

Solar ALMA utils

Generated by Doxygen 1.17.0

1 Namespace Index	1
1.1 Namespace List	1
2 Class Index	3
2.1 Class List	3
3 Namespace Documentation	5
3.1 off_ephem Namespace Reference	5
3.1.1 Detailed Description	6
3.1.2 Function Documentation	6
3.1.2.1 edit_coordinates()	6
3.1.2.2 edit_coordinates2()	6
3.1.3 Variable Documentation	7
3.1.3.1 modified_line	7
3.2 sol_wvgcal Namespace Reference	7
3.2.1 Detailed Description	8
3.2.2 Function Documentation	9
3.2.2.1 acd_caltab_plots()	9
3.2.2.2 acd_calTabs()	10
3.2.2.3 acd_calTabs2()	11
3.2.2.4 blindPhiFix()	11
3.2.2.5 dB_sum()	11
3.2.2.6 dPWV()	12
3.2.2.7 dPWV_spwPart()	12
3.2.2.8 dpwvXphi_corrs()	13
3.2.2.9 dpwvXphi_plots()	13
3.2.2.10 dpwvXsolWVR_raw()	13
3.2.2.11 dpwvXwvr_corrs()	14
3.2.2.12 dWdB()	14
3.2.2.13 dWdB_Tbs()	15
3.2.2.14 dWdB_Tbs_NL()	16
3.2.2.15 dWdBdG()	16
3.2.2.16 dWdBdG2()	17
3.2.2.17 dWdBdG2_weights()	18
3.2.2.18 dWdBdG_calWeights()	19
3.2.2.19 dWdLs()	20
3.2.2.20 dWdLs2()	21
3.2.2.21 dWdLs3()	21
3.2.2.22 dWdLs3_calWeight()	22
3.2.2.23 dWdLs4()	22
3.2.2.24 dWdLs4_weights()	23
3.2.2.25 extMatInv()	23
3.2.2.26 G_off_cal()	24

3.2.2.27 getMeteo()	24
3.2.2.28 getMeteoForScan()	24
3.2.2.29 getPhase()	25
3.2.2.30 getPhaseMS()	25
3.2.2.31 getPWVforAnt()	25
3.2.2.32 paregression()	26
3.2.2.33 phiFixCompare()	26
3.2.2.34 phiFixCompare3()	27
3.2.2.35 plotPhases()	28
3.2.2.36 pwv4scans()	28
3.2.2.37 pwv4ScansCompare()	28
3.2.2.38 pwv4scansRestore()	29
3.2.2.39 pwvCalc()	29
3.2.2.40 pwvCalcVerify()	29
3.2.2.41 pwvCompare()	30
3.2.2.42 rebin()	30
3.2.2.43 rebin2()	30
3.2.2.44 restoreSol_dWdB()	31
3.2.2.45 saveSol_dWdB()	31
3.2.2.46 saveSol_dWdB_Tbs()	31
3.2.2.47 setAtmosSciSPWs()	32
3.2.2.48 setAtmTool()	32
3.2.2.49 setAtmTool4SPWs()	33
3.2.2.50 smooth()	33
3.2.2.51 sol_offPos()	33
3.2.2.52 sol_PhaseFix()	34
3.2.2.53 sol_pwv2selfCal()	34
3.2.2.54 sol_pwvGcal()	34
3.2.2.55 solTgtPos()	35
3.2.2.56 sqld_P0()	35
3.2.2.57 TbStats()	35
3.2.2.58 tgt_pow()	36
3.2.2.59 tgt_Tb()	36
3.2.2.60 tgtPow_crossPlots()	37
3.2.2.61 unwrap2()	37
3.2.2.62 wopacity()	37
3.2.2.63 wrapBack()	38
3.2.2.64 wv2path()	38
3.2.2.65 wvr_dataRaw()	38
3.2.2.66 wvrPhase()	39

4 Class Documentation

41

4.1 SolSG.calibrators Class Reference	41
4.1.1 Detailed Description	41
4.1.2 Constructor & Destructor Documentation	42
4.1.2.1 __init__()	42
4.1.3 Member Function Documentation	42
4.1.3.1 allCalsLst()	42
4.1.3.2 allCalsStr()	42
4.1.3.3 fluxCalQualityMetrics()	42
4.1.3.4 printInfo()	43
4.2 SolSG.observation Class Reference	43
4.2.1 Detailed Description	43
4.2.2 Member Function Documentation	43
4.2.2.1 printInfo()	43
4.2.2.2 rfStr()	44
4.2.3 Member Data Documentation	44
4.2.3.1 obsDate	44
4.3 SolSG.scienceTarget Class Reference	44
4.3.1 Detailed Description	45
4.3.2 Constructor & Destructor Documentation	45
4.3.2.1 __init__()	45
4.3.3 Member Function Documentation	45
4.3.3.1 isMosaic()	45
4.3.3.2 printInfo()	46
4.3.3.3 targetFields()	46
4.4 SolSG.stuffForSolarDataReduction Class Reference	46
4.4.1 Detailed Description	46
4.4.2 Member Function Documentation	46
4.4.2.1 generateSolCalScript()	46
Index	47

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

off_ephem		5
sol_wvgcal		7

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

SolSG.calibrators	41
SolSG.observation	43
SolSG.scienceTarget	44
SolSG.stuffForSolarDataReduction	46

Chapter 3

Namespace Documentation

3.1 off_ephem Namespace Reference

Functions

- [edit_coordinates](#) (time_str, ra_str, dec_str, azShift=2.0)
- [edit_coordinates2](#) (time_str, ra_str, dec_str, azShift=2.0)

Variables

- **TIME_START**
- **TIME_END**
- **RA_START**
- **RA_END**
- **DEC_START**
- **DEC_END**
- int **DEG_TO_RAD** = math.pi/180.
- int **RAD_TO_DEG** = 180/math.pi
- int **HRS_TO_RAD** = math.pi/12.
- float **RAD_TO_HRS** = 1.0/HRS_TO_RAD
- **observatory**
- int **phi** = alma_lat*DEG_TO_RAD
- str **input_file** = "./SC.txt"
- str **output_file** = "./off_pos.txt"
- bool **inside_ephemeris** = False
- **stripped_line** = line.rstrip("\n")
- **time_val** = stripped_line[TIME_START:TIME_END]
- **ra_val** = stripped_line[RA_START:RA_END]
- **dec_val** = stripped_line[DEC_START:DEC_END]
- **new_ra_val**
- **new_dec_val**
- tuple [modified_line](#)

3.1.1 Detailed Description

This code reads the ephemeris of the solar-disc center from the file SC.txt and for each its line it adds a predefined offset (default is +2deg) in *Azimuth* ! Shifted positions (transformed back to RA/Dec) are then written to the modified ephemeris file off_pos.txt. The original solar-center ephemeris is obtained by Solar Ephemeris Generator.

How does it work? For each line of the original ephemeris we

1. Convert RA/Dec to Az/El.
2. Add the desired offset (+2deg is the default) to Az (scaled by $1/\cos(EL)$)
3. Convert back to Ra/Dec.

Rationale: For the sake of test of compatibility between our PWV proxy and WVR data, we observe the Sun with two sub-arrays, Array 1 (~37 antennas) observes defined solar target, Array 2 (2 -3 antennas) measures co-temporally WVRs in the position off-the-Sun. We, however, need that the offset from the Sun is constant in Azimuth in order to preserve the same airmass for both arrays.

Two alternative approaches have been tried for the AZ-shift calculations:

(i) `edit_coordinates()`, which uses the aU utils for transforming to AzEl frame and back, and (ii), `edit_coordinates2()`, which uses the *measure* CASA tool for the same purpose. The results are similar (differences within 30 arcsecs), we use the latter one as it is more accurate.

3.1.2 Function Documentation

3.1.2.1 `edit_coordinates()`

```
edit_coordinates (
    time_str,
    ra_str,
    dec_str,
    azShift = 2.0)
```

Convert to AzEl and shift by *azShift* degs (positive means westward at north hemisphere). Then convert back to RaDec. We use methods from Analysis Utils for doing so.

The shift means the actual angular separation from the solar center in the direction of AZ, so it needs to be scale by $1/\cos(EL)$!

3.1.2.2 `edit_coordinates2()`

```
edit_coordinates2 (
    time_str,
    ra_str,
    dec_str,
    azShift = 2.0)
```

A more accurate alternative to `edit_coordinates()`. Now we use methods of *measure* tool from the CASA toolkit.

Convert to AzEl and shift by *azShift* degs (positive means westward at north hemisphere). Then convert back to RaDec.

The shift means the actual angular separation from the solar center in the direction of AZ! The `measure.shift()` method takes care about it.

3.1.3 Variable Documentation

3.1.3.1 modified_line

tuple off_ephem.modified_line

Initial value:

```
= (stripped_line[:RA_START]
    + new_ra_val
    + stripped_line[RA_END:DEC_START]
    + new_dec_val
    + stripped_line[DEC_END:])
```

3.2 sol_wvgcal Namespace Reference

Functions

- [rebin](#) (array1D, downSampleFactor)
- [rebin2](#) (array1D, origX, downSampledX)
- [smooth](#) (array1D, kernel)
- [paregression](#) (arrX, arrY)
- [unwrap2](#) (phase)
- [wrapBack](#) (phases)
- [extMatInv](#) (Ainv, Lb, Rb, LRC)
- [sqld_P0](#) (vis, ant, spw, myMS=None, myMsmd=None)
- [acd_calTabs](#) (vis, atmScan, ant, spw, tgtScan=None, myMS=None, myMsmd=None, acdLoads=None)
- [G_off_cal](#) (vis, atmScan, tgtScan, ant, acSpw, myMS=None, myMsmd=None)
- [acd_calTabs2](#) (vis, atmScan, ant, spw, myAtm=None, myMS=None, myMsmd=None, acdLoads=None)
- [acd_caltab_plots](#) (vis, atmScans, sciSPWs)
- [tgt_pow](#) (vis, tgtScan, ant, spw, myMS=None, myMsmd=None, subScans='on')
- [tgtPow_crossPlots](#) (vis, ant, tgtScan, spw1, spw2)
- [getPhase](#) (gTable, antID, spw)
- [getPhaseMS](#) (vis, antID, spw)
- [plotPhases](#) (vis, gTable, spw)
- [tgt_Tb](#) (vis, atmScan, tgtScan, antID, spwID, myMS=None, myMsmd=None, acdLoads=None, myAtm=None)
- [setAtmTool](#) (myAtm, vis, spw, scan, myMsmd=None, wvHeightScale=1.5, tempDropPerKm=-5.6)
- [setAtmTool4SPWs](#) (myAtm, vis, scan, myMsmd=None, wvHeightScale=1.5, tempDropPerKm=-5.6)
- [wopacity](#) (vis, spw, scan, userPWV=-1.0)
- [setAtmosSciSPWs](#) (vis, tgtScan, spwList)
- [getPWVforAnt](#) (vis, tgtScan, antID, myMsmd=None)
- [getMeteo](#) (vis)
- [getMeteoForScan](#) (vis, scan, myMsmd=None)
- [wv2path](#) (vis, spw, scan, pwv)
- [dPWV](#) (vis, atmScan, tgtScan, antID, spwAvg=6, spwRes=5)
- [dPWV_spwPart](#) (vis, atmScan, tgtScan, antID, spwRes=5, startChan=8, stopChan=120)
- [dWdB](#) (vis, atmScan, tgtScan, atmModel, antID, spwList)
- [dWdB_Tbs](#) (vis, atmScan, tgtScan, myAtm, antID, spwList, flaggedData=[], myMS=None, myMsmd=None, acdLoads=None)
- [dWdBdG](#) (vis, atmScan, tgtScan, myAtm, antID, spwList, flaggedData=[], myMS=None, myMsmd=None, acdLoads=None)
- [dWdBdG2](#) (vis, atmScan, tgtScan, myAtm, antID, spwList, T_tgt=-1.0e6, flaggedData=[], myMS=None, myMsmd=None, acdLoads=None)
- [dWdB_Tbs_NL](#) (vis, atmScan, tgtScan, myAtm, antID, spwList)

