

```
#!/usr/bin/env python
"""
```

```
@file sol_wvgcal.py
```

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

```
@author barta@asu.cas.cz
```

```
@date 2026-09-28 (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 $T_{\text{sky}}+T_{\text{src}}$ (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.

"""

```
import numpy as np
import math
import os
import sys
import matplotlib.pyplot as plt
from copy import deepcopy
```

```
# Where (in which Tsys SPW) we can find ATM/ACD data for our science SPW?
from casarecipes.almahelpers import tsysspwmap
```

```
#-----
#      1. 'Universal' helpers - general-purpose auxilliary functions
#-----
```

```
def 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).

"""

```
origDim=array1D.size
```

```
# Check, whether we can do the down-sampling. Return original array if not
# and print warning.
```

```
if (origDim % downSampleFactor):
    print("rebin(): Array dimension is not divisible by the re-bin factor!")
    return array1D
```

```

# Down-sampled size of the array.
newDim=origDim// downSampleFactor

# Make an artificial 2nd axis inside the new broader bin.
array2D=array1D.reshape((newDim,downSampleFactor))

# Now 'contract' this 2nd axis by averaging operation.
array1DResampled=array2D.mean(axis=1, dtype=np.float64)

return array1DResampled

#-----

def 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.
    """

    #-- Find the (equidistant) gridding.

    # Original-array grid.
    oSize=origX.size
    oDx=(origX[oSize-1]-origX[0])/oSize

    # Resampled-array grid.
    rSize=downSampledX.size
    rDx=(downSampledX[rSize-1]-downSampledX[0])/rSize

    #-- Create a new grid, evenly-divisible to the down-sampled mesh.

    # Down-sampling factor, as much close to the original cadence-ratio.
    dsFactor=round(rDx/oDx)

    # A new cell-size.
    nDx=rDx/dsFactor

    # A new grid/x-axis.
    nxGrid=nDx*np.arange(dsFactor*rSize)+downSampledX[0]

```

```

#-- Interpolate the data to the new grid.
nArray1D=np.interp(nxGrid,origX,array1D)

#-- Now we can call rebin().
return rebin(nArray1D,dsFactor)

#-----

def 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.
    """

    #-- Prepare the kernel and array for a convolution.

    # Normalize the smoothing kernel to 1.0
    kNorm=1.0/ kernel.sum()
    kernel*=kNorm

    # Find the kernel 'half-width'.
    khw=kernel.shape[0]// 2

    # Array last index.
    aLast=array1D.shape[0]- 1

    # Form the mirrored 'ghost' edges.
    edgeLeft=np.flip(array1D[1:1+khw])
    edgeRight=np.flip(array1D[aLast-khw:aLast])

    # Prepend/append them to the array1D.
    array1D_ext=np.append(edgeLeft,array1D)
    array1D_ext=np.append(array1D_ext,edgeRight)

    #-- Convolve, preserving the original array shape.

    aSmooth=np.convolve(array1D_ext,kernel,'valid')

    return aSmooth

#-----

def 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.

"""

Size of the time series

`N=arrX.size`

Check that the series are compatible (of the same size).

`if (arrY.size != N):`

`print("Sizes of input arrays must be equal!")`

`return -1`

Create the 'time' array - a vector of indices starting from 1.

`iVec=np.arange(N)+1`

#-- Evaluate quantities that repeat in further calculations.

`xMean=arrX.mean()`

`yMean=arrY.mean()`

`ix=iVec*arrX`

`ixSum=ix.sum()`

`iy=iVec*arrY`

`iySum=iy.sum()`

`xx=arrX*arrX`

`xxSum=xx.sum()`

`xy=arrX*arrY`

`xySum=xy.sum()`

#-- Evaluate the 2x2 linear system matrix for ' t ' and ' K '

```

A11=xxSum- N* xMean* xMean
A12=ixSum- 0.5*N* (N+1.0)* xMean

A21=A12
A22= (N- 1.0)* N* (N+1.0)/ 12.0

# Right-hand side of that lin. system.
C1=xySum- N* xMean* yMean
C2=iySum- 0.5*N* (N+1.0)* yMean

#-- Solve the system by Cramer's rule.

iD=1.0/ (A11*A22- A12*A21) # Inverse determinant of the system.

K=iD* (C1*A22- C2*A12)
tTrend=iD* (C2*A11- C1*A21)

# Find the 3rd parameter, expresed analytically in PR12.
Q=yMean- K* xMean- tTrend* 0.5* (N+1.0)

#-- How good is the fit? Calculate Xi-square.

errs=arrY- tTrend* iVec- K* arrX- Q
Xi2= (errs*errs) .sum()

#-- Form the result as a dictionary/map and return it.
rCoeffs={'tTrend':tTrend, 'K':K, 'Q':Q , 'Xi2':Xi2}

return rCoeffs

#-----

def 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.
    """

    # The difference array, with 0.0. prepended, it contains phase[0] in its
    # 1st element. Usefull when reverting back using np.cumsum()
    phiDiff=np.diff(np.unwrap(phase),prepend=0.0)

    # Identify and fix the positive jumps. Threshold 2.5 is approx. 80% of PI.
    posJumps=np.where(phiDiff > 2.5)
    phiDiff[posJumps]-=math.pi

```

```

# Fix the negative jumps the same way.
negJumps=np.where(phiDiff < -2.5)
phiDiff[negJumps]+=math.pi

# Return the array re-integrated back.
return np.cumsum(phiDiff)

#-----

def wrapBack(phases):
    '''
    An 'inverse' function to np.unwrap()
    '''
    wPhases = np.arctan2(np.sin(phases), np.cos(phases))

    return wPhases

#-----

def extMatInv(Ainv,Lb,Rb,L Rc):
    """
    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 = [[AA,Rb],[Lb,LRc]]

    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 = [[AA,Rb],[Lb,LRc]].
    """

    # Pre-calculate frequently repeating terms Ainv x Rb and Lb x Ainv
    AinvRb=np.matmul(Ainv,Rb)
    LbAinv=np.matmul(Lb,Ainv)

    # Calculate the Schur complement and its inversion.
    S=LRc- np.matmul(Lb,AinvRb)
    Sinv=np.linalg.inv(S)

    #- Calculate the blocks of the inverted matrix.

    # Start with the lower-left block as it repeats in upper-left block, too.
    Minv21=-np.matmul(Sinv,LbAinv)

    # Upper-left block.
    Minv11=Ainv- np.matmul(AinvRb,Minv21)

    # Upper-right block.
    Minv12=-np.matmul(AinvRb,Sinv)

    # Lower-right block.
    Minv22=Sinv

    #- Compose the Minv matrix from its blocks

```

```

# Compose the rows.
Minv1=np.hstack([Minv11,Minv12])
Minv2=np.hstack([Minv21,Minv22])

# Now the rows together
Minv=np.vstack([Minv1,Minv2])

return Minv

#-----
#                               2. ACD/dual-load (DL) calibration routines
#-----

def 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().
    """

    # P zero scan. Has to be looked in listobs.txt
    PoScan=1

    # ms and msmd tools flags.
    ownMS=False
    ownMsmd=False

    # Open own tools if 'global' not provided
    if (not myMS):
        # Open the ms tool
        myMS=mstool()
        myMS.open(vis)
        ownMS=True

    if (not myMsmd):
        # Open the ms tool.
        myMsmd=msmdtool()
        myMsmd.open(vis)
        ownMsmd=True

    # Find the data_desc_id for a given SPW.
    # Perform the [0,1,2,3] -> [13,14,15,16] mapping.
    dataDescID=int(myMsmd.datadescids(spw=spw+13)[0])

    # Get sub-scan info for the zero-scan (scan no.1).
    sub_info=aU.computeDurationOfScan(PoScan,mymsmd=myMsmd,
                                      returnSubscanTimes=True)

    # The sub_info indicates 4 subscans. However, the only non-zero selection
    # is for the sub-scan no. 2 - the others say

```

```

# RuntimeError: MSSelectionNullSelection : The selected table has zero rows.
# Hence, select just the sub-scan no. 2 by its time-span.
timeRange=sub_info[2][2]

# Init pre-selection by the SPW id.
myMS.selectinit(datadescid=dataDescID)

# Select 'autocorrelations' for a given antenna, the zero-scan number,
# and the sub-scan defined by its time-range.
myMS.select({'antenna1':ant, 'antenna2':ant, 'scan_number':PoScan,
            'time':timeRange})

# Extract the data from the reduced/filtered MS.
acdPow=myMS.getdata("data")["data"]

# The above array has dimensions 2 x nChans x nIntegs, where nIntegs is
# a number of integrations in the subscan (20 for sci spws, 240 for SQLDs).
# The 1st dim represents polarization, the 2nd is the freq.-channel number.
# The array is complex but its imaginary parts are zeroes (experimentally
# verified).
# We now take just the real part and do averaging over integrations.
acdPowAvg=acdPow.real.mean(axis=2, dtype=np.float64)

#- Clean-up the tools.

# Reset always - it resets the selection criteria.
myMS.reset()

# Close only if we use 'local' tools.
if (ownMsmid): myMsmid.done()
if (ownMS): myMS.close()

# Return the time-averaged P_zero for XX and YY as the 2 x 1 array.
return acdPowAvg

#-----

def 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 $nPol \times nChans$, i.e., 2×128 , if we use the science SPWs (5,7,9,11) and 2×1 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 $2 \times nPol \times nChans$, 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 $2 \times 2 \times 1$. 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 $2 \times 2 \times 128$. 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.*

"""

#- Global tools & data

ms and msmd tools flags, checked at closing

`ownMS=False`

`ownMsmd=False`

Open own tools if 'global' not provided

if (not myMS):

Open the ms tool.

`myMS=mstool()`

`myMS.open(vis)`

`ownMS=True`

if (not myMsmd):

Open the ms tool.

`myMsmd=msmdtool()`

`myMsmd.open(vis)`

`ownMsmd=True`

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

if (not acdLoads):

`acdLoads=aU.getLoadTemperatures(vis,doplot = False,`
`warnIfNoLoadTemperatures = True,`
`mymsmd=myMsmd)`

#-- Find the data_desc_ids for the given sciSPW and its Tsys: Data and Tsys

may have different SPWs.

```

# Description ID for the science data SPW.
sciDataDescID=int(myMsmc.datadescids(spw=spw)[0])

# The CALIBRATE_ATMOSPHERE scan (our *atmScan*) used for the ACD has data
# present only in some SPWs. Check, whether ours is among them.

acdSpws=myMsmc.spwsforscan(atmScan)

if (spw in acdSpws):
    # OK, we have atm/tsys data in the sciSPW
    tsysSPW=spw

    print('ACD calibration: Using spw ',spw, 'for ACD (Tsys) data.')

else:
    # We do not have atm measurement, must use a mapped SPW
    tsysmap = tsysspwmap(vis, tsystable = vis+'.tsys', tsysChanTol = 1)

    # Find correct SPW for the atmScan corresponding to *spw*.
    # Note: tsysspwmap() above maps the channel-integrated spws to their
    # resolved counterparts although we have Tsys/atm for the integrated spws,
    # too. Fix this by adding 1 to integrated spws: Chan-integrated SPWs will
    # be mapped to chan-integrated Tsys SPWs
    tsysSPW=tsysmap[spw]-spw%2+1

    print('ACD calibration: Mapping spws for obtaining the ACD (Tsys) data: ',
          spw, ' --> ',tsysSPW)

# Description ID for the corresponding Tsys/ATM SPW.
tsysDescID=int(myMsmc.datadescids(spw=tsysSPW)[0])

#--- Extract the ACD powers for sky, ambient and hot loads for Tsys spws

# Get sub-scan info for our (atm) scan.
sub_info=aU.computeDurationOfScan(atmScan,mymsmd=myMsmc,
                                  returnSubscanTimes=True)

# An empty dictionary for calibration powers.
PowCals={}
# Map the sub-scan numbers to names: The keys to our dictionary.
# Note: P_src need not to be present. It has been introduced at ~2023 in order
# to cope with the 'renormalization issue' (emission seen in autocorrs,
# typically in strong lines; the Sun has strong autocorr signal elsewhere,
# that is why we use T_ant calculations - the P_src sub-scan is irrelevant
# for the solar observations).
ssNames={1:'P_sky',2:'P_amb',3:'P_hot',4:'P_src'}

#- Iterate the atm scan sub-scans.
for key,value in sub_info[2].items():

    # Initialize the MS selector by the Tsys SPW.
    myMS.selectinit(datadescid=tsysDescID)

    # The sub-scan is defined by its time range. Here as [MJDs_start, MJDs_stop]
    timeRange=value

    # Select autocorrelations for a given antenna, the scan number,

```

```

# and the sub-scan defined by its time-range.
myMS.select({'antenna1':ant, 'antenna2':ant, 'scan_number':atmScan,
            'time':timeRange})

# Extract the data from the reduced/filtered MS.
acdPow=myMS.getdata("data")["data"]

# The above array has dimensions nPols x nChans x nIntegs, where nIntegs is
# a number of integrations in the subscan.
# Do time average over integrations.
acdPowAvg=acdPow.real.mean(axis=2,dtype=np.float64)

# Append the result to a dictionary.
PowCals[ssNames[key]]=acdPowAvg

# Reset all so-far applied selectors/filters before the next round.
myMS.reset()

#--> endfor(subScans) in ATM scan.

#-- Get the Tsys SPW definition (frequency channels in GHz). This defines
# also the channels, for which we have the gain and Trx.
freqs=myMsm.d.chanfreqs(tsysSPW)*1.0e-9 # in GHz

#--- Check our SPW type. For SQLDs we need also the power offset Po.

# Get the power offset Po. It is non-zero just for the SQLD spws.
# Set the isSQLD flag as well, it will be used later.
if spw in [0,1,2,3,13,14,15,16,25,26,27,28]:
    isSQLD=True
    Pzero=sqld_P0(vis,ant,spw,myMS,myMsm.d)
else:
    Pzero=np.array([[0.0],[0.0]])
    isSQLD=False

# Add the value to our collectuon of calibration powers.
PowCals['P_zero']=Pzero

#--- If we solve gains for the solar scan, extract also P_off necessary
# for the G* calculation.

# If target scan is not specified, use the one following after ATM scan
if (not tgtScan): tgtScan=atmScan+1

# All Sun scans.
solScans=myMsm.d.scansforintent('OBSERVE_TARGET#ON_SOURCE')

# Do the solar off-scan and G* calculation only if the Sun scan follows.
doSolarGain=(tgtScan in solScans)

if (doSolarGain):

    print("Solving for solar gain G* ...")

#-- Extract the P_off values. Do it even if we are in the AC SPW and these
# values are suspicious - in this case we fix it later by SQLD transfer.

```

```

# Extract sub-scan information for the solar scan.
sunSub_info=aU.computeDurationOfScan(tgtScan, mymsmd=myMsmmd,
                                     returnSubscanTimes=True)

# The off-source (i.e., hi-attenuated sky/atm) sub-scans are the first
# and the last. So, we need to find the number of sub-scans. It is the 2nd
# item of the info - see computeDurationOfScan() in sun_reduction_util.py.
nSubScans=sunSub_info[1]

# The off-source scans. Note: Keys in subInfo start with 1.
offSubScan1=1
offSubScan2=nSubScans

# Extract their time-ranges - we select the data later by them.
tRange1=sunSub_info[2][offSubScan1] # Time-range for the leading subscan.
tRange2=sunSub_info[2][offSubScan2] # Time-range for the trailing subscan.

#- Process the leading off-source sub-scan

# Initialize the MS selector by the science SPW.
myMS.selectinit(datadescid=sciDataDescID)

# Select autocorrelations for a given antenna, the scan number,
# and the sub-scan defined by its time-range.
myMS.select({'antenna1':ant, 'antenna2':ant, 'scan_number':tgtScan,
            'time':tRange1})

# Extract the data from the reduced/filtered MS.
sol_off_pow=myMS.getdata("data")["data"]

# The above array has dimensions nPols x nChans x nIntegs, where nIntegs is
# a number of integrations in the subscan.
# Do time average over integrations of the subscan.
P_off_lead=sol_off_pow.real.mean(axis=2, dtype=np.float64)

# Reset the selection.
myMS.reset()

#- Process the trailing off-source sub-scan

# Initialize the MS selector by the science SPW.
myMS.selectinit(datadescid=sciDataDescID)

# Select autocorrelations for a given antenna, the scan number,
# and the sub-scan defined by its time-range.
myMS.select({'antenna1':ant, 'antenna2':ant, 'scan_number':tgtScan,
            'time':tRange2})

# Extract the data from the reduced/filtered MS.
sol_off_pow=myMS.getdata("data")["data"]

P_off_trail=sol_off_pow.real.mean(axis=2, dtype=np.float64)

# Reset the selection.
myMS.reset()

```

```

# Combine the leading and trailing value into a single numpy array.
# The 1st dimension is now time: 0=leading, 1=trailing.
P_off=np.array([P_off_lead, P_off_trail])

# Add the value to our collection of calibration powers for completion.
PowCals['P_off']=P_off

# P_off has been calculated from solar (target-scan) data, these may have
# different frequency resolution than the Tsys used for basic ACD calibs:
# Get the solar SPW definition (frequency channels in GHz).
sunFreqs=myMsmc.chanfreqs(spw)*1.0e-9 # in GHz

#--> endif(doSolarGain)

#--- Extract the corresponding load temperatures.

# We select just the values for our ATM scan and antenna.
# For the Sun we scale to kiloKelvins!!!
T_amb = 0.001*acdLoads[ant][atmScan]['amb']
T_hot = 0.001*acdLoads[ant][atmScan]['hot']

#--- Calculate gains G & G*, and T_rx

# Calculate the gain (Eq. 7 in White et al. 2017).
# In fact, we wish to have 1/G, because we shall later calculate T_sky+src
# from the measured power P_sky+src.

# Inverse power-difference.
iPowDiff=1.0/ (PowCals['P_hot']-PowCals['P_amb'])

# The (inverse) gain: 1/G
iGain=(T_hot-T_amb)*iPowDiff
# Calculate T_rx (Eq. 9 in White et al. 2017).
T_rx=(T_hot* (PowCals['P_amb']-Pzero)-T_amb* (PowCals['P_hot']-Pzero))*iPowDiff
# Calculate effective instrumental T_inst combining T_rx and G-scaled Pzero.
T_inst=T_rx+Pzero*iGain

#-- The (inverse) solar gain: 1/G* (Eq. 8 in White et al. 2017)
if (doSolarGain):

    # Actual calculation, the result is of the shape
    # 2(lead/trail) x 2(XX/YY) x nChans(=1 for SQLDs)

    # Numerators of the fraction in Eq. 8
    iGStarXX=iGain[0]* (PowCals['P_sky'][0]-Pzero[0])
    iGStarYY=iGain[1]* (PowCals['P_sky'][1]-Pzero[1])

    # We shall divide this by numerator (P_off-Pzero), which has channel
    # structure as the solar-target scan SPW. This may differ from
    # the Tsys spw (check tsysmap). In order to solve this mixed-channel issue
    # (gain and Trx at Tsys spw, P_off at solar spw), we interpolate everything
    # to the solar spw. Must be done per polarization, as np.interp() works for
    # 1-d arrays only.
    if (tsysSPW!=spw):

        print('--- SPW mapping: Interpolating everything to solar SPW channels!')
```

```

# ----- !!! Attention !!! -----
#
# @todo This needs to be reconsidered because we have troubles when
# interpolating the LSB spws (5 and 7), which have decreasing order
# of frequencies. The np.interp() requires the *freqs* array to be
# sorted in ascending order. Suggested solution: Flip the arrays,
# make interpolation, and flip the arrays back in order to have them
# sorted in 'natural' order as they come from correlator. The 2023 solar
# data are not affected, they use no spw mapping and need no interpolation
# but for general handling this should be sorted out.

# Numerators of the fraction in Eq. 8 to be interpolated
iGStarXX=np.interp(sunFreqs,freqs,iGStarXX)
iGStarYY=np.interp(sunFreqs,freqs,iGStarYY)

# Let us interpolate the iG and T_rx, too.
iGain[0]=np.interp(sunFreqs,freqs,iGain[0])
iGain[1]=np.interp(sunFreqs,freqs,iGain[1])

T_rx[0]=np.interp(sunFreqs,freqs,T_rx[0])
T_rx[1]=np.interp(sunFreqs,freqs,T_rx[1])

# Replace the spw channels definition for entire caltab
freqs=sunFreqs

#-->endif(tsys spw mapping)

# Denominator (P_off-Pzero) is already at sun-spw freq. resolution.
# This is our G* as inferred from P_off subscans.
iGainStar=np.array([[iGStarXX,iGStarYY],[iGStarXX,iGStarYY]])/(P_off-Pzero)

# Fix effective instrumental T_inst: Combine T_rx and G_star-scaled Pzero.
'''
# Use time-average for the G-star.
T_inst[0]=T_rx[0]+0.5*Pzero[0]*(iGainStar[0,0]+iGainStar[1,0])
T_inst[1]=T_rx[1]+0.5*Pzero[1]*(iGainStar[0,1]+iGainStar[1,1])
'''

# Use the leading P_off because it is closer to P_sky - see the comment
# inside tgt_Tb(), after 'if(doSolarGain)'.
T_inst[0]=T_rx[0]+Pzero[0]*(iGainStar[0,0])
T_inst[1]=T_rx[1]+Pzero[1]*(iGainStar[0,1])

else:
    # Return a neutral factor of 1.0, just organised in the expected shape.
    iGainStar=np.array([[[1.0],[1.0]],[[1.0],[1.0]])

# Data are extracted, we can close the tools (if local).
if (ownMsm): myMsm.done()
if (ownMS): myMS.close()

# Return inverse gains iG/iG*, T_rx, and T_inst for each freqs channel in
# a dictionary. T_inst differs from T_rx only for SQLD SPWs, which have
# Pzero offset in power.
return {'freqs':freqs,'iG':iGain,'iG*':iGainStar,'T_rx':T_rx,
        'T_inst':T_inst,'calPows':PowCals}

```

```
#-----

def 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 ( $P_{sqld} = K \cdot P_{ac} + Q$ ) for the same frequency domain,
    where the parameter  $K$  depends on the ratio  $G^*_{sqld}/G^*_{ac}$ , 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.
    """

    # Mapping between SPWs of the same frequency domain: AC -> SQLD
    spwMap={5:0, 6:0,
            7:1, 8:1,
            9:2, 10:2,
            11:3, 12:3}

    sqldSpw=spwMap[acSpw]

    # For non-resolved SPWs we have a single channel, for resolved integrate
    # over a range with the edges trimmed.
    if (acSpw in [6, 8, 10, 12]):
        fChans=0
        resolved=False
    else:
        fChans=range(8, 120)
        resolved=True

    # Powers for the on-source subscans: AC and SQLD
    acData=tgt_pow(vis, tgtScan, ant, acSpw, myMS=myMS, myMsmd=myMsmd, subScans='on')
    sqldData=tgt_pow(vis, tgtScan, ant, sqldSpw, myMS=myMS, myMsmd=myMsmd,
                    subScans='on')

    # SQLD powers rebined to te same cadence as ACs.
    sqPowXX=rebin(sqldData[1][0, 0], 12)
```

```

sqPowYY=rebin(sqldData[1][1,0],12)

# ACD calibration table: AC
acCtb=acd_calTabs(vis, atmScan, ant, acSpw, myMS=myMS, myMsm= myMsm)
sqldCtb=acd_calTabs(vis, atmScan, ant, sqldSpw, myMS=myMS, myMsm= myMsm)

# G* as per P_off in AC spw (in question))
acGxx=0.5/ acCtb['iG*'][0,0,fChans]+0.5/ acCtb['iG*'][1,0,fChans]
acGyy=0.5/ acCtb['iG*'][0,1,fChans]+0.5/ acCtb['iG*'][1,1,fChans]

# Trx
acTRxx=acCtb['T_rx'][0,fChans]
acTRYy=acCtb['T_rx'][1,fChans]

# ACD calibration table: SQLD

# G* in SQLD spw (trusted)
sqGxx=0.5/ sqldCtb['iG*'][0,0,0]+0.5/ sqldCtb['iG*'][1,0,0]
sqGyy=0.5/ sqldCtb['iG*'][0,1,0]+0.5/ sqldCtb['iG*'][1,1,0]

sqTRxx=sqldCtb['T_rx'][0,0]
sqTRYy=sqldCtb['T_rx'][1,0]

sqPoXX=sqldCtb['calPows']['P_zero'][0,0]
sqPoYY=sqldCtb['calPows']['P_zero'][1,0]

if (not resolved):
    #-- Integrated AC SPWs are easy

    # AC powers.
    acPowXX=acData[1][0,fChans]
    acPowYY=acData[1][1,fChans]

    # Spectral correction factors. Trivial here, we have no spectral dependency
    # within the integrated AC SPW.
    SpCxx=1.0
    SpCyy=1.0

    # Offset Q: Assumed SQLD power at the zero AC power in the correlation
    # P_sqld=K*P_ac+Q. Essentially Po, but may be modified if Trx differs
    # between the AC and corresponding SQLD spw.
    P00xx=sqGxx*(sqTRxx- acTRxx)+sqPoXX
    P00yy=sqGyy*(sqTRYy- acTRYy)+sqPoYY

else:
    #-- Resolved AC SPWs need much more care - we take into account spectral
    # weighting by normalized spectral gain, atmospheric transmissivity
    # (+ atm. radiation), and solar (QSR) spectrum.

    # AC powers averaged over channels.
    acPowXX=acData[1][0,fChans].mean(axis=0)
    acPowYY=acData[1][1,fChans].mean(axis=0)

    #- Spectral gain

```

```

# Low-attenuation ( $P_{\text{sky}}$ ) gains in the AC SPW (limited to fChans range).
laGxx=1.0/ acCtb['iG'][0,fChans]
laGyy=1.0/ acCtb['iG'][1,fChans]

# Spectral  $*P_{\text{sky}}$  gains normalised to 1.0
# We assume these remain untouched by the attenuation level switching,
# i.e., they are the same also for the  $*P_{\text{off}}$  sub-scans.
gXX=laGxx/ laGxx.mean()
gYY=laGyy/ laGyy.mean()

#- Atmospheric transmissivity, atm. radiation, and instrumental effects.

# Calculate the ATM model for our actual PWV as per SYSCAL table.
pwv=getPWVforAnt(vis,tgtScan,ant,myMsmd) ['pwvAvg']
atmParams=wopacity(vis,acSpw,tgtScan,userPWV=pwv)

# Estimated atmospheric (spectral) transmissivity for the acSpw.
# Limit the array to our fChans range.
tr=atmParams['tr'][fChans]

# Relative spectral transmissivity normalized to unity.
trn=tr/ tr.mean()

# Atmospheric (spectral) effective BB brightness temperature within fChans.
T_atm=atmParams['T_atm'][fChans]
TeBB=T_atm*(1.0-tr)

# Rear-spillover + ohmic-loss antenna efficiency.
eta_l=0.95
# Forward-spillover antenna efficiency.
eta_F=0.90

# Ambient-load temperature for rear-spillover estimation.
TLoad = aU.getLoadTemperatures(vis, doplot = False,
                                warnIfNoLoadTemperatures = True,
                                mymsmd=myMsmd)
Tamb=TLoad[ant][atmScan]['amb']

#- Solar normalised spectrum

# Reference  $T_b$  for solar disc center @198GHz (see solar SD cal script).
TsunCenter=6300.0 # K
# Solar radius
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.

