LOFAR synthesis data handling
casacore

Ger van Diepen

ASTRON

casacore

- Written before STL existed (continuation of AIPS++)
- Working on thread-safety of statics (casa, tables, scimath, measures done on a branch)
- namespace casa
- See http://www.astron.nl/casacore/trunk/casacore/doc/html/
  and http://www.astron.nl/casacore/trunk/casacore/doc/notes/notes.html
- Used by WSRT, CASA, LOFAR, ASKAP, MeqTrees

- casa
  - Utilities (sort, median)
  - Containers (Record, ValueHolder)
  - Array, Vector, Matrix, Cube
  - Quanta (value with unit)
- tables
  - Creation, read/write, iteration, selection, sort
- measures
  - Quanta with reference frame
- ms/MeasurementSets
- images and coordinates
- scimath
  - Functionals and Fitting (linear and non-linear)
  - Mathematics (FFT, DFT, interpolation)

Array

N-dimensional array templated on data type

- Unlike Blitz and boost::multi_array dimensionality is no template parameter
- Data are in Fortran order (first axis varies fastest)

- Vector, Matrix, and Cube are 1,2,3D specializations
  - casa::Vector is very different from std::vector (which is similar to casa::Block)
- Class IPosition is used for shape and location

```
#include <casa/Arrays/Array.h>
// define the 3-dim shape
IPosition shape(3, npol, nchan, nbaseline);
// create the array
Array<double> arr(shape);
// set entire array to 0
arr = 0.;
// set first element to 1
arr(IPosition(3,0,0,0)) = 1.;
```

Array reference and copy

- Copy constructor make a shallow copy (i.e., references same data)
- Assignment makes a deep copy
  - requires lvalue to be empty or same shape as rvalue
  - `assign` function resizes if needed

```
Array<T> arr1(IPosition(2, 10, 20));
Array<T> arr2 = arr1;         // copy ctor (shallow copy)
   // arr2 and arr1 point to the same data!

Array<T> arr2;
arr2 = arr1;                  // assignment (deep copy)
  // arr2 and arr1 point to different data
```

- Other similar functions:

`copy`	makes a deep copy
`reference`	references another array
`unique`	makes object unique (copies if multiple references)

Array slicing

- A view on a subset of an Array
  - references the same data
  - is a normal Array object (with possibly non-contiguous data)

```
Array<int> arr (IPosition(2, 100, 100));
arr = 0;
// Set inner 60*60 part to 1
//   Use start,end
arr(IPosition(2,20,20), IPosition(2,79,79)) = 1;
//   Use start,length
arr(Slicer(IPosition(2,20,20), IPosition(2,60,60))) = 1;
//   Keep slice as an Array object
Array<int> subarr(arr(Slicer(IPosition(2,20,20), IPosition(2,60,60))));
subarr = 1;
```

Array iteration

- STL-style iteration

```
Array<T>::const_iterator endIter = arr.end();
for (Array<T>::const_iterator iter=arr.begin();
          iter!=endIter; ++iter) {
    *iter = T();
}
```

Important:

- create endIter before the loop (expensive to do it in the loop)
- use prefix increment (iter++ is more expensive)

For contiguous arrays using *cbegin* and *cend* is faster.

```
Array<T>::const_contiter endIter = arr.cend();
for (Array<T>::const_contiter iter=arr.cbegin();
          iter!=endIter; ++iter) {
    *iter = T();
}
```

- casa/Arrays/ArrayIter.h

  iterate in chunks of lower dimensionality (e.g., over all planes in a cube)

Full Array operations

Mathematical	+ - * / sin exp max .....
Logical	== > || .....
IO	<<

- operate element-wise (thus Matrix*Matrix is **not** matrix product)
- array-array, array-scalar, and scalar-array versions
- shapes must match (no shape broadcasting like numpy)

```
#include <casa/Arrays/ArrayMath.h>
Array<T> result = arr1 + arr2 * sin(arr3);
Array<T> result = max(arr1, arr2);

#include <casa/Arrays/ArrayLogical.h>
Array<Bool> result(arr == 0);

#include <casa/Arrays/ArrayIO.h>
cout << arr;
```

N.B.

A Matrix object is printed transposed (as a proper matrix)

Other Array functionality

- MaskedArray
- Reduction functions (median, min, max, sum, ntrue, ...)

```
#include <casa/Arrays/ArrayMath.h>
double minValue = min(arr);
```

- casa/Arrays/ArrayPartMath.h
  - Partial array reduction operations
    - median, etc. on one or a few axes
  - Sliding array operations
    - E.g., running median
  - Boxed array operations
    - E.g., downsampling by averaging every m*n subset
- casa/Arrays/MatrixMath.h
  - Linear algebra
    - Matrix multiplication
    - In-product, etc.

