The MeerKAT Science Data Processing Pipeline

Ludwig C. Schwardt,[1] Thomas G. H. Bennett,[1,2] Mattieu S. de Villiers,[1] Kim McAlpine,[1] Thomas Mauch,[1,3] Bruce Merry,[1] Simon J. Perkins,[1] Simon M. Ratcliffe,[1,2] Laura Richter,[1,4] and Martin Slabber[1,2]

[1]*South African Radio Astronomy Observatory, Cape Town, Western Cape, South Africa;* schwardt@sarao.ac.za

[2]*Tsolo.io, Cape Town, Western Cape, South Africa*

[3]*University of Sydney, Sydney, New South Wales, Australia*

[4]*HPE Aruba Networking, Cape Town, Western Cape, South Africa*

Abstract. The MeerKAT radio telescope has been producing high-quality scientific data for more than six years. Its science data processing (SDP) subsystem produces signal displays, calibration products, continuum images, and spectral images in a fully automated and pipelined fashion from the outputs of the correlator and tied-array beamformer. We review the major parts of the MeerKAT SDP pipeline with a focus on the software architecture, design choices, and lessons learnt.

1. Introduction

The MeerKAT radio telescope is an interferometric array of 64 dishes located in the semi-desert Karoo region of South Africa that observes in the ultra-high frequency (UHF), L, and S bands. Jonas & the MeerKAT Team (2016) provided a high-level description of the telescope, including a brief summary of its science data processing (SDP) subsystem, but this summary was already out of date by the time of the telescope's inauguration in July 2018. We aim to update and expand the SDP description.[1]

MeerKAT digitizes the radio signals directly at each receiver and transmits them over a 40 Gbps fibre network to the Karoo Array Processor Building (KAPB) at the telescope site. There it enters the correlator/beamformer (CBF), a combined FX correlator and tied-array beamformer implemented on multiple FPGA-based processing nodes and network switches (van der Byl et al. 2022). The high-speed data is transported into, inside, and out of the CBF as Streaming Protocol for Exchanging Astronomical Data (SPEAD)[2] streams, which allows the efficient chunked transfer of multidimensional ar-

[1]A detailed write-up is also available in the *Science Data Processor* section of the *MeerKAT Knowledge Base* at https://skaafrica.atlassian.net/wiki/spaces/ESDKB/overview.

[2]The SPEAD protocol is described in https://casper.berkeley.edu/wiki/SPEAD and is implemented by the spead2 library (https://spead2.readthedocs.io/) which relies on C++ for speed.

rays between nodes using multicast UDP. The SDP subsystem is co-located with the CBF in the KAPB, with a 25 Gbps internal network.

The SDP subsystem sits between the CBF output and the end user. Its main requirements are to record visibilities, to calibrate the telescope by e.g. phasing up the beamformer, and to provide ways to check the signal quality, such as signal displays and calibration reports. It also produces continuum and spectral images, but these are fully automated best-effort 'quick-look' images for quality assurance purposes. Although these images may satisfy the requirements of some science cases, such as Christy et al. (2024), the majority of science users download the visibility data and do their own calibration and imaging.

2. Overall design

The SDP subsystem is a containerized pipeline running on a cluster of primarily commodity servers. It uses Docker (`https://www.docker.com/`) containers and manages computing resources in the cluster with Apache Mesos (`https://mesos.apache.org/`). It stores metadata in a telescope state (or *telstate*) database based on Redis (`https://redis.io/`), an in-memory key-value store that is fast and atomic, and bulk visibility data in Ceph (`https://ceph.io/`), a scalable storage service configured as an S3 object store. The SDP codebase is written in Python as far as possible, with only a handful of tasks that require C++ or GPU acceleration.