```

```

r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta)
mu=np.sqrt(1.0-(x_h**2+y_h**2)/r_sol**2)

b_2=-1127.0 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature @f_LO (198GHz) corrected to limb brightening.
Tsun0=TsunCenter+0.5*b_2*np.log(mu)

# LO frequency - the reference frequency for TsunCenter
f_LO=1.98e11 # Hz

freqs=myMsmc.chanfreqs(acSpw)[fChans] # Hz

# Make SPW arrays in increasing order for the reverse-channel (LSB) SPWs.
# We flip arrays related to measurements in acSpw only, the models
# (like ATM) are already in the correct order.
if (freqs[0] > freqs[1]):

    print("Reversed SPW.")

    freqs=np.flip(freqs)

    gXX=np.flip(gXX)
    gYY=np.flip(gYY)

    acTRxx=np.flip(acTRxx)
    acTRYy=np.flip(acTRYy)

# Central frequency of the AC SPW
f0=freqs.mean()

# Mean solar T_B at the current SPW central frequency.
Tsun0=Tsun0+b_2*np.log(f0/f_LO)

print('Tsun=',Tsun0)

# Normalized (to unity) solar spectrum (linearized).
sf=1.0+b_2/Tsun0*(freqs/f0-1.0)

#- Spectral correction factors

SpCxx=1.0/np.mean(gXX*trn*sf)
SpCyy=1.0/np.mean(gYY*trn*sf)

plt.plot(gXX,'r-')
plt.plot(trn,'g-')
plt.plot(sf,'b-')

print('Spectral correction factors: ',SpCxx,SpCyy)

#- Offsets (P_sqld for zero P_ac)

# Separate the individual contributions in order to compare their weights.
# Pol XX
TrxDiff=sqTRxx-SpCxx*np.mean(gXX*acTRxx)

```

```

TskyDiff=(1.0-eta_F+eta_l)*(TeBB.mean()-SpCxx*np.mean(gXX*TeBB))
TspillDiff=(1.0-eta_l)*Tamb*(1.0-SpCxx)

P00xx=sqGxx*(TrxDiff+TskyDiff+TspillDiff)+sqPoXX

print('XX offset components: dTrx=',TrxDiff,'dTsky=',TskyDiff,
      'dTspill=',TspillDiff,'Po/G=',sqPoXX/sqGxx)

# Pol YY
TrxDiff=sqTRyy-SpCyy*np.mean(gYY*acTRyy)
TskyDiff=(1.0-eta_F+eta_l)*(TeBB.mean()-SpCyy*np.mean(gYY*TeBB))
TspillDiff=(1.0-eta_l)*Tamb*(1.0-SpCyy)

P00yy=sqGyy*(TrxDiff+TskyDiff+TspillDiff)+sqPoYY

print('YY offset components: dTrx=',TrxDiff,'dTsky=',TskyDiff,
      'dTspill=',TspillDiff,'Po/G=',sqPoYY/sqGyy)

#--> endif(resolved)

# Add the [0.0, P00] point to the arrays of measured powers in order
# to extent the proportionality line.
acPowXXext=np.append(acPowXX,0.0)
sqPowXXext=np.append(sqPowXX,P00xx)

acPowYYext=np.append(acPowYY,0.0)
sqPowYYext=np.append(sqPowYY,P00yy)

# Find the correlation slopes
Kxx,Qxx=np.polyfit(acPowXXext,sqPowXXext,1)
Kyy,Qyy=np.polyfit(acPowYYext,sqPowYYext,1)

# AC solar-attenuated gains as recalculated from their SQLD counterparts.
acGxxM=SpCxx*sqGxx/Kxx
acGyyM=SpCyy*sqGyy/Kyy

#-- The result: Gain correction factors
gainCorrectionXX=acGxx.mean()/acGxxM
gainCorrectionYY=acGyy.mean()/acGyyM

print('direct/derived G* XX=',gainCorrectionXX)
print('direct/derived G* YY=',gainCorrectionYY)

return (gainCorrectionXX,gainCorrectionYY)

#-----

def acd_calTabs2(vis,atmScan,ant,spw,myAtm=None,myMS=None,myMsmid=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 *atool* 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.*

"""

#- Global tools & data

ms and msmd tools flags, checked at closing

ownMS=False

ownMsmd=False

Open own tools if 'global' not provided

if (not myMS) :

Open the ms tool.

myMS=mstool()

myMS.open(vis)

ownMS=True

if (not myMsmd) :

Open the msmd tool.

myMsmd=msmdtool()

myMsmd.open(vis)

ownMsmd=True

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

if (not acdLoads) :

acdLoads=aU.getLoadTemperatures(vis,doplot = False,
warnIfNoLoadTemperatures = True,
mymsmd=myMsmd)

#- Get the standard ACD calibration tables.

calDict=acd_calTabs(vis,atmScan,ant,spw,myMS=myMS,myMsmd=myMsmd,
acdLoads=acdLoads)

#- Check, what else is necessary

sqldSPWs=[0,1,2,3,13,14,15,16,25,26,27,28]

solScans=myMsmd.scansforintent('OBSERVE_TARGET#ON_SOURCE')

tgtScan=atmScan+1

if ((spw in sqldSPWs) or (tgtScan not in solScans)):

It is easy, we may return the standard caltables 'as is'.

return calDict

#- We are in the AC SPW and doing G for the Sun. We need to transfer
the P_off/P_sky gain-drop from the SQLD spw.*

Check, whether we have ATM tool ready. Exit with an error if not.

if (not myAtm) :

```

    print("Error: myAtm ATM tool must be defined for solar G* calibration!")
    print("Exiting...")

    return False

#-- OK, continue with G* corrections.

# Mapping between SPWs of the same frequency domain: AC -> SQLD
spwMap={5:0,6:0,
        7:1,8:1,
        9:2,10:2,
        11:3,12:3}

sqldSpw=spwMap[spw]

print("Transferring gains from the SQLD spw",sqldSpw,...')

# For non-resolved SPWs we have a single channel, for resolved integrate
# the spectra over a range with the edges trimmed.
# Note: We trim the edges symmetrically, so flipping of reversed SPWs
# does not affect the result.
if (spw in [6,8,10,12]):
    fChans=[0]
    resolved=False
else:
    fChans=range(8,120) # Trimm edge channels symmetrically.
    resolved=True

#-- Powers for on-source subscans: AC and SQLD, we use their proportionality.

# Retrieve the AC-data time series.
(acTime,acData)=tgt_pow(vis,tgtScan,ant,spw,myMS=myMS,myMsm= myMsm,
                        subScans='on')

# Extract AC powers per polarization, averaged over channels.
acPowXX=acData[0,fChans].mean(axis=0)
acPowYY=acData[1,fChans].mean(axis=0)

# Retrieve the SQLD-data time series.
(sqTime,sqldData)=tgt_pow(vis,tgtScan,ant,sqldSpw,myMS=myMS,myMsm= myMsm,
                          subScans='on')

# Extract SQLD powers per polarization, rebin to the same timebase as the ACs.
sqPowXX=rebin2(sqldData[0,0],sqTime,acTime)
sqPowYY=rebin2(sqldData[1,0],sqTime,acTime)

#-- Standard ACD caltables for our AC and adjunged SQLD spws.

#- 1. Standard ACD caltable for our AC SPW - we have it already.

acCtb=calDict

# Low-attenuation (P_sky-based) gains G. The P_off-based G* is untrusted.
# Limit to fChans range for calculations.
laGxx=1.0/ acCtb['iG'][0,fChans]
laGyy=1.0/ acCtb['iG'][1,fChans]

# Spectral P_sky-based gains normalised to 1.0 with fChans range.

```

```

# We assume these remain untouched by the attenuation level switching,
# i.e., they are the same also for the P_off sub-scans.
gXXnorm=laGxx.mean()
gYYnorm=laGyy.mean()

gXX=laGxx/ gXXnorm
gYY=laGyy/ gYYnorm

# Trx in AC. Limited to fChans range.
acTRxx=acCtb['T_rx'][0,fChans]
acTRYy=acCtb['T_rx'][1,fChans]

#- 2. ACD caltable for the adjunged SQLD spw.

sqldCtb=acd_calTabs(vis, atmScan, ant, sqldSpw, myMS=myMS, myMsmid=myMsmid,
                    acdLoads=acdLoads)

# G* in SQLD spw (trusted).
# We have leading and trailing values from the two (lead/trail) P_off subscans
sqGxxLead=1.0/ sqldCtb['iG*'][0,0,0]
sqGxxTrail=1.0/ sqldCtb['iG*'][1,0,0]

sqGyyLead=1.0/ sqldCtb['iG*'][0,1,0]
sqGyyTrail=1.0/ sqldCtb['iG*'][1,1,0]

'''
# For an offset calculation (see below) we need also averages over a scan.
sqGxx=0.5*(sqGxxLead+sqGxxTrail)
sqGyy=0.5*(sqGyyLead+sqGyyTrail)
'''

# As it has been found, the high-attenuation gain calculated from P_off is
# affected by changing WVR during solar scan - see comment in acd_calTabs().
# It is therefore better to take the gain calculated from P_off measurement
# immediately following the low-attenuation P_sky ATM scan, i.e., from the
# leading P_off sub-scan of solar scan.
sqGxx=sqGxxLead
sqGyy=sqGyyLead

# Trx in SQLD
sqTRxx=sqldCtb['T_rx'][0,0]
sqTRYy=sqldCtb['T_rx'][1,0]

# Pzero in SQLD
sqPoXX=sqldCtb['calPows']['P_zero'][0,0]
sqPoYY=sqldCtb['calPows']['P_zero'][1,0]

if (not resolved):
    #-- Integrated AC SPWs are easy

    # Spectral correction factors. Trivial here, we have no spectral dependency
    # within the integrated AC SPW.
    SpCxx=1.0
    SpCyy=1.0

```

```
# Offset Q: Assumed SQLD power at the zero AC power in the correlation
#  $P_{sqld} = K \cdot P_{ac} + Q$ . Essentially  $P_o$ , but may be modified if Trx differs
# between the AC and corresponding SQLD spw.
```

```
P00xx=sqGxx* (sqTRxx- acTRxx[0])+sqPoXX
P00yy=sqGyy* (sqTRYy- acTRYy[0])+sqPoYY
```

```
else :
```

```
-- Resolved AC SPWs need much more care - we take into account spectral
# weighting by normalized spectral gain, atmospheric transmissivity
# (+ atm. radiation). Solar spectrum slope is negligible, we ignore it.
```

```
#- Atmospheric transmissivity, atm. radiation, and instrumental effects.
```

```
# Map our science SPW to the 0 - 3 range for spws in *myAtm* attool.
```

```
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}
atmSPW=sciAtmMap[spw]
```

```
# Setup the ATM model for our actual PWV as per SYSCAL table.
```

```
pwv=getPWVforAnt(vis,tgtScan,ant,myMsmd=myMsmd) ['pwvAvg']
myAtm.setUserWH2O(qa.quantity(pwv, 'mm'))
```

```
# Extract the empty-sky (atm+CMB) brightness temperature (spectrum) in K.
# Limit its domain to our fChans range.
```

```
TeBB=myAtm.getTebbSkySpec(atmSPW,qa.quantity(pwv, 'mm')) [1] ['value'] [fChans]
```

```
# Calculate transmissivity.
```

```
# Dry zenith opacity
```

```
tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW) [1])
```

```
# Water-vapour related 'wet' opacity
```

```
tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW) [1] ['value'])
```

```
# Airmass for the current scan.
```

```
airMass=myAtm.getAirMass()
```

```
# Total opacity (spectral). Limit it to fChans range.
```

```
opacity=(airMass*(tau_wet+tau_dry)) [fChans]
```

```
# Transmissivity (spectral).
```

```
tr=np.exp(- opacity)
```

```
# Relative spectral transmissivity normalized to unity.
```

```
trn=tr/ tr.mean()
```

```
#- Antenna efficiencies
```

```
antNames=myMsmd.antennanames()
```

```
myAntName=antNames[ant]
```

```
antType=myAntName[:2]
```

```
# Map the antenna efficiencies by antenna type. As per investigation from
# merged T_sun temperatures from 2023 and 2026 data (project_22_25_Tb.ods).
```

```
eta_l_map={'CM':0.941, 'DA':0.947, 'DV':0.956}
```

```
eta_F_map={'CM':0.891, 'DA':0.897, 'DV':0.906}
```

```
# Rear-spillover + ohmic antenna efficiency - see project_22_25_Tb.ods
```

```
eta_l=eta_l_map[antType]
```

```
# Forward-spillover antenna efficiency - see project_22_25_Tb.ods
```

```
eta_F=eta_F_map[antType]
```

```

# We have two exception that seem to have systematically lower
# antenna efficiencies: DA64 and DV08 - see project_22_25_Tb.ods
if myAntName=='DA64':
    eta_l=0.929
    eta_F=0.880
if myAntName=='DV08':
    eta_l=0.943
    eta_F=0.894

# Rear-spillover temperature, in Kelvins.
Tamb=acdLoads[ant][atmScan]['amb']

#- Channels in ascending order.

freqs=myMsmc.chanfreqs(spw)[fChans] # Hz

# Make SPW arrays in increasing order for the reverse-channel (LSB) SPWs.
# We flip arrays related to measurements in the acSpw only, the models
# (like ATM) are already in the correct order.
if (freqs[0] > freqs[1]):

    freqs=np.flip(freqs)

    gXX=np.flip(gXX)
    gYY=np.flip(gYY)

    acTRxx=np.flip(acTRxx)
    acTRYy=np.flip(acTRYy)

#- Spectral correction factors

SpCxx=1.0/ np.mean(gXX*trn)
SpCyy=1.0/ np.mean(gYY*trn)

#- Offsets (P_sqld for zero P_ac)

# Separate the individual contributions in order to compare their weights.
# Pol XX
TrxDiff=sqTRxx- SpCxx* np.mean(gXX* acTRxx)
# T_rx is in kiloKelvins(!), TeBB and Tamb not: Scale all to kK.
TskyDiff=0.001* (1.0- eta_F+eta_F*eta_l) * (TeBB.mean() - SpCxx* np.mean(gXX* TeBB))
TspillDiff=0.001* eta_F* (1.0- eta_l) * Tamb* (1.0- SpCxx)

P00xx=sqGxx* (TrxDiff+TskyDiff+TspillDiff)+sqPoXX

# Pol YY
TrxDiff=sqTRYy- SpCyy* np.mean(gYY* acTRYy)
TskyDiff=0.001* (1.0- eta_F+eta_F*eta_l) * (TeBB.mean() - SpCyy* np.mean(gYY* TeBB))
TspillDiff=0.001* (1.0- eta_l) * Tamb* (1.0- SpCyy)

P00yy=sqGyy* (TrxDiff+TskyDiff+TspillDiff)+sqPoYY

#--> endif(resolved)

```

```

# Add the [0.0, P00] point to the arrays of measured powers in order
# to extent the proportionality line.

acPowXXext=np.append(acPowXX,0.0)
sqPowXXext=np.append(sqPowXX,P00xx)

acPowYYext=np.append(acPowYY,0.0)
sqPowYYext=np.append(sqPowYY,P00yy)

# Find the correlation slopes
Kxx,Qxx=np.polyfit(acPowXXext,sqPowXXext,1)
Kyy,Qyy=np.polyfit(acPowYYext,sqPowYYext,1)

# AC solar-attenuated gains as recalculated from their SQLD counterparts.
# These represent the spw-averaged values, i.e. the scales.
acGxxLead=SpCxx*sqGxxLead/ Kxx
acGxxTrail=SpCxx*sqGxxTrail/ Kxx

acGyyLead=SpCyy*sqGyyLead/ Kyy
acGyyTrail=SpCyy*sqGyyTrail/ Kyy

# Now use them to scale the spectral normalized gains.
# We wish now to have the gains G* over full SPW, including the trimmed edge
# channels. So, produce first the gXX/gYY versions with the edge channels
# included. Moreover, we have natural channel order - no flipping in LSB spws.
gXXext=(1.0/ acCtb['iG'][0])/ gXXnorm
gYYext=(1.0/ acCtb['iG'][1])/ gYYnorm

# Now apply the scaling.

gStarXXLead=acGxxLead*gXXext
gStarXXTrail=acGxxTrail*gXXext

gStarYYLead=acGyyLead*gYYext
gStarYYTrail=acGyyTrail*gYYext

# Replace original iG* in the standard-ACD calDict with the new values.
calDict['iG*'][0,0]=1.0/ gStarXXLead
calDict['iG*'][1,0]=1.0/ gStarXXTrail

calDict['iG*'][0,1]=1.0/ gStarYYLead
calDict['iG*'][1,1]=1.0/ gStarYYTrail

# Return the modified calDict.
return calDict

#-----

def 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.
    """

    #-- This MS-specific defs.

```

```

# Extract info about antennas and SPWs.
myMsm = msmdtool()
myMsm.open(vis)

# Find the antenna list (the IDs and corresponding names).
antIDs = myMsm.antennasids()
antNames = myMsm.antennanames()

# All Sun scans.
solScans = myMsm.scansforintent('OBSERVE_TARGET#ON_SOURCE')

myMsm.done()

# Output PNG directory.
pngDir = os.getcwd + '/acdCals_plots/'

## Loop over all atm scans

for aScan in atmScans:

    # The next-target scan: Solar or BP-cal?
    tgtScan = atmScan + 1

    # Do the solar off-scan and G* calculation only if the Sun scan follows.
    doSolarGain = (tgtScan in solScans)

    # Set the output PNG file path - the source description (Sun/BP).
    fPath = ""

    if doSolarGain:
        fPath = pngDir + "Sun_"
    else:
        fPath = pngDir + "BP_"

    ## Loop over all antennas

    for myant in antIDs:

        # Report progress.
        print('For scan ', aScan, ' calculating antenna ',
              int(myant), '<-->', antNames[myant])

        ## Init plot for antenna.

        plt.rcParams["figure.figsize"] = (20, 80)
        # Make XX and YY plots as subplots (horizontally)
        ff, sps = plt.subplots(4, 2)

        # Antenna name as the entire-plot title.
        plt.suptitle(antNames[myant])

        # X-axis label just for the bottom sub-plot.
        sps[3, 0].set_xlabel('frequency [GHz]')
        sps[3, 1].set_xlabel('frequency [GHz]')

        ## Loop over SPWs. Access them by index in order to map them to subplots

```

```

for spwIndex in range(len(sciSPWs)):

    spw=sciSPWs[spwIndex]

    calFactors=acd_calTabs(vis,aScan,int(myant),spw)

    # Extract the frequency SOW definition.
    freqs=calFactors['freqs']
    nChans=len(freqs)

    # Extract the gain G/G* and the additive constant T_rx
    Trx_XX=calFactors['T_rx'][0]
    Trx_YY=calFactors['T_rx'][1]

    if doSolarGain:
        # We are doing solar scan, show G* as the 'main' gain.
        # We have two values for the beginning and the end of the solar scan,
        # show their average.

        iG_XX=0.5*(calFactors['iG*'][0][0]+calFactors['iG*'][1][0])
        iG_YY=0.5*(calFactors['iG*'][0][1]+calFactors['iG*'][1][1])

        # Over-plot the MD gain before being scaled by the P_off factor.
        # Prepare it trimmed by edge channels on-the-fly - see below.
        iG0_XX=calFactors['iG'][0][7:nChans-8]
        iG0_YY=calFactors['iG'][1][7:nChans-8]

    else:
        # We are doing BP scan, show G as the gain.
        iG_XX=calFactors['iG'][0]
        iG_YY=calFactors['iG'][1]

    # Trim the edge channels
    spwFreqs=freqs[spwIndex][7:nChans-8]

    Trx_XX=Trx_XX[7:nChans-8]
    Trx_YY=Trx_YY[7:nChans-8]

    iG_XX=iG_XX[7:nChans-8]
    iG_YY=iG_YY[7:nChans-8]

    # Y-axis labels for each sub-plot
    sps[spwIndex,0].set_ylabel('T_rx [K]')
    sps[spwIndex,1].set_ylabel('Inverse gain')

    # Plot T_rx in the left column, XX as red, YY as blue.
    sps[spwIndex,0].plot(spwFreqs,Trx_XX,'-r')
    sps[spwIndex,0].plot(spwFreqs,Trx_YY,'-b')

    # Plot the (inverse) gain in the right column, XX as red, YY as blue.
    sps[spwIndex,1].plot(spwFreqs,iG_XX,'-r')

```

```

sps[spwIndex,1].plot(spwFreqs,iG_YY,'-b')

# If we are in solar scan, overplot the non-scaled gain as dashed.
if (aScan==8):
    sps[spwIndex,1].plot(spwFreqs,iG0_XX,'--r')
    sps[spwIndex,1].plot(spwFreqs,iG0_YY,'--b')

# Save to a file and close.
plt.savefig(fPath+antNames[myant]+'.png')
plt.show()
plt.close()

```

return

```

#-----
#           3. Extract raw data from the MS & Gain calTables
#-----

```

```

def 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 nIntgs.
The 2nd component: Array of powers in the form 2 x nChans x nIntgs.
I.e., 2 x 1 x nIntgs for the SQLD and chan-integrated SPWs, and
2 x 128 x nIntgs for channel-resolved SPWs. The first dim (2) means polarization (XX, YY).*

```

    """

```

```

# ms and msmd tools flags.