Quanta

- Value with unit
  - SI and derived units
  - Prefixes like k, m, etc.   (kpc, mm)
  - Composite units   (m/s)
  - Units divided into in groups (time, length, etc.)
  - Can convert within a group

```
#include <casa/Quanta/Quantum.h>
Quantity q(10, “m”);
// Convert from m to mm
cout << q.getValue (”mm”) << endl;
```

Used in MeasurementSets and TaQL

Often used casa functionality

- casa/BasicSL/Complex.h
  - Complex, Dcomplex   typedefs for std::complex<float>, <double>
- casa/BasicSL/String.h
  - String   std::string with some extra functions
- casa/Utilities/GenSort.h
  - sort
    - also indexed (not available in STL)
  - kthLargest (for median or fractile)
- casa/Containers/Record.h
  - dict-like

Table access

- TableProxy (not discussed)
  - Simple access (slightly more overhead)
  - Type-independent column access (uses ValueHolder)
  - Similar functions as in pyrap.tables (which uses TableProxy)

- Native classes

– Table	opening/creating a table
– TableDesc	table description
– TableIter	table iteration
– ScalarColumn, ArrayColumn	access to data in a column

  - must match column’s data type
  - used by MeasurementSet classes

- Makes heavily use of casa classes

  Array, Complex, String, Record

Getting data from a table

```
#include <tables/Tables/Table.h>
#include <tables/Tables/ScalarColumn.h>
#include <tables/Tables/ArrayColumn.h>
#include <casa/Arrays/Vector.h>
#include <casa/Arrays/Cube.h>

// Open main table of an MS (as readonly)
Table tab(“~/GER.MS”);
// Create readonly accessor object for column TIME
ROScalarColumn<Double> timeCol(tab, “TIME”);
// Get the time from the first row.
Double time0 = timeCol(0);
// Get the times for the entire column.
Vector<Double> allTimes = timeCol.getColumn();
// Get the data.
ROArrayColumn<Complex> dataCol(tab, “DATA”);
Array<Complex> data(dataCol.getColumn());
// Get XX data by getting a slice from each array in the column,
Cube<Complex> xxData(dataCol.getColumn (Slicer(IPosition(2,0,0),
                                               IPosition(2,1,Slicer::MimicSource)));
```

Note:

- the last operations are not full-proof; it can fail for large MSs
- for getColumn the arrays must have the same shape in all rows
- Slicer::MimicSource means till the end of that axis
- In this example getColumn returns a 3-dim Array, so it can be turned into a Cube object

Putting data into a table

```
#include <tables/Tables/Table.h>
#include <tables/Tables/ScalarColumn.h>
#include <tables/Tables/ArrayColumn.h>
#include <casa/Arrays/ArrayMath.h>       // for array addition

// Open the main table of an MS (as read/write)
Table tab(“~/GER.MS”, Table::Update);
// Create read/write accessor object for column TIME
ScalarColumn<Double> timeCol(tab, “TIME”);
// Update the times in the entire column (were 10 seconds off).
timeCol.putColumn (timeCol.getColumn() + 10.);
// Flush the table
tab.flush();
```

Note:

- The Table system uses internal buffers that are written to disk if needed
- Function flush writes all modified internal buffers
- If not flushed explicitly, it will be done by the Table destructor (unless called from an exception)

Table sort

```
#include <tables/Tables/Table.h>
#include <casa/Containers/Block.h>

Table tab(“~/GER.MS”);

// Sort on a single key
Table tabSort1 = tab.sort (“TIME”);

// Sort on multiple keys
Block<String> sortKeys(2);
sortKeys[0] = “ANTENNA1”;
sortKeys[1] = “ANTENNA2”;
Table tabSort2 = tabSort1.sort(sortKeys);
```

Note:

- The resulting Table object is internally a RefTable, thus it references the rows in the original table.
    - Table::deepCopy can be used to make a copy of such a table.
- The sort of a table is stable, thus it preserves the original order of equal keys.
- Accessing bulk data in a sorted table can be much slower (may require many disk seeks).

Table selection

- Using overloaded C++ functions on class TableExprNode

```
#include <tables/Tables/Table.h>
#include <tables/Tables/ExprNode.h>

Table tab("~/GER.MS");
// Select a specific baseline
Table tabSel = tab(tab.col("ANTENNA1") == 0 && tab.col("ANTENNA2") == 1);
```

- Using TaQL

```
#include <tables/Tables/TableParse.h>

Table seltab = tableCommand ("select from ~/GER.MS where ANTENNA1==0 &&
   ANTENNA2==1")
```

Note:

- Again the internal result is a RefTable.
- Both ways result in the same expression tree, thus are equally fast when evaluated.

