hypercube Documentation

Release 0.3.2

Simon Perkins

Apr 07, 2017

Contents

Contents:

CHAPTER 1

Introduction

hypercube is a package for reasoning about Radio Interferometry problem sizes.

Radio Interferometry data, such as visibilities, is both large and dense. Computation on this data is therefore generally suited to distributed, data parallel computation. To achieve this parallelism it is necessary to subdivide this data into tiles, large enough to fit within the memory budgets of individual nodes and GPUs.

hypercube is an simple abstraction that uses labelled Dimensions and abstract numpy array shapes, defined in terms of these Dimensions, to:

- Reason about memory budgets
- Choose array subdivision strategies
- Iterate over a problem space

```
from hypercube import HyperCube

cube = HyperCube()
cube.register_dimension("ntime", 10000,
    description="Timesteps")
cube.register_dimension("nbl", 2016,
    description="Baselines")
cube.register_dimension("nchan", 16384,
    description="Channels")

cube.register_array("uvw", ("ntime", "nbl", 3),
    np.float64)
cube.register_array("frequency", ("nchan",),
    np.float64)
cube.register_array("model_vis", ("ntime", "nbl", "nchan", 4),
    np.complex128)
```

```
print cube

Registered Dimensions:
Dimension Name    Description      Global Size  Extents
```

```
---------------  -------------  -------------  ----------
nbl              Baselines               2016  (0, 2016)
nchan            Channels               16384  (0, 16384)
ntime            Timesteps              10000  (0, 10000)

Registered Arrays:
Array Name          Size     Type         Shape
------------------  -------  -----------  --------------------
frequency           128.0KB  float64      (nchan)
model_vis           19.2TB   complex128   (ntime,nbl,nchan,4)
uvw                 461.4MB  float64      (ntime,nbl,3)
Local Memory Usage  19.2TB
```

CHAPTER 2

Tutorial

Dimensions

The use of python and numpy in the Radio Astronomy community naturally results in representation of data as multi-dimensional numpy arrays. hypercube, similarly to xray and pandas utilises the concept of labelling Dimensions. For example, a hypercube can be created and **Time**, **Baseline**, and **Channel** dimensions can be registered with various global sizes.

```
from hypercube import HyperCube

cube = HyperCube()
cube.register_dimension("ntime", 10000,
    description="Timesteps")
cube.register_dimension("nbl", 2016,
    description="Baselines")
cube.register_dimension("nchan", 16384,
    description="Channels")
```

Printing the cube then yields information about the registered Dimensions. Note that the **Global Size** matches the **Extents**.

```
print cube

Registered Dimensions:
Dimension Name    Description      Global Size  Extents
----------------  -------------  -------------  ----------
nbl               Baselines             2016  (0, 2016)
nchan             Channels             16384  (0, 16384)
ntime             Timesteps            10000  (0, 10000)
```

Arrays

Then we can register an abstract definition, or schema, of the **Model Visibility** array on the hypercube defined using the names of the previously registered dimensions.

```
cube.register_array("uvw", ("ntime", "nbl", 3), np.float64)
cube.register_array("frequency", ("nchan",), np.float64)
cube.register_array("model_vis", ("ntime", "nbl", "nchan", 4),
    np.complex128)
```

Printing the cube now displays additional information about the arrays and their sizes in terms of the dimension **extents**.

```
Registered Dimensions:
Dimension Name    Description      Global Size  Extents
----------------  -------------  -------------  ----------
nbl               Baselines              2016  (0, 2016)
nchan             Channels              16384  (0, 16384)
ntime             Timesteps             10000  (0, 10000)

Registered Arrays:
Array Name          Size     Type         Shape
------------------  -------  -----------  --------------------
frequency           128.0KB  float64      (nchan)
model_vis           19.2TB   complex128   (ntime,nbl,nchan,4)
uvw                 461.4MB  float64      (ntime,nbl,3)
Local Memory Usage  19.2TB
```

