pycrtools TUTORIAL

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

The library makes use of algotithms 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/Pypeline directory.

Table of Contents

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 definitons 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 .bashr
or .profile files, like

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

then you can simply start the pyrtools 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 UNIX prompt.

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
diretcly 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 exculsively 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 inputs AND output vectors. I.e.: you need to know beforehand what sized vector you expect in return!

That 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 creat a vector most efficently, use the original vector constructors:

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

e.g.

```
>>> v=FloatVec(); v
  Vec(float,0)=[]
```

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 => Vec(float,4)=[1.0,2.0,3.0,4.0]
```

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].

As the latest addition some simple support for multiple dimensions has
been implemented, using the methods:

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

This will be described later ....

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's 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 note 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
  Vec(float,6)=[0.0,1.0,2.0,3.0,4.0,5.0]
>>> v2=Vector(float,len(v1),2.0);v2
  Vec(float,6)=[2.0,2.0,2.0,2.0,2.0,2.0]
```

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
  Vec(float,8)=[2.0,2.0,2.0,2.0,2.0,2.0,0.0,0.0]
```

Resize a vector and fill with non-zero values:

```
>>> v2.resize(10,-1)
>>> v2
  Vec(float,10)=[2.0,2.0,2.0,2.0,2.0,2.0,0.0,0.0,-1.0,-1.0]
```

Resize a vector to same size as another vector:

```
>>> v2.resize(v1)
>>> v2
  Vec(float,6)=[2.0,2.0,2.0,2.0,2.0,2.0]
```

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

```
>>> v3=v2.new()
>>> v3
  Vec(float,6)=[0.0,0.0,0.0,0.0,0.0,0.0]
```

Fill a vector with values

```
>>> v3.fill(-2)
>>> v3
  Vec(float,6)=[-2.0,-2.0,-2.0,-2.0,-2.0,-2.0]
```

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(Vector(float))
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dict__', '__div_
_', '__doc__', '__format__', '__getattribute__', '__getitem__', '__hash__', '__iadd__', '_
_idiv__', '__imul__', '__init__', '__instance_size__', '__isub__', '__iter__', '__len__',
'__module__', '__mul__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr_
_', '__setitem__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__weakref__',
'abs', 'acos', 'add', 'addadd', 'addaddconv', 'append', 'asin', 'atan', 'ceil', 'convert',
 'copy', 'cos', 'cosh', 'div', 'divadd', 'divaddconv', 'downsample', 'exp', 'extend', 'ext
endflat', 'fft', 'fill', 'findgreaterequal', 'findgreaterequalabs', 'findgreaterthan', 'fi
ndgreaterthanabs', 'findlessequal', 'findlessequalabs', 'findlessthan', 'findlessthanabs',
 'findlowerbound', 'floor', 'iadd', 'idiv', 'imul', 'isub', 'log', 'log10', 'mean', 'media
n', 'mul', 'muladd', 'muladdconv', 'new', 'norm', 'normalize', 'resize', 'sin', 'sinh', 's
ort', 'sortmedian', 'sqrt', 'square', 'stddev', 'sub', 'subadd', 'subaddconv', 'sum', 'tan
', 'tanh']
```

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

You can add a scalar to a vector by

```
>>> v1+3
  Vec(6)=[3.0,4.0,5.0,6.0,7.0,8.0]
```

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

You can also add two vectors:

```
>>>v1+v2
  Vec(6)=[2.0,3.0,4.0,5.0,6.0,7.0]
```

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

Adding a vector in place:

```
>>>v1+=v2;
>>>v1
  Vec(6)=[2.0,3.0,4.0,5.0,6.0,7.0]
```

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() => 2.5
```

Median:

```
    v1.median() => 3.0
```

Summing all elements in a vector:

```
    v1.sum() => 15.0
```

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, who's 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(10))
```

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

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

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()
  Vec(int,9)=[0,1,2,3,4,5,6,7,8]
```

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]=9, [0:9]) -> [0,2,4,6,8,10,12,14,16]

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

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()
>>> a.mul(2,tmp_array)

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

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

An important constraint is that all these functions or methods only work with either vector or array clases, a mix in the paramters 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()
  [3, 3]

>>> a.setDim([3,3,2])
  hArray(int, [3, 3, 2]=18, [0:18]) -> [0,1,2,3,4,5,6,7,8,0,...]

>>> a.setDim([3,3])
  hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
```

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 => Vec(int,9)=[-1,1,2,3,4,5,6,7,8]
    a => hArray(int, [3, 3]=9, [0:9]) -> [-1,1,2,3,4,5,6,7,8]
```

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]=9, [0:9]) -> [-2,1,2,3,4,5,6,7,8]
    a => hArray(int, [3, 3]=9, [0:9]) -> [-2,1,2,3,4,5,6,7,8]
    v => Vec(int,9)=[-2,1,2,3,4,5,6,7,8]

>>> v[0]=0

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

To actually make a pyhsically 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 initates 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]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
```

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]=9, [0:3]) -> [0,1,2]
```

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()
  Vec(int,3)=[0,1,2]
```

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.

