LOFAR synthesis data handling
pyrap

Ger van Diepen
ASTRON

pyrap

Python interface on top of some casacore functionality
Use with numpy, matplotlib, scipy, etc.

- libpyrap.so contains converters between C++ (casacore) and python
- pyrap packages:
  - `pyrap.tables`
  - `pyrap.quanta`
  - `pyrap.measures`
  - `pyrap.images` and `pyrap.images.coordinates`
  - `pyrap.functionals` and `pyrap.fitting`

See http://www.astron.nl/casacore/trunk/pyrap/docs/index.html

pyrap vs. casapy

- casapy is primarily a package to do calibration and imaging
  - Python layer on top of CASA
- pyrap is a toolkit
  - Python layer on top of casacore
  - More pythonic than casapy
  - Richer functionality
- Important difference:
  - casapy uses arrays with axes in Fortran order
  - pyrap uses arrays with axes in C order (reverses casacore axes)
    - natural numpy order is C order (first axis varies slowest)

Python to casacore

- Type conversion done if needed

- Numeric/Bool C++ scalar
  - Python or numpy scalar
- casa::String or std::string
  - Python string
- casa::Vector<T> or std::vector<T>
  - Any python sequence (e.g., list, tuple, numpy vector)
    - Can contain mixed types; 'highest' type is used
  - Scalar (results in vector of length 1)
- casa::IPosition
  - Any numeric python sequence
    - Values are reversed
  - Numeric scalar
- casa::Array<T>
  - Numpy array (also slice, empty array, and scalar array)
    - Axes are reversed
  - Any python sequence (results in 1D Array)
  - Scalar
- casa::Record
  - Python dict

Casacore to python

- Python types can be converted, numpy types are not

- Numeric/Bool C++ scalar
  - Python scalar
- casa::String or std::string
  - Python string
- std::vector<T>
  - Python list
- casa::IPosition
  - Python list
    - Values are reversed
- casa::Array<T> (also casa::Vector<T>)
  - numpy array
    - Axes are reversed
  - casa::Array<String> as dict of list of strings and shape
- casa::Record
  - Python dict

pyrap.tables

- Built on top of class TableProxy
  - Open or create a table or open table concatenation
  - Get and put column data or data slices
  - Get, put, and remove header keywords
  - Add, rename, and remove columns
  - Selection, sorting
  - Iteration
  - Get miscellaneous info
  - Can use python indexing (also negative increment)
- Derived from old Glish interface
  - All function and argument names are in lowercase

pyrap.tables classes

- table
  - Open, create, select, sort, column and keyword access
- tablecolumn
  - Easy access to data in a column
- tablerow
  - Easy access to data in a row
- tableiter
  - Iteration over a table
- tableindex
  - Temporary index of a table on one or more columns
    Useful if having to lookup many values in a table column

- Global functions
  - Create column and table description
  - TaQL command
  - Create a table from an ASCII file

Get data from a table

```
from pyrap.tables import *

t = table('mytable')
t.getcol('ANTENNA1')                    # get all values of ANTENNA1 column
t[0]                                    # get contents of first row
t[-1]                                   # get last row

tc = t.col("ANTENNA1")                  # tablecolumn object
tc[0]                                   3 get value of first row of ANTENNA1
tc[:]                                   # get entire column
tc[::-1]                                # get in reversed order

t.getkeywords()  # t.keys or t._        # all table keywords (as a dict)
tc.getkeywords()                        # all keywords of column ANTENNA1
t.getcolkeywords('TIME')                # all keywords of column TIME
   {'MEASINFO': {'Ref': 'UTC', 'type': 'epoch'}, 'QuantumUnits': ['s']}

        NEW: can use attribute names for column or keyword
t.ANTENNA1[0]                          # attribute name is same as t.col(name)[0]
t.DATA[0][:,0]                         # get XX data (but reads entire row!)
t.keys (or t._)                        # get table keywords
```

Put data into a table

```
import pyrap.tables as pt

t = pt.table('mytable', readonly=False)
t.putcol ('ANTENNA1', t.getcol('ANTENNA1')+1)

tc = t.col("ANTENNA1")                    # tablecolumn object
tc[0] = 1

keys = t.getcolkeywords('TIME')
keys
    {'MEASINFO': {'Ref': 'UTC', 'type': 'epoch'}, 'QuantumUnits': ['s']}

# Set new Measure Ref
keys[' MEASINFO']['Ref'] = 'UT1'
t.putcolkeywords ('TIME', keys)

# easier!!
t.putcolkeyword ('TIME', 'MEASINFO.Ref', 'UT1')
tc.putkeyword ('MEASINFO.Ref', 'UT1')

# Always flush
# ipython keeps objects alive, so table destructor is not called
t.flush()
```

