montblanc Documentation

Release 0.4.0-alpha3

Simon Perkins

Nov 13, 2017

Contents

Contents:

CHAPTER 1

Requirements

If you wish to take advantage of GPU Acceleration, the following are required:

- CUDA 8.0.
- cuDNN 6.0 for CUDA 8.0.
- A Kepler or later model NVIDIA GPU.

CHAPTER 2

Installation

Certain pre-requisites must be installed:

2.1 Pre-requisites

- GPU Acceleration requires CUDA 8.0 and cuDNN 6.0 for CUDA 8.0.
  - It is often easier to CUDA install from the NVIDIA site on Linux systems.
  - You will need to sign up for the NVIDIA Developer Program to download cudNN.

  During the installation process, Montblanc will inspect your CUDA installation to determine if a GPU-supported installation can proceed. If your CUDA installation does not live in `/usr`, it helps to set a number of environment variables for this to proceed smoothly. For example, if CUDA is installed in `/usr/local/cuda-8.0` and cuDNN is unzipped into `/usr/local/cudnn-6.0-cuda-8.0`, run the following on the command line or place it in your `.bashrc`

```
# CUDA 8
$ export CUDA_PATH=/usr/local/cuda-8.0
$ export PATH=$CUDA_PATH/bin:$PATH
$ export LD_LIBRARY_PATH=$CUDA_PATH/lib64:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH=$CUDA_PATH/extras/CUPTI/lib64/:$LD_LIBRARY_PATH

# CUDNN 6.0 (CUDA 8.0)
$ export CUDNN_HOME=/usr/local/cudnn-6.0-cuda-8.0
$ export C_INCLUDE_PATH=$CUDNN_HOME/include:$C_INCLUDE_PATH
$ export CPLUS_INCLUDE_PATH=$CUDNN_HOME/include:$CPLUS_INCLUDE_PATH
$ export LD_LIBRARY_PATH=$CUDNN_HOME/lib64:$LD_LIBRARY_PATH

# Latest NVIDIA drivers
$ export LD_LIBRARY_PATH=/usr/lib/nvidia-375:$LD_LIBRARY_PATH
```

  If the installer cannot find CUDA it will only install the CPU version of tensorflow and only **compile multi-threaded CPU operators**.

- casacore and the measures found in casacore-data. Gijs Molenaar has kindly packaged this on Ubuntu/Debian style systems.

  On Ubuntu 14.04, these packages can be added from the radio astronomy PPA :

  ```
  $ sudo apt-get install software-properties-common
  $ sudo add-apt-repository ppa:radio-astro/main
  $ sudo apt-get update
  $ sudo apt-get install libcasacore2-dev casacore-data
  ```

  On Ubuntu 16.04 these packages can be added from the kernsuite PPA:

  ```
  $ sudo apt-get install software-properties-common
  $ sudo add-apt-repository ppa:kernsuite/kern-1
  $ sudo apt-get update
  $ sudo apt-get install casacore-dev casacore-data
  ```

  Otherwise, casacore and the measures tables will need to be manually installed.

- Check that the python-casacore and casacore dependencies are installed. By default python-casacore builds from pip and therefore from source. To succeed, library dependencies such as `libboost-python` must be installed beforehand. Additionally, python-casacore depends on casacore. Even though kernsuite installs casacore, it may not install the development package dependencies (headers) that python-casacore needs to compile.

2.2 Installing the package

Set the `CUDA_PATH` so that the setup script can find CUDA:

```
$ export CUDA_PATH=/usr/local/cuda-8.0
```

If `nvcc` is installed in `/usr/bin/nvcc` (as in a standard Ubuntu installation) or somewhere on your `PATH`, you can leave `CUDA_PATH` unset. In this case setup will infer the CUDA_PATH as `/usr`

It is strongly recommended that you perform the install within a Virtual Environment. If not, consider adding the `--user` flag to the following pip and python commands to install within your home directory.

```
$ virtualenv $HOME/mb
$ source virtualenv $HOME/mb/bin/activate
(mb) $ pip install -U pip setuptools wheel
```

Then, run:

```
(mb) $ pip install --log=mb.log git+git://github.com/ska-sa/montblanc.git@master
```

2.3 Installing the package in development mode

Clone the repository, checkout the master branch and pip install montblanc in development mode.

```
(mb) $ git clone git://github.com/ska-sa/montblanc.git
(mb) $ pip install --log=mb.log -e $HOME/montblanc
```

2.4 Possible Issues

- Montblanc doesn't use your GPU, install the GPU version of tensorflow or compile GPU tensorflow operators. The installation process attempts to find your CUDA install location. It will log information about where it thinks this is and which GPU devices you have installed. Check the install log generated by the `pip` commands given above to see why this fails, searching for "**Montblanc Install**" entries.