Similarly,

```
>>> a[0,1].vec()
  Vec(int,1)=[1]
```

will access a slice consisting of a single element (returned as a vector). To obtain it as a scalar value, use

```
>>> a[0,1].val()
  1
```

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()
  [0, 1]
```

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

```
>>> a[0:2]
  hArray(int, [3, 3]=9, [0:6]) -> [0,1,2,3,4,5]
```

is possible, but not

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

where the first slice is simply ignored.

3.3.5. Applying Methods to Slices

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

```
>>> a[0].sum()
  Vec(int,1)=[3]
```

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]=9, [0:9]) -> [0,-1,-2,3,4,5,6,7,8]
```

```
>>> a[0].negate()
    a => hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
```

returns nothing, but will actually change ths 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, 1, 2]  => a = hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
  Row 1 : [3, 4, 5]  => a = hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
  Row 2 : [6, 7, 8]  => a = hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
```

However, looping over slices in simple way is aready 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]=9, [0:3]*) -> [0,1,2]
```

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.

```
>>> while l.isLooping():
...     print "Row",l.loop_nslice(),":",l.val(), " => l =", l
...     l.next()
  Row 0 : [0, 1, 2]  => l = hArray(int, [3, 3]=9, [0:3]*) -> [0,1,2]
  hArray(int, [3, 3]=9, [3:6]*) -> [3,4,5]
  Row 1 : [3, 4, 5]  => l = hArray(int, [3, 3]=9, [3:6]*) -> [3,4,5]
  hArray(int, [3, 3]=9, [6:9]*) -> [6,7,8]
  Row 2 : [6, 7, 8]  => l = hArray(int, [3, 3]=9, [6:9]*) -> [6,7,8]
  hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]

    l => hArray(int, [3, 3]=9, [0:9]) -> [0,1,2,3,4,5,6,7,8]
```

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

Here isLooping() 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 isLooping is reset to false).

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's calculate the mean value of each slice at the to level of our example array.

First, we reset the loop in l (which still references to a) to its starting values.

```
>>> l.resetLoop()
  hArray(int, [3, 3]=9, [0:3]*) -> [0,1,2]
```

and then calculate the mean

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

```
  Vec(float,3)=[1.0,4.0,7.0]
```

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()
  Vec(float,2)=[4.0,7.0]
```

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

```
>>> a[:2,...].mean()
  Vec(float,2)=[1.0,4.0]
```

will loop over the first two top-level slices.

```
>>> a[0:3:2,...].mean()
  Vec(float,2)=[1.0,7.0]
```

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()
  Vec(float,3)=[1.0,4.0,7.0]
```

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

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

```
>>> a[[0,2],...].mean()
  Vec(float,2)=[1.0,7.0]
```

will loop over slices 0 and 2.

Looping can also be done for methods that require multiple arrays as inputs (remember a mix of vectors and arrays is forbidden). In this case the next() method will be applied to every array in the paramter 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

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

```
    x => hArray(int, [3, 3]=9, [0:9]) -> [0,2,4,0,0,0,12,14,16]
```

and indeed now thw 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
mutliplication of the last slice in a.

```
>>> x=hArray(int,a); a[...].mul(2,x)

    x => hArray(int, [3, 3]=9, [0:9]) -> [12,14,16,0,0,0,0,0,0]
```

4. File I/O

4.1. Opening and Closing a CR Data File

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

```
>>> LOFARSOFT=os.environ["LOFARSOFT"]
>>> filename=LOFARSOFT+"/data/lofar/trigger-2010-02-11/triggered-pulse-2010-02-11-TBB1.h5"

>>> filename
  '/Users/falcke/LOFAR/usg/data/lofar/trigger-2010-02-11/triggered-pulse-2010-02-11-TBB1.h
5'
```

