# Design of the AWProjectFT framework

S. Bhatnagar

Feb., 2022

**Abstract**

This memo describes the design and implementation of the AWProjectFT framework and the related classes.

## 1 Introduction

The AWProjectFT class implements a frame-work with pluggable components. The design was driven to follow the interface of the FTMachine base class so that it can be plugged-in in the SynthesisImagerVi2 framework like any other FT machine. This required that any and all operations required to realize the AWProject and WB AWProject algorithms be managed internally in these classes. The resulting class structure is shown as a UML diagram in Fig. 1.

The need to enable A- *and* W-projection together leads to both compute-load and memory-footprint (of the required CFs) issues. This therefore required development of *extendable, easily configurable and efficient* code for pre-computing and caching CFs as far as possible. The number of combined convolution functions in the A-W plane for wide-band imaging quickly becomes prohibitive, due to compute load for computing the CFs and the resulting memory footprint if they are all held in the memory. The CFCache class was therefore developed to manage the in-memory and on-disk models of a CF cache. For a general implementation of a wide-band AW-Projection algorithm that includes rotation of the PB on the sky, correction for the pointing and mosaic offsets for heterogeneous array, the address (indexing) to access a given 2D CF is five dimensional. Those dimensions are BaselineType, PA, W, Freq, Pol.

The CFStore2, CFBuffer and CFCell class, which works with the CFCache class, were developed to access the 2D CF in a three-level indirection shown in Fig. 2. This three-level indirection was felt necessary to realize the flexibility necessary for optimal run-time and memory footprint implementation that gave control, where required at the user level, e.g. on the granularity on the W-planes, PA-, BaselineType-axis or Freq-axis. With the current implementation the framework can be configured for a range of imaging cases and it has been measured to scale well in terms of memory footprint and computing efficiency for a wide range of use-cases ranging from sparse Mueller Matrix to Full-Mueller imaging, including support for time-variable pointing offset correction, support for homogeneous and heterogeneous arrays (the latter functionality is not yet fully deployed), rotation with PA and use of all of the available projection algorithms (see Table 1). In such a framework, distinction between single-pointing and mosaic imaging is not necessary and many of the settings can be derived automatically from the input meta data (but can be easily overridden by user settings, if necessary).

The following specializations of the AWProjectFT class extends its functionality for wide-band AW-Projection algorithm and use of external accelerator (GPU) for gridding:

- AWProjectWBFT: This is a specialization of the AWProjectFT class with wide-band specific code for the WB AWP algorithm.
- AWProjectWBFTHPG: This is a specialization of the AWProjectWBFT class with the necessary code to interface with the AWVisResamplerHPG, which in-turn uses the HPG.

## 2 The pluggable components in the AWProjectFT framework

The classes with AWProjectFT heritage operates with three pluggable components: VisibilityResampler, ConvolutionFunction and CFCache objects which are described below.

Figure 1: The class diagram for the `AWProjectFT` framework. Ⓐ indicates an abstract, Ⓘ an interface and Ⓒ an implementation class.

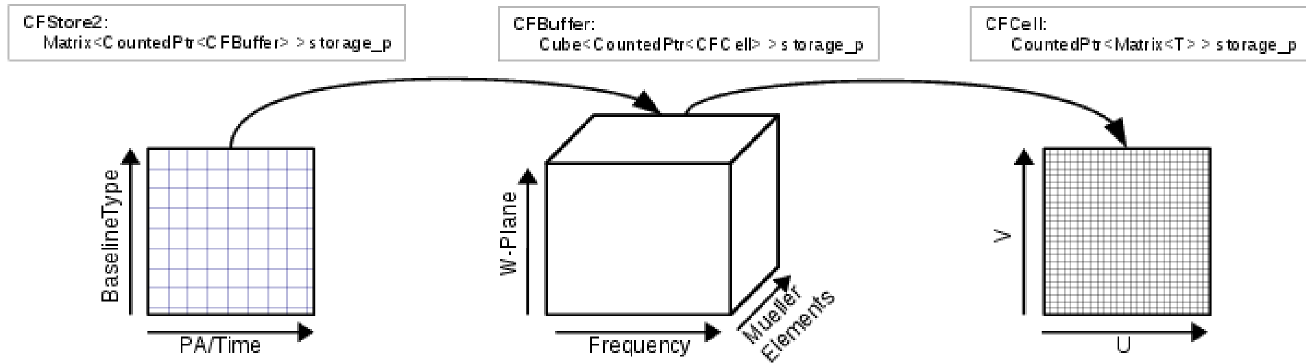


Figure 2: The structure of, and the relationship between the CFStore2, CFBuffer and CFCell objects.