  It is possible to see if the GPU version of tensorflow is installed by running the following code in a python interpreter:

  ```
  import tensorflow as tf
  with tf.Session() as S: pass
  ```

  If tensorflow knows about your GPU it will log some information about it:

  ```
  2017-05-16 14:24:38.571320: I tensorflow/core/common_runtime/gpu/gpu_device.
  ↪cc:887] Found device 0 with properties:
  name: GeForce GTX 960M
  major: 5 minor: 0 memoryClockRate (GHz) 1.176
  pciBusID 0000:01:00.0
  Total memory: 3.95GiB
  Free memory: 3.92GiB
  2017-05-16 14:24:38.571352: I tensorflow/core/common_runtime/gpu/gpu_device.
  ↪cc:908] DMA: 0
  2017-05-16 14:24:38.571372: I tensorflow/core/common_runtime/gpu/gpu_device.
  ↪cc:918] 0:   Y
  2017-05-16 14:24:38.571403: I tensorflow/core/common_runtime/gpu/gpu_device.
  ↪cc:977] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX␣
  ↪960M, pci bus id: 0000:01:00.0)
  ```

- cub 1.6.4. The setup script will attempt to download this from github and install to the correct directory during install. If this fails do the following:

  ```
  $ wget -c https://codeload.github.com/NVlabs/cub/zip/1.6.4
  $ mv 1.6.4 cub.zip
  $ pip install -e .
  ```

- python-casacore is specified as a dependency in setup.py. If installation fails here:

  1. Check that the *python-casacore dependencies* are installed.
  2. You will need to manually install it and point it at your casacore libraries.

CHAPTER 3

Concepts

Montblanc predicts the model visibilities of an radio interferometer from a parametric sky model. Internally, this computation is performed via either CPUs or GPUs by Google's tensorflow framework.

When the number of visibilities and radio source is large, it becomes more computationally efficient to compute on GPUs. However, the problem space also becomes commensurately larger and therefore requires subdividing the problem so that *tiles*, or chunks, can fit both within the memory budget of a GPU and a CPU-only node.

3.1 HyperCubes

In order to reason about tile memory requirements, Montblanc uses hypercube to define problem `Dimension`, as well as the Schemas of input, temporary result and arrays.

For example, given the following expression for computing the complex phase ϕ.

$$n = \sqrt{1 - l^2 + m^2} - 1$$
$$\phi = e^{\frac{2\pi(ul+vm+wn)}{\lambda}}$$

we configure a hypercube:

```
# Create cube
from hypercube import HyperCube
cube = HyperCube()

# Register Dimensions
cube.register_dimension("ntime", 10000, description="Timesteps")
cube.register_dimension("na", 64, description="Antenna")
cube.register_dimension("nchan", 32768, description="Channels")
cube.register_dimension("npsrc", 100, description="Point Sources")

# Input Array Schemas
cube.register_arrays("lm", ("npsrc", 2), np.float64)
cube.register_arrays("uvw", ("ntime", "na", 3), np.float64)
```

```
cube.register_arrays("frequency", ("nchan",), np.float64)

# Output Array Schemas
cube.register_array("complex_phase", ("npsrc", "ntime", "na", "nchan"),
    np.complex128)
```

and iterate over it in tiles of 100 timesteps and 64 channels:

```
# Iterate over tiles of 100 timesteps and 64 channels
iter_args = [("ntime", 100), ("nchan", 64)]
for (lt, ut), (lc, uc) in cube.extent_iter(*iter_args):
    print "Time[{}:{}] Channels[{}:{}]".format(lt,ut,lc,uc)
```

This roduces the following output:

```
Time[0:100] Channels[0:64]
...
Time[1000:1100] Channels[1024:1088]
...
Time[9900:10000] Channels[32704:32768]
```

Please review the hypercube Documentation for further information.

3.2 Data Sources and Sinks

The previous section illustrated how the computation of the complex phase could be subdivided. Montblanc internally uses this mechanism to perform memory budgeting and problem subdivision when computing.