Before an observation, the telescope control and monitoring (CAM) subsystem (Marais 2015) builds a subarray consisting of a set of dishes and one or more CBF instances (or *instruments*) producing different channelizations of the frequency band (with 1024, 4096, or 32 768 channels). It configures the SDP subsystem to produce a *subarray product* per CBF instrument. Each of these subarray products is guaranteed to have a fixed accumulation interval (or *dump period*) and a fixed set of dishes and channel frequencies for the duration of the observation, which simplifies its processing. If any of these parameters need to change for the next observation, the SDP containers are stopped, reconfigured, and started from scratch—a process that takes about a minute.

The SDP subsystem distinguishes between three levels of visibility data products:

- 'raw' uncalibrated visibilities (L0),
- visibilities calibrated by transferring solutions from a nearby calibrator (L1), and
- self-calibrated visibilities associated with an imaging target (L2).

The archive only stores L0 visibilities, while the telstate metadata store contains the relevant calibration solutions that can be used to calculate L1 and L2 visibilities as needed. The Redis database is dumped to an RDB file (a Redis-specific backup format) which is stored in the archive alongside the visibilities as the main source of metadata.

3. Components

Figure 1 gives an overview of the various components making up the SDP subsystem during a typical observation. We describe each component in more detail and point out the relevant SARAO code repository.

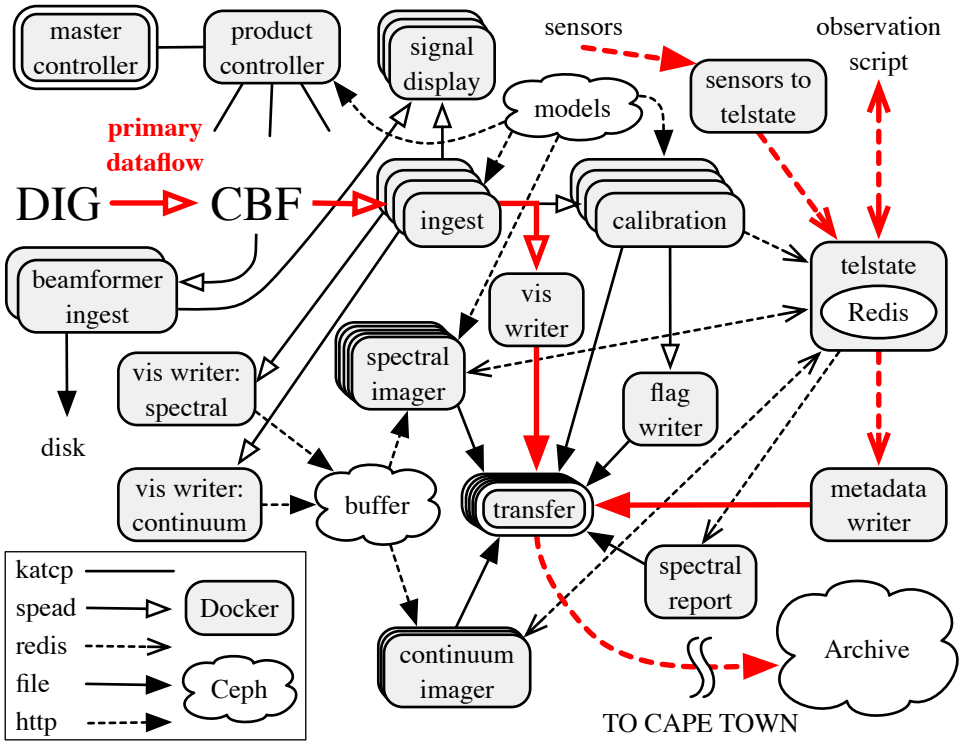

Figure 1. An overview of the MeerKAT science data processing (SDP) pipeline for a single subarray product. This logical graph highlights the various Docker containers and dataflows between them. Digital signals start at the digitizers (DIG) and proceed to the correlator/beamformer (CBF) before entering the SDP subsystem. The primary dataflow is the recording of the raw (L0) visibilities and basic metadata, indicated by thick coloured lines. The containers with double borders (*master controller* and *transfer*) have long-running processes that do not form part of the subarray product. Most containers read metadata from telstate but only the most prominent telstate interactions are shown to avoid cluttering up the graph.