## 2.1 The VisibilityResampler object

The visibility-resampler (a.k.a. the "gridder") is an object of type VisibilityResampler. The primary purpose of this object is to provide gridding and degridding services. These services are provided by the overloadable methods DataToGrid() which has a single- and double-precision variants and GridToData() respectively. For efficient implementation of the gridding loops which drive a large fraction of the total cost of imaging, a number of other methods are defined to set various parameters and maps that are then used in the gridding loops.

## 2.2 The CFCache object

The CFCache object manages the disc cache of pre-computed convolution functions (CF). A functional goal that drove the design was to also have a simple mechanism to possibly build a database of CFs which becomes increasingly complete with usage for a given telescope. E.g. one can imagine building a central respiratory of CFs which is complete with a sky-grid that covers the range of w-terms encountered by a given array telescope, effectively eliminating the cost of computing CFs (which can be prohibitive).

Separating the management of CFs (calculations, storage, access, memory footprint, etc.) into a separate object also decouples observatory-specific terms like the A-term from general imaging framework. This cleanly separates the responsibilities – e.g. managing the A-term remain an observatory responsibility, leaving the responsibility of the software with the scientific software team independent of the observatory operations.

CFCache works with the classes that encapsulate the in-memory model for the CFs required from wide-band AW-Projection with support for mosaic imaging and antenna pointing offset corrections. These are the CFStore2, CFBuffer and CFCell classes, which implement smart data structures that encapsulate the in-memory storage model of the CF pixels which are indexed by a 7-dimensional indexing involving BaselineType, PA, W, Freq, Pol, X,Y (see Fig. 2). These classes implement access methods to ultimately get CFCell and associated parameters of the CF. The 7-dimensional indexing is separated into these three classes for efficient access, optimized memory management and possibility of paging algorithms to control memory footprint.

1. CFStore2 is a 2D array of pointers to CFBuffer. The two axis of this are indexed by BaselineType and PA indices.
2. CFBuffer is a cube of pointer to CFCell. The three axis of this are index by W, Freq, and Pol indices.
3. CFCell is a 2D array of pixel values. This class also caches a number of CF parameters that are required in tight-loops (e.g. in the gridder).
4. CFDefs has the namespace for enums used in AWProjectFT framework.

The PA and BaselineType indices of CFStore2 returns a CFBuffer appropriate for the BaselineType at the given PA. Since the CFBuffer can change with these two indices, this separation of indices is necessary to allow access to

possibly multiple CFBuffers that might be needed, e.g. for heterogeneous array support with antennas potentially at different PA value.

The W, Freq and Pol indices for the CFBuffer returns the appropriate CFCell. Depending on the shape of CFBuffer one can easily realize A-, W-, AW-Projection, with or without Full-Mueller imaging. This has also quite easily allowed LAZYFILL mode where individual CFs are paged-in only when accessed and at appropriate boundaries garbage collection algorithm keeps the CF memory footprint within a reasonable value. Crucially for using external compute accelerators like a GPU, this also allowed easy identification of and loading only a subset of the CFBuffer based on the data access pattern. Both of these have proved to be capability-enabling features for many use-cases. All this was possible without changes in the rest of the framework.

CFCell gives access to a simple 2D array of pixels, which is the actual CF used in gridding. Apart from the pixel array, this also caches a number of meta parameters, some required due to the limitations of the image coordinate system in casacore.

### 2.3 The ConvolutionFunction object

The ConvolutionFunction object encapsulates the code required for the calculations for the pixel values of the CFs. This class works with the CFCache object to populate the disk CF cache (either a fresh CF cache or add to an existing CF cache if new CFs are computed on-the-fly, e.g. for a new PA value not found in the CF cache).

This object in-turn has three pluggable components, namely the ATerm, WTerm and PSTerm of the CFTerm base-class heritage which are used to compute the required CFs. Each of these have the primary method named applySky() which computes the sky-domain functions describing the antenna aperture function, the W-term function and the PS function respectively. Each of these can be individually configured to be a NoOp, enabling the AWProjectFT framework to realize standard imaging, W-only-, A-only- and AW-projection imaging via user-driven parameters as shown in Table 1. To limit the support size of the resulting CF to a reasonable value an addition constraint that one out the ATerm and PSTerm must be operational is typically imposed in the client code.

| Operation | aterm | psterm | wprojplanes | CF |
|---|---|---|---|---|
| AW-Projection | T | T or F | >1 | PS*A*W or A*W |
| A-Projection | T | T or F | 1 | PS*A or A |
| W-Projection | F | T | >1 | PS*W |
| Standard | F | T | 1 | PS |

Table 1: Table of user-settings to realize various types of projection algorithms with the AWProjectFT framework. The form of the resulting convolution function is shown in the CF column.