Using attributes and indexing

pyrap.tables implements __getattr__, __getitem__ and __setitem__

__getattr__	gets tablecolumn or keyword (opens if keyword is subtable)
__getitem__	gets row(s)
__setitem__	puts row(s)

Can be slightly more expensive, so preferably only use interactively.

```
t[0]                # get first row of all columns
t.ANTENNA1[0]       # get first row of column ANTENNA1
t.col('ANTENNA1')[0]

t.DATA[-1][:,0]     # XX data in last row (last [] is numpy!)

t.FEED.TIME[:]      # all rows of column TIME in subtable FEED
```

Sorting

```
t = pt.table('~/GER1.MS')

# Get all unique times (max 18 and in descending order)
t1 = t.sort ('unique desc TIME', limit=18)
t1.getcol('TIME') - t1.getcell('TIME',0)
   array([   0.,  -10.,  -20.,  -30.,  -40.,  -50.,  -60.,  -70.,  -80.,
          -90., -100., -110., -120., -130., -140., -150., -160., -170.])

# Get all unique baselines (ANTENNA1 is ascending, ANTENNA2 is descending)
# Note: give all sort keys in a single string, not as e.g. a list
t2 = t.sort ('unique ANTENNA1, ANTENNA2 desc')

# Sort and make the result persistent (as a RefTable!)
t3 = t.sort ('TIME, FIELD_ID, ANTENNA1, ANTENNA2', name='GER1_sorted.MS')
```

Selection (using TaQL)

```
t = pt.table('~/GER1.MS')

# select all baselines using station 1
t1 = t.query ('ANTENNA1 = 1 || ANTENNA2 = 1')

# select all baselines where first station is 1, 3, or 8
t1 = t.query ('ANTENNA1 in [1,3,8]')

# select all baselines with core stations ('advanced' TaQL)
# But see msselect (discussed later)
t1 = t.query ('any([ANTENNA1,ANTENNA2] in [select rowid() from ::ANTENNA where NAME ~ p/
    CS*/])')

# select the first N rows.
t1 = t.query (limit = N)

# select given rows (e.g., same rows as in another table selection)
t1 = t.selectrows (othertable.rownumbers())

# select a few columns (called projection in data base terms)
t1 = t.select ('ANTENNA1, ANTENNA2')
t1.colnames()
    ['ANTENNA1', 'ANTENNA2']
```

Copying a table (selection)

Make a copy of a selection of an MS.
Usually this will be done as a RefTable (which is fast), but it can be done as a proper PlainTable which takes (much) more time.

```
t = table('my.ms')
t1 = t.query('ANTENNA1 != ANTENNA2')
t2 = t1.copy ('crossref.ms')                    # shallow copy (as RefTable)
t3 = t1.copy ('crosspln.ms', deep=True)         # deep copy (as PlainTable)
```

Sometimes it can be useful to turn the LofarStMan storage manager into a standard one.
This can be done by making a value copy.

```
t = table('raw.ms')
t1 = t.copy ('normal.ms', valuecopy=True)
```

Advanced users can use the *dminfo* argument to define new data managers.

Function *copyrows* makes it possible to do more selective copying of table rows.

Table iteration

```
t = table('~/GER1.MS')

# Print nr of rows in each time slot
# No sorting is done (table is known to be in TIME order)
for t1 in t.iter('TIME', sort=False):
   print t1.nrows()

# Print nr of rows in each baseline
for t1 in t.iter(['ANTENNA1', 'ANTENNA2']):
   print t1.nrows()
```

Note that getting bulk data can be slow when iterating in non physical order.

Obtaining table info

```
t.summary()                          # Print summary of table (columns, keywords, subtables)

t.getdminfo()                        # Get data manager info (as a dict)
t.getdmprop('TIME')                  # Get properties of a column's data manager

print t.showstructure()              # Print the table structure and optionally of subtables
                                     # Shows which file (e.g., table.f0) belongs to a storage manager

t.nrows()                            # Get nr of rows
t.ncols()                            # Get nr of columns
t.colnames()                         # Get the column names

t.info()                             # Get the ASCII table info

t.getdesc()                          # Get the table description
t.getcoldesc('TIME')                 # Get the column description

t.partnames()                        # Get parent table name(s)

t.endianformat()                     # Tells if data are stored in little or big endian format

t.fieldnames()                       # Get names of table keywords

t.subtables()                        # Get names of all subtables

t.browse()                           # Start casabrowser
t.view()                             # Start casaviewer (for image and MS) or casabrowser
```

Opening a subtable

Similar to casacore: do NOT open a subtable by name because it'll fail for RefTable's.