```

```

ownMS=False
ownMsmd=False

```

```

# Open own tools if 'global' not provided

```

```

if (not myMS):

```

```

# Open the ms tool.
myMS=mstool()
myMS.open(vis)
ownMS=True

if (not myMsmd):
    # Open the ms tool.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    ownMsmd=True

#-- Selection by the sub-scans. This we will need if the source is the Sun:
#   The very firsts and very last sub-scans in the solar scan are for
#   taking the P_off. This is like P_sky in dual-load but with high
#   attenuation level, suitable for the solar scans.

# Find the data_desc_id for a given science SPW.
sciDataDescID=int(myMsmd.datadescids(spw=spw)[0])

# Initialize the MS selector by SPW.
myMS.selectinit(datadescid=sciDataDescID)

# Select ACs/SQLDs for a given antenna and the scan number.
myMS.select({'antenna1':ant, 'antenna2':ant, 'scan_number':tgtScan})

#-- Are we (tgtScan) the solar scan? If so, refine the selection.

# All Sun scans.
solScans=myMsmd.scansforintent('OBSERVE_TARGET#ON_SOURCE')

if (tgtScan in solScans):
    #- In case of the solar scan we have more strict selection by subscans.

    # Get sub-scan info for our tgtScan.
    sub_info=aU.computeDurationOfScan(tgtScan, mymsmd=myMsmd,
                                      returnSubscanTimes=True)

    # How many sub-scans do we have?
    nSubScans=sub_info[1]

    # The on-source subscans are just these in the middle (1st/last are off).
    # Note: This is a key to the map/dictionary of subscans and it starts at 1.
    offSubScans=[1, nSubScans]
    onSubScans=range(2, nSubScans)

    # Make and fill a list of time-ranges (as MJDs) for the on-source sub-scans.
    # We use the string representation of date-time - that is why sub_info[3] is
    # extracted here. This is suitable for ms::msselect() used subsequently -
    # see the comments below.
    tRanges=[]

    if (subScans=='on'):
        for oss in onSubScans:

```

```

        tRanges.append(sub_info[3][oss])
    elif (subScans=='off'):
        for offs in offSubScans:
            tRanges.append(sub_info[3][offs])

    # Make the string-list to be a single comma-separated string.
    trString=', '.join(tRanges)

    # On demand, refine selection to on-/off- subscans defined by time-range.
    # Note: We use ms::msselect() (and not ms::select()) here as it allows
    # for non-contiguous union of time intervals for the sub-scans
    if (trString): myMS.msselect({'time':trString})

    #-- Filtration done, now pick-up and process the data.

    # Extract the (complex) data and their 'coordinates' from the filtered MS.
    tgtData=myMS.getdata(["data","axis_info"],ifraxis=True)

    # Get the data (complex amplitudes)
    tgt_cAmp=tgtData["data"]

    # Get their corresponding times as MJD.
    tgt_mjd=tgtData["axis_info"]["time_axis"]["MJDseconds"]

    # The data are complex by nature, but in fact having just the real part.
    tgtPow=tgt_cAmp.real

    # The 'ifraxis=True' in the ms.getdata() above causes expansion of the antenna
    # axis. In our case (auto-corrs or SQLDs) it is degenerated to dimension=1.
    # Let us remove it (its is the 3rd axis of the array).
    tgt_pow=np.squeeze(tgtPow,axis=2)

    #print('Tgt pow: ',tgt_pow)

    # Reset the selection.
    myMS.reset()

    # Data are extracted, we can close the tools (if local).
    if (ownMsmd): myMsmd.done()
    if (ownMS): myMS.close()

    # Return the time axis as MJD.
    # Return the power as a numpy array of nPols x nChans x nIntegs dims.
    return (tgt_mjd,tgt_pow)

#-----

def 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.*

"""

Extract the data

time1,data1=tgt_pow(vis,tgtScan,ant,spw1)

time2,data2=tgt_pow(vis,tgtScan,ant,spw2)

We use channel-integrated windows - get rid of the fake frequency dimension.

d1=np.squeeze(data1,axis=1)

d2=np.squeeze(data2,axis=1)

Separate by polarizations, keeping just the varying parts.

d1MeanXX=d1[0].mean()

d1VarsXX=d1[0]-d1MeanXX

d1MeanYY=d1[1].mean()

d1VarsYY=d1[1]-d1MeanYY

d2MeanXX=d2[0].mean()

d2VarsXX=d2[0]-d2MeanXX

d2MeanYY=d2[1].mean()

d2VarsYY=d2[1]-d2MeanYY

plt.plot(d1VarsXX,d2VarsXX,'r-')

plt.plot(d1VarsYY,d2VarsYY,'b.')

return

#-----

def 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*.*

"""

Create the table tool and use it to open the gain caltable.

cTab=tbtool()

cTab.open(gTable)

Construct a query string: Selection by antenna and spw.

selection="ANTENNA1==" + str(antID) + " && SPECTRAL_WINDOW_ID==" + str(spw)

Extract the sub-table that correspond to the selection criteria.

```

# We need just the time and gain columns.
ctSel=cTab.query(selection,sortlist="TIME",columns="TIME,CPARAM")

# Extract the column data
times=ctSel.getcol("TIME")
gains=ctSel.getcol("CPARAM")

print("--- Sizes: ", len(times), gains.shape)

# Data are extracted, we may close the tab tools (incl. the sub-table ctSel).
ctSel.close()
cTab.done()

# The gains array has a shape (nPolS x 1 x nTimeSamples) for standard
# G-tables and only (1 x nTimeSamples) for the WVR/PWV caltables.
# Check the actual shape.
nPolS=gains.shape[0]

if (nPolS==2):
    # Extract the polarizations separately and contract the fake second axis.
    # As a result, we get just 1D array with sizeequal to nTimeSamples for
    # each polarization.
    gXX=gains[0][0]
    gYY=gains[1][0]

    # We are interested just in the phases.
    phaseXX=np.angle(gXX)
    phaseYY=np.angle(gYY)

else:
    # Just Stokes-I
    gII=gains[0][0]

    # In order to have uniform results, duplicate the phases for XX/YY pols.
    phaseXX=np.angle(gII)
    phaseYY=phaseXX

# Use np.unwrap in order to get rid of the phase jumps at +/- PI.
upXX=np.unwrap(phaseXX)
upYY=np.unwrap(phaseYY)

# Form the returned result as a dictionary.
pDict={'mjd':times,'phiXX':upXX,'phiYY':upYY}

return pDict

#-----

def 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.

"""

#-- Hard-wired settings for the solar test MS.

Reference antenna for the solar MS.

refAnt=37

Solar observe-target scan.

tgtScan=9

myMsm=msmdtool()

myMsm.open(vis)

#- 1. Identify the SPW by the data_desc_id

myDataDescID=int(myMsm.datadescids(spw=spw)[0])

#print('Selected data desc ID: ',myDataDescID)

#- 2. Identify the off-source subscans by their time range.

Get sub-scan info for our tgtScan.

sub_info=aU.computeDurationOfScan(tgtScan,myms=myMsm,
returnSubscanTimes=True)

How many sub-scans do we have?

nSubScans=sub_info[1]

The on-source subscans are those middle, except the first and last one.

Note: This is a key to the map/dictionary of subscans and it starts at 1.

onSubScans=range(2,nSubScans)

Make and fill a list of time-ranges (as MJDs) for the on-source sub-scans.

We use the string representation of date-time - that is why sub_info[3] is

extracted here. This is suitable for ms::msselect() used subsequently -

see the comments below.

tRanges=[]

for onss **in** onSubScans:

tRanges.append(sub_info[3][onss])

Make the string-list to be a single comma-separated string.

trString=', '.join(tRanges)

We are done with the time-range extraction.

myMsm.done()

#-- Use the above identifiers/selectors for filtering the data from MS.

myMs=msstool()

myMs.open(vis)

Initialize the MS selector by SPW

myMs.selectinit(datadescid=myDataDescID)

Matrix of cross-correlations/visibilities is just the upper-triangular:

The 'antenna2' column starts with antenna that have by 1 bigger ID than

antenna1 (checked in browsetable()). So, we have to select carefully.

if (refAnt > antID):

```

    ant1=antID
    ant2=refAnt
else:
    ant1=refAnt
    ant2=antID

# Select data for a given antenna and the scan number.
myMs.select({'antenna1':ant1, 'antenna2':ant2, 'scan_number':tgtScan})

# Refine selection by on-source subscans only.
# Note: We use ms::msselect() (and not ms::select()) here as it allows
# for non-contiguous union of time intervals for the sub-scans
myMs.msselect({'time':trString})

#--- Filtration done, now pick-up and process the data.

# Extract the (complex) data and their 'coordinates' from the filtered MS.
allData=myMs.getdata(["data", "model_data", "axis_info"], ifraxis=True)

# Clean-up the tool MS tool.
myMs.reset()
myMs.close()

#- Separate the complex-structured data to components
# Remove the degenerated baseline axis of dim=1.

# Get the raw-data visibilities stored as complex amplitudes.
rawVis=np.squeeze(allData["data"], axis=2)
# Get the model visibilities stored as complex amplitudes.
modVis=np.squeeze(allData["model_data"], axis=2)
# Get their corresponding times as MJD secs.
time_mjds=allData["axis_info"]["time_axis"]["MJDseconds"]

#- Data processing: Extract the phases only.

# Complex amplitudes for MODEL - trim the more extended edge channels because
# the MFS in tclean() does not work with them either.

mcAmpXX=modVis[0, 20:107, :]
mcAmpYY=modVis[1, 20:107, :]

# Complex amplitudes for DATA - keep the same shape as for model.

dcAmpXX=rawVis[0, 20:107, :]
dcAmpYY=rawVis[1, 20:107, :]

# Process the phases. Correct wraps along the f-axis for each time instant.

# Get the size of t-axis, it is the same for all arrays.
tSize=dcAmpXX.shape[1]

# Arrays for time-evolution of the mean phase delays and their STDdevs.
phiXX=np.empty(tSize)
phiYY=np.empty(tSize)

```

```
stdPhiXX=np.empty(tSize)
stdPhiYY=np.empty(tSize)
```

```
# Loop over time (integrations).
```

```
for tt in range(tSize):
```

```
    # DATA phases as functions of channel, unwrapped.
```

```
    dPhiXX=unwrap2(np.angle(dcAmpXX[:,tt]))
```

```
    dPhiYY=unwrap2(np.angle(dcAmpYY[:,tt]))
```

```
    # MODEL phases as functions of channel, unwrapped.
```

```
    mPhiXX=unwrap2(np.angle(mcAmpXX[:,tt]))
```

```
    mPhiYY=unwrap2(np.angle(mcAmpYY[:,tt]))
```

```
    # Make a difference DATA - MODEL
```

```
    dmPhiXX=dPhiXX-mPhiXX
```

```
    dmPhiYY=dPhiYY-mPhiYY
```

```
    # Find the mean (over f-channels) and RMS.
```

```
    phiXX[tt]=np.mean(dmPhiXX)
```

```
    phiYY[tt]=np.mean(dmPhiYY)
```

```
    # Find the stddev (over f-channels) and RMS.
```

```
    stdPhiXX[tt]=np.std(dmPhiXX)
```

```
    stdPhiYY[tt]=np.std(dmPhiYY)
```

```
# Unwrap the phase jumps also in the time dimension.
```

```
pXX=np.unwrap(phiXX)
```

```
pYY=np.unwrap(phiYY)
```

```
return (time_mjds,pXX,pYY,stdPhiXX,stdPhiYY)
```

```
#-----
```

```
def 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*.
    """
```

```
#-- This MS-specific defs.
```

```
# Extract info about antennas and SPWs.
```

```
myMsmD=msmdtool()
```

```
myMsmD.open(vis)
```

```
# Find the antenna list (the IDs and corresponding names).
```

```
antIDs=myMsmD.antennasids()
```

```
antNames=myMsmD.antennanames()
```

```
myMsmD.done()
```

```
# Gain table may reside in another directory (e.g., calibrated,self_cal,...)
```

```
# Do not confuse the output dir, use just its base-name for id of outputs.
```

```

gTableName=os.path.basename(gTable)

# Output PNG directory. One per gain table.
pngDir='./phase_plots/'+gTableName+'/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

#-- Loop over all antennas

for myant in antIDs:

    # Report progress.
    print('Plotting antenna ',int(myant),'<-->',antNames[myant])

    # Get phases for a given time as the time->phase map (a dictionary).
    tpDict=getPhase(gTable,int(myant),spw)

    time_axis=tpDict['mjd']
    phx=tpDict['phiXX']
    phy=tpDict['phiYY']

    #- Make a plot with an overplot

    # Plot phases in red: XX solid line, YY dashed.
    plt.plot(time_axis,phx,'-r')
    plt.plot(time_axis,phy,'--r')

    # Add the antenna name as a plot title.
    plt.title(antNames[myant])

    # Save to a file and close.
    plt.savefig(pngDir+'phi_'+antNames[myant]+'.png')
    plt.show()
    plt.close()

```

```

return

```

```

#-----
#                2.+3. ACD-calibrated data: From powers to temperatures
#-----

```

```

def tgt_Tb(vis,atmScan,tgtScan,antID,spwID,myMS=None,myMsmid=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.

```

    """

```

```

#-- MS and msmd tool: Do we have global or local instances?
casalog.post("tgt_Tb() ")

# MS and msmd tools flags (local/global)
ownMS=False
ownMsmd=False

# Open own tools if 'global' not provided
if (not myMS):
    # Open the ms tool.
    myMS=msstool()
    myMS.open(vis)
    ownMS=True

if (not myMsmd):
    # Open the ms tool.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    ownMsmd=True

if (not acdLoads):
    acdLoads=aU.getLoadTemperatures(vis,doplot = False,
                                    warnIfNoLoadTemperatures = True,
                                    mymsmd=myMsmd)

#-- Read ACD calibration tables and the raw powers.

# ACD calibration 'tables': The (inverse) gain(s) and T_rx/T_inst.

gTrx=acd_calTabs2(vis,atmScan,antID,spwID,myAtm=myAtm,myMS=myMS,
                  myMsmd=myMsmd,acdLoads=acdLoads)

# Alternatively, use the less accurate acd_calTabs().
# gTrx=acd_calTabs(vis,atmScan,antID,spwID,myMS=myMS,myMsmd=myMsmd,
#                  acdLoads=acdLoads)

# Raw total-power data for the target, preceded by their times as MJD
mjd_tgt,Ptgt=tgt_pow(vis,tgtScan,antID,spwID,myMS,myMsmd)

# Power has dims 2 x nChans x nIntgs, where 2 = num of pols.
# Keep just the time (nIntgs) and possibly freq. (nChans) dimensions.
sXX=Ptgt[0]
sYY=Ptgt[1]

# Analyse MS: Resolve between solar and BP tgtScan, find native sci SPW
# resolution for the target.

solScans=myMsmd.scansforintent('OBSERVE_TARGET#ON_SOURCE')
tgtFreqs=myMsmd.chanfreqs(spwID)*1.0e-9

# We can close the tools, if local.
if (ownMsmd): myMsmd.done()
if (ownMS): myMS.close()

```

```

#-- We return time-dependent (and for some SPWs also frequency-dependent)
# quantities. In order to perform operations with multi-dimensional
# arrays correctly, we have to use tensor multiplications (np.tensordot()).
# To that end, the time-dimension needs to be incorporated as a vector.

# Num. of integrations in the target-power data.
nIntegs=Ptgt.shape[2]

# Time-dimension stepping vector [ 0.0, 1.0, 2.0, ... , float(nIntegs-1)]
# (works like findgen() in IDL).
tdVec=np.arange(nIntegs)
# Time-dimension unit vector [1.0, 1.0, 1.0, ... , 1.0]
# (works like std::vector<float>(size_t s, float, val=1.0) in C++).
tdUnitVec=np.ones(nIntegs)

#-- Calculate the time-domain gains.

#Are we (the tgtScan) a solar scan?
doSolarGain=(tgtScan in solScans)

if (doSolarGain):
    # Use G* if we (the tgtScan) are a solar scan.

    #--- Attention !!!
    #
    # Think about linear interpolation in time (expressed as integration index)
    # as we have iG* defined for the beginning and the end of the solar scan.
    # (No frequency-domain interpolation is required, for the solar gains we
    # have done it in acd_calTab() already.)
    #
    # On the other hand !!! Change of the P_off between leading and trailing
    # sub-scan *need not* to be solely because of the G* gain changes but also
    # because T_sky has changed between (i) ATM scan preceding the target scan,
    # (ii) leading P_off sub-scan of tgt scan, and (iii) trailing P_off subscan.
    # Derivation of Eq. (8) in White et al. (2017) assumes that T_sky does
    # not change, but as we are in Band 5 (high opacity) and usually with
    # medium to high PWV, change in the PWV may significantly change value
    # of T_sky. In principle, we should be able to infer T_sky(t) from the
    # raw WVR data as per dWdLs(). T_sky(t) can be then used to fixing the
    # P_off subscans w.r.t P_sky of preceding atm scan - this is todo item.
    # Until it is implemented, the safest way is to use just G* derived from
    # P_off of the *leading* sub-scan (that is why iGend below takes index '0').
    # The leading sub-scan follows immediately after atm scan and is observed
    # at the same sky position, so the T_sky change between those two should be
    # (hopefully) small.
    iGbeg=gTrx['iG*'][0]
    iGend=gTrx['iG*'][0]

    # To be uncommented after acd_Caltabs() takes into account T_sky time
    # variation inferred from the raw WVR data.
    # iGend=gTrx['iG*'][1]

    # In order to prevent mess with multi-dimensional arrays, we split
    # processing by polarization. As a result, we get just time- and
    # channel-dependent quantities (of the shape nChans x nIntegs)

    iGbegXX=iGbeg[0]

```

```

iGbegYY=iGbeg[1]

iGendXX=iGend[0]
iGendYY=iGend[1]

# The gain step (from integration-to-integration). 1-D arrays of nChans dim.
gStepXX=(iGendXX-iGbegXX)/(nIntegs-1.0)
gStepYY=(iGendYY-iGbegYY)/(nIntegs-1.0)

# Linearly interpolated (in time) i-gains for XX and YY pols.
iGainXX=np.tensordot(iGbegXX,tdUnitVec,0)+np.tensordot(gStepXX,tdVec,0)
iGainYY=np.tensordot(iGbegYY,tdUnitVec,0)+np.tensordot(gStepYY,tdVec,0)

# Extract T_inst - i.e. Trx corrected by Po/G for SQLD spws than Trx itself.
# T_inst are constant in time, will be just extracted. The time-dimension
# will be added below, during calibration.
Tixx=gTrx['T_inst'][0]
Tiyy=gTrx['T_inst'][1]

#--> endif(Sun)

else:
    # Non-solar target. Use iG as a (inverse) gain, no time-interpolation.
    # If the gain/T_rx lives at different frequency channels (see acd_calTab()),
    # do interpolation in the frequency domain. This should never happen in
    # the solar observation, as it has both target powers (sci SPWs) and the
    # Tsys SPW (for ACD cal) in the same shape (TDM = 128 channels).

    # Frequency domain for the gain G and T_rx
    gainFreqs=gTrx['freqs']

    # Just a very basic check - if it does not work, check yet f[0] and df.
    if (len(tgtFreqs) != len(gainFreqs)):

        # Gain and target spw definitions differ: Do interpolate.
        iGxx=np.interp(tgtFreqs,gainFreqs,gTrx['iG'][0])
        iGyy=np.interp(tgtFreqs,gainFreqs,gTrx['iG'][1])

        Tixx=np.interp(tgtFreqs,gainFreqs,gTrx['T_inst'][0])
        Tiyy=np.interp(tgtFreqs,gainFreqs,gTrx['T_inst'][1])

    else:
        # The same channels for gain and tgt data, just extract to a new array.

        iGxx=gTrx['iG'][0]
        iGyy=gTrx['iG'][1]

        Tixx=gTrx['T_inst'][0]
        Tiyy=gTrx['T_inst'][1]

    # Replicate in time-domain: Constant (in time) i-gains for XX and YY pols.
    iGainXX=np.tensordot(iGxx,tdUnitVec,0)
    iGainYY=np.tensordot(iGyy,tdUnitVec,0)

    #-->endelse(non-solar tgt)

```

```

#-- Now apply the calibration separately for XX and YY pols.
# Add the time-dimension also to the Trx values.
Txx=sXX*iGainXX-np.tensordot(Tixx,tdUnitVec,0)
Tyy=sYY*iGainYY-np.tensordot(Tiyy,tdUnitVec,0)

# Return the brightness temperature for XX and YY, preceded by the time-axis
# in MJD, as a tuple.
return (mjd_tgt,Txx,Tyy)

#-----
#                               4. Interface to ATM model
#-----

def setAtmTool(myAtm,vis,spw,scan,
               myMsmd=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 effectivelly 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)
    eventhough 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.
    """

    #-- Get info about the SPW definition, weather, and target elevation.
    # We need msmd tool attached to our MS for that.

    # Local msmd-tool flag.
    ownMsmd=False

```

```

if (not myMsmd):
    # Open the ms tool.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    ownMsmd=True

#-- Weather & airmass (1/sin(elevation)) - per-scan specific.
# We have to start with that and set the SPW later, reverted order
# does not work!

# We use here the standard aU.getWeather() just for extracting target
# elevation in order to calculate airmass. For the meteo characteristics
# we use our (more accurate) approach in getMeteoForScan().

# getWeaheer() returns a tuple, the weather dictionary is its 1st component.
wDict = (aU.getWeather(vis, scan=scan, mymsmd=myMsmd))[0]
# getMeteoForScan() returns directly a map/dictionary.
wDict2=getMeteoForScan(vis, scan, myMsmd)

# Elevation/airmass from aU.getWeather()
elevation = wDict['elevation']
airMass = 1/ np.sin(np.radians(elevation))

# Meteo from getMeteoForScan()
humidity = wDict2['humidity']
temperature = wDict2['temperature']
pressure = wDict2['pressure']

#-- Telescope location & clima - ALMA-specific

# Default climatic model for ALMA( 'midLatitudeWinter' or 'tropical').
almaClima=aU.translateLatitudeClass('midLatitudeWinter')
# WV scale-height for ALMA [km].
h0=wavHeightScale
# Temperature drop per 1 km
tempGrad=tempDropPerKm

# Altitude of ALMA [m].
altitude = 5059.0

#-- Default ATM model params.

# Top altitude of the atmosphere [km].
maxAltitude=60.0
# Presurre step [mbar].
dP=5.0
# Pressure-step factor [dimension-less].
dPm=1.1

#- These parameters are requested as quantities in at.initAtmProfile()

tempQa=qa.quantity(temperature, "K")
altQa=qa.quantity(altitude, "m")
pressQa=qa.quantity(pressure, "mbar")
h0Qa=qa.quantity(h0, "km")

```

```

maxAltQa=qa.quantity(maxAltitude, "km")
dPQa=qa.quantity(dP, "mbar")
tempGradQa=qa.quantity(tempGrad, "K/km")

```

#-- Setup the atmospheric vertical profile.

```

atmModel= myAtm.initAtmProfile(humidity=humidity,
                                temperature=tempQa,
                                altitude=altQa,
                                pressure=pressQa,
                                h0=h0Qa,
                                dTem_dh=tempGradQa,
                                maxAltitude=maxAltQa,
                                dP=dPQa,
                                atmType=almaClima,
                                dPm=dPm)

```

#-- SPW definition.

Extract all SPW channels from metadata.

```

freqs = myMsm.d.chanfreqs(spw)*1.0e-9 # ndarray, in GHz

```

#- We have to make a specific settings for SPW 4 - the WVR data.

Why? It has 4 channels, but they have different bandwidths (see Figure in Appendix A6 (WVRs) of ALMA Technical Handbook.

Solution: We define extra ATM SPW for each of the 4 WVR channels.

The 'sub-channel' resolution in these ATM SPWs is arbitrary, but it helps us to properly integrate over the band-width of each WVR channel.

For the science SPWs we are using the natural channel resolution of the

channel-resolved SPWs, even for the integrated ACs and SQLDs, these are

always mapped to corresponding resolved SPW in the same frequency domain.

```

if (spw == 4):

```

#-- WVR SPW: 1 ATM SPW per 1 WVR channel, make 100 sub-channels in each.

```

    print("Specific setup for the WVR SPW.")

```

Extract the WVR channel widths.

```

    dfs = myMsm.d.chanwidths(4)*1.0e-9 # ndarray, in GHz

```

#- at.initSpectralWindow() requires some args as quantities.

```

    fCenterQa=qa.quantity(list(freqs), 'GHz')

```

```

    fWidthQa=qa.quantity(list(dfs), 'GHz')

```

Define 100 (arbitrary) virtual sub-channels per each WVR channel.

```

    dfQa=qa.quantity(list(0.01*dfs), 'GHz')

```

Setup our spectral windows: Frequency-domain for the attool.

```

    myAtm.initSpectralWindow(len(freqs), fCenterQa, fWidthQa, dfQa)

```

```

else:

```

#-- Sci SPW: 1 ATM SPW per 1 sci SPW, keep 128-channel resolution.

We are always mapped to channel-resolved SPWs (even for single-channel ACs and SQLDs). Calculate the derived SPW characteristics.

```

# Channel are evenly distributed for sci SPWs, so we can determine the
# channel width as a difference of any two central channel frequencies.
# If the spw has decreasing frequencies, we revert the order.
df=abs(freqs[1]-freqs[0])
# Ref. rrequency is in the center of the SPW.
refFreq = np.mean(freqs)

# This is the full SPW width. The freqs[] array stores *central* frequencies
# for each channel, so in order to reach the very SPW edges we have to
# extend by the channel half-width at both sides.
frequencyRange = [np.min(freqs)-0.5*df, np.max(freqs)+0.5*df]

#- at.initSpectralWindow() requires some args as quantities.

fCenterQa=qa.quantity(refFreq, 'GHz')
fWidthQa=qa.quantity(frequencyRange[1]-frequencyRange[0], 'GHz')
dfQa=qa.quantity(df, 'GHz')

# Setup our single spectral window: Frequency-domain for the attool.
myAtm.initSpectralWindow(1,fCenterQa,fWidthQa,dfQa)
#--> endelse(Sci SPW)

# Close the msmd tool if local.
if (ownMsmd): myMsmd.done()

#- Set the target (scan) specific airmass (cosec(elevation)).
myAtm.setAirMass(airMass)

"""
# Debug: Print SPW summary.

nChans=myAtm.getNumChan(0)
cWidth=myAtm.getChanSep(0)
for c in range(nChans):
    cFreq=myAtm.getChanFreq(c,0)
    print("Channel ", c, ": central frequency = ",cFreq['value'][0],
          cFreq['unit'],", channel width: ", cWidth['value'][0],cWidth['unit'])

"""

return

#-----

def setAtmTool4SPWs(myAtm,vis,scan,
                    myMsmd=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().
    """

    #- Get info about the SPW definition, weather, and target elevation.

```

```

# We need msmd tool attached to our MS for that.

# Local msmd-tool flag.
ownMsmd=False

if (not myMsmd):
    # Open the ms tool.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    ownMsmd=True

#-- Weather & airmass (1/sin(elevation)) - per-scan specific.
# We have to start with that and set the SPW later, reverted order
# does not work!

