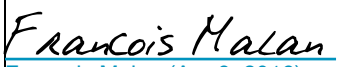
SDP Memo: iPython Performance Model Handbook

Document Number	SKA-TEL-SDP-0000041
Document Type	MNL
Revision	01C
Author	Francois Malan
Release Date	2016-04-08
Document Classification	Unrestricted
Status	Draft

Lead Author	Designation	Affiliation
Francois Malan	SE Team Member	SKA SA
Signature & Date:	 Francois Malan (Apr 8, 2016) francois@scs-space.com	

Released by	Designation	Affiliation
Rosie Bolton	SDP Project Scientist	University of Cambridge
Signature & Date:	 Rosie Bolton (Apr 8, 2016) rosie@mrao.cam.ac.uk	

Version	Date of Issue	Prepared by	Comments
1	2015-02-09	F Malan	PDR Submission
1A	2015-05-29	R Bolton	Minor fix post PDR
01C	2016-04-08	F Malan	Updated for SDP delta-PDR

ORGANISATION DETAILS

Name	Science Data Processor Consortium

Table of Contents

List of Figures

List of Tables

1 Introduction

This document is intended as supporting material for [AD01]. This document explains how the SDP performance model was implemented as an iPython notebook, and how to use this model to compute results.

The primary purposes of the performance model are to estimate the required amount of storage, bandwidth, processing power and electricity that are needed to build and operate the SDP. These parameters in turn inform the cost estimates of the SDP compute platform.

2 References

2.1 Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

Reference Number	Reference
AD01	SKA-TEL-SDP-0000040 Parametric Models of SDP Compute Requirements

2.2 Reference Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	Reference
RD01	SKA-TEL-SDP-0000043 SDP Cost Model
RD02	SKA-TEL-SDP-0000017 Theoretical analysis of baseline-dependent averaging
RD03	SKA-TEL-SDP-0000038 SDP System Sizing

3 What are “Python”, “IPython” and “Jupyter Notebook”?

The SDP parametric model is a collection of equations and software code that model the performance aspects of the SDP. We created an interactive software interface for performing computations and displaying their results so that people within SDP can easily interact and experiment with the model. Python, IPython and the Jupyter Notebook are the tools that we used to accomplish this.

Interactive functionality is implemented in IPython, a command shell for interactive computing. Although IPython currently supports an extensive set of programming languages, Python is the only programming language used in the SDP Parametric model.

Jupyter Notebook (formerly known as IPython Notebook) is a web application that allows you to create “notebook” documents that contain live code, equations, visualisations and explanatory text. Being a web application, Jupyter runs in your web browser while in the background interfacing with a “kernel”. When you open a notebook on your own computer a kernel will automatically start up and run locally. Your web browser will open a new tab that interfaces to this local kernel (without requiring an internet connection).

(IPython) Jupyter notebook sessions may be saved as files with the .ipynb extension. A notebook file can be shared with other people in much the same way as MS Word or PDF document. A notebook including all formatted text, output and plotted figures can be exported to a static PDF file. Alternatively, a notebook may be used to interactively re-compute results (when used in conjunction with the accompanying code).

For an overview of The IPython project, see: http://ipython.org/

For an overview of Project Jupyter, see: http://jupyter.org/

The “SDP parametric model” contains several Jupyter Notebook files. These will be briefly listed and explained later in this document. In the rest of this document either “IPython” and/or “notebook” will refer to our Python and IPython Jupyter Notebook implementation as a whole.

A central goal of the notebook is to compute performance metrics and general sizing estimates for the SDP. We attempted a 1:1 implementation of the equations described in AD01. As the notebook is being developed in parallel with AD01, some discrepancies may occur. Refer to AD01 for the fundamental choices in which algorithms are modelled. The w-snapshot algorithm with the use of faceting is assumed. Certain parameters may be toggled, e.g. where uncertainty remains (e.g. in choices relating to baseline-dependent convolution kernel sizes and baseline-dependent gridding coalescing).

4 Obtaining a copy of the SDP parametric model notebook

The IPython Notebook is hosted on Github as part of the following "SDP-par-model" project. You'll need to be a member of the SKA Github group in order to access this repository: https://github.com/SKA-ScienceDataProcessor/sdp-par-model

If you're not familiar with using git, a user-friendly GUI such as Atlassian Sourcetree (available for Mac & Windows) is a way to get started. A list of other git GUIs can be found here.