Table iteration

Sometimes it is useful to step through a table (e.g., per baseline)

```
#include <tables/Tables/Table.h>
#include <tables/Tables/TableIter.h>

Table tab(“~/GER.MS”);
// Define the columns to iterate on.
Block<String> iterKeys(2);
iterKeys[0] = “ANTENNA1”;
iterKeys[1] = “ANTENNA2”;
TableIterator tabIter(tab, iterKeys);
// Iterate until no more baselines.
while (! tabIter.pastEnd()) {
    Table chunk = tabIter.table();
    tabIter.next();
}
```

Note:

- Again each iteration step results internally in a RefTable.
- By default TableIterator will sort on the iteration keys, but that can be bypassed.
- Accessing a table using TableIterator can be slow if the iteration mismatches the physical order.

Opening a subtable

People are tended to open a subtable like:

```
   Table antTab (“~/GER.MS/ANTENNA”);
```

Never do it that way, but use

```
   Table mainTab (“~/GER.MS”);
   Table antTab (mainTab.keywordSet().asTable(“ANTENNA”);
```

or

```
   Table antTab(Table::openTable(“~/GER.MS::ANTENNA”));
```

The very first version will only work if the table is a PlainTable,
but not for a persistent RefTable (or ConcatTable).
The second versions always work.

Miscellaneous Table func

- Test if a table exists

```
Table::isReadable (tableName)
```

- Test if a column exists in a table

```
tab.tableDesc().isColumn (columnName);
```

- Make a copy of a table and its subtables
  - Using a simple file copy (shallow copy)
    - Keeps table type and storage managers

```
tab.copy (newName, Table::New);
```

  - Making a true copy (deep copy)
    - turns a RefTable into a PlainTable
    - turns LofarStMan into a standard storage manager

```
tab.deepCopy (newName, Table::New);
```

  - Class TableCopy offers more selective copying

Measures

A Measure consists of one or more values (in an MVxxx object) and a reference type and frame

MDirection	direction in sky
MPosition	position on earth
MEpoch	epoch

MBaseline
Muvw
MFrequency
MDoppler
MRadialVelocity
MEarthMagnetic

Their MV counterparts (in casa/Quanta) contain the values as Quantity

Measure reference frame

- A Measure object contains its reference type and frame

  E.g., MDirection::J2000   does not need a frame

  MDirection::App   needs a frame

- Class MeasFrame contains frame information
  - Epoch
  - Position
  - Phase center direction
  - Radial velocity
  - MeasComet (for non-planet solar system objects)

- Some frame information can be required when converting

  E.g., for J2000 to apparent (needs epoch and position)

Measure example

```
#include <measures/Measures/MDirection.h>
#include <measures/Measures/MCDirection.h>
#include <measures/Measures/MeasConvert.h>

// Create an J2000 direction for an RA and DEC in radians.
Quantity ra(1., “rad”);
Quantity dec(1.5, “rad”);
MVDirection mvdir(ra, dec);
MDirection dir(mvdir, MDirection::J2000);

// Convert it to B1950 (requires no frame information).
// Note that operator () does the conversion.
MDirection dirB1950 = MDirection::Convert(dir, MDirection::B1950)();
```

http://www.astron.nl/casacore/trunk/casacore/doc/html/group__Measures__module.html#_details
contains a detailed description and more examples.

TableMeasures

- Stores Measures in a Table
- Only fixed reference types are permitted (like J2000)
- Units and reftype are stored in the column's keywords
- Used by MeasurementSet classes

- Access using:
  - measures/TableMeasures/ScalarMeasColumn.h
    for a column holding one measure per row
  - measures/TableMeasures/ArrayMeasColumn.h
    for a column holding an array of measures per row

  - ScalarQuantColumn.h and ArrayQuantColumn.h
    for access using Quantity only (only unit support)

MeasurementSet

- A class for the main table and each subtable
  - Derived from Table, so all Table functionality can be used

    E.g., MeasurementSet.h
    MSAntenna.h
    MSField.h

- A class for the columns in main table and each subtable

  E.g., MSMainColumns.h
  MSAntennaColumns.h
  MSFieldColumns.h

  Has functions to access each column using

  ScalarColumn<T> or ArrayColumn<T> object

  If a column holds a Measure, also access using

  ScalarMeasColumn<T> or ArrayMeasColumn<T> object

MSLofar

- Extension of MeasurementSet with LOFAR specific columns and subtables
- Follows LOFAR MS ICD