# We use here the standard aU.getWeather() just for extracting target
# elevation in order to calculate airmass. For the meteo characteristics
# we use our (more accurate) approach in getMeteoForScan().

# getWeaheer() returns a tuple, the weather dictionary is its 1st component.
wDict = (aU.getWeather(vis, scan=scan, mymsmd=myMsmd))[0]
# getMeteoForScan() returns directly a map/dictionary.
wDict2=getMeteoForScan(vis, scan, myMsmd)

# Elevation/airmass from aU.getWeather()
elevation = wDict['elevation']
airMass = 1/ np.sin(np.radians(elevation))

# Meteo from getMeteoForScan()
humidity = wDict2['humidity']
temperature = wDict2['temperature']
pressure = wDict2['pressure']

#-- Telescope location & clima - ALMA-specific

# Default climatic model for ALMA( 'midLatitudeWinter' or 'tropical').
almaClima=aU.translateLatitudeClass('midLatitudeWinter')
# WV scale-height for ALMA [km].
h0=wwHeightScale
# Temperature drop per 1 km
tempGrad=tempDropPerKm

# Altitude of ALMA [m].
altitude = 5059.0

#-- Default ATM model params.

# Top altitude of the atmosphere [km].
maxAltitude=60.0
# Presurre step [mbar].
dP=5.0
# Pressure-step factor [dimension-less].
dPm=1.1

#- These parameters are requested as quantities in at.initAtmProfile()

```

```

tempQa=qa.quantity(temperature, "K")
altQa=qa.quantity(altitude, "m")
pressQa=qa.quantity(pressure, "mbar")
h0Qa=qa.quantity(h0, "km")
maxAltQa=qa.quantity(maxAltitude, "km")
dPQa=qa.quantity(dP, "mbar")
tempGradQa=qa.quantity(tempGrad, "K/km")

```

#-- Setup the atmospheric vertical profile.

```

atmModel= myAtm.initAtmProfile(humidity=humidity,
                                temperature=tempQa,
                                altitude=altQa,
                                pressure=pressQa,
                                h0=h0Qa,
                                dTem_dh=tempGradQa,
                                maxAltitude=maxAltQa,
                                dP=dPQa,
                                atmType=almaClima,
                                dPm=dPm)

```

#-- Setup the 4 ATM SPWs corresponding to chan-resolved ACs.

Arrays for spw-defining parameters.

```

refFreq=np.zeros(4)
df=np.zeros(4)
bandWidth=np.zeros(4)

```

```
atmSpwIndex=0
```

```
for sciSPW in [5,7,9,11]:
```

Extract all SPW channels from metadata.

```
freqs=myMsmc.chanfreqs(sciSPW)*1.0e-9 # ndarray, in GHz
```

Channels are evenly distributed for science SPWs, so we can determine the

channel width as a difference of any two central-channel frequencies.

If the spw has decreasing frequencies, we revert the order.

```
df_cur=abs(freqs[1]-freqs[0])
```

```
df[atmSpwIndex]=df_cur
```

Ref. frequency is in the center of the SPW.

```
refFreq[atmSpwIndex] = np.mean(freqs)
```

*# This is the full SPW width. The freqs[] array stores *central* frequencies*

for each channel, so in order to reach the very SPW edges we have to

extend it by the channel half-width at both ends.

```
frequencyRange = [np.min(freqs)-0.5*df_cur, np.max(freqs)+0.5*df_cur]
```

```
bandWidth[atmSpwIndex]=frequencyRange[1]-frequencyRange[0]
```

```
atmSpwIndex+=1
```

#--> endfor (sci SPWs)

#- at.initSpectralWindow() requires some args as quantities. Convert

the np.array to python lists.

```

fCenterQa=qa.quantity(list(refFreq), 'GHz')
fWidthQa=qa.quantity(list(bandWidth), 'GHz')
dfQa=qa.quantity(list(df), 'GHz')

# Setup our spectral windows: Frequency-domain for the attool.
myAtm.initSpectralWindow(4, fCenterQa, fWidthQa, dfQa)

# Close the msmd tool if local.
if (ownMsmd): myMsmd.done()

#-- Set the target (scan) specific airmass (cosec(elevation)).
myAtm.setAirMass(airMass)

return

#-----

def 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_1: '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-trans)*Tatm expr.).
    """

    # This method is intended just for channel-resolved science SPWs.
    # Return immediately if we are trying to calculate it for other.

    if (spw not in [5,7,9,11]):
        print("wopacity() is intended just for resolved sci SPWs, which ", spw,
              " is not.")
        return

    #-- Open and set the atm tool.
    myAtm=attool()
    setAtmTool(myAtm, vis, spw, scan)

    #-- Calculate the opacities for PWV=1.0 (zenith value).

    pwv=1.0
    myAtm.setUserWH2O(qa.quantity(pwv, 'mm'))

    #- Extract the opacities from the ATM model.

    # Dry zenith opacity
    tau_d = np.array(myAtm.getDryOpacitySpec(0)[1])

```

```

# Water-vapour related 'wet' opacity for PWV=1.0
tau_1 = np.array(myAtm.getWetOpacitySpec(0)[1]['value'])

#--- We need also the T_atm - estimation of the effective gas temperature.
# This to be calculated for the actual PWV during the observation.
# PWV may be supplied by the user (e.g., per antenna). If not, we
# use median PWV over entire EB and array.

if (userPWV>0.0):
    pwv=userPWV
else:
    # Get the median PWV over entire array for entire EB duration.
    pwv=aU.getMedianPWV(vis)[0]

# Set the actual PWV as to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv, 'mm'))

#- Extract the opacities & friends from the ATM model.

# Dry zenith opacity
tau_dry = np.array(myAtm.getDryOpacitySpec(0)[1])
# Water-vapour related 'wet' opacity
tau_wet = np.array(myAtm.getWetOpacitySpec(0)[1]['value'])

# Brightness temperature (Effective Black-Body) of the atmosphere
TebbSky = myAtm.getTebbSkySpec(spwid=0)[1]['value']

# Extract the airmass for the current scan.
airMass=myAtm.getAirMass()

# Recalculate it to (effective) gas temperature via  $T_b = T_{atm} \cdot (1 - \text{trans})$ 
opacity = airMass * (tau_wet + tau_dry)
transmission = np.exp(-opacity)

# Shape: 1D array of size nChans-1. Anyway, it is almost constant, may be
# easily represented by a single value (e.g., the mean).
T_atm = TebbSky / (1 - transmission)

#--- Compose the return value as a dictionary. Clean-up the atm tool

atmProp={'airmass':airMass, 'tau_d':tau_d, 'tau_w':tau_wet, 'tau_1':tau_1,
        'tr':transmission, 'T_atm':T_atm}

myAtm.done()

return atmProp

#-----

def 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.

"""

Collect the sub-models for resolved spws as a map.

atmModelRes={}

We do it just for the resolved SPWs

for rspw **in** [5,7,9,11]:

atmModelRes[rspw]=wopacity(vis,rspw,tgtScan)

#-- Now we map any SPW to the res-spw atm model.

*# Use the *resolved* swps corresponding to our integrated ACs and SQLDs.*

spwRes={0:5,1:7,2:9,3:11,5:5,6:5,7:7,8:7,9:9,10:9,11:11,12:11}

Collect the sub-models for requested spws as a map.

atmModel={}

for spw **in** spwList:

atmModel[spw]=atmModelRes[spwRes[spw]]

return atmModel

#-----

def getPWVforAnt(vis,tgtScan,antID,myMsmd=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.

"""

#-- Explore the MS metadata

ownMsmd=False

if (**not** myMsmd):

myMsmd=msmdtool()

myMsmd.open(vis)

ownMsmd=True

```

# Scan timing
scanInfo=aU.computeDurationOfScan(tgtScan, mymsmd=myMsmd,
                                   returnSubscanTimes=True)

# Antenna ID to name conversion.
antNames=myMsmd.antennanames()
antName=antNames[antID]

if (ownMsmd): myMsmd.done()

#-- Query the WV data

# Open the MS/ASDM_CALWVR table
wvTab = tbtool()
wvTab.open(vis+"/ASDM_CALWVR")

# Select just the data for antID, the result is a sub-table.
wvAntTab=wvTab.query(query="antennaName==\'"+antName+\'"',
                     sortlist="startValidTime",
                     columns="startValidTime,water")

# Extract time and pwv for the antenna.
pwvTimes = wvAntTab.getcol('startValidTime')
pwv = wvAntTab.getcol('water')

# Close both the per-antenna subtable and the main WV table.
wvAntTab.close()
wvTab.done()

#-- Find the time-range for the tgtScan

# Num. of subscans in the tgtScan.
nSubScans=scanInfo[1]

# First and last subscan: The result is a map starting from 1, not an array.
ssTimesFirst=scanInfo[2][1]
ssTimesLast=scanInfo[2][nSubScans]

# The sub-scan times are 2-element arrays - subscans beginning and end.
# The full-scan beginning is the beginning of the first sub-scan.
tBeg=ssTimesFirst[0]
# The full-scan end is the end of the last sub-scan.
tEnd=ssTimesLast[1]

#-- Interpolate PWV to the tgtScan range. Find scan-center average and slope.

# Interpolated PWV in millimeters (original value is in meters).
pwvTgt=1000.0*np.interp([tBeg,tEnd],pwvTimes,pwv)

# Scan-center average PWV.
pwvAvg=0.5*(pwvTgt[0]+pwvTgt[1])

# PWV linear time-trend / slope.
pwvSlope=(pwvTgt[1]-pwvTgt[0])/(tEnd-tBeg)

#-- Return the average, time-slope, and the full-PWV array as a map.
return {'pwvAvg':pwvAvg, 'pwvSlope':pwvSlope, 'allPWVs':(pwvTimes,pwv)}

```

```
#-----

def 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_CALATMOPHERE 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.
    """

    #-- Extract the METEO data

    # Open the MS/ASDM_CALWVR table
    meteoTab = tbtool()
    meteoTab.open(vis+"/WEATHER")

    # Extract time and meteo quantities
    time=meteoTab.getcol('TIME')

    # As described in aU.getWeather(), the times may not be sorted in increasing
    # order and we do not use tb.query() here, which normally assures ordering.
    # We have to do sorting by increasing time manually.

    # Find the permuted indices.
    pIndex = np.argsort(time)

    # Apply the same permutation to all arrays.
    time=time[pIndex]
    pressure=meteoTab.getcol('PRESSURE')[pIndex]
    humidity=meteoTab.getcol('REL_HUMIDITY')[pIndex]
    temperature=meteoTab.getcol('TEMPERATURE')[pIndex]

    # Close the table.
    meteoTab.done()

    # Return the meteo characteristics
    return {'time': time, 'pressure': pressure, 'humidity':humidity,
            'temperature':temperature}

#-----

def getMeteoForScan(vis, scan, myMsmid=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. Thsi 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 dPWV, has a typical value about 3K, so difference in T_{atm} of 0.5K makes ~20% error!

```
"""
```

```
#-- Find the scan timing.
```

```
#- Msmd tool for scan-info extraction.
```

```
# Flag indicating, whether we had to create our own msmd tool for scan timing.  
ownMsmd=False
```

```
if (myMsmd==None) :  
    # No msmd tool supplied, create a new one and attach it to our MS.  
    myMsmd=msmdtool()  
    myMsmd.open(vis)  
    # Set the flag  
    ownMsmd=True  
    print('msmdtool attached to ',vis)
```

```
scanInfo=aU.computeDurationOfScan(scan,mymsmd=myMsmd,  
                                   returnSubscanTimes=True)
```

```
# Close only our own local msmd tool, if we have used the 'global' one,  
# leave it as is.
```

```
if (ownMsmd): myMsmd.done()
```

```
#- Analyse the scan-info results.
```

```
# Num. of subscans in the tgtScan.  
nSubScans=scanInfo[1]
```

```
# First and last subscan: The result is a map starting from 1, not an array.  
ssTimesFirst=scanInfo[2][1]  
ssTimesLast=scanInfo[2][nSubScans]
```

```
# The sub-scan times are 2-element arrays - subscans beginning and end.  
# The full-scan beginning is the beginning of the first sub-scan.  
tBeg=ssTimesFirst[0]  
# The full-scan end is the end of the last sub-scan.  
tEnd=ssTimesLast[1]
```

```
# Create scan time-sampling axis (100 samples per scan).  
scanTimes=tBeg+0.01*(tEnd-tBeg)*np.arange(101.0)
```

```
#-- Get meteo time series.
```

```

meteo=getMeteo(vis)

meteoTimes=meteo['time']
pressure=meteo['pressure']
humidity=meteo['humidity']
temperature=meteo['temperature']

#-- Interpolate the meteo characteristics to the scan time axis.

sPressure=np.interp(scanTimes,meteoTimes,pressure)
sHumidity=np.interp(scanTimes,meteoTimes,humidity)
sTemperature=np.interp(scanTimes,meteoTimes,temperature)

#-- Return the scan mean values.

return {'pressure':sPressure.mean(),
        'humidity':sHumidity.mean(),
        'temperature':sTemperature.mean()}

#-----

def 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 modela and its get***Path() routines.

    Result: Indeed, the non-dispersive part is constant over SPW 5 and its value
    is ~6.7e-3 m(!) per 1 mm of PWV. The dispersive part is very slightly
    changing around its average value of 1.0e-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 ~6.9e-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.
    """

    #-- Open and set the atm tool.
    myAtm=attool()
    setAtmTool(myAtm,vis, spw, scan)

    # Set the PWV for the model.
    myAtm.setUserWH2O(qa.quantity(pwv, 'mm'))

    #-- Calculate the paths per channel relative to 1mm of PWV.

    nChans=myAtm.getNumChan(0)

    # Allocate the arrays for dispersive and non-dispersive paths.
    pDisp=np.zeros(nChans)

```

```

pNonDisp=np.zeros(nChans)

for ch in range(nChans):

    pDisp[ch]=myAtm.getDispersivePathLength(nc=ch) ['value'] [0]/ pwv
    pNonDisp[ch]=myAtm.getNonDispersivePathLength(nc=ch) ['value'] [0]/ pwv

# Close the ATM tool.
myAtm.done()

# Return dispersive and non-dispersive paths in mm as a tuple.
return 1000.0*pDisp,1000.0*pNonDisp

#-----
#                               5. Calculate the proxies of differential PWV
#-----

def 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.
    """

    #-- Power temporal variations.

    # Get the measured power, preceded by times as MJD.
    mjd_src,Psrc=tgt_pow(vis,tgtScan,antID,spwAvg)

    # 1st index is pol., 2nd means channel num. We have just a single channel.
    Pxx=Psrc[0][0]
    Pyy=Psrc[1][0]

    # Subtract the time-averaged value over scan to get the variations only.
    PxxM=Pxx.mean()
    PyyM=Pyx.mean()

    dPxx=Pxx- PxxM
    dPyy=Pyx- PyyM

    #-- Calculate the "sensitivity factor S"

    #- Find G/G*, Trx, and Tinst

```

```

ctACD=acd_calTabs(vis,atmScan,antID,spwAvg)

iG=ctACD['iG']
iGstar=ctACD['iG*']
T_inst=ctACD['T_inst']

# What is the target? Should we use the solar-attenuation gain G* or
# the 'standard' G?
if (iGstar[0][0][0] == 1.0): # G* not set (has its default value) -> not Sun

    print('Solving for CALIBRATOR scan.')

    # G* was not calculated => non-solar target, use (inverse) G.
    # 1st index is pol., 2nd means channel num. We have just a single channel.
    iGx=iG[0][0]
    iGy=iG[1][0]

else:

    print('Solving for SOLAR scan.')

    # Solar target-scan: Use (inverse) G*, make an average from the values
    # derived for the beginning and the end of the scan.
    # 1st index is 'time' (just begin and end), 2nd means pol. and 3rd the
    # channel number. We have just a single channel.
    iGx=0.5*(iGstar[0][0][0]+iGstar[1][0][0])
    iGy=0.5*(iGstar[0][1][0]+iGstar[1][1][0])

Gx=1.0/ iGx
Gy=1.0/ iGy

# The 'instrumental' temperature (T_RX + contribution from the SQLD offset).
T_iXX=T_inst[0]
T_iYY=T_inst[1]

#- Find spectral slope (linear) for the source (+atm.) brightness temperature.

# Calculate the brightness temperature as a function of frequency and
# time using ACD calibration.
mjd_tb,Txx,Tyy=tgt_Tb(vis,atmScan,tgtScan,antID,spwRes)

# We use time-averaged (over a scan) values for the slope determination.
Txm=Txx.mean(axis=1)
Tym=Tyy.mean(axis=1)

# Trim the edge channels, these are always problematic.
nChans=len(Txm)

TxM=Txm[7:nChans-8]
TyM=Tym[7:nChans-8]

# Find the corresponding frequency axis. Trim the edges accordingly.
msmd.open(vis)
freqs = myMsmd.chanfreqs(spwRes)*1.0e-9 # in GHz
msmd.done()

```

```

fr=freqs[7:nChans-8]
# Find the entire SPW bandwidth [GHz].
BW=np.max(freqs)-np.min(freqs)

# Use linear (poly=1) regression to find the slope.
dTxx_df=np.polyfit(fr,TxM,1)[0]
dTyy_df=np.polyfit(fr,TyM,1)[0]

#-- Find the opacities and T_atm using the ATM model.

atmParams=wopacity(vis,spwRes,tgtScan)

# 1/sin(h) for the target.
airMass=atmParams['airmass']

# Effective gas temperature of the atmosphere.
T_atm=atmParams['T_atm'].mean()

# Zenith wet opacity for PWV=1.0.
tau_1z=atmParams['tau_1']

# The same quantity scaled by airmass.
tau_1=airMass*tau_1z

#- Extract its mean value and the ranges for high- and low-frequency ends.

tau_1M=tau_1.mean()

# Highest-frequency channel
hiChan=tau_1.size-1

# Opacity span over the SPW.
tau_1Hi=tau_1[hiChan]
tau_1Lo=tau_1[0]

#-- Now we can calculate the S-factors for each polarization extra.
SFxx=-Gx*tau_1M*(iGx*PxxM-T_iXX-T_atm)-Gx*BW/12.0*(tau_1Hi-tau_1Lo)*dTxx_df
SFyy=-Gy*tau_1M*(iGy*PyyM-T_iYY-T_atm)-Gy*BW/12.0*(tau_1Hi-tau_1Lo)*dTyy_df

# Diagnostic prints: Find out, how much each term contributes.

print('Psrc/G = Tsrc = ',iGx*PxxM,iGy*PyyM)
print('Tinst = ',T_iXX,T_iYY)
print('Tatm = ',T_atm)
print('dTsky/df = ',dTxx_df,dTyy_df)
print('SF_corr = ',Gx*BW/12.0*(tau_1Hi-tau_1Lo)*dTxx_df,
      Gy*BW/12.0*(tau_1Hi-tau_1Lo)*dTyy_df)

print('SF = ',SFxx,SFyy)

#-- Eventually, get the differential modelled PWVs.

dPWVxx=dPxx/SFxx
dPWVyy=dPyy/SFyy

# Return them as a tuple.

```

```
return (mjd_src,dPWVxx,dPWVyy)
```

```
#-----
```

```
def 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).

```
    """
```

```
#-- Power temporal variations.
```

```
# Get the measured power, preceded by times as MJD.
# Shape of the power Psrc is nPols x nChans x nIntegs
time_src,Psrc=tgt_pow(vis,tgtScan,antID,spwRes)
```

```
# Separate polarizations.
```

```
Pxx=Psrc[0]
Pyy=Psrc[1]
```

```
# Find the frequency and time dimensions.
```

```
nChans=Pxx.shape[0]
nIntegs=Pxx.shape[1]
```

```
#- Subtract <P>, the time-averaged value over scan to get the variations only.
```

```
# Calculate <P>. result is a vector of size nChans.
```

```
PxxM=Pxx.mean(axis=1,dtype=np.float64)
PyyM=Pyy.mean(axis=1,dtype=np.float64)
```

```
# We need the <P> vector (in frequency) to be a matrix (const in time).
# Define the time-dimension const (unit) vector.
```

```
tdUnitVec=np.ones(nIntegs)
# Perform the outer (tensor) product.
PxxMM=np.tensordot(PxxM,tdUnitVec,0)
PyyMM=np.tensordot(PyyM,tdUnitVec,0)
```

```
# Now we can subtract - the result is a matrix with shape nChans x nIntegs
```

```
dPxx=Pxx-PxxMM
dPyy=Pyy-PyyMM
```

```

#- Find G/G*, Trx, and Tinst
ctACD=acd_calTabs(vis,atmScan,antID,spwRes)

# iG and T_rx are of shape nPols x nChans, iGstar (solar) nPols x nChans x 2,
# the last dimension of size 2 representing 2 time instants at which the
# gain is measured (the 1st and last sub-scans of the solar scan).
iG=ctACD['iG']
iGstar=ctACD['iG*']
T_rx=ctACD['T_rx']

# What is the target? Should we use the solar-attenuation gain G* or
# the 'standard' G?
if (iGstar[0][0][0] == 1.0): # G* not set (has its default value) -> not Sun

    print('Solving for CALIBRATOR scan.')

    # G* was not calculated => non-solar target, use (inverse) G.

    iGx=iG[0]
    iGy=iG[1]

else:

    print('Solving for SOLAR scan.')

    # Solar target-scan: Use (inverse) G*, make an average from the values
    # derived for the beginning and the end of the scan.
    iGx=0.5*(iGstar[0][0]+iGstar[1][0])
    iGy=0.5*(iGstar[0][1]+iGstar[1][1])

# The 'instrumental' temperature - just T_RX in case of the AC SPW.
T_iXX=T_rx[0]
T_iYY=T_rx[1]

#-- Find the opacities and T_atm using the ATM model.

atmParams=wopacity(vis,spwRes,tgtScan)

# 1/sin(h) for the target.
airMass=atmParams['airmass']

# Effective gas temperature of the atmosphere.
# We use the scalar value representing the almost constant array - see
# the comment in wopacity(). Keep it constant but still a vector in frequency.
T_atm=atmParams['T_atm'].mean()*np.ones(nChans)

# Zenith wet opacity for PWV=1.0.
tau_1z=atmParams['tau_1']

# The same quantity scaled by airmass.
tau_1=airMass*tau_1z

#-- We are going to integrate (sum) over the defined frequency range.
# Some SPWs have, however, reversed frequency axis. For P, T_RX and G/G*
# this is not an issue (we are just summing backwards). Nevertheless,
# tau_1 has been calculated by atm model in wopacity() with the frequency
# axis always in increasing order. Hence, in reversed SPWs we reverse tau_1,

```

```

# too, in order to make all values having the same per-channel order.
# Moreover, we swap the args startChan/stopChan in reversed SPWs, so that
# they keep their intuitive meaning (corresponding to increasing frequency).

# Check the SPW definition.
myMsmc=msmdtool()
myMsmc.open(vis)
freqs = myMsmc.chanfreqs(spwRes)
myMsmc.done()

lastChan=len(freqs)-1

if (freqs[1] < freqs[0]):
    # Reversed-order SPW
    print("Reversed SPW, swapping start/stop channels.")
    startChan,stopChan = lastChan-stopChan,lastChan-startChan

    tau_1=np.flip(tau_1)

#-- In spectrally-resolved SPWs the gain/Trx may live in a different frequency
# domain than the data - see acd_calTabs(). We have to interpolate first.
# Note: tau_1 has the same freq. domain as science data as set in wopacity()
# just it may be in reversed order (solved above by flipping).

# Reveal the frequency domain for gains.
gainFreqs=ctACD['freqs']

# Just a very basic frequency-domain compatibility check. @todo If it does not
# work, check yet f[0] and df.
if (len(freqs) != len(gainFreqs)):

    print('Interpolating gain/T_rx from ',len(gainFreqs),' to ',len(freqs),
          ' frequency channels.')

    # Gain and target spw definitions differ: Do interpolate.
    iGx=np.interp(freqs,gainFreqs,iGx)
    iGy=np.interp(freqs,gainFreqs,iGy)

    Trx=np.interp(freqs,gainFreqs,Trx)
    Try=np.interp(freqs,gainFreqs,Try)

#-- Apply the integration limits to all relevant arrays.

freqs=freqs[startChan:stopChan]

iGx=iGx[startChan:stopChan]
iGy=iGy[startChan:stopChan]

Ti_XX=Ti_XX[startChan:stopChan]
Ti_YY=Ti_YY[startChan:stopChan]

dPxx=dPxx[startChan:stopChan]
dPyy=dPyy[startChan:stopChan]

```

```
PxxM=PxxM[startChan:stopChan]
```

```
PyyM=PyyM[startChan:stopChan]
```

```
tau_1=tau_1[startChan:stopChan]
```

```
T_atm=T_atm[startChan:stopChan]
```

```
#-- Calculate the dPWV by formula (1) in PR13 (August 2025).
```

```
# Again, we need the iG (inv. gain) vector (in freq.) as a matrix - see above.
```

```
iGxM=np.tensordot(iGx,tdUnitVec,0)
```

```
iGyM=np.tensordot(iGy,tdUnitVec,0)
```

```
# Frequency-integrated sky-temperature (not power like in dPWV()) variations.
```

```
dTxx=np.sum(iGxM*dPxx,axis=0)
```

```
dTyy=np.sum(iGyM*dPyy,axis=0)
```

```
# The sensitivity factors. They differ by G-factor from those in dPWV().
```

```
SF_XX=-np.sum((iGx*PxxM-Ti_XX-T_atm)*tau_1)
```

```
SF_YY=-np.sum((iGy*PyyM-Ti_YY-T_atm)*tau_1)
```

```
dPWVxx=dTxx/ SF_XX
```

```
dPWVyy=dTyy/ SF_YY
```

```
# Return them as a tuple, preceded by time.
```

```
return (time_src,dPWVxx,dPWVyy)
```

```
#-----
```

```
def 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 atmospheric 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 easily 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().
 """

#-- Static parameters

Reference frequency for the source spectrum - the LO frequency (in Hz)
 f_LO=1.98e11

Spectral index for the source. Spix=2.0 corresponds to R-J law.
Spix=0.0 represents 'gray' enhancements due to atmospheric-bubble lensing.
 spix=2.0

Integration limits for SPWs. To be done more selective, e.g., to exclude
stratospheric lines like O3.

startChan=7
 stopChan=120

#-- Average PWV over a scan per antenna: Required for tau calculations.

pwvMean=getPWVforAnt(vis,tgtScan,antID) ['pwvAvg']

#-- Matrices (so far empty placeholders) for sets of linear equations.

Axx=np.array([])
 Ayy=np.array([])

```

#-- RHS vectors (so far empty placeholders) for sets of linear equations.
Cxx=np.array([])
Cyy=np.array([])

#-- Loop over the SPWs.

spwIndex=0

for mySPW in spwList:

    #-- Raw powers: Temporal variations.