The development of the IPython model has so far been carried out by many sometimes-overlapping contributions. Direct contributors to the code repository include Rosie Bolton, Anna Scaife, Bojan Nikolic, Juande Santander-Vela, Francois Malan, Peter Wortmann and Tim Cornwell. These contributions are often committed to different branches that are later merged. The most up-to-date working copy of the model is typically represented by the head of the "master" branch.

The evolution of the IPython model is recorded in git as a series of immutable revisions that is each uniquely identified by a 40-character ID. We may refer to a revision by the first 7 characters of its ID, e.g. [9ffbb2e]. Descriptive git "tags" are additionally used to identify major milestones. Because the IPython model is continually evolving, each document that uses the model's results (e.g. RD03) should reference the unique revision ID that was used for its generation.

For Delta-PDR the relevant commit ID is 9ffbb2ed872bc5cc405f1ca84c511d7dfc5b2124, i.e. [9ffbb2e] in shorthand. This version was committed on 24 March 2016 and is additionally identified by the tags "deltaPDR" and "20160324_System_Sizing."

5 Overview of the available SDP notebook files

In revision [9ffbb2e] there are several notebook files (all with extension .ipynb) that illustrate different parts of the Model. Ranked in order of typical importance (high to low) these are:

5.1 SKA1_Imaging_Performance_Model

This notebook contains cells for computing tables of values (data rates, FLOPs, image sizes etc), as well as for generating 1D and 2D parameter sweep plots (as used in some of the workshop presentations of Rosie Bolton and Paul Alexander). Computed values can be exported for external analysis.

5.2 SKA1_SDP_DesignEquations

This algorithmically constructs the "design equations" contained in AD01 by using the IPython model's underlying code. Symbolic equations are formatted in human-readable format, while numerical limits corresponding to these equations are also computed.

5.3 SKA1_Dataflow

This creates a graphical (graph) representation of the SDP data pipeline. It uses the IPython model to illustrate numerical data rates, computational loads, and various other parameters in the output. The graph can be exported as a PDF file or PNG image (as out.pdf and out.png in the notebook directory). This notebook have not kept pace with recent developments and therefore revision [9ffbb2e] tagged for delta-PDR may issue an error. Refer to revision [4ab3ded] for an example of a working copy of this notenbook.

5.4 SKA1_Faceting_Model

This investigates the influence of faceting in the SKA Performance Model. It generates a host of values and 1D and 2D plots. This notebook provides the plots for the TCC memo TCC-SDP-151123-1-1.

5.5 SKA1_Sensitivity_Analysis

This investigates the parameter sensitivity in the SKA Performance Model. This notebook may not run in its current form as it has not kept up with recent changes to the code.

5.6 Absolute_Baseline_length_distribution

This notebook reads antenna station locations (as latitude / longitude coordinates) from text file (located in the /Reference_layouts subdirectory) to compute and visualize the baseline distributions.

5.7 Other (obsolete) notebook files

5.7.1 SKA1_Performance_Model_PDR05

This notebook was used in the original PDR of February 2015 and is now obsolete and unable to recompute some values due to changes in the accompanying Python files. To re-evaluate this model you will need to consult an older revision of the entire IPython model (e.g. [f42e653]).

5.7.2 SKA1_SDP_Products

This is a short experimental document that computes FLOP values for some of the imaging pipeline components of a specified telescope.

5.7.3 SKA1_SDP_DesignEquations_False

An obsolete file that will be removed in future.

5.7.4 Reference notebooks

Additional non-SKA reference notebooks are located in */IPython/Reference Notebooks.* Some of these notebooks are obsolete SKA SDP-related notebooks, while others showcase examples of what can be done using the IPython Jupyter Notebook environment (e.g. Nature.ipynb).

6 Overview of the Python code (the computational engine)

Python code can be executed directly in a notebook and is indeed entered and executed directly in some of the notebooks listed in the previous section. It is, however, best practice to keep the bulk of the code separate from the notebooks themselves. This practice facilitates easier code writing, better integration with integrated development environments (such as PyCharm), better coding practice, easier source control, and more advanced features (such as unit testing).

Code that is maintained in proper Python code libraries (with .py extension) can be imported into a notebook using standard Python syntax. This is typically done in the first code block of a notebook file.

Two important concepts are those of “PipelineConfig” and “ParameterContainer”. These are Python classes (defined in *implementation.py* and *parameter_definitions.py*) that respectively contain the relevant telescope configuration and the associated formulas. These two objects are passed to the relevant methods that act on them, analyze them, or display their contents. We typically assign the variable name “o” to one of their instances, subsequently using the prefix “o.”

to refer to fields. (The reasoning behind this name was that “o” is the shortest abbreviation for the generic name “object”, and doesn’t impact the readability of the code too much).

