describe the instruments' sensitivity to the total intensity

$$I = \langle |e_x|^2 + |e_y|^2 \rangle. \tag{2.48}$$

Combining these terms and assuming that all antennas are identical (so that they have the same primary beam patterns), we finally arrive at the renowned *van Cittert-Zernike* (vCZ) theorem (Zernike, 1938)

$$V(u,v,w) = \int \mathcal{A}(l,m) I(l,m) e^{-2\pi\imath \frac{\nu}{c}(ul+vm+w(n-1))} \frac{dldm}{n}, \tag{2.49}$$

where $\mathcal{A}(l,m)$ represents the combined sensitivity to the total intensity, also sometimes referred to the *power beam*. In going from Eq. (2.43) to Eq. (2.49), we have made a number of crucial simplifying assumptions. In particular, our assumption that all antennas share the same (stationary) primary beam patterns means that $\mathcal{A}(l,m)$ is baseline-independent, something which is never really true in practice. That being said, Eq. (2.49) is a valuable theoretical tool that can be used to understand how Eq. (2.49) can be inverted to reconstruct the thing we are really interested in: creating an image $I(l,m)$ of the sky.

2.3 Image reconstruction

We can now describe how radio maps are actually made, at least in the un-polarised case. In practice, since all computations eventually have to be evaluated numerically on a computer, the measurement operator Eq. (2.49) is implemented by discretising the image $I(l,m)$ into pixels. If we also assume Gaussian noise during the measurement process, the discretised measurement operator can be written as

$$V = RI + \epsilon, \quad \text{where} \quad \epsilon \sim \mathcal{G}(0,\Sigma), \quad E\left[\epsilon\epsilon^H\right] = \Sigma, \quad E\left[\epsilon\epsilon^T\right] = 0, \tag{2.50}$$

where R is a discretisation of the linear mapping defined by Eq. (2.49), $I \in \mathbb{R}^N$ is a discretisation of the image into N pixels and ϵ is a realisation of Gaussian noise described by the zero mean circularly complex Gaussian distribution with covariance Σ. Note that our definition of R may include the assumed known quantity $\mathcal{A}(l,m)$. If this term is omitted the algorithm reconstructs the apparent image, i.e. the image attenuated by the power beam $\mathcal{A}(l,m)I(l,m)$.

An interferometer with N_a antennas measures $V \in \mathbb{C}^M$ data points (some of which may be redundant) with $M = N_t N_\nu N_a(N_a-1)/2$ where N_t is the number of time stamps, N_ν the number of frequency channels of the observation and $N_a(N_a-1)/2$ is the number of antenna pairs, excluding auto-correlations. Therefore, a naive implementation of the measurement operator would cost $\mathcal{O}(MN)$ in time and storage corresponding to a matrix multiplication with $M \times N$ matrix R. This is not usually feasible, even for modest data and image sizes. It is possible to drastically reduce the time and memory complexity by making use of the approximate Fourier relationship of Eq. (2.49). To be able to do this,

note that, when $w \approx 0$ or $n \ll 1$, the final term in the exponent of Eq. (2.49) is negligible. In this case, the mapping Eq. (2.49) corresponds, approximately, to a two-dimensional non-uniform Fourier transform and the measurement operator can be approximated using the *non-uniform Fast Fourier Transform* (NUDFT) (Bagchi and Mitra, 2012; Gentleman and Sande, 1966). Combined with w-stacking (Offringa, McKinley, et al., 2014) or w-projection (Cornwell, Golap, et al., 2008) techniques, this means the measurement operator can be approximately implemented with a complexity of $\mathcal{O}(jM + N \log_2 N)$ where j is the support of the convolution kernel used for gridding and degridding.

As the uv-plane is only partially sampled, the image reconstruction problem from Eq. (2.50) is ill-posed. Thus it is not possible to invert Eq. (2.50) directly and some additional prior information is required to regularise the problem. The most common approach uses sparsity-promoting priors (e.g. the Laplace prior) to promote sparsity in some appropriate dictionary of functions, for example $I = \Psi\alpha$. In this case, the problem can be solved by minimising an energy function of the form

$$\Phi(I) = -\log(P(V, I)) \propto (V - RI)^\dagger \Sigma^{-1} (V - RI) + \lambda \|\Psi^\dagger I\|_1, \tag{2.51}$$

where $P(V, I)$ is the joint probability of data and signal and $\|\Psi^\dagger I\|_1$ is the l1 norm of the image projected onto the space of the dictionary (typically some wavelet basis; the exact details are not of concern here), and λ is the strength of the regulariser relative to the data fidelity term. Importantly, any iterative image reconstruction algorithm will have to implement the measurement operator R multiple times. Since data rates from modern interferometers are approaching the PetaByte scale, the number of times that an algorithm has to apply the measurement operator can become a prohibitive factor in algorithm design. Fortunately, because of the Fourier-like nature of Eq. (2.50), there is a fast approximation of Eq. (2.51), which can yield significant computation performance improvements. The approximation relies on projecting the data fidelity term into image space by noting that

$$\begin{aligned} (V - RI)^\dagger \Sigma^{-1} (V - RI) = & \quad V^\dagger \Sigma^{-1} V - 2 I^\dagger R^\dagger \Sigma^{-1} V + I^\dagger R^\dagger \Sigma^{-1} R I, & (2.52) \\ \approx & \quad V^\dagger \Sigma^{-1} V - 2 I^\dagger I^D + I^\dagger I^{PSF} * I, & (2.53) \end{aligned}$$

where $I^D = R^\dagger \Sigma^{-1} V$ is known as the dirty image and $I^{PSF} = R^\dagger \mathrm{diag}(\Sigma^{-1})$ is the PSF of the instrument and, because of the approximate Fourier nature of the measurement operator, we can approximate the operator $R^\dagger \Sigma^{-1} R$ as a convolution with the PSF. Taking the gradient of Eq. (2.53) and setting it to zero we find

$$I^D = I^{PSF} * I, \tag{2.54}$$

i.e. we recover the familiar result that the dirty image is simply the model convolved by the PSF. It is important to realise the significance of Eq. (2.54), as Chapter 6 is based on this principle. This is why the interferometric imaging problem is often referred to as a deconvolution problem, but it has to be understood that this is an approximation that becomes worse with increasing field of view (FOV) and increasing spread in w coordinate [2]. Historically, this problem has been solved by using the idea of major and minor cycles (as was done for the Cotton-Schwab CLEAN algorithm (Schwab, 1984)). The main

[2] This approximation becomes exact only when there are not DDEs (eg. w-term) present.

idea here is to utilise the image space approximation Eq. (2.53) some of the time (the so-called minor cycle) but also to reconsider the full data occasionally to compute the mismatch between model and data more accurately. Such an approach, the precise details of which are again not relevant here, can significantly reduce the number of evaluations of the full measurement operator. However, we should be aware that any approach that exclusively utilises the image space approximation Eq. (2.53) during the model update step might never converge exactly to the minimum of Eq. (2.51) and may limit the obtainable dynamic range.

In Chapter 6, we evaluate a deep neural network approach, which uses the approximation Eq. (2.53) in an attempt to solve this problem in an efficient way. However, as discussed in Subsection 2.2, raw interferometric data is usually corrupted by instrumental and atmospheric affects. Calibration is therefore required before any imaging can be performed.

2.4 Calibration

Generally, calibration can be thought of as adjusting a measurement process so that an instrument responds as expected to a known model. For radio interferometers this is usually achieved using successive refinements of the model in Eq. (2.45), or some discretisation thereof. This is particularly difficult when observing celestial objects since it places us in a classic "chicken and egg" situation, i.e. we do not usually have a perfectly known model to calibrate with in the first place. In this section we give a brief overview of the steps required to calibrate radio interferometers. We will restrict the discussion to calibration of dish based tracking telescopes but there are similar considerations for phased array feeds, for example.

Reference calibration

Usually, whenever we embark on a new observation, only a few bright sources in the vicinity of the target field are known. Because of their relative strength compared to most other sources in the field, they completely swamp the visibilities. If they are sufficiently stable in time and we know enough about them (eg. known position, frequency dependence (spectrum) and polarisation structure) they can be used to start the calibration process. Ignoring all other sources in the field, we have, from Eq. (2.39), a measurement model of the form

$$\boldsymbol{V}_{pq} = \boldsymbol{J}_p \boldsymbol{M}_{pq} \boldsymbol{J}_q^H + \epsilon, \tag{2.55}$$

where $M_{pq} = \boldsymbol{B}\boldsymbol{K}_{pq}$ is known as the *source coherency* and we recall that $\boldsymbol{K}_{pq}$ is the scalar phase delay matrix. If we again assume Gaussian noise, calibration consists of finding the Jones matrices $\boldsymbol{J}_q$ and $\boldsymbol{J}_p$ that minimise

$$\chi^2 = \sum_{pq} \|\boldsymbol{V}_{pq} - \boldsymbol{J}_p \boldsymbol{M}_{pq} \boldsymbol{J}_q^H\|_F, \tag{2.56}$$

where $\|\cdot\|_F$ denotes the Frobenius norm (Golub, 1968). Since there are more baselines than antennas, this usually results in an over-determined optimisation problem, which can be solved using for example non-linear least-squares. However, since the model is not perfect, some care has to be taken to prevent the algorithm from over-fitting which, in

this context, means mistaking some unmodelled signal (or the unwanted RFI mentioned in Section 2.1) for noise. This can be achieved by prescribing a parametrisation for the Jones matrices, which corresponds to the physical effects we expect to be present in the data and only solving for these effects. We might, for example, specify a model of the form

$$\boldsymbol{J}_p = \boldsymbol{G}_p(t)\boldsymbol{B}_p(\nu)\boldsymbol{\Gamma}_p(t,\nu), \tag{2.57}$$

where we restrict $\boldsymbol{G}_p$ to be a complex diagonal matrix that only depends on time, $\boldsymbol{B}_p(\nu)$ is a complex 2×2 matrix which only depends on frequency and $\boldsymbol{\Gamma}_p(t,\nu)$ is further parametrised as

$$\boldsymbol{\Gamma} = \begin{bmatrix} e^{\imath(a(t)\nu+b(t))} & 0 \\ 0 & e^{\imath(c(t)\nu+d(t))} \end{bmatrix}. \tag{2.58}$$

The parametrisation Eq. (2.57) attempts to parametrise certain instrumental effects. For example, the $\boldsymbol{\Gamma}$ term can account for delay errors (eg. incorrect cable length when steering the telescope). The diagonal part of $\boldsymbol{B}$ models the overall frequency response of the antenna feeds (bandpass) to incoming electromagnetic radiation. Similarly, the off-diagonal terms can model leakages between feeds. Both of these effects are expected to be stable in time. Finally, the $\boldsymbol{G}$ term can describe the time dependence of the instrumental gain.

In general, there can be many different effects to account for and these depend on a number of factors, for example the frequency of the observation and the type of interferometer being calibrated. The more complicated we make the model, the more parameters are introduced, meaning that more signal is required to be able to solve for all of them. Thus, because of the scarcity of bright, well-modelled compact sources, there are practical limitations on how accurately an interferometer can be calibrated using only calibrator sources. In particular, since the frequency response of an instrument can be quite variable, there are only a handful of sources that can be used for bandpass calibration. These sources are sometimes referred to as *primary calibrators* and, for most observations, these are usually situated quite far away from the target field. As a result, observations usually consist of multiple scans where, during calibrator scans the instrument tracks calibrators and during the target scan it is pointing squarely at the field of interest, which is the target field. These scans have to be combined in a way that results in us being able to calibrate the observation. For example, it is not uncommon for an observation to start and end with scans of very bright primary calibrators, which can be used to infer effects that do not change quickly over time, eg. bandpass and leakages. Unless the primary calibrator is very close to the target field, it is also usually necessary to observe objects that are closer to the target field to capture the time variation of the Jones matrices. These objects, known as *secondary calibrators*, are usually dimmer and astronomers do not necessarily have good models for them.

The previous discussion focuses on using external or reference sources to calibrate the instrument. Since the Jones matrices are dependent on time (and the sources are in a different position in the sky), the solutions derived from these calibrator scans are not necessarily valid for the target field. To derive solutions for the target field, it is customary to interpolate these solutions across scan boundaries. It is then possible to roughly correct the data for the target field by applying the inverse of the interpolated solutions as

$$\boldsymbol{V}'_{pq} = \boldsymbol{J}_p^{-1}\boldsymbol{V}_{pq}[\boldsymbol{J}_p^H]^{-1}, \tag{2.59}$$

where $\boldsymbol{V}'_{pq}$ then gives the *corrected data*. The above procedure is sometimes called *calibrator transfer* or first-generation calibration (1GC) (Noordam and Smirnov, 2010). It provides a starting point for calibrating the target field in a process known as *self-calibration*.

Self-calibration

Armed with an approximately correct model for the telescope we can finally move on to the target field. Self-cal (Pearson and Readhead, 1984), is a technique that enables obtaining calibration solutions for the target field in an iterative manner. The process starts by imaging the corrected data Eq. (2.59) to obtain an approximate model for the target field. Since the solutions derived during 1GC are not perfect, these models usually contain calibration artefacts and only the brightest features in the derived maps can be assumed to be correct. If we were to rerun the calibration using these maps, there is a significant risk of biasing the calibration solutions in a way that enhances artefacts. On the other hand, if only the brightest features in the map are included in the model, there is a risk of absorbing some of the unmodelled flux into the calibration solutions. Fortunately, the calibration problem is usually over-determined with the number of degrees of freedom in the data usually about N_a times the number of parameters, at least in the direction-independent case. This, combined with the fact that we expect the Jones matrices to be smooth functions of time, frequency and direction, implies that the latter option is usually the safer route. And, the more parameters we include in the model, the fewer the effective degrees of freedom and the easier it becomes to take into account the model flux into the calibration solutions. Thus, it is customary to first perform direction-independent self-cal, or in the language of Noordam and Smirnov (2010), 2GC. The idea behind 2GC is to successively enhance the calibration solutions and model by alternating between making images of the corrected data obtained using Eq. (2.59), and then using the brightest features in this model to recalibrate the data. The steps can be summarised as follows:

1. Image and deconvolve the corrected data obtained by applying the gain solutions to the data using Eq. (2.59).

2. Select the brightest features in the map, compute the model visibilities and recalibrate using a model of the form Eq. (2.45).

3. Repeat until there is no significant improvement in the model.

While this procedure has been shown to be quite successful in practice, the model that results from it is still corrupted by DDEs. The process of correcting for DDEs during self-cal is know as 3GC. From the above discussion about the dangers of increasing the number of degrees of freedom, it should be clear that correcting for direction dependent effects is a subtle process that can lead to flux absorption and ghosting (Grobler et al., 2014; Wijnholds et al., 2016). Thus, 3GC is usually only performed when an experienced astronomer spots that there are actually DDEs corrupting the data and special care is required to limit the number of parameters of the model. At present, this remains largely a dark art, and no formal quantitative criteria have been developed. This is typically

done by manually isolating troublesome sources (usually corresponding to the brightest sources in the field), computing per source model visibilities and using the model given by Eq. (2.40) to solve for the gains in the direction of the troublesome sources. Unlike DIEs, DDEs cannot be used to form corrected data. Instead, these sources are usually subtracted from the data (Noordam and Smirnov, 2010). The remaining data can then be imaged to obtain a final map of the target field.

Chapter 3

KERN

3.1 Introduction

The installation of scientific software for use in astronomy can be notoriously challenging. The radio astronomy community has a limited number of dedicated software engineers and often lacks the human resources to dedicate to industrial-quality software development and maintenance. It is not uncommon that poorly written and maintained software packages of high complexity are used by scientists around the world, as these provide some set of algorithmic features not available elsewhere. These packages are often difficult to install and compile, since they have not been created with portability in mind.

To prevent repetition, frustration and mistakes during the tedious task of software installation, a more organised structure is desired. In this chapter, we describe KERN, which deals in detail with the described issues. KERN is the name of the project to structure and automate the packaging of scientific software. The main deliverable is the KERN suite, a bi-annually released set of third-party open-source scientific software packages. This relieves the astronomers from the problem of working out installation procedures and enables focus on the relevant science.

The primary goals of the KERN project are: to make it easier to install the scientific software, to supply a consistent working environment to a scientist and to improve interoperability and interchangeability.

Because of human resource limitations, we target KERN at one operating system and distribution. This is unfortunate, but recent development and adaptation of containerisation technology make it easier to deploy packaged software on most platforms. Limiting us to only one platform enables us to focus on performing the packaging only once, and to do this well. The choice of this one platform is then based on install base (desktop, server) and ease of use for user and developer (package creator).

The people who would benefit most from the KERN project are:

- the astronomer, who is interested in using the software bundled with KERN,
- the developer, who wants his radio astronomy software available to a wider range of users and
- the system administrator, who is setting up systems intended to be used for radio astronomical data reduction.

The name KERN means ‘core’ in Dutch and Afrikaans.

3.2 The target platform

A quick look around various astronomy institutes and universities shows that GNU/Linux and OS X are the most frequently used personal computing platforms. On the server side it is without question GNU/Linux. Compared to OS X, GNU/Linux is an open-source and a freely available platform, which is also a clear advantage. These facts combined result in the choice of Linux as the KERN platform.

However, further consideration was required before selection of the most suitable platform. There are numerous variations of GNU/Linux distributions, with different design philosophies and varying packaging formats. The most popular distributions can be split into two groups, RPM (Red Hat Package Manager) package and Debian package-based distributions. There is no major advantage or disadvantage to either package format.

Although there are diverse local trends, it is our experience that in the South African radio astronomy community, the majority of frequently used platforms are Debian-based, specifically Ubuntu LTS. This distribution also appears to enjoy popularity worldwide. Therefore, it was the most logical choice as KERN's target platform.

3.3 Other packaging methods

In this section we discuss other packaging systems available to us, and discuss their relationship to KERN.

Anaconda

A packaging effort named Anaconda is currently gaining popularity. Anaconda is a cross-platform set of scientific software, with a focus on Python. It supports GNU/Linux, Windows and OS X. OS X is also often used as a desktop environment in radio astronomy. Supporting OS X would be advantageous for many end users. We have performed experiments with packaging packages for Anaconda. Users have reported that Anaconda is easy to operate; however, at the time of writing, the packaging procedure is cumbersome for the developer. In effect, developers cannot generate the same high-quality, seamlessly installable packages as are achievable with native Linux packaging methods. Additionally, Anaconda lacks an equivalent to Debian's Lintian, a packaging tool that dissects a Debian package in an attempt to find bugs and violations of the Debian policies.

A significant number of software packages in KERN are not created with OS X support in mind, thus requiring modifications to the source code. Also, compilation procedures can vary greatly across platforms, doubling the packaging and maintenance effort if we would support OS X as a platform.

In addition, Linux distributions come with a large set of prepackaged software, which eliminates the need to package many dependencies. Using Anaconda would necessitate packaging up many dependencies ourselves.

The limitations of Anaconda led to the preference for Debian- over Anaconda packages.

Python and pip

The Python programming language has become the most widely used language in astronomy (Momcheva and Tollerud, 2015). Moreover, this language is bundled with a package manager called pip. Pip assists in downloading and installing Python packages from the Python package index (PyPi). Another useful tool for setting up Python environments is called `virtualenv`. Virtualenv enables a user to set up one or more isolated Python environments without system administrator rights. The combination of these two tools enables the end-user to set up various custom environments with specific versions of dependencies. For pure Python projects, pip and virtualenv are cross-platform and independent of the host operating system package manager. However, pip is less suitable for impure Python projects. Some Python libraries depend on non-Python run time libraries and/or non-Python development headers compile-time, making them 'impure'. A recent improvement to the Python packaging system is the introduction of wheels. Wheels are pre-compiled binary Python packages. These do not require compilation and will work if the packaged library does not have unusual requirements. An example of an

independent binary wheel is Numpy. Numpy only depends on Python and a small set of system libraries. The Application binary interface (ABI) differs across host platforms and Python versions, requiring a wheel for every platform and Python combination. These are supplied on PyPi, and the correct version is automatically selected by pip on installation.

Although KERN supports both Python 2 and Python 3, most Python packages in KERN only support Python version 2. This will soon become an issue, since Python 2 is reaching the end of life phase, meaning it will not receive any (security) updates and support in major Linux distribution will be dropped. Fortunately, the software packages with an active developer community, such as Meqtrees, are rewriting the software to add Python 3 support. At the time of writing, the only package KERN that supports Python 3 is Python-Casacore, but more will follow shortly. All Python 3 packages have the python3 -prefix, so the KERN Python 3 package for Casacore is named python3-casacore.

Although the binary Debian packages are the preferred way to guarantee a stable and compatible runtime environment, it is not always possible for a user to install packages with administrator rights on a system. In the case of a Python package with a binary part like Python-Casacore, users are forced to compile the package from scratch. To avoid disregarding this use-case, binary wheels for Python-Casacore were created. Binary wheels are special Python packages that contain the Python code combined with a pre-compiled form of the code requiring compilation. We follow the 'manylinux2010' platform specification, which is defined in the 'Python Enhancement Proposal' (PEP) number 571[1]. Making a binary wheel that is likely to function on the great variety of available Linux platforms is not a trivial task. To guarantee binary compatibility, an older Linux platform is used to make the packages, hence the 2010 date in the specification. The binary wheel is made using Docker containers based on an old CentOS 6 version. The assumption is that it is extremely like the user's system is be backwards compatible with the older CentOS 6 on the library and binary level.

To keep the binary wheel reasonably small, it was decided not to include the Casacore data. The size of the data is significant, and not all Casacore functionality like opening MSs, requires the data. In addition, the data requires regular updates. The binary data is available for download independently, as a tarball or KERN package.