- Specific classes (work exactly like MS counterparts)

– MSLofar	for MeasurementSet
– MSLofarAntenna	for MSAntenna
– MSLofarObservation	for MSObservation
– MSLofarField	for MSField
– MSStation	for LOFAR_STATION
– MSAntennaField	for LOFAR_ANTENNA_FIELD
– MSElementFailure	for LOFAR_ELEMENT_FAILURE

Accessing an existing MS

```
#include <ms/MeasurementSets/MeasurementSet.h>
#include <ms/MeasurementSets/MSMainColumns.h>
#include <ms/MeasurementSets/MSAntennaColumns.h>
#include <vector>

// Open the MS readonly.
MeasurementSet ms(“~/GER>MS”);
// Get readonly access to its main columns.
ROMSMainColumn msCols(ms);
// Get readonly access to the columns in the ANTENNA subtable.
ROMSAntennaColumns antCols(ms.antenna());

// Get row 10 of the DATA column.
Matrix<Complex> data10 (msCols.data()(10));

// Get the positions of all antennae (stations) and convert to ITRF.
std::vector<MPosition> positions;
positions.reserve (msCols.nrow());       // good practice to reserve if possible
ROScalarMeasColumn<MPosition>& posCol = antCols.positionMeas();
for (uInt i=0; i<msCols.nrow(); ++i) {
    positions.push_back (posCol.convert( i, MPosition::ITRF));
}
```

Creating an MS

Steps to take:

- define the descriptions
  - Required columns
  - Optional columns
  - Possible site-specific columns (like LOFAR_*)
- Attach columns to table data managers
- Create the MS and its subtables
  - Required subtables
  - Optional subtables
  - Possible site-specific subtables

See MSCreate.cc in LOFAR/CEP/MS/src

Creating an MS (simple)

```
// Create Table description of all required columns (does not create DATA column!).
TableDesc td(MeasurementSet::requiredTableDesc());
// Setup and create the new MS (will use default data managers).
SetupNewTable setup("test.ms", td, Table::New);
MeasurementSet ms(setup);
// Create all required subtables.
ms.createDefaultSubtables(Table::New);
// Make sure everything is on disk.
ms.flush();

// Now fill the main table (only a few columns shown).
int nbl = nant * (nant+1) / 2;
int row = 0;
ms.addRow (ntime * nbl);
MSMainColumns msCols(ms);
for (int i=0; i<ntime; ++i) {
  for (int j=0; j<nant; ++j) {
    for (int k=j; k<nant; ++k) {
      msCols.antenna1().put (row, j);
      msCols.antenna2().put (row, k);
      msCols.time().put (row, time);
    }
  }
}
// Subtables should be filled similarly.
```

Creating an MS (advanced)

```
// Create description of main columns
TableDesc td(MS::requiredTableDesc());
// Add the DATA column (which is optional).
MS::addColumnToDesc(td, MS::DATA, 2);
// Add an extra LOFAR column.
td.addColumn (ScalarColumnDesc<Double>(”LOFAR_EXTRA_COLUMN"));

// Setup the new table.
// Most columns vary little and can use the IncrementalStMan to save storage.
SetupNewTable newTab(msName, td, Table::New);
IncrementalStMan incrStMan("ISMData");
newTab.bindAll (incrStMan);

// Use StandardStMan for faster varying columns.
StandardStMan    stanStMan;
newTab.bindColumn(MS::columnName(MS::ANTENNA1), stanStMan);
newTab.bindColumn(MS::columnName(MS::ANTENNA2), stanStMan);

// Use a TiledColumnStMan for the data, flags and UVW.
// Tileshape is [npol,nchan,nrow]
IPosition dataTileShape(3,4,32,128);
TiledColumnStMan tiledData("TiledData", dataTileShape);
newTab.bindColumn(MS::columnName(MS::DATA), tiledData);
TiledColumnStMan tiledFlag("TiledFlag", dataTileShape);
newTab.bindColumn(MS::columnName(MS::FLAG), tiledFlag);
TiledColumnStMan tiledUVW("TiledUVW", IPosition(3,128));
newTab.bindColumn(MS::columnName(MS::UVW), tiledUVW);

// Now create the table with 10 rows.
MeasurementSet ms(setup, 10);
```

Least squares fitting

- in scimath/Fitting
- linear and non-linear (iterative Levenberg-Marquardt)
  - optionally with SVD
- merging of matrices containing normal equations
  - makes distributed processing possible (as used by BBS)
- fitting to any function in scimath/Functionals
  - auto-differentation is possible
- fitting to own model parameters
- coarse and fine control
  - do entire fit (function fit) or step by step (solveLoop)
- extra constraints are possible
- can get chiSquare, covariance, nr of iterations, ...