3.1. Control and scheduling

SDP is controlled through the long-running master controller service. It supports the Karoo Array Telescope Communication Protocol (KATCP),[3] a simple TCP/IP-based protocol for sending requests and receiving sensor updates.

The CAM subsystem configures the SDP subsystem by sending it a KATCP request with a JSON payload describing the various input SPEAD streams (including IP addresses) and desired output streams. The master controller then starts up a product controller for each CBF instrument, using the high-availability Singularity[4] scheduler.

The product controller is itself a scheduler or custom Mesos framework that adds the ability to share GPU and network resources, pin containers to CPU cores, and bypass the kernel in containers running on high-performance NICs. It first constructs a

[3] `https://github.com/ska-sa/katcp-guidelines`

[4] `https://getsingularity.com/`

logical graph of the relevant tasks and their dependencies as illustrated in Figure 1. It then resolves this to a physical graph by assigning task containers to hosts based on the information provided by Mesos. These containers are also controlled by KATCP by relaying requests via the master and product controllers.

The controllers are implemented in the `katsdpcontroller`[5] package.

3.2. Ingest

The ingest task receives the correlator output and performs basic data conditioning, averaging, and flagging on it to produce L0 visibilities. It converts 32-bit integers to appropriately scaled 32-bit floating-point numbers, relieving the FPGAs of this task. It averages the fixed 0.5-second dumps to a more appropriate duration, nominally 8 seconds. It also produces auxiliary SPEAD streams for the imagers and signal displays.

The high input dump rate allows radio frequency interference (RFI) flagging on short timescales. The `ingest_rfi` flagger is conservatively tuned to identify only the most egregious spikes in the data. It operates independently on each dump and correlation product. The flagger finds a smooth spectral background by taking the visibility amplitudes and applying a median filter with a kernel width of 13 channels along the frequency axis, estimates the residual noise level using the median absolute deviation (MAD) estimator, and flags visibilities that are more than 11 standard deviations above the background.

Flagged CBF dumps can be skipped during time averaging to obtain unflagged L0 output dumps. This excision step cleans the data but distorts the signal statistics and can potentially remove scientifically useful data, so it is disabled by default.

In order to keep up with a CBF output data rate of up to 4.4 GB/s, the ingest task is split over four containers, each handling a contiguous part of the frequency band. Each container runs on a dedicated host that can access the 40 Gbps CBF network. The main ingest processing happens on a GPU (NVIDIA GeForce GTX TITAN X, recently upgraded to NVIDIA A30).

There is also a beamformer ingest task that receives the beamformer SPEAD streams for two polarizations to produce signal displays for it. An engineering test mode allows the beamformer data to be captured to disk as two HDF5 files.

The ingest task is implemented in the `katsdpingest`[6] package, using GPU kernels in the `katsdpsigproc`[7] package, while the beamformer ingest task is implemented in the `katsdpbfingest`[8] package.

3.3. Signal displays

The signal display task serves dynamic interactive plots to web browsers using HTML5 Canvas elements. It receives a SPEAD stream of lower resolution data (averaged to 256 channels) from the ingest task. It can also request up to 256 full-resolution signals to display outliers and user selections. The data is stored in a ring buffer that ages out old data. Only the data needed to update each plot is requested by the browser

---

[5] `https://github.com/ska-sa/katsdpcontroller`

[6] `https://github.com/ska-sa/katsdpingest`

[7] `https://github.com/ska-sa/katsdpsigproc`

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

via WebSockets to minimize data transfers and increase responsiveness. The available plots include time-series plots, spectrum plots, and waterfall plots. These are annotated with metadata drawn from telstate and can be customized via a command-line interface embedded in the page. Advanced displays used during early stages of testing and commissioning, such as lag plots, are also supported.

The signal display task is implemented in the `katsdpdisp`[9] package.