Collaboration with Debian

Ubuntu is directly based on Debian and is thus similar to Debian. Nonetheless, due to version differences in the bundled libraries in each distribution, KERN packages are unlikely to run on Debian. Fortunately the packaging procedure is identical for Debian and Ubuntu, which makes creating true Debian packages a matter of a recompilation of the source package.

In contrast, the build system, dependency management and library management for RPM are completely different. Porting our packages to RPM is non-trivial, and maintaining support for RPM-based distributions would imply doubling the required effort. That said, the Linux packaging landscape is a constantly evolving world, and basing KERN on RPM is not ruled out for future releases. Fedora and other RPM-based distributions have useful online build tools and hosting facilities, such as the open build

[1] https://www.python.org/dev/peps/pep-0571/

service[2] and Fedora Copr[3].

We have established collaboration with Debian developers, and some packages from KERN (e.g. `casacore` and `aoflagger`) have been incorporated into Debian directly. These packages have been uploaded to the Debian archive, and changes are synchronised between KERN and the Debian archive.

Not all KERN packages are suitable for uploading to Debian. Packages with a small user base or packages that are fragile and receive continuous fixes (as opposed to formal release) are not well suited to this distribution model, since it can take some time before a package ends up in a Debian release. For the more stable packages in KERN, a continuation of this effort is expected, with more packages ending up in Debian in the future.

3.4 Usage

To use the packages of KERN, one needs to add the KERN remote repository to the system. It is recommended to use the latest released version, which is KERN-5 at the time of writing. KERN-5 is packaged for Ubuntu 18.04; using the packages on a different distribution or version will most likely fail. If running Ubuntu 18.04 is not an option on a particular system, using Docker, Singularity (see below) or a virtual machine is recommended.

The `add-apt-repository` command, which is a Ubuntu utility to configure remote repositories, should be used to add the KERN repository to a system. Some packages in KERN depend on Ubuntu packages in the `multiverse` and `restricted` repositories. CUDA, the library required for utilizing a NVidia branded GPU, is an example of a dependency contained in the restricted repository. Once the local cache is updated using `apt-get update` the package cache can be searched using `apt-cache search` *`PACKAGE`* and packages can be installed using the `apt-get install` *`PACKAGE`* command.

In case of an unexpected fault, it is important to ensure that the latest versions of all packages are being used (by running `apt update` and `apt upgrade`), before reporting new issues. Missing packages can be nominated for inclusion in KERN by requesting the packaging on the issue tracker[4]. For further instruction on how to install and use KERN packages, refer to the KERN suite documentation[5].

3.5 Notable packages

Here we discuss the important or non-trivial packages part of KERN. Note that this is not the full list of packages. The full list of packages contained in KERN-5 is provided in Table A.1 in Chapter A.

Casacore

Most of the packages in KERN depend on Casacore (van Diepen, 2015). Casacore is a suite of C++ libraries for radio astronomy data processing. The most important library

[2] https://openbuildservice.org/
[3] https://copr.fedorainfracloud.org/
[4] https://github.com/kernsuite/packaging/issues
[5] http://kernsuite.info/

is the table system for working with MSs, which is currently the most frequently used data storage format in radio astronomy. Since it is such an important package, additional effort has been put into making the quality of this package quality high, and it should be seen as an example and reference for all other packages. The package has also been accepted in the main Debian repository and updates are synchronised between Debian and KERN[6].

Casacore data

Casacore has a 'soft' dependency on the 'casacore data' package. The latter contains ephemerides, geodetic data and other tables required for performing calculations such as coordinate conversions. Strictly speaking, the Casacore data package is not required if one is not calculating e.g. coordinate conversions, but in practice many components of Casacore will fail or give warnings if the data package is missing. The Casacore data package is updated on a regular basis using a cron job[7]. Updated content typically consists of accurate GPS movements of the tectonic plates, new or updated radio telescope positions, leap seconds, etc. There is no central authority controlling the content of the Casacore data package, and various institutes around the world create their own versions. We base the KERN package on the data supplied and updated weekly by the ASTRON, which is published on its public FTP server[8].

MeqTrees

MeqTrees (Noordam and Smirnov, 2010) is a software suite developed at ASTRON and subsequently Rhodes and SARAO. MeqTrees is aimed at implementing various versions of the RIME (Smirnov, 2011a), which is used for simulation and calibration of radio interferometric data. MeqTrees consists of two core packages: Timba (the C++-based computational implementation) and Cattery (a set of Python frameworks providing end-user tools). Other packages in the suite include Tigger (a sky model and FITS image viewer), Owlcat (a set of MS manipulation utilities) and Pyxis (a scripting/pipelining framework).

CASA

CASA (McMullin et al., 2007b), is a popular system for the reduction of radio astronomy data. CASA is maintained by the National Radio Astronomy Observatory[9]. It has proven to be one of the most challenging pieces of software to construct a package of. CASA is delivered as one monolithic self-contained tarball, which bundles a complete set of dependencies, including its own Python interpreter and IPython interface. This scheme has the advantage that it runs on most systems without modification or additional library requirements. However, the tarball is sizable (over 1 Gb at the time of writing). In addition, various dependencies bundled with CASA are dated (for example, the bundled IPython package is version 0.10, dating from 2010). This makes it hard to install updated packages into the CASA bundled Python, since CASA itself may break if packages are updated. Since CASA depends on so many old libraries, it is close to impossible to

[6] https://packages.qa.debian.org/c/casacore.html
[7] https://github.com/casacore/casacore-data-update
[8] ftp://ftp.astron.nl/outgoing/Measures/
[9] http://casa.nrao.edu/

install it as an overlay on the default Debian filesystem and make use of the system Python interpreter and libraries. Some effort has been made towards unbundling an earlier version of CASA (4.3) into individual packages, but the CASA team was unable to dedicate resources to this effort in further releases. We have therefore adopted the single large package option as the only practical possibility for now. Other attempts have been made to bridge the gap between the system and CASA Python interpreter, but the result is still far from ideal (Staley and Anderson, 2015).

We have decided to take a pragmatic approach and package a subset of CASA and instead of installing it as an overlay on the system, choosing to do so in a separate directory. Having a package simplifies the installation and helps users to manage dependencies since there are packages that depend on CASA. We reduced the installation size by unbundling the experimental Carta viewer, which reduced the final installation footprint by about 1 Gb. In the case of CASA 4.7.1 the tarball is reduced by 60% to about 400 MB. The software is installed in `/opt/casa` and a symlink to the casa binary is created under `/usr/bin`. The package works exactly like the upstream CASA installation (apart from the omitted Carta), but unfortunately still cannot interoperate with the Debian system Python installation. Since the package is a modified version of the original release, we decided to rename the package to `casalite`.

Since 17 December 2019, CASA 6 has been available as a binary wheel. This greatly enhances the operability of CASA in combination with other Python packages in a custom Python environment. Unfortunately, the new package layout has not been adopted in KERN yet.

AIPS

NRAO's venerable AIPS is the predecessor of CASA and, despite its age, maintains an extensive and enthusiastic user base. AIPS has some of the oldest code of all the packages in KERN and depends on various (nowadays rather arcane and archaic) system configuration modifications. We bundle AIPS with a minimal configuration that assumes a single system (localhost) setup. All software is installed under `/opt/aips` , similar to CASA. The software requires a writable data folder in order to operate, and thus when the package is installed, an `aips` group is created if it does not exist. Users who want to use AIPS need to be added to this group. Owing to the complexity of AIPS compilation, the KERN packages are not compiled from source, but are instead based on the binary tarball distribution.

LOFAR

Another notable package is the `lofar` package, which contains all the code bundled in the LOFAR imaging pipeline. Most of it is used by astronomers working with data from the LOFAR radio telescope; however, it has a number of general-interest components, such as the PyBDSM / PyBDSF source finder and the `makems` MS creation tool. For the reader familiar with building and using the LOFAR pipeline, it is relevant to know that the full 'Offline' bundle is contained in the KERN package. The LOFAR software is one of the most frequently used KERN packages.

Pulsar software

KERN contains a number of software packages from the pulsar community, such as `tempo`, `presto` and `psrchive`. These tools were particularly difficult to create packages from, since most of them do not perform normal release management and tend to have broken build scripts. For packages without version numbers, we introduce our own date-based versioning scheme and the packages are updated on request.

Unversioned packages

The pulsar software developers are not the only astronomers who do not use software versioning. For example, the Miriad software developers do not version this software either. For the KERN versioning and as a method to check if the source might have changed, the upload timestamp on the publishing FTP server is used. This is far from ideal. To check if there have been improvements to the software, the timestamp needs to be manually checked, which is a deviating workflow compared to all other astronomy packages. In addition, a change in modification timestamp does not always imply an update to the package content.

3.6 Containerisation

Docker

Docker is a mature and popular concept for distributing software and managing processes. Although it is easy to create a Docker container from a piece of software, it is no replacement for proper software packaging. We are of the opinion that proper packaging and containerisation go hand in hand: Debian packages provide robustness and dependency management while containerisation provides portability and distributability.

We have prepared an easy-to-use base Docker image that can be used to create custom Docker images containing all the KERN packages combined with end-user scripts. The Dockerfile below is all that is needed to set up a Docker container for any given package in KERN. The example below is for AOFlagger:

```
FROM kernsuite/base:5
RUN docker-apt-install aoflagger
```

The `kernsuite` Docker image is a clean Ubuntu system with the KERN suite repository enabled. It also contains an up-to-date pip so that one can directly install Python libraries. The `docker-apt-install` command is just a wrapper script that updates the apt cache before installing the package, then removes the cache. The latter is done to prevent cluttering of the Docker image, which could otherwise lead to exploding image sizes.

Singularity

A number of assumptions made by Docker creators have created security concerns and have made Docker a poor fit to the typical high performance computing (HPC) environment. This has motivated the development of an alternative containerisation technology called Singularity. While this is at present less popular, Singularity is more suitable for

deploying software in a multi-tenant cluster environment, which is the most common environment in science. We have created scripts to set up Singularity images containing all KERN software easily; these can be used to deploy all of KERN on any cluster supporting Singularity[10].

3.7 Project structure

All the packaging effort of KERN is strictly open-source and open-development. All source code is publicly available on Github[11]. We use the Github bug tracker to interact with users. Users can report problems, ask questions and request new features via the bug tracker. Participation is encouraged by the means of pull requests. There is a central entry point website that contains a list of common questions and links to all related services and pages[12].

The release cycle

Originally it was anticipated to have a KERN release recycle of approximately six months, but in practice it has proven difficult to keep up this pace. At the time of writing, the latest release is KERN-6, released on 17 June 2020. The previous release, KERN-5, was released on 15 January 2019. Meanwhile, KERN-7 is planned to be released in March 2021, which again violates the six-months commitment. The delays are caused by limited available manpower and a lack of funding.

Every KERN release is 'fixed', in the sense that no package updates are planned post-release, unless some critical issues need to be addressed. Between KERN releases, development activity proceeds on an active development branch called KERN-dev. This repository will constantly be updated with new packages, but these should only be used for testing and experiments rather than for production science.

KERN is an open-source and open-development effort; any institute or individual is free to contribute work, fund the project or pick up responsibilities. In practice, the project is currently managed by two individuals, Gijs Molenaar and Athanaseus Ramaila. To guarantee the project's continuation in case of an unexpected incident, administrative access to the repositories and project secrets have been shared with a small set of trusted individuals working at ASTRON and SARAO. Nonetheless, the coordination of the KERN releases costs developer time and thus money, so the long-term continuation of the KERN project depends on institutes' willingness to fund the project.

Technical structure

The proposed packaging procedure makes extensive use of git. Every release of every package added to KERN is mirrored on a packaging Github repository[13]. Every new release is a new commit to the git repository. These commits are then augmented with Debian metadata files. The metadata files contain a description, build and runtime dependencies as well as a robust script to build, install and clean the package. These

[10] https://github.com/kernsuite/singularity
[11] http://github.com
[12] http://kernsuite.info/
[13] https://github.com/kernsuite-debian

Table 3.1: KERN download statistics

KERN Version	Released	Number of Packages	Total Download Count
1	August 2016	65	12 544
2	March 2017	69	41 655
3	November 2017	78	120 447
4	June 2018	88	64 460
5	January 2019	117	166 669
6	June 2020	65	4 907
total			410 682

metadata files can be very minimal if properly written build scripts are provided by the original software authors, but can be more complicated in the absence of these.

New packages are published to Launchpad, which is a free to use service maintained by Canonical, the company responsible for the Ubuntu distribution. Packages are built on the Launchpad build farm, and we simply upload the base source image. This is an extra quality check since the build farm makes sure that the package compiles correctly and that all the dependencies are correctly defined.

If the original source is provided with (unit)tests, as is the case for Casacore, we run these tests during the creation of the package. Unfortunately, most software packages in KERN do not have a test suite provided by the developers.

All scripts and packaging files are released under the conditions of the Massachusetts Institute of Technology (MIT) license. The MIT license is a permissive and straightforward license; it only requires preservation of copyright and license notices. Licensed works, modifications and larger works may be distributed under different terms and without source code. Note that this only applies to the KERN files and not to the software contained by the KERN packages. The licenses of these packages are respected and are bundled with every package.

3.8 Recommended usage

KERN is freely available and maintained as a service to the community. If one uses KERN in a published work, it should be stated which version of KERN was used and include it as a citation and/or an acknowledgment.

3.9 Usage numbers

The launchpad website hosts a public API, which can be queried for usage statistics[14].

Table 3.1 lists the total number of package downloads. A growth in numbers is clearly visible. There has been a dip in the usage for KERN-4, which is most likely due to two 'core' (Casacore and Python-casacore) packages migrating from KERN into the standard Debian and Ubuntu distributions. From KERN-4 onwards, KERN packages that depend on the core packages use the versions in the Debian/Ubuntu archives by default, so the core downloads are not counted by launchpad. (However, when a newer version of a core package needs to be released to the community, this is initially uploaded to KERN,

[14] https://help.launchpad.net/API/

Table 3.2: Top 10 packages per KERN release

	KERN-1	KERN-2	KERN-3	KERN-4	KERN-5	KERN-6
1.	casacore	casacore	casacore	lofar	casacore	casacore
2.	python-casacore	lofar	python-casacore	losoto	casarest	casarest
3.	casarest	python-casacore	casarest	casarest	meqtrees	python-casacore
4.	kittens	casarest	lofar	meqtrees	kittens	meqtrees
5.	meqtrees	kittens	kittens	prefactor	makems	aoflagger
6.	tigger	meqtrees	meqtrees	factor	aoflagger	pybdsf
7.	pyxis	pyxis	wsclean	tempo2	wsclean	casalite
8.	aoflagger	aoflagger	makems	casalite	pybdsf	wsclean
9.	lofar	wsclean	tigger	blimpy	miriad	makems
10.	wsclean	tirific	losoto	psrchive	lsmtool	stationresponse

and is then served up from KERN until the Debian/Ubuntu versions catch up. During such interim periods, downloads of the core packages **are** counted by launchpad, which explains why they appear in the top 10 anyway.) Other popular KERN packages that have found their way into the Debian and Ubuntu archives are AOFlagger and WSClean. Because of these successes, the true number of package downloads for which KERN should be credited is probably much higher than the launchpad numbers alone.

Table 3.2 lists the top 10 most popular packages per KERN release. Casacore and Python-casacore are clearly the most popular, which is not surprising, since most other packages depend on these libraries. Surprisingly, the top 10 for the KERN-4 release contains a number of LOFAR packages, which could be explained by the activities for the European Open Science Cloud For Research Pilot Project[15] around that time.

3.10 Conclusions

Although the KERN project does not introduce any novel algorithmic techniques as such, we believe that it is a foundational block for a robust radio astronomy software environment. Radio astronomy software has historically been challenging to build and install for the end-user. A growing number of radio astronomers use the packages distributed via KERN (Creaner and Carozzi, 2019; Sabater et al., 2018; Sabater et al., 2019; van Hateren, 2019).

Based on the subjective observation of users within our team at SARAO, ease of software installation was vastly improved after KERN became available. Subsequently, the KERN ready-to-install packages resulted in a noticeable reduction in time and effort required to achieve successful software installation. While we believe this experience will not be unique to SARAO, a further quantitative survey of the global community could not be carried out.

We believe that these packages, as well as the regular release cycle, are an excellent improvement for reproducible science, problem isolation and discoverability of tools. KERN is not only of benefit to astronomers, but also to system administrators, who now have to spend less time installing software and tracking updates and changes.

[15] https://eoscpilot.eu

Chapter 4

Kliko

4.1 Introduction

Software in science

The use of computer software in research has resulted in significant hardware and software developments in computing science. Nowadays, the number of different scientific software packages is overwhelming, and it has become progressively difficult for users (e.g. a scientist) to evaluate the relevance, usage and performance of these packages.

Firstly, installing scientific software can be cumbersome, especially when the installation and/or compilation is poorly designed. The software code, the library dependencies, the host platform and the compilers may change over time, making it unclear how the original developer(s) intended to install and use the software. Secondly, conflicting dependencies may arise when different software packages are built together, making it difficult to install them on the same system. Thirdly, software packages have non-uniform interfaces, as they have varying expectations of interaction with a user or with other packages on the same system.

Kliko is a Docker-based encapsulating and chaining framework that purports to mitigate these issues by creating a container of the software, thereby solving the first and second issues above. The third issue can then be solved by building a Docker container that has minimal extra requirements, i.e. the Kliko definition.

Kliko consists of two parts: i) a set of utilities for creating a container, including parsers to check if all (meta) data is valid; and ii) a support library that can be used to schedule a Kliko container and run it from a command line or from a web interface.

Kliko is not a pipeline construction tool itself, nor a web interface, but it can assist in making these.

Software containerisation with Docker

Containerisation is a method for building self-contained environments (called 'containers') for applications. These containers can then be distributed and used with minimal effort on a large variety of platforms.

Containerising applications is not novel. Similar techniques have been applied before, e.g. jail for FreeBSD [1], zones for Solaris (Price and Tucker, 2004) and chroot for GNU/Linux [2]. However, their application was mostly limited to enhancing security and carrying out clean builds of the UNIX system. The addition of an operating system (OS) level process isolation, named control groups or cgroups (Rosen, 2013)), to the popular Linux kernel (since 3.8, 2008) accelerated the adoption of containerisation for the usage of software distribution.

There are multiple software projects leveraging cgroups, for example rkt[3], Docker (Boettiger, 2014)[4], Singularity (Kurtzer, 2016; Veiga et al., 2019)[5] and LXC[6]. Docker (Merkel, 2014) is currently the most popular container technology with the largest community of users and most momentum for future development and support. Kliko aims to be

[1] https://www.freebsd.org/doc/handbook/jails.html
[2] http://man7.org/linux/man-pages/man2/chroot.2.html
[3] https://github.com/coreos/rkt
[4] https://www.docker.com
[5] http://singularity.lbl.gov
[6] https://linuxcontainers.org

agnostic of the container technology, but since Docker has the biggest user community, we focus on this implementation.

In Docker, an image is built using an initialisation script (a 'Dockerfile'), which contains the recipe to install or build the application. The Dockerfile is a series of commands applied to a basic and clean Docker image, typically a headless Linux distribution. These base images are retrieved from an online database provided by Docker, and stored locally. The Dockerfile, when executed, will create an 'image', which is a 'inactive' snapshot of the virtualised application. An image becomes a container when instantiated (e.g. the application runs). The difference between active and inactive is important; a container is an image in an unwritten (dirty) state.

An application that is containerised is self-contained and can be seen as a complete OS without a kernel. The container could even only contain a statically compiled binary, but in practice, it is useful to have the tools and package manager of a Linux distribution available inside the container. Theoretically, to run a Docker container using Docker on a host machine, the only requirement is to have the Docker daemon running on the host. Unfortunately, there are some hardware-specific edge cases such as CPU register usage optimisation and GPU acceleration. These cases will be discussed in Subsection 4.7.

A Docker container 'image' is basically a file system snapshot of a minimal OS The 'target' application (i.e. the one to host) and its library dependencies are installed inside this virtual isolated file system. When the application is started, the container file system is exposed to the application as the working environment.

On a kernel level, cgroups and namespaces are used to create a new isolated environment for the application, limiting access to other processes to the host and presenting the isolated environment as if it is a separate host to the application. Intuitively, this can be seen as similar technology as CPU level OS virtualisation such as VirtualBox; however, in the case of containers, the kernel is shared by the host and the guest.

By default, a Docker container is assigned a private IP address on an internal network range. This makes the container appear as a separate networked machine to the host. By default, access to network ports are restricted, and access needs to be granted per port. One can also forward the port to an external interface where it will appear as the service is running on the host itself.

All of the above might appear similar to simple virtualisation, but containerisation has some clear additional advantages. Firstly, when using Docker, available physical resources do not need to be partitioned between the host and the guest. While memory size allocated to a virtual machine is fixed or not easy to change, running containers does not require the user to fix this memory size, although it is still possible to limit the amount of memory allocatable by the process. Secondly, there is no CPU instruction emulation, as the process is directly executed on the host kernel. Thirdly, there is minimal startup and shutdown overhead for starting containers, as the containerised OS is reduced to minimal consumption. Startup time is instantaneous (in the millisecond range), and loading time will only become noticeable when high numbers of containers are spawned.

In addition to containerisation, Docker also offers other features: it uses a 'union' file system to join multiple layers of file systems together. The intermediate result of each command in the Dockerfile is cached and stored in layers. These layers can be reused by other containers, allowing data sharing between them, which reduces the size of the storage requirements. These layers can also be stored in a central location, where they can be distributed and reused in either a public or a private way.

4.2 The Kliko specification

The Kliko specification is designed to extend containerisation with a uniform interface, resulting in simplified interaction with the containerised application.

The Kliko specification describes what a Kliko container should look like and what a Kliko container should expect during run-time. The relevant terminology is listed below:

Def 1: The Kliko image
A Docker image complying to the Kliko specification. An image is a read-only ordered collection of root file system changes and the corresponding execution parameters for use within a container run-time.

