

```
=====
                        pycrtools TUTORIAL
=====
```

Version History:

- 2010-03-01 - started (H. Falcke)
- 2010-03-24 - added hArray description (HF)

The library makes use of algorithms and code developed by Andreas Horneffer, Lars B"ahren, Martin vd Akker, Heino Falcke, ... "

To create a PDF version of the tutorial.py script use

```
./prettypy tutorial.py
```

in the \$LOFARSOFT/src/CR-Tools/implement/Pypline/doc directory.

Table of Contents

```
=====
```

1. StartUp
2. Getting Help
3. Vectors
  - 3.1. Some Basics
  - 3.2. Construction of STL Vectors
    - 3.2.1. Referencing, memory allocation, indexing, slicing
    - 3.2.2. Vector arithmetic
  - 3.3. Working with the hArray class
    - 3.3.1. Creating Arrays and basic operations
    - 3.3.2. Changing Dimensions
    - 3.3.3. Memory sharing
    - 3.3.4. Basic Slicing
    - 3.3.5. Selecting & Copying Parts of the Array - a List as Index
    - 3.3.6. Applying Methods to Slices
    - 3.3.7. Units and Scale Factors
    - 3.3.8. Keywords and Values
    - 3.3.9. History Logbook
4. File I/O
  - 4.1. Opening and Closing a CR Data File
  - 4.2. Setting and retrieving metadata (parameters)
  - 4.3. Reading in Data
    - 4.3.1. Reading different blocks of data
5. Basic Plotting
  - 5.1. Mathgl
  - 5.2. Matplotlib
  - 5.3. Plotting Using the hArray Plotting Method
6. CR Pipeline Modules
  - 6.1. Quality Checking of Time Series Data
  - 6.2. Finding peaks in a vector
  - 6.3. Fourier Transforms (FFT) & Cross Correlation
  - 6.4. Coordinates
  - 6.5. Coordinates
7. Appendix: Listing of all Functions:

1. StartUp

```
-----
```

First one needs to load the library. This can be done by "from hftools

import \*" which makes all the c++ functions and their python wrappers available.

In addition there are some helpful definitions in python as well, which are stored in "pycrtools.py". This file actually imports hftools for you. Hence, all you actually need to do is (make sure the file is in your search path)

```
>>> from pycrtools import *
```

The most convenient way is probably to define an alias in your .bashrc or .profile files, like

```
alias pycr='/sw/bin/python2.6 -i $LOFARSOFT/src/CR-Tools/implement/Pyipeline/pycrinit.py'
```

then you can simply start the pyrttools with 'pycr'.

The file pycrinit.py also allows you to read in a second file to be executed, like this one here. So, you can run the tutorial with

```
pycr -i tutorial.py
```

from the console prompt.

Alternatively use ipython

1. Download and install the latest development release of ipython

```
tar xvf ipython-0.11.tar.gz
cd ipython-0.11
sudo python setup.py install
```

2. Copy the profile from \$LOFARSOFT/src/CR-Tools/implement/Pyipeline/extras to your ~/.ipython directory

```
cp $LOFARSOFT/src/CR-Tools/implement/Pyipeline/extras/ipython_config_pycr.py ~/.ipython/
```

3. Compile hftools and set the following variables

```
PYTHONPATH=$PYTHONPATH:$LOFARSOFT/release/lib/python
PATH=$PATH:$LOFARSOFT/release/bin/python
```

4. Run the profile

```
ipython -p pycr
```

or set an alias

```
alias pycr='ipython -ppycr'
```

5. Use %run (with the full path to the script) to run the scripts from ipython.

The testscripts can be found in a directory that is stored in the variable PYCRBIN, so say

```
%run $PYCRBIN/testcr2.py
```

or actually in the scripts directory which is simply:

```
%run $PYCR/testcr2.py
```

-----

To use ipython with pycr within an emacs shell.

Copy the file

\$LOFARSOFT/src/CR-Tools/implement/Pyipeline/extras/ipython.el

to ~/Library/Preferences/Emacs/ipython.el

and then put a line

```
(load-file "~/Library/Preferences/Emacs/ipython.el")
```

into the ~/.emacs file in your home directory

## 2. Getting Help

=====

The pycrtools have a built-in help system, which can be accessed with:

```
>>> help()
```

To get information on a specific function or method type `help(func)`. This will essentially print the docstring of the python object and list all its methods. Hence, `help` will work on any decently implemented python object, including the standard ones.

For example,

```
>>> help(IntVec)
```

will give documentation on the integer vector, while

```
>>> help(IntVec.sin)
```

will give the documentation on the "sin" method associated with it.

For a listing of all available functions in the pycrtools type

```
>>> help(all).
```

## 3. Vectors

=====

### 3.1. Some Basics

-----

The fundamental data structure we use is a standard c++ vector defined in the c++ standard template library (STL). This is wrapped and exposed to python using the Boost Python system.

(NB: Unfortunately different systems provide different python data structures. Hence a function exposed to python with SWIG or SIP is not directly able to accept a BOOST PYTHON wrapped vector as input or vice

versa. If you want to do this you have to provide extra conversion routines.)

In line with the basic python philosophy, vectors are passed as references. Since we are working with large sets of data processing time is as important as convenience. Hence the basic principle is that we try to avoid copying large chunks of data as much as possible.

The basic functions operating on the data in c++ either take STL iterators as inputs (i.e. pointers to the begin and end of the memory where the data in the STL vector is stored) or `casa::Vectors`, which are created with shared memory (i.e. their physical memory is the same as that of the STL vector).

For that reason MEMORY ALLOCATION is done almost exclusively in the Python layer. The fast majority of c++ functions is not even able to allocate or free any memory. This allows for very efficient memory management and processing, but also means that the user is responsible for providing properly sized vectors as input AND output vectors. I.e.: you need to know beforehand what sized vector you expect in return!

This may be annoying, but forces you to think carefully about how to use memory. For example, if there is a processing done multiple times using a scratch vector of fixed size, you reuse the same vector over and over again, thus avoiding a lot of unnecessary allocation and deallocation of memory and creation of vectors.

Also, the basic vectors are inherently one-dimensional and not multi-dimensional. On the other hand, multi-dimensional data is always simply written sequentially into the memory - you just need to know (and think about) how your data is organized. Some rudimentary support for multi-dimensions has been added (if the data you need is contiguous), but needs further work (see `setDim`, `getDim`, `elem`).

### 3.2. Construction of STL Vectors

-----

A number of vector types are provided: `bool`, `int`, `float`, `complex`, and `str`.

To create a vector most efficiently, use the original vector constructors:

```
- BoolVec()  
- IntVec()  
- FloatVec()  
- ComplexVec()  
- StringVec()
```

e.g.

```
>>> v=FloatVec(); v
```

will create a floating point vector of size 0.

The vector can be filled with a python list or tuple, by using the `extend` attribute:

```
>>> v.extend([1,2,3,4])
```

```
      v => Vector(float, 4, fill=[1,2,3,4])
```

Note, that python has automatically converted the integers into floats, since the STL vector does not allow any automatic typing.

The STL vector can be converted back to a python list by using the python list creator:

```
list(v) => [1.0, 2.0, 3.0, 4.0]
```

or use the list or val methods of the vector (where the latter has the extra twist that it will return a scalar value, if the vector has a length of one).

```
v.val() => [1.0, 2.0, 3.0, 4.0]
v.list() => [1.0, 2.0, 3.0, 4.0]
```

However, the basic Boost Python STL vector constructor takes no arguments and is a bit cumbersome to use in the long run. Here we provide a wrapper function that is useful for interactive use.

Usage:

Vector(Type) - will create an empty vector of type "Type", where Type is a basic Python type, i.e. bool, int, float, complex, str.

Vector(Type,size) - will create an vector of type "Type", with length "size".

Vector(Type,size,fill) - will create an vector of type "Type", with length "size" and initialized with the value "fill"

Vector([1,2,3,...]) or Vector((1,2,3,...)) - if a list or a tuple is provided as first argument then a vector is created of the type of the first element in the list or tuple (here an integer) and filled with the contents of the list or tuple.

So, what we will now use is:

```
>>> v=Vector([1.,2,3,4]); v
Vec(4)=[1.0,2.0,3.0,4.0]
```

Note, that size and fill take precedence over the list and tuple input. Hence if you create a vector with Vector([1,2,3],size=2) it will contain only [1,2]. Vector([1,2,3],size=2,fill=4) will give [4,4].

Some simple support for multiple dimensions had been implemented, using the methods:

```
vector.setDim([n1,n2,...])
vector.getDim()
vector.elem(n)
```

However, this is already depreciated, since there is an array class to do this better.