3.4. Calibration

The calibration task obtains bandpass, gain, delay, and reference pointing solutions while observing targets indicated as calibrators, performs more advanced RFI flagging, and produces calibration reports for quality assurance. It receives L0 visibilities from the ingest task via SPEAD and writes the calibration solutions[10] to telstate. It is semi-realtime as it first buffers the data in memory for up to 12 minutes (or until the end of a scan) before running the solvers on it.

In the case of 32 768 channels, the calibration task is split over four containers like ingest, each handling a contiguous part of the frequency band and synchronizing via telstate thanks to the atomicity of Redis. In the case of 1024 or 4096 channels, only a single container is needed. Each container runs several processes communicating via queues and shared memory.

The main solver is a Python implementation of the StEFCal algorithm (Salvini & Wijnholds 2014) that is accelerated by Numba.[11] The `cal_rfi` flagger is likewise a Numba-accelerated Python implementation of the SumThreshold algorithm of AOFlagger (Offringa et al. 2010) with parameters tuned for MeerKAT's L-band data. It fits a smooth two-dimensional background across both time and frequency on previously unflagged cross-hand visibilities, flags data points using the SumThreshold approach, and extends any flags to all polarization products at the same dump, channel, and baseline. The calibration task also introduces static flags that block out regions known to contain high levels of RFI, but only on baselines shorter than 1 km (Mauch et al. 2020).

The calibration containers run on hosts without a GPU but with sufficient memory to buffer the data (512 GiB, recently upgraded to 1536 GiB). The calibration task is implemented in the `katsdpcal`[12] package, using solvers in the `katsdpcalproc`[13] package.

3.5. Continuum imaging

The continuum imager produces self-calibration solutions, a local sky model and a 'quick-look' continuum image of each target that is tagged to be imaged in an observation and observed for at least 15 minutes. It waits until observation and calibration are done before it starts as a post-processing step. The imager is designed so that the

---

[9] `https://github.com/ska-sa/katsdpdisp`

[10] `https://katdal.readthedocs.io/en/latest/mvf_v4.html#calibration-solutions`

[11] `https://numba.pydata.org/`

[12] `https://github.com/ska-sa/katsdpcal`

[13] `https://github.com/ska-sa/katsdpcalproc`

imaging products are of sufficient quality to do continuum subtraction for subsequent spectral imaging. The products are also used for quality assurance.

The visibility data to be imaged arrives from ingest as a SPEAD stream averaged down to 1024 channels. It is temporarily stored in a Ceph-based 'hot' buffer until the imager can start. This buffer is built on top of solid-state drives (SSDs) located on the imaging hosts, providing an effective usable capacity of 0.2 PiB after 2:1 replication.

The continuum imager task starts a separate container for each target to image (although observations tend to have a single imaging target). When the container is ready, the L0 visibilities are loaded from the hot buffer, calibrated on the fly to L1 (see Section 4.2 for further details), and converted to AIPS UV format[14] on the imaging host. The L1 visibilities also undergo baseline-dependent time averaging with time bins ranging up to 60 seconds in length.

The imager uses the Obit package (Cotton 2008), specifically its state-of-the-art wide-band wide-field MFImage[15] task. This uses automated faceting to deal with wide fields, placing circular facets (around 140 in L band) in the main lobe of the primary beam and additional smaller facets on known bright sources in the first and second sidelobes. The wide frequency band is divided into coarse subbands to handle frequency structure inherent to radio sources and the antenna primary beam.

In the first round the imager does a shallow deconvolution, resulting in about a thousand CLEAN components, followed by phase-only self-calibration with a solution interval of 60 seconds. The second round involves a deeper CLEAN down to the desired imaging depth, which is constrained by the sensitivity of a single frequency channel in the spectral imager, to ensure thorough continuum subtraction. This typically results in a few tens of thousands of CLEAN components. After a second phase-only self-calibration the deep CLEAN is repeated to produce the final image. The calibration solutions and sky model are stored in telstate where they can be used by the spectral imager to produce L2 visibilities. The continuum image is transferred to the archive as both a FITS file for scientific use and a PNG file as a quick-look image on the archive page.