Def 2: The Kliko container
A container is an active (or inactive if exited) stateful instantiation of a Kliko image.

Def 3: The Kliko runner
A process that can run a Kliko image to make a container. For example the Kliko-run command line tool, or RODRIGUES (see Section 4.5).

Def 4: The Kliko parameters
A list of parameters that can influence the behaviour of the software in the container. The list can be arbitrary in size and may consist of any combination of primitive types listed in Table 4.2.

The Kliko image

A Kliko image should contain a `/kliko.yml` file in YAML[7] syntax following the Kliko schema described in Section 4.2. YAML is a human-readable data serialisation language and stands for *YAML Ain't Markup Language*. The Kliko image should also contain a `/kliko` file that is called during run-time by the Kliko runner. This Kliko script can be anything executable but in most cases, it will be a Python script using the Kliko library to check and parse all related Kliko tasks during run-time. Note that we have deliberately chosen not to use the ENTRYPOINT or CMD statements supported by Docker. This way, Kliko is non-intrusive and can easily be added to existing containers that already set an ENTRYPOINT or CMD.

Expected run-time behaviour

During run-time, the Kliko runner will gather the parameters and expose them to the Kliko container. The content of the variables is exposed by the Kliko runner in the `/parameter.json` file, which should contain a flat dictionary in JSON syntax[8]. JSON and YAML are structurally very similar, but YAML is designed to be more human-readable, hence the choice of YAML for the Kliko definition. Future versions of Kliko will support both formats.

While reading this text, one might get confused by the context of the file location (inside or outside the container). As a rule of thumb, if a path in this text starts with a slash (`/`), it is *inside* the container.

If one or more of the parameters is a file, those will be exposed by the Kliko runner in the read-only `/param_files` folder during run-time. It is the responsibility of the Kliko

[7] http://www.yaml.org
[8] https://www.json.org

container to parse the `/parameters.json` file, perform the potential run-time housekeeping and convert the parameter keys, values and/or files into an eventual command to be executed.

It is recommended to write logging to stdout and stderr. This makes it easier for the Kliko runner to visualise or parse the output of a Kliko image.

Flavours of Kliko images

We distinguish two flavours of Kliko containers, *joined Input/Output* (read-write) and *split IO* (read-only). The style of container is specified in the `io` field in the `/kliko.yml` file inside the container; see Section 4.2.

The difference is in the way the contained software interacts with the working data. In the case of *split IO* the Kliko runner exposes the input data to the container in the `/input` folder. This folder is read-only, to prevent accidental manipulation of the data. The Kliko container is expected to write any output data into the `/output` folder. The Kliko runner will then handle this output data after the container reaches the end of its lifetime. A *split IO* Kliko container should always yield the same results for multiple independent runs when presented with the same data and parameters (formally is called, 'having no side effects'). This is basically the essence of the functional programming paradigm.

In the case of *joint IO* the `/work` path is the only one point of interaction with the Kliko host and is exposed as a read/write volume. Basically, the input and output folders are combined into one that is mounted with read/write permissions. Contrary to the *split IO* flavour, this might be potentially dangerous for data processing, as it can alter the original data.

From a run-time parallelisation perspective, the *split IO* flavour is preferred. A container without side effects enables the Kliko runner to make a graph-based logical inference of dependencies and execution scheduling, reuse results and also run various containers in parallel, potentially resulting in faster execution. In practice, existing software does not always support this type of operation, or it is simply not feasible to create a copy of the data. In that case, the *joined IO* style has to be used.

The `/kliko.yml` schema

A *kliko.yml* file is a YAML file and it *should* contain the fields listed in Table 4.1.

Each section contains a list of fields. Each field's statement should contain a list of field elements. Each field element has two mandatory keys, a name and a type. Name is a short reference to the field which needs to be unique. This will be the name for internal reference. The type defines the type of the field; possible types are listed in Table 4.2. Depending on the type there are optional extra fields, listed in Table 4.3.

The schema described above is defined in the Kwalify format. The full schema is listed in Appendix B Kwalify is a parser and schema validator for YAML and JSON[9]. The definition itself is also written in YAML . The Kliko library pykwalify[10] is used to validate the YAML file against a schema. The full Kliko version 2 schema is listed as Listing 10 in Section B.1.

[9] http://www.kuwata-lab.com/kwalify
[10] https://github.com/Grokzen/pykwalify

Table 4.1: Required Kliko fields

field	description
schema_version	The version of the Kliko specification, independent of the versioning of the Kliko library
name	Name of the Kliko image. For example 'radioastro/simulator' for RODRIGUES.
description	A more detailed description of the image.
url	Website of project or repository where project is maintained
io	'join' or 'split'. See the two flavours of Kliko containers in Subsection 4.2
Sections	A list of one or more sections, grouping fields together.

Table 4.2: Kliko variable types

Type	Description
choice	Field with a predefined set of options, see the optional choices field below
str	String value
float	Float value
file	A file path. This file will be exposed in `/param_files` at run-time by the Kliko runner
bool	A boolean value
int	An integer value

The `/parameters.json` file

When a container is started, the Kliko runner will mount a `/parameters.json` file into the container. This file contains all parameters for the container in the JSON format. The `/kliko` script supplied by the container author should read and parse the `/parameters.json file`. The Kliko library (Subsection 4.3) supports helper functions and scripts to parse and validate this file. Validation is done based on the `/kliko.yml` definition, which is useful for preventing or tracking down problems.

An example of a parameter file that could be generated based on the kliko.yml definition is shown in Listing 1.

```
{
   "int": 10,
   "file": "some-file",
   "char": "gijs",
   "float": 0.0,
   "choice": "first"
}
```

Listing 1: Example `parameters.json` file

Note that the sections are not supplied, since they are only used for grouping and representation to the user.

Table 4.3: Kliko field types

Field	Description
initial	Supply an initial (default) value for a field
max_length	Define a maximum length in case of string type
choices	Define a list of choices in case of a choice field. The choices should be a mapping
label	The label used for representing the field to the end user. If no label is given, the name of the field is used
required	Indicates if the field is required or optional
help_text	An optional help text that is presented to the end user next to the field

4.3 Running Kliko containers

Running a container manually

As an example, we will describe a straightforward Kliko container named 'fitsimagerescaler", available on the github repository described in Section 4.6. This container takes a FITS image file, which resides in the `/input` directory, opens it, multiplies the pixel values by a parameterised value (2 by default) and exports the result as a new FITS image in `/output`. The actual code that is run in `/kliko` is shown in Listing 5.

Starting the Kliko container is nothing more than starting the container using Docker with some specific flags. If the `parameters.json` file already exists, starting the container from the command line looks like this:

```
$ docker run -t -i \
   -v `pwd`/parameters.json:/parameters.json:ro \
   -v `pwd`/input:/input:ro \
   -v `pwd`/output:/output:rw \
   kliko/fitsimagerescaler /kliko
```

Listing 2: Command for running a Kliko container manually. The `/parameters.json` file is mounted, as well as the input and output directories, in the 'split' mode. The `kliko.yml` is already inside the container.

`‘pwd‘/input` and `‘pwd‘/output` are input and output folders in the current working directory outside the container. `‘pwd‘` is required since the Docker engine can only work with absolute paths.

This command fires up the `fitsimagerescaler` container, mounts the `parameters.json` file as well as input/output directories and runs the `/kliko` script located in the root directory. In this case, the FITS image file has to be present in the local input directory for the script to run correctly.

For the reader unfamiliar with Docker, this command might look cumbersome and error-prone. However, the command constitutes the fundamental principle of Kliko (in addition to specification and the extensive test suite). This can be used as a base to create a Kliko runner in any language that has Docker bindings.

This way of implementing inputs, outputs and running generic scripts demonstrates that it becomes relatively easy to connect the input parameters and data (generated by

scripting and/or a web form) to software living inside the container. Kliko implements a set of tools to ensure the robustness of this implementation.

Inside the Kliko container

The `/kliko` script is the first entry point into the specifics of the container. We can easily parse the `/parameters.json` file using a JSON parser in Python, by performing the commands in code Listing 3.

```
import json
parameters = json.load(open('/parameters.json', 'r'))
```

Listing 3: Example of how to parse `parameters.json` file with standard Python packages.

However, at this point, the parameter file has not yet been validated. We can be sure that the parameters file is generated from our Kliko definition by installing the Kliko library inside the Kliko container and using it from the `/kliko` Listing 4. Validation helps reduce human or programming error.

```
from kliko.validate import validate
parameters = validate()
```

Listing 4: Example of how to parse `parameters.json` using the Kliko library

After the Kliko validation has been performed, a dictionary is created, and all values can be used freely inside the script itself (by passing them to functions) or passed directly to the container OS as environment variables. All this validation is intended to reduce human or programming error as early as possible.

```
import kliko
from kliko.validate import validate
from astropy.io import fits

parameters = validate()
file = parameters['file'] # filename in /input
factor = parameters['factor'] # multiplying factor

print('welcome to fits multiply!')
print("'%s' multiplied by '%s':" % (file, factor))

data = fits.getdata(file)
multiplied = data * factor
output = path.join(kliko.output_path, path.basename(file))
fits.writeto(output, multiplied, clobber=True)
```

Listing 5: Example of `/kliko` file scale the values in a FITS image

Kliko-run

Instead of calling Docker directly, Kliko is bundled with `kliko-run`, a command-line utility that enables a user to run a Kliko container seamlessly. It also assists in exploring

the parameters that a given Kliko container supports. Listing 6 shows the docstring of the kliko-run command for a simple container (available as a test container shipped with Kliko). The optional arguments are generated automatically from the YAML file. It shows how any shipped application can easily be interfaced with the host system, in such a way that part (or all) of the variable names of the application can be modified directly from the command line. This enables Kliko, with the help of Docker, to ship sophisticated software as an application that is equipped with a simple interface. Kliko provides a simple way to implement this interface in a controlled and robust way while being completely agnostic about the mechanics happening inside the container.

```
$ kliko-run kliko/fitsimagerescaler --help

   usage: kliko-run [-h] [--target_folder TARGET_FOLDER] --choice {second,first}
                    --char CHAR [--float FLOAT] --file FILE --int INT
                    image_name

   positional arguments:
     image_name

   optional arguments:
     -h, --help            show this help message and exit
     --target_folder TARGET_FOLDER
                           specify output or work folder (default: ./output)
     --choice {second,first}
                           choice field (default: second)
     --char CHAR           char field, maximum of 10 chars (default: empty)
     --float FLOAT         float field (default: 0.0)
     --file FILE           file field, this file will be put in /input in case
                           of split io, /work in case of join io
     --int INT             int field
```

Listing 6: Output of the `kliko-run` command

4.4 Chaining containers

Kliko containers can also be chained. Chaining means that the output of a container is connected to the input of a consecutively executed container. This enables the creation of workflows. In addition, if the Kliko containers are 'split IO", we can execute containers that do not depend on each other in parallel. Their intermediate results can be cached, which can speed up the execution time of future workflow runs and can help debugging problems with the workflow by examining intermediate results.

There are various workflow creation frameworks and libraries available. We evaluated two popular Python-based workflow management libraries, airflow[11] and Luigi[12]. Although Kliko is designed to be workflow management independent, Luigi is a better fit. Airflow is intended to visualise automated repetitive tasks such as cron jobs, while Luigi is more oriented towards once-off batch processing. Luigi is an open-source Python library that handles dependency resolution, does workflow management, optionally visualises data in a web interface and can handle and retry failures. At the core of a Luigi workflow is the task, which is a Python class that defines what is to be executed, how to

[11] https://airflow.apache.org
[12] https://github.com/spotify/luigi

check if this task has already been executed and optionally if it depends on the result of another task. This is a straightforward but powerful concept that integrates fluently with Kliko. The Kliko library contains a KlikoTask definition that can be used to integrate Kliko in a Luigi pipeline.

4.5 Example of usage of Kliko

VerMeerKAT

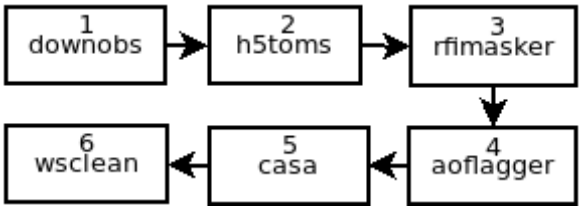


Figure 4.1: Flow diagram of the VerMeerKAT data reduction pipeline.

To illustrate the mechanics of chaining containers together, we explain a real-world application here, the VerMeerKAT pipeline.

VerMeerKAT is a semi-automated data reduction pipeline for the first phase of deployment of the MeerKAT telescope[13] (Booth et al., 2009; Jonas, 2009). All steps in this pipeline are shown in Fig. 4.1. It is a closed source project used internally at SKA South Africa and is based on a set of bash scripts. Using bash for this is not ideal; it is hard to make a portable pipeline, not trivial to recover and continue from errors, reusing intermediate results. Parallelisation is possible, but this needs to be manually and explicitly defined in the scripts.

For this paper, we made a Kliko version of this pipeline[14]. Using Kliko for composing this pipeline has some key advantages: i) easy installation and deployment of the software; ii) optional caching of intermediate data products; ii) implicit parallelisation of task-independent steps; and iii) progress visualisation and reporting using a tool such as Luigi.

The VerMeerKAT pipeline, (see Fig. 4.1), starts by querying the MeerKAT data archive for a given set of observations (step 1), along with the meta-data for those observations. The next step is to convert the downloaded data from the hdf5 format to a MS (Kemball and Wieringa, 2000) (step 2), since most radio astronomy tools only support this format. Once the data is in the MS format, it is then taken through a series of manual and automated tools that excise data points that are contaminated by radio frequency interference (Offringa, van de Gronde, et al., 2012; Prasad and Chengalur, 2012) (step 3, 4 and 5). The data is then calibrated (Hamaker, 2006; Smirnov, 2011b) and imaged (D. S. Briggs et al., 1999) (step 6).

For the creation of the VerMeerKAT Kliko containers, we make use of the packages from KERN (Molenaar and Smirnov, 2018). KERN is a bi-annually released set of radio astronomical software packages. This suite contains most of the tools that a radio astronomer needs to process radio telescope data. These packages are precompiled binaries

[13] http://www.ska.ac.za/gallery/meerkat/
[14] https://github.com/gijzelaerr/vermeerkat-kliko

in the Debian format and contain all the metadata required for installing the package, such as dependencies and conflicts. KERN is only supported on Ubuntu 18.04 at the time of writing, but that is no problem when running inside a Docker container.

Listing 7 is an example of Dockerfile for the WSClean (Offringa, McKinley, et al., 2014) Kliko container in VerMeerKAT. When this file is built, it will install the KERN package of WSClean inside the container and bundle the container with the Kliko definition and parser script. The Docker files for the other steps are very similar.

```
FROM kernsuite/base:1
RUN docker-apt-install wsclean
ADD kliko.yml /
ADD kliko /
```

Listing 7: Dockerfile for a KERN package

Listing 8 is an example of a Kliko task definition. This example will use the rfimasker Kliko containers. It depends on the H5tomsTask Kliko task. When this task is invoked using Luigi, Luigi will make the dependency resolution, check if the required tasks have run and if not, run them. The progress can be visualised with the Luigi interface (Fig. 4.2). All other steps in the workflow are very similar to this example.



Figure 4.2: Screenshot of Luigi, running the vermeerkat pipeline

RODRIGUES

Another project using Kliko is RODRIGUES (RATT Online Deconvolved Radio Image Generation Using Esoteric Software)[15]. RODRIGUES is a web-based Kliko job scheduling tool, and it uses Kliko as a required format for the job. RODRIGUES acts as a

[15] https://github.com/ska-sa/rodrigues

```python
from kliko.luigi_util import KlikoTask

class RfiMaskerTask(KlikoTask):
    @classmethod
    def image_name(cls):
        return "vermeerkat/rfimasker:0.1"

    def requires(self):
        return H5tomsTask()
```

Listing 8: An example of a `KlikoTask`

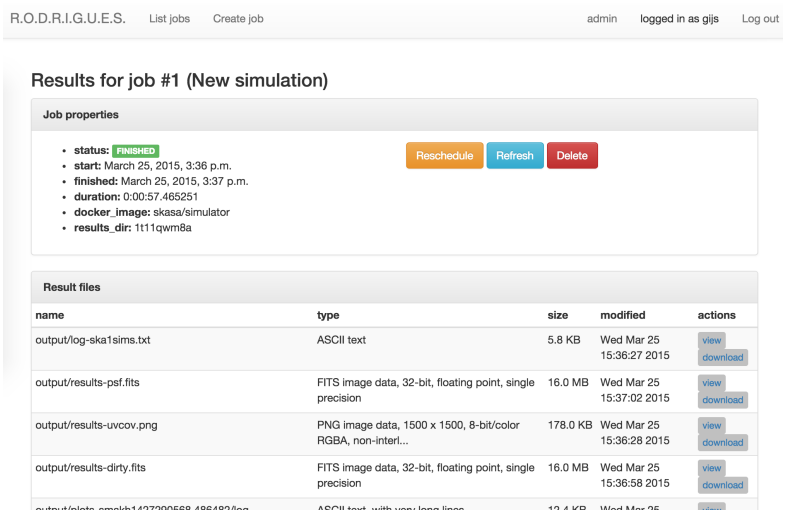


Figure 4.3: Screenshot of RODRIGUES; the result visualiser.

'kliko runner'. A user of RODRIGUES can log into RODRIGUES and add a new Kliko container. RODRIGUES will open the Kliko container, parse the parameters and expose these parameters to the user using a web form (Figure 4.4). The user can then fill in the parameters in this form and submit the job into the RODRIGUES container queue. The container will be run on the system configured by the RODRIGUES system administrator. Once the job is finished, the results are presented to the user in the same web interface (Figure 4.3).

RODRIGUES makes it considerably more convenient to schedule new jobs with varying parameters, enabling scientists with minimal programming or computing knowledge to run experiments in a clean, visual, structured and reproducible procedure.

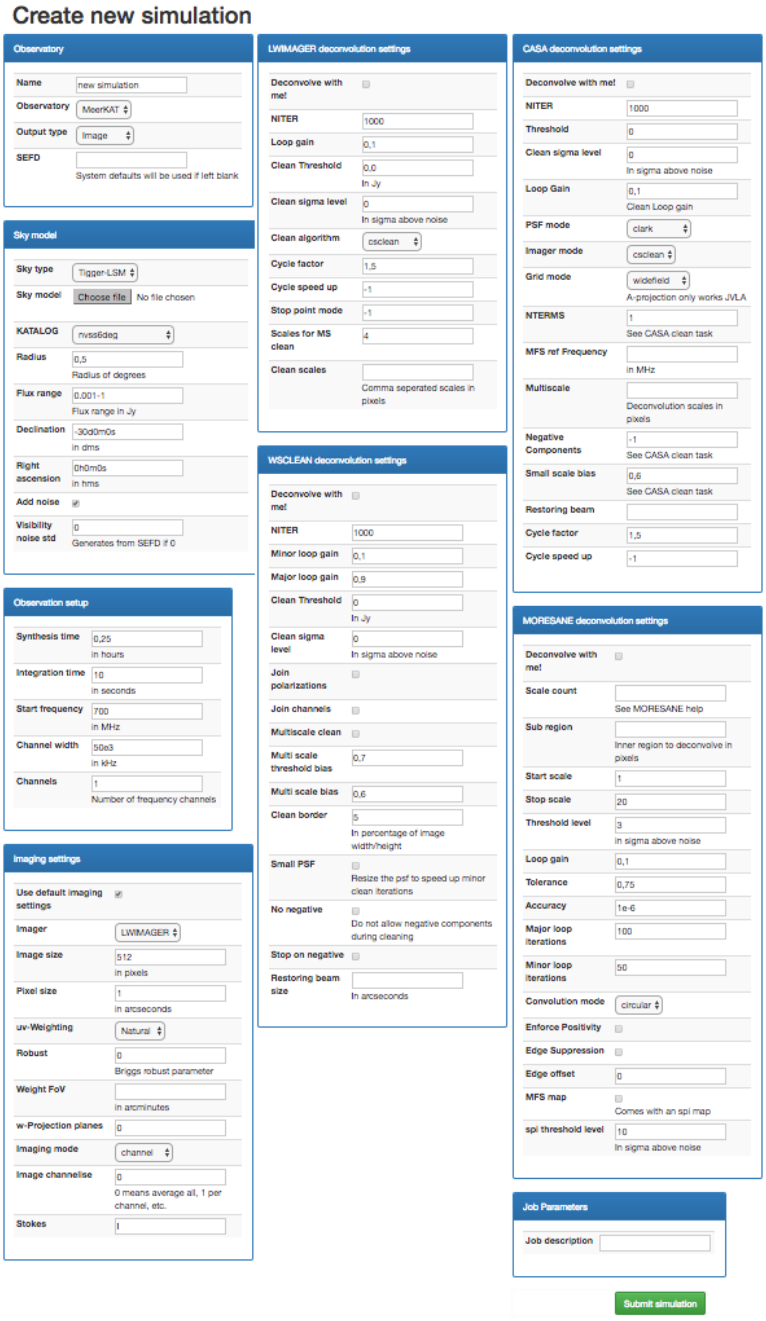


Figure 4.4: Screenshot of RODRIGUES, parameter form is generated from Kliko definition.

4.6 Software availability

Kliko and the Kliko library are open-source, licensed under the GNU Public License 2.0[16]. Kliko is bundled with an extensive test suite, which covers 80% of the source code as of the current release `0.6.1`. The Kliko library is written in Python and is compatible with Python 2.7, all Python 3 versions and even PyPy. Development and distribution are done on Github, and a third-party continuous integration service runs the full test suite on all supported platforms for every commit and every Github pull request.