    # Get the measured power, preceded by times as MJD.
    mjd_src, Psrc=tgt_pow(vis, tgtScan, antID, mySPW)

    # Separate polarizations.
    Pxx=Psrc[0]
    Pyy=Psrc[1]

    # Find the frequency and time dimensions.
    nChans=Pxx.shape[0]
    nIntegs=Pxx.shape[1]

    #- Subtract <P>, the time-average over a scan to get the variations only.

    # Calculate <P>. Result is a vector of size nChans.
    PxxM=Pxx.mean(axis=1, dtype=np.float64)
    PyyM=Pyx.mean(axis=1, dtype=np.float64)

    # We need the <P> vector (in frequency) to be a matrix (const in time).
    # Define the time-dimension const (unit) vector.
    tdUnitVec=np.ones(nIntegs)
    # Perform the outer (tensor) product.
    PxxMM=np.tensordot(PxxM, tdUnitVec, 0)
    PyyMM=np.tensordot(PyyM, tdUnitVec, 0)

    # Now we can subtract - the result is a matrix with shape nChans x nIntegs
    dPxx=Pxx- PxxMM
    dPyy=Pyx- PyyMM

    #-- Calibration to Tsky: Find G/G* and Trx/Tinst

    ctACD=acd_calTabs(vis, atmScan, antID, mySPW)

    # iG and T_rx are of shape nPols x nChans, iG* (solar) nPols x nChans x 2,
    # the last dimension of size 2 representing 2 time instants at which the
    # gain is measured (the 1st and last sub-scans of the solar scan).
    iG=ctACD['iG']
    iGstar=ctACD['iG*']
    T_inst=ctACD['T_inst']

    # What is the target? Should we use the solar-attenuation gain G* or
    # the 'standard' G?
    if (iGstar[0][0][0] == 1.0): # G* not set (has its default value) -> not Sun

```

```

print('Solving for CALIBRATOR scan.')

# G* was not calculated => non-solar target, use (inverse) G.
iGx=iG[0]
iGy=iG[1]

else:

    print('Solving for SOLAR scan.')

    # Solar target-scan: Use (inverse) G*, make an average from the values
    # derived for the beginning and the end of the scan.
    iGx=0.5*(iGstar[0][0]+iGstar[1][0])
    iGy=0.5*(iGstar[0][1]+iGstar[1][1])
    #--> endif(solar)

    # The 'instrumental' temperature - just T_RX in case of the AC SPWs.
    T_iXX=T_inst[0]
    T_iYY=T_inst[1]

    #-- Atmosphere: Find the opacities and T_atm using the ATM model.

    # 1/sin(h) for the target.
    airMass=atmModel[mySPW]['airmass']

    # Effective gas temperature of the atmosphere.
    # T_atm=atmModel[mySPW]['T_atm'].mean()*np.ones(nChans)
    T_atm=atmModel[mySPW]['T_atm']

    # Zenith wet opacity for PWV=1.0.
    tau_lz=atmModel[mySPW]['tau_l']

    # Zenith wet opacity for the actual PWV.
    tau_wz=atmModel[mySPW]['tau_w']

    # Zenith dry opacity.
    tau_dz=atmModel[mySPW]['tau_d']

    # Wet unit opacity scaled by airmass.
    tau_l=airMass*tau_lz
    # Wet actual opacity scaled by airmass.
    tau_w=airMass*tau_wz
    # Dry opacity scaled by airmass.
    tau_d=airMass*tau_dz
    # Total opacity scaled by airmass. Do NOT use tau_w - it is an average
    # over entire ALMA array, see wopacity(). We rather use meanPwv per antenna
    # and tau_l.
    tau_t=pwvMean*tau_l+tau_d

    #-- Check the SPW definition. We branch for resolved and averaged SPWs.

    myMsmd=msmdtool()
    myMsmd.open(vis)
    freqs = myMsmd.chanfreqs(mySPW)

```

```

myMsmc.done()

nChans=len(freqs)

if (nChans > 1):

    #-- This is a channel-resolved SPW.

    # We are going to integrate (sum) over the defined frequency range.
    # Some SPWs have, however, reversed frequency axis. For P, T_RX and G/G*
    # this is not an issue (we are just summing backwards). Nevertheless,
    # tau_* have been calculated by atm model in wopacity() with the frequency
    # axis always in increasing order. Hence, in reversed SPWs we reverse tau
    # too, in order to make all values having the same per-channel order.
    # Moreover, we swap the args startChan/stopChan in reversed SPWs, so that
    # they keep their intuitive meaning (corresponding to increasing
    # frequency).

    lastChan=nChans-1

    if (freqs[1] < freqs[0]):
        # Reversed-order SPW
        print("Reversed SPW, swapping start/stop channels.")
        startChan, stopChan = (lastChan-stopChan), (lastChan-startChan)

        tau_l=np.flip(tau_l)
        tau_w=np.flip(tau_w)
        tau_d=np.flip(tau_d)
        tau_t=np.flip(tau_t)
        T_atm=np.flip(T_atm)

    # In resolved SPWs the gain/Trx may live in a different frequency
    # domain than the data - see acd_calTabs(). We have to interpolate first.
    # Note: tau_* have the same freq. domain as science data as set in
    # wopacity() just it may be in reversed order (solved above by flipping).

    # Reveal the frequency domain for gains.
    gainFreqs=ctACD['freqs']

    # Just a very basic frequency-domain compatibility check. @todo If it does
    # not work, check yet f[0] and df.
    if (nChans != len(gainFreqs)):
        print('Interpolating gain/T_rx from ',len(gainFreqs),' to ',len(freqs),
              ' frequency channels.')

    # Gain and target spw definitions differ: Do interpolate.
    iGx=np.interp(freqs,gainFreqs,iGx)
    iGy=np.interp(freqs,gainFreqs,iGy)

    Ti_XX=np.interp(freqs,gainFreqs,Ti_XX)
    Ti_YY=np.interp(freqs,gainFreqs,Ti_YY)

    #-- Apply the integration limits to all relevant arrays.

    freqs=freqs[startChan:stopChan]

```

```

iGx=iGx[startChan:stopChan]
iGy=iGy[startChan:stopChan]

Ti_XX=Ti_XX[startChan:stopChan]
Ti_YY=Ti_YY[startChan:stopChan]

dPxx=dPxx[startChan:stopChan]
dPyy=dPyy[startChan:stopChan]

PxxM=PxxM[startChan:stopChan]
PyyM=PyyM[startChan:stopChan]

tau_l=tau_l[startChan:stopChan]
tau_w=tau_w[startChan:stopChan]
tau_d=tau_d[startChan:stopChan]
tau_t=tau_t[startChan:stopChan]
T_atm=T_atm[startChan:stopChan]

#-- Partial sensitivity factors for resolved SPWs

# Again, we need the iG (inv. gain) vector (in freq.) as a matrix.
iGxM=np.tensordot(iGx,tdUnitVec,0)
iGyM=np.tensordot(iGy,tdUnitVec,0)

# Frequency-averaged sky-temperature (not power like in dPWV()) variations
# This is going to be an RHS vector of our set of equations.
dTxx=np.mean(iGxM*dPxx,axis=0)
dTyy=np.mean(iGyM*dPyy,axis=0)

# The sensitivity factor for WV variations. Column 1 in matrix.
SW_XX=-np.mean((iGx*PxxM-Ti_XX-T_atm)*tau_l)
SW_YY=-np.mean((iGy*PyyM-Ti_YY-T_atm)*tau_l)

# The sensitivity factor for source-brighthness vars. Col 2 in matrix.
SB_XX=np.mean((freqs/f_LO)**spix*np.exp(-tau_t))
SB_YY=SB_XX

#--> endif(resolved spw)

```

else:

```

#-- This is a channel-averaged AC or SQLD spw.

# Calculate channel-averages over the SPW for atmospheric model params.
tau_lM=tau_l.mean()
tau_wM=tau_w.mean()
tau_dM=tau_d.mean()
tau_tM=tau_t.mean()
T_atmM=T_atm.mean()

print('Opacities w1:',tau_lM,'wet:',tau_wM, 'dry: ',tau_dM,
      'Total:',tau_tM, 'Transmisivity:',np.exp(-tau_tM))

#-- Integrated sensitivity factors for averaged SPWs.
# The index '0' represents the single frequency channel we have.

```

```

# Frequency-averaged sky-temperature variations.
# This is going to be an RHS vector of our set of equations.
dTxx=iGx[0]*dPxx[0]
dTyy=iGy[0]*dPyy[0]

# The SQLD spws have 12x higher cadence than the ACs. Rebin the SQLD
# data in time to have a homogeneous series. Do the same for time axis MJD
if mySPW in [0,1,2,3,13,14,15,16]:
    dTxx=rebin(dTxx,12)
    dTyy=rebin(dTyy,12)
    mjd_src=rebin(mjd_src,12)

# Shortcut for 'net' sky temperature, cleaned by atm. and inst. contribs.
TnetXX=iGx[0]*PxxM[0]-T_iXX[0]-T_atmM
TnetYY=iGy[0]*PyyM[0]-T_iYY[0]-T_atmM

# The sensitivity factor for total WV variations. Column 1 in matrix.
SW_XX=-TnetXX*tau_1M
SW_YY=-TnetYY*tau_1M

# The sensitivity factor for source-brightness vars. Col 2 in matrix.
SB_XX=(freqs[0]/f_LO)**spix*np.exp(-tau_tM)
SB_YY=SB_XX

"""
Alternative/additional decompositions of a signal. Using either option
1 or 2 below, we arrive to in total 3 unknown variables:
[wet-slow, wet-fast, source-brightness] for Option 1, and
[wet-total, dry, source-brightness] for Option 2.
@todo: If it shall be implemented, do it for chan-resolved SPW above, too.

Option 1: Use these two instead of total wet component.

# Slow WV variations: Humidity / WV rel. abundance.
SWS_XX=-TnetXX*tau_wM
SWS_YY=-TnetYY*tau_wM

# Fast WV variations: Air-mixture total density.
SWF_XX=-TnetXX*tau_tM
SWF_YY=-TnetYY*tau_tM

Option 2: Add this expression for the dry component to the total wet vars.

# Total dry variations.
SD_XX=-TnetXX*tau_dM
SD_YY=-TnetYY*tau_dM

As it is shown in PR15, both options are mathematically equivalent, we
just decompose to two different bases. However, none is working - either
the system is ill-conditioned (Option 1) or we are just amplifying noise
in because very low sensitivity factor for dry component (Option 2).
"""
#--> endelse(averaged spw)

```

```

#-- Set of linear equations for dW and dB: Assembly the system.

# Set the sensitivity factors for the current SPW as a row in the matrix.
Axx=np.append(Axx, [SW_XX, SB_XX])
Ayy=np.append(Ayy, [SW_YY, SB_YY])

# Set the sky temperature variations in the current SPW as the RHS vectors.
Cxx=np.append(Cxx, dTxx)
Cyy=np.append(Cyy, dTyy)

spwIndex+=1

#--> endfor(spws)

# The last spw index is the spw count.
numSpw=spwIndex

# Re-shape the matrix to be nVars x numSpw array
Axx=np.reshape(Axx, (numSpw,-1))
Ayy=np.reshape(Ayy, (numSpw,-1))

# Re-shape the RHS vectors to be nTimeSamples x numSpw array
Cxx=np.reshape(Cxx, (numSpw,-1))
Cyy=np.reshape(Cyy, (numSpw,-1))

#-- We most likely have over-determined system. Use the least-square
# approach to get quadratic form and its minimizer

# Transposed matrices.
AxxT=np.transpose(Axx)
AyyT=np.transpose(Ayy)

# Quadratic-form matrix 2x2 (we have two variables).
AxxK=np.matmul(AxxT,Axx)
AyyK=np.matmul(AyyT,Ayy)

# Modified RHS (nTimeSamples x 2)
CxxK=np.matmul(AxxT,Cxx)
CyyK=np.matmul(AyyT,Cyy)

#-- Solve the nVarsx x nVars set of equations, one per polarization,
# using Cramer's rule. So far just for 2 x 2 - need to be updated in case
# of extended decomposition (see commented 'Alternative...' part above).

#- Pol XX

A11=AxxK[0,0]
A12=AxxK[0,1]

A21=AxxK[1,0]
A22=AxxK[1,1]

print('A11=',A11,', A12=',A12)
print('A21=',A21,', A12=',A22)

```

```

C1=CxxK[0]
C2=CxxK[1]

print('C1=',C1)
print('C2=',C2)

iD=1.0/ (A11*A22-A12*A21) # Inverse determinant of the system.

print("Inverse determinant (XX): ",iD)

dWWxx=iD* (A22*C1-A12*C2)
dBBxx=iD* (A11*C2-A21*C1)

#- Pol YY

A11=AyyK[0,0]
A12=AyyK[0,1]

A21=AyyK[1,0]
A22=AyyK[1,1]

print('A11=',A11,', A12=',A12)
print('A21=',A21,', A12=',A22)

C1=CyyK[0]
C2=CyyK[1]

print('C1=',C1)
print('C2=',C2)

iD=1.0/ (A11*A22-A12*A21) # Inverse determinant of the system.
print("Inverse determinant (YY): ",iD)

dWWyy=iD* (A22*C1-A12*C2)
dBByy=iD* (A11*C2-A21*C1)

#-- Diagnostic prints & plots

#- All variations should essentially have zero mean.

print("dWxx, dWyy mean: ", dWWxx.mean(),dWWyy.mean())
print("dBxx, dByy mean: ", dBBxx.mean(),dBByy.mean())

#- Make a plot comparing the measured variations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

#-Set the plotting device.

plt.rcParams["figure.figsize"] = (30,70)
# Make XX and YY plots as subplots (horizontally)
ff,sps=plt.subplots(3,2)

# X-axis label just for the bottom sub-plot.
sps[2,0].set_xlabel('Time [MJDsec]')
sps[2,1].set_xlabel('Time [MJDsec]')

```

```

# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Color-map for SPWs in increasing order, incl. line-style.
# We have minimum 2 and maximum 12 SPWs in the spwList argument.
# If we use all 12 spws, linestyle correspondence to the spw type follows:
cslime=['-r','-g','-b','-k',          # AC resolved
        '--r','--g','--b','--k',    # AC integrated
        ':r':g':b':k']              # SQLD

#- Calculate the errors for overdetermined set of eqs.

# Compose the variables into a stack of vectors nTimeSamples x 2
varVecXX=np.array([dWWxx,dBBxx])
varVecYY=np.array([dWWyy,dBByy])

# Subtract 'theoretical' and real sky-temperature variations.
errVecXX=np.matmul(Axx,varVecXX)-Cxx
errVecYY=np.matmul(Ayy,varVecYY)-Cyy

#- Make the analytical plots - least-square residuals etc.

# *** 1st row
# Left panel: XX errs
spwCount=0
for spw in spwList:
    sps[0,0].plot(mjd_src,errVecXX[spwCount],cslime[spwCount])
    spwCount+=1

# Right panel: YY errs
spwCount=0
for spw in spwList:
    sps[0,1].plot(mjd_src,errVecYY[spwCount],cslime[spwCount])
    spwCount+=1

# *** 2nd row
# Left panel: XX RHS
spwCount=0
for spw in spwList:
    sps[1,0].plot(mjd_src,Cxx[spwCount],cslime[spwCount])
    spwCount+=1

# Right panel: YY RHS
spwCount=0
for spw in spwList:
    sps[1,1].plot(mjd_src,Cyy[spwCount],cslime[spwCount])
    spwCount+=1

# *** 3rd row
# Left panel: XX pol for T_src variations contrib to the total.
spwCount=0
for spw in spwList:
    sps[2,0].plot(mjd_src,Axx[spwCount,1]*dBBxx,cslime[spwCount])
    spwCount+=1

```

```

# Right panel: YY pol for T_src variations contrib to the total.
spwCount=0
for spw in spwList:
    sps[2,1].plot(mjd_src,Ayy[spwCount,1]*dBByy,csline[spwCount])
    spwCount+=1

#- Save the plots to a file and close.
plt.savefig('./dwdp_4SPWs/errs_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Make the *solution* plots: dW(t) and dB(t).

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Left panel: Fast variations - apparent target brightness
sp1.plot(mjd_src,dBBxx, '-r')
sp1.plot(mjd_src,dBByy, '-b')

# Right panel: Slow variations - total PWV (compression + humidity).
sp2.plot(mjd_src,dWWxx, '-r')
sp2.plot(mjd_src,dWWyy, '-b')

plt.savefig('./dwdp_4SPWs/dWdB/dWdB_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Return the dW and dB variations preceded by time.
# Add the (original over-determined) matrix and the RHS vector, too.

# Compose the XX and YY polarization component to a single arrays
dWW=np.array([dWWxx,dWWyy])
dBB=np.array([dBBxx,dBByy])
AA=np.array([Axx,Ayy])
CC=np.array([Cxx,Cyy])

# Return all as a map/dictionary.
return {'time':mjd_src, 'dW':dWW, 'dB':dBB, 'A':AA, 'C':CC}

#-----

def 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 ehpemeris-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_I` 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' temnperture, 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 pther 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 inrinsic 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 channnel-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 ton 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).
"""*

#-- Static parameters

#- Dirs

Output/diagnostics plots.

plotDir='./sciPWVs'

#- Tools, global data

Flags used to check, whether we should close the tool later.

ownMS=False

ownMsmd=False

Open own tools if 'global' not provided

if (not myMS):

Open the ms tool

myMS=mstool()

myMS.open(vis)

ownMS=True

if (not myMsmd):

Open the ms tool.

myMsmd=msmdtool()

myMsmd.open(vis)

ownMsmd=True

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

if (not acdLoads):

acdLoads=aU.getLoadTemperatures(vis, doplot = False,
warnIfNoLoadTemperatures = True,
mymsmd=myMsmd)

#- Source

Initial guess of the solar brightness temperature.

SD calibration script: Reference T_b for solar disc center @198GHz

TsunCenter=6300.0 # K

Solar radius.

Rsun=696000 # km

*# *** This part is observation specific - need to read ephemeris !!!*

Heliocentric coordinates for Dec 2023 Band 5 observation.

```

x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020).
b_2=-1127.0 # = 2x (-563.5K) as per Table 2 in Allisandrakis et al. (2020).

# Brightness temoerature corrected to limb brightening - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu)

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

#- SPWs

# Integration limits for resolved AC SPWs.
startChan=8
stopChan=120

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
eta_F=0.90

# Extract the ambient temperature for current antenna and scan.
# It determines the rear-spillover contribution to noise.
Tamb = acdLoads[antID][atmScan]['amb']

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
w0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',w0,' @airMass = ',airMass)
print('Estimated T_B = ',Tsun00)

```

```

print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#-- Read-out the sky+src brightness temperatures for all SPWs in the spwList.

# Store the frequency-averaged sky+src temperatures in a map
# spwId -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW
# id, which is sparse in the set of integers.
Txx={}
Tyy={}

for sciSPW in spwList:

    # Read ACD calibrated sky temperatures for the target defined by targetScan.
    time,TxxS,TyyS=tgt_Tb(vis,atmScan,tgtScan,antID,sciSPW,
                           myMS=myMS,myMsmid=myMsmid,myAtm=myAtm,acdLoads=acdLoads)

    # We have different handling for frequency-resolved and integrated SPWs.
    nChans=TxxS.shape[0]

    if (nChans > 1):

        # Resolved AC SPWs.
        # Trim the spectral edges by posing the integration limits and average
        # over the remaining channels.

        TxxM=np.mean(TxxS[startChan:stopChan],axis=0)
        TyyM=np.mean(TyyS[startChan:stopChan],axis=0)

    else:

        # Integrated ACs or SQLDs.
        # The single channel already represents a mean over the band-pass.

        TxxM=TxxS[0]
        TyyM=TyyS[0]

        # The SQLD spws have 12x higher cadence than the ACs. Rebin the SQLD
        # data in time to have a homogeneous series. Do the same for time axis.
        if sciSPW in [0,1,2,3,13,14,15,16]:
            TxxM=rebin(TxxM,12)
            TyyM=rebin(TyyM,12)
            time=rebin(time,12)

        # Append the frequency-averaged time series for each SPW to the Tsky arrays.
        Txx[sciSPW]=TxxM
        Tyy[sciSPW]=TyyM

print('SPW ',sciSPW,': <Tsky+src_XX> = ',Txx[sciSPW].mean(),' +/- ',
      Txx[sciSPW].std())
print('          <Tsky+src_YY> = ',Tyy[sciSPW].mean(),' +/- ',

```

```

        Tyy[sciSPW].std())

#-->endfor(sciSpws in list)

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# The data are read-out, we may close the (local) tools.
if (ownMsmd): myMsmd.done()
if (ownMS): myMS.close()

#-- Allocate arrays to be used for matrix A and the RHS-vector C calculations
# from the ATM model. They have just nAtmSPWs components.
nAtmSPWs=myAtm.getNumSpectralWindows()
nChans=myAtm.getNumChan(0)

# Allocate the model sky (source + atm) temperature.
TskyXX=np.zeros(nAtmSPWs)
TskyYY=np.zeros(nAtmSPWs)

# Sensitivity factors for dW and dB (Jacobian elements).
SWxx=np.zeros(nAtmSPWs)
SWyy=np.zeros(nAtmSPWs)

SBxx=np.zeros(nAtmSPWs)
SByy=np.zeros(nAtmSPWs)

# Frequencies of channels in our ATM SPWs.
spw_freqs=np.zeros((nAtmSPWs,nChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6 # MHz to Hz

    spw_freqs[atmSpw]=fMin+df*np.arange(nChans)

#-- Improve a guess of the solar brightness temperature.

TsolEst=np.zeros(numSpw)

spwCount=0
for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    # Sky brightness in a given SPW for the current PWV estimate.
    TbsEst=myAtm.getAverageTebbSky(atmSPW,qa.quantity(w0,'mm')) ['value'] [0]

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW) [1])

```

```

# Water-vapour related 'wet' opacity
tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
# Total opacity (spectral).
opacity=airMass*(tau_wet+tau_dry)

# Mean (spw-averaged) transmissivity
transM=np.mean(np.exp(- opacity))
# Frequency-dependent part of sol-flux spectrum weighted by transmissivity.
SFtrans=np.mean(np.exp(- opacity)* np.log(spw_freqs[atmSPW]/ f_LO))

# <Tsky+src> estimation from measurements, averaged over time and pols.
TskyEst=0.5*(Txx[sciSPW].mean()+Tyy[sciSPW].mean())

# Solar <Tsrc> temperature estimation in the current SPW.
TsolEst[spwCount]=(TskyEst-(1.0-eta_F+eta_l)*TbsEst
                  - eta_l*eta_F*b_2*SFtrans
                  - (1.0-eta_l)*Tamb)/(eta_l*eta_F*transM)

spwCount+=1

#--> endfor(sci SPWs)

print("Estimated Tsun from sci SPWS: ",TsolEst)

# Make a new initial guess of Tsun as an average.
Tsun00=TsolEst.mean()

print("Tsun = ",Tsun00,"+/-",TsolEst.std())

# Set it as the starting value for iterations.
Tsun0xx=Tsun00
Tsun0yy=Tsun00

#-- Average Solar-Flux spectrum (dimension-less, normalised to 1.0 at f=f_LO)

# It has the same dimensions as spw_freqs, i.e. nAtmSPWs x nChans
SF=1.0+b_2/ Tsun00*np.log(spw_freqs/ f_LO)

#--- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01
dbMean_max=0.01

# Initial values for the dW and dL means - anything bigger then the stoppers.
dW_mean=0.5
dB_mean=1.0

# Iteration counter.
itCount=0

```

```

#-- Iterations: Minimize the mean value of variations.

while (((abs(dw_mean) > dwMean_max) or (abs(db_mean) > dbMean_max))
       and (itCount < 20)):

    #- Prepare/reset the linearized system of equations.

    # (Jacobian linearization) Matrices (so far empty placeholders).
    JJ=np.array([])

    # RHS vectors (so far empty placeholders).
    CC=np.array([])

    # Rezidual mean sky temperature - after subtratced brightness of the
    # atmosphere (+CMB) and (attenuated) source. We keep it for all sci SPWs and
    # both polarizations and print it out at the end of every iteration cycle
    # as a diagnostics of convergence and solution quality.

    resTxx={}
    resTyy={}

    # Mapping between spw IDs and the row number in the system of equations.
    # We need it for final diagnostic plots. It may be non-trivial as some
    # spw:pol pairs may be flagged and ignored, casuing irregularity in the map.

    spw2rowXX={}
    spw2rowYY={}

    #- Re-calculate ATM model with updated <PWV> and <Tsun>.

    # Set current estimation of PWV to the ATM model.
    myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

    #- Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
    # Some factors depend only on the ATM mmodel, not the sciSPW data.
    for atmSPW in range(nAtmSPWs):

        # Model value of the sky/atm brightness with current pwv.
        TbsW0=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']

        # Slope (centered difference) of the atm brightness with current pwv.
        TbsW0H=myAtm.getTebbSkySpec(atmSPW,
                                     qa.quantity(w0+dws, 'mm'))[1]['value']
        TbsW0L=myAtm.getTebbSkySpec(atmSPW,
                                     qa.quantity(w0-dws, 'mm'))[1]['value']

        dTdW=(TbsW0H-TbsW0L)/(2.0*dws)

    #- Opacities & transmisivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Wet opacity per 1mm PWV scaled by airmass.