Each input array, specified in the hypercube and required by Montblanc, must be supplied by the user via a *Data Source*. Conversely, output arrays are supplied to the user via a *Data Sink*. Data Sources and Sinks request and provide tiles of data and are specified on *Source* and *Sink Provider* classes:

```
lm_coords = np.ones(shape=[1000,2], np.float64)
frequencies = np.ones(shape=[64,], np.float64)

class MySourceProvider(SourceProvider):
    """ Data Sources """
    def lm(self, context):
        """ lm coordinate data source """
        (lp, up) = context.dim_extents("npsrc")
        return lm_coords[lp:up,:]

    def frequency(self, context):
        """ frequency data source """
        (lc, uc) = context.dim_extents("nchan")
        return frequencies[lc:uc]

    def updated_dimensions(self):
        """ Inform montblanc about global dimensions sizes """
        return [("npsrc", 1000), ("nchan", 64),
            ("ntime" , ...), ("na", ...)]

class MySinkProvider(SinkProvider):
    """ Data Sinks """
    def complex_phase(self, context):
        """ complex phase data sink """
```

```
        (lp, up), (lt, ut), (la, ua), (lc, uc) = \
            context.dim_extents("npsrc", "ntime", "na", "nchan")

        print ("Received Complex Phase"
               "[{}:{},{}:{},{}:{},{}:{}]"
                    .format(lp,up,lt,ut,la,ua,lc,uc))
        print "Data {}", context.data
```

Important points to note:

1. Data sources return a numpy data tile with shape `SourceContext.shape` and dtype `SourceContext.dtype`. `SourceContext` objects have methods and attributes describing the *extents* of the data tile.
2. Data sinks supply a numpy data tile on the context's `SinkContext.data` attribute.
3. `AbstractSourceProvider.updated_dimensions()` provides Montblanc with a list of dimension global sizes. This can be used to set the number of Point Sources, or number of Timesteps.
4. `SourceContext.help()` and `SinkContext.help()` return a string providing help describing the data sources, the extents of the data tile, and (optionally) the hypercube.
5. If no user-configured data source is supplied, Montblanc will supply default values, [0, 0] for lm coordinates and [1, 0, 0, 0] for stokes parameters, for example.

3.3 Provider Thread Safety

Data Sources and Sinks should be thread safe. Multiple calls to Data sources and sinks can be invoked from multiple threads. In practice, this means that if a data source is accessing data from some shared, mutable state, that access should be protected by a `threading.Lock`.

CHAPTER 4

Computing the RIME

Montblanc solves the Radio Inteferometer Measurement Equation (RIME).

4.1 Example Source Providers

Although it is possible to provide custom *Source Providers* for Montblanc's inputs, the common use case is to specify parameterised Radio Sources.

Here is a Source Provider that supplies Point Sources to Montblanc in the form of three numpy arrays containing the lm coordinates, stokes parameters and spectral indices, respectively.

```
class PointSourceProvider(SourceProvider):
    def __init__(self, pt_lm, pt_stokes, pt_alpha):
        # Store some numpy arrays
        self._pt_lm = pt_lm
        self._pt_stokes = pt_stokes
        self._pt_alpha = pt_alpha

    def name(self):
        return "PointSourceProvider"

    def point_lm(self, context):
        """ Point lm data source """
        lp, up = context.dim_extents('npsrc')
        return self._pt_lm[lp:up, :]

    def point_stokes(self, context):
        """ Point stokes data source """
        (lp, up), (lt, ut) = context.dim_extents('npsrc', 'ntime')
        return np.tile(self._pt_stokes[lp:up, np.newaxis, :],
            [1, ut-lt, 1])

    def point_alpha(self, context):
        """ Point alpha data source """
```

```
        (lp, up), (lt, ut) = context.dim_extents('npsrc', 'ntime')
        return np.tile(self._pt_alpha[lp:up, np.newaxis],
            [1, ut-lt])

    def updated_dimensions(self):
        """
        Inform montblanc about the number of
        point sources to process
        """
        return [('npsrc', self._pt_lm.shape[0])]
```