4.7 Discussions and prospects

Limitations

While developing Kliko, we encountered various problems with Docker, which might limit its applicability. It is up to the user to decide if this affects the usefulness of Docker and Kliko. These issues are listed hereafter for the user's consideration. That said, the field of containerisation is evolving very fast, and it is hoped that, most of these issues will be resolved soon or can be worked around.

First of all, being able to run a Docker container on a system is very similar to giving the user administrative access to the machine; that is, the user can escalate quickly to root privileges (Bui, 2015)[17]. The singularity containerisation technology has a more secure design, and we are planning to add support for this framework in future versions of Kliko.

In addition, using GPU acceleration with NVidia hardware is not trivial, since the kernel driver version and library version need to match up, breaking the independence between host and container. There is a workaround available, but this requires a replacement of the Docker daemon with a custom one[18].

A similar issue arises with optimisation flags. For example, SIMD instructions can significantly enhance the run-time speed, but not all x86 processors support all SIMD optimisation. Enabling these optimisations will result in crashes of the binary if it is compiled with optimisation not supported by the host. Again, this breaks the platform independence assumption. A good strategy is to be conservative and compile binaries for the oldest architecture one intends to support. The good news is that it is easier to support multiple target platforms in the same binary when using modern versions of GCC [19].

Another issue is that it is easy to inherit Docker definitions from other Docker definitions, but it is currently not possible to combine Docker definitions or to inherit from multiple Docker definitions at the same time (merge). Following the Docker philosophy, Docker containers should have a single responsibility, so this should not be a problem; it does not require mixing of Docker definitions. However, in practice, this is not always possible: sometimes various big libraries need to communicate in the same memory space, so a new Docker container with all software needs to be created. The results are far away from minimal small single-purpose Docker containers.

[16] https://github.com/gijzelaerr/kliko
[17] https://github.com/docker/docker/issues/6324
[18] https://github.com/NVIDIA/nvidia-docker
[19] https://lwn.net/Articles/691932

For network-intensive applications, Docker may be less suitable, since the use of network address translation (NAT) for network isolation (Felter et al., 2015). In this case too a workaround is available by disabling the translation and using the host network stack directly.

Future work

During the development and usage of Kliko we became aware of CommonWL (Amstutz et al., 2016), a more generic approach to describe applications input/output flow and parameters. We will investigatemethods to incorporate CommonWL into Kliko to extend the usability and user base.

At the moment Kliko is designed with other container solutions in mind. Singularity is an alternative that seems to be gaining momentum within the HPC community, since it is more aware and careful of the security implications that result from allowing running containers on a shared infrastructure.

Streaming Kliko

Kliko was born in the field of radio astronomy. Most tools in this field operate on data living on disk. Radio astronomy uses several file formats, for example, Casacore MSs, FITS images and, in some cases HDF5. Using the file system is an easy to understand and technically stable approach, but problems arise when the size of the dataset grows. Emerging telescope arrays such as MeerKAT, followed by SKA phase 1, will result in an exponential growth in data rates. Repeatedly reading and writing data from the slowest medium in a computer – the disk – is not going to scale and a new strategy is needed. Directly streaming data between processing tasks will be required. Although this is already being done [20] in some pipelines, there is no field-wide accepted standard that fits all needs. Our plan of action is to investigate existing solutions being used, investigate industry standards [21], optionally create a sub-specification and open-source reference implementation with support libraries for the most frequently used public languages[22].

Runtime limits

System administrators sometimes want to set limits on resources allocated to a process. Kliko could help here. The Kliko specification could be extended to support defining the (absolute) minimal requirements for an encapsulated process. Kliko could also be useful the other way around; if a Kliko runtime would be resource-limited, the numerical values used for setting up the limits could be exposed using the Kliko interface.

4.8 Conclusions

Kliko is a Docker-based container specification. It is used to create abstract descriptions of the input and output of existing software resulting in Kliko containers. These Kliko containers can be used to encapsulate a single job or can be chained together in a pipeline. In the future, we will probably adopt the CWL standard to extend the interoperability

[20] https://github.com/ska-sa/spead2
[21] https://github.com/google/protobuf
[22] http://arxiv.org/pdf/1507.03989.pdf

with other existing workflow tools. Kliko is written in Python and is open-source and available free to use.

Chapter 5

CWL and Buis

5.1 Introduction

In Chapter 4 we introduced Kliko, a container-based pipeline modelling framework. Unfortunately, shortly after finishing the bulk of the work, it became apparent that a very similar solution already existed. This project is called the CommonWL. CommonWL is an open standard for describing analysis workflows and is similar to the design of Kliko, while being more mature and having a significantly larger user and developer base. The project originates from the bioinformatics field, where it is rapidly growing (Leipzig, 2017). The CommonWL project composed an extensive list of existing pipeline frameworks, which have similar functionality to CommonWL[1]. Since a great variety of workflow frameworks already exist, there is no point in investing time and energy in the enhancement and maintenance of yet another framework. Therefore, we decided to focus on the CommonWL standard and the vast ecosystem of tools around it. Adopting the CommonWL standard would open up a useful repertoire of existing tools for running, monitoring and deploying data reduction pipelines on most, if not all, relevant platforms. In addition, if we could develop new tools that supports the CommonWL language, these tools could process existing CommonWL pipelines.

One of the missing tools in the CommonWL ecosystem was a standalone web-based pipeline scheduling and monitoring tool. Multiple full-fledged frameworks exist, of which Arvados[2] is arguably the most mature. Setting up an Arvados environment is not trivial, and the framework is bundled with its own runner and scheduler. This complicates the integration in an existing environment. A more lean front-end would be more suitable for assisting in small experiments. Our own narrow purpose-focused web interface was created to address this issue. This program, named 'Buis', after the Dutch word for 'tube', is described in this chapter. Buis is free and open-source software, available online[3]. Note that Buis is intended as a prototype software, which might evolve into maturity if the community should decide it is a valuable tool. The project is currently in an alpha stage, and not all planned features have been implemented yet. In particular, data management functionality is virtually non-existent, as data sets need to be set up manually.

5.2 The CommonWL standard

Before elaborating on Buis, it is necessary to explain a little more about CommonWL. CommonWL is not a framework or library, but an open standard for creating pipelines. In that respect, it is similar to the Kliko specification described in Section 4.2. The standard is developed by a multi-vendor working group to guarantee an open standard and open development process. It differs from the Kliko specification mainly in that the Kliko specification expects one to bundle every container with a Kliko file, while CommonWL has an optional configuration statement for specifying a container name. In retrospect, the design of the latter is leaner, since it decouples the container and pipeline specification fully. Moreover, this approach does not require a custom container. A CommonWL workflow can also nest other CommonWL workflows, enabling the construction of a graph of sub-graphs. A CommonWL project consists of one or more CommonWL files and

[1] https://github.com/common-workflow-language/common-workflow-language/wiki/Existing-Workflow-systems

[2] https://arvados.org/

[3] https://github.com/gijzelaerr/buis

optionally other arbitrary files such as documentation or input files. Each CommonWL file has a class and can be written in either YAML or JSON.