Below follows the general structure of the IPython Model’s code base, ranked in order of decreasing importance:

6.1 equations.py

This **crucially important** file contains a single class, “**Equations**”, that contains most of the actual equations that are described in [AD01]. Equations are grouped into parameter classes and pipeline components by methods, e.g. _apply_channel_equations() or _apply_coalesce_equations().

6.2 parameter_definitions.py

This contains the generic “ParameterContainer” class definition, along with several classes enumerating and supplying the predefined input parameters, such as the:

- telescopes and their properties
- available imaging modes
- imaging Modes
- high priority science objectives
- physical constants

6.3 implementation.py

This contains two classes: **PipelineConfig** and **Implementation**. These classes provide the machinery by which equations are evaluated, error checking is performed, minimization of a given metric (e.g. Flops) over a given parameter (e.g. Snapshot time), etc.

6.4 api.py

This contains a class (SkaPythonAPI) that provides an easy-to-comprehend application program interface (API) by which the SKA Parametric Model can be invoked. Functions that may be performed include:

- Evaluating an expressions for a specific set of parameters
- Performing a parameter sweep to see how an expression’s output changes for a range of input values

6.5 api_ipython.py

This contains a class (SkaIPythonAPI) that extends the basic SkaPythonAPI with functions specific to the IPython Notebook environment. Examples include:

- Formatting results
- Plotting results (as tabulated values, or as pie charts or other types of figures)
- Providing code for setting up frequently-used GUI elements

6.6 dataflow.py

This is a helper module for reasoning with large data flow graphs. It creates a description of pipelines as networks of nodes and edges enumerated by/aligned to regions in domains such as time, baseline, frequency or loop iteration. It is used by pipeline.py in order to explore the properties of the pipeline data flow.

6.7 pipeline.py

This uses the dataflow.py framework in order to generate a model for the data flow pipeline arising from a certain pipeline configuration. It verifies that the parametric model predictions match the properties of the data flow.

6.8 design_equations.py

This is a nearly-empty file containing only a list of symbolic variables (used in SKA1_SDP_DesignEquations) with their display formatting. Will in future most likely be integrated into parameter_definitions.py.

7 Opening, displaying and printing the notebook

7.1 Viewing the IPython notebook without installing anything

Setting up your own IPython environment on your local computer involves a certain amount of one-off effort (see "Setting up the IPython environment" below). Luckily, it is possible to view IPython Notebook files without installing IPython.

This can be done in at least two ways, both of which work well for statically reading a notebook but don't allow executing any of the cells:

- Clicking on a notebook file in github's web interface opens the file in github's own web-based viewer
- using a web-based service called the nbviewer, available at http://nbviewer.ipython.org/. For this you'll need to provide it with a URL pointing to your notebook file (e.g. a dropbox link).

7.2 Setting up a local IPython environment

To interactively open or run the IPython Notebook document on your own computer you will need to have IPython installed with Python 2.x (at the time of writing this was version 2.7*). Instructions can be found at http://ipython.org/install.html which cover Windows, Mac and Linux.

For new users we suggest installing the Anaconda distribution, which provides Python 2.7, IPython and all of its dependences as well as several relevant packages, including **Numpy, Scipy and Matplotlib**. Anconda can be downloaded from http://continuum.io/downloads.

Additional packages that are used in the IPython performance model include **sympy** and **graphviz**. If using Anaconda, these can be installed using ***conda install <pkg name>*** e.g. "*conda install sympy*". Dependencies are automatically handled by anaconda's package manager. If you use a different Python installation, please refer to the relevant documentation (e.g. using *pip* or *easy_install*).

To generate data flow images using the SKA1_Dataflow Notebook, you will also need the GraphViz visualisation tool installed. It is available from http://graphviz.org/Download.php. Make sure that the iPython kernel can find the "dot" program on the program search path.

*In future we will probably migrate the whole IPython model to Python 3.x (which may cause some minor backwards-compatibility issues with the existing codebase and require all users to switch their Python version as well in order to run the models).

7.3 Locally opening a notebook

Assuming that the Jupyter IPython notebook environment has been correctly set up (see “Setting up your local IPython environment”), a notebook may be opened by running the following command from the command line:

>> jupyter notebook

This will open a new tab in your default web browser. This tab has a clickable interface with file browser. Navigating to the relevant directory and clicking on the .ipynb notebook file will allow you to open the notebook, which will then display in its own tab.