A typical 8-hour L-band observation takes about 10 hours to image. If the field to be imaged is complicated it can take much longer, in which case imaging is terminated after twice the observation time to keep up with the average data flow. The MeerKAT-to-Obit interface is implemented in the `katsdpcontim`[16] package.

3.6. Spectral imaging

The spectral imager produces an image per spectral channel on wideband data for quality assurance purposes. It has been built from scratch, with a strong focus on autonomy, memory usage, GPU support, and overall speed, rather than outright imaging precision. It loads L2 visibilities from the hot buffer, subtracts the local sky model produced by the continuum imager from the visibilities by direct evaluation of the measurement equation, and produces a residual image per channel.

---

[14] See AIPS Memo 117: `https://library.nrao.edu/public/memos/aips/memos/AIPSM_117.pdf`

[15] MFImage is described in the *Obit Development Memo Series no. 63* which can be downloaded from `https://www.cv.nrao.edu/~bcotton/ObitDoc/MFImage.pdf`.

[16] `https://github.com/ska-sa/katsdpcontim`

The spectral imaging problem is embarrassingly parallel over channels but is hampered by the default visibility chunking scheme which stores data one dump at a time instead of the ideal of one channel at a time. A dedicated spectral visibility writer partially transposes the data by rechunking it to have fewer channels (128 by default) and more dumps per chunk. The spectral imager task then starts a container for each batch of 128 channels. The container process completes this transposition as a preprocessing step on the CPU (implemented in C++) and writes out the transposed data with some conservative baseline-dependent averaging to a temporary HDF5 file or memory before turning to the GPU for the main imaging steps.

The imaging gridder and degridder use a hybrid of W-stacking (Offringa et al. 2014) and W-projection (Cornwell et al. 2008) which is more efficient than either separately (Merry 2016a). Deconvolution is done by a standard CLEAN algorithm and primary beam correction uses a simple spherical model. There is no support for faceting, self-calibration, or Doppler correction. Imaging is inefficient on short scans so targets are required to be observed for at least 45 minutes. Some effort went into making the convolutional gridding step efficient on a GPU (Merry 2016b).

Images are nominally 4608 pixels per side[17] and are written to the archive as an individual FITS file per channel. The SDP cluster has 24 imaging hosts with four NVIDIA GeForce GTX 1080 Ti GPUs each, all together providing about 1 PFLOPS of theoretical single-precision performance. A typical 8-hour L-band observation with 32 768 channels takes 30 minutes to 5 hours to image on these hosts (depending on image complexity), excluding continuum imaging and assuming half the channels are flagged or masked. A separate spectral report task shows the comparison of imaged to theoretical noise across the band, and highlights channels with strong sources in them.

The spectral imager is implemented in the `katsdpimager`[18] package.

3.7. Model data and archive

The SDP subsystem has a central repository for storing model data shared between various tasks, such as RFI masks, band masks and primary beam models. The models are version-controlled and can be shared among the relevant targets. For example, a primary beam model can be shared among various antennas. The actual models are stored as HDF5 files in a Ceph object store, together with various aliases. The code to retrieve and sample the models is provided by the `katsdpmodels`[19] package.

The main SDP archive[20] is located in Cape Town and consists of a Ceph object store with 6.45 PiB of usable capacity, implemented by a 6+2 erasure-coded pool backed by hard drives. Data is then aged out to a 20 PiB tape archive for long-term storage. The visibility, flag, and metadata writer tasks in the SDP pipeline write their output to a 0.5 PiB buffer on site, which protects the data against outages on the 10-Gbps fibre link to Cape Town. All data produced on site is picked up by transfer and trawler processes permanently running on each host in the cluster and pushed to the main archive.