CommandLineTool class file

The CommonWL describes a series of abstract units called classes. The most important class is the *CommandLineTool* class (CLT). Just like Kliko, CommonWL is in essence a wrapper around command line applications. A CLT file describes a command line tool and lists all possible arguments and flags.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: echo
stdout: output.txt
inputs:
message:
type: string
inputBinding:
position: 1
outputs:
example_out:
type: stdout
```

Listing 9: Example of CWL command line tool definition

Listing 9 shows a simple CommonWL definition of the *echo* command. This definitions accepts one argument, which is passed on to the echo command. This argument in this case has the *label* 'message'. It produces no output files, since echo by default prints the given argument to *stdout*.

Job file

It is the responsibility of the CommonWL runner (discussed later) to present a pipeline with the relevant input files. These files can be gathered on the console as command-line arguments, in a graphical interface such as Buis, or can be defined in a job file. A job file is usually defined in YAML format and is a mapping from labels to values. Each value can be a primitive type, such as a string, int or a reference to a file. In short, a job file is just a list of values for variables that act as input for the pipeline.

Workflow class file

The workflow CommonWL file brings two or more CLT steps together into a directed acyclic graph (DAG). Each step is a wrapped command line program. If two steps depend on each other because the second step needs input from the first step's outputs, their execution will be performed in serial. If there are no indirect dependencies, the steps could be run in parallel. This execution could even be distributed over multiple machines, but this depends on the CommonWL runner and scheduler used, which will be discussed later.

Runners

There is a wide variety of software that support the CommonWL standard. If the software can parse a CommonWL definition and can run the listed commands, it is called a CommonWL runner. The CommonWL website has compiled a full list of all platforms supporting the standard[4].

Discussing all runners is out of scope for this thesis, but we will describe the most relevant runners below.

CommonWL reference runner. The CommonWL reference runner is maintained by the CommonWL consortium, and was created primarily as a reference implementation. The project follows the CommonWL specification strictly, and will be the first runner to implement changes in the specification. Since this runner is very strict in following the CommonWL standard, it is a suitable runner to develop a pipeline and verify all files are following the standard. The capabilities of the reference runner are limited on purpose, and although parallel execution is supported, there is no support for external schedulers. In production, a more advanced runner with support for more multi-host and industry-standard schedulers such as Slurm can be desired.

Toil. Toil is a popular CommonWL runner, since it easy to install and requires minimal to no post-installation setup. The software is purely Python and is installed using the pip installer. In addition, it offers support for various schedulers and distributed computing platforms. This list includes among others AWS, Azure, GCP, Mesos, OpenStack, Slurm and PBS/Torque.

Rabix Composer. Rabix Composer is an open-source editor for CommonWL documents. It has a graphical wizard-like mode allowing drag and drop creation of workflows of individual tools. Figure 5.1 shows a screenshot of Rabix.

5.3 Buis – the web-based frontend for CommonWL runners

Together with introducing the CommonWL standard to astronomers for pipeline deployment, the need for a graphical parameter input method emerged. Astronomers wanted a quick way to deploy an existing pipeline, change the parameters and dataset, and initiate the processing. Unfortunately, such a tool was not available yet. We decided to compile a list of minimal functional requirements given below.

Functional design

When building a web-frontend, it is tempting to build a full-featured product with numerous additional features. For Buis we wanted to keep the design neat and simple, reuse existing tools and frameworks where possible, and try to provide an addition to the CommonWL ecosystem. Buis was created with emphasis on the following requirements:

- **Distinct administrator and user roles.** There should be a clear split between administrator activities and user activities. Administrator activities are related to

[4] https://www.commonwl.org/#Implementations

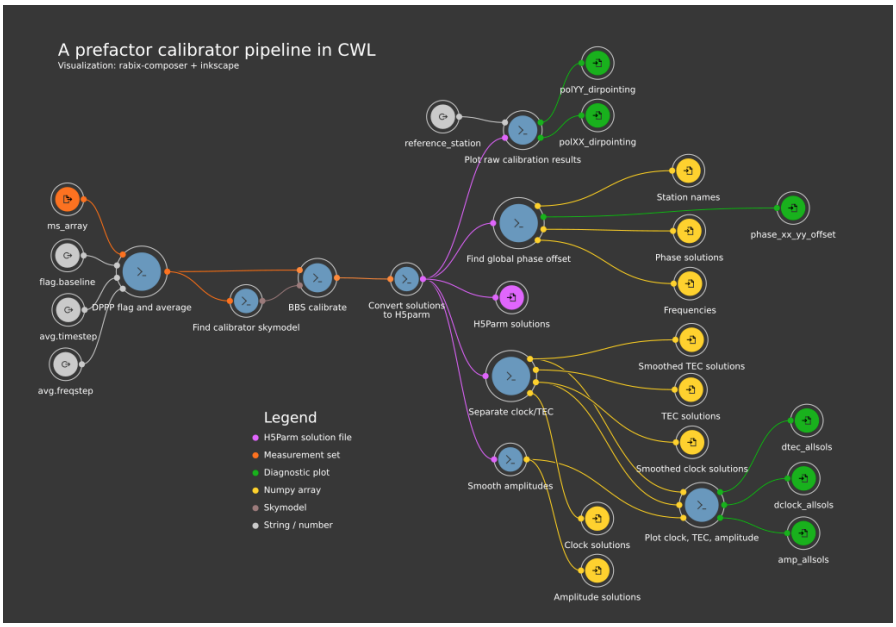


Figure 5.1: A screenshot from the Rabix composer, made by Tammo-Jan Dijkema, ASTRON

configuring the interaction with the CommonWL runner, or exposing datasets that are too big in size to upload manually through a web interface.

- **CommonWL runner configurable by administrator.** A custom CommonWL runner should be configurable by the administrator by means of a configuration file. The CommonWL runner can be configured to handle containerisation backends such Singularity and Docker, but also make use of supported distribution schedulers, such as Slurm.

- **The user role functionally should be fully web-based.** The software should be accessible, and all features should be usable through a web browser. The administrator activities could be a mix of web interface action and command-line actions or configuration scripts.

- **A multi-user authenticated environment.** The software should have an authentication procedure, where optionally, some aspects from the program are publicly accessible. In addition, it should be possible to have multiple users.

- **Checkout existing pipelines from Github.** There are countless ways a user could upload or define a pipeline, but for the first release of Buis, we will focus on a git-based workflow. Users will have to create or modify a pipeline on their own machines and put all pipeline files into a git repository. This git repository should be publicly available, from where Buis would be able to download it. The repository acts as the central place for distributing changes. These changes should be pushed by the astronomer to the repository, from where Buis can be triggered to pull the changes. Since astronomical datasets are typically too large in size to be contained in a git repository, these datasets need to be exposed to the workflow by a different route, discussed later.

- **Scheduling of a CommonWL pipeline to a CommonWL runner.** The software should be able to submit a parameterised pipeline to the configured CommonWL runner.
- **User-configurable job parameters.** The software should be able to inspect a given pipeline and extract the parameters. These parameters should be exposed to the user, where the user can change their default values. These tuned parameters should be stored for later reuse.
- **Ability to monitor pipeline status and progress while running.** After a user has submitted a parameterised pipeline to the runner, the user should be able to visualise the progress.
- **Ability to stop and delete failed, successful or stopped runs.** The users should be able to clean up previous runs by cancelling and/or deleting a run.
- **Configuration of dataset functionality.** The administrator should be able to configure multiple files or directories, which can be used as inputs to the pipeline. Since managing large datasets is a complex task depending on available storage managers and business logic, we decided that for now, a Buis user needs to request a Buis administrator to add a large dataset to Buis manually.
- **Visualisation of compute graph.** The software should be capable of visualising the graph of the CommonWL pipeline.
- **Visualisation of pipeline output.** The software should be capable of visualising the output of the pipeline run. The most common data formats should be supported. For the first release of Buis we will focus on radio image astronomy formats, foremost the FITS format.

Technical design

Buis is similar to RODRIGUES in its design (see Section 4.5). The most significant difference is that Buis only supports CommonWL pipelines, while RODRIGUES only supports Kliko. Since Kliko is not maintained anymore, RODRIGUES has also been abandoned.

- **Django.** Django[5] is the web framework used to create Buis. The framework is based on Python, is mature and contains most of the features needed. These include the creation of models for storing temporary results, authentication and authorisation of users, a useful administrator panel for managing users, in-browser debugging facilities, a bundled HTML templating engine, and more. Django has a large user community writing reusable Django components (apps), which could be reused and recombined with other software. Django can be modified and reconfigured to integrate with other supported tools by the Buis administrator.
- **Database.** A metadata storage location for keeping track of the Buis internal state is required. For storing meta-data, a database back-end is required. Django supports multiple back-ends. An example of a backend is the high-performance PostgreSQL[6] backend, which is a popular open-source service-based database en-

[5] https://www.djangoproject.com/
[6] https://www.postgresql.org/

gine. Another supported backend is the file-based SQLite[7] environment, which does not require a running daemon. Since it is not expected that more than a few simultaneous users will be accessing Buis, the storage configuration of Buis can be kept simple, and the non-service file-based SQLite solution is used. Other databases can be used if required and can be configured in the Django settings.

- **Celery.** Celery[8] is the job distribution and management platform we use which integrates neatly with Django. Celery is commonly used for websites where it is important to return a response to a website visitor as quickly as possible to guarantee a smooth user experience. Non-critical tasks that take longer, such as updating an extensive database, are not always required to be processed and executed before a response is sent back to the client. By executing the job asynchronously using Celery, the website stays responsive. In present case, the tasks we schedule asynchronously are the cloning and updating of Github repositories and running the actual pipelines. When a user starts a pipeline, the tasks are delegated to the Celery framework, which will forward the request to a broker service controlled by Celery. In the background, multiple Celery workers are monitoring the broker queue for new tasks. When one comes in, the worker executes the job, and will update the database when the job finishes or fails for whatever reason. Django will then visualise these updates on the website, so the user stays up to date about the status of the job.

- **Bootstrap.** For the frontend in the browser, we use Bootstrap[9], which is a set of CSS rules and icons that make visually attractive and user-friendly websites. Also, Bootstrap is a widely used platform and an industry standard.

Usage

As discussed before, there is a distinction between a Buis administrator and a Buis user. The Buis administrator sets up the Buis installation and modifies the configuration to match the deployment environment. In the role of an administrator, there are multiple ways to deploy Buis. The best way is dependent on the requirements on the deployments. We have added a `Makefile` with shortcuts that act as examples on how to proceed. The `Makefile` should be examined for all the commands and options. The main options are:

- **A manual deployment.** The manual strategy is useful for development. This strategy requires the installation and configuration of a message broker. During the development of Buis, the RabbitMQ[10] broker, a fast open-source message broker, was used. This broker is also configured as the default broker. Next, one needs to manually start a webserver (`make django-server`), which will accept web requests from the client. Note that one also needs to initialise the database (`make django-migrate`) and create a user for authentication on the website (`make django-createsuperuser`). To be able to schedule the jobs in the background, a Celery worker also needs to be started (`make celery-worker`).

[7] https://www.sqlite.org/
[8] http://www.celeryproject.org/
[9] https://getbootstrap.com/
[10] https://www.rabbitmq.com/

- **Using docker-machine.** If one has Docker installed on one's system, one can use docker-machine[11]. This is a more robust strategy, and suitable for a quick, permanent deployment. First, one has to start with the `make docker-compose-up` command, which will create and download the required Docker containers, followed by a start of all the defined services, including the Celery worker. Next, one needs to also initialise the database and create a user with `make docker-compose-migrate` and `make docker-compose-createsuperuser`. For further details, for example of how to start these services during booting of the machine, users are advised to read the docker-compose manual.

- **A advanced manual deployment behind a webserver.** This is the recommended permanent setup if one wants to integrate Buis into one's existing infrastructure. The details of this deployment are beyond the scope of this thesis. If this strategy is of interest, the user should read to read the Django deployment manual[12].

When a user is created by the administrator, this user can navigate to the Buis web server using a browser and the credentials used during the creation of this account. The user is then confronted with three main menu options, *Repositories*, *Workflows* and *Datasets*. The repositories are the git repositories containing CommonWL pipelines. The user should add one or more git repositories here, and once added, the user can select a pipeline, load in parameters and start a workflow job. The workflows page lists all running and completed pipeline runs. The datasets page lists all configured datasets added by the administrator. As discussed before, for now, if a user wants to add a dataset to Buis, this needs to be coordinated with the Buis administrator.

5.4 Use case example: a 1GC pipeline

To demonstrate the functionality of Buis, a simplified version of an existing science pipeline is taken and converted to CommonWL. This pipeline is based on a small subset of the CARACal (Józsa et al., 2020b) workflow (Subsection 1.11), which uses Stimela. Stimela can transpile (compiler from source to source) a Stimela script into the CommonWL format. The CommonWL transpiled 1GC pipeline is available on github[13].

The pipeline performs 1GC (Subsection 2.4), a.k.a. reference calibration (see Section 2.4), on a given dataset. The calibration procedure consists of multiple sub-tasks, each either modelling and solving for a particular gain effect (se eq. 2.57), or manipulating the results. The calibration is based on a nearby bright reference source observation with known properties.

The CARACal transpiled pipeline consists of the parts listed below. Note that, from a functional point of view, most of these are wrappers around the eponymous CASA tasks:

- **Listobs:** This step is a wrapper around the CASA `listobs` task. This task is diagnostic in nature, and does not manipulate the incoming data: the only action taken is printing a summary of the data to the console.

[11] https://docs.docker.com/machine/
[12] https://docs.djangoproject.com/en/3.0/topics/install/#install-apache-and-mod-wsgi
[13] https://github.com/gijzelaerr/1gc-pipeline

- **Setjy:** This is the first step of the absolute flux calibration. Here the `MODEL_DATA` column of the MS is filled with the visibilities corresponding to the calibrator sources. This is the $\boldsymbol{M}$ term in eq. 2.56.
- **Delay calibration:** The first Jones term to be solved for is usually delay ($\boldsymbol{\Gamma}$ in eq. 2.57). Multiple phenomena can cause delays in signal propagation, causing a mismatch in correlating the signals. The most dominant corrupting effects are caused by the atmosphere, inaccuracies in the electronics and inaccuracies in the geometric model of the array. Regardless of how well the array hardware is produced, modelled and maintained, the the inaccuracies in electronics and the geometric model are unavoidable and should always be accounted for. Since the effect of delay is a linear ramp in phase, unaccounted-for delays will influence all other Jones solutions downstream, therefore this effect needs to be solved for first.
- **Complex gain calibration:** Incoming signals are time-variable owing to atmospheric and environmental influences, causing fluctuations in mostly the phase but also the amplitude of the gain. Gain calibration mitigates the large fluctuations, which increases the coherence of the signal. This is the $\boldsymbol{G}$ term of Eq. 2.57.
- **Bandpass calibration:** This calculates a bandpass calibration solution, i.e. the frequency-variable $\boldsymbol{B}$ term in Eq. 2.57. Multiple external conditions can influence the receivers' response to an incoming signal and each effect influences the response per frequency differently. The effects are specific and unique to the array itself, but are also affected by external influences such as temperature. The bandpass calibration step is used to estimate and compensate for these influences (using a known calibrator source frequency spectrum, which we set up at step `setjy`).
- **Plot_bandpass:** This is another diagnostic step, which plots the bandpass solution found in the previous step. An astronomer can use these plots to identify any potential problems with the observation.
- **Plot_gains:** This is another diagnostic step, taken for the same purposes.
- **Fluxscale:** This is the second step of absolute flux calibration. Note that bandpass calibration, above, requires a very bright reference source with a well-known flux and spectrum. Very few such calibrator standards exist, so it is rare to have one suitably close to the observational target. This usually means a large difference in atmospheric phase between the two, making the bandpass calibrator unsuitable for gain calibration. Typically, this is overcome by selecting a 'secondary' calibrator source close to the target. Provided this secondary calibrator is sufficiently bright and point-like, complex gain calibration can be done using the secondary without knowing its absolute flux. The unknown absolute flux scaling is then corrected for by transferring the flux scale from the primary to the secondary solutions, using this task.
- **Applycal:** This step applies calibrations solutions derived from the previous calibration steps to visibilities in the MS, forming up the so-called *corrected visibilities.*
- **WSClean:** The final step is imaging (Sect. 2.3). Here, this is done by the WSClean task (a wrapper around the eponymous software package[14]). WSClean uses the

[14]https://sourceforge.net/p/wsclean/

corrected visibilities and the *uv*-sampling to produce a dirty image and a PSF, then applies one of several possible deconvolution techniques to derive a deconvolved ('clean') image.

To run the pipeline, the user adds the public 1GC repository on the repository page. A Celery worker will clone the Github repository on the machine running Buis and will update the status page when ready. Next, the user can explore the content of the repository. Buis will list all the CommonWL files in the repository (Figure 5.2), and give the option to visualise the pipeline or prepare a workflow run. Figure 5.3 shows how Buis visualises the full 1GC pipeline.

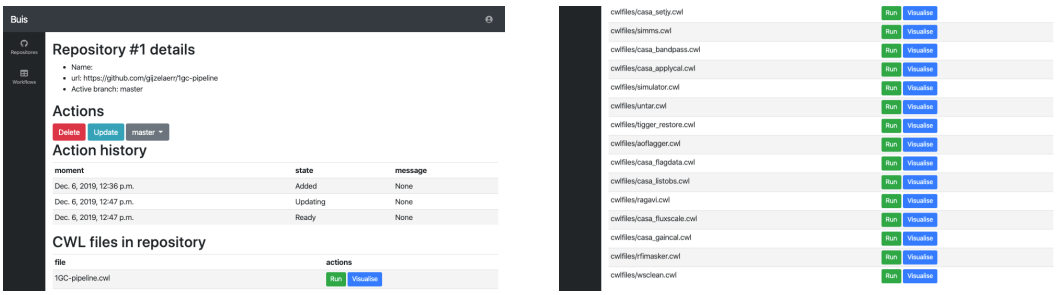


Figure 5.2: The repository view of the 1GC pipeline project

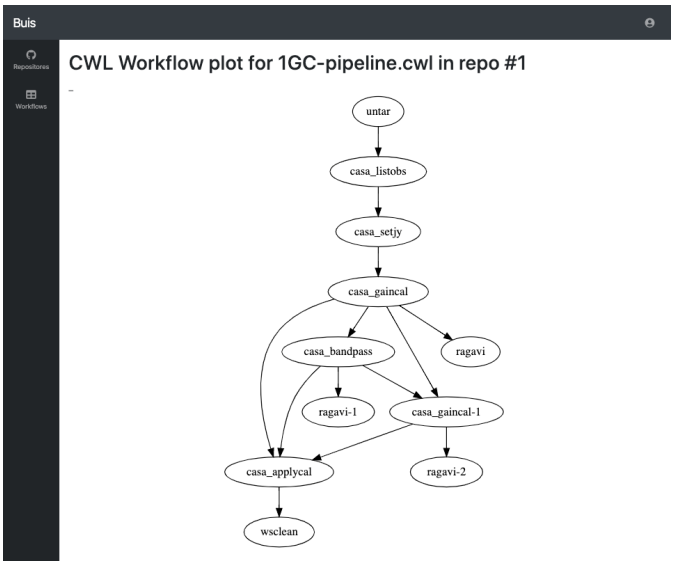


Figure 5.3: A graph representation of the 1GC pipeline

Note that Buis presents the user with all CommonWL files. The 1GC repository contains multiple CLT files, and one *Workflow* file, the latter chaining the former into the 1GC workflow. When the user proceeds to the workflow job preparation step, Buis will list all job files in the repository containing presets for the variables in the pipeline. The user can select a preset or choose to start with the default values defined in the workflow. Next, the user is presented with all the parameters, which can be modified when needed (see Figure 5.4). If the pipeline has files or directories as input, the user can attach prepared datasets here.

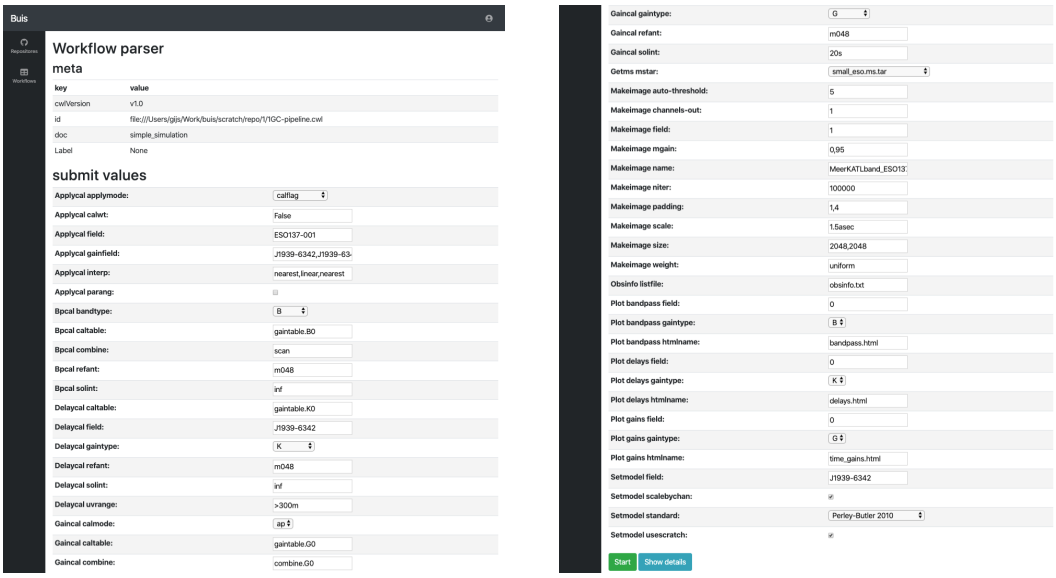


Figure 5.4: The Buis workflow parser view, the user can change parameters here.

Finally, the user can initiate the workflow run. When the run button is pressed, the job is submitted by Celery to the CommonWL runner, which will deploy the pipeline depending on the setup configured by the administrator. Once the run is complete, Celery will update the website with all the command output and output files. Figure 5.5 shows the final page listing all output data products. Note how Buis offers support for astronomy-specific files such as FITS.

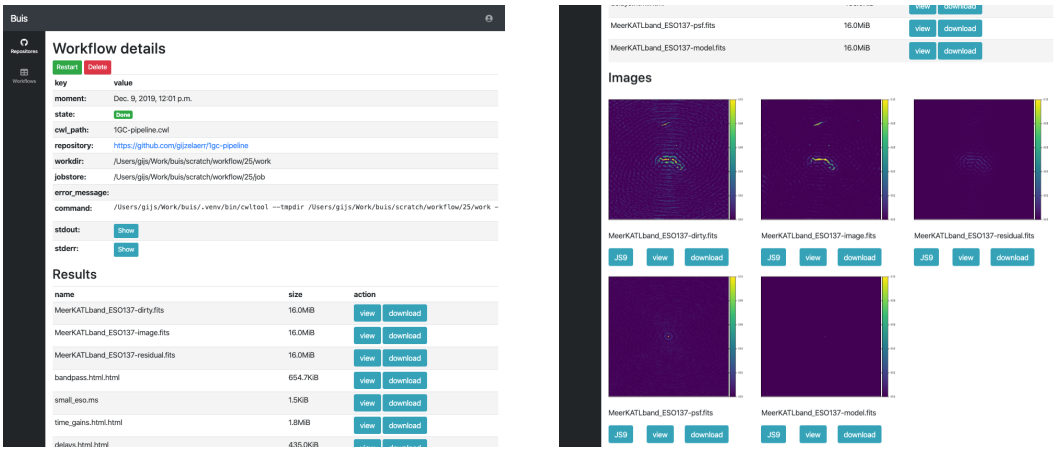


Figure 5.5: The Buis workflow detail view for the 1GC workflow run

5.5 Discussion

In this chapter, we presented the Common Workflow Language, a workflow abstraction framework. We also described Buis, a CommonWL pipeline scheduling web framework and result set visualiser. Buis is a prototype, but this does not imply that it is not ready for use nor that it cannot grow into a complete project. During the writing of this

chapter, we became aware of the release of a new but similar tool named CWLab[15]. In design Buis and CWLab seem similar, but a feature that is missing from CWLab is a result viewer. Since both projects are open-source, future work could involve comparing the two frameworks and see if concepts can be copied in one way or another. If all parties are interested, it might even be fruitful to combine the work efforts and merge the project.

[15] https://github.com/CompEpigen/CWLab

Chapter 6

Vacuum Cleaner

6.1 Introduction

With the advent of new radio telescopes such as MeerKAT (Jonas, 2015), ASKAP (Johnston et al., 2008), LOFAR (de Vos et al., 2009) and the soon to be built SKA (Dewdney et al., 2009), it would seem that radio interferometry is entering an unprecedented era of expansion. While the increased FOV, bandwidth and angular resolution possible with modern instruments significantly improves the science capability of radio telescopes, it comes at the cost of greatly increased data rates (D. L. Jones et al., 2012). The science, on the other hand, still relies very strongly on our ability to turn the data measured by an interferometric array (known as visibilities) into an image of the sky. As a result, legacy image reconstruction algorithms such as Hogbom's CLEAN algorithm (Högbom, 1974) are continually being optimised and extended (Offringa, McKinley, et al., 2014; Offringa and Smirnov, 2017; Tasse et al., 2018). While these algorithms are computationally inexpensive, it is known that they are sub-optimal at capturing the complex morphologies uncovered by modern radio interferometers. On the other hand, with the recent demonstration of super-resolution (Dabbech, Onose, et al., 2018), it has been realised that sparsity-promoting techniques (Carrillo et al., 2014; Dabbech, Ferrari, et al., 2014; Onose et al., 2016) as well as techniques based on Bayesian inference (Junklewitz et al., 2016) can greatly improve the quality of reconstructed images. This is important in the context of modern instruments, which aim to maximise the the science capability of an instrument built from finite resources. Unfortunately, the computational complexity of these more sophisticated techniques render them impractical to the majority of astronomers who do not have access to supercomputing facilities or to readily available implementations of these algorithms. In addition, with current data sets already easily reaching Terabytes, it becomes more and more impractical to ship the raw data from the storage facility to the user.

Previously mentioned challenges make any machine learning based approach based on interferometric image data extremely attractive. In this chapter we demonstrate that such an approach is indeed feasible and discuss some of the challenges and benefits that it offers. We specifically focus on deep neural networks, since these networks have recently shown impresive results in the domain of computer vision. The technology has been applied to deal with related problems such as image denoising (Fu et al., 2017; Mao et al., 2016; Schuler et al., 2013; Zhang et al., 2017), inverse problems (Adler and Andöktem, 2017; Jin et al., 2017; McCann et al., 2017; Xu et al., 2014), medical (MRI) deconvolution (Mardani et al., 2018). In the radio astronomy field these methods are also gaining traction, for example translation between different radio astronomy surveys (Glaser et al., 2019) and classification of fast radio bursts (Connor and van Leeuwen, 2018).

6.2 Radio interferometric imaging

In Section 2.3 we explained the basic principles of radio interferometric imaging and image brightness reconstruction. This chapter describes a series of experiments performed to reconstruct the original image from an observation using a deep neural network. However, there are a number of complicating factors. Consider that we are essentially trying to infer how to invert the mapping Eq. (2.50) from training data. Clearly, since the problem is ill-posed, it would be necessary infer the correct prior with which to regularise the problem. This however, is not the main difficulty. The main complication that arises is

that the (u, v, w) locations at which we have measured data changes from observation to observation and it is simply not feasible to train a separate network for each observation. The lack of sufficient realistic training data also makes drowning the problem in training data infeasible. A practical solution would therefore have to incorporate knowledge of the sampling pattern during training and prediction, and also perform well with relatively little training data. The approximation Eq. (2.53) suggests an attractive possibility that circumvents many of these complications.

We could utilise the expression Eq. (2.53) during training of a neural network using a loss function looking like:

$$\Phi(I) = \left(I^\dagger I^{PSF} * I - 2I^\dagger I^D \right), \tag{6.1}$$

with a network that has the dirty image I^D and PSF I^{PSF} as inputs, I is the model and $I^\dagger$ is the ground truth. Loss functions will be discussed further in Subsection 6.3. Note that by using the PSF as one of the inputs, we are effectively incorporating knowledge of the sampling function during both training and prediction. Furthermore, by incorporating the notion of convolution with the PSF as part of the loss function, we are informing the network of the relation between the dirty image and the PSF, albeit only approximately and only during the training phase. This should drastically reduce the training time of the network and improve performance in cases where the sampling pattern changes from observation to observation.

6.3 Method

A deep neural network is a machine learning method based on artificial neural networks (ANNs). Such a network is inspired by the biological neural network of animal brains, although current ANNs are not even close to approaching the complexity of natural neural networks (Barrett et al., 2019; Geirhos et al., 2017; Sinz et al., 2019). An ANN consists of multiple nodes or neurons, where subsets of the nodes are grouped into layers. These layers are connected to one another by edges. Each node has one or more inputs and outputs. In operation, each neuron receives signals from the connected input nodes, and the output is computed by applying a non-linear function of the sum of the weighted inputs. In this paper we explore convolutional neural networks (CNNs). CNNs are a specific type of multilayer perceptron. Multilayer perceptrons usually refer to fully connected networks, while CNNs are typically not fully connected. These networks are sensitive to overfitting and are computationally expensive, since their number of connections between layers grows exponentially. As the name suggests, CNNs employ a mathematical operation called convolution. In the deep learning context, convolution means that a node is only connected to neurons in the previous layer that is spatially nearby.

Before we can use the network, it requires training to make sure it maps the correct output to the given input. Training a neural network is accomplished by adjusting the weights inside the network. For weight optimisation, we use an algorithm called the adam optimiser (Kingma and Ba, 2014). During training, the network is given an input image, which will propagate through the network and generate a prediction. This prediction is compared to a ground truth using a loss function, which will be described later. The loss function indicates how wrong the prediction is. The adam algorithm estimates how much each weight contributed to this error, and will update the weight accordingly. Training is done multiple times on the same dataset, where each iteration is referred to as an epoch.

Network architecture

Most neural network frameworks have a notion of channels that are used to encode colour information. Since, in this context, we only use a singly frequency, we only use one channel (grayscale). For the base of our network, the autoencoder architecture is used. An autoencoder is a type of NN used to learn efficient data codings in an unsupervised manner. The aim of an autoencoder is to learn a dimensionality reduction representation (code) of a set of data. Along with the reduction side, a reconstruction side is learnt, where the autoencoder tries to generate a representation as close as possible to its original training target from the reduced encoding. Intuitively an autoencoder can be seen as a universal dimensionality reduction tool, similar to PCA analysis and compression techniques. The method projects the original data in a feature space that facilitates sorting by the most significant shared features of the input data. Next, the dimensions representing the less prominently shared features (which are assumed to be noise) can be discarded, reducing the computational storage requirements, but, more importantly, and it is to be hoped, removing the noise or artifacts. We call the reduction side the encoder function ϕ and the reconstruction side decoder function ψ. The input image and target output image are directly fed into these functions during usage. Also, the input of the encoder matches the shape of the input image, the dirty image.

A spatial discrete convolution is defined as:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2]\dot{g}[x-n_1, x-n_2]. \tag{6.2}$$

Each layer $f[x,y]$ in the encoder is a spatial convolution layer with a normalised kernel $g[x,y]$ and stride (how far the filter moves every step in one direction) of 2, resulting in each consecutive layer having half the neurons in each spatial dimension. The decoder has exactly the opposite structure, and the final layer matches the dirty spatial image dimensions.

During training, ϕ and ψ are optimised such that:

$$\begin{aligned} &\phi : I^D \to \mathcal{F} \\ &\psi : \mathcal{F} \to I^M \\ &\phi, \psi = \arg\min_{\phi,\psi} \|I^D - (\psi \circ \phi) I^M\|^2, \end{aligned} \tag{6.3}$$

where $\circ$ is a function composition and $\mathcal{F}$ is the dimensionally reduced representing code. Since we expect high spatial similarity between the dirty and clean images, we use skip connection (direct propagation connections without convolution) between the encode and decode layers, making the network a 'U-Net' architecture (Ronneberger et al., 2015). Here, 'similar' means that there is a positional correlation between input and output pixels. Input pixels in one corner of the image correlate with output pixels in the same corner, but much less (although not zero) with output pixels in the opposite corner. The rationale behind adding these skip layers is that the network has additional information about the input data and can converge faster. The architecture of the Vacuum Cleaner network is shown in Fig. 6.1.

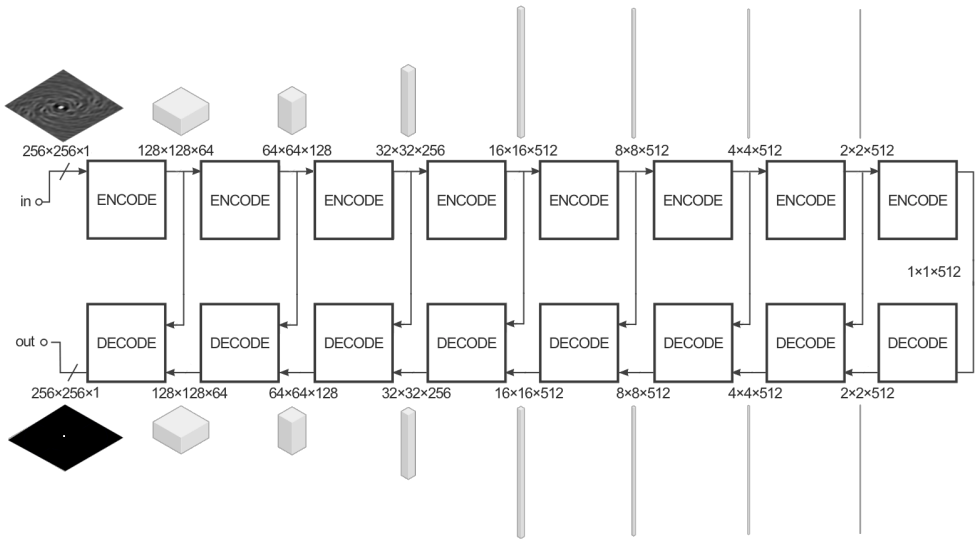


Figure 6.1: The architecture of the network

Objective function

To update the weights in the network, an objective function is required. This could be any function, as long as it is differentiable. Usually, this function represents the distance S between the target value (y_i) and the estimated value $f(x_i)$. Commonly used definitions for distance are the L_1-norm, also known as the Manhattan distance:

$$S_{L_1} = \sum_{i=1}^{n} |y_i - f(x_i)| \tag{6.4}$$

and the L_2-norm, also known as the Euclidean norm:

$$S_{L_2} = \sum_{i=1}^{n} (y_i - f(x_i))^2. \tag{6.5}$$

Using these norms as an objective function often results in blurry output images (Pathak et al., 2016). To address this issue, we use a conditional generative adversarial network (GAN) (Goodfellow et al., 2014). GAN's consist of two networks, the generator and discriminator. The generator generates the images, while the discriminator is trained to distinguish between real and fake data. We use the previously described U-net as the generator. The judgement of the discriminator is used as an objective function to train the generator. Eventually, the generator should produce outputs that cannot be distinguished from the training data by the discriminator. The discriminator network evaluated has a PatchGAN architecture (Isola et al., 2016), is only used during training and acts as a loss function.

Conditional GANs learn a mapping from a dirty image I^D and a random noise vector Z to clean image I^M:

$$\epsilon : I^D, Z \rightarrow I^M, \tag{6.6}$$

where ϵ is the generator. The network could learn a mapping from I^D to I^M without noise vector Z, but this would produce deterministic outputs. Consequently, it would fail to learn anything except a delta function. The noise is provided in our GAN model in the form of dropout (Srivastava et al., 2014). The discriminator δ is trained to score how likely it is that a dirty image I^M was generated by ϵ or is real data from the training set.

A GAN can be seen as a two-player minimax game (Aumann and Maschler, 1972) with loss function $L_{gan}(\epsilon, \delta)$:

$$\begin{aligned} G* =& \min_{\delta} \max_{\epsilon} L_{gan}(\delta, \epsilon) \\ =& \mathbb{E}_{I^D, I^M}[\log \delta(I^D, I^M)] + \\ & \mathbb{E}_{I^D, Z}[\log(1 - \delta(x, \epsilon(I^D, Z)))]. \end{aligned} \tag{6.7}$$

where $\mathbb{E}$ is the expectation, or the weighted average over the values in subscript.

δ is trained to distinguish between training examples and generated samples from ϵ. Simultaneously, ϵ is trained to minimise $\log(1 - \delta(\epsilon(Z)))$, which means it tries to reduce the performance of the discriminator.

Previous experiments have shown that it can be beneficial to regulate the loss function with the L1 distance between the output of ϵ and the training data I^M (Isola et al., 2016; Pathak et al., 2016), making the final objective:

$$G* = \min_{\delta} \max_{\epsilon} L_{gan}(\delta, \epsilon) + \lambda L_{L1}(G), \tag{6.8}$$

where λ is a hyper-parameter that can be fine-tuned.

In this chapter we will describe using a conditional GAN applied to the image reconstruction in radio astronomy. In addition, we will investigate if adding additional information in equation Eq. (6.1) described in the introduction of this chapter, to the loss function will improve the results.