When the notebook is opened, it will display similar to a regular HTML document. Figures and output that were previously saved can be viewed without running any commands. The notebook may be printed or exported.

8 Running code blocks (requires a local IPython installation)

An explanation of the IPython user interface is given here. A cell is executed by selecting it and pressing Shift+Enter.

All cells may be executed in a single command, using the toolbar command Cell -> Run All.

However, it is advisable to run the topmost code block and *waiting for it to finish before* running any of the subsequent blocks, as this code block performs necessary set-up of the namespace and environment required by subsequent blocks.

Most of the actual computation takes place via methods defined in the accompanying .py Python libraries.

9 Features of the current implementation

9.1 Symbolic equations

Generally, our model works by internally constructing symbolic equations (via Python’s sympy package) of the parameters that need to be computed. These equations are used for optimizing free parameters and for evaluating results numerically.

Symbolically defined parameters may alternatively be displayed as symbolic equations, which may or may not be useful, e.g. for comparing results to the Parametric model derivations or definitions.

9.2 Optimising Tsnap (to minimise computational load)

Snapshot time, Tsnap, is a free variable that is optimized for each set of parameters. Increasing the snapshot time (“Tsnap”) has two opposing effects. It:

- increases the parameter “DeltaW_max”, the W deviation catered for by W kernel. This increases the size of the support of the w kernel, and consequently the total linear size of the convolution kernel. This increases Rccf and Rgrid, the FLOPS required for constructing the convolution kernels and applying the convolution.
- decreases the amount of FLOPS required for performing FFTs (doubling Tsnap halves this value).

The snapshot time is therefore a parameter that may be optimised for each set of telescope parameters to minimize the main cost driver -- typically assumed to be the computational load (Rflop).

This optimisation is performed in the the "`find_optimal_Tsnap_Nfacet`" method in Implementation.py, and setting the expression to minimize equal to "Rflop".

9.3 Inclusion of faceting, and optimising Nfacet

The use of faceting is described in AD01. The parameter "Nfacet" is the side length of the square facets that are used - for example, if Nfacet=2, we divide the field of view into 2x2 =4 separate fields and recombine these after FFT. It enables the target grid size to be reduced significantly as the field of view supported at gridding is smaller (there are $Nfacet^2$ times fewer pixels), with the trade-off that one must perform multiple ($Nfacet^2$ times *more*) parallel sets of gridding. The computational savings can arise because the convolution functions required to support the field of view are smaller with faceting, and the smaller field of view also allows additional time averaging at the gridding step, and (for continuum modes) fewer frequency channels at gridding step also. There is a further additional cost of phase-rotating the visibilities prior to gridding. In small field-of-view modes, especially in the spectral line case where additional frequency averaging is not supported, introducing faceting does not decrease the required FLOP rate but significant FLOP savings can be made in the large field of view continuum modes.

Each time a new facet number is explored, the optimal value of Tsnap can, in principle, change. Decreasing the flop rate may not be the (only) reason for applying facets. We possibly won't have enough memory available to keep one image plane in memory. In that case, facets provide another axis of distribution.

At this time (delta-PDR delivery) the IPython model is able to optimise Tsnap using a nonlinear optimization technique, but not Nfacet. As Nfacet can only assume integer values over a small finite range, it is feasible to minimize Nfacet in a brute-force way. We implement this by exploring increasing values of Nfacet until the compute load begins to increase again.

This optimisation is performed in the the "`find_optimal_Tsnap_Nfacet`" method in Implementation.py, and setting the expression to minimize equal to "Rflop".

9.4 Handling baseline dependence

Several of the derived values in the performance model are strongly affected by the baseline length they accommodate. Some depend only on the longest baseline being imaged (e.g. the final cell size in the map). For others, such as the gridding step (and calculation of convolution functions) we use values appropriate for the specific baselines being used. This means that at gridding we should attempt to use values for the kernel size required to support w deviations (Ngw), time averaging, and (for continuum modes,) frequency averaging which varies with baseline. This significantly reduces the gridding load from the short baselines, which dominate the baseline numbers. We have described in [RD02] that we believe there is a good theoretical basis for doing baseline dependent time averaging not just at the gridding step, but at *ingest,* giving significant cost savings in the buffer size required to store a full observation's uv data.