- [dWdBdG2_weights](#) (vis, atmScan, tgtScan, myAtm, antlID, spwList, T_tgt=-1.0e6, flaggedData=[], my←MS=None, myMsmd=None, acdLoads=None)
- [dWdBdG_calWeights](#) (vis, atmScan, tgtScan, myAtm, antlID, spwList, flaggedData=[], myMS=None, my←Msmd=None, acdLoads=None)
- [saveSol_dWdB](#) (vis, atmScan, tgtScan, spwList)
- [saveSol_dWdB_Tbs](#) (vis, atmScan, tgtScan, spwList)
- [restoreSol_dWdB](#) (vis, filePath, runID=100000)
- [dB_sum](#) ()
- [TbStats](#) (resCG, resVG)
- [dpwvXphi_plots](#) (doSun=False, useSQLD=False)
- [dpwvXphi_corrs](#) (doSun=False, useSQLD=False)
- [blindPhiFix](#) (doSun=False, useSQLD=False)
- [wvrPhase](#) (vis, antlID, spw)
- [pwvCompare](#) ()
- [phiFixCompare](#) (vis, atmScan, tgtScan, refAnt, spwID, spwRes=-1)
- [phiFixCompare3](#) (vis, atmScan, tgtScan, refAnt, spwLow, spwRes=-1)
- [dpwvXwvr_corrs](#) (SQLD=False)
- [sol_PhaseFix](#) (antlID, spw, pwvMap)
- [sol_pwv2selfCal](#) (antlID, spw)
- [sol_pwvGcal](#) (pwvMap, vis, refAnt, pwvCalTableName='sol_pwvCalTable.W')
- [sol_offPos](#) (vis, tgtScan, refAnt)
- [solTgtPos](#) (vis, tgtScan, refAnt)
- [wvr_dataRaw](#) (vis, tgtScan, antlID, subScans="")
- [pwvCalc](#) (vis, atmTool, scan, antlID)
- [pwvCalcVerify](#) ()
- [dWdLs](#) (antlID, myAtm, fullPWV=False)
- [dWdLs2](#) (myAtm, antlID, tgtScan, flaggedWVRchannels=[], ssType="", solIllumination=True, myMsmd=None, acdLoads=None)
- [dWdLs3](#) (vis, myAtm, antlID, tgtScan, flaggedWVRchannels=[], ssType="", solIllumination=True, my←Msmd=None, acdLoads=None)
- [dWdLs4](#) (vis, myAtm, antlID, tgtScan, flaggedWVRchannels=[], ssType="", solIllumination=True, my←Msmd=None, acdLoads=None)
- [dWdLs4_weights](#) (vis, myAtm, antlID, tgtScan, flaggedWVRchannels=[], ssType="", solIllumination=False, myMsmd=None, acdLoads=None)
- [dWdLs3_calWeight](#) (myAtm, antlID, tgtScan, flaggedWVRchannels=[], ssType="", solIllumination=True, my←MS=None, myMsmd=None, acdLoads=None)
- [pwv4scans](#) (vis, solWVR=False)
- [pwv4scansRestore](#) (vis, hasSolWVR=True, runID=100000)
- [dpwvXsolWVR_raw](#) ()
- [pwv4ScansCompare](#) (vis)

3.2.1 Detailed Description

```
@file sol_wvgcal.py
@brief Use WV line absorption/emission in AC/sqlD SPWs for phase correction.

@author barta@asu.cas.cz
@date 2026-04-03 (last update)
```

Experimentation with WV-line features - emission in case of the weak source (typically calibrator) and absorption for the Sun - in the science SPWs: either auto-correlations (ACs) of SQLD. We aim to use those features for eventual phase-error corrections on the short time-scales, as an alternative to WVR data and wvrGcal(), which are unavailable (saturated) in case of solar on-source scans.

The collection of routines here is devised for the following group of tasks:

1. Multi-purpose (non-specific) helpers: `rebin()`, `smooth()`, `paregression()`.
 2. Perform dual-load/ACD calibration: `acd_calTables()`; apply it to MS powers to convert them to calibrated `T_B: tgt_Tb()`. Diagnostics: `acd_caltab_plots()`.
 3. Extract the raw data from MSs and `calTables: tgt_pow()` and `getPhase()`. Diagnostics: `plotPhases()`.
 4. Setup the atmosphere (ATM) model and use it for calculation of opacities: `setAtmTool()`, `wopacity()`.
 5. Use all above + mathematical mode derived in PR11 to calculate the variations of PWV (proxy/model): `dPWV()`
 6. Analysis: Compare the dPWV proxy vs. phase variations
 7. Verification: Compare our dPWV proxy for the BP-calibrator with an actual PWV/WVR data measured in ASDM_CALWVR (accessed via `wvrgcal()`)
- 7.1 Specific for 2022.1.00031.CSV-Sun10a-05-INT SB: Analysis of unsaturated raw WVR data

Notes
=====

Ad (2): Note on comparison with `sol_sqld_mean_2()` in `sun_reduction_utils.py`. We extract the full AC and SQLD data from the MS (using `ms.select()`) and make their averaging here using `numpy.mean()`. The `sun_reduction_utils.py` uses `ms.statistics()` internally in its `sol_sqld_mean_2()` helper. For SQLDs, the results of `np.mean()` are very similar to those obtained by `sol_sqld_mean_2()`. The difference between our approach and that in `sol_sqld_mean_2()`, which uses `ms.statistics()` internally, is less than 0.015% (the relative error) and looks to be related by rounding inaccuracy (less accurate representation of the float data in our approach). Our approach is, however, more general - in the already mentioned sense of its applicability to both SQLD and AC (channelized or integrated) spws, but also because it can be used for the concrete selected antenna: `sol_sqld_mean_2()` accepts the antenna-names list, too, but it MUST be a complete list of **all** antennas for the given MS, otherwise mapping between antenna name and its ID (int number), which is internally used in the `sol_sqld_mean_2()` method, gets broken.

Ad (3): Note to experimenting with Dual-Load (DL: Hot, Ambient, Sky) calibration using ALMA ACD. We extract `P_hot`, `P_amb`, and `P_sky` for a given antenna and ATM scan. For the Sun, we extract also `P_off`, which is basically `P_sky`, but with the attenuation set to the high, solar-level, value. This is not present in the ATM scan preceding the solar-target scan, but in the Sun-scan itself: In its very first and very last sub-scan (see Fig. 4 in Shimojo et al., 2017). We extract also ACD load temperatures and - for SQLD spws - also the `P_zero` offset for the scan and calculate the gain `G` and `T_rx` according to formulae (7) and (9), and "solar" (i.e. in hi-attenuation setup) gain `G*` according to Eq. (8) in White et al. (2017).

The main difference to solar flux calibration implemented in `sol_ampcal2()` is that we can use here also

1. autocorrelation (AC) data and not just the SQLDs,
2. we may extract ACs in the ATM scans in SPWs with the channelized data in order to reconstruct the spectrally resolved `Tsky+Tsrc` (128 channels).

The obtained ACD calibration (gains `G` and `G*`, `T_rx` and in case of SQLD data also the offset `P_zero`) is subsequently applied to the target scan in order to convert measured raw powers (AC or SQLD) into brightness 'sky' temperature - the sum of source (attenuated by the terrestrial atmosphere) and the contribution of our atmosphere.

3.2.2 Function Documentation

3.2.2.1 `acd_caltab_plots()`

```
acd_caltab_plots (
    vis,
    atmScans,
    sciSPWs)
```

A diagnostic routine that plots G/G^* and T_{rx} as found in `acd_calTabs()` as a function of frequency in the channel-resolved SPWs for each antenna.

3.2.2.2 `acd_calTabs()`

```
acd_calTabs (
    vis,
    atmScan,
    ant,
    spw,
    tgtScan = None,
    myMS = None,
    myMsmid = None,
    acdLoads = None)
```

Calculates the (inverse) gains (G and, possibly, G^* in White et al. 2017), and T_{rx} for a given `*ant*` and `*spw*` from the ATM scan `*atmScan*` using its three sub-scans for Sky, Ambient, and Hot loads.

The `*ant*`, `*spw*`, and `*atmScan*` parameters MUST be set as `*int*` (not string), otherwise `ms.select()`, used internally, returns completely wrong selection - silently, without any warning! The 'global' tools `*myMS*` and `*myMsmid*` may be conveniently supplied, attached to the `MS *vis*`. If not, local tools are created and destroyed with the routine, with some additional cost on CPU time. The same holds for the global dictionary `*acdLoads*` that is obtained by call to `aU.getAntennaLoads()`.

If the `*atmScan*` precedes directly the solar scan, we calculate G^* , too. For the G^* , we use (in addition to the three ATM subscans) also the 'sky'/off-source sub-scans (the very first and last) in the immediately-following solar scan.

Returns: A dictionary with the inverse gain G (Eq. 7 in White et al.), inverse 'solar' gain G^* (Eq. 8), T_{rx} (Eq. 9), and effective total 'instrumental' temperature $T_{inst}=T_{rx}+P_o/G$. The G and T_{rx} components are numpy arrays of shape `nPols x nChans`, i.e., `2 x 128`, if we use the science SPWs (5,7,9,11) and `2 x 1` for channel-integrated SPWs (6,8,10,12) or SQLDs (0,1,2,3). The first dimension represents polarization and the second the channel number.

For G^* , we provide its two values - at the very beginning and very end of the solar scan (the very 1st and very last sub-scans in each solar scan are the off-source subscans). Hence, the shape of the returned array is `2 x nPol x nChans`, the first dim =2 representing the beginning and end values. According to Shimojo et al. (2017), the G^* should be calculated from the SQLD spws solely (it is said that P_{off} is too low for AC spws). Consequently, it should be an array of the shape `2 x 2 x 1`. However, we are experimenting here with the auto-correlations (AC) SPWs, too, both for spectral-resolved and channel-integrated spws, so the return shape may be well also `2 x 2 x 128`. The reason is rooted in the fact that the attenuation factor G^*/G found in the Band 5 data is just about 1.07. In case of the ATM scan precedes non-solar scan (e.g., BP-cal scan), we return `[[[1.0],[1.0]],[[1.0],[1.0]]]` - i.e. factor of 1, with the shape corresponding to the channel-integrated SPWs. Nevertheless: It has been found - see detailed comment in `tgt_Tb()`, in the section starting with 'if(doSolarGain)' - that PWV changing during the solar scan affects measured P_{off} and consequently the attenuation factor and resulting gain G^* . Since the attenuation factor $AF=(P_{sky}-P_o)/(P_{off}-P_o)$, it is better to take just the leading-subscan P_{off} measurement, because it follows immediately after the P_{sky} measurement in preceding ATM scan.

3.2.2.3 acd_calTabs2()

```
acd_calTabs2 (
    vis,
    atmScan,
    ant,
    spw,
    myAtm = None,
    myMS = None,
    myMsmd = None,
    acdLoads = None)
```

Improved version of `acd_calTabs()` for solar AC data. It uses transfer of relative gain-drop between `P_sky` and `P_off` scans from `SQLD` to AC spws - see experimentation in `G_off_cal()`.

Because of corrections to frequency-dependent transmissivity and intrinsic radiation of the terrestrial atmosphere in channel-resolved AC spws, we need to supply the ATM model `*myAtm*` for the solar G* calibration. It is the `*attool*` set for our atmosphere, for all 4 resolved sci SPWs, and it can be conveniently obtained via `setAtmTool4SPWs()` - see also `dWdB_Tbs()`.

If we do not calibrate for the Sun, i.e. when `*atmScan*` is not followed by a solar scan, `*myAtm*` parameter may remain empty. Similarly, if the `*spw*` is `SQLD`, we do nothing more on top of `acd_calTabs()`, so `*myAtm*` is ignored.

3.2.2.4 blindPhiFix()

```
blindPhiFix (
    doSun = False,
    useSQLD = False)
```

*** So far just for the solar MS project 2022.1.00031.CSV, Band 5!!! ***

Basically, we do the same calculations like in `dpwvXphi_plots()` but now we try to make blind corrections of the phases by dPWV with two options:

1. We fix phase by subtracting dPWV scaled by the theoretical factor -25.0.
2. Like in case (1.), but use the factor found by the correlation analysis.

Phases fixed by both methods are plotted into a single chart for comparison. We would like to see, what is residual phase variation after the correction.

@todo: For the Sun we do NOT have yet 'independent' phase solution to compare, we do it for the phases of the BP cal. Of course, it has no sense to compare the dPWV curve (scaled by -25.0) for the Sun with the phases for the BP-cal. The only goal is to see, whether we have similar amplitude and typical dynamics comparable or not.

3.2.2.5 dB_sum()

```
dB_sum ()
```

Sum the brightness variations as found by `dWdB()` for all antennas.

Idea: If there is an intrinsic `T_B` variation of the Sun, we arrive to this 'coherent' contribution in the sum. If not, and the dB changes are just because of atmospheric lensing, the average of dB over all antennas should tend to zero.

3.2.2.6 dPWV()

```
dPWV (
    vis,
    atmScan,
    tgtScan,
    antID,
    spwAvg = 6,
    spwRes = 5)
```

Calculate dPWV - our proxy to differential PWV (water-vapour column) based on the variations of received power. See formulae (13) and (14) in the Progress Report no. 11 (March 27, 2025). We may use both the auto-correlation (AC) channel-integrated SPW (i.e. SPW 6 in solar case) or, almost equally, the lowest SQLD spw (SPW 0) for that purpose. What spw is actually used is controlled by **spwAvg** parameter.

The **tgtScan** parameter defines, for what object is dPWV calculated, the **atmScan** argument is the scan used for ACD calibration (see `acd_calTabs()`; *atmScan* should precede the *tgtScan* as close as possible, ideally by one). Antenna, for which dPWV is calculated, is represented by its integer **antID**.

Returns: A tuple of three 1-D arrays, each of the size of *nIntegs*: First represents the time-axis as MJD (merely reproduction of the axis returned by `tgt_pow()`), the other two are values of dPWV for XX and YY polarizations, respectively.

3.2.2.7 dPWV_spwPart()

```
dPWV_spwPart (
    vis,
    atmScan,
    tgtScan,
    antID,
    spwRes = 5,
    startChan = 8,
    stopChan = 120)
```

Find dPWV - our proxy to the WV contents but now by using just a part of the SPW. Unlike in `dPWV()` we now use the channel-resolved SPW **spwRes**, by default that one closest to the WV line center (i.e. SPW 5).