### 3.2.1. Referencing, memory allocation, indexing, slicing .....

Following basic python rules, the STL vector will persist in memory as

long as there is a python reference to it. If you destroy v also the c++ memory will disappear. Hence, if you keep a pointer to the vector in c++ and try to work on it after the python object was destroyed, your programme may crash. That is why by default memory management is done ONLY on one side, namely the python side!

To illustrate how Python deals with references, consider the following example:

```
>>> x=v
>>> x[0]=3
>>> v
      Vec(4)=[3.0,2.0,3.0,4.0]
```

Hence, the new python object x is actually a reference to the same c++ vector that was created in v. Modifying elements in x modifies elements in v. If you destroy v or x, the vector will not be destroyed, since there is still a reference to it left. Only if you destroy x and v the memory will be freed.

As noted above, this vector is subscriptable and sliceable, using the standard python syntax.

```
>>> v[1:3]
      Vec(2)=[2.0,3.0]
```

We can also resize vectors and change their memory allocation:

```
>>> v1=Vector([0.0,1,2,3,4,5]);v1
>>> v2=Vector(float,len(v1),2.0);v2
```

With the resize attribute you allocate new memory while keeping the data. It is not guaranteed that the new memory actually occupies the same physical space.

```
>>> v2.resize(8);
>>> v2
```

Resize a vector and fill new entries with non-zero values:

```
>>> v2.resize(10,-1)
>>> v2
```

Resize a vector to same size as another vector:

```
>>> v2.resize(v1)
>>> v2
```

Make a new vector of same size and type as the original one:

```
>>> v3=v2.new()
>>> v3
```

Fill a vector with values

```
>>> v3.fill(-2)
>>> v3
```

3.2.2. Vector arithmetic

.....

The vectors have a number of mathematical functions attached to them. A full list can be seen by typing

```
>>> dir(v1)
```

Some of the basic arithmetic is available in an intuitive way.

You can add a scalar to a vector by

```
>>> v1+3
```

This will actually create a new vector (and destroy it right away, since no reference was kept). The original vector is unchanged.

A technical limitation is that - even though addition and multiplication is commutative, the scalar (i.e., non-vector) values has to come as the second argument.

You can also add two vectors (which is commutative):

```
>>> v1+v2
```

In order to change the vector, you can use the "in place" operators `+=`, `-=`, `/=`, `*=` :

Adding a vector in place:

```
>>> v1+=v2
v1 => Vector(float, 6, fill=[2,3,4,5,6,7])
```

now v1 was actually modified such that v2 was added to the content of v1 and the results is stored in v1.

Similarly you can do

```
- v1-=v2
- v1*=v2
- v1/=v2
```

Here are examples of some basic statistics functions one can use

Mean:

```
v1.mean() => 4.5
```

Median:

```
v1.median() => 5.0
```

Summing all elements in a vector:

```
v1.sum() => 27.0
```

Standard Deviation:

```
v1.stddev() => 1.87082869339
```

3.3. Working with the hArray class

### 3.3.1. Creating Arrays and basic operations

While the basic underlying data structures are plain STL vectors, in many cases, however, one has to deal with multi-dimensional data. For this case we introduce a new wrapper class, named `hArrays`, that mimicks a multi-dimensional array, but still operates on an underlying vector with essentially a flat, horizontal data structure. Given that a major concern is to minimize duplication of large data structures, the array class will share memory with its associated vector and also with arrays that are derived from it. Explicit copying will have to be done in order to avoid this. Access to various dimensions (rows, columns, etc...) is done via slices that need to be contiguous in memory! Since the array is vector-based, all methods defined for vectors are also inherited by `hArrays` and can be applied to slices or even automatically loop over multiple slices (e.g., rows or columns).

An array is defined using the `hArray` function. This is a constructor function and not a class of its own. It will return array classes of different types, such as `IntArray`, `FloatArray`, `ComplexArray`, `StringArray`, `BoolArray`, referring to the different data types they contain. As for vectors, each array can only contain one type of data.

```
hArray(Type=float,dimensions=[n1,n2,n3...],fill=None) -> FloatArray
```

where `Type` can be a Python type, a Python list/tuple (where the first element determines the type), an STL vector, or another `hArray`.

Dimensions are given as a list of the form `[dim1,dim2,dim3, ...]`. The size of the underlying vector will automatically be resized to `dim1*dim2*dim3* ...` to be able to contain all elements. Alternatively, one can provide another array, whose dimensions will be copied. '

The array can be filled with an initialization values that can be either a single value, a list, a tuple, or an STL vector of the same type.

```
>>> v=Vector(range(9))
```

```
>>> a=hArray(v,[3,3])
```

```
a => hArray(int, [3, 3], fill=[0,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])
```

One may wonder what the representation of the Array actually means.

```
a => hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,0,0,0]
```

After "`hArray("` first the data type is given, then the array dimensions and total vector size, and finally the currently active slice (given as start and end index of the vector). An optional asterisk indicates that the next operation will actually loop the previously specified slices (see below). At the end the currently selected slice is displayed (while the array may actually hold more data).

The underlying vector of an array can be retrieved with the `.vec()` method. I.e.,



```
>>> a.vec()
```

The arrays have most of the vector methods defined, so you can also add, multiply, etc. with scalars or other arrays.

```
a*2 => hArray(int, [3, 3], fill=[0,2,4,6,8,10,12,14,16]) # len=9 slice=[0:9])
```

```
a*a => hArray(int, [3, 3], fill=[0,1,4,9,16,25,36,49,64]) # len=9 slice=[0:9])
```

Underlying these operations are the basic hftools functions, e.g. the multiplication is essentially a python method that first creates a new array and then calls hMul.

```
>>> tmp_array=a.new()
>>> tmp_array.mul(a,2)
```

```
a => hArray(int, [3, 3], fill=[0,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])
```

BTW, this could also be done calling the function hMul(tmp\_array,a,2), rather than the corresponding method.

An important constraint is that all these functions or methods only work with either vector or array classes, a mix in the parameters between vectors and arrays is currently not supported.

### 3.3.2. Changing Dimensions

.....

The dimensions can be obtained and set, using the getDim and setDim methods. If the length of the underlying vector changes due to a change in the dimensions, the vector will be resized and padded with zeros, if necessary.

```
>>> a.getDim()
```

```
>>> a.setDim([3,3,2])
```

```
>>> a.setDim([3,3])
```

### 3.3.3. Memory sharing

.....

Note, that the array and vector point share the same memory. Changing an element in the vector

```
>>> v[0]=-1
```

```
v => Vector(int, 9, fill=[-1,1,2,3,4,5,6,7,8])
a => hArray(int, [3, 3], fill=[-1,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])
```

will also change the corresponding element in the array. The same is, btw, true if one creates an array from an array. Both will share the same underlying data vector. They will, btw, also share the same size and dimensions.

```
>>> b=hArray(a)
```

```
>>> b[0,0]=-2
```

```

b => hArray(int, [3, 3], fill=[-2,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])
a => hArray(int, [3, 3], fill=[-2,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])
v => Vector(int, 9, fill=[-2,1,2,3,4,5,6,7,8])

```

```
>>> v[0]=0
```

```

a => hArray(int, [3, 3], fill=[0,1,2,3,4,5,6,7,8]) # len=9 slice=[0:9])

```

To actually make a physically distinct copy, you need to explicitly copy the data over.

```
>>> c=hArray(int,a)
```

```
>>> a.copy(c)
```

or more simply

```
>>> c=hArray(int,a,a)
```

(the 2nd parameter is for the dimensions, the third one is the fill parameter that initiates the copying).

### 3.3.4. Basic Slicing

.....

The main purpose of these arrays is, of course, to be able to access multiple dimensions. This is done using the usual `__getitem__` method of python.

Let us take our two-dimensional array from before:

```

a => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])

```

The vector followed by a single number in square brackets will "in principle" obtain the first column of the array.

```
a[0] => hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[0:3])
```

It says "in principle", because the only thing which that command does is to return a new `hArray` python object, which will point to the same data vector, but contain a different data slice which is then returned whenever a method tries to operate on the vector.

```
>>> a[0].vec()
```

This retrieves a copy of the data, since assigning a sub-slice of a vector to another vector actually requires copying the data - as vectors do not know about slicing (yet). Use one-dimensional arrays if you want to have are reference to a slice only.

In contrast, `a.vec()`, without slicing, will give you a reference to the underlying vector.

For convenience `a[0,1]` will return the value, rather than a one element slice. That behaviour changed from earlier versions and is a bit inconsistent ...

One may wonder, why one has to use the extra methods `vec` and `val` to access the data. The reason is that slicing on its own will return an array (and not a vector), which we need for other purposes still.

Slicing can also be done over multiple elements of one dimension, using the known python slicing syntax:

```
>>> a[0,0:2].val()
```

however, currently this is restricted to the last dimension only, in order to point to a contiguous memory slice. Hence:

```
>>> a[0:2]
```

is possible, but not

```
>>> a[0:2,0:2]
```

where the first slice is simply ignored.