We currently do not consider every individual baseline in turn - instead we use a binned fractional baseline distribution for each telescope (based on antenna layouts from the SKAO ECP page[1] and derived using several different but hopefully typical observational setups). In each case we binned baselines into bins of similar length, e.g. grouping baselines shorter than a certain minimum length (~ 5km) into a single bin. This simplifies the calculation compared to treating every single baseline separately.

The tables below show the fractional baseline numbers in each baseline bin that have been used in the model. Roughly 50% of the baselines for each telescope are in the shortest (≤ 5km) baseline length bin.

Table 1: Baseline distribution used for SKA1 LOW

Baseline bin : lower bound (km)	Baseline bin : upper bound (km)	Fraction of baselines in this bin
0	4.9	49.4%
4.9	7.1	7.2%
7.1	10.4	7.8%
10.4	15.1	5.8%
15.1	22.1	10.5%
22.1	32.2	9.2%
32.2	47.0	8.1%
47.0	68.5	2.0%
68.5	100	0.1%

[1] https://skaoffice.atlassian.net/wiki/display/EP/ECP+Register see ECP140021, ECP140022, ECP140023

Table 2: Baseline distribution used for SKA1 MID

Baseline bin : lower bound (km)	Baseline bin : upper bound (km)	Fraction of baselines in this bin
0	4.4	57.5%
4.4	6.7	5.2%
6.7	10.3	5.6%
10.3	15.7	5.7%
15.7	24.0	6.1%
24.0	36.7	5.8%
36.7	56.0	6.4%
56.0	85.6	5.9%
85.6	130.8	1.8%
130.8	200	0.1%

In the IPython model we generate formulas that sum over all baseline bins. Such summation terms appear in:

- Memory requirements
- I/O rate
- FLOP requirements for:
    - Gridding
    - Calculation of convolution kernels
    - FFT
    - Reprojection and
    - Phase rotation

10 Limitations of the IPython model

10.1 Automated verification of results

The current IPython and Python codebase does not implement unit tests to verify whether changes to the code introduce errors. As the complexity of the software grows it becomes harder and more time-consuming to check the generated results for consistence

10.2 Execution speed

Some of the older versions of the IPython Notebook executed very slowly due to the symbolic equations that grew too large for Sympy to handle efficiently. This issue has been significantly improved since the original PDR submission in 2015. However, code execution is still slower than it could be; for example equations.py takes a full second to execute mainly because of sympy creating the large sum terms for the baseline bins. This issue prevents the baselines to be binned into narrower individual bins.

10.3 Baseline-dependent time and frequency averaging

Time- and frequency averaging are currently being modelled, for example in the gridding step. Our current understanding of the way in which averaging will be implemented is not yet complete, and therefore this aspect will require more rigorous analysis and possibly future adjustments to the model. Time and frequency averaging usually have large impacts on the sizing of the SDP.

10.4 Linking results to power consumption and monetary cost

High level requirements of the SDP may be specified in the amount of electrical power that is available, or in the monetary cost (both in capital and operational cost). The IPython model currently computes computational rates and data rates that have to be converted to power and cost via a separate cost model.

10.5 Optimizing SDP parameters according to criteria other that FLOPs

Currently the optimal values for free parameters such as *Tsnap* and *Nfacet* are determined according to an optimization process that minimizes the estimated peak FLOP rate of the relevant mode of the SDP. It may be desirable to optimize according to different criteria, such as data rate, buffer size or any user-defined other parameter. This is possible to implement in code, but not an option that is supplied in any of the current notebooks

11 Examples of visual output generated by the model

Plotting output in the IPython Notebook is performed via the integrated matplotlib environment. We can, for example, plot pie charts, or parameter dependency graphs (in 1D or 2D). Refer to the individual IPython notebooks for more examples than the ones shown here:

11.1 Visualizing pipeline dataflow

Generated with SKA1_Dataflow.ipynb revision [4ab3ded]