Modifying Dimension Extents

The problem in the previous section is too large (19.2TB) to fit within a single compute node's memory, so it is necessary to subdivide or tile the problem. The **extents** of the **Time** and **Channel** dimension are modified as follows:

```
cube.update_dimension("ntime", lower_extent=0, upper_extent=100)
cube.update_dimension("nchan", lower_extent=0, upper_extent=64)

print cube

Registered Dimensions:
Dimension Name    Description      Global Size  Extents
----------------  -------------  -------------  ---------
nbl               Baselines              2016  (0, 2016)
nchan             Channels              16384  (0, 64)
ntime             Timesteps             10000  (0, 100)

Registered Arrays:
Array Name          Size     Type         Shape
------------------  -------  -----------  --------------------
frequency           512.0B   float64      (nchan)
model_vis           787.5MB  complex128   (ntime,nbl,nchan,4)
uvw                 4.6MB    float64      (ntime,nbl,3)
Local Memory Usage  792.1MB
```

Note how the dimension extents of the **Time** and **Channel** dimensions have changed. The problem now fits within a reasonable memory budget of 792.1MB.

Querying Dimension Extents

The dimension extents can be queried on the cube:

```
cube.dim_lower_extent("ntime,nbl,nchan")
[0, 0, 0]

cube.dim_upper_extent("ntime,nbl,nchan")
[100, 2016, 64]

cube.dim_extent_size("ntime,nbl,nchan")
[100, 2016, 64]

cube.dim_extents("ntime,nbl,nchan")
[(0, 100), (0, 2016), (0, 64)]
```

Iterating over Cubes

The cube supports iteration over tiles defined by dimensions. The *hypercube.base_cube.HyperCube.extent_iter()* method produces tuples of lower extents for each dimension provided to it. Here, it produces extents for tiles of 100 Timesteps and 64 Channels.

```
for (lt, ut), (lc, uc) in cube.extent_iter(("ntime", 100), ("nchan", 64)):
    print ("lower time {} upper time {} "
            "lower channel {} upper channel{}".format(
                lt, ut, lc, uc)

lower time 0 upper time 100 lower channel 0 upper channel 64
lower time 0 upper time 100 lower channel 64 upper channel 128
lower time 0 upper time 100 lower channel 128 upper channel 192
lower time 0 upper time 100 lower channel 192 upper channel 256
lower time 0 upper time 100 lower channel 256 upper channel 320
```

Other methods of iteration include producing dictionaries defining dimension updates

```
for d in cube.dim_iter(("ntime", 100), ("nchan", 64)):
    print d
    cube.update_dimensions(d)

({'lower_extent': 0, 'upper_extent': 100, 'name': 'ntime'},
 {'lower_extent': 64, 'upper_extent': 128, 'name': 'nchan'})
```

and producing cubes defining the tile on each iteration

```
for c in cube.dim_iter(("ntime", 100), ("nchan", 64)):
    ...
```

CHAPTER 3

API

Dimension

class **Dimension**(*name*, *global_size*, *lower_extent=None*, *upper_extent=None*, *description=None*)

The Dimension class describes a hypercube dimension.

__init__(*name*, *global_size*, *lower_extent=None*, *upper_extent=None*, *description=None*)

Create a Dimension from supplied arguments

Parameters

- **name** (*str*) – Dimension name
- **global_size** (*int*) – Global dimension size
- **local_extent** (*int*) – Lower dimension extent
- **upper_extent** (*int*) – Upper dimension extent
- **description** (*str*) – Dimension description

copy()

Returns

Return type A copy of the dimension

description

Dimension description

extent_size

Size of the dimension extents. Equal to *upper_extent* - *lower_extent*

global_size

Global dimension size

lower_extent

Lower dimension extent

name
Dimension name

update(*global_size=None*, *lower_extent=None*, *upper_extent=None*, *description=None*)
Update the dimension properties

Parameters

- **global_size** (*int*) – Global dimension size (Default value = None)
- **lower_extent** (*int*) – Lower dimension extent (Default value = None)
- **upper_extent** (*int*) – Upper dimension extent (Default value = None)
- **description** (*str*) – Dimension description (Default value = None)

upper_extent
Upper dimension extent

validate()
Validate the contents of a dimension data dictionary

HyperCube

class HyperCube(**args*, ***kwargs*)
Hypercube.