```

```

tau_l=airMass*tau_wet/ w0
# Total opacity (spectral).
opacity=airMass* (tau_wet+tau_dry)
# Transmissivity (spectral).
trans=np.exp(- opacity)

#- Sensitivities = Jacobian elements. Weighted averages over the SPW.

# Sensitivity factors for PWV.
SWxx[atmSPW]=(- eta_l* eta_F* Tsun0xx* np.mean(trans* tau_l* SF[atmSPW])
              +(1.0- eta_F+eta_l)* np.mean(dTdW))

SWyy[atmSPW]=(- eta_l* eta_F* Tsun0yy* np.mean(trans* tau_l* SF[atmSPW])
              +(1.0- eta_F+eta_l)* np.mean(dTdW))

# Sensitivity factors for the (apparent) solar-brighthness temperature.
SBxx[atmSPW]=eta_l* eta_F* np.mean(trans* SF[atmSPW])
SByy[atmSPW]=eta_l* eta_F* np.mean(trans* SF[atmSPW])

#- T_sky model. Weighted averages over the SPW.
TskyXX[atmSPW]=(eta_l* eta_F* Tsun0xx* np.mean(trans* SF[atmSPW])
               +(1.0- eta_F+eta_l)* np.mean(TbsW0)
               +(1.0- eta_l)* Tamb)

TskyYY[atmSPW]=(eta_l* eta_F* Tsun0yy* np.mean(trans* SF[atmSPW])
               +(1.0- eta_F+eta_l)* np.mean(TbsW0)
               +(1.0- eta_l)* Tamb)

#-->endfor(ATM spws)

#-- Loop over all sciSPWS in the list.

# Count the actual number of equations - some rows may be flagged.
eqCount=0

for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    #- Separate procedure by polarizations

    #- Pol XX

    # Full identificator of data: sciSPW:pol
    sciDataID=str(sciSPW)+' :XX'

    if (sciDataID not in flaggedData):

        # Set the sensitivity factors for the current SPW and pol as a row
        # in the linearized-system Jacobian matrix.
        JJ=np.append(JJ, [SWxx[atmSPW], SBxx[atmSPW], 0.0])

        # RHS: Txx/Tyy with their stationary (ATM model-based) part subtracted.
        dTxx=Txx[sciSPW]- TskyXX[atmSPW]

```

```

CC=np.append(CC,dTxx)

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTxx[sciSPW]=dTxx.mean()

# Mapping between SPW:XX and equation number.
spw2rowXX[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#- Pol YY

# Full identificator of data: sciSPW:pol
sciDataID=str(sciSPW)+' :YY'

if (sciDataID not in flaggedData):

    # Set the sensitivity factors for the current SPW and pol as a row
    # in the linearized-system Jacobian matrix.
    JJ=np.append(JJ, [SWyy[atmSPW],0.0,SByy[atmSPW]])

    # RHS: Txx/Tyy with their stationary (ATM model-based) part subtracted.
    dTyy=Tyy[sciSPW]-TskyYY[atmSPW]
    CC=np.append(CC,dTyy)

    # Residua of sky tempereture for mean PWV and Tsun (operation point).
    resTyy[sciSPW]=dTyy.mean()

    # Mapping between SPW:YY and equation number.
    spw2rowYY[sciSPW]=eqCount

    eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#--> endfor(all sciSpws in the list)

# Re-shape the matrix to be nVars x numEqations array
JJ=np.reshape(JJ,(eqCount,3))

# Re-shape the RHS vectors to be nTimeSamples x numEquations array
CC=np.reshape(CC,(eqCount,-1))

#- We most likely have over-determined system. Use the least-square
# approach to get quadratic form and its minimizer ('normal' equations).

# Transposed matrices.
JJ_T=np.transpose(JJ)

# Quadratic-form matrix 3x3 (we have three variables).

```

```

J_KK=np.matmul(JJ_T, JJ)

# Modified RHS (nTimeSamples x nVars)
C_KK=np.matmul(JJ_T, CC)

# Solve the nVarsx x nVars set of so called normal equations.
varVec=np.linalg.solve(J_KK, C_KK)

# Decompose the solution vector to our components dW, dBxx, and dByy.

dW = varVec[0]
dBxx=varVec[1]
dByy=varVec[2]

#- All variations should tend to zero means.

dW_mean=dW.mean()
dBxx_mean=dBxx.mean()
dByy_mean=dByy.mean()

dB_mean=np.sqrt(dBxx_mean**2+dByy_mean**2)

# Update stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0xx+=dBxx_mean
Tsun0yy+=dByy_mean

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and Tsun after completed iteration.
print('..... After iteration no. ', itCount)
print('Mean PWV for the scan, ', tgtScan, ' w0 = ', w0,
      ", mean Tsun (XX/YY) = ", Tsun0xx, '/', Tsun0yy)

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ", dW_mean)
print("dB mean: ", dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
resTxxArr=np.array(list(resTxx.values()))
resTyyArr=np.array(list(resTyy.values()))

print("res Txx: ", resTxx, "mean +/- RMS: ",
      resTxxArr.mean(), "+/-", resTxxArr.std())
print("res Tyy: ", resTyy, "mean +/- RMS: ",
      resTyyArr.mean(), "+/-", resTyyArr.std())

print('.....')

#--> endwhile(non-zero-means)

#-- Diagnostic plots.
# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

```

```

#-Set the plotting device.

plt.rcParams["figure.figsize"] = (30,50)
# Make XX and YY plots as subplots (horizontally)
ff,sps=plt.subplots(3,2)

# X-axis label just for the bottom sub-plot.
sps[2,0].set_xlabel('Time [MJDsec]')
sps[2,1].set_xlabel('Time [MJDsec]')

# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Color-map for SPWs, incl. line-style.
# We have minimum 2 and maximum 12 SPWs in the spwList argument.
# If we use all 12 spws, linestyle correspondence to the spw type follows:
csline={5: '-r',7: '-g',9: '-b',11: '-k',          # AC resolved
        6: '--r',8: '--g',10: '--b',12: '--k',     # AC integrated
        0: ':r',1: ':g',2: ':b',3: ':k'}          # SQLD

#- Calculate the errors for overdetermined set of eqs.

# Subtract 'theoretical' and real sky+src temperature variations.
errVec=np.matmul(JJ,varVec)-CC

#- Make the *analytical* plots - least-square residuals etc.

# XX pol

for spw,eqRow in spw2rowXX.items():

    # Panel row 1: Residuals/errors.
    sps[0,0].plot(time,errVec[eqRow],csline[spw])
    # Panel row 2: RHS vectors = measured Tsky+src minus model Tsky.
    sps[1,0].plot(time,CC[eqRow],csline[spw])
    # Panel row 3: T_b variation contribution to total Tsky+src
    sps[2,0].plot(time,JJ[eqRow,1]*dBxx,csline[spw])

# YY pol

for spw,eqRow in spw2rowYY.items():

    # Panel row 1: Residuals/errors.
    sps[0,1].plot(time,errVec[eqRow],csline[spw])
    # Panel row 2: RHS vectors = measured Tsky+src minus model Tsky.
    sps[1,1].plot(time,CC[eqRow],csline[spw])
    # Panel row 3: T_b variation contribution to total Tsky+src
    sps[2,1].plot(time,JJ[eqRow,2]*dByy,csline[spw])

#- Save the plots to a file and close.
plt.savefig(plotDir+'/errs_ant_'+str(antID)+'.png')
plt.show()
plt.close()

```

```

#-- Make the *solution* plots: dW(t) and dB(t).

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff, (sp1, sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Left panel: Fast variations - apparent target brightness
sp1.plot(time, dBxx, '-r')
sp1.plot(time, dByy, '-b')

# Right panel: Slow variations - total PWV (compression + humidity).
sp2.plot(time, dW, '-g')

plt.savefig(plotDir+'dWdB/dWdB_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Prepare resulting data for export.

# Add variations to the mean (scan-averaged) values.
TsunXX=Tsun0xx+dBxx
TsunYY=Tsun0yy+DByy

pwv=w0+dW

# 'Compactify' the per-polarization arrays.
Tsun=np.array([TsunXX, TsunYY])

return {'time':time, 'pwv':pwv, 'T_B':Tsun, 'J':JJ, 'C':CC}

#-----

def dWdBdG(vis, atmScan, tgtScan, myAtm, antID, spwList, flaggedData=[],
          myMS=None, myMsmd=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  $P_{inst}=G*Trx+P_o$  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 tempeprature 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

```

easily 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 than 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()` auxiliary 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.

"""

#--- Static parameters

#- Dirs

Output/diagnostics plots.

`plotDir='./sciPWVs_GP'`

`if (not os.path.exists(plotDir)): os.makedirs(plotDir)`

`if (not os.path.exists(plotDir+'/'+'dWdB')): os.makedirs(plotDir+'/'+'dWdB')`

#- Tools, global data

Flags used to check, whether we should close the tool later.

`ownMS=False`

`ownMsmid=False`

Open own tools if 'global' not provided

`if (not myMS):`

Open the ms tool

`myMS=msstool()`

`myMS.open(vis)`

`ownMS=True`

`if (not myMsmid):`

Open the ms tool.

`myMsmid=msmidtool()`

```

myMsmd.open(vis)
ownMsmd=True

# Create our dictionary with ACD load temperatures (ambient, hot) if needed.
if (not acdLoads):
    acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                     warnIfNoLoadTemperatures = True,
                                     mymsmd=myMsmd)

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.3 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff (arcsec, incl. cos(delta)): 822   155.8
'''
x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightenning correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temoerature corrected to limb brightenning - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu)

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

#- SPWs

# Integration limits for resolved AC SPWs.
startChan=8
stopChan=120

```

```

fChans=range(startChan,stopChan)

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
eta_F=0.90

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# Weight of the ACD calibration equations in the system. Essentially the
# number, with which appended block of the ACD equations is repeated in order
# to balance the much larger sciData-equations block. Infinity means that
# the instrumental parameters G and P_inst are essentially fixed and given
# by results of the acd_calTabs(). Low weight, on the other hand, leads to
# an ill-coinditioned system.
# The selected value of 200 has been found experimentally: With lower value
# we have a large error in the instrumental part of the solution.
# On the other hand, increasing beyond this value increseases the error in
# the natural part with no significant improvement in the instrumental error.
# See dWdBdG_calWeights(), which has been designed for exerimentation with
# the calWeight parameter.
calWeight=200.0

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd)['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,' @airMass = ',airMass)
print('Estimated T_B = ',1000.0*Tsun00)
print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#--- Read-out the sky+src powers for all SPWs in the spwList.

# Store the frequency-averaged sky+src powers in a map.
# spwId -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW
# id, which is sparse in the set of integers.
Pxx={}

```

```

Pyy={}

#- Similar maps for the gains and instrumental powers P_inst=G*T_inst.
# Here, all gains and P_inst are, in fact, the G* and P_inst* values for the
# higher-attenuation solar regime. In order to append the ACD calibration
# equations, whose measurements are performed in the norma-attenuation regime
# in the preceding ATM scan, we determine also the attenuation factors
# AF=G*/G and related power offsets AO.

# Frequency-averaged gains. The scale for the spectral-resolved norm. gains.
Gxx={}
Gyy={}

# Attenuation factors G/G*=(P_sky-P0)/(P_off-Po). Attributed to every-spw
# Gxx & Gyy, but actually calculated just in SQLD spws, the rest is mapped.
AFxx={}
AFyy={}

afMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

# Attenuation-based power offsets AO=-Po*(P_sky-P_off)/(P_off-Po).
# The same holds as for the AFs - we use afMap.
AOxx={}
AOyy={}

# Spectral gain normalized to 1.0. Actual ACD-based values in spectral AC spws,
# a trivial 1.0 ndarray for integrated-AC and SQLD spws.
gXX={}
gYY={}

# Frequency-averaged instrumental powers.
PIxx={}
PIyy={}

#- Finally, the raw ACD-calibration powers.

# Ambient-load powers.
PxxAmb={}
PyyAmb={}

# Hot-load powers.
PxxHot={}
PyyHot={}

#-- Read the data and ACD calibration tables in a loop over all sciSPWs.

for sciSPW in spwList:

    #- Raw data & ACD tables

    # Read raw science powers for the target defined by targetScan.
    time,P_src=tgt_pow(vis,tgtScan,antID,sciSPW,myMS=myMS,myMsmid=myMsmid,
                       subScans='on')

    PxxS=P_src[0]
    PyyS=P_src[1]

    # Read ACD calibrations - gains,Tinst, and calibration powers.

```

```

calDict=acd_calTabs(vis,atmScan,antID,sciSPW,myMS=myMS,myMsmd=myMsmd,
                    acdLoads=acdLoads)

Pcals=calDict['calPows']

PxxAmbS=Pcals['P_amb'][0]
PyyAmbS=Pcals['P_amb'][1]

PxxHotS=Pcals['P_hot'][0]
PyyHotS=Pcals['P_hot'][1]

# We have different handling for frequency-resolved and integrated SPWs.
nChans=PxxS.shape[0]

if (nChans > 1):

    #- Resolved AC SPWs.
    # Trim the spectral edges by posing the integration limits and average
    # over the remaining fChans channels.

    # Science-target powers.
    PxxM=np.mean(PxxS[fChans],axis=0)
    PyyM=np.mean(PyyS[fChans],axis=0)

    # Calibration (ACD) powers.
    PxxAmbM=np.mean(PxxAmbS[fChans],axis=0)
    PyyAmbM=np.mean(PyyAmbS[fChans],axis=0)

    PxxHotM=np.mean(PxxHotS[fChans],axis=0)
    PyyHotM=np.mean(PyyHotS[fChans],axis=0)

    #- Instrumental factors.
    # Store the normalized (to 1.0) spectral gains (arrays, constant) and
    # their scales (numbers, to be re-fined by the solver) separately.

    # Raw G* gains in a full SPW. We may flip the array (see below) so better
    # to do so before its limiting to fChans - wee need not to think whether
    # the fChans central range is placed symmetrically within its SPW.
    # The 1st index 0 means we pick the leading P_off subscan - see the note
    # in acd_calTabs2().
    GxxS=1.0/(calDict['iG*'][0,0])
    GyyS=1.0/(calDict['iG*'][0,1])

    # T_inst in full SPW. The same holds as for G*.
    TinstXX=calDict['T_inst'][0]
    TinstYY=calDict['T_inst'][1]

    # Check, whether we need flip them - this is the case of the LSB spws,
    # which have decreasing frequency channels.
    spwFreqs=calDict['freqs']

    if (spwFreqs[0] > spwFreqs[1]):
        # Reversed SPW, flip the gain and T_inst arrays.

        GxxS=np.flip(GxxS)
        GyyS=np.flip(GyyS)

```

```

    TinstXX=np.flip(TinstXX)
    TinstYY=np.flip(TinstYY)

    # Now it is safe to select the central fChans band-pass.
    GxxS=GxxS[fChans]
    GyyS=GyyS[fChans]

    TinstXX=TinstXX[fChans]
    TinstYY=TinstYY[fChans]

    # Calculate the unity-normalized gains by dividing out the mean value.
    GxxM=GxxS.mean()
    GyyM=GyyS.mean()

    gXX[sciSPW]=GxxS/ GxxM
    gYY[sciSPW]=GyyS/ GyyM

    # Instrumental power averaged over our fChans range.
    PIxxM=np.mean(GxxS*TinstXX)
    PIyyM=np.mean(GyyS*TinstYY)

    # Keep the time-axis of the last resolved SPW as a reference timebase,
    # to which we intepolate the SQLD signals.
    timeBase=time

else:
    #- Integrated ACs or SQLDs.
    # The single channel already represents a mean over the spw band-pass.

    # Single-channel science data.
    PxxM=PxxS[0]
    PyyM=Pyys[0]

    # Channel-integrated SPWs may have higher cadence. For SQLDs it is true
    # always, but it may be the case of integrated ACs, too. Let us
    # put all data to the same, slowest, timebase (as per resolved AC spws).

    # Compare the cadence (elementary time-step, a.k.a. integration).
    if ((timeBase[1]-timeBase[0]) > (time[1]-time[0])):

        # Indeed, we have higher cadence. Re-bin in time.
        # Use the timeBase defined as a time-axis for resolved SPW.

        PxxM=rebin2(PxxM,time,timeBase)
        PyyM=rebin2(PyyM,time,timeBase)
        time=timeBase

    #--> endif(higher cadence)

    # The attenuation factors  $AF=(P_{sky}-P_o)/(P_{off}-P_o)$  and offsets AO should
    # be, moreover, extracted from the SQLDss (ACs use mapping afMap
    # defined above).
    if sciSPW in [0,1,2,3,13,14,15,16]:

        # Extract the caliibration powers relevant for AF calculation

```

```

    Patm=Pcals['P_sky']
    Poff=Pcals['P_off']
    Pzero=Pcals['P_zero']

    # Calculate AFs.
    # The 1st index in powers is pol, the 2nd means (the single) channel.
    AFxx[sciSPW]=(Patm[0,0]-Pzero[0,0])/(Poff[0,0]-Pzero[0,0])
    AFyy[sciSPW]=(Patm[1,0]-Pzero[1,0])/(Poff[1,0]-Pzero[1,0])

    # Calculates AOs.
    AOxx[sciSPW]=Pzero[0,0]*(AFxx[sciSPW]-1.0)
    AOyy[sciSPW]=Pzero[1,0]*(AFyy[sciSPW]-1.0)
    #--> endif(SQLD spws)

    # Calibration data.
    PxxAmbM=PxxAmbS[0]
    PyyAmbM=PyxAmbS[0]

    PxxHotM=PxxHotS[0]
    PyyHotM=PyxHotS[0]

    #- Instrumental factors.
    GxxM=1.0/(calDict['iG*'][0,0,0])
    GyyM=1.0/(calDict['iG*'][0,1,0])

    # Create a fake spectral normalized gain for integrated SPWs.
    # We use it in products with the ATM arrays (e.g. transmissivity), which
    # are always spectral.
    gXX[sciSPW]=np.ones(len(fChans))
    gYY[sciSPW]=np.ones(len(fChans))

    # Instrumental power: The channel 0 is the average already
    PIxxM=GxxM*calDict['T_inst'][0,0]
    PIyyM=GyyM*calDict['T_inst'][1,0]
    #--> endif(resolved SPW)

    # Append the frequency-averaged time series for each SPW to the maps.
    Pxx[sciSPW]=PxxM
    Pyy[sciSPW]=PyxM

    # The calibration powers are already time-averaged.
    PxxAmb[sciSPW]=PxxAmbM
    PyyAmb[sciSPW]=PyxAmbM

    PxxHot[sciSPW]=PxxHotM
    PyyHot[sciSPW]=PyxHotM

    # Set the ACD-based gains (scales) and instrumental contributions.
    Gxx[sciSPW]=GxxM
    Gyy[sciSPW]=GyyM

    PIxx[sciSPW]=PIxxM
    PIyy[sciSPW]=PIyyM

```

```

print('SPW ',sciSPW,': <Tsky+src_XX> = ',
      (Pxx[sciSPW].mean()-PIxx[sciSPW])/Gxx[sciSPW],'+/-',
      Pxx[sciSPW].std()/Gxx[sciSPW])
print('          <Tsky+src_YY> = ',
      (Pyy[sciSPW].mean()-PIyy[sciSPW])/Gyy[sciSPW],'+/-',
      Pyy[sciSPW].std()/Gyy[sciSPW])

#-->endfor(all sciSPWs in the list)

# Save the original gains and instrumental powers for comparison with
# the updated values after the solution.
# Note: We have to use copy.deepcopy(), the simple assignment operator
# copies just the reference!

GxxOrig=deepcopy(Gxx)
GyyOrig=deepcopy(Gyy)

PIxxOrig=deepcopy(PIxx)
PIyyOrig=deepcopy(PIyy)

#-- Dimensions of the system.

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# How many net equations do we have per block of 1 time-sample.
# Be carefull: Do not flag the data that are *not* present in the spwList!
eqBlockSize=2*numSpw-len(flaggedData)

# The data has been read-out, we may close the (local) tools.
if (ownMsm): myMsm.done()
if (ownMS): myMS.close()

#-- Arrays for calculations of the matrix and the RHS-vector components
#   of the linearized system that depend just on the models (ATM + quiet Sun).

# Number of the model spectral windows.
nAtmSPWs=myAtm.getNumSpectralWindows()
# Number of all channels in the model SPWs.
nChans=myAtm.getNumChan(0)
# Number of central band-pass (with edges trimmed) channels.
cChans=len(fChans)

# Frequencies of channels in our model SPWs (central band-pass only).
spw_freqs=np.zeros((nAtmSPWs,cChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6   # MHz to Hz

    spw_freqs[atmSpw]=(fMin+df*np.arange(nChans))[fChans]

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.