Because the single-channel data are too noisy (that is the reason why spw-integrated ACs or SQLDs are being used), we do integration over multiple channels, but not entirely over full SPW. The frequency range for integration is defined by **startChan** and **stopChan** parameters, which default to the first and last 'usable' channel (the firsts and last 8 channels are typically flagged in any analysis). Note: For SPWs with reversed order the start/stop chan is also swapped so that the *startChan* always means the low-frequency boundary.

Idea: We may avoid O3 lines or use just that part of the SPW 5 closer to the WV line center and increase sensitivity to the WV contents. Suggested on PM12 in May 2025 (see PR12 - post meeting Action item 6), implemented in line with formula (1) in PR13 (August 2025).

3.2.2.8 dpwvXphi_corrs()

```
dpwvXphi_corrs (
    doSun = False,
    useSQLD = False)
```

*** So far just for the solar MS project 2022.1.00031.CSV, Band 5!!! ***

Calculate actual correlations between dPWV and phases. Just indicative plots are performed by dpwvXphi_plots() with anticipated coefficient of linear regression to be -25.0 (see the study in PR11, March 2025). Here, we perform actual correlation calculations, finding the coefficients of linear regression between dPWV and phases. However, the situation is not that simple: the preliminary analysis has shown (see PR11) that the phases tend to (perhaps linearly) grow also with time and time-based de-trending of the phase series should be performed. In order to do that, parametrized linear regression is computed here using paregression().

We may use ACs or SQLD for that - see dPWV(); this is controlled by *useSQLD*. Whether we do this for the BP-calibrator or the Sun is controlled by *doSun*.

3.2.2.9 dpwvXphi_plots()

```
dpwvXphi_plots (
    doSun = False,
    useSQLD = False)
```

*** So far specific just for the solar MS 2022.1.00031.CSV, Band 5 !!! ***

Plot the phase variations overplotted by our dPWV proxy, scaled by the theoretical factor dPhi/dPWV=-25.0 for each antenna - see the PR11. This is just the quick-check diagnostics.

We may use ACs or SQLD for that - see dPWV(); this is controlled by *useSQLD*. Whether we do this for the BP-calibrator or the Sun is controlled by *doSun*.

3.2.2.10 dpwvXsolWVR_raw()

```
dpwvXsolWVR_raw ()
```

A simple 'by-eye' comparison our dPWV proxy for the *solar* scan with the *raw* WVR data in the solar scan.

dPWV proxy is calculated from *science SPWs* using dWdB(). We utilize here the pre-calculated output saved in dWdB_data.npy

WVR is calculated from raw WVR data (SPW 4) via dWdLs().

We plot both time profiles as left and right panel of a single figure to see the differences.

*** Specific for 2022.1.00031CSV project! ***

This method cannot be used outside of the project - it was just a good luck that the WVR were pointing out of the Sun because the Band 5 receiver was pointed close to the limb - see fixSolWVRs() for details.

3.2.2.11 dpwvXwvr_corrs()

```
dpwvXwvr_corrs (
    SQLD = False)

***So far specific just for the solar project 2022.1.00031.CSV, Band 5 !!! ***

Find correlation and linear trends between our dPWV()-proxy calculated phase
correction and that found by the WVRgcal().

Very similar approach as used in dpwvXphi_corrs(), now used for correlation
analysis between two phase corrections cacluated by two different methods.

This routine cannot be even theoretically used for the Sun (no wvrgcal table).
```

3.2.2.12 dWdB()

```
dWdB (
    vis,
    atmScan,
    tgtScan,
    atmModel,
    antID,
    spwList)

*** The main routine for calculation of the dW(t) for solar scans.***

For the BP calibrator we may use dPWV() or its dPWV_spwPart() variant as there
is no further source of signal variations except of the atmopsheric column
mass changes manifested as variations in PWV.

For the Sun, this is, however, not true - it can have its own brightness
changes. Here we attempt to decompose the signal variations using a collection
of science SPWs in *spwList* (may be AC integrated, AC resolved, SQLD or any
combination, which incorporates at least two frequency domains) into:

(i) dW = total variations of the PWV - like in dPWV, and
(ii) dB = variations of the (apparent) source brightness.

The latter may involve both intrinsic changes of Tb_src and the lensing
effects of the turbulent atmosphere (a.k.a. caustics in low-frequency radio) -
these two are not easilly mutually distinguishable.

In order to do the disentagling, we need to use multiple SPWs.
In the single-channel SPWs we omit the correction implying from the slope of
Tsky(f) and tau(f), it has been experimentally found to be less then 5% for
dPWV().

The *atmModel* is a set of static 'tabelised' profiles (frequency-dependent)
for opacities, Tatm, etc., as returned by wopacity(). We have it as an array
for all science channel-resolved SPWs (5,7,9, and 11). It is supplied as an
parameter because it is the same for all antennas: We do not want to
re-creaaate it inside this routine whenever it is called for another antenna,
as it is usually done in a loop over entire array, e.g., in saveSol_dWdB().
When this routine is invoked in a loop for all array, the *atmModel* can be
conveniently pre-set by call to setAtmosSciSPWs().

The overdetermined system (we have just two variables - dW and dB, and up
to 12 equations for 12 SPWs) is then solved by the least-square principle,
leading to a minimizer of a quadratic form (which is another linear system of
eqs., but now regular one: 2x2 matrix with a nTimeSamples-long collection of
2-component RHS vectors).
```

Least-square residuals and other analytics are plotted in a compound 2 x 3 panels graph as a by-product of calculations. In the './dWdB/' subdir we plot the solutions dW(t) and dB(t).

Commented-out is a perspective extension, which tries to decompose the total dW variations into two parts, caused by (1) relative-humidity changes, and (2) total air-preessure variations. It has been experimentally found, however, that the system is then ill-conditioned for this decomposition. It can be improved by changing the vector basis and decomposing into total wet and total dry variations. However, even with this setup, the total wet contribution to the signal remains almost unchanged (so it is quite a robust quantity), the total dry component exhibits a large error and cannot be used.

@todo: In the ASDM_CALATMOPHERE table we have Tatm (a single value) per SPW. Try to compare its value with our Tatm as calculated by wopacity().

3.2.2.13 dWdB_Tbs()

```
dWdB_Tbs (
    vis,
    atmScan,
    tgtScan,
    myAtm,
    antID,
    spwList,
    flaggedData = [],
    myMS = None,
    myMsmd = None,
    acdLoads = None)
```

*** Need to be edited if used for obsevation different from 2023.12.16 ! ***
Namely the ehpeemeris-dependend data at the beginning of the script (indicated in comment).

An alternative approach to dWdB(), which, instead of using Tatm in the expression Tbs=Tatm*(1-tr) for atmospheric contribution to the sky brightness, uses directly Tbs=at.TebbSky(), smiliraly to our approach for PWV calculations from the WVR temperatures, like in pwvCalc() or dWdLsun().

It should be of the same order of accuracy and complexity, because the Tatm is calculated from the same ATM model anyway, just we have to scale it by the factor (1-tr).

The big advantage of this alternative approach is that we can easilly find also the scan-averaged PWV value, not just its differential variations.

We use the same approach as in dWdLsun(): Linearizing the radiative-transfer equation around its "operation point", which is defined by the scan-averaged PWV w0 and scan-averaged 'luminosity' of the Sun Tsun0. The linear set of equations is overdetermined for more than two SPWs (if they occupy distinct frequency domains), so we solve the set of 'normal' equations, ipmlying from the least-square principle, like in dWdB(). However, in our linearized set of equations, the variations need not to have zero mean. So we add the mean to the stationary 'operation point' defined by [w0,Tsun0] and make a new iteration, like in dWdLsun(), until we reach zero-mean variations. This way, both the variations and their stationary part is found. This approach has been verified via dWdB_Tbs_NL(), which solves the non-linear RT equation directly by Newton-Raphson iterations: Results of both differ insignificantly.

We do appropriate integration in the frequency domain for all model-based quantities. The solar QSR spectrum is being scaled by formula (4) in Alissandrakis et al. (2020), the original power-law estimation has been abandoned beacuse of large residua (see below in the code).

Moreover, we apply consequently the antenna efficiencies `eta_l` and `eta_F` (see White et al. 2017): For the fluctuations within in a single SPW they may be incorporated into effective gain and 'instrumental' temperature, but here, when using all SPWs and absolute calibration to solar `T_B` and full PWV, it is not possible.

And the last innovation: The PWV values should be common for both XX and YY polarization. On the other hand, the solar brightness `T_B` may differ in XX and YY channels (e.g., the apparent changes due to atm.-turbulence lensing may be polarization dependent, even the solar FoV may have a small amount of intrinsic polarization). Therefore, we have 3 independent variables: PWV, `TB_XX`, and `TB_YY`.

Arguments (these that differ from `dWdB()`):

myAtm: Here, `myAtm` is the `*attool*` set for our atmosphere, for all 4 resolved sci SPWs. It is a difference to `dWdB()`, where we use already pre-calculated ('tabelized') model, i.e., 'static' arrays for opacities, `Tatm`, transmissivity etc. for a specific value of time-averaged PWV for the scan. Here it is not possible as the mean PWV (the `w0` parameter) is one of the unknowns (in `dWdB()` we use its `ASDM_CALWVR` value interpolated to the center of the scan). The `*myAtm*` `attool` set for the frequency domain defined by the AC channel-resolved SPWs can be conveniently obtained via `setAtmTool4SPWs()`.

flaggedData: A list of strings in the form of (example) `['9:XX','3:YY']` with the `spw:polarization` pairs that should be ignored in the analysis. This can be used for multi-passage of this method, where in the first run we find the final-solution residua and the `SPW:Pol` pair, whose residuum is more than desired distance (e.g., 3 sigma) appart from the mean, can be flagged for the repeated run.

myMsmd, ***myMS***: Conveniently we can pass the tools already attached to the MS `*vis*`. If nothing is provided, the method allocates and attaches its own local tools (whoch costs some CPU time)

acdLoads: A dictionary of ACD temperatures (got by `aU.getLoadTemperatures()`) from which, for example, the ambient-load temperature can be extracted by `Tamb = acdLoads[antID][atmScan]['amb']`. If not provided, it is generated inside the routine (cumbersome, if we call it in the loop for all antennas).

3.2.2.14 dWdB_Tbs_NL()

```
dWdB_Tbs_NL (
    vis,
    atmScan,
    tgtScan,
    myAtm,
    antID,
    spwList)
```

Like `dWdB_Tbs()`, but now we do not use linearization. We apply brutal force to solve the full, overdetermined system of non-linear equations in least-square sense. Intended just for verification of the linearization method.

Results: It has been found that the difference in PWV is max. 0.01 mm and in brightness temperature ~3 K - see PR16. As a result, we further develop just the linearized version `dWdB_Tbs()`.

3.2.2.15 dWdBdG()

```
dWdBdG (
    vis,
```

```

    atmScan,
    tgtScan,
    myAtm,
    antID,
    spwList,
    flaggedData = [],
    myMS = None,
    myMsmid = None,
    acdLoads = None)

```

Extended dWdB_Tbs(): Solve for antenna gains and instrumental powers, too.

The most complex treating the RT problem for the Sun: Like with dWdB_Tbs() we get the time-series of full PWV and the (apparent, i.e. including the atmospheric lensing) solar brightness temperature – actually its reference value for $f_{LO}=198.0\text{GHz}$ – in XX and YY polarizations. But, here, we moreover set the antenna gains $*G*$ and the instrumental power $*P_{inst}=G*Trx+Po*$ as unknown parameters to be solved by the extended set of equations.

Rationale: dWdB_Tbs() seems to perfectly reproduce the variations in the solar brightness temperature and PWV: They are very similar in all SPWs in both XX and YY polarizations. Yet, there is a stationary part of the T_B , which is different among SPWs and pols – see the residuum plots. Actually, this static-part residua have been significantly reduced, already when we have introduced the corrected ACD calibration method `acd_calTabs2()` for getting measured $T_{sky}+src$ from directly recorded powers: The improved ACD calibration fixes the AC-spw P_{off} -based (i.e., with solar attenuation level) gain $G*$ by transferring the gain-drop level (given by P_{off}/P_{sky} ratio) from the adjunged (the same frequency domain) SQLD spw. Hence, it looks these differences are due to imperfect ACD gain estimation: 1% error in the gain easilly produces ~70K difference in the solar T_B . Therefore, this method has been devised to try to improve the gain calibration.

How it works: We assume that the antenna gains and P_{inst} are constant in time over a scan. Moreover, for the spectral AC SPWS, we assume that the normalised gain $g(f)=G(f)/\langle G(f) \rangle$, as used in `acd_calTabs2()`, is correctly calculated by the ACD calibration, and here we calculate just its scaling. Hence, for NT time-samples and $NDC = nSPW \times nPols = 24$ data-channels, we have $2 \times NDC$ unknown instrumental variables ($G + P_{inst}$ per SPW and pol) and, $3 \times NT$ 'natural' variables (PWV, T_{B_XX} , T_{B_YY}). This is still significantly less then number of equations, which is $NT \times NDC$. Nevertheless, the matrix of such system has very high condition number. It is connected with the fact that we have no anchor for the instrumental parameters (namely gain scale). Situation is improved a lot if we add a few further equations – the equations for the ADC calibration – into the system. This way we solve the ATM-RT problem and (correction to) ACD calibration simultaneously. For that reason, the 'grand' system is really large – we have $(NT + 2) \times NDC$ equations (after linearization of the NL RT equations), where '+2' is for the ambient- and hot-load ACD calibration equations, so it is overdetermined. We use the 'normal' equation approach, i.e., solving minimizer for the quadratic form of the least-squares. Still the normal square matrix is of the order $3 \times NT + 2 \times NDC$ and its direct inversion takes ages. However, it has been found that the normal matrix for the extended system (natural variables + instrumental parameters) can be conveniently inverted with knowledge of the (easily calculated) inversion of the original matrix for the natural variables (as used in `dWdB_Tbs()`) – the procedure is demonstrated in `extMatInv()` auxilliary routine at the beginning of this script. Lastly, we express the temperatures in kiloKelvins in order to have comparable sensitivities for dW and dT_B (which play role of matrix elements) and improve thus the condition number of the matrix.