---

[17] The spectral imager has produced images up to 23 040 pixels per side on an NVIDIA A100 GPU.

[18] `https://github.com/ska-sa/katsdpimager`

[19] `https://github.com/ska-sa/katsdpmodels`

[20] `https://archive.sarao.ac.za/`

4. Data formats

We provide an overview of the way metadata and visibilities are stored and accessed in the MeerKAT SDP subsystem.

4.1. Metadata

The observation metadata is stored in a telescope state store (telstate) built on top of a Redis backend. It supports three types of metadata:

- immutable *attributes* that store a single value that cannot change once set,
- mutable *sensors* that store a sequence of timestamped values which can be extended, and
- *indexed* attributes that store a dictionary of key-value pairs, each of which behaves like an immutable, avoiding a cluttered key space and allowing more general keys than just strings.

The three telstate types are realized by the Redis string, sorted set and hash types, respectively. Python objects need to be encoded since Redis operates on bytestrings only. The keys are UTF-8 encoded strings while the values (and the sub-keys of indexed attributes) are encoded with MessagePack,[21] a fast and compact binary serialization format with a similar set of primitive types as JSON and the ability to add custom types. We extended it to support tuples, complex numbers and NumPy arrays. Telstate originally used pickle serialization but that was replaced by `msgpack` in 2019 due to security and compatibility concerns.

The sensor implementation deserves further mention. The Redis sorted set (or `zset`) maintains the ordering of entries based on a score, or preserves lexicographical order if the scores are all equal. The encoded sensor values are prefixed with a big-endian representation of the corresponding timestamps and the scores are set to zero, so that lexicographical order becomes chronological order as well. This representation makes the common request of a contiguous time range of sensor values very efficient, by using the ZREVRANGEBYLEX Redis command.

The telstate library supports Python's asynchronous I/O functionality and provides a memory backend that is useful for tests and for loading RDB files. It is implemented in the `katsdptelstate`[22] package.

4.2. Visibility data

The MeerKAT Visibility Format (MVF)[23] has had several iterations over the lifetime of the project. The undocumented version 0 was used on the XDM prototype single dish from 2006 to 2009 and was based on a sequence of FITS files. The next two versions (1 and 2) were used on the KAT-7 precursor array from 2009 to 2013 and were based on HDF5. All the data and metadata were contained in a single HDF5 file, which was convenient and feasible since KAT-7 only had seven dishes.

---

[21] `https://msgpack.org/`

[22] `https://github.com/ska-sa/katsdptelstate`

[23] `https://katdal.readthedocs.io/en/latest/data_set_format.html`

The MVF was upgraded to version 3 in preparation for the small Receptor Test System (RTS) which qualified the MeerKAT dishes. This version was used during early integration of the telescope from 2013 to 2017 and was still a monolithic HDF5 file. However, as data rates increased, the MVF3 format struggled to achieve the required write speed and the file also became unwieldy in size. HDF5 files could not easily be written in parallel, and virtual datasets (VDS) support[24] was still more than a year away.

The current MVF version 4 was introduced in 2017. It replaces the single HDF5 file with a collection of smaller files in NumPy's NPY format,[25] where each file represents a chunk of a large multidimensional NumPy array. Several of these arrays (containing e.g. visibilities, flags and weights) are stored in a *chunk store*, which could be a directory on a local disk or an S3 bucket in a Ceph object store. The chunks are loaded and reassembled in a lazy way with the help of the Dask[26] parallel computing library which MeerKAT SDP adopted early. This also addresses the problem of having datasets that are larger than the available physical memory.

The chunking scheme is stored in telstate with the rest of the metadata. The standard chunk size is around 15 MB, which was chosen to achieve good read performance from the S3 chunk store. Datasets larger than 15 TB will therefore be split into more than a million objects.