In general scripts use:

```
tab = pt.table('main.tab')
subtab = pt.table(tab.getkeyword('ANTENNA'))
```

or

```
tab = pt.table('main.tab')
subtab = tab.ANTENNA
```

or easier:

```
subtab = pt.table('main.tab::ANTENNA')
```

Use

```
subtab = pt.table('main.tab/ANTENNA')
```

only interactively!!

Concatenate tables

Concatenate 3 time slices of a WSRT MS; also concatenate its SYSCAL subtable.

```
t = table(['wsrt.ms1', 'wsrt.ms2', 'wsrt.ms3'], concatsubtables='SYSCAL')
t1 = t.sort ('unique TIME')
syscaltab = table(t.getkeyword('SYSCAL'))
```

Concatenating LOFAR subbands requires more effort to get correct indices for the SPECTRAL_WINDOW subtable

Each MS uses DATA_DESC_ID=0, but in the concatenation it should be 0,1,2...

It can be done by function msconcat, which uses data manager ForwardColumn to reference most of the columns, but creates a proper column for the DATA_DESC_ID.

```
msconcat (['lofar.ms1', 'lofar.ms2', 'lofar.ms3'], 'new.ms')
t = table('new.ms')
```

Advanced pyrap

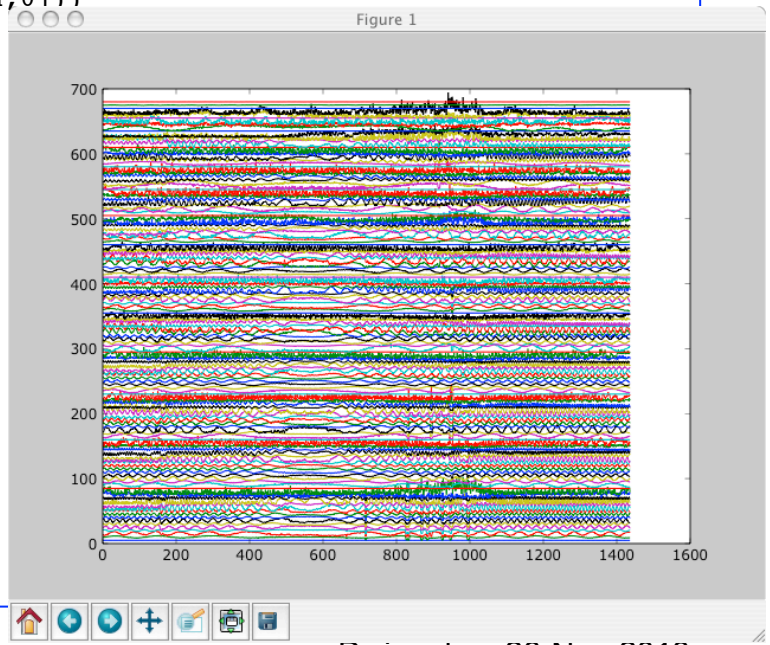
It uses the dminfo argument to tell a newly created table how to store columns.

```
tnew = pt.table(newname, intab.getdesc(), nrow=intab.nrows(), dminfo={'1':
   {'TYPE':'ForwardColumnEngine', 'NAME':'ForwardData',
   'COLUMNS':tn.colnames(), 'SPEC':{'FORWARDTABLE':intab.name()}}})
```

plot baselines

```
import numpy
import pylab
import pyrap.tables as pt

def plotbl (ms, band=0, ch=0, sep=5.0, fig=None):
    t  = pt.table(ms);
    t1 = t.query ('DATA_DESC_ID=%d' % band)
    pylab.figure(fig)
    pylab.clf()
    offset = 0.0
    for t2 in t1.iter(["ANTENNA1","ANTENNA2"]):
        # Get XX data of given channel
        ampl = numpy.absolute (t2.getcolslice("DATA", [ch,0], [ch,0]))
        sc = sep/numpy.mean(ampl)
        ampl = ampl.reshape(ampl.shape[0])
        pylab.plot(sc*ampl + offset)
        offset += sep
    pylab.show()
```



High level MS functions

msconcat

- concatenates multiple tables in a (mostly) virtual way.
- Tables split in time (like WSRT) are concatenated directly.
- Tables split in subband (like LOFAR) need to update the spectral window indices which is done in a cheap way.
- Look at this function to learn how data manager manipulation can be done :-)

msregularize

- creates a regular MS from an input MS.
- It means that the same number of baselines are used for each time slot.
- It is useful for an WSRT MS that has to used in BBS.