- For examples see test programs in scimath/Fitting/test

Non-linear fitting example

```
// Make a (gaussian) data set 20.0 * exp (-((x-25)/4)^2) to fit against
const uInt n = 100;
Vector<Double> x(n);
Vector<Double> y(n);
Gaussian1D<Double> gauss1(20, 25.0, 4.0);
for (uInt i=0; i<n; i++) x[i] = i*0.5;
for (uInt i=0; i<n; i++) {
  value = gauss1(x[i]);
  y[i] = abs(value);
}

// Construct a gaussian function for fitting
// It has to be a Gaussian1D instantiated with an AutoDiff.
Gaussian1D<AutoDiff<Double> > gauss;

// Must give an initial guess for the set of fitted parameters.
Vector<Double> v(3);
v[0] = 2;
v[1] = 20;
v[2] = 10;
for (uInt i=0; i<3; i++) gauss[i] = AutoDiff<Double>(v[i], 3, i);
// Set the function in the Levenberg-Marquardt fitter.
NonLinearFitLM<Double> fitter;
fitter.setFunction(gauss);

// Perform fit and test if it converged.
Vector<Double> solution = fitter.fit(x, y, sigma);
if (! fitter.converged()) {
  cout << “no convergance” << endl;
}
```

Images

- N-dim rectangular array with coordinates and optional mask
  - Usual coordinates are ra,dec,stokes,freq
  - Can have multiple masks
  - Logging subtable for logging image operations
  - Auxiliary info (imageinfo, miscinfo) like telescope name

- Base class is ImageInterface<T>
  - Derived from MaskedLattice<T> (and Lattice<T> and LatticeBase)

- Four basic types

– PagedImage<T>	native casacore image type (tiled storage)
– FITSImage	image in FITS format
– MIRIADImage	image in MIRIAD format
– HDF5Image<T>	image stored in HDF5 format (tiled storage) Not according to ICD (might change in future)

- ImageOpener recognizes type and opens it correctly
- ImageProxy is high-level interface (similar to TableProxy)

Virtual Image types

SubImage<T>	a region in another image object
ImageConcat<T>	concatenate images
ImageExpr<T>	an expression of images
CurvedImage2D<T>	a cut through an image cube (e.g., following a spiral arm)

Image expression:

- E.g., addition or subtraction of two or more images

  ‘~/image1’ - ‘~/image2’
- Many operators and functions defined
- Calculated on-the-fly
- Automatically recognized by ImageProxy
- See note 223 about LEL

  http://www.astron.nl/casacore/trunk/casacore/doc/notes/223.html

Image regions

- World and pixel coordinates

- Box
- Ellipsoid (circle, ellipse, sphere, ...)
- Polygon
- Mask
- Combination of regions of any type
  - union
  - intersection
  - difference
    - e.g,. an annulus (ring) can be made by differencing 2 circles
- Stretch/extend (extend any region into other dimensions)
  - e.g., a cylinder can be made by extending a circle

Image access

- Direct access using functions in class (Masked)Lattice:
  - get
  - getSlice
  - getMask
  - getMaskSlice
  - put
  - putSlice

- Iterating using class LatticeIterator and LatticeNavigator
  - Get by using Array in cursor()
  - Put by assigning Array to woCursor() and rwCursor()
  - Also available as Vector, Matrix, and Cube

Find sum of image pixels

```
// Open the image (of any type). Make sure the data type is float.
// Note that use of CountedPtr takes care of automatic object deletion.
CountedPtr<LatticeBase> lattice (ImageOpener::openImage (fileName));
ImageInterface<float>* imagePtr = dynamic_cast<ImageInterface<float>*>
          (lattice.operator->());
AlwaysAssert (imagePtr, AipsError);

// Simple implementation; may fail for very large images.
Float total = sum(imagePtr->get());


// Better implementation by doing it in chunks using an iterator.

// Construct the iterator.  since we only want to read the image,
// use the read-only class, which disallows writing back to the cursor.
// No navigator is given, so the default TileStepper is used
// which ensures optimum performance.
RO_LatticeIterator<Float> iterator(*imagePtr);
// Add for each iteration step the sum of the cursor elements to sum.
// Note that the cursor is an Array object and that the function sum
// is defined in ArrayMath.h.
Float total = 0.0;
for (iterator.reset(); !iterator.atEnd(); iterator++) {
    total += sum(iterator.cursor());
}
```