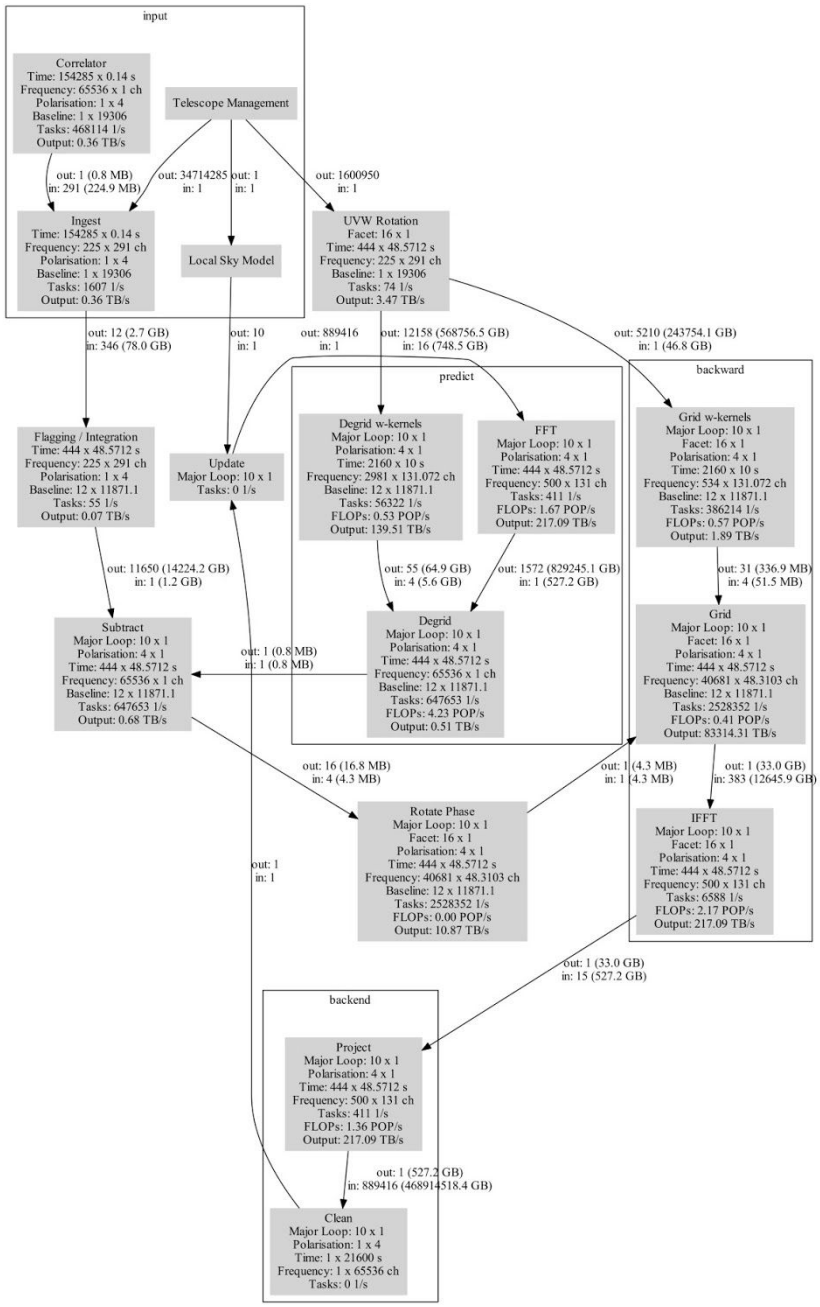


Figure 1: Visualizing SDP Pipeline Data Flow

11.2 Comparing FLOP requirements

Generated with SKA1_Imaging_Performance_Model.ipynb revision [9ffbb2e]

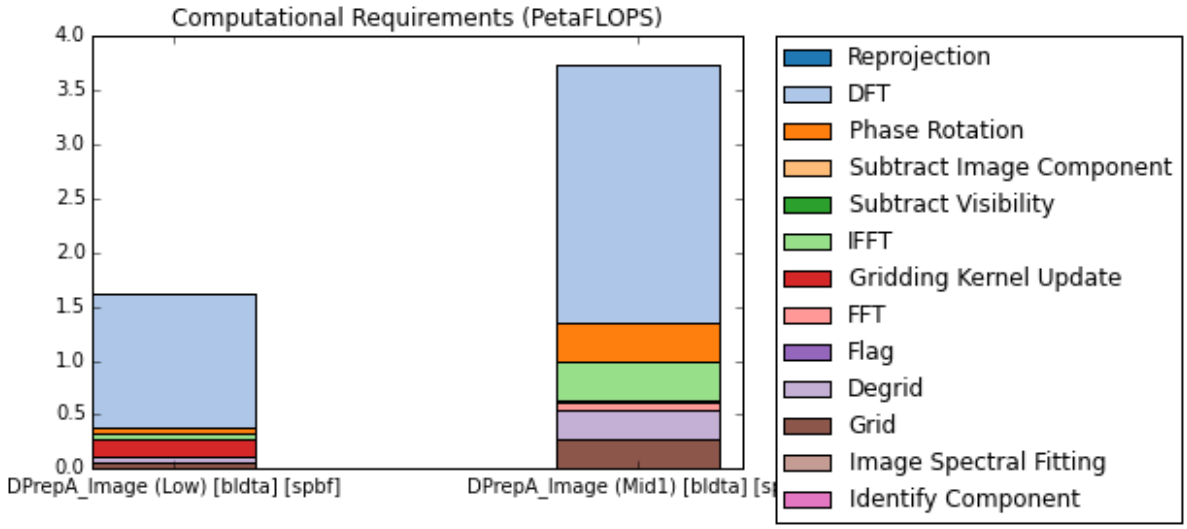


Figure 2: Comparing the FLOP requirements of two telescopes side-by-side

11.3 Analysing High Priority Science Objectives

Generated with SKA1_Imaging_Performance_Model.ipynb revision [9ffbb2e]

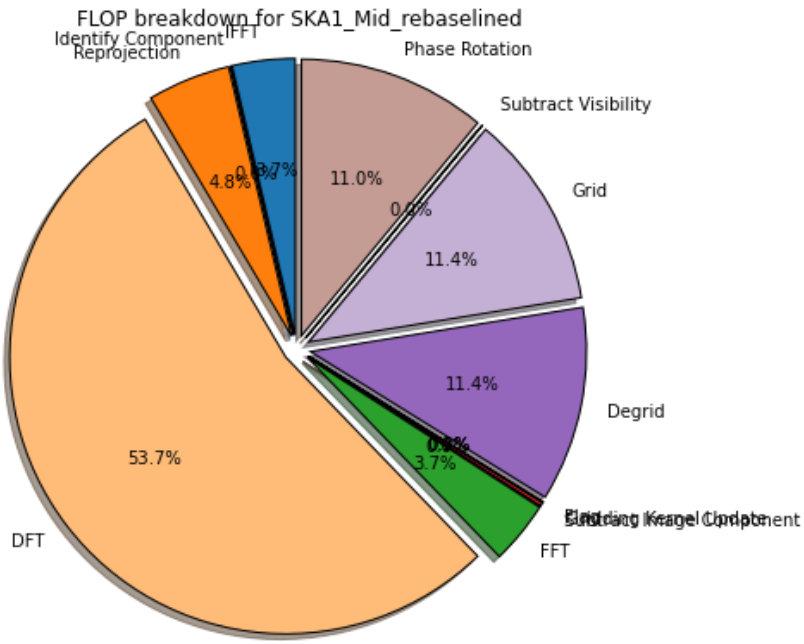


Figure 3: Analysis of the FLOP for a high priority science objective (37a)

Table 3: Analysis of the FLOP for a high priority science objective (37a)

-- Parameters --	
Telescope	SKA1_Mid_rebaselined
Band	HPSO 37a
Pipeline	DPrepA
BL-dependent averaging	True
On-the-fly kernels	False
Scale predict by facet	True
Max # of channels	65536
Max Baseline	150000 m
Snapshot Time	334 s
Facets	7
-- Image --	
Facet side length	3.63e+04 pixels
Image side length	2.12e+05 pixels
-- I/O --	
Visibility Buffer	7.77 PetaBytes
Working (cache) memory	0.2 TeraBytes
Visibility I/O Rate	106 TeraBytes/s
Inter-Facet I/O Rate	0.553 TeraBytes/s
-- Compute --	
Total Compute Requirement	4.4 PetaFLOPS
-> DFT	2.36 PetaFLOPS
-> Degrid	0.501 PetaFLOPS
-> FFT	0.161 PetaFLOPS
-> Flag	0.00251 PetaFLOPS

-> Grid	0.501 PetaFLOPS
-> Gridding Kernel Update	0.00974 PetaFLOPS
-> IFFT	0.161 PetaFLOPS
-> Identify Component	0.000132 PetaFLOPS
-> Phase Rotation	0.485 PetaFLOPS
-> Reprojection	0.213 PetaFLOPS
-> Subtract Image Component	1.71e-07 PetaFLOPS
-> Subtract Visibility	0.00158 PetaFLOPS

11.4 A 1D Parameter Sweep

Generated with SKA1_Imaging_Performance_Model.ipynb revision [9ffbb2e]