```

```

TbsW0=np.zeros((nAtmSPWs,cChans))
dTdW=np.zeros((nAtmSPWs,cChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,cChans))
tau_1=np.zeros((nAtmSPWs,cChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(spw_freqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
#   Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01    #mm
dBMean_max=0.001    # kiloKelvin

# Initial values for the dW and dL means - anything bigger then the stoppers.
dW_mean=0.5
dB_mean=0.5

# Iteration counter.
itCount=0

# Set the starting values for iterations.
w0=pwv0
Tsun0xx=Tsun00
Tsun0yy=Tsun00

#-- Iterations: Minimize the mean value of variations.

while ((abs(dW_mean) > dwMean_max) or (abs(dB_mean) > dBMean_max))
    and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] 'natural' variables.
    J_wb=np.zeros((eqBlockSize,3)) # 3 vars repeating in all spws
    # G-block matrix: Jacobian part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each SPW:Pol.
    # A-block matrix: Jacobian part for [dG,dP_inst] for ACD calibration powers.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, ambient.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, hot.

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))
    # RHS-vector 'appendix' - amb. & hot ACD calibration powers (time-averaged).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

```

```
# Residual mean sky temperature - after subtrated brightness of the
# atmosphere (+CMB) and (attenuated) source. We keep it for all sci SPWs and
# both polarizations and print it out at the end of every iteration cycle
# as a diagnostics of convergence and solution quality.
```

```
resTxx={}
resTyy={}

```

```
# Mapping between spw IDs and the row number in the system of equations.
# We need it for final diagnostic plots. It may be non-trivial as some
# spw:pol pairs may be flagged and ignored, casuing irregularity in the map.
```

```
spw2rowXX={}
spw2rowYY={}

```

```
#- Re-calculate ATM model with updated <PWV> and <Tsun>.
```

```
# Set current estimation of PWV to the ATM model.
```

```
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))
```

```
#- Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
# Calculate factors dependent only on the mmodel, not the data.
```

```
for atmSPW in range(nAtmSPWs):
```

```
#- Model value of the sky/atm brightness with current pwv.
```

```
Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
```

```
# Limit the spectrum to the central fChans reagon as we have trimmed
```

```
# the edge channels in ACD-based spectral normalized gain gXX/gYY.
```

```
# Scale to kiloKelvins.
```

```
TbsW0[atmSPW]=0.001*Tebb[fChans]
```

```
#- Slope (centered difference) of the atm brightness with current pwv.
```

```
TebbH=myAtm.getTebbSkySpec(atmSPW,
```

```
qa.quantity(w0+dws, 'mm'))[1]['value'][fChans]
```

```
TebbL=myAtm.getTebbSkySpec(atmSPW,
```

```
qa.quantity(w0-dws, 'mm'))[1]['value'][fChans]
```

```
# In kiloKelvins/mm
```

```
dTdW[atmSPW]=0.001*(TebbH-TebbL)/(2.0*dws)
```

```
#- Opacities & transmisivity.
```

```
# Dry zenith opacity
```

```
tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
```

```
# Water-vapour related 'wet' opacity
```

```
tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
```

```
# Total opacity (spectral).
```

```
opacity=airMass*(tau_wet+tau_dry)
```

```
# Transmissivity (spectral). Limited to fChans.
```

```
trans[atmSPW]=np.exp(-opacity[fChans])
```

```
# Wet opacity per 1mm PWV scaled by airmass. Limited to fChans.
```

```
tau_1[atmSPW]=(airMass/w0)*tau_wet[fChans]
```

```
#--> endfor(modelSPWs)
```

```
#- Loop over all SCI spws - combine model-only dependent factors with
# sciSPW-dependent data and instrumental factors and build the system.
```

```

# Count the actual number of equations - some rows may be flagged.
eqCount=0

for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Separate procedure by polarizations for combination with real data.

    #- Pol XX

    # Full identificator of data: sciSPW:pol
    sciDataID=str(sciSPW)+' :XX'

    if (sciDataID not in flaggedData):

        #- Model + instrumental factors combined.

        # Sensitivity factor for PWV.
        SWxx=(- eta_l* eta_F* Tsun0xx* np.mean(gXX[sciSPW]* trans_M* tau_1M* SF_M)
              +(1.0- eta_F+ eta_F* eta_l)* np.mean(gXX[sciSPW]* dTdW_M) ) * Gxx[sciSPW]

        # Sensitivity factor for the (apparent) solar-brighthness temperature.
        SBxx=Gxx[sciSPW]* eta_l* eta_F* np.mean(gXX[sciSPW]* trans_M* SF_M)

        # Model T_sky as viewed by the receiver with spectral gain gXX.
        TskyXX=(eta_l* eta_F* Tsun0xx* np.mean(trans_M* SF_M* gXX[sciSPW])
                +(1.0- eta_F+ eta_F* eta_l)* np.mean(TbsW0_M* gXX[sciSPW])
                +eta_F* (1.0- eta_l)* Tamb)

        #- Build the matrix: Natural (dW & dB sensitivities) block.

        # Set the sensitivity factors for the current SPW and pol as a row
        # in the block/tile for the linearized-system Jacobian matrix.
        J_wb[eqCount]=np.array([SWxx, SBxx, 0.0])

        #- Build the matrix: Instrumental (G & P_inst sensitivities) block.

        # We have to prepend a proper offset for the G and P_inst vars as
        # they are independendt for each SPW and polarization.
        bOffset=2* eqCount

        J_gp[eqCount, bOffset]=1.0

```

```

J_gp[eqCount,bOffset+1]=TskyXX

#- Build the matrix: ACD powers (amb & hot) 2-block.

# Ambient-power sub-block. Scale between the  $G^* / P_{inst}$  used in
#  $J_{wb}$  and  $J_{gp}$ , and  $G / P_{inst}$  used in the ACD cal equations by the
# attenuation factor  $AF = G^* / G = (P_{sky} - P_o) / (P_{off} - P_o)$ 
af=AFxx[afMap[sciSPW]]

J_cdA[eqCount,bOffset]=af
J_cdA[eqCount,bOffset+1]=af*Tamb

# Hot-power sub-block.
J_cdH[eqCount,bOffset]=af
J_cdH[eqCount,bOffset+1]=af*Thot

#- Build the RHS vector: Target data

# RHS:  $P_{xx}(t)$  with their stationary (in the working point) instrumental
# and ATM-based parts subtracted.
dPxx=Pxx[sciSPW]-PIxx[sciSPW]-Gxx[sciSPW]*TskyXX

# Set it as the RHS component.
CC[eqCount]=dPxx

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOxx[afMap[sciSPW]]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=PxxAmb[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Tamb+aoff
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=PxxHot[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Thot+aoff

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resTxx[sciSPW]=dPxx.mean()/Gxx[sciSPW]

# Mapping between SPW:XX and equation number.
spw2rowXX[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#- Pol YY

# Full identificador of data: sciSPW:pol
sciDataID=str(sciSPW)+' :YY'

if (sciDataID not in flaggedData):

    #- Model + instrumental factors combined.

    # Sensitivity factor for PWV.

```

```

SWyy=(- eta_l* eta_F* Tsun0yy* np.mean(gYY[sciSPW]* trans_M* tau_1M* SF_M)
      +(1.0- eta_F+ eta_F* eta_l)* np.mean(gYY[sciSPW]* dTdW_M))* Gyy[sciSPW]

# Sensitivity factor for the (apparent) solar-brighthness temperature.
SByy=Gyy[sciSPW]* eta_l* eta_F* np.mean(gYY[sciSPW]* trans_M* SF_M)

# Model T_sky as viewed by the receiver with spectral gain gYY.
TskyYY=(eta_l* eta_F* Tsun0yy* np.mean(trans_M* SF_M* gYY[sciSPW])
        +(1.0- eta_F+ eta_F* eta_l)* np.mean(TbsW0_M* gYY[sciSPW])
        +eta_F* (1.0- eta_l)* Tamb)

#- Build the matrix: Natural (dW & dB sensitivities) block.

# Set the sensitivity factors for the current SPW and pol as a row
# in the block/tile for the linearized-system Jacobian matrix.
J_wb[eqCount]=np.array([SWyy,0.0,SByy])

#- Build the matrix: Instrumental (G & P_inst sensitivities) block.

# We have to prepend a proper offset for the G and P_inst vars as
# they are independendt for each SPW and polarization.
bOffset=2* eqCount

J_gp[eqCount,bOffset]=1.0
J_gp[eqCount,bOffset+1]=TskyYY

#- Build the matrix: ACD powers 2-block.

# Attenuation factor G*/G.
af=AFyy[afMap[sciSPW]]

# Ambient-power sub-block.
J_cdA[eqCount,bOffset]=af
J_cdA[eqCount,bOffset+1]=af* Tamb

# Hot-power sub-block.
J_cdH[eqCount,bOffset]=af
J_cdH[eqCount,bOffset+1]=af* Thot

#- Build the RHS vector: Target data.

# RHS: Pyy(t) with their stationary (in the working point) instrumental
# and ATM-based parts subtracted.
dPyy=Pyy[sciSPW]- PIyy[sciSPW]- Gyy[sciSPW]* TskyYY

# Set it as the RHS component.
CC[eqCount]=dPyy

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOyy[afMap[sciSPW]]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=PyyAmb[sciSPW]- af* PIyy[sciSPW]- af* Gyy[sciSPW]* Tamb+aoff
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=PyyHot[sciSPW]- af* PIyy[sciSPW]- af* Gyy[sciSPW]* Thot+aoff

```

```

#- Diagnostics

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTyy[sciSPW]=dPyy.mean()/Gyy[sciSPW]

# Mapping between SPW:YY and equation number.
spw2rowYY[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#--> endfor(all sci SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPWs/polarization channels.")
    return False

#-- Grand system assembly and inversion by parts. We use the approach from
# extMatInv() for inversion of extended-system matrix. The procedure
# performed here looks mysterious (sometimes a factor nSamples emerges or
# the RHS vector summed over time-samples), but all directly implies from
# application of the inverted matrix of extended grand-system (preserved
# in the commented section below) to the RHS vector. We use advantage of
# the symmetry of the original grand-system matrix ( $JJ_K^T = JJ_K$ ).
# Advantage: This block-decomposed solution is  $O(ns^1)$  in the number of
# time-samples, while the 'brutal-force' direct method in the commented
# section below scales as  $O(ns^3)$  and takes ages...

#- Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
# Attention!!! NumPy's transpose() makes just another view of a matrix,
# we have to make a copy, otherwise subsequent scaling would scale the
# original J_cdA matrix, too!
J_cdA_T=np.transpose(J_cdA).copy()
J_cdH_T=np.transpose(J_cdH).copy()

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

```

```

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T, J_cdH_T]) from already scaled-up transposed sub-blocks.
J_cd=np.vstack([J_cdA, J_cdH])
J_cd_T=calWeight*np.transpose(J_cd)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T, J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T, J_gp)+np.matmul(J_cd_T, J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(J_K_inv, Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc- nSamples*np.matmul(Lb, AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv, LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=-np.matmul(AinvRb, Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e., multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T, CC)      # 3 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T, CCsum) # 3 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T, CCsum)   # 2*eqCount x 1

```

```

+np.matmul(J_cdA_T, CCA)
+np.matmul(J_cdH_T, CCH)

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (3 x 1)
nvConst=np.matmul(Minv11_OD, C_K_wb_sum)+np.matmul(Minv12, C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv, C_K_wb) # 3 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21, C_K_wb_sum)+np.matmul(Minv22, C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
dW = varNat[0]
dBxx=varNat[1]
dByy=varNat[2]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per SPW:POL.
dPG=np.transpose(np.reshape(varInst, (eqBlockSize, 2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errors for the over-determined set of eqs.

# err=A.x-b: Sci-data part.
errVec=np.matmul(J_wb, varNat)+np.matmul(J_gp, varInst)-CC
# Xi square for the sci-data part.
xiSci=np.sum(errVec*errVec)

# err=A.x-b: ACD-data part.
errVecACDamb=np.matmul(J_cdA, varInst)-CCA
errVecACDhot=np.matmul(J_cdH, varInst)-CCH
# Xi square for the acd-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)

# Grand-total Xi-square normalized to the number of equations (sci/acd).
xiSquare=xiSci/ nSamples+xiACD/ calWeight

...

#--> This is the brutal-force straightforward method for grand-system
# assembly and solution. It works, but takes ages. It is kept here for
# facilitation of understanding of the above efficient but bit mysterious
# procedure, which solves the grand-system matrix inversion block-wise,
# using the method explained in extMatInv().

#-- Loop over time samples: Grand-system assembly.

```

```

# Grand system matrix (placeholder/restart).
JJ=np.zeros((0,2*eqCount+3*nSamples))

# The RHS vector: We have assembled it already, just needs to be re-shaped
# from the form of block-matrix eqBlockSize x nSamples to a 1-column matrix.
# We need to change the order of elements: First by SPW:POL channel, only
# after that by time (so far it is reversed) - that is the reason for
# transposing first.

CC=np.reshape(np.transpose(CC),(-1,1))

# Append the ambient- and hot-load powers. Scale them up by sqrt(calWeight).

scUp=sqrt(calWeight)
CC=np.append(CC,[scUp*CCA,scUp*CCH])

for tt in range(nSamples):

    # Compose a block-row of the grand matrix.
    JJ_row=np.hstack([np.zeros((eqBlockSize,3*tt)),J_wb,
                      np.zeros((eqBlockSize,3*(nSamples-tt-1))),J_gp])

    # Append it to the grand matrix.
    JJ=np.vstack([JJ,JJ_row])

#--> endfor(t-samples)

#- Append the ACD-related rows.

# Scale-up the calibration matrices by the factor sqrt(calWeight).
J_cdA*=scUp
J_cdH*=scUp

# Compose the full ACD block.
J_cd=np.vstack([J_cdA,J_cdH])

# Form the ACD-equations row.
JJ_row=np.hstack([np.zeros((2*eqBlockSize,3*nSamples)),J_cd])
JJ=np.vstack([JJ,JJ_row])

#-- Grand system solution - brutal

#- Make a set of normal equations (quadratic-form minimizer).

JJ_T=np.transpose(JJ)

J_KK=np.matmul(JJ_T,JJ)

JJ_inv=np.linalg.inv(J_KK)

C_KK=np.matmul(JJ_T,CC)

#- Solve the quadratic-form minimizer.
varVec=np.linalg.solve(J_KK,C_KK)

```

```

#- Decompose the solution to the instrumental and natural parts and re-form
# them into proper shapes.

varNatur=varVec[0:3*nSamples]
varInst=varVec[3*nSamples:]

# P_inst and G for each non-flagged SPW:Pol channel.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

# dW, dBxx, and dByy variations for each time sample.
# Do transpose - make the variations time-series to be matrix rows.
varNatur=np.transpose(np.reshape(varNatur,(nSamples,3)))

# Now we can easily decompose the time series.
dW = varNatur[0]
dBxx=varNatur[1]
dByy=varNatur[2]

#- Analytics: Calculate the errors for overdetermined set of eqs.

# err=A.x-b
errVec=np.matmul(JJ,varVec)-CC

# Limit the errors to 'natural' variables.
errVec=errVec[0:nSamples*eqBlockSize]
# Reshape the long column vector to the array of eqBlockSize vectors,
# each representing the nSamples time series.
errVec=np.transpose(np.reshape(errVec,(nSamples,eqBlockSize)))

#--> End of the brutal-force method
'''

#- All variations should tend to zero means.

dW_mean=dW.mean()
dBxx_mean=dBxx.mean()
dByy_mean=dByy.mean()

dB_mean=np.sqrt(dBxx_mean**2+dByy_mean**2)

# Update stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0xx+=dBxx_mean
Tsun0yy+=dByy_mean

# Update also the P_inst and G by the found differences.
# Start with XX pol.
for sciSPW,eqCount in spw2rowXX.items():
    # Update the P_inst and G.
    P1xx[sciSPW]+=dPinst[eqCount]
    Gxx[sciSPW]+=dGinst[eqCount]
#--> endfor(SPWs with valid XX pol)

# YY pol.

```

```

for sciSPW,eqCount in spw2rowYY.items():
    # Update the P_inst and G.
    PIyy[sciSPW]+=dPinst[eqCount]
    Gyy[sciSPW]+=dGinst[eqCount]
#--> endfor(SPWs with valid YY pol)

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and Tsun after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('\nMean PWV for the scan,',tgtScan,' w0 = ',w0,
      ", mean Tsun (XX/YY) = ",Tsun0xx,'/',Tsun0yy)

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale back from kiloKelvins to Kelvins.
resTxxArr=1000.0*np.array(list(resTxx.values()))
resTyyArr=1000.0*np.array(list(resTyy.values()))

print("\nres Txx: ", resTxxArr,"\nmean +/- RMS: ",
      resTxxArr.mean(),"+/-",resTxxArr.std())
print("\nres Tyy: ", resTyyArr,"\nmean +/- RMS: ",
      resTyyArr.mean(),"+/-",resTyyArr.std())

print("\nInstrumental characteristics shift [dP_inst, dG]:",dPinst,dGinst)

print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / end of main iteration loop.

#- Diagnostic plots.
# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

#-Set the plotting device.

plt.rcParams["figure.figsize"] = (30,20)
# Make XX and YY plots as subplots (horizontally)
ff,sps=plt.subplots(1,2)

# X-axis label just for the bottom sub-plot.
sps[0].set_xlabel('Time [MJDsec]')
sps[1].set_xlabel('Time [MJDsec]')

# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Color-map for SPWs, incl. line-style.
# We have minimum 2 and maximum 12 SPWs in the spwList argument.
# If we use all 12 spws, linestyle correspondence to the spw type follows:
csline={5: '-r',7: '-g',9: '-b',11: '-k',          # AC resolved
        6: '--r',8: '--g',10: '--b',12: '--k',     # AC integrated

```

```

0:':r',1:':g',2:':b',3:':k'} # SQLD

#- Make the *analytical* plots - least-square residuals etc.

# XX pol

for spw,eqRow in spw2rowXX.items():
    # Panel row 1: Residuals/errors scaled to temperature in Kelvins.
    sps[0].plot(time,1000.0*errVec[eqRow]/ Gxx[spw],csline[spw])

# YY pol

for spw,eqRow in spw2rowYY.items():
    # Panel row 1: Residuals/errors scaled to temperature in Kelvins.
    sps[1].plot(time,1000.0*errVec[eqRow]/ Gyy[spw],csline[spw])

#- Save the plots to a file and close.
plt.savefig(plotDir+'/errs_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Make the full-solution plots: PWV(t) and T_b(t).

# Add variations to their time-averages. Temperatures are scaled back
# from kiloKelvins to Kelvins.
pwv=w0+dW
TsunXX=1000.0* (Tsun0xx+dBxx)
TsunYY=1000.0* (Tsun0yy+dByy)

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff,(sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Left panel: Apparent target brightness per polarization.
sp1.plot(time,TsunXX,'-r')
sp1.plot(time,TsunYY,'-b')

# Right panel: PWV.
sp2.plot(time,pwv,'-g')

plt.savefig(plotDir+'/dWdB/dWdB_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Show, how much the instrumental variables G and P_inst have changed.

# The instrumental characteristics are maps, we cannot process them as arrays.

GxxChange={}
GyyChange={}

PIxxChange={}
PIyyChange={}

for spw in spw2rowXX.keys():

```

```

GxxChange[spw]=(Gxx[spw]-GxxOrig[spw])/GxxOrig[spw]
PIxxChange[spw]=(PIxx[spw]-PIxxOrig[spw])/PIxxOrig[spw]

for spw in spw2rowYY.keys():

    GyyChange[spw]=(Gyy[spw]-GyyOrig[spw])/GyyOrig[spw]
    PIyyChange[spw]=(PIyy[spw]-PIyyOrig[spw])/PIyyOrig[spw]

print("----")

print("G_xx relative changes:", GxxChange)
print("G_yy relative change:", GyyChange)

print("----")

print("P_inst XX relative change:", PIxxChange)
print("P_inst YY relative change:", PIyyChange)

print("----")

# Print total quadratic error for the variables.
print("\n===== Xi-squares =====      SCI: ", xiSci,
      ", ACD:", xiACD,
      ", Total:", xiSquare, "\n")

#-- Export the results.

# 'Compactify' the per-polarization T_B arrays.
Tsun=np.array([TsunXX, TsunYY])

return {'time':time, 'pwv':pwv, 'T_B':Tsun, 'J':J_wb, 'C':CC}

#-----

def dWdBdG2(vis, atmScan, tgtScan, myAtm, antID, spwList, T_tgt=-1.0e6,
            flaggedData=[], myMS=None, myMsmd=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 $\langle T_B(t)^{XX/YY} \rangle = T_{tgt}$ 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) $\langle T_B(t) \rangle$ 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.

"""

#--- Static parameters

#- Dirs

Output/diagnostics plots.

plotDir='./sciPWVs_gt'

if (not os.path.exists(plotDir)): os.makedirs(plotDir)

if (not os.path.exists(plotDir+'dWdB')): os.makedirs(plotDir+'dWdB')

#- Tools, global data

Flags used to check, whether we should close the tool later.

ownMS=False

ownMsmid=False

Open own tools if 'global' not provided

if (not myMS):

Open the ms tool

myMS=msstool()

myMS.open(vis)

ownMS=True

if (not myMsmid):

Open the msmd tool.

myMsmid=msmidtool()

myMsmid.open(vis)

```

ownMsmd=True

# Create our dictionary with ACD load temperatures (ambient, hot) if needed.
if (not acdLoads):
    acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                    warnIfNoLoadTemperatures = True,
                                    mymsmd=myMsmd)

#- Antenna

antNames=myMsmd.antennanames()
myAntName=antNames[antID]
antType=myAntName[:2]

# Map the antenna efficiencies by antenna type. As per investigation from
# merged T_sun temperatures form 2023 and 2026 data (project_22_25_Tb.ods).
eta_l_map={'CM':0.941, 'DA':0.947, 'DV':0.956}
eta_F_map={'CM':0.891, 'DA':0.897, 'DV':0.906}

# Rear-spillover + ohmic antenna efficiency - see project_22_25_Tb.ods
eta_l=eta_l_map[antType]
# Forward-spillover antenna efficiency - see project_22_25_Tb.ods
eta_F=eta_F_map[antType]

# We have two exception that seem to have systematically lower
# antenna efficiencies: DA64 and DV08 - see project_22_25_Tb.ods
if myAntName=='DA64':
    eta_l=0.929
    eta_F=0.880
if myAntName=='DV08':
    eta_l=0.943
    eta_F=0.894

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# Optimized weight of the ACD calibration equations in the system.
# Search mark in dWdBdG2(): CW_1
calWeight=1.0e5

#- Target: The Sun by default but we can handle the non-solar scans, too.

# 1. What type of target do we have: The Sun or calibrator (i.e., the skies)?

# All Sun scans.
solScans=myMsmd.scansforintent('OBSERVE_TARGET#ON_SOURCE')
solarTgt=(tgtScan in solScans)

# 2. The solar target (the default).

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.300 # kiloKelvins!!!

```

```

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

"""
# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff (arcsec, incl. cos(delta)): 822  155.8
'''
x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.
"""

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightenning correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temoerature corrected to limb brightenning - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu)

# Weight of the array-mean equation <T_B(t)>=T_tgt_arrayMean in the system.
# If T_tgt is not explicitly set (the default value is -1MK), do not apply
# this equation at all. This can be achieved by setting its weight to 0.
if (T_tgt < -10000.0):
    # Do not apply T_tgt bond. If mWeight is zero, T_tgt is irrelevant.
    T_tgt=0.0
    mWeight=0.0
else:
    # We use kiloKelvins in calculations, re-scale T_tgt to kK, too.

```

```

T_tgt*=0.001
# Set the experimentally found optimized mWeight.
# Search mark in dWdBdG2(): MW_1
mWeight=5000.0

# 3. Specific handling of non-solar scans - override the solar defaults.
if (not solarTgt):

    print("dWdBdG2(): Non-solar target. Setting the target T_B to zero.")

    T_tgt=0.0
    mWeight=5000.0

    # No solar radiation is falling into antenna.
    Tsun00=0.0

#-->endif(non-solar target)

#- SPWs

# LO frequency. We take it as a reference frequency for the solar spectrum.
f_LO=1.98e11 # in Hz

# Integration limits for resolved AC SPWs.
startChan=8
stopChan=120

fChans=range(startChan,stopChan)

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,' @airMass = ',airMass)
print('Estimated T_B = ',1000.0*Tsun00)
print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#--- Read-out the sky+src powers for all SPWs in the spwList.

# Store the frequency-averaged sky+src powers in a map.
# spwId -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW
# id, which is sparse in the set of integers.
Pxx={}
Pyy={}

```

```
#- Similar maps for the gains and instrumental powers  $P_{inst}=G*T_{inst}$ .
# Here, all gains and  $P_{inst}$  are, in fact, the  $G^*$  and  $P_{ints}^*$  values for the
# higher-attenuation solar regime. In order to append the ACD calibration
# equations, whose measurements are performed in the norma-attenuation regime
# in the preceding ATM scan, we determine also the attenuation factors
#  $AF=G^*/G$  and related power offsets  $AO$ .
```

```
# Frequency-averaged gains. The scale for the spectral-resolved norm. gains.
Gxx={}
Gyy={}

```

```
# Attenuation factors  $G/G^*=(P_{sky}-P_0)/(P_{off}-P_0)$ . Attributed to every-spw
# Gxx & Gyy, but actually calculated just in SQLD spws, the rest is mapped.
```

```
AFxx={}
AFyy={}

```

```
afMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}
```

```
# Attenuation-based power offsets  $AO=-P_0*(P_{sky}-P_{off})/(P_{off}-P_0)$ .
# They are non-zero only in SQLD spws.
```

```
AOxx={}
AOyy={}

```

```
# Spectral gain normalized to 1.0. Actual ACD-based values in spectral AC spws,
# a trivial 1.0 ndarray for integrated-AC and SQLD spws.
```

```
gXX={}
gYY={}

```

```
# Frequency-averaged instrumental powers.
```

```
PIxx={}
PIyy={}

```

```
#- Finally, the raw ACD-calibration powers.
```

```
# Ambient-load powers.
```

```
PxxAmb={}
PyyAmb={}

```

```
# Hot-load powers.
```

```
PxxHot={}
PyyHot={}

```

```
#-- Read the data and ACD calibration tables in a loop over all sciSPWs.
```

```
for sciSPW in spwList:
```

```
    #- Raw data & ACD tables
```

```
    # Read raw science powers for the target defined by targetScan.
```

```
    time,P_src=tgt_pow(vis,tgtScan,antID,sciSPW,myMS=myMS,myMsmid=myMsmid,
                        subScans='on')
```

```
    PxxS=P_src[0]
    PyyS=P_src[1]

```

```
    # Read ACD calibrations - gains,  $T_{inst}$ , and calibration powers.
```

```
    calDict=acd_calTabs(vis,atmScan,antID,sciSPW,tgtScan=tgtScan,
```

```

myMS=myMS, myMsmid=myMsmid, acdLoads=acdLoads)

Pcals=calDict['calPows']

PxxAmbS=Pcals['P_amb'][0]
PyyAmbS=Pcals['P_amb'][1]

PxxHotS=Pcals['P_hot'][0]
PyyHotS=Pcals['P_hot'][1]

# We have different handling for frequency-resolved and integrated SPWs.
# Find the number of channels in the observed SPW.
nChansO=PxxS.shape[0]

if (nChansO > 1):

    #- Resolved AC SPWs.
    # Trim the spectral edges by posing the integration limits and average
    # over the remaining fChans channels.

    # Science-target powers.
    PxxM=np.mean(PxxS[fChans], axis=0)
    PyyM=np.mean(PyyS[fChans], axis=0)

    # Calibration (ACD) powers.
    PxxAmbM=np.mean(PxxAmbS[fChans], axis=0)
    PyyAmbM=np.mean(PyyAmbS[fChans], axis=0)

    PxxHotM=np.mean(PxxHotS[fChans], axis=0)
    PyyHotM=np.mean(PyyHotS[fChans], axis=0)

    #- Instrumental factors.
    # Store the normalized (to 1.0) spectral gains (arrays, constant) and
    # their scales (numbers, to be re-fined by the solver) separately.