Implementation

Our work is based on the open-source tensorflow (Abadi et al., 2016) implementation of the pix2pix network (Isola et al., 2016). The modified network is available on github under the name Vacuum Cleaner[1]; it is also available on pypi[2].

We have modified the code to make it easier to load and write astronomical (FITS (Wells and Greisen, 1979)) data. The network operates internally with pixel values in a fixed range between -1 and 1. Meanwhile, astronomical images can have a huge dynamic range so FITS files do not have a predefined range. For this reason we preprocess and scale the dirty image to the $(-1, 1)$ range, and uses the same scaling factor to restore the original scale for the output of the network. Moreover, we added the option to give the PSF as an additional channel input.

Training

To train a deep neural network, it is important to use the correct learning rate. Setting this too low will slow down the learning process to impracticable levels, while setting it

[1] https://github.com/gijzelaerr/vacuum-cleaner
[2] https://pypi.org/project/vacuum-cleaner/

too high might overstep solutions. Unfortunately, the optimal learning rate might change when the network architecture and/or loss function is modified. This implies that a small hyper-parameter search is required for every experiment to find a good learning rate. We found that most experiments in this chapter converged in scoring at a learning rate of 10^{-3}, while adding the likelihood term to the loss function performed better at a learning rate of 10^{-5}. Setting the learning rate too low tends to make the learning slower, taking days rather than hours to converge, or not even converge at all. Setting the rate too high leads to erratic learning behaviour, where the scores keep increasing and decreasing chaotically while training.

6.4 The simulation

We train the network in a supervised manner, which requires a training dataset with labelled data. A large dataset of dirty images I^D, PSF I^{PSF} and the corresponding true sky model I^M are needed.

As mentioned earlier, imaging radio astronomy data results in the dirty image, which (for a small FOV) is the convolution of the PSF and the observed sky emission. Therefore, to get the sky emission one has to *deconvolve* the PSF from the dirty image. This deconvolution process in radio interferometry is an under-determined problem, which means that sky emission recovered from this process is going to be dependent on the deconvolution algorithm used to obtain it. This makes it impossible to obtain an unbiased (or algorithm-independent) training dataset from real observation data. For this reason, we use a simulated training dataset for which we know the sky emission exactly.

Fortunately, since the mapping from sky to visibilities is known and given by Eq. (2.50), it is possible to simulate as much input data as required. The simulation procedure consists of generating an artificial sky model and computing the visibilities corresponding to the model using Eq. (2.50). From the visibilities and the *uv*-coverage we are then able to produce the corresponding dirty image and PSF, which serve as training data.

The simulation uses the telescope simulator tool `simms`[3] and `MeqTrees` (Noordam and Smirnov, 2010) to simulate a model of the sky emission as well as thermal noise contribution; telescope gains can also be simulated but are not considered in this work. The full simulation pipeline, called `spiel`, is open-source and available online[4]. `spiel` is a system agnostic telescope simulator written in CWL (Amstutz et al., 2016) in order to perform large-scale simulations for modern radio telescopes.

The pipeline output products are:

1. Simulated telescope data; visibility data in the CASA measurement set (MS) format (Kemball and Wieringa, 2000).

2. The dirty image of the simulated data

3. The PSF image of the simulated data

4. Deconvolution image products from WSClean, which can be used as a baseline. These are:

   model image: The `WSClean` reconstruction of the sky emission

[3]https://github.com/SpheMakh/simms
[4]https://github.com/gijzelaerr/spiel

residual image: This is an image of the data after subtracting the reconstructed model.

restored image: The residual image added to the convolution of the model image with a 2D Gaussian fit of the main lobe of the PSF (a.k.a clean/restoring beam).

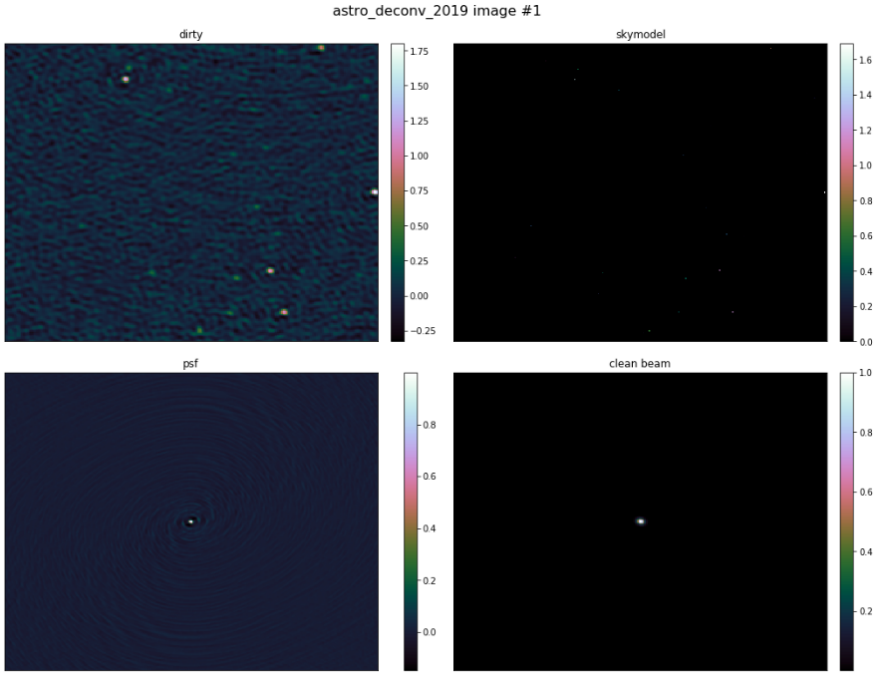


Figure 6.2: Bundle 1 of the ASTRODECONV2019 dataset

Using spiel, we generated 10 000 image products with unique sky emission models, henceforth referred to as sky models. This dataset is released under the name ASTRODECONV2019 and is available online[5]. We encourage future experiments and benchmarks with this dataset. The dataset has a total size of 12 GB compressed and 23 GB uncompressed. To save space and bandwidth, the raw measurement set data and residual images are not distributed. The latter can be reconstructed if needed by convolving the WSClean model with the PSF and subtracting the dirty image. Figure 6.2 visualises image bundle number 1 from the dataset. The simulated data is based on the MeerKAT telescope AR1 phase (16 antennas)[6] with an effective bandwidth of 770 MHz, though in this simulation we used bandwidths of 200 - 300 MHz only. Larger bandwidth would require considerable spectral variation in the sources, which is outside the scope of this work.

In this initial investigation, we restrict the sky models to consist of unresolved point sources randomly scattered across the FOV. All simulations are based on the 16 dish configuration of the MeerKAT telescope, but the pointing location and integration time vary randomly. The varying position and integration time result in diversity in noise levels in the images and PSF shapes. Also, the channel width varies between 200×10^6 and 300×10^6 and declination varies between $-55°$ and $-65°$.

[5]https://archive.kernsuite.info/data/

[6]http://www.ska.ac.za/wp-content/uploads/2016/07/info_sheet_ar1_2016.pdf

6.5 The results

Scoring

To evaluate the performance of the proposed method we compare the output to the output of a widely used cleaning implementation named WSClean. However, to be able to quantify and compare performance we first need a scoring method.

Consider that an interferometric array necessarily has a finite resolution, which is determined, predominantly[7], by the antenna pair with the largest separation. As a result, it is not usually possible to distinguish between a single source or a group of sources lying within the resolution element of the instrument. Therefore, a straightforward comparison based on the norm of the differences between input I^M and output $\hat{I}$ images is not very informative. In order to score the output of the network we therefore rather compute the norm of the difference between the input and output images once they have both been convolved with a kernel, I^α say, which matches the resolution element of the interferometer i.e.

$$E_1 = \|\hat{I} * I^\alpha - I^M * I^\alpha\|_1. \tag{6.9}$$

In practice I^α is obtained by matching a 2D Gaussian to the full width half maximum of the main lobe of the PSF. In addition, to test how closely the output matches the data, we also compute the RMS of the residual image obtained as follows

$$E_2 = \sqrt{\sum_i (I_i^R)^2}, \quad \text{where} \quad I^R = R^\dagger \Sigma^{-1}(V - R\hat{I}), \tag{6.10}$$

where the operator R is implemented with WSClean. We have precomputed the clean beams, which are bundled with the ASTRODECONV2019 dataset. For fairness the major cycle count of WSClean has been restricted to 1 by setting the `-mgain` flag to a value of 0.95.

Evaluation

We evaluated three different approaches to train the neural network:

GAN loss function. The first approach is a GAN, as described in Section 6.3.

GAN loss function and PSF as input channel. The second approach is the same GAN, but with the PSF as secondary input channel, next to the dirty image input.

GAN loss function with likelihood term and PSF as input channel. The final evaluated approach is the same as the second approach, but with the additional likelihood term in the loss function. The likelihood term has been manually weighted with a factor 0.001 to balance the two loss functions.

For the experiments we have split the dataset into 94 000 training images, 3 000 testing images, and 3 000 evaluation images. The training images are randomly flipped horizontally and vertically to maximise their utilisation. Each approach is trained for 100 000 steps, and every 10 000 steps the performance is evaluated on the test set. Figure 6.3

[7] SNR and uv-sampling patterns aside

shows the performance on the test set for each method. The numbers are the average values of 10 repeated randomly initialised runs.



Figure 6.3: Comparison of performance loss functions. Average of 10 runs for each evaluated method. On the horizontal axis are the training steps, for every 10 000 training steps the network performance was evaluated using the test set. On the vertical access is the training score, which is computed with Eq.(6.9). Lower is better. The data shows that initially the *GAN average* loss function learns faster, but beyond 40 000 training steps the performance of all methods converge. The degradation in performance around 80 000 steps for the *GAN average* loss function is caused by a sudden drop in performance in one of the runs, which might suggest that the training rate is slightly off for this specific loss function.

In addition, the performance of one of the GAN loss functions is evaluated against the WSClean output computed in the simulation step, which is plotted in Fig. 6.4

Computational performance

We did not do an extended study on the scalability properties of our method. All development and evaluation of Vacuum Cleaner was done on a computer system equipped with an AMD Ryzen 7 3700X CPU, 32GB of memory and a Nvidia Geforce 1080 TI GPU with 11 GB of memory. Deconvolving 100 images takes about 21 seconds on the test system with GPU acceleration enabled. This includes encoding and writing four FITS (dirty, target, output, residuals) files per image. 9 out of the 21 seconds is used for

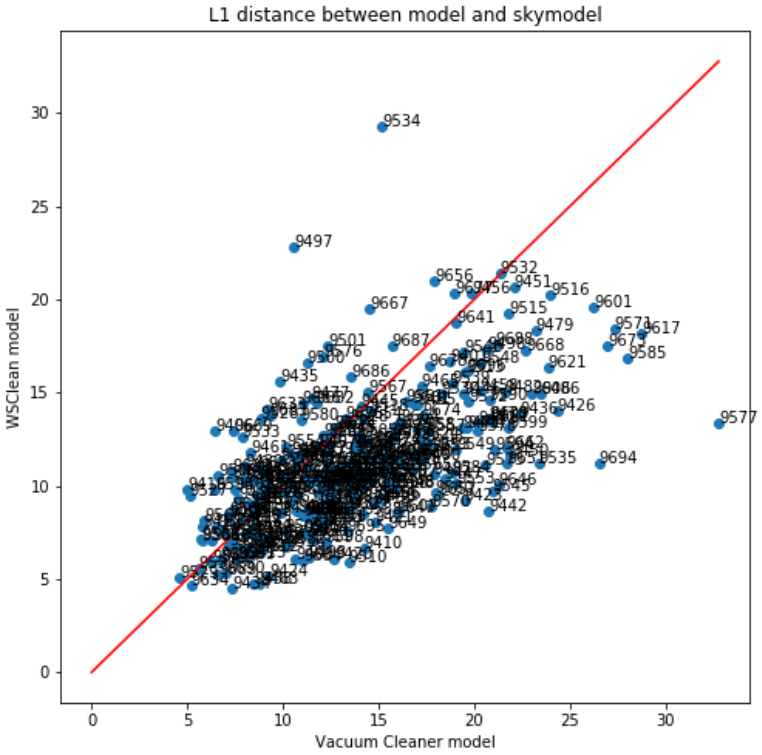


Figure 6.4: Vacuum cleaner performance compared to WSClean. Here, as a scoring function we use Eq.(6.10), so lower is better. Each point represents an image; on the vertical axis the position represents the Vacuum Cleaner error score, on the horizontal scale the position represents the WSClean error score. Below the red line means WSClean performed better, above the red line means Vacuum Cleaner performed better. The centre point of the distribution is in favour of WSClean, but the results look very encouraging and for a significant number of images Vacuum Cleaner outperforms WSClean.

'warming up', i.e. initialising the neural network on the GPU. The CPU-only version is significantly slower, and deconvolves about one image per three seconds.

The expected computational scaling and memory footprint of the problem are quadratic with the total number of pixels. A quick experiment showed that increasing the input and output size to 512 by 512 pixels (by adding a halving layer to the input size of the autoencoder, and a doubling layer to the output layer) is executable on the development machine described, but as expected, slows the performance down quadratically. Larger images might be problematic owing to memory constraints, but given the high rate of improvements made to GPU architectures, we believe that that processing images of doubled or quadrupled sizes is already possible, or soon will be within reach.

6.6 Conclusion and discussion

We have successfully demonstrated the training of a conditional generative adversarial network to restore image brightness levels of simulated radio astronomy images to the point where these get close to the performance of WSclean with a single cleaning iteration. Although the performance is promising, it is not outperforming traditional cleaning methods yet.

We evaluated two modifications to the network, the addition of the PSF as an input channel, and an additional term to the loss function. Both experiments, unfortunately, yielded no significant improvements. The lack of improvement was surprising, since adding domain-specific gradient information would generally improve the performance over a more general loss function.

There could be multiple explanations for why the performance is not better. Firstly, it could be that autoencoders are not quite suitable for reconstructing the delta function point sources in the training set. Most image-to-image applications in the literature process images with complex structure. Secondly, it could be that we are approaching the limits of the representative capabilities of the auto-encoder network. Similar to solving for DDEs during calibration, knowing how to improve the results is sometimes like black magic, which also illustrates an issue with deep learning. It is not trivial to determine why specific methods work or not and why certain architectural modifications do not improve the results.

In this case, adding a better loss function might decrease the training time, but if the network is trained for long enough, the final performance would be equal. This is what is seen in performance comparisons in Figure 6.3.

Thirdly, it is possible that we are just setting the strengths of the individual regularisers incorrectly, since this is a subtle task that usually requires cross-validation.

Lastly, an explanation could be that a GAN in itself is quite powerful and it will learn the loss function by itself. In this case, adding additional information is not beneficial, since the GAN already captures this information by itself.

Although all experiments in this paper have been performed on a resolution of 256 by 256, a small experiment on doubling the auto-encoder input and output size (to 512x512) by adding a convolution layer reveals similar learning performance. The scaling comes at a computational cost; the training takes considerably longer. Given enough GPU memory, we expect a network with higher input and output dimensions to produce similar results. Although we did not manage to improve the cleaning performance of commonly used existing tools, the computational time performance of the proposed approach looks

very promising. We believe that this network is suitable in situations where an accurate image is less important, but a quick cleaned estimate of the sky model is of a high priority. One example of its application is the localisation of bright sources (source finding). These locations can be used in combination with other calibration algorithms.

To facilitate the training of the network, we have created a set of sky models and simulated dirty images named ASTRODECONV2019, which are freely available for download. We hope this dataset will be used for future research in the field of restoring image brightness of point source models observed by radio interferometers.

6.7 Future work

In retrospect, the goals of Vacuum Cleaner have been too ambitious, and a smaller scope should have been the starting point. Optimising the training of a neural network involves a lot of trial and error, and for Vacuum Cleaner, there are still many aspects to be investigated. One example is evaluating and understanding the performance of using a likelihood loss function with a fixed PSF compared to a variable PSF. Another interesting issue to rule out would be to investigate if autoencoders are suitable for generating models containing only point sources rather than complex images. Numerous architectures could be combined and evaluated, for example, a recurrent neural network (Putzky and Welling, 2017).

Chapter 7

Conclusions

We started this work with a historical overview of radio astronomical software. From this, it became clear that radio astronomers are incredibly successful at solving problems and implementing those solutions in individual software packages. At the same time, it is somewhat conspicuous that this has not been matched by a record of putting together successful (user-facing) end-to-end pipelines, despite concerted efforts. Clearly, the Pipeline Problem is more difficult than it seems at first.

There are, clearly, many subtle problems that contribute to this difficulty. In this work, we have identified and attempted to address two of them: software distribution and installation, and software composition and interoperability.

The proposed solution to the distribution and installation problem is the KERN suite (Chapter 3). The KERN suite is available to the worldwide community, and we are carrying on maintenance and releases as an ongoing process. KERN-6 was released shortly before this work was submitted. KERN is actively used at ASTRON and SARAO and other institutes, and there is healthy growth in the usage numbers (see Section 3.9). An increasing number of independent researchers are using KERN to support their work (Creaner and Carozzi, 2019; Sabater et al., 2018; Sabater et al., 2019; van Hateren, 2019). We optimistically believe KERN has now become a crucial building block for software development in radio astronomy.

The emergence of container technologies such as Docker has opened up new possibilities for both software distribution and composition. Much like the different strands of self-cal emerging almost simultaneously at different institutions, the idea of a pipeline as a chain of containers seemed to arise 'because the time was right'. The author's contribution to this, Kliko, is described in Chapter 4. Unfortunately, this turned out to be a duplication of effort. Simultaneously, a larger community with more significant momentum was formed in a similar project named CommonWL. Although Kliko served as a useful testbed for pipeline containerisation ideas, it was decided to cease it's development. Since the astronomy software ecosystem is an organically evolving landscape without central coordination or planning, the end of Kliko did not stop other astronomers from taking inspiration from Kliko. The Stimela framework was created based on concepts and ideas borrowed from Kliko. Time will tell if the radio-astronomy community will adopt Stimela, use CommonWL, or will find a different solution for constructing and deploying portable pipelines.

In Chapter 5, we investigate the use of CommonWL for radio astronomy pipelines, and develop a CommonWL web-based front end and scheduling framework called Buis. As a proof-of-principle example, a Buis pipeline for performing 1GC calibration is shown.

In the final chapter, we show a specific use case for the ideas above, by developing a novel approach for deconvolving interferometry maps. The pipeline used to test the approach (and to generate a range of synthetic datasets for it) is based on CommonWL. We managed to fine-tune the image quality output performance to match the quality of a single-iteration clean step. Although the reconstruction performance does not outperform traditional cleaning methods in image quality output, we believe this approach is an exciting new addition to the radio astronomer's technical repertoire, since it opens up new ways to approach the deconvolution problem and, potentially, speed up the computational performance. This research has by no means been completed, and our experiments are just a preliminary investigation into applying deep learning in the field of image reconstruction. The method is implemented in a freely available open-source project named Vacuum Cleaner. In addition, the ASTRODECONV2019

simulated dataset (generated with the CommonWL based pipeline) is made public. The usage of this dataset is encouraged for experiments and comparisons to our results.

Some of this work has already directly contributed to, or influenced, the development of new radio astronomy pipelines. In particular, the CARACal pipeline (Sect. 5.4) is based on the Stimela framework, which (a) was influenced by Kliko in many ways, and (b) ships a set of base software images entirely based on KERN. If this pipeline is a success, the author will be happy to know that he has contributed some small part to solving the Pipeline Problem.

Meanwhile, software developers and managers working on telescope software are becoming increasingly aware of the usefulness of decomposing pipelines into abstract building blocks using tools such as CommonWL. We hope that this trend will continue, and the usage of CommonWL or similar frameworks for constructing pipelines will keep growing. We believe this technology has a big role to play in the SKA data conundrum. It has been postulated that the data output rates of a telescope such as the SKA (even already at Phase 1) will be on such an immense scale that it would not be practically possible for the end-user astronomer to be involved with the imaging and calibration process. Instead, users would have to settle for receiving the reduced data as-is, coming from a predefined pipeline running at an SKA data centre somewhere, using some predefined parameters and heuristics. This prospect has caused some anguish among radio astronomers, many of whom are used to reducing their own data in their own (naturally, perceived superior) way. Yet even a brief consideration of the SKA1 data rates seems to preclude the possibility of any end-user involvement. Or does it? If the data pipelines are open-source and structured in a recomposable way, the end-user astronomer would, in principle, be able to supply a custom pipeline to the data centre, in the form of something akin to a CommonWL script. Such custom pipelines could be based on the predefined standard pipelines, but fine-tuned and optimised by users (presumably, using smaller datasets in their own computing environments). Packaging, containerisation and workflow tools would simplify this task.

'Take pipelines from the user' instead of 'give data to the user' would be a completely new model for working with data, and much work would have to be done to make it a reality. The skeptical reader may, quite justifiably, consider this impossible. Skeptical readers are hereby encouraged to return to the start of Chapter 1 for a recalibration of their sense of what is possible.

Appendix A

KERN packages

Table A.1: List of packages in KERN

Package name	Description and website	Version in KERN-5
21cmfast	a seminumeric modelling tool to efficiently simulate the cosmological 21-cm signal	
	https://github.com/andreimesinger/21cmFAST	2.0-1
aips	variety of ancillary tasks on Astronomical Data	
	http://www.aips.nrao.edu/	31dec18-1
aoflagger	Find RFI in radio astronomical observations (shared lib)	
	https://sourceforge.net/projects/aoflagger/	2.13.0-1
apsynsim	Aperture Synthesis Simulator for Radio Astronomy	
	https://launchpad.net/apsynsim	1.4v2-1
astro-amber	A many-core transient searching pipeline	
	https://github.com/AA-ALERT/AMBER	2.1-1
astrodata	Set of C++ classes to operate on radio astronomical data.	
	https://github.com/AA-ALERT/AstroData	3.1.1-2
attrdict	Dictionary that allows attribute-style access (Python 3)	
	https://github.com/bcj/AttrDict	2.0.0-1kern6
bifrost	stream processing framework for high-throughput applications	
	https://github.com/ledatelescope/bifrost	0.8.0+git20180512.8e1c0e1-1
bitshuffle	filter for improving typed data compression. (Python 3)	
	https://pypi.python.org/pypi/bitshuffle/	0.3.5-2
bl-dspsr	UCBerkeleySETI specific fork of dspsr	
	https://github.com/UCBerkeleySETI/dspsr	0.0 git20180312.50ea209-1
bl-sigproc	Breakthrough Listen fork of SigProc tools	
	https://github.com/UCBerkeleySETI/bl_sigproc	0.0 git20181130.efd2724-1
blimpy		
	https://github.com/UCBerkeleySETI/blimpy	1.3.5-1kern1
blitz	C++ template class library providing array objects	
	https://github.com/blitzpp/blitz/	1.0.1-1kern2
carta	Cube Analysis and Rendering Tool for Astronomy	
	https://cartavis.github.io/	1.0-1kern1
carta-remote	Remote version of CARTA	
	https://cartavis.github.io/	1.0-1kern1
casacore	CASA table system	
	https://casacore.github.io/casacore	3.0.0-4kern2
casacore-data	Casacore data files	
	http://casacore.github.io/casacore	20190909-000001-1
casalite	Common Astronomy Software Application	
	https://casa.nrao.edu/casa_obtaining.shtml	5.4.1-1kern3
casarest	standalone radio interferometric imager derived from CASA	

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
	https://github.com/casacore/casarest/	1.5.0-1
chgcentre	used to change the phase centre of a measurement set.	
	https://sourceforge.net/p/wsclean/wiki/chgcentre/	1.6-1
coast-guard	A Python/PSRCHIVE-based timing pipeline for reducing Effelsberg data.	
	https://github.com/plazar/coast_guard	0.0+git20151216.201031b-1
ctypesgen	Pure Python wrapper generator for ctypes	
	https://github.com/davidjamesca/ctypesgen	0.0 git20150511.3d2d980-2
cub	flexible library of cooperative threadblock primitives and other	
	http://nvlabs.github.io/cub	1.8.0-1kern1
dadafilterbank	Connect to a PSRdada ringbuffer and write out the data in filterbank format.	
	https://github.com/AA-ALERT/dadafilterbank	0.0 git20180613.fa829f9-1
dadafits	Connect to a PSRDADA ringbuffer and write FITS files	
	https://github.com/AA-ALERT/dadafits	0.0 git20180822.8e0cd24-1
ddfacet	A facet-based radio imaging package.	
	https://github.com/saopicc/DDFacet	0.3.4.1-1kern1
dedisp-ajameson	CUDA Based De-dispersion library	
	https://github.com/ajameson/dedisp	0.0 git20180426.8a3d017-1
dedisp-ewanbarr	CUDA Based De-dispersion library	
	https://github.com/ewanbarr/dedisp	0.0 git20171127.7aa2a81-2
dedisp-flexi	CUDA Based De-dispersion library	
	https://github.com/ewanbarr/dedisp-flexi	0.0 git20171127.19826ae-1
dedispersion	Many-core incoherent dedispersion algorithm in OpenCL	
	https://github.com/AA-ALERT/Dedispersion	4.1-1
difmap	Difference mapping interferometry tool.	
	ftp://ftp.astro.caltech.edu/pub/difmap/difmap.html	2.5b-1
dp3	streaming processing pipeline for radio data (Python 2)	
	https://github.com/lofar-astron/DP3	3.1.7+git20181205.395619b-1kern1
drive-casa	Python package for scripting the NRAO CASA pipeline routines	
	https://github.com/timstaley/drive-casa	0.7.6-1kern1
dspsr	library for DSPing of pulsar astronomical timeseries (headers)	
	http://dspsr.sourceforge.net/	1.0+git20181218.ddbbebb-1
dysco	Compressing storage manager for Casacore meaerement sets	
	https://github.com/aroffringa/dysco	1.2-1kern2
eht-imaging	Python modules for simulating and manipulating VLBI data	

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
	https://github.com/achael/eht-imaging	0.1.2-1kern1
eigency	Python3 interface between the numpy arrays and Eigen C++ library	
	https://github.com/wouterboomsma/eigency	1.77-1
factor	Facet calibration for LOFAR (Python 2)	
	https://github.com/lofar-astron/factor	1.3-1kern1
gbt-seti	GUPPI and Breakthrough Listen instruments tools	
	https://github.com/UCBerkeleySETI/gbt_seti/	0.0 git20180503.f42056b-1
gsm	Native Python3 client API for monetDB	
	https://github.com/bartscheers/gsm	2.2.2-1
guppi-daq	Data aquisition software for GUPPI	
	https://github.com/demorest/guppi_daq	0.0 git20120831.35afdbc-1
heimdall-astro	GPU accelerated transient detection pipeline for radio astronomy	
	https://sourceforge.net/projects/heimdall-astro/	0.0 git20180525.86030c5-1kern1
idg	Image Domain Gridding core library	
	https://gitlab.com/astron-idg/idg	0.4+git20190106.4a78e2f-1
integration	Many-core integration algorithm	
	https://github.com/AA-ALERT/Integration	3.1-1
kapteyn	Python tools for astronomy	
	https://www.astro.rug.nl/software/kapteyn/	2.3-1
karma	toolkit for interprocess communications, authentication, encryption, graphics display, user interface and manipulating the Karma network data structure	
	http://www.atnf.csiro.au/computing/software/karma/	1.7.25-1kern3
katdal	Data access library for the MeerKAT project	
	https://github.com/ska-sa/katdal	0.11-1
katpoint	Karoo Array Telescope pointing coordinate library	
	https://github.com/ska-sa/katpoint	0.7-1kern2
katversion	Provides proper versioning for Python packages	
	https://github.com/ska-sa/katversion	0.9-1kern1
kittens	Collection of Python utility functions for purr, tigger, meqtrees and others	
	https://github.com/ska-sa/kittens	1.4.2-1kern1
libisaopencl	Simple C++ library containing OpenCL utilities.	
	https://github.com/isazi/opencl	1.1-1
libisautils	Simple C++ library containing generic utilities.	
	https://github.com/isazi/utils	0.1.2-1
lofarbeam	Stand-alone version of the LOFAR station response library	
	https://github.com/lofar-astron/LOFARBeam	4.0-1kern2
losoto	LOFAR solutions tool (Python 2)	
	https://github.com/revoltek/losoto	2.0-1kern1
lsmtool	LOFAR Local Sky Model Tool (Python 2)	

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
	https://github.com/darafferty/LSMTool/	1.3.1-1kern1
makems	tool to create empty Measurement Sets	
	https://github.com/ska-sa/makems	1.5.1-1
meqtrees-cattery	Frameworks for simulation and calibration of radio interferometers	
	https://github.com/ska-sa/meqtrees-cattery	1.5.3-1kern2
meqtrees-timba	implementing and solving arbitrary Measurement Equations	
	http://www.astron.nl/meqwiki/MeqTrees	1.6.0-1
miriad	radio interferometry data reduction package	
	http://www.atnf.csiro.au/computing/software/miriad/	20181011-1
msutils	A set of CASA Measurement Set manipulation tools	
	https://github.com/SpheMakh/msutils	0.9.6-1kern1
multinest	a Bayesian inference tool	
	https://github.com/JohannesBuchner/MultiNest	3.10-1kern1
nifty	Numerical Information Field Theory (Python 3)	
	https://gitlab.mpcdf.mpg.de/ift/nifty	5.0.0-1kern1
obit	Obit for ParselTongue	
	http://www.c6.nrao.edu/~bcotton/Obit.html	22JUN10l-1kern2
oskar	Simulator for the Open Square Kilometre Array Radio Telescope	
	http://www.oerc.ox.ac.uk/~ska/oskar2/	2.7.0-1kern2
owlcat	miscellaneous utility scripts for manipulating interferometry data	
	https://github.com/ska-sa/owlcat	1.5.1-1
parseltongue	Python scripting interface for classic AIPS	
	http://www.jive.nl/jivewiki/doku.php?id=parseltongue:parseltongue	2.3-1ubuntu1ppa1 bionic1
peasoup	C++/CUDA GPU pulsar searching library	
	https://github.com/ewanbarr/peasoup	0+git20171127-1kern1
polygon2	Polygon is a Python package that handles polygonal shapes in 2D.	
	http://www.j-raedler.de/projects/polygon	2.0.8-1kern1
ppgplot	Pythonic Interface to PGPLOT	
	https://github.com/npat-efault/ppgplot	1.4-1kern1bionic1
presto	Large suite of pulsar search and analysis software	
	https://github.com/scottransom/presto/	2.1-1kern2
psrcat	ATNF Pulsar Catalogue	
	http://www.atnf.csiro.au/people/pulsar/psrcat/	1.59-1
psrchive	analysis of pulsar astronomical data (Python library)	
	http://psrchive.sourceforge.net/	2012.12+git20190107.455515c-1
psrdada	Distributed Acquisition and Data Analysis	
	http://psrdada.sourceforge.net/	0.0 git20181218.2527e4c-1

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
psrfits-utils	library for working with search- and fold-mode PSRFITS pulsar data	
	https://github.com/demorest/psrfits_utils	0.0+git20170120.1fbf51b-2
purr	Data reduction logging tool, Useful for remembering reductions	
	https://github.com/ska-sa/purr	1.4.3-1
pybdsf	Python Blob Detection and Source Finder	
	http://www.astron.nl/citt/pybdsf/	1.8.15-1
pyfftw	Pythonic wrapper around FFTW - Python 3	
	https://github.com/pyFFTW/pyFFTW	0.11.1.1-1kern1
pygsm	Global Sky Model (GSM) for the radio sky between 10 MHz - 5 THz	
	https://github.com/telegraphic/PyGSM	1.1.1-1
pymonetdb	Native Python 3 client API for monetDB	
	https://github.com/gijzelaerr/pymonetdb	1.1.1-1kern2
pymoresane	Python version of the MORESANE deconvolution algorithm	
	https://github.com/ratt-ru/PyMORESANE	0.3.6.1-1kern1
pyslalib	f2py and numpy based wrappers for SLALIB	
	https://pypi.python.org/pypi/pySLALIB/	1.0.4+git20170925.fcb0650-1
python-casacore	Python 3 bindings to the casacore library	
	https://github.com/casacore/python-casacore	3.0.0-1kern1
python-graphviz	Simple Python 3 interface for Graphviz	
	https://github.com/xflr6/graphviz	0.10.1-1
python-typing	Backport of the standard 3.5 library typing module	
	https://pypi.python.org/pypi/typing/	3.6.6-1
pyxis	meta package for Python Extensions for Interferometry Scripting	
	https://github.com/ska-sa/pyxis	1.6.1-1
rfimasker	Tool to apply RFI masks	
	https://github.com/bennahugo/RFIMasker	1.0.1-1kern2
ringbuffer-sc4	copy data from the network into a ringbuffer and do checking	
	https://github.com/AA-ALERT/ringbuffer-sc4	0.0 git20180724.a7e6fcf-1
rmextract	LOFAR solutions tool (Python 2)	
	https://github.com/lofar-astron/RMextract/	0.2-1kern1
rpfits	data-recording format	
	http://www.atnf.csiro.au/computing/software/rpfits.html	2.25-1
sagecal	GPU accelerated radio interferometric calibration	
	https://sourceforge.net/projects/sagecal/	0.5.0+git20181010.56e7526-1
scatterbrane	adding realistic scattering to astronomical images	
	https://github.com/krosenfeld/scatterbrane	0.0+git20160122-1kern3
sched	VLBI scheduling tool	

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
	http://www.aoc.nrao.edu/~cwalker/sched/sched.html	11.5-1
sharedarray	share numpy arrays with other processes on the same computer	
	https://gitlab.com/tenzing/shared-array	3.1.0-1
shm	System V shared memory and semaphores	
	http://nikitathespider.com/python/shm/	1.2.2-1kern1
sigproc	pulsar search and analysis software	
	https://github.com/SixByNine/sigproc	0.0+git20171113-1kern1
sigpyproc	Python-based pulsar search data manipulation package	
	https://github.com/ewanbarr/sigpyproc	0.0+git20160115-1kern2
simfast21	generates a simulation of the cosmological 21cm signal	
	https://github.com/mariogrs/Simfast21	2.0.1 beta-1kern1
singularity-container	container platform focused on supporting "Mobility of Compute"	
	http://www.sylabs.io	2.6.0-1
snr	Computing signal-to-noise ratio of dedispersed and folded time series.	
	https://github.com/AA-ALERT/SNR	3.1-1
sourcery	Tools for creating high fidelity source catalogues from radio interferometric datasets	
	https://github.com/radio-astro/sourcery	1.2.6-1kern4
spead2	Streaming Protocol for Exchange of Astronomical Data	
	https://github.com/ska-sa/spead2	1.10.0-1
stimela	Dockerised Radio Interferometry Scripting Framework	
	https://github.com/SpheMakh/Stimela	1.0.0-1kern1
tempo	pulsar timing data analysis package	
	https://sourceforge.net/projects/tempo/	0.0 git20181219.9e3092e-1
tempo2	pulsar timing package	
	https://bitbucket.org/psrsoft/tempo2/	2018.09.01-1
tigger	dependency package for FITS and MeqTrees LSM viewer	
	https://github.com/ska-sa/tigger	1.4.0-1kern4
tigger-lsm	dependency package for FITS and MeqTrees LSM viewer	
	https://github.com/ska-sa/tigger-lsm	1.5.0-1
tirific	simulate kinematical and morphological models	
	http://gigjozsa.github.io/tirific/	2.3.9-1kern2
tkp	A transients-discovery pipeline for astronomical image-based surveys	
	https://github.com/transientskp/tkp	4.0.1-1kern2
tmv	Fast, intuitive C++ linear algebra library (header files)	

Continued on next page

Table A.1 – continued from previous page

Package name	Description and website	Version in KERN-5
	https://github.com/rmjarvis/tmv	0.75-1kern1
transitions	lightweight, object-oriented finite state machine implementation	
	https://github.com/pytransitions/transitions	0.6.9-1
turbo-seti	analysis tool for the search of narrow band drifting signals	
	https://github.com/UCBerkeleySETI/turbo_seti	0.7.2 git20181014.e702a35-1
unittest-xml-reporting	PyUnit-based test runner with JUnit like XML reporting	
	https://github.com/danielfm/unittest-xml-reporting	2.2.0-1kern2
wsclean	Fast generic widefield interferometric imager (development files)	
	https://sourceforge.net/projects/wsclean/	2.6-1kern2
ymw16	model for the distribution of free electrons	
	http://www.xao.ac.cn/ymw16/	1.3.2-1

Appendix B

Kliko specification and example

B.1 An example of a kliko.yml file