Similarly, here is a Source Provider that supplies Gaussian Sources to Montblanc in four numpy arrays containing the lm coordinates, stokes parameters, spectral indices and gaussian shape parameters respectively.

```
class GaussianSourceProvider(SourceProvider):
    def __init__(self, g_lm, g_stokes, g_alpha, g_shape):
        # Store some numpy arrays
        self._g_lm = g_lm
        self._g_stokes = g_stokes
        self._g_alpha = g_alpha
        self._g_shape = g_shape

    def name(self):
        return "GaussianSourceProvider"

    def gaussian_lm(self, context):
        """ Gaussian lm coordinate data source """
        lg, ug = context.dim_extents('ngsrc')
        return self._g_lm[lg:ug, :]

    def gaussian_stokes(self, context):
        """ Gaussian stokes data source """
        (lg, ug), (lt, ut) = context.dim_extents('ngsrc', 'ntime')
        return np.tile(self._g_stokes[lg:ug, np.newaxis, :],
            [1, ut-lt, 1])

    def gaussian_alpha(self, context):
        """ Gaussian alpha data source """
        (lg, ug), (lt, ut) = context.dim_extents('ngsrc', 'ntime')
        return np.tile(self._g_alpha[lg:ug, np.newaxis],
            [1, ut-lt])

    def gaussian_shape(self, context):
        """ Gaussian shape data source """
        (lg, ug) = context.dim_extents('ngsrc')
        gauss_shape = self._g_shape[:,lg:ug]
        emaj = gauss_shape[0]
        emin = gauss_shape[1]
        pa = gauss_shape[2]

        gauss = np.empty(context.shape, dtype=context.dtype)

        # Convert from (emaj, emin, position angle)
        # to (lproj, mproj, ratio)
        gauss[0,:] = emaj * np.sin(pa)
        gauss[1,:] = emaj * np.cos(pa)
        emaj[emaj == 0.0] = 1.0
        gauss[2,:] = emin / emaj
```

```
        return gauss

    def updated_dimensions(self):
        """
        Inform montblanc about the number of
        gaussian sources to process
        """
        return [ ('ngsrc', self._g_lm.shape[0])]
```

These Source Providers are passed to the solver when computing the RIME.

4.2 Configuring and Executing a Solver

Firstly we configure the solver. Presently, this is simple:

```
import montblanc

slvr_cfg = montblanc.rime_solver_cfg(dtype='double',
    version='tf', mem_budget=4*1024*1024*1024)
```

dtype is either *float* or *double* and defines whether single or double floating point precision should be used to perform computation.

Next, the RIME solver should be created, using the configuration.

```
with montblanc.rime_solver(slvr_cfg) as slvr:
```

Then, source and sink providers can be configured in lists and supplied to the *solve* method on the solver:

```
with montblanc.rime_solver(slvr_cfg) as slvr:
    # Create a MS manager object, used by
    # MSSourceProvider and MSSinkProvider
    ms_mgr = MeasurementSetManager('WSRT.MS', slvr_cfg)

    source_provs = []
    source_provs.append(MSSourceProvider(ms_mgr, cache=True))
    source_provs.append(FitsBeamSourceProvider(
        "beam_$(corr)_$(reim).fits", cache=True))
    source_provs.append(PointSourceProvider)
    source_provs.append(GaussianSourceProvider)

    sink_provs = [MSSinkProvider(ms_mgr, 'MODEL_DATA')]

    slvr.solve(source_providers=source_provs,
        sink_providers=sink_provs)
```

CHAPTER 5

API

5.1 Contexts

Contexts are objects supplying information to implementers of Providers.

class InitialisationContext

Initialisation Context object passed to Providers.

It provides initialisation information to a Provider, allowing Providers to perform setup based on configuration.

```
class CustomSourceProvider(SourceProvider):
    def init(self, init_context):
        config = context.cfg()
        ...
```

cfg

Configuration

class StartContext

Start Context object passed to Providers.

It provides information to the user implementing a data source about the extents of the data tile that should be provided.

```
# uvw varies by time and baseline and has 3 coordinate components
cube.register_array("uvw", ("ntime", "nbl", 3), np.float64)

...

class CustomSourceProvider(SourceProvider):
    def start(self, start_context):
        # Query dimensions directly
        (lt, ut), (lb, ub) = context.dim_extents("ntime", "nbl")
        ...
```

Public methods of a HyperCube are proxied on this object. Other useful information, such as the configuration, iteration space arguments are also present on this object.

cfg
Configuration

class StopContext
Stop Context object passed to Providers.

It provides information to the user implementing a data source about the extents of the data tile that should be provided.

```
# uvw varies by time and baseline and has 3 coordinate components
cube.register_array("uvw", ("ntime", "nbl", 3), np.float64)

...

class CustomSourceProvider(SourceProvider):
    def stop(self, stop_context):
        # Query dimensions directly
        (lt, ut), (lb, ub) = context.dim_extents("ntime", "nbl")
        ...
```

Public methods of a HyperCube are proxied on this object. Other useful information, such as the configuration, iteration space arguments are also present on this object.

cfg
Configuration

class SourceContext
Context object passed to data sources.

