Heterogeneous Pointing Corrections in AW-Projection

P. Jagannathan and S. Bhatnagar

Oct., 2019

Abstract

Due to the high wide-band sensitivity of EVLA and ALMA telescopes, imaging performance can be limited by the antenna pointing errors (see Rau & Bhatnagar, 2018 for VLASS 1.1 imaging limits). These pointing errors in general also vary significantly across the array and with time. Corrections for their effects can be included during imaging with projection algorithms. The general requirement therefore is for an algorithm to handle use-case ranging from a homogeneous array treatment in pointing errors (same pointing error for all antennas) to a fully heterogeneous array treatment (different pointing errors for each antenna). This lead to the development of a general framework for on-the-fly application of the pointing corrections during imaging within the AW-Projection algorithmic framework.

This memo describes the implementation of antenna pointing corrections for the AW-Projection algorithm in CASA. We describe the theoretical framework used for antenna pointing corrections, the implementation and its scaling in computer resource usage. A carefully simulated data for regression testing and VLASS 1.1 data set were used for the scientific verification of the implementation. We briefly also show the effectiveness in improving imaging performance.

Executive Summary

Algorithm for correcting antenna pointing offsets with respect to the center of the image was implemented and tested to work as expected for both single-pointing and mosaic imaging. For this, two new user parameters in the `tclean` task of CASA are introduced which allow the full range of treatment of *known* antenna pointing errors – from homogeneous array to fully heterogeneous array in antenna pointing. Based on a user parameter (`pointingoffsetsigdev`), group of antennas for which the same pointing offset applies are dynamically discovered internally. Tests were done to ensure that the computing load and memory footprint scale as expected. Specifically, the extra compute load and memory footprint for the homogeneous array (e.g., `pointingoffsetsigdev=3000.0`) case is insignificant compared to no pointing correction (`usepointing=False`). Extra resources used with a few antenna groups is also minimal and is maximum for the fully heterogeneous case. Tests with EVLA data suggest that even for the fully heterogeneous treatment, a typical computer currently in use is sufficient.

User parameters: Two new user parameters were introduced as described below in the `tclean` task of CASA.

1. `usepointing` (Default: `False`): When set to `True`, the antenna pointing vectors are fetched from the POINTING sub-table. When set to `False`, the vectors are determined from the FIELD sub-table (effectively disabling correction of antenna pointing errors).
2. `pointingoffsetsigdev` (Default: [10.0,10.0] arcsec): The first value is the size in arcsec of the bin used to discover antenna grouping for which phase gradients are computed. A compute for a new phase gradient is triggered for a bin if the length of the mean pointing vector of the antennas in the bin changes by more than the second value.

Compute load scaling: The run time expense of using pointing correction is only in computing the phase gradients per antenna group and is therefore cached in the computer RAM. The compute load scales with the frequency at which one of the following conditions occur, which triggers a cache re-compute: **(1)** the maximum size of the convolution function changes, **(2)** field ID changes, **(3)** antenna groups change (in the number of groups or in antenna distribution within a group), or **(4)** change in the number of rows in the `VisBuffer` being processed. For cases tested, the extra overhead is typically in the few percent range.

Memory footprint scaling: Memory footprint overhead scales linearly as a product of the number of antenna groups and the square of the maximum convolution function (CF) size.

Contents

1 Problem Definition

The effect of antenna pointing offsets is a Direction Dependent (DD) effect and fundamentally cannot be corrected as Direction Independent (DI) calibration term. I.e., correction for it must be included during imaging. Correction for array-wide offset is relatively simple and can be applied during imaging with minimal extra computing load. Correction for antenna-dependent pointing offsets requires modifying the A-term per affected baseline as[1] (Bhatnagar & Cornwell 2017):

$$\mathsf{A}_{ij} = \mathsf{A}^{\circ}_{ij}\left[e^{-\frac{\left(l_i-l_j\right)^2\alpha^2}{2}}\right]e^{\iota\pi u_a\left(l_i+l_j\right)} \tag{1}$$