## 3 Interfacing with the HPG

As mentioned before, in the AWProjectFT framework interfacing with the HPG library could be done by specializing the VisibilityResampler class that uses the hpg::Gridder internally, and a light-weight specialization of the AWProjectWBFT (see Fig. 1). These specializations implement the following required differences from the standard FTMachine call pattern:

- The HPG uses a transport data structure that is, unfortunately, slightly different from the casa::VisBuffer2 data structure. This requires packing (to send the data and images to the HPG) and unpacking (to receive the data and images from the HPG). This is implemented in the AWVisResamplerHPG::DataToGrid() method which packs the data for the HPG. It also unpacks the image from the HPG into a CASA image object.

- The AWVisResamplerHPG::DataToGrid() method manages the i/o between the CPU and GPU memory, and combines the degridding-gridding sequence to compute the residual images. This is different from standard FTMachine interface where the ::get() (for degridding the model visibilities and compute residual visibilities) and ::put() (to grid the residual visibilities) methods are called in a sequence. This is optimal on the CPU with a single memory address space, but not on a GPU (or any external accelerator in general). On the GPU combining these operations optimizes the compute-to-i/o ratio.

- Finally, the standard FTMachine pattern performs the gridding/degridding via the ::put() and ::get() methods. At the end of these operations the imaging framework SynthesisImagerVi2 calls FTMachine::getImage() which applies the FFT on the gridded data and performs any necessary normalization. Use of this pattern with the HPG would apply the FFT on the CPU. This (FFT on the CPU) was found to limit the runtime performance. Solution is to apply the FFT on the grid on the GPU itself, which is achieved by overloading the getImage() method in AWProjectWBFTHPG specialization.

# 4 The AWProjectFT Control flow

The following pseudo code shows the control flow in the AWProjectFT and derived classes.

## 4.1 Pseudo code 1

The pseudo code below shows the AWProjectFT constructor. Apart from other initialization, pointers to CF storage (the variables cfs2_p and cfwts2_p of type CFStore2) are initialized to the internal storage of the CFCache object. These structures are initialized with the CFs found in the initialization of the CFCache. When the CFCache object is configured in the LAZYFILL mode, only the meta data of the CFs is loaded and the associated CFCell storage is initialized to NULL. The CF pixels are paged-in when the CF is accessed in the tight gridding loops where runtime performance is all important.

```
//-------------------------------------
AWPFT::AWPFT()
{
            :
            :
  //get CFStore2 from CFCache:
  cfs2_p = CFCache->memCache2_p[0];
  cfwts2_p = CFCache->memCacheWt2_p[0];
            :
            :
}
```

## 4.2 Pseudo code 2

The pseudo code below shows the change in the overloaded method AWProjectFT::put(). Similar differences are in AWProjectFT::get().

```
//-------------------------------------
AWPFT::put()
{
  ::findConvFunction(Grid,VB);
      :
      :
  <<  Standard code in FTM::put() >>
      :
      :
  ::setupVBStore();
  ::resampleDataToGrid();
}
```

AWProjectFT::findConvFunction() is called in the beginning. If the required CFs are not found via the CFCache, this method triggers the calculations for the CFs. Otherwise this becomes a NoOp. This is followed by the code to prepare the data for gridding, like in other standard FTMachines.

At the end of this method, AWProjectFT::setupVBStore() and the virtual AWProjectFT::resampleDataToGrid() methods are called. For the AWProjetFT class itself, this directly calls the gridder/degridder (via virtual VisibilityResampler::Dat and virtual VisibilityResampler::GridToData() methods of the supplied resampler). This allows making AWProjectFT a framework class which can be extended for gridding/degridding by supplying specializations of the resampler.

The method setupVBStore() extracts the required references from the casa::VisBuffer2 in a smart structure called the VBStore. This keeps the FTMachine decoupled from the gridder and allows independent development on either side. This was originally designed (in 2015!) when the gridder on the GPU was written to work directly with CASA

data structures (like VisBuffer2). This has proved useful now, when the HPG data structures are in fact different. To send the minimum data over to the GPU and keep the interface stable with change in the data/parameters required for gridding (e.g. standard gridder vs. partitioned gridder), such a data structure was necessary. This is now used for similar purpose (to collect the required pointers/references needed for gridding/degridding) and also a convenient place to do any data/CF preparation (like CF rotation with PA, if needed).