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:

```
>>> file=crfile(filename)
  -- nof dipole datasets = 16
  -- Setting up DataIterator objects ...
  -- Setting up header record ...
  Opening LOFAR File=/Users/falcke/LOFAR/usg/data/lofar/trigger-2010-02-11/triggered-pulse
-2010-02-11-TBB1.h5
  [DataReader::summary]
   -- blocksize ............ = 1024
   -- FFT length ........... = 513
   -- file streams connected?  0
   -- shape(adc2voltage) ... = [1024, 16]
   -- shape(fft2calfft) .... = [513, 16]
   -- nof. antennas ........ = 16
   -- nof. selected antennas = 16
   -- nof. selected channels = 513
   -- shape(fx) ............ = [1024, 16]
   -- shape(voltage) ....... = [1024, 16]
   -- shape(fft) ........... = [513, 16]
   -- shape(calfft) ........ = [513, 16]
>>> file.set("Blocksize",1024*2)
```

The associated filename can be retrieved with

```
>>> file.filename
  '/Users/falcke/LOFAR/usg/data/lofar/trigger-2010-02-11/triggered-pulse-2010-02-11-TBB1.h
5'
```

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 done using a
single method "get". This method actually calls the function
"hFileGetParameter" defined in the c++ code.

Which observatory did we actually use?

```
>>> obsname=file.get("Observatory");
    obsname => LOFAR
```

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

```
>>> keywords=file.get("help")
  hFileGetParameter - available keywords: nofAntennas, nofSelectedChannels, nofSelectedAnt
ennas, nofBaselines, block, blocksize, stride, fftLength, nyquistZone, sampleInterval, ref
erenceTime, sampleFrequency, antennas, selectedAntennas, selectedChannels, positions, incr
ement, frequencyValues, frequencyRange, Date, Observatory, Filesize, dDate, presync, TL, L
TL, EventClass, SampleFreq, StartSample, AntennaIDs, help
```

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

```
>>> file.get("frequencyRange")
  Vec(float,2)=[0.0,100000000.0]
```

actually returns a vector.

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

Now we will define a number of useful variables that contain essential
parameters that we later will use.

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

    obsdate => 1265926154
    filesize => 132096
    blocksize => 2048
    nAntennas => 16
    antennas => Vec(int,16)=[0,1,2,3,4,5,6,7,8,9,...]
```

```
    antennaIDs => Vec(int,16)=[128002016,128002017,128002018,128002019,128002020,128002021
,128002022,128002023,128003024,128003025,...]
    selectedAntennas => Vec(int,16)=[0,1,2,3,4,5,6,7,8,9,...]
    nofSelectedAntennas => 16
    fftlength => 1025
    sampleFrequency => 200000000.0
    maxblocksize => 132096
    nBlocks => 64
```

We can also change parameters in a very similar fashion, using the "set" method, which is an implementation of the "hFileSetParameter" function. E.g. changing the blocksize we already did before. This is simply

```
>>> file.set("Blocksize",blocksize);
```

again the list of implemented keywords is visible with using

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

The set method actually returns the crfile object. Hence you can append multiple set commands after each other.

```
>>> file.set("Block",2).set("SelectedAntennas",[0,2]);
```

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

```
>>>file.get("nofSelectedAntennas")
 2
```

However, now we want to work on all antennas again:

```
>>> file.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 mutliple 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 programme carefully and to understand the data structrue.

First we create a FloatVec, which is BoostPython wrapped standard (STL) vector of doubles.

```
>>> fxdata=Vector()
```

and resize it to the size we need

```
>>> fxdata.setDim([nofSelectedAntennas,blocksize])
  Vec(float,32768)=[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...]
```

```
This is now a large vector filled with zeros.

>>> fxdata
Vec(2048)=[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...]

Now we can read in the raw time series data, using "file.read" and a
keyword. Currently implemented are the data fields "Fx", "Voltage",
"FFT", "CalFFT","Time", "Frequency".

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) ).

>>> file.read("Fx",fxdata)
  <hftools.DataReader object at 0x772d0>

and voila the vector is filled with time series data from the data file..

Access the various antennas through slicing

>>> ant0data=fxdata[0:blocksize]
or use the .elem method, which returns the nth element of the highest dimension

>>> ant0data=fxdata.elem(0)
>>> ant1data=fxdata.elem(1)
>>> ant0data
  Vec(float,2048)=[16.0,17.0,10.0,18.0,17.0,6.0,16.0,17.0,-1.0,0.0,...]

This makes a copy of the data vector if used in this way, while

>>> fxdata[0:3]=[0,1,2]
>>> fxdata
  Vec(float,32768)=[0.0,1.0,2.0,18.0,17.0,6.0,16.0,17.0,-1.0,0.0,...]

actually modifies the original data vector.

To get the x -Axis we create a second vector

>>> timedata=Vector(float,blocksize)
>>> file.read("Time",timedata)
  <hftools.DataReader object at 0x77228>

This is the time relative to the trigger in seconds. So, let's have
that in microseconds, by multiplying with one million.

>>> timedata *= 10**6
>>> timedata
  Vec(float,2048)=[0.0,0.005,0.01,0.015,0.02,0.025,0.03,0.035,0.04,0.045,...]

We do the same now for the frequency axis, which we convert to MHz. As
length we have to take the length of the Fourier transformed time
block (which is blocksize/2+1).

>>> freqdata=Vector(float,fftlength)
>>> file.read("Frequency",freqdata)
  <hftools.DataReader object at 0x77228>
>>> freqdata/=10**6
>>> freqdata
  Vec(float,1025)=[0.0,0.09765625,0.1953125,0.29296875,0.390625,0.48828125,0.5859375,0.683
59375,0.78125,0.87890625,...]
```