where A_{ij} is the modified A-term, A°_{ij} is the original A-term computed without antenna-dependent pointing offsets, l_i and l_j are the pointing offsets for the two antennas and α is $2^{-1/2}$ times the inverse of the beam-width. The origin of the u_a co-ordinate is at the center of the A_{ij} function and spans the support size of A_{ij} on the uv-plane. Given A°_{ij}, this modified A-term can be computed on-the-fly. The complex exponential in Eq. 1 is the phase ramp across the A-term. This is similar to the mosaic-term, except that the mosaic-term is array-wide which is computed and stored (in memory) as a single copy for all baselines and computed only for finite change in the phase center. The phase term in Eq. 1 on the other hand needs to be computed per affected baseline and potentially as a function of time.

The term in square brackets is the reduction in the amplitude of the correlation coefficient due to antenna-pointing related decorrelation and changes across the band (i.e., it is a frequency dependent correction). This is close to unity and can be ignored for small offsets (few percent of the beam-width) – it is unity for baselines where the two pointing offsets are the same. The effect of this term needs to be evaluated carefully before ignoring this term for large pointing offsets (e.g. $> 5\%$ of the PB HPBW). Including this term will further add to the computing load.

As mentioned above, an array-wide phase gradient is already in use for mosaic-imaging. The requirement for pointing error corrections is therefore to augment this existing functionality to apply potentially different phase gradients for each baseline. However that adds extra computing load which can be comparable to the gridding load for the affected baselines. To keep the run time and memory footprint costs reasonable, the requirement for the implementation is to trigger a re-compute of the phase gradients only on a state change, and a computationally cheap algorithm to discover state changes. Briefly, the solution used here is to cache computations for A_{ij} and the associated meta-data used for state change discovery (see Sections. 2, 3 for details). This reduces the computational overhead to acceptable levels at the cost of modest increase in the memory footprint even for the fully heterogeneous array case. The memory footprint overhead can also be further reduced with caching and opportunistic re-loading/re-computations.

2 Algorithm

As mentioned earlier, the extra compute load for antenna pointing corrections is almost entirely in the computation of the phase gradients, potentially different for each baseline. This overhead can be significantly minimized by caching and a state machine model for triggering the re-computation of the phase gradients. State transitions that triggers a cache re-compute are described below. Implementation details and scaling laws for computing and memory footprint are described in more details in Section 3.

1. **Change in the maximum size of the convolution function (CF) encountered.**
   For AW-Projection, the support size and therefore the in-memory size of the CF changes with frequency and w-term. Since the phase gradient in Eq. 1 is independent of frequency and the value of the w co-ordinate, it is sufficient to

[1] Written in 1-dimensional form for clarity

compute and cache this gradient for each unique value of $(l_i + l_j)$ for the largest CF support size encountered. A re-compute will be triggered only when a CF with a larger support size is encountered. Once the CF with the largest support size in the CFCache is used for imaging, this state parameter will be ineffective (i.e., this parameter will no more trigger a phase gradient re-compute).

2. **Change in antenna group (in the number of groups or in antenna distribution within a group).**

   A 2D grid of square pixels of size given by the `pointingoffsetsigdev` parameter is maintained. Each pixel of this grid holds a list of antennas who's pointing vector lies within the cell. For an array with N_{ant} antennas, this is equivalent of nearest-neighbor gridding of the N_{ant} vectors which is computationally cheap and is done for pointing vectors fetched for each VB. This makes the implementation sensitive to time-dependent changes in antenna pointing or antenna grouping. A phase gradient is computed and cached for each populated cell of this grid. A recompute is triggered for a change in the number of populated cells or in the list of antennas in a cell.

   The additional run time expense is controlled by the frequency of change in the antenna pointing and grouping. This in-turn is controlled by the `pointingoffsetsigdev` parameter. For a large value (e.g. $\gg$ the antenna FoV), all antennas fall in a single group and the group does not change with time. A single phase radiant is computed only once leading to run time cost same as that for standard mosaic imaging. For small values (e.g. $\ll$ resolution element), antennas will each fall in a different bin if their pointing vector differs by greater than `pointingoffsetsigdev`. A given antenna will move to a different bin if it's vector changes by greater than `pointingoffsetsigdev` (due to time variability).

3. **Change in the field ID (change in the phase center of the data being added to the grid).**

   This is typically the case for data acquired in the mosaicking mode. Data for each pointing arrives with a different FIELD ID. When this ID changes, the state variables 1 and 2 above are also re-activated and the re-compute for the entire phase gradient cache is triggered. This is potentially computationally expensive state transition, but is the fundamental cost for the correction of antenna pointing errors in mosaic imaging which cannot be further reduced.

4. **Change in the number of rows in the `VisBuffer` (VB) being processed compared to the previous VB.**

   If the number of rows in the VB changes, the entire phase gradient cache is marked as stale, which re-actives all of the above state variables. This is typically infrequent and also an expensive state-change, but can be made more efficient with some additional bookkeeping (see Section 5).

3 Implementation

For N_{AntGrp} number of antenna groups, the number of unique phase gradients required is $N_{PG} = N_{AntGrp} + \frac{N_{AntGrp}(N_{AntGrp}-1)}{2}$. We discover the number of groups present using the `pointingoffsigdev` parameter. The parameter is an array of floats where the first value (units of arcsec) represents the threshold deviation to be used to derive the antenna groups and the second value (units of arcsec) represents the threshold of change of the mean pointing offset across time for which the phase gradients and pointing offset groups are recomputed. For example if `pointingoffsetsigdev = [10,30]` then for the first visibility buffer timestamp the antenna groups are discovered by performing a nearest neighbour gridding of the pointing offsets corresponding to the first time-step. The grid is constructed by placing the antenna with the least extent in RA and Dec (Antenna 1 in Fig 1) at the bottom corner of the grid and the antenna with the maximum extent in RA and Dec at the top corner (Antenna 2 in Fig 1). This range is segmented into *n* pixels where each pixel is of the size of the first parameter of the `pointingoffsetsigdev = [10,30]` as shown in Fig 1. The phase gradients are computed for the difference of the two vectors. The phase gradients are then used to grid the visibilities of the antennas in each group. At the next time-step the new antenna pointing vectors are differenced against the first time-step used and if the mean offset has changed by more than the second parameter of `pointingoffsetsigdev = [10,30]` then the updated antenna pointing table is re-gridded and a new set of phase gradients computed, else we grid visibilities for the second time-step also utilizing the phase gradients from the first time-step.

3.1 Computational overhead

Computational complexity for each evaluation of the phase gradient cache (for all affected baselines) is given by

$$O\left(N_{PG} \times S^2_{Max} \times CostOf(e^{-\iota x})\right) \tag{2}$$

where and S_{Max} is the in-memory size of the *largest* CF encountered. To get the run time computational cost, this expression needs to be multiplied by the number of times the cache re-compute is triggered. This depends on the combination of the user defined parameters and the input data.

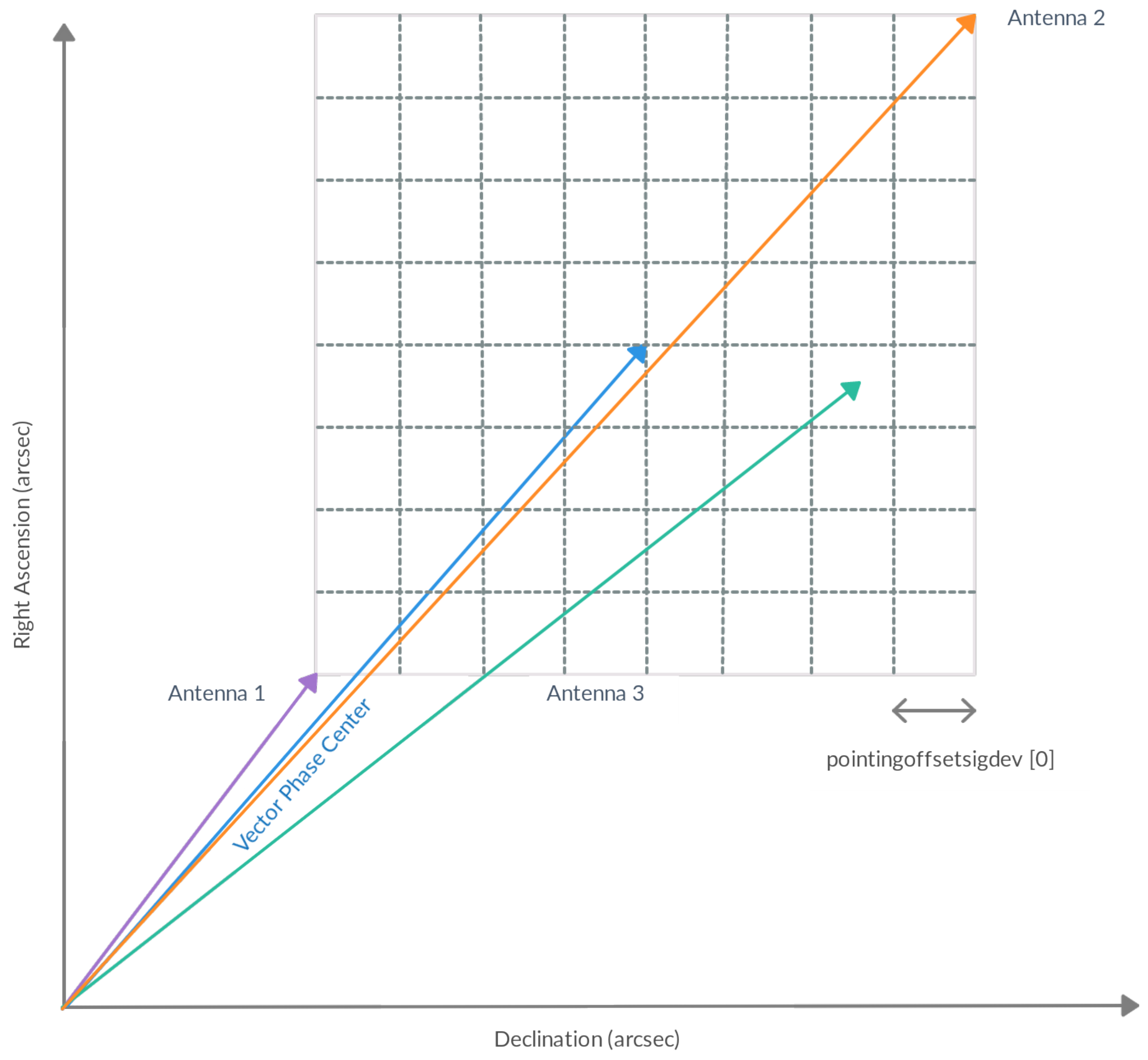

Figure 1: Shown is an illustration of the implementation. For an image with the phasecenter shown in blue, when `usepointing=True` the first `pointingoffsetsigdev` parameter decides the nearest neighbour grid size on which the antenna pointing vectors are gridded. In this case 3 unique antenna groups are found. The phase gradient is computed as the difference of the antenna pointing vector and the phase center.

3.2 Memory footprint overhead

$$8 \times N_{PG} \times S_{Max}^2 \text{ bytes} \tag{3}$$

Note that both, memory footprint and computational overhead will in general be dynamic since the N_{AntGrp} parameter is dynamically discovered and S_{Max} is dynamic with the `LazyFill` mode of the CFCache. Overheads can of course be measured for edge cases by appropriate setting of the user parameters. Lets examine the edge case of using a small enough `pointingoffsetsigdev` parameter such that the entire array is heterogenous in which case we have 351 phasegradients, one for each antenna baseline. This translate to an instantaneous increase in the memory footprint by 351. This is both prohibitive in memory and the in the compute cost to constantly. At the other end of the spectrum a really large value of `pointingoffsetsigdev` parameter such that the entire array is homogeneous will result in all the all antennas falling within a single cell of the pointing offset grid. This will result in a single phase gradient computed using the mean of all the pointing vectors lying within that cell, being applied to all the antennas. In which case the memory footprint and compute are comparable to a mosaicked AW-Projection run.

4 Code Overview

The pointing correction algorithm within the AW-Projection framework led to the creation of a series of new class objects. I will detail the overall flow and a general description of a subset of vital methods and class members. We began by re-factoring the method `AWConvFunc::makeVB2Row2CFBMap` into a class named `VB2CFBMap` which produces a map of the convolution function buffer required for the gridding and de-gridding of each row of the visibility buffer. This re-factoring allowed for the `VB2CFBMap` object to be a smart control object for the implementation of the algorithm. We also defined and created the `BaselineType` object to determine and cache the number of antenna groups present in the data given the user defined `pointingoffsetsigdev`. Two additional workhorse classes were also created `PointingOffsets` and `PhaseGrad`, the first to obtain the pointing offsets of the antennas from the POINTING table of the measurement set, and the second to derive a phase gradient with respect to the image phase center given the a pointing offset.

The method `VB2CFBMap::makeVBRow2CFBMap` is called from `AWProject` to produce a map of the convolution function buffer and produce the phase gradient to be used to grid the visibilities and weights. The `VB2CFBMap::setBLPhaseGrad` method is utilized to set the phase gradient for a given row of the visbility buffer. `VB2CFBMap::setBLPhaseGrad` requires prior knowledge of treatment of pointing corrections. When `usepointing` is set to False the pointing vectors are determined by the FIELD sub-table and the number of antenna groups and consequently the number of baseline groups is set to unity. When `usepointing` is set to True, the antenna groups are determined by calling `BaselineType::findAntennaGroups`. The antenna groups are then used to derive of the baseline group for each row of the visibility buffer using the method `BaselineType::makeVBRow2BLGMap`. The method `VB2CFBMap::setBLPhaseGrad` then utilizes the a priori map of the baseline groups in a visibility buffer and associated pointing offset for the baseline group to compute the phase gradient via `PhaseGrad::ComputeFieldPointingGrad`.

5 Future Directions

1. Optimize re-computing the phase gradient cache on VB size changes.
2. Optimize re-computing the phase gradient cache on antenna grouping changes.
3. Optimize the use of phase gradient cache for multiple instances of the AW-Project FT machine. Currently, this results in multiple copies of the in-memory cache for Multi-term (`nterms>1`) imaging.
4. A setting of `usepointing=True` and `pointingsigmadev>FoV` will treat the array as homogeneous in pointing errors for imaging, but will read the POINTING sub-table for each VB. The run time for this is observed to be longer than with `usepointing=False` (which also treats the array as homogeneous in pointing errors). This extra load scales with the number of rows in the POINTING sub-table, suggesting that the inefficiency is in accessing the POINTING sub-table. The POINTING sub-table in the test data sets we used (VLASS) typically has $5 - 10$ million rows (the MS has an order of magnitude less rows).

   Two solutions are possible to mitigate this extra compute load, listed below in the order of preference:

   (a) The time resolution of the POINTING sub-tables is few orders of magnitude finer than in the MS main table. Pruning the POINTING sub-table before imaging will largely eliminate this extra compute load.

   (b) Provide another user control to compute the antenna groups and the required phase gradients only once at the beginning (from the first valid VB). This will reduce the number of times the POINTING sub-table is accessed at the cost of making the imaging insensitive to temporal changes in the antenna pointing errors.

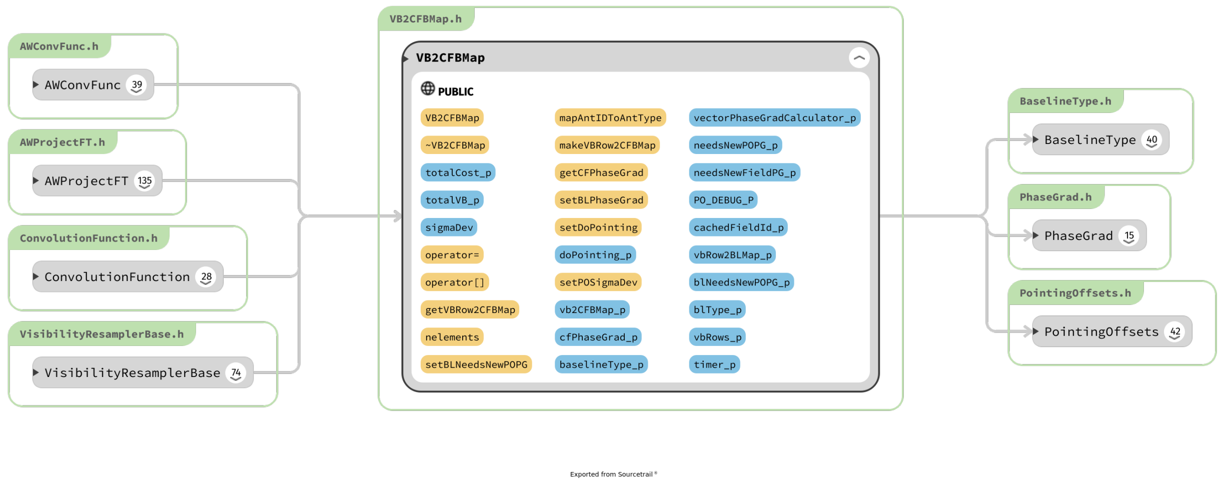

Figure 2: The members of the class VB2CFBMap are shown here. The methods are shown in yellow while fields are shown in blue. The arrows indicate the flow, Classes that are dependent on VB2CFBMap are shown to the left while the classes BaselineType PhaseGrad and PointingOffsets are referenced as counted pointers within VB2CFMap and shown leading out of it.

References

Bhatnagar, S. & Cornwell, T. J. 2017, The Astronomical Journal, 154, 197

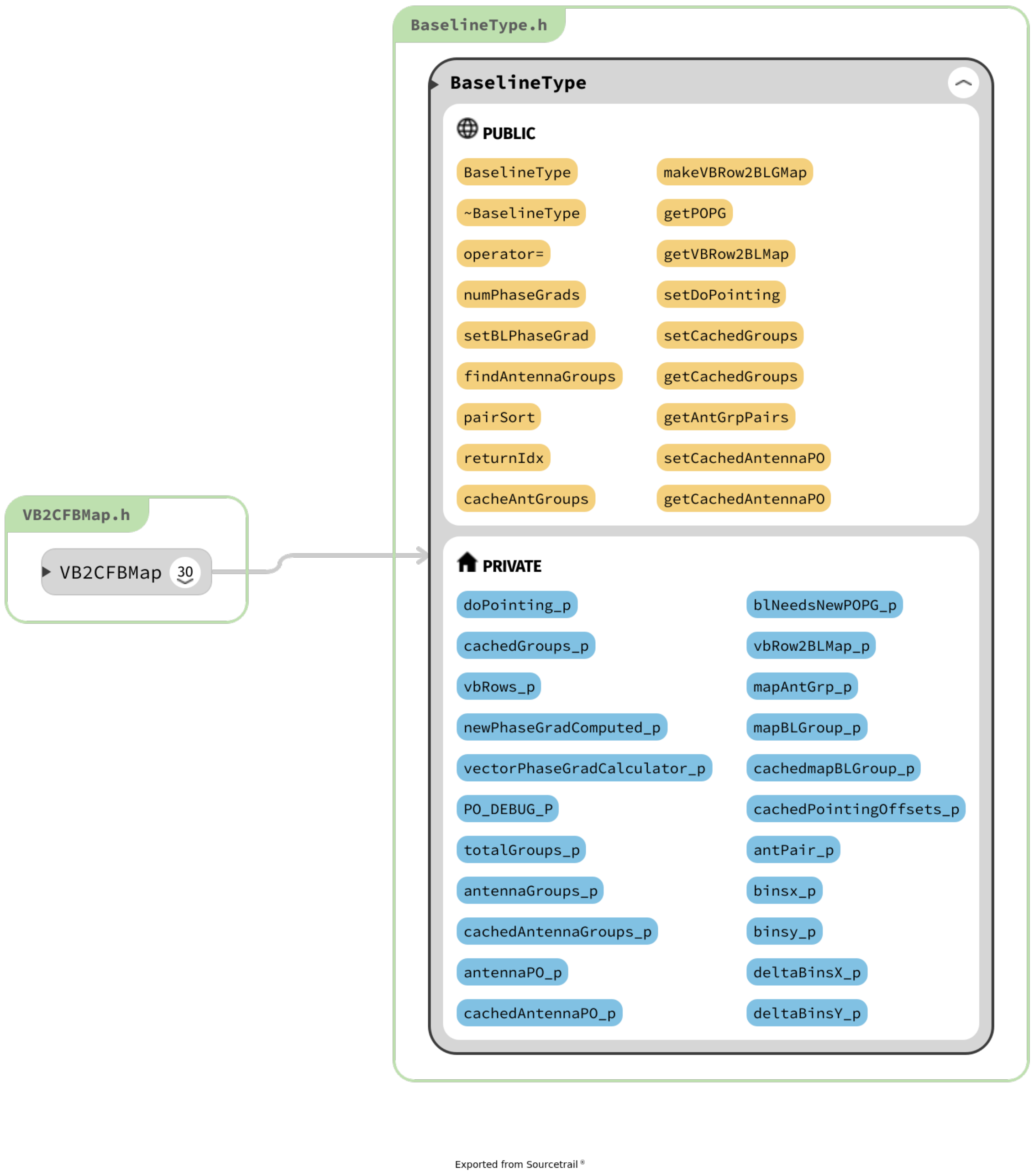

Figure 3: The members of the class `BaselineType` are shown here. The public methods are shown in yellow while private fields are shown in blue. The `BaselineType` class is the only class under the purview of `VB2CFBMap` it is setup to identify antenna groups as a function of the pointing table and will then trigger the

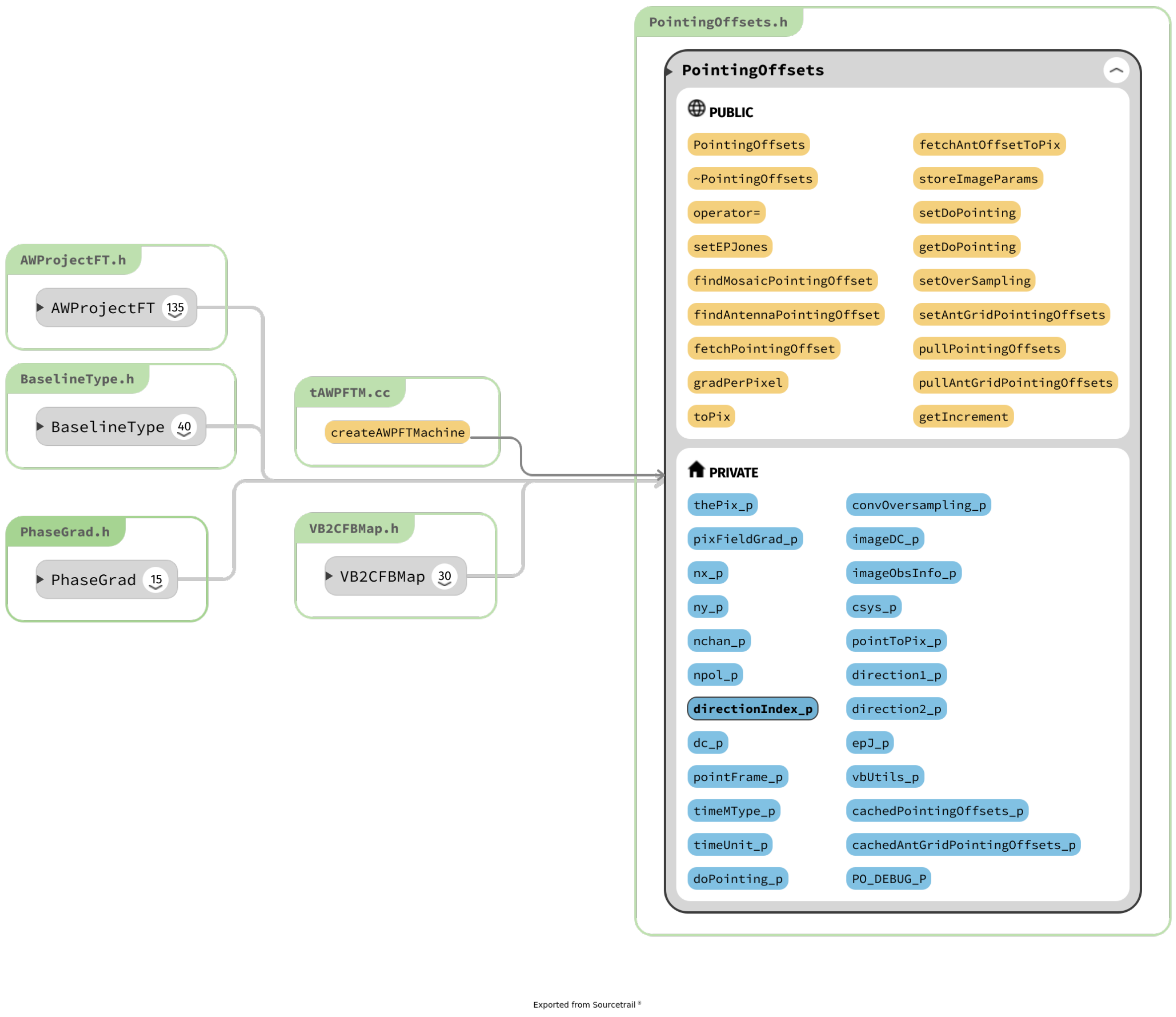

Figure 4: The members of the class `VB2CFBMap` and `BaselineType` are shown here. The public methods are shown in yellow while private fields are shown in blue. The class `BaselineType` is referenced as a counted pointer within VB2CFMap.

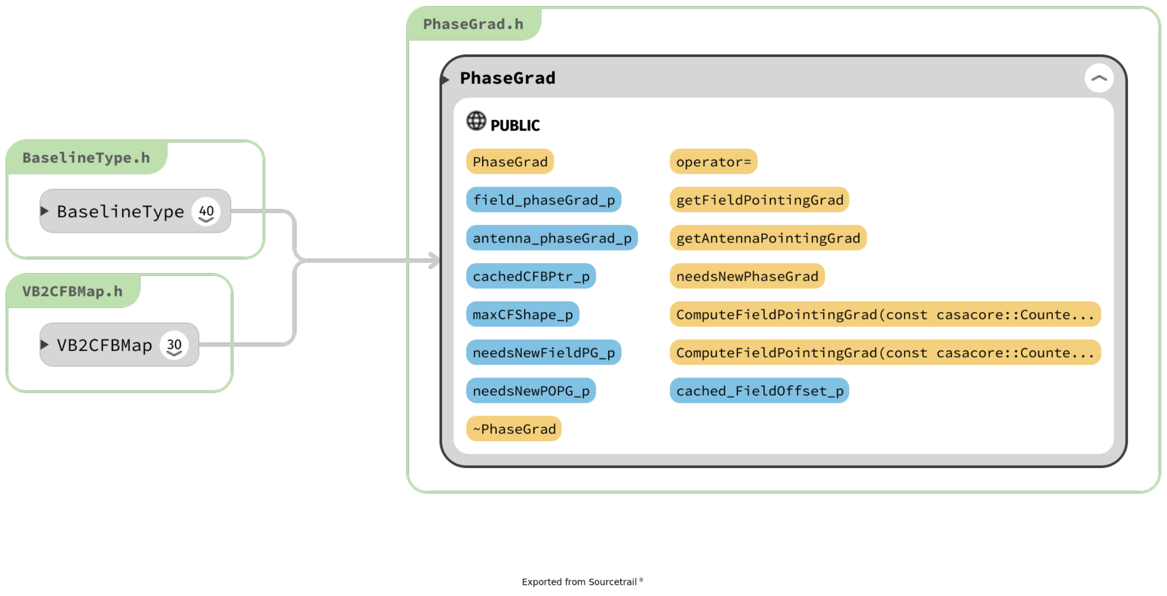

Figure 5: The members of the class `VB2CFBMap` and `BaselineType` are shown here. The public methods are shown in yellow while private fields are shown in blue. The class `BaselineType` is referenced as a counted pointer within VB2CFMap.