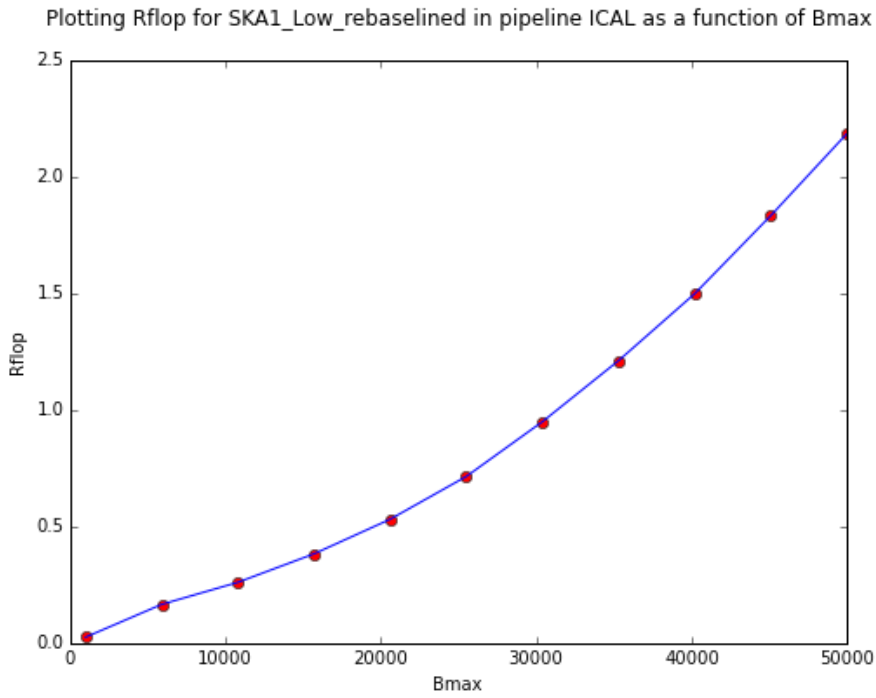


Figure 4: An example of a 1D parameter sweep plot

11.5 A 2D Parameter Sweep

Below is an example of a 2D parameter sweep generated with SKA1_Imaging_Performance_Model.ipynb revision [9ffbb2e]. This data can also be plotted as a 3D surface plot, although such a plot is typically harder to interpret than the 2D contour plot shown below.

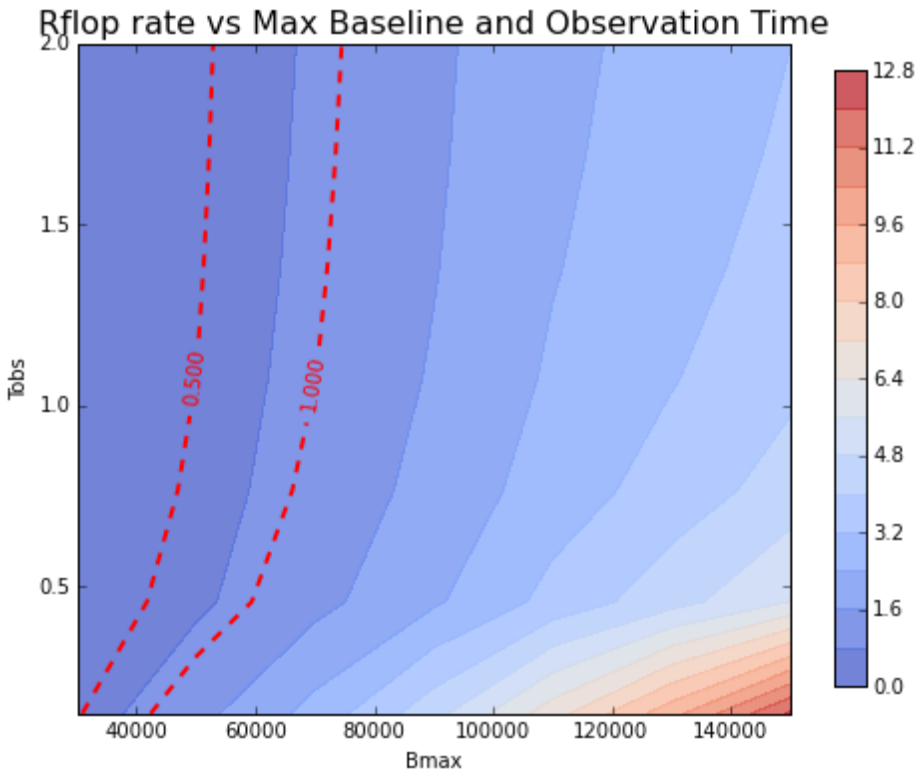


Figure 5: An example of a 2D parameter sweep contour plot