```yaml
schema_version: 2
name: Kliko test image
description: for testing purposes only
container: kliko/klikotest
author: Gijs Molenaar
email: gijsmolenaar@gmail.com
url: http://github.com/gijzelaerr/kliko
io: split

sections:
 -
   name: section1
   description: The first section
   fields:
     -
       name: choice
       label: choice field
       type: choice
       initial: second
       required: True
       choices:
         first: option 1
         second: option 2
     -
       name: char
       label: char field
       help_text: maximum of 10 chars
       type: char
       max_length: 10
       initial: empty
       required: True
     -
       name: float
       label: float field
       type: float
       initial: 0.0
       required: False
 -
    name: section2
    description: The final section
    fields:
      -
        name: file
        label: file field
        help_text: a helpful text
        type: file
        required: True
      -
        name: int
        label: int field
        type: int
        required: True
```

Listing 10: Example `/kliko.yml`

B.2 The Kliko validation specification

```
schema;fields:
 type: map
 mapping:
   name:
     type: str
     required: True
   type:
     type: str
     required: True
     enum: >
       ['choice', 'char', 'float', 'file', 'bool', 'int']
   initial:
     type: any
     required: False
   max_length:
     type: int
     required: False
   choices:
     type: map
     required: False
     mapping:
       regex;(.*):
         type: str
   label:
     type: str
     required: False
   help_text:
     type: str
     required: False
   required:
     type: bool
     required: False
```

Listing 11: The Kliko definition version 2 - part I