It provides information to the user implementing a data source about the extents of the data tile that should be provided.

```
# uvw varies by time and baseline and has 3 coordinate components
cube.register_array("uvw", ("ntime", "nbl", 3), np.float64)

...

class UVWSourceProvider(SourceProvider):
    def __init__(self, uvw_data):
        # All UVW coordinates
        self._uvw_data = uvw_data

    def uvw(self, context):
        print context.help(display_cube=True)

        # Query dimensions directly
        (lt, ut), (lb, ub) = context.dim_extents("ntime", "nbl")
        # Get the cube extents, ignoring
        # last dimension which is always (0, 3)
        (lt, ut), (lb, ub), (_, _) = context.array_extents("uvw")
        # Return data tile from larger array
        return self._uvw_data[lt:ut, lb:ub, :]
```

Public methods of a HyperCube are proxied on this object. Other useful information, such as the configuration, iteration space arguments, expected array shape and data type, and the abstract array schema are also present on this object.

array_schema
The array schema of the array associated with this data source. For instance if *model_vis* is registered on a hypercube as follows:

```
# Register model_vis array_schema on hypercube
cube.register_array("model_vis",
    ("ntime", "nbl", "nchan", "ncorr"),
    np.complex128)

...
# Create a source context for model_vis data source
context = SourceContext("model_vis", ...)
...
# Obtain the array schema
context.array_schema == ("ntime", "nbl", "nchan", "ncorr")
```

cfg
Configuration

dtype
The expected data type of the array that should be produced by the data source

help(*display_cube=False*)
Get help associated with this context

Parameters **display_cube** (*bool*) – Add hypercube description to the output

Returns A help string associated with this context

Return type str

iter_args
Iteration arguments that describe the tile sizes over which iteration is performed. In the following example, iteration is occuring in tiles of 100 Timesteps, 64 Channels and 50 Point Sources.

```
context.iter_args == [("ntime", 100),
        ("nchan", 64), ("npsrc", 50)]
```

name
The name of the data source of this context.

shape
The expected shape of the array that should be produced by the data source

class **SinkContext**
Context object passed to data sinks.

Primarily, it exists to provide a tile of output data to the user.

```
class MySinkProvider(SinkProvider):
    vis_queue = Queue(10)

    ...
    def model_vis(self, context):
        print context.help(display_cube=True)
        # Consume data
        vis_queue.put(context.data)
```

Public methods of a HyperCube are proxied on this object. Other useful information, such as the configuration, iteration space arguments and the abstract array schema are also present on this object.

array_schema

The array schema of the array associated with this data source. For instance if *model_vis* is registered on a hypercube as follows:

```
# Register model_vis array_schema on hypercube
cube.register_array("model_vis",
    ("ntime", "nbl", "nchan", "ncorr"),
    np.complex128)

...
# Create a source context for model_vis data source
context = SourceContext("model_vis", ...)
...
# Obtain the array schema
context.array_schema == ("ntime", "nbl", "nchan", "ncorr")
```

cfg

Configuration

data

The data tile available for consumption by the associated sink

help(*display_cube=False*)

Get help associated with this context

Parameters **display_cube** (*bool*) – Add hypercube description to the output

Returns A help string associated with this context

Return type str

input

The dictionary of inputs used to produce `data`. For example, if one wished to find the antenna pair used to produce a particular model visibility, one could do the following:

```
def model_vis(self, context):
    ant1 = context.input["antenna1"]
    ant2 = context.input["antenna2"]
    model_vis = context.data
```

iter_args

Iteration arguments that describe the tile sizes over which iteration is performed. In the following example, iteration is occuring in tiles of 100 Timesteps, 64 Channels and 50 Point Sources.

```
context.iter_args == [("ntime", 100),
        ("nchan", 64), ("npsrc", 50)]
```

name

The name of the data sink of this context.