3.2.2.16 dWdBdG2()

```

dWdBdG2 (
    vis,

```

```

    atmScan,
    tgtScan,
    myAtm,
    antID,
    spwList,
    T_tgt = -1.0e6,
    flaggedData = [],
    myMS = None,
    myMsmid = None,
    acdLoads = None)

```

The `dWdBdG()` extended by a bond to the expected time-averaged target brightness temperature `*T_tgt*` (in Kelvins).

Rationale: When observing the target, e.g. the Sun, we expect to obtain the same value of its time-averaged brightness temperature at all antennas. The time variations of `T_B` might be a bit different per antenna due to scintillation, but the time average over ~7minute scan should give the same value. If we use `dWdBdG()`, which calculates `T_B(t)` independently per antenna (and yet per XX/YY polarization), it is not very true: The `T_Bs` are somehow centered around the theoretical value calculated by Alissandrikis formula (see below in the code) but they have per-antenna scattering around that value with RMS ~200 K - see Fig. 11 in PR16. For that sake, we complete our grand-system of equations, described in `dWdBdG()`, yet by two additional equations (one per polarization), which enforce that the `<T_B(t)_XX/YY>` equals to the prescribed value of `*T_tgt*`, if it is supplied (the default value -1MK means "do not use it").

Implementation: The new equations make the grand-system yet more complicated, namely, the 'natural' part of the $(J^T \times J)$ matrix is no longer block-diagonal but still it is possible to invert the matrix relatively 'cheap' way. In addition to N-, G, and A- blocks, known from `dWdBdG()`, we introduce yet the M-block for the two additional "Mean" (over the array) equations. Similarly to the A-block equations, that enforce fulfilling of the ACD calibration equalities, and the 'relative strength', of how much we push to the ACD equations to be exactly fulfilled, is expressed by the factor 'calWeight', here we introduce a factor 'mWeight', which quantifies the relative strength of the `<T_B(t)^XX/YY>=T_tgt` in the grand-total system of equations.

Moreover:

1. Based on the results shown in the PR16, Fig. 11, we have introduced a bit modified antenna efficiencies, set by antenna type (CM, DA, and DV). The goal is to diminish apparent systematic (i.e. by antenna type) `<T_B(t)>` variations - the 'steps' in Fig. 11/PR16. The relative antenna efficiencies have been studied also in the XLS table `project22_25_Tb.ods`.
2. SQLD spws have significantly shorter integrations and therefore also received powers and gains (because $G=dP/dT$). In order to compensate this, we scale all equations by the gains, i.e., we express them in temperatures.
3. A new diagnostic tool has been added in order to auto-detect corrupted data. It is based on the error-vector per SPW/pol channel. The SPW:Pol data with extreme error are now reported, in future version they will be auto-flagged.

3.2.2.17 dWdBdG2_weights()

```

dWdBdG2_weights (
    vis,
    atmScan,
    tgtScan,
    myAtm,

```

```

    antID,
    spwList,
    T_tgt = -1.0e6,
    flaggedData = [],
    myMS = None,
    myMsmid = None,
    acdLoads = None)

```

Essentially the same as `dWdBdG2()`, but now we compose & solve the grand-system for entire range of `calWeights` (from 100 to 1000) and `mWeights` (100 - 10000) run in a loop.

Why we do not make the `dWdBdG2()` parametrized by `*cal/mWeight*` and run just the wrapping function, which calls `dWdBdG2()` in a loop with varying weights? The reason is technical: The most time-consuming operation is reading and processing the sci-SPW data. In this function we have a loop just over the grand-system solution, which represents less than 10% of CPU time spent in `dWdBdG2()`.

The goal is to find, how much the mean parameters `<PWV>` and `<TsunXX/YY>` are sensitive to the `calWeight/mWeight` selection. Note, that the fluctuations `dW`, `dtBxx`, and `dtbyy`, are quite robust - they do not significantly change even between various `dPWV` proxies like `dWdB()`, `dWdB_Tbs()`, `dWdBdG[2]()` or `dWdB_Tbs_NL()` and are, to a large extent, independent of their respective means.

Moreover, we are checking more in detail (with much finer `cal/mWeight` sampling over the selected interval), how the particular errors (Xi-squares) for the natural (`pwv`, `T_bXX/YY`), instrumental (`G`, `P_inst`), and array-mean (`T_tgt`) parts of the solution vector depend on `cal/mWeight`. Lastly, the number of iterations needed for the convergence is being watched.

Implementation: We vary always just one weight (`cal-` or `mWeight`): the second is set static to its optimised value found by variation. Selection is performed in the code - comment/uncomment selection of `*theWeight*` below. Weight range and sampling is controlled also manually by `wLow` and `wHigh` range and the number of samples code in `wItSize`.

Results: It has been found that the optimum value for `calWeight` lies between 100 and 300. At low values, the Xi^2 for instrumental variables is high and is improving with increasing `calWeight`. Nevertheless, around 200 the improvement of Xi^2 inst. slows down, while, at the same moment, the 2nd part of the error vector, the Xi^2 for natural variables, start rapidly increasing. The `<PWV>` and `<TsunXX/YY>` are pretty flat in this region of `calWeights`, so it is safe to select `catWeight=200`. This is actually the value eventually used in production version of `dWdBdG2()`.

3.2.2.18 dWdBdG_calWeights()

```

dWdBdG_calWeights (
    vis,
    atmScan,
    tgtScan,
    myAtm,
    antID,
    spwList,
    flaggedData = [],
    myMS = None,
    myMsmid = None,
    acdLoads = None)

```

--- OBSOLETE---

This is an older version of `dWdBdG2_weights()`, which supersedes this function.

3.2.2.19 dWdLs()

```
dWdLs (
    antID,
    myAtm,
    fullPWV = False)

*** Specific for 2022.1.00031CSV project! ***
```

Decompose the WVR raw variations for antenna `*antID*` in the solar scan into PWV variations and varying illumination of the WVRs by scattered solar irradiance `Lsun`. These two should be separable because we have 4 WVR channels (three of them usable, one saturated) and `dPWV` and `dLsun` have different frequency dependence - PWV-related signal increases towards the WV line center while the solar illumination (thinned by atmospheric absorption) decreases. The atmosphere tool `*myAtm*` must be set beforehand using `setAtmTool()` for the scan no 9 (the solar scan) and SPW 4 (WVR data). The boolean switch `*fullPWV*` controls, whether just variations around the means shall come out for the extracted PWV data (the default is false, which is useful for comparison with `dPWV` proxy) or full PWV value with the mean component added should come out. The `Lsun` contribution is always returned including its mean.

Reasoning: By comparison between our `dPWV` data and raw WVR signal it has been found that the DV antennas exhibit, in general, excellent correlation between those two quantities (WVR raw data and `dPWV` from science SPWs), while the DA antennas are obviously much worse. It looks like the reason is to be found in a different antenna construction - namely supports of the sub-reflector. These obstacles may lead to shadowing or, vice versa, additional reflection of the solar illumination to the WVR. Hence, the solar illumination needs to be handled as variable (cannot be fully subtracted in the stationary part).

We use formula (16) in PR15 to make the decomposition. Algorithm for numerical solution of Eq. (16) is the following:

1. Make scan-averaged (may be decomposed to off-source and on-source subscans) WVR channel temperatures `<Twvr>`. Since `dW(t)` and `dL(t)` variations should have (by definition) zero time average, three equations

$$\langle Twvr \rangle = T_{BS}(w) + \langle Lsun \rangle * tr(w)$$

should hold for WVR channels 1 - 3. We neglect the WVR channel 0, as it is saturated, exhibiting minimum variations (even the off-source -> on-source transition and back is not visible in the data). We vary the PWV content `w` such a way, that dispersion of three results for `<Lsun>` is minimized. The sky brightness `T_BS(w)` is approximated by `at.getTebbSkySpec()` using the `*myAtm*` model of the atmosphere, the transmissivity `tr(w)` is calculated from the same model. Newton-Raphson (N-R) iterations are used for minimizing the `<Lsun>` dispersion over the WVR channels.

2. Once optimized `<PWV>` and `<Lsun>` are found, we calculate the matrix coefficients and the RHS for the overdetermined system of equations (three equations for 3 usable WVR channels and 2 variables - `dW` and `dLsun`) and use the standard quadratic-form minimizing to find the solution (see the same procedure in `dWdB()` for details). The solutions `dW(t)` and `dLsun(t)` are checked for zero time-average: If a non-negligible value is found, the mean PWV as found in Step 1 is corrected

$$\langle PWV \rangle_{new} = \langle PWV \rangle_{old} + \langle dW(t) \rangle$$

and likewise the `<Lsun>`, and Step 2 is repeated until near-zero averages are reached.

3.2.2.20 dWdLs2()

```
dWdLs2 (
    myAtm,
    antID,
    tgtScan,
    flaggedWVRchannels = [],
    ssType = '',
    solIllumination = True,
    myMsmd = None,
    acdLoads = None)

*** Specific for 2022.1.00031CSV project! ***
```

An updated version of dWdLs(), which is improved by the following features:

- Included antenna efficiencies.
- Improved solar spectrum as in dWdB_Tbs()/dWdBdG().
- Improved solution of the system using Jacobians as in dWdB_Tbs()/dWdBdG().

Essentially, we may use the same basic equation for Tsrc+sky as we did for sci SPWs in dWdB_Tbs() - the scattered solar radiation (both forward or rear) contributes always by term of the form $r_{\text{eff}} \cdot \text{tr}(w(f)) \cdot T_{\text{sun}}(f)$, where r_{eff} is the fraction of solar illumination effectivelly scattered into WVR detector and this factor already contains the antenna efficiencies. Since r_{eff} is time dependent and unkonwn, we calculate just the product $(r_{\text{eff}}(t) \cdot T_{\text{sun0}}(t))$. Here, we absorb r_{eff} into the solar temperature and we designate the entire that product simply by $T_{\text{sun0}}(t)$. As in dWdBdG() we write $T_{\text{sun}}(f,t)$ by means of the normalised spectrum as $T(f,t) = T_{\text{sun0}}(t) \cdot SF(f)$. Hence, differences to dWdBdG() are not so big and they are contained in the following:

- We do not have instrumental vars (gains/Pinst), the normalised spectral gains are set to be uniform and equal to 1.0.
- There is no polarization separation as the WVR data are just Stokes I.
- We have only 4 WVR channels modelled by 4 ATM spws: No data->model SPW map.
- $T_{\text{sun0}}(t)$, representing just the part of solar radiation scaterred to WVR, is typicallyly 50 - 100 K only.

As explained in dWdLs(), this procedure is bound to the specific MS, where we had good luck and the WVRs are not saturated thanks to close-to-limb target observation.

3.2.2.21 dWdLs3()

```
dWdLs3 (
    vis,
    myAtm,
    antID,
    tgtScan,
    flaggedWVRchannels = [],
    ssType = '',
    solIllumination = True,
    myMsmd = None,
    acdLoads = None)
```

Not strictly specific for the project 2022.1.00023.CSV (Band 5 data), but for the other Band 5 datasets it gives nonsense for the solar on-source

subscan because of the WVR saturation (only the above mentioned project is 'lucky' as the WVR has been pointing slightly off-limb - see `dWdLs()`.)

An extended version of `dWdLs2()`: Similarly to `dWdBdG()` we now allow for limited corrections of the 'gains' and 'instrumental offsets' of WVR radiometers. The reason is similar to the sci-SPW data: The calibration is not exact and the gains may be wrong within a couple of percents. Combination with the science data can help to fix these little discrepancies.

Since the WVR data contain already ACD-calibrated temperatures, we have just a relative gains (close to 1.0) and offsets (close to 0.0) to become new instrumental variables of the system. The 'ACD' calibration equations thus take the form:

$$G \cdot T_{amb} + P_i = T_{amb}$$

$$G \cdot T_{hot} + P_i = T_{hot}$$

If left alone, they have solution $G=1.0$, $P_i=0.0$. But now they are used in combination with the science data in the grand-system like in `dWdBdG()`.

3.2.2.22 `dWdLs3_calWeight()`

```
dWdLs3_calWeight (
    myAtm,
    antID,
    tgtScan,
    flaggedWVRchannels = [],
    ssType = '',
    solIllumination = True,
    myMS = None,
    myMsmid = None,
    acdLoads = None)
```

--- OBSOLETE ---

Replaced/superseded by a new `dWdLs4_weights()` function.