5. Fourier Transforms (FFT)

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.

So, let's do the transform:

```
>>> fftdata=Vector(complex,fftlength)
>>> fxdata.elem(0).fft(fftdata,1)
>>> fftdata
  Vec(complex,1025)=[(14736+0j),(-38.0622213601+23.4859171199j),(9.68095724748+20.05462266
52j),(-29.6260665947+15.6167985077j),(-37.7150725681-45.6666687196j),(-10.7191165725-6.871
66502804j),(-51.8022613662+5.13208605582j),(-12.0958025896+51.3491571044j),(32.7738487563-
53.4058296902j),(9.04631849557+9.31539046418j),...]
```

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

to get the power, we have to square the complex data and convert it to floats. This can be done using the complex function "norm"

```
>>> spectrum=Vector(float,fftlength)
>>> fftdata.norm(spectrum)
```

We can now try to calulcate the average spectum of the data set for one antenna, by looping over all blocks.

```
>>> avspectrum=Vector(float)
>>> avspectrum.setDim([nofSelectedAntennas,fftlength])
  Vec(float,16400)=[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...]
>>> fftall=Vector(complex)
>>> fftall.setDim([nofSelectedAntennas,fftlength])
  Vec(complex,16400)=[0j,0j,0j,0j,0j,0j,0j,0j,0j,0j,...]
>>> for block in range(nBlocks):
...     file.set("Block",block).read("FFT",fftall)
...     fftall.spectralpower(avspectrum)
```

6. 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 bea capable of 3D plots (at the time of writing we use 0.98). Mathgl is perhaps a little faster and convenient for real time applications and hence is used in our GUI programming.

6.1. 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")
```

6.2. Matplotlib

Now we import matplotlib

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

And a plot window should pop up somehwere (in the background?) !!"
(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.)

Now, we can issue some of the plotting commands.

```
>>> plt.show()
>>> plt.subplot(1,2,1)
>>> plt.title("Average Spectrum for Two Antennas")
>>> plt.semilogy(freqdata,avspectrum.elem(0))
>>> plt.semilogy(freqdata,avspectrum.elem(1))
>>> plt.ylabel("Power of Electric Field [ADC counts]$^2$")
>>> plt.xlabel("Frequency [MHz]")
```

To plot the time series of the entire data set, we first read in all sample from all anten
nas

```
>>> file.set("Block",0).set("Blocksize",maxblocksize)
>>> fxall=Vector(); fxall.setDim([nofSelectedAntennas,maxblocksize])
  Vec(float,2113536)=[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,...]
>>> timeall=Vector(float,maxblocksize)
>>> file.read("Time",timeall); timeall *= 10**6.
  <hftools.DataReader object at 0x3d9dc0>
>>> file.read("Fx",fxall)
  <hftools.DataReader object at 0x3d7ca8>
```

and then we plot it

```
>>> plt.subplot(1,2,2)
>>> plt.title("Time Series of Antenna 0")
>>> plt.plot(timeall,fxall.elem(0))
```

```
>>> plt.ylabel("Electric Field [ADC counts]")
>>> plt.xlabel("Time [$\mu$s]")
```

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

7. CR Pipeline Modules

```
qualitycriteria=[("mean",(-15,15)),("rms",(5,15)),("nonGaussianity",(-3,3))]
qualityresults=CRQualityCheck(file,qualitycriteria)
```

8. Coordinates

We also have access to a few functions dealing with astronomical
coordinates. Assume, we have a source at an Azmiut/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 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=FloatVec()
>>> azel.extend((178,28,1))
>>> cartesian=azel.new()
>>> azel
  Vec(float,3)=[178.0,28.0,1.0]
>>> cartesian
  Vec(float,3)=[0.0,0.0,0.0]
```

We then do the conversion, using

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

yielding the following output vector:

```
>>> cartesian
  Vec(float,3)=[0.0308144266055,-0.882409725042,0.469471562786]
```

9. Appendix: Listing of all Functions: