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

Version History:
 -    started March 1, 2010 by H. Falcke

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(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(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]
```

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

Fill a vector with values

```
>>> v3.fill(-2)
>>> v3
  Vec(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 is a list of other functions you 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
```

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(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(16)=[0,1,2,3,4,5,6,7,8,9,...]
  antennaIDs = Vec(16)=[128002016,128002017,128002018,128002019,128002020,128002021,128002
022,128002023,128003024,128003025,...]
  selectedAntennas = Vec(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])
```

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 0x738f0>
```

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(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(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 0x73260>
```

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(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 0x73260>
>>> freqdata/=10**6
>>> freqdata
  Vec(1025)=[0.0,0.09765625,0.1953125,0.29296875,0.390625,0.48828125,0.5859375,0.68359375,
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(1025)=[(14736+0j),(-38.0622213601+23.4859171199j),(9.68095724748+20.0546226652j),(-2
9.6260665947+15.6167985077j),(-37.7150725681-45.6666687196j),(-10.7191165725-6.87166502804
j),(-51.8022613662+5.13208605582j),(-12.0958025896+51.3491571044j),(32.7738487563-53.40582
96902j),(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])
>>> fftall=Vector(complex)
>>> fftall.setDim([nofSelectedAntennas,fftlength])
>>> 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])
>>> timeall=Vector(float,maxblocksize)
>>> file.read("Time",timeall); timeall *= 10**6.
  <hftools.DataReader object at 0x3ee35e0>
>>> file.read("Fx",fxall)
  <hftools.DataReader object at 0x73928>
```

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. 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(3)=[178.0,28.0,1.0]
>>> cartesian
  Vec(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(3)=[0.0308144266055,-0.882409725042,0.469471562786]
```

8. Appendix: Listing of all Functions:

```
>>> help(all)
  Class:  <type 'function'>

  SECTION: Administrative Vector Function
  -------------------------------------------------
  hFill(vec, fill_value) - Fills a vector with a constant value.

  hNew(vec)              - Make and return a new vector of the same size and
                           type as the input vector.

  hConvert(vec1, vec2)   - Copies and converts a vector to a vector of another
                           type.

  hCopy(vec, outvec)     - Copies a vector to another one.


  SECTION: Math Function
  -------------------------------------------------
  square(val)            - Returns the squared value of the parameter.

  hPhase(frequency, time) - Returns the interferometer phase in radians for a
                           given frequency and time.

  funcGaussian(x, sigma, mu) - Implementation of the Gauss function.

  hExp(vec)              - Take the exp of all the elements in the vector.

  hExp(vec, vecout)      - Take the exp of all the elements in the vector and
                           return results in a second vector.

  hLog(vec)              - Take the log of all the elements in the vector.

  hLog(vec, vecout)      - Take the log of all the elements in the vector and
                           return results in a second vector.

  hLog10(vec)            - Take the log10 of all the elements in the vector.

  hLog10(vec, vecout)    - Take the log10 of all the elements in the vector and
                           return results in a second vector.

  hSin(vec)              - Take the sin of all the elements in the vector.

  hSin(vec, vecout)      - Take the sin of all the elements in the vector and
                           return results in a second vector.

  hSinh(vec)             - Take the sinh of all the elements in the vector.

  hSinh(vec, vecout)     - Take the sinh of all the elements in the vector and
```

```
                            return results in a second vector.

hSqrt(vec)              - Take the sqrt of all the elements in the vector.

hSqrt(vec, vecout)      - Take the sqrt of all the elements in the vector and
                          return results in a second vector.

hSquare(vec)            - Take the square of all the elements in the vector.

hSquare(vec, vecout)    - Take the square of all the elements in the vector
                          and return results in a second vector.

hTan(vec)               - Take the tan of all the elements in the vector.

hTan(vec, vecout)       - Take the tan of all the elements in the vector and
                          return results in a second vector.

hTanh(vec)              - Take the tanh of all the elements in the vector.

hTanh(vec, vecout)      - Take the tanh of all the elements in the vector and
                          return results in a second vector.

hAbs(vec)               - Take the abs of all the elements in the vector.

hAbs(vec, vecout)       - Take the abs of all the elements in the vector and
                          return results in a second vector.

hCos(vec)               - Take the cos of all the elements in the vector.

hCos(vec, vecout)       - Take the cos of all the elements in the vector and
                          return results in a second vector.

hCosh(vec)              - Take the cosh of all the elements in the vector.

hCosh(vec, vecout)      - Take the cosh of all the elements in the vector and
                          return results in a second vector.

hCeil(vec)              - Take the ceil of all the elements in the vector.

hCeil(vec, vecout)      - Take the ceil of all the elements in the vector and
                          return results in a second vector.

hFloor(vec)             - Take the floor of all the elements in the vector.

hFloor(vec, vecout)     - Take the floor of all the elements in the vector and
                          return results in a second vector.

hAcos(vec)              - Take the acos of all the elements in the vector.

hAcos(vec, vecout)      - Take the acos of all the elements in the vector and
                          return results in a second vector.

hAsin(vec)              - Take the asin of all the elements in the vector.

hAsin(vec, vecout)      - Take the asin of all the elements in the vector and
                          return results in a second vector.

hAtan(vec)              - Take the atan of all the elements in the vector.

hAtan(vec, vecout)      - Take the atan of all the elements in the vector and
```

```
                         return results in a second vector.

hiSub(vec1, vec2)      - Performs a Sub between the two vectors, which is
                         returned in the first vector. If the second vector
                         is shorter it will be applied multiple times.

hiSub(vec1, val)       - Performs a Sub between the vector and a scalar
                         (applied to each element), which is returned in the
                         first vector.

hSub(vec1, vec2, vec3) - Performs a Sub between the two vectors, which is
                         returned in the third vector.

hSubAdd(vec1, vec2, vec3) - Performs a Sub between the two vectors, and adds
                         the result to the output (third) vector.

hSubAddConv(vec1, vec2, vec3) - Performs a Sub between the two vectors, and
                         adds the result to the output (third) vector -
                         automatic casting is done.

hSub(vec1, val, vec2)  - Performs a Sub between the vector and a scalar,
                         where the result is returned in the second vector.

hiMul(vec1, vec2)      - Performs a Mul between the two vectors, which is
                         returned in the first vector. If the second vector
                         is shorter it will be applied multiple times.

hiMul(vec1, val)       - Performs a Mul between the vector and a scalar
                         (applied to each element), which is returned in the
                         first vector.

hMul(vec1, vec2, vec3) - Performs a Mul between the two vectors, which is
                         returned in the third vector.

hMulAdd(vec1, vec2, vec3) - Performs a Mul between the two vectors, and adds
                         the result to the output (third) vector.

hMulAddConv(vec1, vec2, vec3) - Performs a Mul between the two vectors, and
                         adds the result to the output (third) vector -
                         automatic casting is done.

hMul(vec1, val, vec2)  - Performs a Mul between the vector and a scalar,
                         where the result is returned in the second vector.

hiAdd(vec1, vec2)      - Performs a Add between the two vectors, which is
                         returned in the first vector. If the second vector
                         is shorter it will be applied multiple times.

hiAdd(vec1, val)       - Performs a Add between the vector and a scalar
                         (applied to each element), which is returned in the
                         first vector.

hAdd(vec1, vec2, vec3) - Performs a Add between the two vectors, which is
                         returned in the third vector.

hAddAdd(vec1, vec2, vec3) - Performs a Add between the two vectors, and adds
                         the result to the output (third) vector.

hAddAddConv(vec1, vec2, vec3) - Performs a Add between the two vectors, and
                         adds the result to the output (third) vector -
                         automatic casting is done.

hAdd(vec1, val, vec2)  - Performs a Add between the vector and a scalar,
                         where the result is returned in the second vector.

hiDiv(vec1, vec2)      - Performs a Div between the two vectors, which is
                         returned in the first vector. If the second vector
                         is shorter it will be applied multiple times.

hiDiv(vec1, val)       - Performs a Div between the vector and a scalar
                         (applied to each element), which is returned in the
                         first vector.

hDiv(vec1, vec2, vec3) - Performs a Div between the two vectors, which is
                         returned in the third vector.

hDivAdd(vec1, vec2, vec3) - Performs a Div between the two vectors, and adds
                         the result to the output (third) vector.

hDivAddConv(vec1, vec2, vec3) - Performs a Div between the two vectors, and
                         adds the result to the output (third) vector -
                         automatic casting is done.

hDiv(vec1, val, vec2)  - Performs a Div between the vector and a scalar,
                         where the result is returned in the second vector.

hConj(vec)             - Calculate the complex conjugate of all elements in
                         the complex vector.

hCrossCorrelateComplex(vec1, vec2) - Multiplies the elements of the first
                         vector with the complex conjugate of the elements in
                         the second and returns the results in the first.

hReal(vec, vecout)     - Take the real of all the elements in the complex
                         vector and return results in a float vector.

hArg(vec, vecout)      - Take the arg of all the elements in the complex
                         vector and return results in a float vector.

hImag(vec, vecout)     - Take the imag of all the elements in the complex
                         vector and return results in a float vector.

hNorm(vec, vecout)     - Take the norm of all the elements in the complex
                         vector and return results in a float vector.

hNegate(vec)           - Multiplies each element in the vector with -1 in
                         place, i.e. the input vector is also the output
                         vector.

hSum(vec)              - Performs a sum over the values in a vector and
                         returns the value.

hNorm(vec)             - Returns the lengths or norm of a vector (i.e.
                         Sqrt(Sum_i(xi*+2))).

hNormalize(vec)        - Normalizes a vector to length unity.

hMean(vec)             - Returns the mean value of all elements in a vector.

hSort(vec)             - Sorts a vector in place.
```

```
hSortMedian(vec)       - Sorts a vector in place and returns the median value
                         of the elements.

hMedian(vec)           - Returns the median value of the elements.

hStdDev(vec, mean)     - Calculates the standard deviation around a mean
                         value.

hStdDev(vec)           - Calculates the standard deviation of a vector of
                         values.

hFindLessEqual(vec, threshold, vecout) - Find the samples that are LessEqual
                         a certain threshold value and returns the number of
                         samples found and the positions of the samples in a
                         second vector.

hFindLessEqualAbs(vec, threshold, vecout) - Find the samples whose absolute
                         values are LessEqual a certain threshold value and
                         returns the number of samples found and the
                         positions of the samples in a second vector.

hFindGreaterThan(vec, threshold, vecout) - Find the samples that are
                         GreaterThan a certain threshold value and returns
                         the number of samples found and the positions of the
                         samples in a second vector.

hFindGreaterThanAbs(vec, threshold, vecout) - Find the samples whose absolute
                         values are GreaterThan a certain threshold value and
                         returns the number of samples found and the
                         positions of the samples in a second vector.

hFindGreaterEqual(vec, threshold, vecout) - Find the samples that are
                         GreaterEqual a certain threshold value and returns
                         the number of samples found and the positions of the
                         samples in a second vector.

hFindGreaterEqualAbs(vec, threshold, vecout) - Find the samples whose
                         absolute values are GreaterEqual a certain threshold
                         value and returns the number of samples found and
                         the positions of the samples in a second vector.

hFindLessThan(vec, threshold, vecout) - Find the samples that are LessThan a
                         certain threshold value and returns the number of
                         samples found and the positions of the samples in a
                         second vector.

hFindLessThanAbs(vec, threshold, vecout) - Find the samples whose absolute
                         values are LessThan a certain threshold value and
                         returns the number of samples found and the
                         positions of the samples in a second vector.

hDownsample(vec1, vec2) - Downsample the input vector to a smaller output
                         vector.

hDownsample(vec, downsample_factor) - Downsample the input vector by a cetain
                         factor and return a new vector.

hFindLowerBound(vec, value) - Finds the location (i.e., returns integer) in a
                         monotonically increasing vector, where the input
                         search value is just above or equal to the value in
                         the vector.

hFlatWeights(wlen)     - Returns vector of weights of length len with
                         constant weights normalized to give a sum of unity.