3.2.2.23 `dWdLs4()`

```
dWdLs4 (
    vis,
    myAtm,
    antID,
    tgtScan,
    flaggedWVRchannels = [],
    ssType = '',
    solIllumination = True,
    myMsmid = None,
    acdLoads = None)
```

Improved version of `dWdLs3()`.

Inspired by `dWdBdG2()`, we now handle the situation of enforced zero solar illumination (setting `*solIllumination=False*`) more appropriately. The insofar procedure in `dWdLs3()` sets $T_{sun}=0.0$ hard and 'degrades' the T_B variations dB from a variable to a constant (we narrow the J_{wb} matrix by cutting the dB column) and it is set as a 'hard-wired solution' to 0.0.

In this new implementation we keep the `T_B` to be a variable, but we set its time-averaged value `Tsun` to 0 by implicit setting of parameter, corresponding to `*T_tgt*` in `dWdBdG2()`, to zero. In other words, we add the equations that force the time-averaged solar-illumination `Tsun` to zero, still keeping its small variations `dB`. As a result, the zero-illumination condition is not super-seded to other equations, but it has its relative strength in the entire system, expressed by internal (experimentally found) parameter `mWeight`.

As in `dWdBdG2()`, we have specific antenna-efficiencies setup per antenna type: This is another 'upgrade' to `dWdLs3()`. Unlike the `dWdBdG2()` we need not to introduce a 'natural' weight factor $WF=1/G$, because our 'raw' measured WVR signal is already expressed as temperature and all 4 WVR channels have the same weight (this is not true for sci SPWs, handled in `dWdBdG2()`, where SQLD spws have different integration time and, consequently, powers and gains).

3.2.2.24 dWdLs4_weights()

```
dWdLs4_weights (
    vis,
    myAtm,
    antID,
    tgtScan,
    flaggedWVRchannels = [],
    ssType = '',
    solIllumination = False,
    myMsmd = None,
    acdLoads = None)
```

Similar to `dBdWdG2_weights()`. We are varying the `*calWeight*` and `*mWeight*` parameters (one at a time, selected by commenting/uncommenting the line with `*theWeight*` below in the code) in order to find their optimized values, which somehow minimize the combined error (residuum) of natural, instrumental, and mean-array parts of our grand-system of equations.

Implementation: Basically, we use the `dWdLs4()`, where the main solution-finder iteration loop is nested in an outer loop of varying `calWeight/mWeight`. Here, some explaining comments are abbreviated - see `dWdLs4()` for full detail. Moreover, the plots of all time-dependent quantities are omitted.

3.2.2.25 extMatInv()

```
extMatInv (
    Ainv,
    Lb,
    Rb,
    LRc)
```

If we know inversion `*Ainv*` of a square regular matrix `AA`, we can efficiently calculate inversion the matrix `AA` extended by a lower belt `*Lb*`, right belt `*Rb*`, and a lower-right corner `*LRc*`, i.e. of a compound matrix $M = \begin{bmatrix} AA & Rb \\ Lb & LRc \end{bmatrix}$

This is used in `dWdBdG()` method below, where a large, block-tridagonal matrix (whose inversion is easy to compute) is extended by the (relatively narrow) belt-matrices at its right and low ends connected with the instrumental variables.

Returns: M^{-1} , i.e. the square matrix (2-d ndarray) that is inversion of the extended matrix $M = \begin{bmatrix} AA & Rb \\ Lb & LRc \end{bmatrix}$.

3.2.2.26 G_off_cal()

```
G_off_cal (
    vis,
    atmScan,
    tgtScan,
    ant,
    acSpw,
    myMS = None,
    myMsmd = None)
```

An experiment, how to transfer relative gain-drop between G* and G in the *sky* and *off* scans (due to switching the increased attenuation level on) from SQLD to the corresponding (occupying the same frequency domain) AC SPWs. Findings reached by experimentation with this method are then used in improved ACD calibration method for the Sun `acd_calTabs2()`.

In theory, the AC SPWs have low dynamic range for being able to measure `P_off` in the highly-attenuated off-Sun position (Shimojo et al. 2017, p. 10, right above Eq. (6)). Here, by means of linear proportionality between SQLD and AC powers ($P_{\text{sqld}} = K \cdot P_{\text{ac}} + Q$) for the same frequency domain, where the parameter `K` depends on the ratio $G_{\text{sqld}}/G_{\text{ac}}$, we evaluate the the derived `G_ac` and compare it with its (suspicious) value obtained by measuring `P_off` directly in the AC spws. The scaling factor between the `sqld`-to-`ac` transfer derived `G*` and the AC-`P_off` calculation is returned.

For the spectral-resolved AC spws we assume that the spectral gain normalized to unity remains constant during switching to the higher attenuation level, i.e., that only its scale is changing. Here we apply as much as exact calculation, which takes into account frequency dependence not only of this normalised gain, but also absorption/emission of the terrestrial atmosphere, and the normalized spectrum of the solar incident radiation. These factors enter into calculations of `P_ac` averaged over the resolved SPW as wieghts in the sums over the SPW channels. It has been experimentaly found, that the influence of solar spetral slope is negligible in comparison with the other two channel-dependent factors (normalised spectral gain and transmissivity). Hence, based on this research it is neglected in the improved `acd_calTabs2()` routine.

3.2.2.27 getMeteo()

```
getMeteo (
    vis)
```

Get time series of the basic meteo characteristics. They are not per antenna as there is just a single meteo station for entire array. Still we have much better time coverage than the `ASDM_CALATMOSPHERE` table. The latter has, seemingly, pr-antenna values but in fact they are the same for all antennas and differ only for different times. There are much less time samples in `ASDM_CALATMOSPHERE` than in `WEATHER` table.

We return the meteo characteristics as a time series in a map.

3.2.2.28 getMeteoForScan()

```
getMeteoForScan (
    vis,
    scan,
    myMsmd = None)
```

The meteo characteristics as extracted by `getMeteo()` form the MS `*vis*`, now averaged and interpolated to the center of the `*scan*`. The `msmd` tool `*myMmsmd*` may be conveniently supplied if it is already attached to `*vis*` - it improves performance.

Basically, it is intended as a more accurate replacement for `aU.getWeather()`. What is the difference between the two? The `aU.getWeather()` finds the meteo values 'bracketted' by the beginning and end of the scan and makes their average. This approach forgets even the recent history - if there is a meteo measurement point just a second before the scan begins, its value is neglected. The same holds on the 'future' end. In contrast, `getMeteoForScan()` samples the scan by 100 time samples, for each of them makes linear interpolation of the meteo values and eventually, this array is averaged. This approach takes into account the pre- and post-scan meteo measurements, too, with their weights naturally set by the time distance of the meteo record from the actual time-sample of the scan.

For the solar scans the tiny differences are not so important, the signal is dominated by the brightness temperature of the Sun, the T_{atm} correction is negligible. On the other hand, for the BP scans, the $(T_{\text{sky}} - T_{\text{atm}})$ expression, which stands in the sensitivity factor for $d\text{PWV}$, has a typical value about 3K, so difference in T_{atm} of 0.5K makes ~20% error!

3.2.2.29 getPhase()

```
getPhase (
    gTable,
    antID,
    spw)
```

Extract phase variations as a function of time from the gain-cal calibration table `*gTable*` for antenna with `*antID*` and in the spectral window `*spw*`.

3.2.2.30 getPhaseMS()

```
getPhaseMS (
    vis,
    antID,
    spw)
```

Instead of using the `gaincal()` solution contained in the `gainTable` (as `getPhase()` does), try to extract phases directly from the MS using the `DATA` and `MODEL` columns.

We are doing this trick just for the solar MS, so the `refAnt` and `tgtScan` parameters are hardwired at the beginning of this routine.

3.2.2.31 getPWVforAnt()

```
getPWVforAnt (
    vis,
    tgtScan,
    antID,
    myMmsmd = None)
```

In `wopacity()` we set averaged PWV over entire array and entire EB duration. For estimation of `T_atm` (which depends on atmospheric transmissivity) this is probably OK.

On the other hand, in `dWdB()` we have the matrix coefficients for (apparent) source brightness variations (dB) depending on the transmissivity, too, now calculated per-antenna. To that end, we extract the PWV values right before and right after the `*tgtScan*` for specific antenna `*antID*`. We use the `MS/ASDM_CALWVR` table to get the PWV per time and antenna. We find two PWV values that most closely (in time) embrace the `*tgtScan*` and return their average - value interpolated in time to the time-center of the `*tgtScan*`. Moreover, as a second value in the tuple, we return the linear slope of the PWV change during the `*tgtScan*`

See also `aU.readPWVFromMS()` and related functions for explanation, how we extract the PWV data.

3.2.2.32 `paregression()`

```
paregression (
    arrX,
    arrY)
```

Generalized/parametrized linear regression calculation between the arrays `arrX` and `arrY`. In addition to the standard lin. regression (handled, e.g., by `np.polyfit()`), here the values `y_i` of the `arrY` may be dependent not only on `x_i`, but also on the index `i` itself.

Mathematically, we are finding the best fit of the data in the form

$$y_i - t \cdot i = K \cdot x_i + Q$$

Rationale: If both arrays `X` and `Y` are time series (with regular cadence), the `Y` values may be dependent not only at `X`, but also on time (represented by the index `i`). Here we expect that there is - in addition to linear dependence on `X` - also a linear trend of `Y`-array in time. The goal is to optimize the parameter '`t`' of linear trend in time such a way, that we obtain the best linear correlation between the `X`-array and time-detrended `Y`-array.

Math behind the calculations here is described in PR12 (May 2025). Application in the context of this research is for correlation between phases and PWV variations as calculated in `dwXdpi_corrs()` below.

Input: `arrX`, `arrY` - 1D float np arrays (time series of the same size).

Returns: A dictionary with the values of

`tTrend`: Time-trend parameter.

`K`: Coefficient of linear correlation between `X` and time-detrended `Y`

`Q`: Offset for linear correlation between `X` and time-detrended `Y`.

`Xi2`: `Xi-square` - how good is the fit.

3.2.2.33 `phiFixCompare()`

```
phiFixCompare (
    vis,
    atmScan,
    tgtScan,
    refAnt,
    spwID,
    spwRes = -1)
```

Compare standard WVRgcal()-based correction of *BP-cal* phases with that based on our dPWV() proxy.

The dPWV() can be calculated either from SQLD or AC SPWs, the AC still having two sub-options: Channel-integrated or channel resolved SPW. In the latter case, dPWV_spwPart() is called internally, with integration only over the lower half of the SPW channels.

Which data are used, it is auto-detected from the *spwID* parameter. In case of the channel-integrated data (namely SQLDs) we need to specify the SPW with channel-resolved data in addition: This is used (i) for the calculation of (usually small) correction factor dTsky/df that implies from integration over the SPW bandwidth in case that tau_1 and Tsky are function of frequency, (ii) for comparison with raw and wvrgcal()-corrected phases; these are measured always in spec-resolved SPWs. The corresponding resolved SPW is given by the parameter *spwRes*. If spwID is a SQLD spw, then this *spwRes* MUST be set (we have no chance to deduce it). On the other hand, in AC-integrated data it can be inferred as spwID-1 and in case of AC-resolved data it is just directly the *spwID*.

First part of the job basically replicates the code in blindPhiFix() in order to see comparison between uncorrected phase and its improved counterpart fixed using our dPWV() proxy and the theoretical factor -25.0 (called "Method 1" in blindPhiFix())

The second part lies in taking the p-cal table (pre-Bandpass) with data corrected also by WVR calibration (the 'standard' p-cal table for the BP calibrator is NOT corrected for WVR) as the source. The following sequence of steps has been performed in order to obtain the WVR-corrected p-cal table mso+'.ap_pre_bandpass-wvrFix' in CASA shell:

```
@code{sh}
clearcal('uid___A002_X110fd13_X1099e.ms',field='1')

wvrgcal(vis=vis,caltable='wvr_gcal.W', segsource=True, sourceflag=['Sun'],
statsource='J1256-0547',refant='DV10',maxdistm=1.0,minnumants=3)

from casarecipes.almahelpers import tsysspwmap
tsysmap = tsysspwmap(vis = mso, tsystable = mso+'.tsys', tsysChanTol = 1)

gaincal(vis=mso,caltable=mso+'.ap_pre_bandpass-wvrFix',field='1',scan='5',
solint='1.92s',refant='DV10',calmode='p',
gaintable=[mso+'.antpos',mso+'.tsys','wvr_gcal.W'],
gainfield=['','1','1'],interp=['','linear,spline','nearest'],
spwmap=[[],tsysmap,[]])

@endcode
```

This way, antpos, Tsys, and WVR corrections has been applied on-the-fly.

This method cannot be used - even hypothetically - for the solar scans as we have no wvrgcaltable for the Sun to be compared with the dPWV() proxy.

3.2.2.34 phiFixCompare3()

```
phiFixCompare3 (
    vis,
    atmScan,
    tgtScan,
    refAnt,
    spwLow,
    spwRes = -1)
```

Like phiFixCompare(), but we use dWdP() - the method that takes into account

the dry-component-caused power fluctuations - for determining the PWV variations instead of `dPWV()` (which takes just a single SPW for the power fluctuations).

Since `dWdP()` works currently only for chan-integared SPWs, the `spwLow` may be just the lowest SQLD or AC-integrated SPW.