MVF4 datasets can be accessed with the `katdal`[27] data access library. The entry point to the dataset is a telstate instance; either a live Redis server or an RDB file. The library supports various chunk stores and can also calibrate the data on the fly using calibration solutions in telstate, by adding the appropriate tasks to the Dask task graph. This how L1 and L2 visibilities are obtained. The `katdal` library also reconciles the sensor metadata with the visibility data, allows advanced selection of the data, and converts MVF4 to other radio astronomy formats such as CASA MeasurementSets.

5. Conclusions

Some aspects of the pipeline appear to reimplement functionality found in established modern libraries. For example, the signal displays use a similar approach to the Bokeh[28] and Plotly[29] libraries to produce plots in the browser, and the `katdal` chunk store is very reminiscent of Zarr.[30] One has to bear in mind, though, that the development of these libraries happened in parallel and that it typically takes a few years before the utility of a new library is established. The MeerKAT implementations are optimized for the problem domain and the benefit of rewriting it to use the now established libraries has to be considered against the risk of supporting custom software in the long run.

---

[24] VDS support was added to h5py 2.9 at the end of 2018. It breaks the HDF5 file into multiple smaller files linked to the main file.

[25] `https://numpy.org/doc/stable/reference/generated/numpy.lib.format.html`

[26] `https://www.dask.org/`

[27] `https://github.com/ska-sa/katdal`

[28] `https://bokeh.org/`

[29] `https://plotly.com/python/`

[30] `https://zarr.dev/`

Some design decisions might also need to be reconsidered in future. Apache Mesos has not had a release in four years, while tools like Kubernetes[31] are now popular for container orchestration. The use of the Redis-specific RDB format as the basis for MVF4 metadata is also a risk, given that the RDB parser library[32] used by `katdal` also hasn't had a release in four years. This risk might be mitigated by the development of a new officially sanctioned RDB parser.[33]

Overall, the containerized architecture of the SDP subsystem is a flexible and scalable approach to high-performance computing. It integrates both GPU and CPU resources, as well as large storage and high-speed networks. It is quick to reconfigure and resilient to long-distance link outages. It is well placed to support MeerKAT science for the rest of its operational lifespan, as well as the commissioning of new dishes for the MeerKAT Extension and SKA-Mid as part of the Dish Verification System (DVS).

Acknowledgments. The authors would like to thank Bill Cotton (NRAO) for his invaluable assistance with the continuum imager by providing the Obit package, testing the `katsdpcontim` wrapper, and helping to tune the imaging parameters. The MeerKAT radio telescope is a facility operated by the South African Radio Astronomy Observatory, a business unit of the National Research Foundation (NRF) of South Africa.

References

Christy, C. T., et al. 2024, ApJ, 974, 18. 2404.12431
Cornwell, T. J., Golap, K., & Bhatnagar, S. 2008, IEEE Journal of Selected Topics in Signal Processing, 2, 647. 0807.4161
Cotton, W. D. 2008, PASP, 120, 439
Jonas, J. L., & the MeerKAT Team 2016, in MeerKAT Science: On the Pathway to the SKA, 1
Marais, N. 2015, in 15th International Conference on Accelerator and Large Experimental Physics Control Systems, MOPGF067
Mauch, T., et al. 2020, ApJ, 888, 61. 1912.06212
Merry, B. 2016a, MNRAS, 456, 1761. 1511.07152
— 2016b, Astronomy and Computing, 16, 140. 1605.07023
Offringa, A. R., de Bruyn, A. G., Biehl, M., Zaroubi, S., Bernardi, G., & Pandey, V. N. 2010, MNRAS, 405, 155. 1002.1957
Offringa, A. R., et al. 2014, MNRAS, 444, 606. 1407.1943
Salvini, S., & Wijnholds, S. J. 2014, A&A, 571, A97. 1410.2101
van der Byl, A., et al. 2022, Journal of Astronomical Telescopes, Instruments, and Systems, 8, 011006

[31] `https://kubernetes.io/`

[32] `https://pypi.org/project/rdbtools/`

[33] `https://github.com/redis/librdb`