__init__(**args*, ***kwargs*)
Hypercube Constructor

array(*name*, *reify=False*)

```
cube.register("uvw", ("ntime", "na", 3), np.float64)

cube.array("uvw", reify=False)["shape"] == ("ntime", "na", 3)
cube.array("uvw", reify=True)["shape"] == (100, 7, 3)
```

Parameters

- **name** (*str*) – Array name
- **reify** (*bool*) – If True, converts abstract array schemas into concrete shapes.

Returns An array dictionary

Return type dict

array_extents(*array_name*, ***kwargs*)
Return a list of lower and upper extents for the given array_name.

```
cube.register("uvw", ("ntime", "na", 3), np.float64)

(lt, ut), (la, ua), (_, _) = cube.array_extents("uvw")
```

Parameters **array_name** (*string*) – Name of an array registered on this hypercube

Returns List of (lower_extent, upper_extent) tuples associated with each array dimension

Return type list

array_slice_index(*array_name*, ***kwargs*)
Returns a tuple of slices, each slice equal to the slice(lower_extent, upper_extent, 1) of the dimensions of array_name.

```
cube.register("uvw", ("ntime", "na", 3), np.float64)

idx = cube.array_slice_index("uvw")
uvw[idx].sum()
ntime, na, components = cube.array_slice_index("uvw")
A[ntime, na, components].sum()
```

Parameters **array_name** (*str*) – Name of the array on this cube to slice

Returns Tuple of `slice(lower,upper,1)` objects

Return type tuple

arrays(*reify=False*)

Parameters **reify** (*bool*) – If True, converts abstract array schemas into concrete shapes.

Returns A dictionary of array dictionaries, keyed on array name

Return type dict

bytes_required()

Returns Estimated number of bytes required by arrays registered on the cube, taking their extents into account.

Return type int

copy()

Return type Return a copy of the hypercube

cube_iter(**dim_strides*, ***kwargs*)
Recursively iterate over the (dimension, stride) tuples specified in dim_strides, returning cloned hypercubes with each of the specified dimensions modified accordingly.

For example, the following call effectively produces 2 loops over the 'ntime' and 'nchan' dimensions in chunks of 10 and 4 respectively.:

```
A = np.ones(size=(100, 4))
for c in cube.cube_iter(('ntime', 10), ('nchan', 4))
    assert c.dim_extent_size('ntime', 'nchan') == (10, 4)
```

Parameters ***dim_strides** (*list*) – list of (dimension, stride) tuples

Returns Iterator producing hypercubes

Return type iterator

dim_extent_size(**args*, ***kwargs*)
Returns extent sizes of the dimensions in args.

```
ts, bs, cs = cube.dim_extent_size('ntime', 'nbl', 'nchan')
```

or

```
ts, bs, cs, ss = cube.dim_extent_size('ntime,nbl:nchan nsrc')
```

dim_extents(**args*, ***kwargs*)
Returns extent tuples of the dimensions in args.

```
(tl, tu), (bl, bu) = cube.dim_extents('ntime', 'nbl')
```

or

```
(tl, tu), (bl, bu) = cube.dim_upper_extent('ntime,nbl')
```

dim_global_size(**args*, ***kwargs*)
Return the global size of the dimensions in args.

```

```

ntime, nbl, nchan = cube.dim_global_size('ntime', 'nbl', 'nchan')

or

```

```

ntime, nbl, nchan, nsrc = cube.dim_global_size('ntime,nbl:nchan nsrc')

dim_global_size_dict()
Returns a mapping of dimension name to global size

dim_iter(**dim_strides*, ***kwargs*)
Recursively iterate over the (dimension, stride) tuples specified in dim_strides, returning a tuple of dictionaries describing a dimension update.