`dWdP()` omits the correction for slopes of $\langle T_{sky}(f) \rangle$ and $\tau_{u1}(f)$ so it does not need the `spwRes`. However, we still need it for the raw and `wvrgcal()`-fixed phases to be compared with our corrections.

3.2.2.35 `plotPhases()`

```
plotPhases (
    vis,
    gTable,
    spw)
```

Extract phases from the table `*gTable*` for a given `*spw*` using `getPhase()` and make series of plots (as PNG files) for all antennas contained in the current MS `*vis*`.

3.2.2.36 `pwv4scans()`

```
pwv4scans (
    vis,
    solWVR = False)
```

Calculate PWV(t) from the raw WVR data using `dWdLs2()/dWdLs3()` in four consecutive scans around the solar scan - start with leading phase-scan and finish with the trailing phase-cal scan.

For comparison, calculate the sciSPW-based PWV(t) proxy by `dWdBdG()` for the solar on-source subscans, too.

WVRs in the solar on-source subscan are disabled by default (`*solWVR* = F`), because, in general, they are saturated. May be set to T for the specific MS: Band 5 data of 2022.1.00023.CSV (WVR are ~10 arcmin off-limb)

Save WVR + sciSPW data to `'./pwv4scans.npy'` for 12m antennas and sciSPW data extra for all antennas to `'./PWV_TB_GP_DATA-antEff.npy'`. Both files are open in 'append mode, the recent-most results are the last (the older may be used for comparison of different approaches, e.g. `dWdLs2()` vs. `dWdLs3()`, solar contamination in off-the-Sun scans allowed or not, etc.)

3.2.2.37 `pwv4ScansCompare()`

```
pwv4ScansCompare (
    vis)
```

More sophisticated comarison between PWV as determined from sci SPWs via `dWdBdG()` method with the value calculated from the raw WVR temperatures via `dWdLs3()`.

We compare for all 12m antennas (the 9m CM antennas do not have WVRs) and for sake of efficiency we use pre-calculated PWVs stored in `'./pwv4scans.npy'` via `pwv4scans()` and re-read by `pwv4scansRestore()`.

3.2.2.38 pwv4scansRestore()

```
pwv4scansRestore (
    vis,
    hasSolWVR = True,
    runID = 100000)
```

Extract data saved by `pwv4scans()`. The parameter `*runID*` controls, which run of `pwv4scans()` should be extracted (`pwv4scans()` appends the new results to the end of existing files). The default value of 100000 means to take the recent-most data. The `*runID*` starts counting from 1.

Parameter `*vis*` is here for determining the antenna array.

The flag `*hasSolWVR*` (default = T) indicates, whether the data contain also the on-source subscan WVR data.

3.2.2.39 pwvCalc()

```
pwvCalc (
    vis,
    atmTool,
    scan,
    antID)
```

Calculate PWV for the `*scan*` and `*antID*` using raw WVR data. We use linearized formula (13) in the PR15 in an iterative way to converge to $dW=w-w_0$ to be zero – basically we use the Newton-Raphson (N-R) method. The atmosphere tool `*atmTool*` must be set beforehand via `setAtmTool()` for SPW 4. The sky brightness temperature `Tbs` is calculated from the ATM model using `at.getTebbSky()` method. We have also tried `at.getTrjSky()` as an alternative and compared the results with the (rare) SYSCAL PWVS via `pwvCalcVerify()` – see the comment on comparison results there.

Returns: PWV obtained by averaging the partial results from WVR channels 1-3 and its `stddev` in a tuple. We skip channel 0 (the line core) as it gives outlied data from the rest (frequently also NaNs). Essentially, for the ‘solar observing conditions’ with PWV typically > 3.0 it is, anyway, mostly saturated and, hence, insensitive to smaller PWV variations.

3.2.2.40 pwvCalcVerify()

```
pwvCalcVerify ()
```

Try to ‘verify’ our PWV calculation in `pwvCalc()` against the ALMA-calculated values in `ASDM_CALWVR` table, accessed by `getPWVforAnt()`.

*** Specific for this MS! ***

We calculate the PWVs by both methods for all antennas and all non-solar scans and make a scatter plot. We can expect some differences as our `pwvCalc()` method uses WVR temperatures over a scan, while the system-calculated PWVs are a bit opaque – neither the method of their calculation is clear (nothing found in literature except they use the ATM model) nor the strange times at which we have their values in `ASDM_CALWVR` table are explained. The goal is to see, whether the discrepancies are small and do not have systematic trends.

Results: The correlation works basically well, but there are, indeed, systematic differences: If `at.getTebbSky()` is used for sky brightness estimation (to be interpreted as `Twvr` values in WVR raw data), our `pwvCalc()` method gives systematically underestimated PWV values. When `at.getTrjSky()` is used, then our PWVs are systematically over-estimated. Detailed results are in `./wvr_raw/pwv-SYS_vs_ours` directory.

3.2.2.41 pwvCompare()

```
pwvCompare ()
```

```
*** So far just for the solar MS project 2022.1.00031.CSV, Band 5!!! ***
```

Compare systematically (for all antennas) atmospheric phase delays as predicted from:

1. Our dPWV() proxy (calculated from signal of science SPWs),
2. wvrgcal()-produced caltable (calculated from the hi-cadence WVR data).

We can do it just for the BP calibrator, so no free source parameter in this case. We are using the channel-integrated AC windows (not SQLD) in dPWV().

Creates two-panel plot (left for XX, right for YY) for each antenna, having our dPWV() prediction in red and wvrgcal() calculation in blue.

3.2.2.42 rebin()

```
rebin (
    array1D,
    downSampleFactor)
```

This is a general-purpose helper analogical to the IDL's routine `rebin()`. It down-samples 1D-array `*array1D*` by the `*downSampleFactor*`. The dimension of the `array1D` must be evenly divisible by the `downSampleFactor`! If not, the original (non-resampled) array is returned and a warning issued. In such a case, it is advisable to use more complex `rebin2()` method.

Rationale for utilization here: We are frequently trying to find correlations between quantities like phase variations and our model PWV-proxy, and these need not to have the same sampling rate (typically our PWV-proxy from auto-correlations has cadence 0.192s; from the SQLD data yet shorter, by an integer factor of 12, while the BP-calibrator phases are solved at interval of 1.92 sec).

3.2.2.43 rebin2()

```
rebin2 (
    array1D,
    origX,
    downSampledX)
```

The `rebin()` method generalized for the case that down-sampled array elements cannot be simply calculated as averages over `*downSampleFactor*` elements of the original array, because the dimensions do not agree.

We submit three argument: `*array1D*` are the 1D data (e.g., recorded powers in our SQLD spw), `*origX*` is the x-grid of the original data (i.e., the times at which the SQLD powers are measured in the above example), and the `*downSampledX*` is the new, downsampled x-grid (in the used example it is typically the lower-cadence time-axis for the corresponding AC resolved spw).

We first define a new high-cadence x-axis with a gridding that is (i) as much close to the original x-grid as possible, and, at the same moment (ii) the new grid represents an even division of the low-cadence grid. Then we can already call our standard `rebin()` method.

Rationale: Like with `rebin()`. This more generalised routine has been introduced namely because of the non-comensurable (no even division) time meshes for SQLD and the AC spws in some MSs.

3.2.2.44 restoreSol_dWdB()

```
restoreSol_dWdB (  
    vis,  
    filePath,  
    runID = 100000)
```

Load the results of dWdB() & friends for entire ALMA array from a file and store it as a map with antenna as its key. Restoring works as a reverse routine to both saveSol_dWdB() and saveSol_dWdB_Tbs()

The **filePath** points to the NPY file with data.

The **runID** parameter (starting from 1) controls, which of the appended records should be read.

We need **vis** in order to determine the antenna array (Todo: Think about re-implementation of the corresponding save_() routines so that they store antenna array at the beginning of the file).

It is a solar-specific routine.

See also saveSol_dWdB() / saveSol_dWdB_Tbs().

3.2.2.45 saveSol_dWdB()

```
saveSol_dWdB (  
    vis,  
    atmScan,  
    tgtScan,  
    spwList)
```

Create ATM model framework and then run dWdB() for all antennas in the array. Save the complete results (dW, dB, matrix, and RHS) into a binary NPY file.

See np.save() and np.load().

3.2.2.46 saveSol_dWdB_Tbs()

```
saveSol_dWdB_Tbs (  
    vis,  
    atmScan,  
    tgtScan,  
    spwList)
```

As saveSol_dWdB(), but now for data produced by either dWdB_Tbs() or dWdBdG().

The main difference is in the ATM model: While dWdB() requires static opacities (i.e. arrays) pre-calculated by setAtmosSciSPWs(), the dWdB_Tbs() and dWdBdG() routines require the full ATM tool (attool) as their parameter, here set and coupled to actual meteo situation via setAtmTool4SPWs().

3.2.2.47 setAtmosSciSPWs()

```
setAtmosSciSPWs (
    vis,
    tgtScan,
    spwList)
```

Calculate the ATM profiles using `wopacity()` for all channel-resolved sci SPWs. Then, using the `spw-mapping` attribute the appropriate ATM profile to every science SPW (chan-resolved and integrated ACs and SQLDs) in the `*spwList*`. Use `*tgtScan*` for determining the appropriate ATM model - we have scan-dependent meteo parameters and airmass.

Used as a single-shot function in the methods that iterate over the antenna array, like `saveSol_dWdB()`: We do initiation of the atm model for all spws just once.

3.2.2.48 setAtmTool()

```
setAtmTool (
    myAtm,
    vis,
    spw,
    scan,
    myMsm = None,
    wvHeightScale = 1.5,
    tempDropPerKm = -5.6)
```

Set the atmosphere tool (`casatools::at`) `*myAtm*` for later calculation of the opacity (wet & dry component) as a function of frequencies for the single spectral window `*spw*`. The `*spw*` may be either one of the AC-resolved sci SPWs or the SPW=4 for the WVR data. The `attool` effectively represents CASA interface to the underlying terrestrial-atmosphere radiative-transfer model ATM by J.R. Pardo.

Input:

The `spw` setup and the atmosphere (meteo) profile model are extracted from the input measurement set `*vis*`. Parameter `*scan*` determines (i) the time-range, for which we seek the meteo parameters, and (ii) the target object and its elevation (H) - we need it for the airmass ($AM = 1/\sin(H)$) calculation. The WV height scale and temperature drop per 1km are set to their default values, which give the best fit between our PWVs as calculated by `pwvValc()` from the raw WVR data and the SYSCAL PWV values for the 2023 Band 5 data - see `pwvCalcVerify()` for details.

Output:

The `atmTool` `*myAtm*` gets modified in this routine so that it is ready for further use in, e.g., `wopacity()`. With this flexibility, we can later force the PWV value to be set by a user.

Our `attool` interface has been loosely inspired by `aU.plotAtmosphere()`. The `aU.plotAtmosphere()` utility, however, uses `aU.getMedianPWV(vis)` even though its `*pwv*` arg is set, if the `vis` argument is present, so the `pwv` argument is actually ignored there. Moreover, `aU.plotAtmosphere()` returns just the representative-frequency (scalar) values; we need the frequency-dependent opacity values (see `wopacity()`).

See `aU::plotAtmosphere()` and `casatools::at` for details.

3.2.2.49 setAtmTool4SPWs()

```
setAtmTool4SPWs (
    myAtm,
    vis,
    scan,
    myMsmc = None,
    wvHeightScale = 1.5,
    tempDropPerKm = -5.6)
```

Similar to setAtmTool() but now it makes the setup for the 4 channel-resolved AC SPWS (i.e. no WVR spw possible to be set here).

As result, *myAtm* has 4 spws 0 - 3 mapped from the sci SPWs 5,7,9, and 11. For the temperature, pressure and humidity profile we use the same setup as in setAtmTool().

3.2.2.50 smooth()

```
smooth (
    array1D,
    kernel)
```

A simple smoothing of *array1D* performed by a convolution with the much shorter 1-D array *kernel*. Kernel may be a box or a triangle function or whatever similar with the finite (and rather short, in comparison with the array1D length) support.

In order to handle the edge effects, we apply 'mirror boundary condition' to array1D prior smoothing - we extend the original array at both edges prior the convolution. This prevents wild behaviour at the edges (so the edges do not affect, e.g., plotting y-range), but the edge data carry inherently lower/degraded information and must be taken cautiously.

3.2.2.51 sol_offPos()

```
sol_offPos (
    vis,
    tgtScan,
    refAnt)
```

Trying to find the minimum off-source separation from the Sun so that WVRs are not affected by direct solar illumination.

We calculate, how far from the Sun (the disc center) we are when pointing OFF_SOURCE during the solar scan. This is the position taken at the first and last subscan of the solar scan and its purpose is to be basically sky/atm measurement, but now with the attenuator settings used for the solar on-source observation - see the G-star calculation in acd_calTabs(). It seems that the position of the first sub-scan is identical with those taken in the preceding ATM scan.

3.2.2.52 sol_PhaseFix()

```
sol_PhaseFix (
    antID,
    spw,
    pwvMap)
```

Compare the suggested phase-fix based on the full-PWV calculations as performed by `dWdBdG()`, with the linear interpolation of the phase-inf solution obtained from the adjacent (bracketing) phase-cal scans. We would like to see whether the phase-inf interpolated solution has at least some significance or whether it is totally 'out', e.g. the phase wraps over 2π in reality.