addImagingInfo

- adds the columns needed by CASA or lwimager (only if not existing).
- It is useful to prevent such tasks from removing the CORRECTED_DATA column.

addDerivedMSCal

- adds virtual columns for hourangle, AzEl, parallactic angle, UVW in J2000.
- They can be useful for plotting.
- But see similar TaQL functions later

pyrap.measures

Convert measures to other frames

- mainly used for celestial coordinates

- Bas van der Tol has used it quite extensively
- Examples courtesy of Maaijke Mevius

- TaQL has functions to do measure conversions

Convert Ra,Dec to Az,El

```
import numpy as np;
from pyrap.measures import measures;
import pyrap.quanta as qa;

def radec2azel (ra, dec, time, itrf):
    me=measures();
    phasedir=me.direction('J2000', ra, dec)
    t=me.epoch("UTC", qa.quantity(time));
    me.do_frame(t);

    p = me.position('ITRF',str(itrf[0])+'m',str(itrf[1])+'m',str(itrf[2])+'m')
    me.do_frame(p);

    azel = me.measure(phasedir, 'azel');
    return azel;
```

Use as:

```
radec2azel ('12h34m56', '50d34m23.87', 'today', [3e6,3e6,3e6])
```

```
    {'m0': {'unit': 'rad', 'value': 0.37755678376678325},
     'm1': {'unit': 'rad', 'value': 0.34341534300523713},
     'refer': 'AZEL',
     'type': 'direction'}
```

Get position of the moon

```
from pyrap.measures import measures;
import pyrap.quanta as qa;

def getMoonPos(time):
    me = measures();
    moondir = me.direction("MOON");
    t = me.epoch("UTC", qa.quantity(time));
    me.do_frame(t);
    mydir = me.measure(moondir, "J2000");
    ra    = mydir['m0']['value'];
    dec   = mydir['m1']['value'];
    return (ra,dec);
```

Use as:

```
getMoonPos ('12Jul2011/14:56:31')
```

(-1.8256634076120826, -0.407712079287842)

pyrap.images

- Built on top of class ImageProxy
  - Open image, image expression, or concatenation
    - casacore, HDF5, FITS and/or Miriad format
  - Create image given coordinates and shape
  - Get and put data (slices) as numpy array
  - Get and put mask (slices) as numpy mask
    - meaning of True is opposite, so mask is negated
  - Create subimage
    - Only box regions, no support for other casacore regions yet (polygon, ellipsoid, union, intersection, difference, extend)
  - Save in FITS, HDF5, or casacore format
  - Get statistics
  - Get coordinates
  - Get attributes (in extra tables like LOFAR_ANTENNA))
  - Convert pixel to world coordinates and vice-versa
  - Regrid
  - Start casaviewer (makes virtual image persistent)

Creating an image object

```
import pyrap.images as pim

# Open a casacore image
im = pim.image('my.img')

# Open a FITS image
im = pim.image('my.fits')

# Open a virtually concatenated set of images (of any type)
im = pim.image(['my.img1', 'my.img2', 'my.fits3'])

# Open an image expression
im = pim.image('(my.img1 +  my.img2 + my.fits3) / 3')

# Create a new float32 image (using the coordinates of another one)
im = pim.image('new.img', coordsys=other.coordinates(),
                shape=other.shape())

# Create a subimage of first channel
imsub = im.subimage(0, 0)
```

get/put image data

```
im = pim.image('my.img')

# Get data or mask
# Optionally a blc and trc can be given
data = im.getdata()                  # numpy.array
mask = im.getmask()                  # numpy.array
maskedarray = im.get()               # numpy.(core.)ma.MaskedArray

# Get statistics (mean, min, max, median, ...)
# Also possible per plane, etc.
im.statistics()

# Put data or mask.
# If needed (and possible), the image is reopened for read/write.
im.putdata (data)
im.putmask (mask)
im.put (data)                        # only put data if numpy.array
im.put (maskeddata)                  # put data and mask if MaskedArray

im.attrgroupnames()                  # get names of attribute groups
                                       (extra subtables like LOFAR_STATION)
```

pyrap.images.coordinates

```
im = pim.image('my.img')
crd = im.coordinates()

# Print info
crd.summary()

# Get the various coordinates
directionCoord = crd['direction']
spectralCoord = crd['spectral']
stokesCoord = crd['stokes']

# Get reference value, pixel, and delta.
directionCoord.get_referencevalue()
directionCoord.get_referencepixel()
directionCoord.get_increment()
```