For example, the following call effectively produces 2 loops over the 'ntime' and 'nchan' dimensions in chunks of 10 and 4 respectively.

```
for d in cube.dim_iter(('ntime', 10), ('nchan', 4))
    cube.update_dimensions(d)
```

Parameters ***dim_stride** (*list*) – list of (dimension, stride) tuples

Returns Iterator produces dictionaries describing dimensions updates. `{'name':'ntime', 'lower_extent': 100, 'upper_extent': 200 }`

Return type iterator

dim_lower_extent(**args*, ***kwargs*)
Returns the lower extent of the dimensions in args.

```
t_ex, bl_ex, ch_ex = cube.dim_lower_extent('ntime', 'nbl', 'nchan')
```

or

```
t_ex, bl_ex, ch_ex, src_ex = cube.dim_lower_extent('ntime,nbl:nchan nsrc')
```

dim_lower_extent_dict()
Returns a mapping of dimension name to lower_extent

dim_upper_extent(**args*, ***kwargs*)
Returns the upper extent of the dimensions in args.

```
t_ex, bl_ex, ch_ex = cube.dim_upper_extent('ntime', 'nbl', 'nchan')
```

or

```
t_ex, bl_ex, ch_ex, src_ex = cube.dim_upper_extent('ntime,nbl:nchan nsrc')
```

dim_upper_extent_dict()
Returns a mapping of dimension name to upper_extent

dimension(*name*, *copy=True*)
Returns the requested *Dimension* object

Parameters

- **name** (*str*) – Name of the *Dimension* object
- **copy** (*boolean*) – Returns a copy of the *Dimension* object if True (Default value = True)

Returns A *Dimension* object.

Return type *Dimension*

dimensions(*copy=True*)
Return a dictionary of *Dimension* objects.

Parameters **copy** (*boolean:*) – Returns a copy of the dimension dictionary if True (Default value = True)

Returns Dictionary of *Dimension* objects.

Return type dict

endpoint_iter(**dim_strides*, ***kwargs*)
Recursively iterate over the (dimension, stride) tuples specified in dim_strides, returning the start and end indices for each chunk.

For example, the following call effectively produces 2 loops over the 'ntime' and 'nchan' dimensions in chunks of 10 and 4 respectively.

```
for (ts, te), (cs, ce) in cube.endpoint_iter(('ntime', 10), ('nchan', 4))
    print 'Time range [{ts},{te}] Channel Range [{cs},{ce}]'.format(
        ts=ts, te=te, cs=cs, ce=ce)
```

Parameters ***dim_strides** (*list*) – list of (dimension, stride) tuples

Returns An iterator producing lists of lower and upper extent tuples `[(d0_low, d0_high), (d1_low, d1_high), ..., (dn_low, dn_high)]`

Return type iterator

extent_iter(**dim_strides*, ***kwargs*)
Recursively iterate over the (dimension, stride) tuples specified in dim_strides, returning the start and end indices for each chunk.

For example, the following call effectively produces 2 loops over the 'ntime' and 'nchan' dimensions in chunks of 10 and 4 respectively.

```
for (ts, te), (cs, ce) in cube.endpoint_iter(('ntime', 10), ('nchan', 4))
    print 'Time range [{ts},{te}] Channel Range [{cs},{ce}]'.format(
        ts=ts, te=te, cs=cs, ce=ce)
```

Parameters ***dim_strides** (*list*) – list of (dimension, stride) tuples

Returns An iterator producing lists of lower and upper extent tuples `[(d0_low, d0_high), (d1_low, d1_high), ..., (dn_low, dn_high)]`

Return type iterator

mem_required()

str String approximately describing the memory required by arrays registered on the cube, taking their extents into account.

properties()
Returns a dictionary of properties

property(*name*)

Parameters **name** (*str*) – Property to return

Returns A property dictionary

Return type dict

register_array(*name*, *shape*, *dtype*, ***kwargs*)
Register an array with this cube.

```
cube.register_array("model_vis", ("ntime", "nbl", "nchan", 4), np.complex128)
```

Parameters

- **name** (*str*) – Array name
- **shape** (*A tuple containing either Dimension names or ints*) – Array shape schema
- **dtype** – Array data type

register_arrays(*arrays*)
Register arrays using a list of dictionaries defining the arrays.