```
type: map
mapping:
 schema_version:
   type: int
 author:
   type: str
 name:
   type: str
 description:
   type: str
 container:
   type: str
   pattern: .+/.+
 email:
   type: str
   pattern: .+@.+
 url:
   type: str
   pattern: >
     https?:\/\/(www\.)?[-a-zA-Z0-9@:
     %._\+~#=]{2,256}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#?&//=]*)
 io:
   type: str
   required: True
   enum: ['split', 'join']
 sections:
   type: seq
   matching: "any"
   sequence:
     - type: map
       mapping:
         name:
           type: str
           required: True
         description:
           type: str
           required: True
         fields:
           type: seq
           required: True
           sequence:
             - include: fields
```

Listing 12: The Kliko definition version 2 - part II

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. et al. (2016). Tensorflow: A system for large-scale machine learning., In *Osdi*.

Adler, J., & Andöktem, O. (2017). Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems*, *33*(12), 124007.

Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Menager, H., Nedeljkovich, M., Scales, M., Soiland-Reyes, S., & Stojanovic, L. (2016). Common workflow language, v1.0. *UC Davis*. https://doi.org/10.6084/m9.figshare.3115156.v2

Aumann, R. J., & Maschler, M. (1972). Some thoughts on the minimax principle. *Management Science*, *18*(5-part-2), 54–63.

Baade, W., & Minkowski, R. (1954). Identification of the radio sources in cassiopeia, cygnus a, and puppis a. In *Classics in radio astronomy* (pp. 251–272). Springer.

Bagchi, S., & Mitra, S. K. (2012). *The nonuniform discrete fourier transform and its applications in signal processing* (Vol. 463). Springer Science; Business Media.

Barrett, D. G., Morcos, A. S., & Macke, J. H. (2019). Analyzing biological and artificial neural networks: Challenges with opportunities for synergy? *Current Opinion in Neurobiology*, *55*, 55–64.

Berliner, H. J. (1978). A chronology of computer chess and its literature. *Artificial Intelligence*, *10*(2), 201–214.

Boettiger, C. (2014). An introduction to docker for reproducible research, with examples from the r environment. *Operating Systems Review*, *abs/1410.0846*.

Booth, R., De Blok, W., Jonas, J. L., & Fanaroff, B. (2009). Meerkat key project science, specifications, and proposals. *arXiv preprint*arXiv 0910.2935.

Born, M., & Wolf, E. (1964). Principles of optics, second (revised) edition, *12*, 124.

Bridger, A., Williams, S., McLay, S., Yatagai, H., Schilling, M., Biggs, A., Tobar, R., & Warmels, R. H. (2012). The alma ot in early science: Supporting multiple customers, In *Software and cyberinfrastructure for astronomy ii*. International Society for Optics and Photonics.

Briggs, D. S., Schwab, F. R., & Sramek, R. A. (1999). Imaging (G. B. Taylor, C. L. Carilli, & R. A. Perley, Eds.). In G. B. Taylor, C. L. Carilli, & R. A. Perley (Eds.), *Synthesis imaging in radio astronomy ii*.

Broderick, J. W., Fender, R., Breton, R., Stewart, A., Adam J. andRowlinson, Swinbank, J. D., Hessels, J., Staley, T. D., van der Horst, A., Bell, M. E. et al. (2016). Low-radio-frequency eclipses of the redback pulsar j2215+ 5135 observed in the image plane with lofar. *Monthly Notices of the Royal Astronomical Society*, *459*(3), 2681–2689.

Carbone, D., Garsden, H., Spreeuw, H., Swinbank, J. D., van der Horst, A., Rowlinson, A., Broderick, J. W., Rol, E., Law, C., Molenaar, G. J. et al. (2018). Pyse: Software for extracting sources from radio images. *Astronomy and Computing*, *23*, 92–102.

Carrillo, R. E., McEwen, J. D., & Wiaux, Y. (2014). Purify: A new approach to radio-interferometric imaging. *Monthly Notices of the Royal Astronomical Society*, *439*(4), 3591–3604. https://doi.org/10.1093/mnras/stu202

Connor, L., & van Leeuwen, J. (2018). Applying deep learning to fast radio burst classification. *The Astronomical Journal*, *156*(6), 256.

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, *19*(90), 297–301.

Cornwell, T. J., Golap, K., & Bhatnagar, S. (2008). The noncoplanar baselines effect in radio interferometry: the w-projection algorithm. *IEEE Journal of Selected Topics in Signal Processing*, *2*, 647–657. https://doi.org/10.1109/JSTSP.2008.2005290

Cornwell, T. J., & Wilkinson, P. N. (1981). A new method for making maps with unstable radio interferometers. *Monthly Notices of the Royal Astronomical Society*, *196*(4), 1067–1086.

Cotton, W. D. (2008). Obit: A development environment for astronomical algorithms. *Publications of the Astronomical Society of the Pacific*, *120*(866), 439.

Creaner, O., & Carozzi, T. D. (2019). Beammodeltester: Software framework for testing radio telescope beams. *Astronomy and Computing*, *28*, 100311.

Croes, G. A. (1993). On aips++, a new astronomical information processing system, In *Astronomical data analysis software and systems ii*.

Dabbech, A., Ferrari, C., Mary, D., Slezak, E., & Smirnov, O. M. (2014). Moresane: Model reconstruction by sythesis-analysis estimators - deconvolution algorithm for radio interferometric imaging. *Astronomy and Astrophysics*.

Dabbech, A., Onose, A., Abdulaziz, A., Perley, R. A., Smirnov, O. M., & Wiaux, Y. (2018). Cygnus A super-resolved via convex optimization from VLA data. *Monthly Notices of the Royal Astronomical Society*, *476*, 2853–2866. https://doi.org/10.1093/mnras/sty372

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in github: Transparency and collaboration in an open software repository, In *Proceedings of the acm 2012 conference on computer supported cooperative work*. ACM.

de Vos, M., Gunst, A. W., & Nijboer, R. (2009). The lofar telescope: System architecture and signal processing. *Proceedings of the IEEE*, *97*(8), 1431–1437.

Dewdney, P., Hall, P., Schillizzi, R., & Lazio, J. (2009). The square kilometre array. *Proceedings of the Institute of Electrical and Electronics Engineers IEEE*, *97*(8), 1482–1496.

Ewen, H. I., & Purcell, E. M. (1951). Observation of a line in the galactic radio spectrum: Radiation from galactic hydrogen at 1,420 mc./sec. *Nature*, *168*(4270), 356.

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers, In *2015 ieee international symposium on performance analysis of systems and software (ispass)*, Philadelphia, PA, USA, IEEE. https://doi.org/10.1109/ispass.2015.7095802

Fu, X., Huang, J., Ding, X., Liao, Y., & Paisley, J. (2017). Clearing the skies: A deep network architecture for single-image rain removal. *IEEE Transactions on Image Processing*, *26*(6), 2944–2956.

Geirhos, R., Janssen, D. H., Schütt, H. H., Rauber, J., Bethge, M., & Wichmann, F. A. (2017). Comparing deep neural networks against humans: Object recognition when the signal gets weaker. *arXiv preprint*arXiv 1706.06969.

Gentleman, W. M., & Sande, G. (1966). Fast fourier transforms: For fun and profit, In *Proceedings of the november 7-10, 1966, fall joint computer conference*.

Glaser, N., Wong, O. I., Schawinski, K., & Zhang, C. (2019). Radiogan–translations between different radio surveys with generative adversarial networks. *Monthly Notices of the Royal Astronomical Society*, *487*(3), 4190–4207.

Golub, G. H. (1968). Least squares, singular values and matrix approximations. *Aplikace Matematiky*, *13*(1), 44–51.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets, In *Advances in neural information processing systems*.

Greisen, E. W. (2003). Aips, the vla, and the vlba. In *Information handling in astronomy-historical vistas* (pp. 109–125). Springer.

Grobler, T. L., Nunhokee, C. D., Smirnov, O. M., van Zyl, A. J., & de Bruyn, A. G. (2014). Calibration artefacts in radio interferometry–i. ghost sources in westerbork synthesis radio telescope data. *Monthly Notices of the Royal Astronomical Society*, *439*(4), 4030–4047.

Hamaker, J. P. (1979). Kneading: The adjustment of instrumental phase and gain parameters to suppress error patterns in a synthesis map, In *International astronomical union colloquium*. Cambridge University Press.

Hamaker, J. P. (2006). Understanding radio polarimetry. *Astronomy and Astrophysics*, *456*(1), 395–404. https://doi.org/10.1051/0004-6361:20065145

Hamaker, J. P., Bregman, J. D., & Sault, R. J. (1996a). Understanding radio polarimetry. i. mathematical foundations. *Astronomy and Astrophysics Supplement Series*, *117*(1), 137–147.

Hamaker, J. P., Bregman, J. D., & Sault, R. J. (1996b). Understanding radio polarimetry. I. Mathematical foundations. *Astronomy and Astrophysics*, *117*, 137–147.

Hamaker, J. P., Harten, R. H., van Diepen, G. N. J., & Kombrink, K. (1985). Dwarf: The dwingeloo-westerbork astronomical reduction facility. In *Data analysis in astronomy* (pp. 449–455). Springer.

Harten, R. H., Grosbol, P., Greisen, E. W., & Wells, D. C. (1988). The fits tables extension. *Astronomy and Astrophysics Supplement Series*, *73*, 365–372.

Heideman, M., Johnson, D., & Burrus, C. (1984). Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, *1*(4), 14–21.

Högbom, J. A. (1974). Aperture synthesis with a non-regular distribution of interferometer baselines. *Astronomy and Astrophysics Supplement Series*, *15*, 417.

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2016). Image-to-image translation with conditional adversarial networks. *arxiv preprint*arXiv 1703.04887.

Jansky, K. G. (1933). Electrical disturbances apparently of extraterrestrial origin. *Proceedings of the Institute of Radio Engineers*, *21*(10), 1387–1398.

Jin, K. H., McCann, M. T., Froustey, E., & Unser, M. (2017). Deep convolutional neural network for inverse problems in imaging. *IEEE Transactions on Image Processing*, *26*(9), 4509–4522.

Johnston, S., Taylor, R., Bailes, M., Bartel, N., Baugh, C., Bietenholz, M., Blake, C., Braun, J., R .and Brown, Chatterjee, S. et al. (2008). Science with askap. *Experimental Astronomy*, *22*(3), 151–273.

Jonas, J. L. (2009). Meerkat—the south african array with composite dishes and wideband single pixel feeds. *Proceedings of the IEEE*, *97*(8), 1522–1530.

Jonas, J. L. (2015). The meerkat telescope.

Jonas, J. L., de Jager, G., & Baart, E. E. (1985). A 2.3 ghz radio continuum map of the southern sky. *Astronomy and Astrophysics Supplement Series*, *62*, 105–128.

Jones, D. L., Wagstaff, K., Thompson, D. R., D'Addario, L., Navarro, R., Mattmann, C., Majid, W., Lazio, J., Preston, R., & Rebbapragada, U. (2012). Big data challenges for large radio arrays, In *2012 ieee aerospace conference*. IEEE.

Jones, R. C. (1941). A new calculus for the treatment of optical systems i. description and discussion of the calculus. *Journal of the Optical Society of America*, *31*(7), 488–493.

Jones, R. C. (1947). A new calculus for the treatment of optical systems vi. experimental determination of the matrix. *Journal of the Optical Society of America*, *37*(2), 110–112.

Józsa, G. I. G., White, S. V., Thorat, K., Smirnov, O. M., Serra, P., Ramatsoku, M., Ramaila, A. J. T., Perkins, S. J., Moln á r, D. á. n. C., Makhathini, S. et al. (2020a). Caracal: Containerized automated radio astronomy calibration pipeline. *Astrophysics Source Code Library*.

Józsa, G. I. G., White, S. V., Thorat, K., Smirnov, O. M., Serra, P., Ramatsoku, M., Ramaila, A. J. T., Perkins, S. J., Moln á r, D. á. n. C., Makhathini, S. et al. (2020b). Meerkathi–an end-to-end data reduction pipeline for meerkat and other radio telescopes. *arXiv preprint*arXiv 2006.02955.

Junklewitz, H., Bell, M. R., Selig, M., & Enßlin, T. A. (2016). RESOLVE: A new algorithm for aperture synthesis imaging of extended emission in radio astronomy. *Astronomy and Astrophysics*, *586*, A76. https://doi.org/10.1051/0004-6361/201323094

Kemball, A., & Wieringa, M. (2000). Measurementset definition version 2.0. http://casa.nrao.edu/Memos/229.html

Kettenis, M., van Langevelde, H. J., Reynolds, C., & Cotton, W. D. (2006). Parseltongue: Aips talking python, In *Astronomical data analysis software and systems xv*.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint*arXiv 1412.6980.

Kraus, J. D. (1986). *Radio astronomy*. Cygnus-Quasar Books.

Kuiack, M., Huizinga, F., Molenaar, G. J., Prasad, P., Rowlinson, A., & Wijers, R. A. (2018). Aartfaac flux density calibration and northern hemisphere catalogue at 60 mhz. *Monthly Notices of the Royal Astronomical Society*, *482*(2), 2502–2514.

Kurtzer, G. M. (2016). Singularity 2.1.2 - Linux application and environment containers for science. https://doi.org/10.5281/zenodo.60736

Leipzig, J. (2017). A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, *18*(3), 530–536.

Mao, X., Shen, C., & Yang, Y.-B. (2016). Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections, In *Advances in neural information processing systems*.

Mardani, M., Gong, E., Cheng, J. Y., Vasanawala, S. S., Zaharchuk, G., Xing, L., & Pauly, J. M. (2018). Deep generative adversarial neural networks for compressive sensing mri. *IEEE Transactions on Medical Imaging*, *38*(1), 167–179.

McCann, M. T., Jin, K. H., & Unser, M. (2017). Convolutional neural networks for inverse problems in imaging: A review. *IEEE Signal Processing Magazine*, *34*(6), 85–95.

McMullin, J. P., Waters, B., Schiebel, D., Young, W., & Golap, K. (2007a). Casa architecture and applications, In *Astronomical data analysis software and systems xvi*.

McMullin, J. P., Waters, B., Schiebel, D., Young, W., & Golap, K. (2007b). CASA architecture and applications (R. A. Shaw, F. Hill, & D. J. Bell, Eds.). In R. A. Shaw, F. Hill, & D. J. Bell (Eds.), *Astronomical data analysis software and systems xvi*.

Mechev, A., Oonk, R., Shimwell, T., Plaat, A., Intema, H., & Rottgerin, H. (2018). Fast and reproducible lofar workflows with aglow, In *2018 ieee 14th international conference on e-science (e-science)*. IEEE.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239).

Molenaar, G. J., Makhathini, S., Girard, J. N., & Smirnov, O. M. (2018). Kliko — the scientific compute container format. *Astronomy and Computing*, *25*, 1–9.

Molenaar, G. J., & Smirnov, O. M. (2018). Kern. *Astronomy and Computing*, *24*, 45–51.

Momcheva, I., & Tollerud, E. (2015). Software use in astronomy: An informal survey.

Murnaghan, F. D. (1938). The analysis of the kronecker product of irreducible representations of the symmetric group. *American Journal of Mathematics*, *60*(3), 761–784.

Noordam, J. E. (1997). A generic solver for aips++ using mns trees.

Noordam, J. E., & De Bruyn, A. G. (1982). High dynamic range mapping of strong radio sources, with application to 3c84. *Nature*, *299*(5884), 597.

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

Noutsos, A., Sobey, C., Kondratiev, V., Weltevrede, P., Verbiest, J., Karastergiou, A., Kramer, M., Kuniyoshi, M., Alexov, A., Breton, R. et al. (2015). Pulsar polarisation below 200 mhz: Average profiles and propagation effects. *Astronomy and Astrophysics*, *576*, A62.

Offringa, A. R. (2010). Aoflagger: Rfi software. *Astrophysics Source Code Library*.

Offringa, A. R., McKinley, B., Hurley-Walker, N., Briggs, F. H., Wayth, R., Kaplan, D., Bell, M. E., Feng, L., Neben, A., Hughes, J. et al. (2014). Wsclean: An implementation of a fast, generic wide-field imager for radio astronomy. *Monthly Notices of the Royal Astronomical Society*, *444*(1), 606–619.

Offringa, A. R., & Smirnov, O. M. (2017). An optimized algorithm for multiscale wideband deconvolution of radio astronomical images. *Monthly Notices of the Royal Astronomical Society*, *471*, 301–316. https://doi.org/10.1093/mnras/stx1547

Offringa, A. R., van de Gronde, J. J., & Roerdink, J. B. T. M. (2012). A morphological algorithm for improved radio-frequency interference detection. *Astronomy and Astrophysics*, *539*.

Onose, A., Carrillo, R. E., Repetti, A., McEwen, J. D., Thiran, J.-P., Pesquet, J.-C., & Wiaux, Y. (2016). Scalable splitting algorithms for big-data interferometric

imaging in the SKA era. *Monthly Notices of the Royal Astronomical Society*, *462*, 4314–4335. https://doi.org/10.1093/mnras/stw1859

Pandey, V. N., van Zwieten, J. E., de Bruyn, A. G., & Nijboer, R. (2009). Calibrating lofar using the black board selfcal system, In *The low-frequency radio universe*.

Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., & Efros, A. A. (2016). Context encoders: Feature learning by inpainting, In *Proceedings of the ieee conference on computer vision and pattern recognition*.

Pearson, T. J., & Readhead, A. C. S. (1984). Image formation by self-calibration in radio astronomy, *22*(1), 97–130.

Peng, B., Nan, R., Su, Y., Qiu, Y., Zhu, L., & Zhu, W. (2001). Five-hundred-meter aperture spherical telescope project, In *International astronomical union colloquium*. Cambridge University Press.

Prasad, J., & Chengalur, J. (2012). Flagcal: A flagging and calibration package for radio interferometric data. *Experimental Astronomy*, *33*(1), 157–171. https://doi.org/10.1007/s10686-011-9279-5

Price, D., & Tucker, A. (2004). Solaris zones: Operating system support for consolidating commercial workloads, In *Lisa '04: Eighteenth systems administration conference*.

Putzky, P., & Welling, M. (2017). Recurrent inference machines for solving inverse problems. *arXiv preprint*arXiv 1706.04008.

Rohlfs, K., & Wilson, T. L. (2013). *Tools of radio astronomy*. Springer Science; Business Media.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation, In *International conference on medical image computing and computer-assisted intervention*. Springer.

Rosen, R. (2013). Resource management: Linux kernel namespaces and cgroups. *Haifux*, *186*.

Ryle, M., & Neville, A. C. (1962). A radio survey of the north polar region with a 4.5 minute of arc pencil-beam system. *Monthly Notices of the Royal Astronomical Society*, *125*(1), 39–56.

Ryle, M., Smith, F. G., & Elsmore, B. (1950). A preliminary survey of the radio stars in the northern hemisphere. *Monthly Notices of the Royal Astronomical Society*, *110*(6), 508–523.

Ryle, M., & Vonberg, D. D. (1946). Solar radiation on 175 mc./s. In *Classics in radio astronomy* (pp. 184–187). Springer.

Sabater, J., Best, P. N., Hardcastle, M., Shimwell, T., Tasse, C., Williams, W., Brüggen, M., Cochrane, R., Croston, J. H., de Gasperin, F. et al. (2018). The lotss view of radio agn in the local universe.

Sabater, J., Best, P. N., Hardcastle, M., Shimwell, T., Tasse, C., Williams, W., Brüggen, M., Cochrane, R., Croston, J. H., de Gasperin, F. et al. (2019). The lotss view of radio agn in the local universe - the most massive galaxies are always switched on. *Astronomy and Astrophysics*, *622*, A17.

Sault, R. J., Teuben, P. J., & Wright, M. C. H. (1995). A retrospective view of miriad, In *Astronomical data analysis software and systems iv*.

Schuler, C. J., Christopher Burger, H., Harmeling, S., & Scholkopf, B. (2013). A machine learning approach for non-blind image deconvolution, In *Proceedings of the ieee conference on computer vision and pattern recognition*.

Schwab, F. R. (1984). Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry. *The Astronomical Journal*, *89*, 1076–1081. https://doi.org/10.1086/113605

Shepherd, M. C., Pearson, T. J., & Taylor, G. B. (1994). Difmap: An interactive program for synthesis imaging., In *Bulletin of the american astronomical society*.

Shimwell, T., Röttgering, H., Best, P. N., Williams, W., Dijkema, T. J., De Gasperin, F., Hardcastle, M., Heald, G. H., Hoang, D., Horneffer, A. et al. (2017). The lofar two-metre sky survey - i. survey description and preliminary data release. *Astronomy and Astrophysics*, *598*, A104.

Siegman, A. E. (1966). The antenna properties of optical heterodyne receivers. *Applied Optics*, *5*(10), 1588–1594. https://doi.org/10.1364/AO.5.001588

Sinz, F. H., Pitkow, X., Reimer, J., Bethge, M., & Tolias, A. S. (2019). Engineering a less artificial intelligence. *Neuron*, *103*(6), 967–979.

Smirnov, O. M. (2011a). Revisiting the radio interferometer measurement equation - i. a full-sky jones formalism. *Astronomy and Astrophysics*, *527*, A106. https://doi.org/10.1051/0004-6361/201016082

Smirnov, O. M. (2011b). Revisiting the radio interferometer measurement equation. ii. calibration and direction-dependent effects. *Astronomy and Astrophysics*, *527*, A107. https://doi.org/10.1051/0004-6361/201116434

Smirnov, O. M., & Tasse, C. (2015). Radio interferometric gain calibration as a complex optimization problem. *Monthly Notices of the Royal Astronomical Society*, *449*(3), 2668–2684.

Smith, W. D. (1970). Maxi computers face mini conflict: Mini trend reaching computers. *New York Times*. https://www.nytimes.com/1970/04/05/archives/maxi-computers-face-mini-conflict-mini-trend-reaching-computers.html

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

Staley, T. D., & Anderson, G. E. (2015). Chimenea and other tools: Automated imaging of multi-epoch radio-synthesis data with CASA. *Astronomy and Computing*, *13*, 38–49. https://doi.org/10.1016/j.ascom.2015.08.003

Stewart, A. J., Fender, R., Broderick, J. W., Hassall, T. E., Muñoz-Darias, T., Rowlinson, A., Swinbank, J. D., Staley, T. D., Molenaar, G. J., Scheers, B. et al. (2015). Lofar msss: Detection of a low-frequency radio transient in 400 h of monitoring of the north celestial pole. *Monthly Notices of the Royal Astronomical Society*, *456*(3), 2321–2342.

Sullivan, W. T. (2009). *Cosmic noise: A history of early radio astronomy*. Cambridge University Press, Cambridge, UK.

Swinbank, J. D., Staley, T. D., Molenaar, G. J., Rol, E., Rowlinson, A., Scheers, B., Spreeuw, H., Bell, M. E., Broderick, J. W., Carbone, D. et al. (2015). The lofar transients pipeline. *Astronomy and Computing*, *11*, 25–48.

Tasse, C. (2014a). Applying wirtinger derivatives to the radio interferometry calibration problem.

Tasse, C. (2014b). Nonlinear kalman filters for calibration in radio interferometry. *Astronomy and Astrophysics*, *566*, A127. https://doi.org/10.1051/0004-6361/201423503

Tasse, C., Hugo, B., Mirmont, M., Smirnov, O. M., Atemkeng, M., Bester, L., Hardcastle, M. J., Lakhoo, R., Perkins, S., & Shimwell, T. (2018). Faceting for direction-

dependent spectral deconvolution. *Astronomy and Astrophysics*, *611*, A87. https://doi.org/10.1051/0004-6361/201731474

Taylor, G. B., Carilli, C. L., & Perley, R. A. (1999). Synthesis imaging in radio astronomy ii, In *Synthesis imaging in radio astronomy ii*.

Thompson, A. R., Moran, J. M., & Swenson Jr, G. W. (2017). *Interferometry and synthesis in radio astronomy*. Springer Nature.

Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, *4*(1), 25–30.

Bui, T. (2015). Analysis of docker security.

Van der Hulst, J. M., Terlouw, J. P., Begeman, K. G., Zwitser, W., & Roelfsema, P. R. (1992). The groningen image processing system, gipsy, In *Astronomical data analysis software and systems i*.

Van Weeren, R., Williams, W., Hardcastle, M., Shimwell, T., Rafferty, D., Sabater, J., Heald, G. H., Sridhar, S., Dijkema, T. J., Brunetti, G. et al. (2016). Lofar facet calibration. *The Astrophysical Journal Supplement Series*, *223*(1), 2.

van Diepen, G. N. J. (2015). Casacore table data system and its use in the measurementset. *Astronomy and Computing*, *12*, 174–180. https://doi.org//10.1016/j.ascom.2015.06.002

van Diepen, G. N. J., & Dijkema, T. J. (2018). Dppp: Default pre-processing pipeline. *Astrophysics Source Code Library*.

van Hateren, T. (2019). *Paralellization of the ddf-pipeline and introduction to radio astronomy* (Doctoral dissertation). Universiteit van Amsterdam.

Veiga, V. S., Simon, M., Azab, A., Fernandez, C., Muscianisi, G., Fiameni, G., & Marocchi, S. (2019). Evaluation and benchmarking of singularity mpi containers on eu research e-infrastructure, In *2019 ieee/acm international workshop on containers and new orchestration paradigms for isolated environments in hpc (canopie-hpc)*. IEEE.

Wells, D. C. (1985). Nrao's astronomical image processing system (aips). In *Data analysis in astronomy* (pp. 195–209). Springer.

Wells, D. C., & Greisen, E. W. (1979). Fits - a flexible image transport system, In *Image processing in astronomy*.

Wijnholds, S., Grobler, T. L., & Smirnov, O. M. (2016). Calibration artefacts in radio interferometry–ii. ghost patterns for irregular arrays. *Monthly Notices of the Royal Astronomical Society*, *457*(3), 2331–2354.

Xu, L., Ren, J. S., Liu, C., & Jia, J. (2014). Deep convolutional neural network for image deconvolution, In *Advances in neural information processing systems*.

Zernike, F. (1938). The concept of degree of coherence and its application to optical problems. *Physica*, *5*(8), 785–795.

Zhang, K., Zuo, W., Gu, S., & Zhang, L. (2017). Learning deep cnn denoiser prior for image restoration, In *Proceedings of the ieee conference on computer vision and pattern recognition*.