### 4.3 Pseudo code 3

The pseudo code below shows the functional flow for the AWProjectFT::findConvFunction method.

```
//--------------------------------------
AWPFT::findConvFunction(Grid,VB);
{
  if (!paChangeDetector.changed(vb,0)) return;
  //
  //set chanMap, polMap, freqMap in VisResampler
  //
  visResampler_p->setMaps();
  visResampler_p->setFreqMaps();

  //
  // The call below triggers the making of an array of pointers
  // to CFBuffer from CFStore2 for each VB row held in the VB2CFBMap
  // class. Nearest CFBuffers are picked based on (PA,dPA) or rotated
  // to the required PA via
  //   ::setupVBStore()
  //     ->convFuncCtor_p->prepareConvFunction(vb,*vb2CFBMap_p)
  // depending on the user setup.
  //
  // This also computes and caches the CF phase gradients which
  // includes antenna-based pointing error+mosaic pointing offsets.
  // A group of antennas with the same pointing error are discovered
  // and one phase grad is computed per antenna-group.  E.g. for a
  // homogeneous pointing error (VLASS Epoch>=2), one phase grad is
  // computed for the entire array for the current VB.  For VLASS
  // Epoch-1 imaging this computes and caches two phase grads for
  // two groups of antennas with different (time-dependent) pointing
  // error.
  //
  // This return MEMCACHE if a CFBuffer for (PA,dPA) was found.
  // Else return NOTCACED.
  // The enums are defined in CFDefs.h in CFDef namesapce:
  //   enum CACHETYPE {NOTCACHED=0,DISKCACHE, MEMCACHE};
  //
  cfSrc = VB2CFBMap::makeVBRow2CFBMap(CFStore2, PA, dPA);

  //
  // if CF for the given (PA, dPA) is not found, trigger the creation
  // of the CFs
  //
  if (cfSrc == NOTCACHED)
     ConvolutionFunction::makeConvFunc(PA,dPA,...,CFStore2,
                                       !dryRun());

  //
  // set MuellerMaps.  These are used in the
  // VisResampler's internal loops
  //
  if (CFCache2::OTODone() (One-time-operation))
   {
      cfs2_p->initMaps();
      cfwts2_p->initMaps();
   }

  pbReady = cfcache_p->loadAvgPB(avgPB_p);

```

```
57   // If avgPB is not found, make it via the virtual makeSensitivityImage()
58   // method.  For AWProjectWBFT and it's derivatives this sets up
59   // the VR to accumulate the CFs in the first gridding cycle.
60   // At the end of the gridding cycle, the AWProjectWBFT::getImage()
61   //  method then applies the FFT and normalization on the grid.
62
63   if (!pbReady)
64      ::makeSensitivityImage() // This is overloaded in AWProjectWBFT
65
66   // Finally, also write the CFStore2 to the CFCache on disk.
67   // At this point CFStore2 has the required CFs for the current
68   // VB.
69   //
70   if (cfSrc != MEMCACHE)
71    {
72      //
73      // Also save the grid in CFCacheDir/uvGrid.im.
74      // Only the grid coord. sys. is required, but currently
75      // the grid pixels are also saved. This is used for
76      // making the CFs in parallel in a multi-process run.
77      //
78      cfs2_p->makePersistent();
79      cfwts2_p->makePersistent();
80    }
81 }
82 //--------------------------------------
```

This method first determines if a new set of CFs are required (currently only if the PA value changed significantly, as determined by the PA tolerance which is a user-level setup). If a new set of CFs are required, various maps (needed in the tight gridding/degridding loops) are setup, followed by an attempt to setup the map of `CFBuffer` per row of the VB. This is done via the `VB2CFBMap::makeVBRow2CFBMap()` which returns an `enum` of type `CACHETYPE` (defined in `CFDefs.h`). This is captured in the `cfSrc` variable.

If the required CFs were not found in the cache (memory- or disk-caches), calculation for the CFs is triggered via the `virtual ConvolutionFunction::makeConvFunc()` method of the supplied instance of the `ConvolutionFunction` object. Next, a one-time-operation (OTO) is done to initialize internal maps of `CFStore2` which may change due to the new set of CFs in the cache. The `CFCache::loadAvgPB()` method then attempts to load the average PB and returns `true` if it is found (computed in earlier iterations or invocations). If the average PB was not found, the `virtual AWProjectFT::makeSensitivtyImage()` is triggered. For the `AWProjectFT()`, this immediately computes the PB in the *image domain*. This method is overloaded in the `AWProjectWBFT` variant, where it triggers an internal setup of that class to compute the wide-band PB by accumulating CFs on a separate grid using conjugate CFs (depending on the user setup). The class also has the overloaded method `::getImage()` which, at the end of the *first* gridding cycle also applies the FFT on this grid and caches the result as `avgPB` for later use.

Finally, if the new CFs were computed but not cached on the disk, the new set is made persistent (written to the disk cache). This basic method of OTF CF computation allows a given disk-cache to grow with usage. The size of the (disk) cache does not significantly affect the runtime[1], but enables the potential of ultimately building a complete cache. In the `LAZYFILL` mode of the `CFCache`, cache size also does not have significant memory footprint costs.

## 4.4 Pseudo code 4

The pseudo code below shows the functional flow of the `AWProjectFT::setupVBStore()` method.

```
1 //--------------------------------------
2 AWPFT::setupVBStore(VBStore& vbs)
3 {
4   <Extract references from casa::VisBuffer2 into vbs>
5   :
6   :
7   // Lambda function
8   auto cleanup_setup = [&](CountedPtr<CFStore2>& cfs_l, const VisBuffer2& vb_l)
9   {
```

[1] This has a one-time setup cost when `CFCache` loads the meta data from it, which is currently limited by the inefficient implementation of the `casacore::ImageInterface` used for reading *just* the image meta data like the coordinate system and `MiscInfo` records.

```
10      //
11      // invoke CFStore2::invokeGC -- the garbage
12      // collector for CFs not required, at least immediately.
13      //
14      cfs_l->invokeGC(vbs.spwID_p);
15      //
16      // Set up the internal map between the CFStore2
17      // and VB rows in the instance of the VB2CFBMap object
18      //
19      vb2CFBMap_p->makeVBRow2CFBMap(*cfs_l, vb_l,...);
20    };
21    //
22    // Invoke PointingOffsets::fetchPointingOffset().  This sets up an
23    // array of pointing offsets per VB-row.  For doPointing=True, these
24    // offsets are mosaic off-sets + offsets in the POINTING_OFFSETS
25    // sub-table of the MS.  For doPointing=False, these are just the
26    // mosaic offsets.
27    //
28    po_p->fetchPointingOffset(*image, vb, doPointing);
29
30    if (making PSF or WEIGHTS image)
31      cleanup_setup(cfwts2_p, vb);
32    else // Making residual image
33    {
34      //
35      // Remove all Wt. CFs from the memory.
36      //
37      cfwts2_p->clear();
38
39      cleanup_setup(cfs2_p, vb);
40    }
41    //
42    // Apply any operation to prepare the CFs.  If convFuncCtor_p
43    // is of type AzElAperture, this will rotate the CFs for PA,
44    // if necessary.
45    //
46    convFuncCtor_p->prepareConvFunction(vb,*vb2CFBMap_p);
47    //
48    // Send the instance of VB2CFBMap to the VisResampler instance.
49    //
50    visResampler_p->setVB2CFMap(vb2CFBMap_p);
51  }
```

The `setupVBStore` method was originally designed to setup the `VBStore` object (and therefore named as such). It is now used also for other set-up necessary for optimal performance of the gridder code which is invoked immediately after this call. Over time this framework call has proved to be very useful for deploying new functionality without a larger-scale ripple effect in the code base (e.g. support for heterogeneous pointing offset correction, optimized differently for run-time and memory footprint for the CPU and the GPU gridders).

## 5 Deprecable code/files

1. Removed the following files from the repository (commit: `c2020e89ba4330d9b882b03cce816c15e2221da5`; branch: CAS-13581-Cleanup):
   (a) `PixelatedConvFunc.{cc,h}`
   (b) `WOnlyConvFunc.{cc,h}`
   (c) `EVLAConvFunc.{cc,h}`
   (d) `NoOpATerm.h`
2. The following classes were missed in the above check-in, but can also be removed: `AWProjectWBFTNew`, `AWProjectWBFTNewHP` and `AWVisResamplerHPGNew`.

3. CFStore: Not required (superseded by CFStore2), but removing the dependence has proven to be harder (needs some careful testing before committing). To be done. I'll work on this at a lower priority for now.

4. PBMosaic, nPBWProjectFT: These are in MeasurementComponents directory and not used in AWProjectFT line of code, but are still (unfortunately) used in the Calibrator line of code for Pointing SelfCal. Ultimately this can/will be replaced by AWProjecFT. I'll work on this at a lower priority.