The list should itself contain dictionaries. i.e.

```
D = [{ 'name':'uvw', 'shape':(3,'ntime','nbl'),'dtype':np.float32 },
    { 'name':'lm', 'shape':(2,'nsrc'),'dtype':np.float32 }]
```

Parameters **arrays** (*A list or dict.*) – A list or dictionary of dictionaries describing arrays.

register_dimension(*name*, *dim_data*, ***kwargs*)
Registers a dimension on this cube.

```
cube.register_dimension('ntime', 10000,
            decription="Number of Timesteps",
            lower_extent=100, upper_extent=200)
```

Parameters

- **dim_data** (int or *Dimension*) – if an integer, this will be used to define the global_size of the dimension and possibly other attributes if they are not present in kwargs. If a Dimension, it will be updated with any appropriate keyword arguments
- **description** (*str*) – The description for this dimension. e.g. 'Number of timesteps'.

- **lower_extent** (*int*) – The lower extent of this dimension within the global space
- **upper_extent** (*int*) – The upper extent of this dimension within the global space
- **name** (*Dimension name*) –

Returns A hypercube Dimension

Return type *Dimension*

register_dimensions(*dims*)

Register multiple dimensions on the cube.

```
cube.register_dimensions([
    {'name' : 'ntime', 'global_size' : 10,
    'lower_extent' : 2, 'upper_extent' : 7 },
    {'name' : 'na', 'global_size' : 3,
    'lower_extent' : 2, 'upper_extent' : 7 },
])
```

Parameters **dims** (*list or dict*) – A list or dictionary of dimensions

register_properties(*properties*)

Register properties using a list defining the properties.

The dictionary should itself contain dictionaries. i.e.

```
D = [
    { 'name':'ref_wave','dtype':np.float32,
        'default':1.41e6 },
]
```

Parameters **properties** (*A dictionary or list*) – A dictionary or list of dictionaries describing properties

register_property(*name*, *dtype*, *default*, ***kwargs*)

Registers a property with this Solver object

```
cube.register_property("reference_frequency", np.float64, 1.4e9)
```

Parameters

- **name** (*str*) – The name of this property.
- **dtype** (*Numpy data type*) – Numpy data type
- **default** (*Should be convertable to dtype*) – Default value for this value

slice_index(**slice_dims*, ***kwargs*)

Returns a tuple of slices, each slice equal to the slice(lower_extent, upper_extent, 1) of the dimensions supplied in slice_dims. If the dimension is integer d, slice(0, d, 1) will be used instead of the lower and upper extents

```
A = np.ones(ntime, na)
idx = cube.slice_index('ntime','na', 3)
A[idx].sum()
ntime, na, components = cube.slice_index('ntime', 'na', 3)
A[ntime, na, components].sum()
```

Parameters ***slice_dims** (*tuple*) – Tuple of dimensions for which slice objects should be returned.

Returns Tuple of `slice(lower,upper,1)` objects

Return type tuple

slice_iter(**dim_strides*, ***kwargs*)

Recursively iterate over the (dimension, stride) tuples specified in dim_strides, returning the chunk start offsets for each specified dimensions.

For example, the following call effectively produces 2 loops over the 'ntime' and 'nchan' dimensions in chunks of 10 and 4 respectively.

```
A = np.ones(size=(100, 4))
for ts, cs in cube.endpoint_iter(('ntime', 10), ('nchan', 4))
    A[ts, cs].sum()

for i cube.endpoint_iter(('ntime', 10), ('nchan', 4))
    A[i].sum()
```

Parameters ***dim_strides** (*list*) – list of (dimension, stride) tuples

Returns Iterator producing a tuple of slices for each dimension `(slice(d0_low, d0_high, 1), slice(d1_low, d1_high,1))`

Return type iterator

update_dimension(*name*, ***update_dict*)

Update the dimension size and extents.

Parameters

- ****update_dict** (*: dict: dict*) – A dictionary containing keywords passed through to *update()*
- **name** (*str*) – Name of the dimension to update

update_dimensions(*dims*)

Update multiple dimension on the cube.

```
cube.update_dimensions([
    {'name' : 'ntime', 'global_size' : 10,
    'lower_extent' : 2, 'upper_extent' : 7 },
    {'name' : 'na', 'global_size' : 3,
    'lower_extent' : 2, 'upper_extent' : 7 },
])
```

Parameters **dims** (*list or dict:*) – A list or dictionary of dimension updates

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

V