Finally, negative indices count from the end of the slice, i.e.

```
>>> a[-1]
```

gives the last slice of the first index, while

```
>>> a[0:-1]
```

gives all but the last slice of the first index.

### 3.3.5. Selecting & Copying Parts of the Array - a List as Index .....

It is possible to provide a list of indices as the last index to copy parts of the array.

```
>>> print "ERROR! a[[0,2,3]] does not work - error in harray.py hArray_getSlicedArray -r  
aise IndexError"  
ERROR! a[[0,2,3]] does not work - error in harray.py hArray_getSlicedArray -raise Inde  
xError
```

This will return a new array containing the first third and fourth element of array a.

You can do this explicitly, using the copy method:

```
>>> acopy=a.new()  
>>> acopy.copy(a,hArray([0,2,3]),-1)  
acopy => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
```

The last parameter of "copy" (-1) provides the possibility to copy only the first nth elements of the index vector (if the last parameter is n>0) - hence it is possible to operate with fixed length index vectors.

Where this becomes important is when selecting certain elements in an array, e.g. based on its values. For, example, the the 'Find' methods will return a list of indices of elements for which a particular condition is true. Most commonly used may be the methods 'hFindGreaterThan', 'hFindGreaterEqual', 'hFindGreaterEqualAbs', 'hFindLessThan', 'hFindLessEqual', 'hFindLessThanAbs', 'hFindLessEqualAbs', 'hFindBetween', 'hFindBetweenOrEqual', 'hFindOutside', 'hFin

```
dOutsideOrEqual'.
```

Assume, we want to have a list of all the elements of `a` that are between the values (but excluding) 0 and 10 and perform an operation on it. Then we need to create an index vector first.

```
>>> indices=hArray(int,dimensions=a,fill=-1)
```

and fill it with the indices according to our condition

```
>>> number_of_indices=indices[...].findbetween(a[...],0,8)
      number_of_indices => Vector(int, 3, fill=[0,0,0])
>>> indices[...].pprint(-1)
      [-1,-1,-1]
      [-1,-1,-1]
      [-1,-1,-1]
```

As the result we get a vector with the number of elements in each row that have satisfied the condition and in 'indices' we get their position. Note that the indices vector must be of large enough to hold all indices, hence in the general case needs to be of the same size (and dimension) as the input data array. Following our basic philosophy, the index vector will not be automatically resized. If the number of selected indices is smaller than the remaining spaces simply remain untouched (contain whatever was in there before). To illustrate this effect, we filled the indices array with "-1"s. If, on the other hand, the vector were too short it will be filled until the end and then the search stops. No error message will be given in this case - this is a feature.

To retrieve the selected elements we make use of the copy method again to create a new array.

```
>>> b=a.new(); b.fill(-99)
>>> b[...].copy(a[...],indices[...,[0]:number_of_indices],number_of_indices)
>>> b.pprint(-1)
      [-99,-99,-99,-99,-99,-99,-99,-99,-99]
```

This (contiguous) with variable length we can use for further looping operations (as described below) on the rows of the array. E.g.,

```
>>> b[...,[0]:number_of_indices].sum()
```

will take the sum of the first `n` elements in each row of our array, where `n` given by the vector `number_of_indices` that were returned by our find operation. Clearly, the -99 values that we put into our array for demonstration purposes were not taken into account for the sum of the rows. Note, that the slice specification in the line above needs to have either vectors or scalar values, but not a mix of the two. This is the reason for using `[0]:number_of_indices` rather than just `0:number_of_indices`.

BTW, nicer would have been to do right away something like the following:

```
a[indices[...,[0]:number_of_indices],...].sum() (NOT!)
```

but that is not yet implemented, since looping cannot yet be done over nested indices!

### 3.3.6. Applying Methods to Slices

.....

First, of all, we can apply the known vector functions also to array slices directly. E.g.,

```
>>> a[0].sum()
```

will return the sum over the first row of the array, i.e. the first three elements of the underlying vector. While

```
>>> a[0].negate()
a => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
>>> a[0].negate()
a => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
```

returns nothing, but will actually change the sign of the first three elements in the underlying vector.

In principle one could now loop over all slices using a for loop:

```
>>> for i in range(a.getDim()[0]): print "Row",i,":",a[i].val(), " => a =", a
Row 0 : [0, 0, 0] => a = hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
Row 1 : [0, 0, 0] => a = hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
Row 2 : [0, 0, 0] => a = hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
```

However, looping over slices in simple way is already built into the arrays, by appending the Ellipsis symbol "..." to the dimensions. This will actually put the array in "looping mode".

```
>>> l=a[0:3,...]
l => hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[0:3]*)
```

which is indicated in the screen representation of the array by an extra asterisk and actually means that one can loop over all the elements of the respective dimension.

```
>>> iterate=True
>>> while iterate:
...     print "Row",l.loop_nslice(),":",l.val(), " => l =", l
...     iterate=l.next().doLoopAgain()
Row 0 : [0, 0, 0] => l = hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[0:3]*
Row 1 : [0, 0, 0] => l = hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[3:6]*
Row 2 : [0, 0, 0] => l = hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[6:9]*

l => hArray(int, [3, 3], fill=[0,0,0]) # len=9 slice=[0:3]*)
```

This will do exactly the same as the for-loop above.

Here doLoopAgain() will return True as long as the array is in looping mode and has not yet reached the last slice. loop\_nslice() returns the current slice the array is set to (see also loop\_i, loop\_start, loop\_end). next() will advance to the next slice until the end is reached (and doLoopAgain is set to false). The loop will be reset at the next call of next(). Hence, as written above the loop could be called multiple times where the loop will be automatically reset each

time.

We could also explicitly reset the loop in using to its starting values, but that should not be necessary most of the time.

```
>>> l.resetLoop()
```

Now, since this is still a bit too much work, you can actually apply (most of) the available vector methods to multiple slices at once, by just applying it to an array in looping mode.

As an example, let us calculate the mean value of each slice at the to level of our example array, which is simply:

```
>>> l.mean()
```

In contrast to the same method applied to vectors, where a single value is returned, the return value is now a vector of values, each of which corresponds to the mean of one top-level slice. Hence, the vector has looped automatically over all the slices specified in the definition of the array.

The looping over slices can be more complex taking start, stop, and increment values into account:

```
>>> a[1:,...].mean()
```

will loop over all top-level slices starting at the 2nd slice (slice #1) until the last.

```
>>> a[:2,...].mean()
```

will loop over the first two top-level slices.

```
>>> a[0:3:2,...].mean()
```

will loop over the two top-level slices using an increment of 2, i.e. here take the first and third only (so, here non contiguous slices can be put to work).

To loop over all slices in one dimensions, a short-cut can be used by leaving away the slice specification. Hence,

```
>>> a[...].mean()
```

will do the same as `a[0:,...].mean()`.

it is even possible to specify an array of indices for the slicing.

```
>>> a[[0,2],...].mean()
```

will loop over slices 0 and 2.

It is possible to specify a slice after the ellipse, e.g.,

```
>>> a[...,0:2].mean()
```

which means that the mean is taken only from the first two elements of each top-level slice.

Even more complicated: the elements of the slice can be vectors or lists:

```
>>> a[...,[0,1]:[2,3]].mean()
```

over which one can loop as well. Hence, in the operation on the first row, the subslice 0:2 will be taken, while for the second slice/row the slice 1:3 is used.

(+++++) Parameters of looping arrays

-----

Looping can also be done for methods that require multiple arrays as inputs. In this case the next() method will be applied to every array in the parameter list and looping proceeds until the first array has reached the end. Hence, care has to be taken that the same slice looping is applied to all arrays in the parameter list.

As an example we create a new array of the dimensions of a

```
>>> x=hArray(int,a)
```

and fill it with slices from "a" multiplied by the scalar value 2

```
>>> x[[0,2],...].mul(a[[0,2],...],2)
```

```
x => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
```

and indeed now the first and last slice were operated on and filled with the results of the operation.

Forgetting slicing in a parameter can lead to unexpected results, e.g., in the following example "a" is looped over but x is not. Hence, the result will always be written (and overwritten) into the first three elements of x, containing at the end only the results of the multiplication of the last slice in a.

```
>>> x.fill(0); x[...].mul(a,2)
```

```
x => hArray(int, [3, 3], fill=[0,0,0,0,0,0,0,0,0]) # len=9 slice=[0:9])
```

NOTE: There are currently relatively strict rules on how to change the parameters from a vector to an array.

1) When going from a vector to an array, all other vectors in the argument list also have to be provided as arrays!

2) Scalar parameters can be provided as single-valued scalars or as vectors. In the latter case the algorithm will take one element after the other in each loop as input parameter.

3) If one scalar parameter is provided as a vector, all scalar parameters have to be provided as Vectors (they can be of different length and of length unity, though, which means that always the same value is taken.)