Results:

- (1) The trend in PWV-based phases (Sun) roughly corresponds to the linear interpolated phi-Cal solution.
- (2) There is, however, an offset that may be even as large as 720 degs (i.e. 2 lambdas in path).
- (3) A bit worrisome is the difference between XX and YY polarizations in phi-Cal solution.
- (4) Albeit the trend is corresponding between both, the PWV-based phases exhibit variation over the scan that are easily bigger than 360 deg.

3.2.2.53 sol_pwv2selfCal()

```
sol_pwv2selfCal (
    antID,
    spw)
```

Comparison of PWV-based phase calibration of the Sun with self-cal gaintable solution. Because the self-cal phases are already apriori corrected by the linear trend (via `applycal()` of phase-cal gaintable), we plot also the phase-cal linearly-interpolated solution.

3.2.2.54 sol_pwvGcal()

```
sol_pwvGcal (
    pwvMap,
    vis,
    refAnt,
    pwvCalTableName = 'sol_pwvCalTable.W')
```

Create the phase calibration table for the Sun in the same style as `wvrgcal()` makes it for non-solar objects.

We use the full-PWV time series per antenna as found by `dWdBdG()` routine and stored in `*pwvMap*`.

@todo: Generalize for the case we have multiple solar scans.

3.2.2.55 solTgtPos()

```
solTgtPos (
    vis,
    tgtScan,
    refAnt)
```

Similar to `sol_offPos()`, but now we produce a graphic output of the actual target-pointing relative to the solar disc center (i.e., angular separation of those two) as a function of time over entire `*tgtScan*` for a 'reference' antenna (the result should be the same for all) `*refAnt*`.

3.2.2.56 sqld_P0()

```
sqld_P0 (
    vis,
    ant,
    spw,
    myMS = None,
    myMsmd = None)
```

Find the P-zero offset for SQLD detector at antenna `*ant*`.
The `*spw*` parameter designates the spw for `*standard*` SQLD ATM scans `[0,1,2,3]` and maps to the spws for the scan no. 1 (the 'zero' scan) `[13,14,15,16]` in an obvious way `[0,1,2,3] -> [13,14,15,16]`.

Returns: An array of shape `2 x 1`, the first dimension being polarization.

See also `acd_calTabs()`.

3.2.2.57 TbStats()

```
TbStats (
    resCG,
    resVG)
```

The mean brightness temperature of the Sun, as derived from `dWdB_Tbs()` and its generalization `dWdBdG()`, should be essentially the same for all antennas - we expect (like in `dB_sum()` above) that the 'scintillation' burst/drops because of turbulence focusing/de-focusing effects are in average zero over the scan.

Here we make staticstics over all antennas for two sets of the results: One obtained by `dWdB_Tbs()` with constant gains determined by ACD calibration, and the other, resulting from `dWdBdG()`, where we allow for the gain correction via minimizing the grand-system (science + calibration measurements together). The parameters `*resCG*` and `*resVG*` are the maps/dicts obtained as a result of `restoreSol_dWdB()` for the constant-gain and variable-gain solutions, respectively.

3.2.2.58 tgt_pow()

```

tgt_pow (
    vis,
    tgtScan,
    ant,
    spw,
    myMS = None,
    myMsmd = None,
    subScans = 'on')

```

Extract total-power data in **spw** for target defined by the **tgtScan** number at the antenna **ant**. The **myMS** and **myMsmd** tools attached to **vis** may be supplied for speed-up. The **subScan** is relevant just for the solar target and may be empty (entire scan) or 'on'/'off' for the on-source (the default) and off-source sub-scans selection.

Target, antenna, and spw params MUST be int, not strings - see `acd_calTabs()`.

Performs MS selection in order to extract the raw auto-correlations (AC) or SQLD data (depending on the SPW selected). We get the time-series of the **total powers** (either AC or SQLD); if the SPW is channel-resolved (spws 5, 7, 9, and 11), we have even time series of the spectra. If the **tgtScan** is a solar scan, we have to select just the 'internal' subscans, omitting the very first and very last sub-scans, as they are taken in the off-source position for calibration (see `acd_calTabs()` for detail).

Note: If intended to be used with `acd_calTabs()`, the ATM scan in `acd_calTabs()` should as much close as possible precede the target scan in `tgt_pow()`. This is the case, for example, with the arguments **atmScan** and **tgtScan** in `tgt_Tb()`.

Returns: A tuple with two components.

The 1st component: Times for each amplitude as MJD - an array of size `nIntegs`. The 2nd component: Array of powers in the form `2 x nChans x nIntegs`. I.e., `2 x 1 x nIntegs` for the SQLD and chan-integrated SPWs, and `2 x 128 x nIntegs` for channel-resolved SPWs. The first dim (2) means polarization (XX, YY).

3.2.2.59 tgt_Tb()

```

tgt_Tb (
    vis,
    atmScan,
    tgtScan,
    antID,
    spwID,
    myMS = None,
    myMsmd = None,
    acdLoads = None,
    myAtm = None)

```

Calculate ACD-calibrated 'sky' brightness temperatures from the measured total-power (AC or SQLD) data. The 'sky' Tb is a sum of the atmosphere contribution and the attenuated (by passage through the same atmosphere) signal of the source (target) defined by the **tgtScan** parameter. The **atmScan** should be the nearest ATM scan preceding to the target scan.

Return: A tuple of three arrays. The first one is a time-axis, the 1-D array of size `nInteg`, merely reproduction of the time-axis (as MJD) from internally called `tgt_pow()`. The other two arrays - `Txx` and `Tyy`, are each of the shape `nChans x nIntegs`, `nChans` is either 1 (SQLD and chan-integrated AC SPWs) or 128 (chan-resolved AC SPWs) in case of Solar Band 5 hi-res MS.

3.2.2.60 tgtPow_crossPlots()

```

tgtPow_crossPlots (
    vis,
    ant,
    tgtScan,
    spw1,
    spw2)

```

Compare the power variations (mean-subtracted powers) for target in **tgtScan** in two different SPWs **spw1** and **spw2** (should be of the same type: either AC integrated or SQLDs - not checked!) for a given antenna **ant**. Specifically, we do scatter plot of power variations in **spw2** vs. powers in **spw1**.

Idea: Do we see dry-component or source intrinsic contribution to the power variations in the SPWs further from the WV line center? The ultimate question is, which of the approaches to the dPWV proxy shall be used - whether dPWV() / dPWV_spwPart() or dWdB(), which counts with the (apparent) variability of the source, too (and may account for adding the dry-component).

Result: When used for the BP calibrator, we see just a single linear correlation of the signal between the **spw1** and **spw2**. Its slope more or less corresponds to theory, given the big relative uncertainty in Tsky-Tatm. For the Sun, however, we see a sort of two timescales, each with a different slope. It has been found that the slower-variations linear slope corresponds to the ratio of matrix coefficients for dW, while the faster one for dB, when decomposing via dWdB() method is applied. See dWdB() for details.

3.2.2.61 unwrap2()

```

unwrap2 (
    phase)

```

Sometimes we have phase jumps exactly by 180 deg. These seem to be artificial but not cured by the standard np.unwrap(). This method aims to remove also these residuals that are well less the 2PI (which are cured by np.unwrap()).

Input: phase in rads - 1-D numpy ndarray

Output: phase_unwrap in rads - 1-D numpy ndarray with the same dimensions.

How does it work: We differentiate the original array, fix the high jumps (positive and negative extra) in the array of differences and then 're-integrate' the corrected difference series back via the cummulative sum. In order to skip seekeing and evaluating the 'super-high' jumps (2PI and more) we call first the standard np.unwrap() prior to our residual-jump treatment.

3.2.2.62 wopacity()

```

wopacity (
    vis,
    spw,
    scan,
    userPWV = -1.0)

```

Uses the atmosphere tool (`casatools::at`) for calculation of the opacity (wet & dry component) as a function of frequencies for the science spectral window `*swp*`. We preset the `atTool` via `setAtmTool()`.

Return value: dictionary with the following components:
`airMass`: $1/\sin(h)$ for the object observed in the scan `*scan*`
`tau_l`: 'Wet' component of the zenith opacity for PWV=1.0 as a function of frequency channels (1d-array).
`tau_d`: Dry component of the zenith opacity as function of channels.
`tau_w`: Wet zenith opacity as a function of channel for the actual PWV.
`T_atm`: Effective temperature of the atmosphere (used in $(1-\text{trans}) \cdot T_{\text{atm}}$ expr.).

3.2.2.63 wrapBack()

```
wrapBack (
    phases)
```

An 'inverse' function to `np.unwrap()`

3.2.2.64 wv2path()

```
wv2path (
    vis,
    spw,
    scan,
    pwv)
```

We use the factor -25.0 for conversion between `dW` and `dPhi`, as it implies from the Richard Hill's original draft. (RH uses effective refractive index `r_eff=6.7`, independent on frequency. When the optical path is recalculated to phase delay for average `lambda ~1.5 mm`, we arrive to factor -25.0).

This routine verifies, whether this assumption is correct and what are its limitations. We use ATM model and its `get***Path()` routines.

Result: Indeed, the non-dispersive part is constant over SPW 5 and its value is $\sim 6.7 \times 10^{-3}$ m(!) per 1 mm of PWV. The dispersive part is very slightly changing around its average value of 1.0×10^{-4} m per 1 mm PWV and thus, represents a small correction. Both paths depend linearly on `pwv`, so their effective (per 1 mm PWV) value does not depend on the `*pwv*` parameter. On the other hand, they slightly depend on the airmass (i.e. on the `*scan*` parameter). With rising SPW the total (disp.+ non-dispersive) path per 1mm rises (up to $\sim 6.9 \times 10^{-3}$ m per 1 mm PWV). So, for approximate calculations it is fine to use the factor of approx -25 (actually -27). However, for real application, where we use correct central frequencies for each SPW, it is better to calculate the phase-delays directly from the ATM model.

3.2.2.65 wvr_dataRaw()

```
wvr_dataRaw (
    vis,
    tgtScan,
    antID,
    subScans = '')
```

Extract the raw WVR data from SPW 4 for the antenna *antID*

The *subScans* parameter can be either an empty string (the default), or have one of the "on" or "off" values. If empty, entire scan is used, otherwise, just on-source or off-source subscans are involved, respectively.

The four WVR channels (temperatures) are extracted as a function of time. Return value: A tuple of two ndarrays - time (1d) as MJDsecs of dimension nTimeSamples, WVR (2d) of dimensions 4xnTimeSamples.

3.2.2.66 wvrPhase()

```
wvrPhase (  
    vis,  
    antID,  
    spw)
```

Extract WVR-predicted phase delays for the PB-calibrator as a function of time for the WVR gain table that has been created by the call

```
@code{sh}  
wvrgcal(vis=vis,caltable='wvr_gcal.W', segsource=True, sourceflag=['Sun'],  
statsource='J1256-0547',refant='DV10',maxdistm=1.0,minnumants=3)  
@endcode
```

Basically, very similar to the getPhase() above, just we specify also the field inside the table-selection.

The getPhase() method is called as well inside - the only purpose is to get easily the time-range for the BP scan and refine the selection of the WVR-gcal table rows.

The last two args to wvrgcal() (added after PM12 in may 2025) prevents wvrgcal() from making caltables in antennas that do not have the WVR device (namely 7m antennas) or have it broken (without those switches wvrgcal() produces a 'spurious' table for non-working WVR by interpolation of the neighborhood antennas).

Class Documentation

Public Member Functions

- ## Public Attributes

- #### 4.1.1 Detailed Description

Attributes:

```
fluxCalField (string): Field for intent CALIBRATE_FLUX.
phaseCalField (string): Field for intent CALIBRATE_PHASE.
bpassCalField (string): Field for intent CALIBRATE_BANDPASS.
ampCalField (string): Field to be actually used in setjy() - may be
                        bpassCalField or fluxCalField.
bpassCalScan (string): Which scan has been used for B-pass calibration.
ampCalSrcName (string): Source-name for actual flux calibrator.
```

4.1.2 Constructor & Destructor Documentation

4.1.2.1 `__init__()`

```
SolSG.calibrators.__init__ (  
    self,  
    obsMsmmd)
```

Initialize the calibrator attributes

4.1.3 Member Function Documentation

4.1.3.1 `allCalsLst()`

```
SolSG.calibrators.allCalsLst (  
    self)
```

All calibrators (a unique set) as a Python list (of strings).
To be used in a loop over `applycal(field=allCals[i])`.

4.1.3.2 `allCalsStr()`

```
SolSG.calibrators.allCalsStr (  
    self)
```

All calibrators (a unique set) as a single comma-separated string.
To be used in `gaincal(field=allCals)`

4.1.3.3 `fluxCalQualityMetrics()`

```
SolSG.calibrators.fluxCalQualityMetrics (  
    fCalField)
```

Returns a single float number as a measure of quality for the flux calibrator. The quality is based on (1) flux density and, (2) time separation between the last measurements in the ALMA calibrator database and the date of observations.

Rationale: Quite often we have declared (primary) flux calibrator (with intent `CALIBRATE_FLUX`) in solar science data which, however, is either weak or its measurement in database is old. Hence, the band-pass calibrator is frequently use as a primary calibrator: The function `sol_calflux2()` even assumes this as a standard.