                         Can be used by hRunningAverageT.

hLinearWeights(wlen)   - Returns vector of weights of length wlen with
                         linearly rising and decreasing weights centered at
                         len/2.

hGaussianWeights(wlen) - Returns vector of weights of length wlen with
                         Gaussian distribution centered at len/2 and
                         sigma=len/4 (i.e. the Gaussian extends over 2 sigma
                         in both directions).

hWeights(wlen, wtype)  - Create a normalized weight vector.

hRunningAverage(idata, odata, weights) - Calculate the running average of an
                         input vector using a weight vector.

hRunningAverage(idata, odata, wlen, wtype) - Overloaded function to
                         automatically calculate weights.


SECTION: RF (Radio Frequency) Function
--------------------------------------------------
hGeometricDelayFarField(antPosition, skyDirection, length) - Calculates the
                         time delay in seconds for a signal received at an
                         antenna position relative to a phase center from a
                         source located in a certain direction in farfield
                         (based on L. Bahren).

hGeometricDelayNearField(antPosition, skyPosition, distance) - Calculates the
                         time delay in seconds for a signal received at an
                         antenna position relative to a phase center from a
                         source located at a certain 3D space coordinate in
                         nearfield (based on L. Bahren).

hGeometricDelays(antPositions, skyPositions, delays, farfield) - Calculates
                         the time delay in seconds for signals received at
                         various antenna positions relative to a phase center
                         from sources located at certain 3D space coordinates
                         in near or far field.

hGeometricPhases(frequencies, antPositions, skyPositions, phases, farfield) -
                         Calculates the phase gradients for signals received
                         at various antenna positions relative to a phase
                         center from sources located at certain 3D space
                         coordinates in near or far field and for different
                         frequencies.

hGeometricWeights(frequencies, antPositions, skyPositions, weights, farfield)
                         - Calculates the phase gradients as complex weights
                         for signals received at various antenna positions
                         relative to a phase center from sources located at
                         certain 3D space coordinates in near or far field
                         and for different frequencies.

hSpectralPower(vec, outvec) - Calculates the power of a complex spectrum and
```

```
                         add it to an output vector.

hADC2Voltage(vec, adc2voltage) - Convert the ADC value to a voltage.

hGetHanningFilter(vec, Alpha, Beta, BetaRise, BetaFall) - Create a Hanning
                         filter.

hGetHanningFilter(vec, Alpha, Beta) - Create a Hanning filter.

hGetHanningFilter(vec, Alpha) - Create a Hanning filter.

hGetHanningFilter(vec) - Create a Hanning filter.

hApplyFilter(data, filter) - Apply a predefined filter on a vector.

hApplyHanningFilter(data) - Apply a Hanning filter on a vector.

hFFT(data_in, data_out, nyquistZone) - Apply an FFT on a vector.

hInvFFT(data_in, data_out, nyquistZone) - Apply an Inverse FFT on a vector.


SECTION: I/O Function (DataReader)
--------------------------------------------------
hFileSummary(dr)       - Print a brief summary of the file contents and
                         current settings.

hFileOpen(Filename)    - Function to open a file based on a filename and
                         returning a datareader object.

hFileGetParameter(dr, keyword) - Return information from a data file as a
                         Python object.

hFileSetParameter(dr, keyword, pyob) - Set parameters in a data file with a
                         Python object as input.

hFileRead(dr, Datatype, vec) - Read data from a Datareader object (pointer in
                         iptr) into a vector, where the size should be
                         pre-allocated.

hCalTable(filename, keyword, date, pyob) - Return a list of antenna positions
                         from the CalTables - this is a test.


SECTION: Coordinate Conversion (VectorConversion.cc)
--------------------------------------------------
hCoordinateConvert(source, sourceCoordinate, target, targetCoordinate,
                         anglesInDegrees) - Converts a 3D spatial vector into
                         a different Coordinate type (e.g. Spherical to
                         Cartesian).

hReadFileOld(vec, iptr, Datatype, Antenna, Blocksize, Block, Stride, Shift) -
                         Read data from a Datareader object (pointer in iptr)
                         into a vector.
```