4) If an algorithm has a scalar return value, a vector of values will be returned by the same algorithm if invoked with arrays.

5) If a slice is specified with vectors as elements (i.e. [1,2,3]:[5,6,7]), both start and stop have to be vectors. The

algorithm will then loop over all elements in the lists.

### 3.3.7. Units and Scale Factors

.....

Numerical arrays allow one to set a (single) unit for the data. With `setUnit(prefix, unit_name)` one can specify the name of the unit and the scale factor, which is specified as a string being one of 'f', 'p', 'n', 'micro', 'm', 'c', 'd', 'h', 'k', 'M', 'G', 'T', 'P', 'E', 'Z'.

```
>>> a.setUnit("M", "Hz")
```

will set the unitname to MHz without modifying the values in the array (assuming that the values were delivered initially in this unit). However, the scaling can be changed by calling `setUnit` again (with or without a unit name), e.g.:

```
>>> a.setUnit("k", "")
```

Which has converted the values to kHz. The name of the unit can be retrieved with

```
>>> a.getUnit()
```

and cleared with `clearUnit()`.

### 3.3.8. Keywords and Values

.....

For documenting the vector further and to store certain values, one can store keywords and values in the array. This is done with

```
>>> a.setKey("name", "TestArray")
```

The keywords can be arbitrary strings and the values also arbitrary strings. Thus numbers need to be converted to strings and back. The keyword 'name' is special in the sense that it is a default key, that is recognized by a number of other modules (including the `__repr__` method governing array output) to briefly describe the data.

The keyword values can be retrieved using `getKey`:

```
>>> a.getKey("name")
```

### 3.3.9. History Logbook

.....

The array keeps a history of most operations performed with it. This can be viewed by using the `getHistory` (retrieves a vector of strings) or simply the `.history()` method, which prints the activity log.

```
>>> a.history()
```

You can add a line to the history log yourself with

```
>>> a.addHistory("TEST", "This is a test.")
```

and clear the entire log with `.clearHistory()`.



History tracking can be switched off with

```
>>> a.setHistory(False)
```

#### 4. File I/O

-----

##### 4.1. Opening and Closing a CR Data File

-----

Let us see how we can open a file. First define a filename, e.g.:

```
>>> filename_sun=LOFARSOFT+"/data/lopes/example.event"
>>> filename_lofar_big=LOFARSOFT+"/data/lofar/rw_20080701_162002_0109.h5"
>>> filename_lofar=LOFARSOFT+"/data/lofar/trigger-2010-02-11/triggered-pulse-2010-02-11-TB
B1.h5"
>>> filename_lofar_onesecond=LOFARSOFT+"/data/lofar/RS307C-readfullsecond.h5"
```

```
filename_sun => /Users/falcke/LOFAR/usg/data/lopes/example.event
filename_lofar_big => /Users/falcke/LOFAR/usg/data/lofar/rw_20080701_162002_0109.h5
filename_lofar => /Users/falcke/LOFAR/usg/data/lofar/trigger-2010-02-11/triggered-puls
e-2010-02-11-TBB1.h5
```

We can create a new file object, using the "crfile" class, which is an interface to the LOFAR CRTOOLS datareader class and was defined in pycrtools.py.

The following will open a data file and return a DataReader object:

```
>>> datafile=crfile(filename_lofar).set("blocksize",1024*2)
```

The associated filename can be retrieved with

```
>>> datafile.filename
```

The file will be automatically closed (and the DataReader object be destroyed), whenever the crfile object is deleted, e.g. with "file=0".

##### 4.2. Setting and retrieving metadata (parameters)

-----

Now we need to access the meta data in the file. This is can be done in multiple ways. One way is by using the get method. This method actually calls the function 'hFileGetParameter' defined in the c++ code.

Which observatory did we actually use?

```
>>> obsname=datafile.get("observatory");
obsname => LOFAR
```

There are more keywords, of course. A list of implemented parameters we can access is obtained by

```
>>> keywords=datafile.get("help")
hFileGetParameter - available keywords: nofAntennas, nofSelectedChannels, nofSelectedAntennas, nofBaselines, block, blocksize, stride, fftLength, nyquistZone, sampleInterval, referenceTime, sampleFrequency, antennas, selectedAntennas, selectedChannels, positions, shift, increment, frequencyValues, frequencyRange, Date, Observatory, Filesize, presync, TL, L
```

TL, EventClass, SampleFreq, StartSample, AntennaIDs, SAMPLE\_OFFSET, SAMPLE\_NUMBER, TIME, keywords, help

Note, that the results are returned as PythonObjects. Hence, this makes use of the power of python with automatic typing. For, example

```
>>> datafile.get("frequencyRange")
```

actually returns a vector.

Here no difference is made where the data comes from. The keyword Observatory accesses the header record in the data file while the frequencRange accesses a method of the DataReader.

A second way do retrieve data is to use square brackets, since datafile[key] is equivalent to datafile.get(key).

```
>>> datafile["blocksize"]
```

Just for fun let us define a number of variables that contain essential parameters (we will later actually use different ones which are automatically stored in the datafile object).

```
>>> obsdate    =datafile["Date"]
>>> filesize   =datafile["Filesize"]
>>> blocksize  =datafile["blocksize"]
>>> nAntennas  =datafile["nofAntennas"]
>>> antennas   =datafile["antennas"]
>>> antennaIDs=datafile["AntennaIDs"]
>>> selectedAntennas=datafile["selectedAntennas"]
>>> nofSelectedAntennas=datafile["nofSelectedAntennas"]
>>> fftlength  =datafile["fftLength"]
>>> sampleFrequency =datafile["sampleFrequency"]
>>> maxblocksize=min(filesize,1024*1024);
>>> nBlocks=filesize/blocksize;

obsdate => 1265926154
filesize => 132096
blocksize => 2048
nAntennas => 16
antennas => Vector(int, 16, fill=[128002016,128002017,128002018,128002019,128002020,128002021,128002022,128002023,128003024,128003025,128003026,128003027,128003028,128003029,128003030,128003031])
antennaIDs => Vector(int, 16, fill=[128002016,128002017,128002018,128002019,128002020,128002021,128002022,128002023,128003024,128003025,128003026,128003027,128003028,128003029,128003030,128003031])
selectedAntennas => Vector(int, 16, fill=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
nofSelectedAntennas => 16
fftlength => 1025
sampleFrequency => 200000000.0
maxblocksize => 132096
nBlocks => 64
```

To get a readable version of the observing date use the python time module

```
>>> import time
time.asctime(time.localtime(obsdate)) => Thu Feb 11 23:09:14 2010
```

Luckily, you do not have to do this all the time, since all the parameters will be read out at the beginning and will be stored as

attributes to the file object.

```
>>> for kw in datafile.keywords: p_("datafile."+kw)
    datafile.blocksize => 2048
    datafile.nofBaselines => 120
    datafile.filesize => 132096
    datafile.increment => Vector(float, 2, fill=[5e-09,97656.2])
    datafile.selectedAntennas => Vector(int, 16, fill=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
15])
    datafile.SAMPLE_OFFSET => Vector(int, 16, fill=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
    datafile.shift => Vector(int, 16, fill=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
    datafile.nofAntennas => 16
    datafile.sampleFrequency => 200000000.0
    datafile.nofSelectedAntennas => 16
    datafile.nofSelectedChannels => 1025
    datafile.frequencyRange => Vector(float, 2, fill=[0,1e+08])
    datafile.date => 1265926154
    datafile.observatory => LOFAR
    datafile.antennaIDs => Vector(int, 16, fill=[128002016,128002017,128002018,128002019,1
28002020,128002021,128002022,128002023,128003024,128003025,128003026,128003027,128003028,1
28003029,128003030,128003031])
    datafile.nyquistZone => 1
    datafile.sampleInterval => 5e-09
    datafile.antennas => Vector(int, 16, fill=[128002016,128002017,128002018,128002019,128
002020,128002021,128002022,128002023,128003024,128003025,128003026,128003027,128003028,128
003029,128003030,128003031])
    datafile.fftLength => 1025
    datafile.referenceTime => 0.0
    datafile.SAMPLE_NUMBER => Vector(int, 16, fill=[160628224,160628224,160628224,16062822
4,160628224,160628224,160628224,160628224,160628224,160628224,160628224,160628224,16062822
4,160628224,160628224,160628224])
    datafile.stride => 0
    datafile.selectedChannels => Vector(int, 1025, fill=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,1
4,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,4
4,45,46,47,48,49,...,975,976,977,978,979,980,981,982,983,984,985,986,987,988,989,990,991,9
92,993,994,995,996,997,998,999,1000,1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1011
,1012,1013,1014,1015,1016,1017,1018,1019,1020,1021,1022,1023,1024])
    datafile.tIME => Vector(int, 16, fill=[1265926154,1265926154,1265926154,1265926154,126
5926154,1265926154,1265926154,1265926154,1265926154,1265926154,1265926154,1265926154,12659
26154,1265926154,1265926154,1265926154])
    datafile.block => 0
```

They will be updated whenever you do a `file.set(key,value)`, however, assigning a new value to the attribute will NOT automatically change the parameter in the file. For this, you have to use the `'set'` method, which is an implementation of the `'hFileSetParameter'` function. E.g. changing the blocksize we already did before. This is simply

```
>>> datafile.set("blocksize",2048);
```

again the list of implemented keywords is visible with using

```
>>> datafile.set("help",0);
```

Here the listed keywords actually start with capital letters, however, to spare you some annoyance, you can use a spelling which starts with either an upper or a lower case letter.

Another useful feature: The `set` method actually returns the `crfile`

object itself. Hence, you can append multiple set commands after each other.

```
>>> datafile.set("block",2).set("selectedAntennas",[0,2,3]);
```

Alternatively you can also use square brackets:

```
>>> datafile["selectedAntennas"]=[0,2]
```

but then it is not possible to append multiple set commands in one line, so you need to provide lists of keywords and list of values, like

```
>>> datafile["blocksize","selectedAntennas"]=[2048,[0,2]]
```

```
>>> datafile["blocksize","selectedAntennas"]
```

Note, that we have now reduced the number of antennas to two: namely antenna 0 and 2 and the number of selected antennas is

```
>>> datafile["nofSelectedAntennas"]
```

However, in the following we want to work on all antennas again, so we do

```
>>> datafile.set("block",0).set("selectedAntennas",range(nAntennas))
```

#### 4.3. Reading in Data

-----

The next step is to actually read in data. This is done with the read method (accessing 'hFileRead'). The data is read flatly into a 1D vector. This is also true if multiple antennas are read out at once. Here simply the data from the antennas follow each other.

Also, by default memory allocation of the vectors has to be done in python before calling any of the functions. This improves speed and efficiency, but requires one to program carefully and to understand the data structure.

First we create a FloatArray of the correct dimensions, naming it Voltage and setting the Unit to counts.

```
>>> fxdata=hArray(float,[nofSelectedAntennas,blocksize],name="E-Field").setUnit("", "Counts")
```

This is now a large vector filled with zeros.

Now we can read in the raw time series data, either using 'datafile.read' and a keyword, or actually better, use the read method of arrays, as they then store filename and history information in the array.

Currently implemented keywords for reading data fields are: 'Fx', 'Voltage', 'FFT', 'CalFFT', 'Time', 'Frequency' (and 'TimeLag').

So, let us read in the raw time series data, i.e. the electric field in the antenna as digitized by the ADC. This is provided by the keyword 'Fx' (means  $f(x)$  ).

```
>>> fxdata.read(datafile,"Fx")
fxdata => hArray(float, [16, 2048], fill=[16,17,10,18,17,6,16,17,-1,0,10,4,-6,3,1,-8,0
,5,-1,10,15,9,14,20,17,13,16,13,9,6,4,-2,-2,3,1,-8,-3,5,2,4,11,16,12,16,22,16,10,16,16,8,.
..,-5,-23,-24,-1,-2,-12,2,15,1,-4,13,16,-5,-1,5,-8,-10,-12,-22,-19,-7,-18,-22,-5,-3,-9,-1,
5,3,4,9,6,2,1,4,-6,-13,-2,-7,-15,-9,-12,-25,-13,-7,-15,-5,3,-2,1], name="E-Field") # len=3
2768 slice=[0:32768])
```

and voila the vector is filled with time series data from the data file. Note that we had to use the .vec method for the array, since datafile.read does not yet accept arrays (since it cannot handle c++ iterators).

Now, you can access the individual antennas as single vectors through slicing

```
>>> ant0data=fxdata[0].vec()
```

If you do not have yet a pre-existing array into which you want to read data, you can automatically create one, using the square brackets syntax already known from retrieving the file header keywords. So, for example, to get the x-Axis we create a second vector

```
>>> times=datafile["Time"]
(Note: you can also create an empty array with the same properties and
dimensions, but without reading data into the array, by preceding the
keyword with the word 'empty', i.e. times=datafile['emptyTime'].)
```

In the square bracket notation python will actually set the name and units of the data accordingly.

So, let us have the time axis in microseconds, by using setUnit

```
>>> times.setUnit("\\mu","")
```

We do the same now for the frequency axis, which we convert to MHz.

```
>>> frequencies=datafile["Frequency"].setUnit("M","")
```

We can calculate the average spectrum of the data set for one antenna, by looping over all blocks. Here we do not use the square bracket notation, since we want to read the data repeatedly into the same memory location.

```
>>> fftdata=datafile["emptyFFT"]
>>> avspectrum=hArray(float,dimensions=fftdata,name="average spectrum")
>>> for block in range(nBlocks):
...     fftdata.read(datafile.set("Block",block),"FFT").none()
...     avspectrum[...].spectralpower(fftdata[...])
```

(The .none() method is appended to suppress unwanted output in generating the tutorial, when an operation returns an array or vector.)

Alternatively you can use the method

```
>> avspectrum.craveragespectrum(datafile)
```

which does this automatically.

#### 4.3.1. Reading different blocks of data

NB: There is the advanced possibility to specify during a read operation a list/vector of blocks to be read in, while using the array looping mechanism (e.g., to fill the rows of a matrix).

First the usual, boring one-block read:

```
>>> filename=LOFARSOFT+"/data/lofar/rw_20080701_162002_0109.h5"
>>> datafile=crfile(filename)
Opening LOFAR File=/Users/falcke/LOFAR/usg/data/lofar/rw_20080701_162002_0109.h5
[DataReader::summary]
-- blocksize ..... = 1024
-- FFT length ..... = 513
-- file streams connected? 0
-- shape(adc2voltage) ... = [1024, 1]
-- shape(fft2calfft) .... = [513, 1]
-- nof. antennas ..... = 1
-- nof. selected antennas = 1
-- nof. selected channels = 0
-- shape(fx) ..... = [1024, 1]
-- shape(voltage) ..... = [1024, 1]
-- shape(fft) ..... = [513, 1]
-- shape(calfft) ..... = [513, 1]
>>> datafile["blocksize"]=10
>>> f=hArray(float,[10])
>>> f.read(datafile,"Fx")
>>> f.pprint(-1)
[29,29,35,27,9,-8,-45,-74,-92,-109]
```

Then we read

```
>>> datafile["blocksize"]=2
>>> x=hArray(float,[5,2])
>>> hFileRead(datafile,"Fx",x[...],Vector([0,1]))
>>> x.mprint()
[29,29]
[35,27]
[29,29]
[35,27]
[29,29]
```

Note, that here block 0,1 and one are read in 2-3 times, since the array was longer then the block list specified! This is the default behaviour of looping arrays. Also, here you need to specify the list of blocks as a vector (not an array and not a list), since in this example we call the slightly less flexible c++ interface function directly. The .read method to hArray is slightly more relaxed here (see below).

Alternatively, you may want to only read the data into a section of your array:

```
>>> x[1].read(datafile,"Fx",3)
>>> x.mprint()
[29,29]
[-45,-74]
[29,29]
[35,27]
[29,29]
```

Now, we sneaked the 4th block (block 3) into the second row (index 1).

Finally, you can also read the data into an array of different type

```
(but that is slightly slower as scratch arrays are created for every
row that is being read)
>>> c=hArray(complex,[5,2])
>>> c[...].read(datafile,"Fx",[0,1,2,3])
>>> c.mprint()
[(29,0),(29,0)]
[(35,0),(27,0)]
[(9,0),(-8,0)]
[(-45,0),(-74,0)]
[(29,0),(29,0)]
```

## 5. Basic Plotting

-----

In order to plot the data, we can use external packages. Two packages are being provided here: matplotlib and mathgl. The former is specifically designed for python and thus slightly easier to use interactively. Since version 0.99 it is supposed to be capable of 3D plots (at the time of writing we use version 0.98). Mathgl is faster and is therefore used in our GUI programming. Eventually we will make it available for interactive plotting as well.

### 5.1. Mathgl

-----

Here is a simple example on how to use mathgl code (here without a widget)

```
from mathgl import *

width=800
height=600
size=1024
gr=mglGraph(mglGraphPS,width,height)
y=mglData(size)
y.Modify("cos(2*pi*x)")
x=mglData(size)
x.Modify("x*1024");
ymax=y.Maximal()
ymin=y.Minimal()
gr.Clf()
gr.SetRanges(0,0.5,ymin,ymax)
gr.Axis("xy")
gr.Title("Test Plot x")
gr.Label("x","x-Axis",1)
gr.Label("y","y-Axis",1)
gr.Plot(x,y);
gr.WriteEPS("test-y.eps","Test Plot")
```

### 5.2. Matplotlib

-----

Now, in principle, we need to import matplotlib

```
>>> import matplotlib.pyplot as plt
```

however, that should have been done already for you when pycrtools was loaded.

Depending on the system you may have to use

```
>>> plt.show()
```

and an empty plot window should pop up somewhere (in the background?)  
!! On the Mac it seems to pop up automatically.

(NB: At least on a Mac the window likes to stubbornly hide behind other windows, so search your screen carefully if no window pops up.)

Depending on the system, the program could hang here and wait for input. In this case you need to kill the window and re-issue the command!!!

Now, we can use some of the plotting commands.

```
>>> plt.subplot(1,2,1)
>>> plt.title("Average Spectrum for Two Antennas")
>>> plt.semilogy(frequencies.vec(),avspectrum[0].vec())
>>> plt.semilogy(frequencies.vec(),avspectrum[1].vec())
>>> plt.ylabel(avspectrum.getKey("name")+ " ["+avspectrum.getUnit()+"]")
>>> plt.xlabel(frequencies.getKey("name")+ " ["+frequencies.getUnit()+"]")
```

To plot the time series of the entire data set, we first read in all samples from all antennas

```
>>> datafile["block","blocksize"]=(0,maxblocksize)
>>> timeall=datafile["Time"]
>>> fxall=datafile["Fx"]
```

and then we plot it

```
>>> plt.subplot(1,2,2)
>>> plt.title("Time Series of Antenna 0")
>>> plt.plot(timeall.vec(),fxall[0].vec())
>>> plt.ylabel("Electric Field [ADC counts]")
>>> plt.xlabel("Time [ $\mu$ s]")
```

```
- #####
- #### Figure 1 ####
- #####
```

So, for a linear plot use `.plot`, for a loglog plot use `.loglog` and for a log-linear plot use `.semilogx` or `.semilogy` ...

### 5.3. Plotting Using the hArray Plotting Method

-----

There is also a simpler way to make the kind of plots described above, using the built-in plot method of array.

```
>>> avspectrum.par.xvalues=frequencies
>>> avspectrum.par.title="Average Spectrum"
>>> avspectrum[0].plot(logplot="y")
```

```
- #####
- #### Figure 2 ####
- #####
```

This creates a semilog-plot with appropriate labels and units (if they were provided beforehand.)



You can either provide the parameters directly (precedence), or set the plotting parameters as attributes to the .par class of the array, e.g., "array.par.xvalues=x\_vector; array.plot()"

If the array is in looping mode, multiple curves are plotted in one windows. Hence,

```
>>> avspectrum.par.logplot="y"
>>> avspectrum[...].plot(legend=datafile.antennas)
```

```
- #####
- #### Figure 3 ####
- #####
```

will simply plot all spectra of all antennas (=highest array index) in the array.

The available parameters are:

Parameters:

xvalues: an array with corresponding x values, if "None" numbers from 0 to length of the array are used

xlabel: the x-axis label, if not specified, use the "name" keyword of the xvalues array - units will be added automatically

ylabel: the y-axis label, if not specified, use the "name" keyword of the array - units will be added automatically

xlim: tuple with minimum and maximum limits for the x-axis

ylim: tuple with minimum and maximum limits for the y-axis

title: a title for the plot

clf: if True (default) clear the screen beforehand (use False to compose plots with multiple lines from different arrays.

logplot: can be used to make loglog or semilog plots:

"x" ->semilog in x

"y" ->semilog in y

"xy"->loglog plot

## 6. CR Pipeline Modules

### 6.1. Quality Checking of Time Series Data

For an automatic pipeline it is essential to check whether the data is of good quality, or whether one needs to flag particular antennas. Here we demonstrate a simple, but effective way to do this.

The basic parameters to look at are the mean value of the time series (indicating potential DC offsets), the root-mean-square (RMS) deviation (related to the power received), and the number of peaks in the data (indicating potential RFI problems).

For cosmic ray data, we expect spikes and peaks to be in the middle of

a trace, so we will just look at the first or/and last quarter of a data set and set the block size accordingly.

```
>>> blocksize=min(filesize/4,maxblocksize)
```

We will then read this block of data into an appropriately sized data array.

```
>>> dataarray=datafile.set("blocksize",blocksize).set("block",3)["Voltage"]
```

The array now contains all the measured voltages of the selected antennas in the file.

First we calculate the mean over all samples for each antennas (and make use of the looping through the Ellipsis object).

```
>>> datamean = dataarray[...].mean()
    datamean => Vector(float, 1, fill=[6.59021])
```

Similarly we get the rms (where we spare the algorithm from recalculating the mean, by providing it as input - actually a list of means).

```
>>> datarms = dataarray[...].stddev(datamean)
    datarms => Vector(float, 1, fill=[135.734])
```

and finally we get the total number of peaks 5 sigma above the noise.

```
>>> datanpeaks = dataarray[...].countgreaterthanabs(datarms*5)
    datanpeaks => Vector(int, 1, fill=[0])
```

To see whether we have more peaks than expected, we first calculate the expected number of peaks for a Gaussian distribution and our blocksize, as well as the error on that number.

```
>>> npeaksexpected=funcGaussian(5,1,0)*blocksize
    npeaksexpected => 0.0490974252546
>>> npeakerror=sqrt(npeaksexpected)
    npeakerror => 0.221579388154
```

So, that we can get a normalized quantity

$$G = (N_{\text{peaks\_detected}} - N_{\text{peaks\_expected}}) / N_{\text{peaks\_error}}$$

which should be of order unity if we have roughly a Gaussian distribution. If it is much larger or less than unity we have more or less peaks than expected and the data is clearly not Gaussian noise.

We do the calculation of G using our STL vectors (even though speed it no of the essence here)

```
>>> dataNonGaussianity = Vector(float,nAntennas)
>>> dataNonGaussianity.sub(datanpeaks,npeaksexpected)
>>> dataNonGaussianity /= npeakerror
```

The next step is to make a nice table of the results and check whether these parameters are within the limits we have imposed (based on empirical studies of the data).

To ease the operation we combine all the data into one python array (using the zip function - zip, as in zipper).

```
>>> dataproperties=zip(selectedAntennas,datamean,datarms,datanpeaks,dataNonGaussianity)
```

which is a rather nasty collection of numbers. So, we print a nice table (restricting it to the first 5 antennas).

```
>>> for prop in dataproperties[0:5]: print "Antenna {0:3d}: mean={1: 6.2f}, rms={2:6.1f},
npeaks={3:5d}, spikyness={4: 7.2f}".format(*prop)
Antenna   0: mean=   6.59, rms= 135.7, npeaks=    0, spikyness=  -0.22
```

Clearly this is a spiky dataset, with only one antenna not being affected by too many peaks (which is in fact not the case for the first block of that dataset).

To check automatically whether all parameters are in the allowed range, we can use a little python helper function, using a python "dict" as database for allowed parameters.

```
>>> qualitycriteria={"mean":(-15,15),"rms":(5,15),"spikyness":(-3,3)}
>>> CheckParameterConformance(dataproperties[0],{"mean":1,"rms":2,"spikyness":4},qualitycriteria)
```

The first parameter is just the list of numbers of the mean, rms, etc. of one antenna we created above. The second parameter is a dict, describing which parameter to find at which position in the input list, and the third parameter is yet another dict specifying for each parameter the range of allowed upper and lower values. The result is a list of parameter names, where the antennas failed the test. The list is empty if the antenna passed it.

Finally, we do not want to do this manually all the time. So, a little python function is available, that does the quality checking for you and returns a list with failed antennas and their properties.

```
>>> badantennalist=CRQualityCheck(qualitycriteria,datafile=datafile,dataarray=dataarray,blocksize=blocksize,verbose=False)
Block=    0, Antenna   0: mean=   6.59, rms= 141.7, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    1, Antenna   0: mean=   6.63, rms= 138.8, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    2, Antenna   0: mean=   6.62, rms= 141.4, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    3, Antenna   0: mean=   6.59, rms= 135.7, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    4, Antenna   0: mean=   6.61, rms= 140.9, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    5, Antenna   0: mean=   6.60, rms= 137.5, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    6, Antenna   0: mean=   6.62, rms= 143.1, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    7, Antenna   0: mean=   6.62, rms= 141.6, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    8, Antenna   0: mean=   6.63, rms= 135.0, npeaks=    0, spikyness=  -0.22 ['rms']
Block=    9, Antenna   0: mean=   6.63, rms= 146.0, npeaks=    0, spikyness=  -0.22 ['rms']
Block=   10, Antenna   0: mean=   6.63, rms= 140.9, npeaks=    1, spikyness=   4.29 ['rms', 'spikyness']
Block=   11, Antenna   0: mean=   6.60, rms= 142.4, npeaks=    0, spikyness=  -0.22 ['rms']
Block=   12, Antenna   0: mean=   6.61, rms= 143.0, npeaks=    0, spikyness=  -0.22 ['rms']
```

```

rms']
Block=    13, Antenna    0: mean=   6.61, rms= 141.6, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    14, Antenna    0: mean=   6.59, rms= 143.3, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    46, Antenna    0: mean=   6.61, rms= 142.5, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    47, Antenna    0: mean=   6.60, rms= 136.7, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    48, Antenna    0: mean=   6.59, rms= 141.1, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    49, Antenna    0: mean=   6.60, rms= 138.7, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    50, Antenna    0: mean=   6.63, rms= 143.8, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    51, Antenna    0: mean=   6.59, rms= 139.5, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    52, Antenna    0: mean=   6.58, rms= 141.5, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    53, Antenna    0: mean=   6.64, rms= 145.5, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    54, Antenna    0: mean=   6.61, rms= 143.1, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    55, Antenna    0: mean=   6.60, rms= 145.5, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    56, Antenna    0: mean=   6.62, rms= 135.0, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    57, Antenna    0: mean=   6.60, rms= 146.0, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    58, Antenna    0: mean=   6.60, rms= 143.3, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    59, Antenna    0: mean=   6.60, rms= 135.8, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    60, Antenna    0: mean=   6.59, rms= 141.6, npeaks=    0, spikyness= -0.22 ['
rms']
Block=    61, Antenna    0: mean=   6.64, rms= 135.6, npeaks=    0, spikyness= -0.22 ['
rms']

```

```

badantennalist[0] => [0, 0, (6.5949006782945734, 141.70452128542746, 0, -0.22157938815
373929), ['rms']]

```

(first the antenna number, then the block, then a list with the mean, rms, npeaks, and spikyness, and finally the failed fields)

Note, that this function can be called with "file=None". In this case the data provided in the datararray will be used.

## 6.2. Finding peaks in a vector

```

-----

```

First we make a test time series data set for 4 antennas.

```

>>> data=hArray(float,[4,512],name='Random series with peaks')
and fill it with random data that have arbitrary offsets.
>>> data.random(-1024,1024); data[...] += Vector([-128.,256., 385.,-50.])
The we put some peaks at location 2-3, 32, and 64-67 in each of the 4 data sets.
>>> for i in range(4): data[i,[2,3,32,64,65,67],...]=Vector([4096.,5097,-4096,4096,5099,30
96])
Now, we reverse-engineer and try finding all 5 sigma peaks
>>> nsigma=5

```

First make a scratch array that will contain the locations of the peaks. A location is actually a 'slice' in the array, i.e. given by its beginning and ending position (plus one). The length of the return array must be pre-allocated and should be long enough to contain all peaks (at maximum as long as the input array).

```
>>> datapeaks=hArray(int,[4,5,2],name="Location of peaks")
```

Now, retrieve the mean and rms of the array to set the thresholds above and below which one considers a peak to be significant.

```
>>> datamean=data[...].mean()
>>> datathreshold2 = data[...].stddev(datamean)
>>> datathreshold2 *= nsigma
>>> datathreshold1 = datathreshold2*(-1)
>>> datathreshold1 += datamean
>>> datathreshold2 += datamean
```

Finally, determine input parameters for the search algorithm.

```
>>> maxgap=Vector(int,len(datamean),fill=10)
```

The gap vector tells the algorithm how many samples can be between two values that are above threshold, so that the two peaks are considered as one.

```
>>> minlength=Vector(int,len(datamean),fill=1)
```

A minimum length can be specified to exclude peaks that consists of only a single or a few values (no relevant here, so set to 1, i.e. all peaks are relevant). Then call

hFindSequenceOutside (or Between, GreatererThan, LessEqual ...)

```
>>> npeaks=datapeaks[...].findsequenceoutside(data[...],datathreshold1,datathreshold2,maxgap,minlength)
```

The return value is the number of pekas found (in each row of the data set).

```
npeaks => Vector(int, 4, fill=[3,3,3,3])
```

An the slices are actually contained in the return vector.

```
datapeaks.mprint() =>[2,4,32,33,64,66,0,0,0,0]
[2,4,32,33,64,66,0,0,0,0]
[2,4,32,33,64,66,0,0,0,0]
[2,4,32,33,64,66,0,0,0,0]
None
```

### 6.3. Fourier Transforms (FFT) & Cross Correlation

-----

We can make a FFT of a float vector. This function will only return the non-redundant part of the FFT (i.e., just one half). Again we need to provide a properly sized output vector (input length/2+1). We also have to specify as a second parameter in which NyquistZone the data was take.

Nyquist sampling means that one needs, for example, 200 MHz sampling rate to digitize a bandwidth of 100 MHz. The first Nyquist zone is then 0-100 MHz, and the second is 100-200 MHz.

!!!TO BE EDITED: use hFFTw and hNyquistSwap instead for better performance!

So, let us do the transform:

```
>>> fftdata=hArray(complex,[nofSelectedAntennas,fftlength],name="FFT(E)")
>>> fftdata[...].fftcasa(fxdata[...],1)
```

```
fftdata => hArray(complex, [16, 1025], fill=[(14776,0),(-78.062,23.5841),(49.6801,19.8
583),(-69.624,15.9113),(2.28131,-46.0593),(-50.7135,-6.38084),(-11.8104,4.54312),(-52.0847
,52.0362),(72.7594,-54.191),(-30.9354,10.1987),(53.6353,14.9717),(-19.9353,-19.4781),(60.1
922,57.2602),(-98.8014,-51.6593),(146.007,-0.801246),(-50.0402,-27.6258),(-6.15142,2.74722
),(-12.7369,-11.7984),(37.913,56.9744),(-56.9674,-37.3285),(104.01,2.26267),(-64.2157,-78.
```

```

7553), (83.7601,19.6167), (-22.373,10.4778), (43.5451,46.3542), (-60.1809,-48.3899), (80.6444,2
7.6344), (-38.8504,-93.5034), (63.0164,43.7457), (-132.004,-6.32882), (-5.88212,26.3255), (-46.
4038,-4.76213), (15.5905,28.3133), (-29.0051,-58.3956), (127.06,-17.7467), (-56.7613,-43.1064)
, (56.9676,62.3398), (-66.3865,-64.5693), (64.2745,33.29), (-71.1614,16.4923), (14.0896,29.7471
), (-70.9475,-53.3049), (83.5799,57.7581), (-22.2036,-99.6002), (103.718,71.0071), (-81.3453,-4
6.9182), (64.9148,63.1255), (-59.3956,4.42413), (115.647,77.7064), (-81.3578,-73.5854), ..., (44
393.8,-27299.2), (10393.4,46372.6), (-24276.1,10858), (-16920.7,-6167.39), (5551.41,-20743.8),
(17997.3,8216.16), (125.361,-5443.34), (40841,11864.2), (-13171.1,42994.6), (4413.01,-9250.23)
, (14417.3,46503.3), (-33709.4,22021.5), (-11695.7,-1250.47), (8443.05,13106.3), (-23568.2,3826
6.9), (-34727.2,-12451.5), (10221.4,-6791.08), (10090.7,34488), (-53322.7,43903.4), (-52617.5,-
33163.7), (-3433.44,-24837.2), (3738.3,-9032.13), (-14084.1,1933.24), (14374,-22580.5), (10096.
1,48289.3), (-59866.2,19760.7), (-64796.1,-41719.7), (31059.6,-93474.6), (64273.1,34768.7), (-1
8098.2,9779.51), (12377.9,17216.9), (-22820.3,-1466.51), (15847.4,5897.68), (-10508.9,20181.6)
, (-29559.1,28367), (-49279.3,-36090.3), (16348.6,-12199), (-32706.8,-16562), (19230.3,-28269.5
), (-2768.73,-15831.5), (43800.2,-10707.1), (3079.42,32421.3), (-2979.94,11154.5), (-18775.3,97
02.5), (-5757.83,9789.31), (-56922.1,-3858.53), (6154.19,-66381.7), (20462.7,3887.06), (280.294
,-40752.5), (67594,0)], name="FFT(E)") # len=16400 slice=[0:16400])

```

Here we have used the fft method of the float array, which is just a call to the stand-alone function hFFT defined in hftools.cc.

We can convert the data back into the time domain by using the inverse Fourier transform.

```

>>> fxinvdata=hArray(float,dimensions=fxdata,name="E-Field").setUnit("", "Counts")
>>> fxinvdata[...].invfftcasa(fftdata[...],1)

```

To get the spectral power from the FFTed vector, we have to square the complex data and convert it to floats. This can be done using the complex function "norm" (unusual name, but that is what is used in c++).

```

>>> spectrum=hArray(float,dimensions=fftdata,name="E-field Spectrum")
>>> spectrum[...].norm(fftdata[...])