This method should calculate a score (the higher, the better) to help to decide whcih calibrator should be used for `setjy()` and subsequent `fluxscale()`

4.1.3.4 printInfo()

```
SolSG.calibrators.printInfo (
    self)
```

Print info on calibrators.

The documentation for this class was generated from the following file:

- SolSG.py

4.2 SolSG.observation Class Reference

Public Member Functions

- `__init__` (self, asdmName)
- `rfStr` (self)
- `printInfo` (self)

Public Attributes

- `theSun` = `scienceTarget`(obsMsmd)
- `theCalibrators` = `calibrators`(obsMsmd)
- `band` = `aU.bandforspw`(refSPW, mymsmd=obsMsmd)
- tuple `refFreq` = (`obsMsmd.reffreq`(refSPW))["m0"]
- `obsDate`

4.2.1 Detailed Description

Basic minimum data about the observation that are necessary for data reduction & imaging. Contains the children objects `scienceTarget` and `calibrators` plus a few further service data.

Attributes:

```
theSun (class scienceTarget): Encapsulated properties of the target
theCalibrators (class calibrators): Encapsulate properties of calibrators.
band (int): Observing band.
refFreq (CASA quanta): Representing (reference) observing frequency.
obsDate (string): Date of observation in YYYY-MM-DD format.
```

4.2.2 Member Function Documentation

4.2.2.1 printInfo()

```
SolSG.observation.printInfo (
    self)
```

Prints basic info about the observation.

4.2.2.2 rfStr()

```
SolSG.observation.rfStr (  
    self)
```

Returns the reference frequency as a string (value + unit).

4.2.3 Member Data Documentation

4.2.3.1 obsDate

```
SolSG.observation.obsDate
```

Initial value:

```
= qatool().time(self.theSun.refTime,  
                form=["fits","no_time"])[0]
```

The documentation for this class was generated from the following file:

- SolSG.py

4.3 SolSG.scienceTarget Class Reference

Public Member Functions

- [__init__](#) (self, obsMsmd)
- [isMosaic](#) (self)
- [targetFields](#) (self)
- [printInfo](#) (self)

Public Attributes

- **allFields** = allFieldsArray.tolist()
- **sciFields** = sciFieldsArray.tolist()
- **refTime** = qatool().quantity(sciScanTimes[0], 's')
- **refField** = fieldRef
- **fieldsDirections** = dict()

4.3.1 Detailed Description

Attributes & methods for the science target (the Sun).

Attributes:

`allFields` (list of ints): List of all target fields. May contain purely OFF_SOURCE field 0 that is attached to ephemeris and serves as a reference to which the science fields in a mosaic have offsets in the FIELD table.

`sciFields` (list of ints): Just the science fields (the intent OBSERVE_TARGET#ON_SOURCE)

`refField` (int): The reference field attached to ephemeris.

`refTime` (CASA ganta): The time when the first science field started to be scanned. For exactly this time the ephemeris is interpolated and RA/DEC found by this interpolation are set to the reference field. It is also the same date/time that is later written as DATE_OBS keyword in the header of final product FITS image - can be used for transformation to helioprojective X,Y.

`fieldsDirections` (dictionary of measures): Ra/Dec attributed to each field at the reference time.

Methods:

`bool isMosaic()`: Returns T if the target is a mosaic, F if single-pointing.

`string targetFields()`: Returns the all target fields as a CASA-string.

`void printInfo()`: Prints information about science target to terminal.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 __init__()

```
SolSG.scienceTarget.__init__ (
    self,
    obsMsmd)
```

Constructor.
@param obsMsmd: MS meta-data.

4.3.3 Member Function Documentation

4.3.3.1 isMosaic()

```
SolSG.scienceTarget.isMosaic (
    self)
```

Returns True if the target is a mosaic, False if single-pointing.

4.3.3.2 printInfo()

```
SolSG.scienceTarget.printInfo (
    self)
```

Print info/summary about the target for the user.

4.3.3.3 targetFields()

```
SolSG.scienceTarget.targetFields (
    self)
```

Returns a string to be passed as a 'field' argument in applycal() and tclean().

Note: We have to include the field 0 (in case of mosaic) whether it is associated with a science scan or not. The 'phantom' field=0, albeit may not even be present in the mosaic as a science target field (MS.split.cal contains no data associated with this field), it is still in the FIELD table and THIS is the field that is clipped to ephemeris (even if it is purely OBSERVE_TARGET#OFF_SOURCE intent!) It is usually located in the mosaic center, but need not be.

The documentation for this class was generated from the following file:

- SolSG.py

4.4 SolSG.stuffForSolarDataReduction Class Reference

Public Member Functions

- [generateSolCalScript](#) (self, asdmName, doQA2=True)

4.4.1 Detailed Description

Encapsulation for Solar ALMA Data Reduction Utils

4.4.2 Member Function Documentation

4.4.2.1 generateSolCalScript()

```
SolSG.stuffForSolarDataReduction.generateSolCalScript (
    self,
    asdmName,
    doQA2 = True)
```

Generates the script for solar INT data calibration (single EB)

Implementation is very simple:

It reads the static template (the 'skeleton', which remains constant for all solar INT EBs) and adds/replaces the lines with the EB-specific code. Result is written into the 'standard' solar calibration script.

Inputs:

asdmName (string): Name of the ASDM to reduce

doQA2 (bool): Generate the script for the QA2 analyst (True) or for the archive (False).

The documentation for this class was generated from the following file:

- SolSG.py

Index

- `__init__`
 - `SolSG.calibrators`, [42](#)
 - `SolSG.scienceTarget`, [45](#)
- `acd_caltab_plots`
 - `sol_wvgcal`, [9](#)
- `acd_calTabs`
 - `sol_wvgcal`, [10](#)
- `acd_calTabs2`
 - `sol_wvgcal`, [10](#)
- `allCalsLst`
 - `SolSG.calibrators`, [42](#)
- `allCalsStr`
 - `SolSG.calibrators`, [42](#)
- `blindPhiFix`
 - `sol_wvgcal`, [11](#)
- `dB_sum`
 - `sol_wvgcal`, [11](#)
- `dPWV`
 - `sol_wvgcal`, [11](#)
- `dPWV_spwPart`
 - `sol_wvgcal`, [12](#)
- `dpwvXphi_corrs`
 - `sol_wvgcal`, [12](#)
- `dpwvXphi_plots`
 - `sol_wvgcal`, [13](#)
- `dpwvXsolWVR_raw`
 - `sol_wvgcal`, [13](#)
- `dpwvXwvr_corrs`
 - `sol_wvgcal`, [13](#)
- `dWdB`
 - `sol_wvgcal`, [14](#)
- `dWdB_Tbs`
 - `sol_wvgcal`, [15](#)
- `dWdB_Tbs_NL`
 - `sol_wvgcal`, [16](#)
- `dWdBdG`
 - `sol_wvgcal`, [16](#)
- `dWdBdG2`
 - `sol_wvgcal`, [17](#)
- `dWdBdG2_weights`
 - `sol_wvgcal`, [18](#)
- `dWdBdG_calWeights`
 - `sol_wvgcal`, [19](#)
- `dWdLs`
 - `sol_wvgcal`, [20](#)
- `dWdLs2`
 - `sol_wvgcal`, [20](#)
- `dWdLs3`
 - `sol_wvgcal`, [21](#)
- `dWdLs3_calWeight`
 - `sol_wvgcal`, [22](#)
- `dWdLs4`
 - `sol_wvgcal`, [22](#)
- `dWdLs4_weights`
 - `sol_wvgcal`, [23](#)
- `edit_coordinates`
 - `off_ephem`, [6](#)
- `edit_coordinates2`
 - `off_ephem`, [6](#)
- `extMatInv`
 - `sol_wvgcal`, [23](#)
- `fluxCalQualityMetrics`
 - `SolSG.calibrators`, [42](#)
- `G_off_cal`
 - `sol_wvgcal`, [23](#)
- `generateSolCalScript`
 - `SolSG.stuffForSolarDataReduction`, [46](#)
- `getMeteo`
 - `sol_wvgcal`, [24](#)
- `getMeteoForScan`
 - `sol_wvgcal`, [24](#)
- `getPhase`
 - `sol_wvgcal`, [25](#)
- `getPhaseMS`
 - `sol_wvgcal`, [25](#)
- `getPWVforAnt`
 - `sol_wvgcal`, [25](#)
- `isMosaic`
 - `SolSG.scienceTarget`, [45](#)
- `modified_line`
 - `off_ephem`, [7](#)
- `obsDate`
 - `SolSG.observation`, [44](#)
- `off_ephem`, [5](#)
 - `edit_coordinates`, [6](#)
 - `edit_coordinates2`, [6](#)
 - `modified_line`, [7](#)
- `paregression`
 - `sol_wvgcal`, [26](#)
- `phiFixCompare`
 - `sol_wvgcal`, [26](#)

phiFixCompare3
 sol_wvgcal, 27
 plotPhases
 sol_wvgcal, 28
 printInfo
 SolSG.calibrators, 42
 SolSG.observation, 43
 SolSG.scienceTarget, 45
 pwv4scans
 sol_wvgcal, 28
 pwv4ScansCompare
 sol_wvgcal, 28
 pwv4scansRestore
 sol_wvgcal, 28
 pwvCalc
 sol_wvgcal, 29
 pwvCalcVerify
 sol_wvgcal, 29
 pwvCompare
 sol_wvgcal, 29

 rebin
 sol_wvgcal, 30
 rebin2
 sol_wvgcal, 30
 restoreSol_dWdB
 sol_wvgcal, 30
 rfStr
 SolSG.observation, 43

 saveSol_dWdB
 sol_wvgcal, 31
 saveSol_dWdB_Tbs
 sol_wvgcal, 31
 setAtmosSciSPWs
 sol_wvgcal, 31
 setAtmTool
 sol_wvgcal, 32
 setAtmTool4SPWs
 sol_wvgcal, 32
 smooth
 sol_wvgcal, 33
 sol_offPos
 sol_wvgcal, 33
 sol_PhaseFix
 sol_wvgcal, 33
 sol_pwv2selfCal
 sol_wvgcal, 34
 sol_pwvGcal
 sol_wvgcal, 34
 sol_wvgcal, 7
 acd_caltab_plots, 9
 acd_calTabs, 10
 acd_calTabs2, 10
 blindPhiFix, 11
 dB_sum, 11
 dPWV, 11
 dPWV_spwPart, 12
 dpwvXphi_corrs, 12
 dpwvXphi_plots, 13
 dpwvXsolWVR_raw, 13
 dpwvXwvr_corrs, 13
 dWdB, 14
 dWdB_Tbs, 15
 dWdB_Tbs_NL, 16
 dWdBdG, 16
 dWdBdG2, 17
 dWdBdG2_weights, 18
 dWdBdG_calWeights, 19
 dWdLs, 20
 dWdLs2, 20
 dWdLs3, 21
 dWdLs3_calWeight, 22
 dWdLs4, 22
 dWdLs4_weights, 23
 extMatInv, 23
 G_off_cal, 23
 getMeteo, 24
 getMeteoForScan, 24
 getPhase, 25
 getPhaseMS, 25
 getPWVforAnt, 25
 paregression, 26
 phiFixCompare, 26
 phiFixCompare3, 27
 plotPhases, 28
 pwv4scans, 28
 pwv4ScansCompare, 28
 pwv4scansRestore, 28
 pwvCalc, 29
 pwvCalcVerify, 29
 pwvCompare, 29
 rebin, 30
 rebin2, 30
 restoreSol_dWdB, 30
 saveSol_dWdB, 31
 saveSol_dWdB_Tbs, 31
 setAtmosSciSPWs, 31
 setAtmTool, 32
 setAtmTool4SPWs, 32
 smooth, 33
 sol_offPos, 33
 sol_PhaseFix, 33
 sol_pwv2selfCal, 34
 sol_pwvGcal, 34
 solTgtPos, 34
 sqld_P0, 35
 TbStats, 35
 tgt_pow, 35
 tgt_Tb, 36
 tgtPow_crossPlots, 36
 unwrap2, 37
 wopacity, 37
 wrapBack, 38
 wv2path, 38
 wvr_dataRaw, 38
 wvrPhase, 39

- SolSG.calibrators, 41
 - __init__, 42
 - allCalsLst, 42
 - allCalsStr, 42
 - fluxCalQualityMetrics, 42
 - printInfo, 42
- SolSG.observation, 43
 - obsDate, 44
 - printInfo, 43
 - rfStr, 43
- SolSG.scienceTarget, 44
 - __init__, 45
 - isMosaic, 45
 - printInfo, 45
 - targetFields, 46
- SolSG.stuffForSolarDataReduction, 46
 - generateSolCalScript, 46
- solTgtPos
 - sol_wvgcal, 34
- sqld_P0
 - sol_wvgcal, 35
- targetFields
 - SolSG.scienceTarget, 46
- TbStats
 - sol_wvgcal, 35
- tgt_pow
 - sol_wvgcal, 35
- tgt_Tb
 - sol_wvgcal, 36
- tgtPow_crossPlots
 - sol_wvgcal, 36
- unwrap2
 - sol_wvgcal, 37
- wopacity
 - sol_wvgcal, 37
- wrapBack
 - sol_wvgcal, 38
- wv2path
 - sol_wvgcal, 38
- wvr_dataRaw
 - sol_wvgcal, 38
- wvrPhase
 - sol_wvgcal, 39