5.2 Abstract Provider Classes

This is the Abstract Base Class that all Source Providers must inherit from. Alternatively, the `SourceProvider` class inherits from AbstractSourceProvider and provides some useful concrete implementations.

class **AbstractSourceProvider**

close()
Perform any required cleanup

init(*init_context*)
Called when initialising Providers

name()
Return the name associated with this data source

sources()
Returns a dictionary of source methods, keyed on source name

start(*start_context*)
Called at the start of any solution

stop(*stop_context*)
Called at the end of any solution

updated_arrays()
Return an iterable/mapping of hypercube arrays to update

updated_dimensions()
Return an iterable/mapping of hypercube dimensions to update

This is the Abstract Base Class that all Sink Providers must inherit from. Alternatively, the `SinkProvider` class inherits from AbstractSinkProvider and provides some useful concrete implementations.

class AbstractSinkProvider

clear_cache()
Clears any caches associated with the sink

close()
Perform any required cleanup

init(*init_context*)
Called when initialising Providers

name()
Returns this data sink's name

sinks()
Returns a dictionary of sink methods, keyed on sink name

start(*start_context*)
Called at the start of any solution

stop(*stop_context*)
Called at the end of any solution

5.3 Source Provider Implementations

class MSSourceProvider
Source Provider that retrieves input data from a MeasurementSet

__init__(*manager*, *vis_column=None*)
Constructs an MSSourceProvider object

Parameters

- **manager** (MeasurementSetManager) – The MeasurementSetManager used to access the Measurement Set.
- **vis_column** (*str*) – Column from which observed visibilities will be read

antenna1 (*context*)
antenna1 data source

antenna2 (*context*)
antenna2 data source

flag (*context*)
Flag data source

frequency (*context*)
Frequency data source

observed_vis (*context*)
Observed visibility data source

parallactic_angles (*context*)
parallactic angle data source

ref_frequency (*context*)
Reference frequency data source

uvw (*context*)
Per-antenna UVW coordinate data source

weight (*context*)
Weight data source

class **FitsBeamSourceProvider**
Feeds holography cubes from a series of eight FITS files matching a filename_schema. A schema of 'beam_$(corr)_$(reim).fits' matches:

```
['beam_xx_re.fits', 'beam_xx_im.fits',
 'beam_xy_re.fits', 'beam_xy_im.fits',
  ...
  'beam_yy_re.fits', 'beam_yy_im.fits']
```

while 'beam_$(CORR)_$(REIM).fits' matches

```
['beam_XX_RE.fits', 'beam_XX_IM.fits',
  'beam_XY_RE.fits', 'beam_XY_IM.fits',
  ...
  'beam_YY_RE.fits', 'beam_YY_IM.fits']
```

Missing files will result in zero values for that correlation and real/imaginary component. The shape of the FITS data will be inferred from the first file found and subsequent files should match that shape.

The type of correlation will be derived from the feed type. Currently, linear ['xx', 'xy', 'yx', 'yy'] and circular ['rr', 'rl', 'lr', 'll'] are supported.

__init__ (*filename_schema*, *l_axis=None*, *m_axis=None*)
Constructs a FitsBeamSourceProvider object

Parameters

- **filename_schema** (*str*) – See *FitsBeamSourceProvider* for valid schemas
- **l_axis** (*str*) – FITS axis interpreted as the L axis. *L* and *X* are sensible values here. *-L* will invert the coordinate system on that axis.

- **m_axis** (*str*) – FITS axis interpreted as the M axis. *M* and *Y* are sensible values here. *-M* will invert the coordinate system on that axis.

beam_extents (*context*)
Beam extent data source

beam_freq_map (*context*)
Beam frequency map data source

ebeam (*context*)
ebeam cube data source

filename_schema
Filename schema

init (*init_context*)
Perform any initialisation

name ()
Name of this Source Provider

shape
Shape of the beam cube

updated_dimensions ()
Indicate dimension sizes

class **CachedSourceProvider**
Caches calls to data_sources on the listed providers

__init__ (*providers*, *cache_data_sources=None*, *clear_start=False*, *clear_stop=False*)

Parameters

- **providers** (*SourceProvider or Sequence of SourceProviders*) – providers containing data sources to cache
- **cache_data_sources** (*list of str*) – list of data sources to cache (Defaults to None in which case all data sources are cached)
- **clear_start** (*bool*) – clear cache on start
- **clear_stop** (*bool*) – clear cache on stop

init (*init_context*)
Perform any initialisation required

start (*start_context*)
Perform any logic on solution start

stop (*stop_context*)
Perform any logic on solution stop

updated_dimensions ()
Update the dimensions

5.4 Sink Provider Implementations

class **MSSinkProvider**
Sink Provider that receives model visibilities produced by montblanc

__init__(*manager*, *vis_column=None*)
Constructs an MSSinkProvider object

Parameters

- **manager** (`MeasurementSetManager`) – The `MeasurementSetManager` used to access the Measurement Set.
- **vis_column** (*`str`*) – Column to which model visibilities will be read

model_vis(*context*)
model visibility data sink

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

H

I

M

N

O

P

R

S

U

W