```

Finally, we want to see, if we can do some cross correlation, which mathematically is directly related to the Fourier transform.

We will do this with a data set from a solar burst (taken with LOPES in 2003), since that is dominated by a single point source and hence gives a nice and clear cross correlation.

```

>>> sun=crfile(filename_sun)
Opening LOPES File=/Users/falcke/LOFAR/usg/data/lopes/example.event
[DataReader::summary]
-- blocksize ..... = 65536
-- FFT length ..... = 32769
-- file streams connected? 0
-- shape(adc2voltage) ... = [65536, 8]
-- shape(fft2calfft) .... = [32769, 8]
-- nof. antennas ..... = 8
-- nof. selected antennas = 8
-- nof. selected channels = 32769
-- shape(fx) ..... = [65536, 8]
-- shape(voltage) ..... = [65536, 8]
-- shape(fft) ..... = [32769, 8]
-- shape(calfft) ..... = [32769, 8]

```

Now we read in the raw time series data as well as the Frequencies and Time value arrays.

```
>>> sun_time=sun["Time"].setUnit("\\mu","s")
>>> sun_frequencies=sun["Frequency"]
>>> sun_efield=sun["Fx"].setPar("xvalues",sun_time)
```

As a next step we create an empty vector to hold the Fourier spectrum of the data

```
>>> sun_fft=sun["emptyFFT"].setPar("xvalues",sun_frequencies)
```

and then make the Fourier transform (noting that the data is in the second Nyquist domain)

```
>>> sun_fft[...].fftcasa(sun_efield[...],2)
```

We will now try to make a crosscorrelation of the data. Let us start by making a complex scratch vector to hold an intermediate data product which is a copy of the FFT vector.

```
>>> crosscorr_cmplx=hArray(copy=sun_fft)
```

We then multiply the FFTs of all antennas with the complex conjugate of a reference antenna (here antenna 0) and store that in the vector crosscorr\_cmplx, which is done by the following method.

```
>>> crosscorr_cmplx[...].crosscorrelatecomplex(sun_fft[0])
```

This vector will then actually hold the FFT of the cross correlation of the antenna time series. Hence, what we now need to do is to FFT back into the time domain and store it in a vector crosscorr.

```
>>> sun_timelags=sun["TimeLag"].setUnit("\\mu","")
>>> crosscorr=hArray(float,dimensions=sun_efield,name="Cross Correlation",xvalues=sun_time
lags)
>>> crosscorr[...].invfftcasa(crosscorr_cmplx[...],2)
```

We can now plot this and use as the x-axis vector the Time Lags conveniently provided by our cr file object.

```
>>> crosscorr.par.xlim=(-1,1)
>>> crosscorr[1].plot()
```

```
- #####
- #### Figure 4 ####
- #####
```

Note, plotting crosscorr[0].plot() will give the autocorrelation of the first antenna (which we used as reference antenna in this example).

#### 6.4. Coordinates

-----

We also have access to a few functions dealing with astronomical coordinates. Assume, we have a source at an Azimuth/Elevation position of (178 Degree,28 Degree) and we want to convert that into Cartesian coordinates (which, for example, is required by our beamformer).

We first turn this into a STD vector and create a vector that is supposed to hold the Cartesian coordinates. Note that the AzEl vector is actually AzElRadius, where we need to set the radius to unity.

```
>>> azel=hArray([178.,28,1],dimensions=[1,3])
>>> cartesian=azel.new()
```

We then do the conversion, using

```
>>> hCoordinateConvert(azel[...],CoordinateTypes.AzElRadius,cartesian[...],CoordinateTypes
.Cartesian,True)
```

yielding the following output vector:

```
cartesian => hArray(float, [1, 3], fill=[0.0308144,-0.88241,0.469472]) # len=3 slice=[
0:3])
```

## 6.5. Coordinates

-----

For doing actual beamforming we need to know the antenna positions. For this we will use the method getCalData:

```
>>> antenna_positions=sun.getCalData("Position")
antenna_positions => hArray(float, [8, 3], fill=[-84.5346,36.1096,0,-52.6146,54.4736,-
0.0619965,-34.3396,22.5366,-0.131996,-2.3706,40.6976,-0.00299835,41.0804,1.97557,-0.076995
8,22.7764,34.1686,-0.0549927,-20.8546,72.5436,-0.154999,11.1824,90.8196,-0.221992]) # len=
24 slice=[0:24])
```

As a next step we need to put the antenna coordinates on a reference frame that is relative to the phase center. Here we will choose the location of the first antenna as our phase center (that makes life a little easier for checking) and we simply subtract the reference position from the antenna locations so that our phase center lies at (0,0,0).

```
>>> phase_center=hArray(antenna_positions[0].vec())
>>> antenna_positions -= phase_center
```

Now we read in instrumental delays for each antenna in a similar way and store it for later use.

```
>>> cal_delays=sun.getCalData("Delay")
cal_delays => hArray(float, [8], fill=[0,2.3875e-09,-1.3625e-09,-2.8925e-09,-2.4375e-0
8,-2.50415e-08,1.42475e-08,9.29625e-09]) # len=8 slice=[0:8])
```

We now convert the Azimuth-Elevation position into a vector in Cartesian coordinates, which is what is used by the beamformer.

```
>>> hCoordinateConvert(azel,CoordinateTypes.AzElRadius,cartesian,CoordinateTypes.Cartesian
,True)
```

Then calculate geometric delays and add the instrumental delays.

```
>>> delays=hArray(float,dimensions=cal_delays)
>>> hGeometricDelays(delays,antenna_positions,cartesian,True)
```

To get the total delay we add the geometric and the calibration delays.

```
>>> delays += cal_delays
```

The delays can be converted to phases of complex weights (to be applied in the Fourier domain).



```
>>> phases=hArray(float,dimensions=sun_fft,name="Phases",xvalues=sun_frequencies)
>>> phases.delaytophase(sun_frequencies,delays)
```

Similarly, the corresponding complex weights are calculated.

```
>>> weights=hArray(complex,dimensions=sun_fft,name="Complex Weights")
>>> weights.phasetocomplex(phases)
```

To shift the time series data (or rather the FFTed time series data) we multiply the FFT data with the complex weights from above.

```
>>> sun_calfft_shifted=hArray(copy=sun_fft)
>>> sun_calfft_shifted *= weights
```

We can check whether the shifts were doing what we expected it to do by making another crosscorrelation (using the shifted FFT data as starting point).

```
>>> crosscorr_cmplx_shifted=hArray(copy=sun_calfft_shifted)
>>> crosscorr_cmplx_shifted[...].crosscorrelatecomplex(sun_calfft_shifted[0])
>>> crosscorr_shifted=hArray(float,dimensions=sun_efield,name="Cross Correlation",xvalues=
sun_timelags)
>>> crosscorr_shifted[...].invfftcasa(crosscorr_cmplx_shifted[...],2)
>>> crosscorr_shifted[...].plot(xlim=(-0.25,0.25),legend=sun.antennaIDs)
```

```
- #####
- #### Figure 5 ####
- #####
```

And indeed now most of the antennas show a lag peaking around 0, with the exception of two antennas (which likely experienced an additional two-sample shift).

So, let us transform back into time and see the final, shifted time series.

```
>>> sun_efield_shifted = sun["emptyFx"].setPar("xvalues",sun_time)
>>> sun_efield_shifted[...].invfftcasa(sun_calfft_shifted[...],2)
>>> sun_efield_shifted[0:3,...].plot(xlim=(-2,0),ylim=(-1500,1500),legend=sun.antennaIDs[0
:3])
```

```
- #####
- #### Figure 6 ####
- #####
```

Obviously the three time series are now tracing each other nicely, indicating that the delays were about right. To now "form a beam" we just need to add all time series data (or their FFTs),

```
>>> sun_calfft_shifted_added=hArray(sun_calfft_shifted[0].vec())
>>> sun_calfft_shifted[1:,...].addto(sun_calfft_shifted_added)
```

normalize by the number of antennas  
 >>> sun\_calfft\_shifted\_added /= sun.nofSelectedAntennas  
 and then FFT back into the time domain:

```
>>> sun_efield_shifted_added=hArray(float,dimensions=sun_time,name="beamformed E-field",xv
alues=sun_time)
>>> sun_efield_shifted_added.invfftcasa(sun_calfft_shifted_added,2)
```

```
>>> sun_efield[0].plot()
>>> sun_efield_shifted_added.plot(xlim=(-2,0),clf=False,legend=["Reference Antenna","Beam"
])
```

```
- #####
- #### Figure 7 ####
- #####
```

In the plot one can see how the green line (beamformed) traces the data in antenna one (which was our reference antenna).

7. Appendix: Listing of all Functions:  
=====

```
>>> help(all)
Help on built-in function all in module __builtin__:

all(...)
    all(iterable) -> bool

    Return True if bool(x) is True for all values x in the iterable.
```