    # Raw G/G* gains in a full SPW. We may flip the array (below) so better
    # to do so before its limiting to fChans - wee need not to think whether
    # the fChans central range is placed symmetrically within its SPW.

    if (solarTgt):
        # Solar gain G* has shape 2 x nPols x nChans. The 1st index 0 means we
        # pick the *leading* P_off subscan - see the note in tgt_Tb().
        GxxS=1.0/ (calDict['iG*'][0,0])
        GyyS=1.0/ (calDict['iG*'][0,1])
    else:
        # Non-solar standard gain G has shape nPols x nChans.
        GxxS=1.0/ (calDict['iG'][0])
        GyyS=1.0/ (calDict['iG'][1])

    # T_inst in full SPW. The same holds for flipped/reversed SPWs as for G*.
    TinstXX=calDict['T_inst'][0]
    TinstYY=calDict['T_inst'][1]

    # Check, whether we need flip them - this is the case of the LSB spws,
    # which have decreasing/reversed frequency channels.
    spwFreqs=calDict['freqs']

```

```

if (spwFreqs[0] > spwFreqs[1]):
    # Reversed SPW, flip the gain and T_inst arrays.

    GxxS=np.flip(GxxS)
    GyyS=np.flip(GyyS)

    TinstXX=np.flip(TinstXX)
    TinstYY=np.flip(TinstYY)

    # Now it is safe to select the central fChans band-pass.
    GxxS=GxxS[fChans]
    GyyS=GyyS[fChans]

    TinstXX=TinstXX[fChans]
    TinstYY=TinstYY[fChans]

    # Calculate the unity-normalized gains by dividing out the mean value.
    GxxM=GxxS.mean()
    GyyM=GyyS.mean()

    gXX[sciSPW]=GxxS/ GxxM
    gYY[sciSPW]=GyyS/ GyyM

    # Instrumental power averaged over our fChans range.
    PIxxM=np.mean(GxxS*TinstXX)
    PIyyM=np.mean(GyyS*TinstYY)

    # Keep the time-axis of the last resolved SPW as a reference timebase,
    # to which we intepolate the SQLD signals.
    timeBase=time

    # SQLD spws have an instrumental offset ralated to Pzero. It is calculated
    # below, in the single-channel SPWs branch. Here we set it to zero
    # because Pzero for AC-resolved spws is zero.
    AOxx[sciSPW]=0.0
    AOyy[sciSPW]=0.0

else:
    #- Integrated ACs or SQLDs.
    # The single channel already represents a mean over the spw band-pass.

    # Single-channel science data.
    PxxM=PxxS[0]
    PyyM=Pyys[0]

    # Channel-integrated SPWs may have higher cadence. For SQLDs it is true
    # always, but it may be the case of integrated ACs, too. Let us
    # put all data to the same, slowest, timebase (as per resolved AC spws).

    # Compare the cadence (elementary time-step, a.k.a. integration).
    if ((timeBase[1]-timeBase[0]) > (time[1]-time[0])):

        # Indeed, we have higher cadence. Re-bin in time.
        # Use the timeBase defined as a time-axis for resolved SPW.

```

```

print('---> SPW',sciSPW,
      'has higher cadence: Rebinning to time-line of AC-resolved SPW.')

PxxM=rebin2(PxxM,time,timeBase)
PyyM=rebin2(PyyM,time,timeBase)
time=timeBase

#--> endif(higher cadence)

# The attenuation factors  $AF=(P_{sky}-P_o)/(P_{off}-P_o)$  and offsets  $AO$  should
# be, moreover, extracted from the SQLDss (ACs use mapping afMap
# defined above) for the solar target. In non-solar they are trivial.
if sciSPW in [0,1,2,3,13,14,15,16]:

    if (solarTgt):
        # Extract the caliibration powers relevant for AF calculation
        Patm=Pcals['P_sky']
        # Poff is the only calPow that has a structure 2 x nPols x nChans,
        # similarly like gain. The others are just nPols x nChans 2D arrays.
        # Here, the first dimension '2' is for leading/trailing Poff subscans
        # and we pick the leading one (see the gain extraction above).
        Poff=Pcals['P_off'][0]
        Pzero=Pcals['P_zero']

        # Calculate AFs.
        # The 1st index in powers is pol, the 2nd means (the single) channel.
        AFxx[sciSPW]=(Patm[0,0]-Pzero[0,0])/(Poff[0,0]-Pzero[0,0])
        AFyy[sciSPW]=(Patm[1,0]-Pzero[1,0])/(Poff[1,0]-Pzero[1,0])

        # Calculates AOs.
        AOxx[sciSPW]=Pzero[0,0]*(AFxx[sciSPW]-1.0)
        AOyy[sciSPW]=Pzero[1,0]*(AFyy[sciSPW]-1.0)
    else:
        # Non-solar: Trivial.
        AFxx[sciSPW]=1.0
        AFyy[sciSPW]=1.0

        AOxx[sciSPW]=0.0
        AOyy[sciSPW]=0.0

    #--> endif(solar target)

else:
    # For non-SQLD spws the offsets are zero because Pzero is zero.
    # The AFs are mapped from the corresponding SQLD spw.
    AOxx[sciSPW]=0.0
    AOyy[sciSPW]=0.0

#--> endif(SQLD spws)

# Calibration data.
PxxAmbM=PxxAmbS[0]
PyyAmbM=PyyAmbS[0]

PxxHotM=PxxHotS[0]
PyyHotM=PyyHotS[0]

```

```

#- Instrumental factors.

# Gains: Solar or non-solar scan? Anyway, the last index '0' picks the
# single channel of the integrated SPWs as its own average.
if (solarTgt):
    # Solar gain G* has shape 2 x nPols x nChans. The 1st index 0 means we
    # pick the *leading* P_off subscan - see the note in tgt_Tb().
    GxxM=1.0/ (calDict['iG*'][0,0,0])
    GyyM=1.0/ (calDict['iG*'][0,1,0])
else:
    # Non-solar standard gain G has shape nPols x nChans.
    GxxM=1.0/ (calDict['iG'][0,0])
    GyyM=1.0/ (calDict['iG'][1,0])

# Create a fake spectral normalized gain for integrated SPWs.
# We use it in products with the ATM arrays (e.g. transmissivity), which
# are always spectral.
gXX[sciSPW]=np.ones(len(fChans))
gYY[sciSPW]=np.ones(len(fChans))

# Instrumental power: The channel 0 is the average already
PIxxM=GxxM*calDict['T_inst'][0,0]
PIyyM=GyyM*calDict['T_inst'][1,0]
#--> endif(resolved SPW)

# Append the frequency-averaged time series for each SPW to the maps.
Pxx[sciSPW]=PxxM
Pyy[sciSPW]=PyyM

# The calibration powers are already time-averaged.
PxxAmb[sciSPW]=PxxAmbM
PyyAmb[sciSPW]=PyyAmbM

PxxHot[sciSPW]=PxxHotM
PyyHot[sciSPW]=PyyHotM

# Set the ACD-based gains (scales) and instrumental contributions.
Gxx[sciSPW]=GxxM
Gyy[sciSPW]=GyyM

PIxx[sciSPW]=PIxxM
PIyy[sciSPW]=PIyyM

print('SPW ',sciSPW,': <Tsky+src_XX> = ',
      (Pxx[sciSPW].mean()-PIxx[sciSPW])/Gxx[sciSPW],'+/-',
      Pxx[sciSPW].std()/Gxx[sciSPW],
      '..... Pxx=',Pxx[sciSPW].mean(), 'Gxx=',Gxx[sciSPW],
      'PIxx=',PIxx[sciSPW])
print('      <Tsky+src_YY> = ',
      (Pyy[sciSPW].mean()-PIyy[sciSPW])/Gyy[sciSPW],'+/-',
      Pyy[sciSPW].std()/Gyy[sciSPW])

#-->endfor(all sciSPWs in the list)

# Save the original gains and instrumental powers for comparison with
# the upated values after the solution.

```

```

# Note: We have to use copy.deepcopy(), the simple assignment operator
# copies just the reference!

GxxOrig=deepcopy(Gxx)
GyyOrig=deepcopy(Gyy)

PIxxOrig=deepcopy(PIxx)
PIyyOrig=deepcopy(PIyy)

#-- Dimensions of the system.

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# How many net equations do we have per block of 1 time-sample.
# Be carefull: Do not flag the data that are *not* present in the spwList!
eqBlockSize=2*numSpw-len(flaggedData)

# The data has been read-out, we may close the (local) tools.
if (ownMsmd): myMsmd.done()
if (ownMS): myMS.close()

#-- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).

# Pre-calculated frequently repeating factor: Inverse number of time samples.
nsInv=1.0/nSamples

# Number of the model spectral windows.
nAtmSPWs=myAtm.getNumSpectralWindows()
# Number of all channels in the model SPWs. They have all the same structure.
nChans=myAtm.getNumChan(0)
# Number of central band-pass (with edges trimmed) channels.
cChans=len(fChans)

# Frequencies of channels in our model SPWs (central band-pass only).
spw_freqs=np.zeros((nAtmSPWs,cChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6 # MHz to Hz

    spw_freqs[atmSpw]=(fMin+df*np.arange(nChans))[fChans]

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,cChans))
dTdW=np.zeros((nAtmSPWs,cChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,cChans))
tau_1=np.zeros((nAtmSPWs,cChans))

# Target normalised spectrum

```

```

if (solarTgt):
    # Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
    SF=1.0+b_2/ Tsun00*np.log(spw_freqs/ f_LO)
else:
    # A flat normalized spectrum for non-solar target
    SF=np.ones(cChans)

#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
# We used those as weights in calculation of epsilon - the normalised
# inter-iteration change of operation-point at the very end of each iteration.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin
dG_max=0.001 # Relative scale dG/G
dTinst_max=0.001 # dPinst/G in kiloKelvins

# Iteration counter.
itCount=0
# Normalized inter-iteration change of operation point - its initial value.
# Anything bigger than 1.0
epsilon=10.0

# Set the starting values for iterations.
w0=pwv0
Tsun0xx=Tsun00
Tsun0yy=Tsun00

#-- Solution iterations: Minimize the mean value of variations.
while((epsilon>1.0) and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] 'natural' variables.
    J_wb=np.zeros((eqBlockSize,3)) # 3 vars repeating in all spws
    # G-block matrix: Jacobian part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each SPW:Pol.
    # A-block matrix: Jacobian part for [dG,dP_inst] for ACD calibration powers.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, ambient.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, hot.
    # M-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] in array-mean eqs.
    # This matrix is constant and can be defined already here.
    J_M=np.array([[0.0, nsInv, 0.0],
                  [0.0, 0.0, nsInv]])

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))
    # RHS-vector 'appendix' - amb. & hot ACD calibration powers (time-averaged).

```

```

CCA=np.zeros((eqBlockSize,1))
CCH=np.zeros((eqBlockSize,1))
# RHS-vector 'appendix 2' - array-mean values of target T_B
CCM=np.zeros((2,1))

# Rezidual mean sky temperature - after subtracted brightness of the
# atmosphere (+CMB) and (attenuated) source. We keep it for all sci SPWs and
# both polarizations and print it out at the end of every iteration cycle
# as a diagnostics of convergence and solution quality.

resTxx={}
resTyy={}

# Mapping between spw IDs and the row number in the system of equations.
# We need it for final diagnostic plots. It may be non-trivial as some
# spw:pol pairs may be flagged and ignored, causing irregularity in the map.

spw2rowXX={}
spw2rowYY={}

#- Re-calculate ATM model with updated <PWV> and <Tsun>.

# Set current estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

#- Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
# Calculate factors dependent only on the mmodel, not the data.
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Limit the spectrum to the central fChans reagon as we have trimmed
    # the edge channels in ACD-based spectral normalized gain gXX/gYY.
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001*Tebb[fChans]

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value'][fChans]
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0- dws, 'mm'))[1]['value'][fChans]

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001*(TebbH- TebbL)/(2.0*dws)

    #- Opacities & transmisivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)

    # Transmissivity (spectral). Limited to fChans.
    trans[atmSPW]=np.exp(- opacity[fChans])
    # Wet opacity per 1mm PWV scaled by airmass. Limited to fChans.
    tau_1[atmSPW]=(airMass/ w0)*tau_wet[fChans]

```

```

#--> endfor(modelSPWs)

#- Loop over all SCI spws - combine model-only dependent factors with
# sciSPW-dependent data and instrumental factors and build the system.

# Count the actual number of equations - some rows may be flagged.
eqCount=0

for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Separate procedure by polarizations for combination with real data.

    #- Pol XX

    # Full identifier of data: sciSPW:pol
    sciDataID=str(sciSPW)+' :XX'

    if (sciDataID not in flaggedData):

        #- Model + instrumental factors combined.

        # We do weighting of equations by inverse gain in order to compensate
        # different powers (and gains) in AC and SQLD SPWs. Consequently,
        # the equations are expressed in equivalent temperatures, which are
        # (approximatelly) the same for all SPWs.
        # Weight factor
        WF=1.0/Gxx[sciSPW]

        # Sensitivity factor for PWV.
        SWxx=(- eta_l* eta_F* Tsun0xx* np.mean(gXX[sciSPW]*trans_M*tau_1M*SF_M)
            +(1.0- eta_F+eta_F* eta_l)* np.mean(gXX[sciSPW]*dTdW_M)*Gxx[sciSPW])

        # Sensitivity factor for the (apparent) solar-brighthness temperature.
        SBxx=Gxx[sciSPW]* eta_l* eta_F* np.mean(gXX[sciSPW]*trans_M*SF_M)

        # Model T_sky as viewed by the receiver with spectral gain gXX.
        TskyXX=(eta_l* eta_F* Tsun0xx* np.mean(trans_M*SF_M*gXX[sciSPW])
            +(1.0- eta_F+eta_F* eta_l)* np.mean(TbsW0_M*gXX[sciSPW])
            +eta_F*(1.0- eta_l)* Tamb)

        #- Build the matrix: Natural (dW & dB sensitivities) block.

```

```

# Set the (weighted) sensitivity factors for the current SPW:pol as
# a row in the N-block/tile for the linearized-system Jacobian matrix.
J_wb[eqCount]=WF*np.array([SWxx,SBxx,0.0])

#- Build the matrix: Instrumental (G & P_inst sensitivities) block.

# We have to prepend a proper offset for the G and P_inst vars as
# they are independent for each SPW and polarization.
bOffset=2*eqCount

# G-block scaled by the inverse gain.
J_gp[eqCount,bOffset]=WF
J_gp[eqCount,bOffset+1]=TskyXX*WF

#- Build the matrix: ACD powers (amb & hot) 2-block.

# Ambient-power sub-block. Scale between the G* / P_inst* used in
# J_wb and J_gp, and G / P_inst used in the ACD cal equations by the
# attenuation factor AF=G*/G=(P_sky-Po)/(P_off-Po).

# Attenuation factor (af>1) is calculated only in SQLD and mapped.
af=AFxx[afMap[sciSPW]]

# A-ambient sub-block Jacobian elements weighted by inverse gain.
J_cdA[eqCount,bOffset]=af*WF
J_cdA[eqCount,bOffset+1]=af*Tamb*WF

# A-hot-power sub-block weighted by inverse gain.
J_cdH[eqCount,bOffset]=af*WF
J_cdH[eqCount,bOffset+1]=af*Thot*WF

#- Build the RHS vector: Target data

# RHS: Pxx(t) with their stationary (in the working point) instrumental
# and ATM-based parts subtracted.
dPxx=Pxx[sciSPW]-PIxx[sciSPW]-Gxx[sciSPW]*TskyXX

# Set it as the RHS component, weighted by inverse gain.
CC[eqCount]=dPxx*WF

#- Amend the RHS vector: Calibration data.

# Attenuation-based power offset. We have it defined for all SPWs,
# however, non-zero it is only in SQLDs.
aoff=AOxx[sciSPW]

# Ambient-load power with its working-point model part subtracted.
CCA[eqCount]=WF*(PxxAmb[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Tamb
+aoff)

# Hot-load power with its working-point model part subtracted.
CCH[eqCount]=WF*(PxxHot[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Thot
+aoff)

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resTxx[sciSPW]=dPxx.mean()/Gxx[sciSPW]

```

```

# Mapping between SPW:XX and equation number.
spw2rowXX[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#- Pol YY

# Full identificator of data: sciSPW:pol
sciDataID=str(sciSPW)+'::YY'

if (sciDataID not in flaggedData):

    #- Model + instrumental factors combined.

    # Weight factor - see explanation at XX pol above.
    WF=1.0/ Gyy[sciSPW]

    # Sensitivity factor for PWV (weighted by 1/G).
    SWyy=(- eta_l* eta_F* Tsun0yy* np.mean(gYY[sciSPW]*trans_M*tau_1M* SF_M)
          +(1.0- eta_F+eta_F* eta_l)* np.mean(gYY[sciSPW]*dTdW_M))*Gyy[sciSPW]

    # Sensitivity factor for the (apparent) solar T_B (weighted by 1/G).
    SByy=Gyy[sciSPW]* eta_l* eta_F* np.mean(gYY[sciSPW]*trans_M* SF_M)

    # Model T_sky as viewed by the receiver with spectral gain gYY.
    TskyYY=(eta_l* eta_F* Tsun0yy* np.mean(trans_M* SF_M* gYY[sciSPW])
            +(1.0- eta_F+eta_F* eta_l)* np.mean(TbsW0_M* gYY[sciSPW])
            +eta_F* (1.0- eta_l)* Tamb)

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the (weighted) sensitivity factors for the current SPW:pol as
    # a row in the block/tile for the linearized-system Jacobian matrix.
    J_wb[eqCount]=WF* np.array([SWyy,0.0,SByy])

    #- Build the matrix: Instrumental (G & P_inst sensitivities) block.

    # We have to prepend a proper offset for the G and P_inst vars as
    # they are independendt for each SPW and polarization.
    bOffset=2*eqCount

    J_gp[eqCount,bOffset]=WF
    J_gp[eqCount,bOffset+1]=TskyYY*WF

    #- Build the matrix: ACD powers 2-block.

    # Attenuation factor G*/G. Mapped from the corresponding SQLD.
    af=AFyy[afMap[sciSPW]]

    # Ambient-power sub-block.
    J_cdA[eqCount,bOffset]=af*WF
    J_cdA[eqCount,bOffset+1]=af* Tamb*WF

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=af*WF

```

```

J_cdH[eqCount,bOffset+1]=af*Thot*WF

#- Build the RHS vector: Target data.

# RHS: Pyy(t) with their stationary (in the working point) instrumental
# and ATM-based parts subtracted.
dPyy=Pyy[sciSPW]-PIyy[sciSPW]-Gyy[sciSPW]*TskyYY

# Set it as the RHS component.
CC[eqCount]=dPyy*WF

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOyy[sciSPW]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=WF*(PyyAmb[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Tamb
               +aoff)
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=WF*(PyyHot[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Thot
               +aoff)

#- Diagnostics

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTyy[sciSPW]=dPyy.mean()/Gyy[sciSPW]

# Mapping between SPW:YY and equation number.
spw2rowYY[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#--> endfor(all sci SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Ooops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPW:pol channels.")
    return False

#-- Grand system assembly and inversion by parts. We use the approach from
#   extMatInv() for inversion of extended-system matrix.

#- Ammend the RHS vector: Array-mean target brightness temperatures.
# Here we work already with temperatures, thus no 1/G scaling applies.

# Target T_B with the model at work-point subtracted.
CCM[0]=T_tgt-Tsun0xx
CCM[1]=T_tgt-Tsun0yy

#- Calculate the small-block matrices and their inversions needed for

```

```

# grand-system inversion.

# 1. Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)          #  $N^T$ 
J_K=np.matmul(J_wb_T,J_wb)        #  $(N^T \times N)$ 
J_K_inv=np.linalg.inv(J_K)         #  $(N^T \times N)^{-1}$ 

# 2. Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)          #  $G^T$ 

# 3. ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
# Attention!!! NumPy's transpose() makes just another view of a matrix,
# we have to make a copy, otherwise subsequent scaling would scale the
# original J_cdA and J_cdH matrices, too!

J_cdA_T=np.transpose(J_cdA).copy()  #  $A^T$ 
J_cdH_T=np.transpose(J_cdH).copy()  #  $H^T$ 

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T,J_cdH_T]) from already scaled-up transposed sub-blocks.
J_cd=np.vstack([J_cdA_T,J_cdH_T]).copy()
J_cd_T=calWeight*np.transpose(J_cd)  #  $\text{calWeight} * (A | H)^T$ 

# 4. Array-mean (M-block) related matrices. Incorporate mWeight into  $M^T$ , in
# line with what we have done with calWeight for the  $A^T$  and  $H^T$  matrices.
M_T=np.array([[0.0, 0.0],
               [nsInv*mWeight, 0.0],
               [0.0,nsInv* mWeight]])  #  $m\text{Weight} * M^T$ 

# Pre-calculated nSamples*mWeight*( $M^T \times M$ ) matrix.
# Like with  $A^T \times A$ , the  $M^T \times M$  product is scaled-up by mWeight just once
# (no mWeight squared!). Moreover, we scale it up here yet by nSamples,
# in order to avoid trivial product nsInv*nSamples=1 in the following calcs.
nsMTM=np.array([[0.0, 0.0, 0.0],
                 [0.0, nsInv*mWeight, 0.0],
                 [0.0, 0.0, nsInv*mWeight]])

# Frequently repeating blocks. NM_inv plays a role of modified J_K_inv.
NM=J_K+nsMTM                      #  $J_K + n\text{Samples} * m\text{Weight} * M^T \times M$ 
NM_inv=np.linalg.inv(NM)           #  $NM^{-1}$ 

# Off-diagonal blocks of the inverted natural sub-matrix (Ainv).
J_M_inv=nsInv*(NM_inv-J_K_inv)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T,J_gp)         #  $(N^T \times G)$ 

```

```

Lb=np.transpose(Rb)                                # ( $G^T \times N$ )

# Lower-right corner
LRc=nSamples* np.matmul(J_gp_T, J_gp)+np.matmul(J_cd_T, J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(NM_inv, Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc- nSamples* np.matmul(Lb, AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv, LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=J_M_inv- np.matmul(AinvRb, Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e. multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T, CC)+np.matmul(M_T, CCM) # 3 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T, CCsum)+nSamples* np.matmul(M_T, CCM) # 3 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T, CCsum)                # 2*eqCount x 1
        +np.matmul(J_cdA_T, CCA)
        +np.matmul(J_cdH_T, CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (3 x 1)
nvConst=np.matmul(Minv11_OD, C_K_wb_sum)+np.matmul(Minv12, C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the

```

```

# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
# NB: Actually, with M-blocks included, it contains a small constant part,
# too, equal to J_K_inv x M_T x CCM, but we keep it here (hidden in
# C_K_wb).
nvVar=np.matmul(J_K_inv,C_K_wb) # 3 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
dW = varNat[0]
dBxx=varNat[1]
dByy=varNat[2]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per SPW:POL.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errors for the over-determined set of eqs.

# err=f(X)-b =approx. f(X_0)+J(X_0) x (X -X_0) -b = J x dX - [b-f(X_0)]
# In our notation, dX is the [varNat,varInst]^T vector and the
# difference [b-f(X_0)] is [ CC, (CCA | CCH), CCM ]^T
# Now we express the err vector parts for sci, acd, and array-mean eqs.

# err=f(x)-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part. Sum over all SPW:Pol channels & times.
# Normalize it to the number of equations.
xiSci=np.sum(errVec*errVec)/(nSamples*eqBlockSize)

# Xi Squares per SPW:Pol channel. Sum/average only over times.
xiSciPerChan=np.sum(errVec*errVec,axis=1)/nSamples
# Its mean and RMS over the SPW:POL channels.
xscMean=xiSciPerChan.mean()
xscRMS=xiSciPerChan.std()
# Find outliers ( > 3 sigma). The badData array contains indices of
# those equations, where the agreement between model and observation is
# exceptionally weak.
badData=np.flatnonzero(abs(xiSciPerChan-xscMean)>3.0*xscRMS)

# Show all per-channel errors as temperatures (K) and mark (we may auto-flag
# later) the SPW:Pol channels with bad data, if we have some.
# We do not have an inverse map to spw2rowXX/YY, so we have to traverse.

# XX pol
for spw,eqRow in spw2rowXX.items():
    print(str(spw)+' :XX: Error:',np.sqrt(xiSciPerChan[eqRow])*1000.0,'K')
    if eqRow in badData: print('\t\t\t!!! Bad data in',str(spw)+' :XX',

```

```

        (xiSciPerChan[eqRow]-xscMean)/xscRMS,
        'sigma from the mean.')

# YY pol
for spw,eqRow in spw2rowYY.items():
    print(str(spw)+' :YY: Error:', np.sqrt(xiSciPerChan[eqRow])*1000.0, 'K')
    if eqRow in badData: print('\t\t!!! Bad data in', str(spw)+' :YY:',
        (xiSciPerChan[eqRow]-xscMean)/xscRMS,
        'sigma from the mean.')

# err=f(x)-b: ACD-data part.
# Actually, this should be scaled by calWeight factor. On the other hand,
# we are interested in Xi2 normalised per one equation, so we would
# down-scale in the next calculation by the same factor. In order to avoid
# useless product calWeight*1/calWeight, we omit it already now.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the acd-data part.
# Normalize it to the number of ACD equations. Scale by just single
# block-size - we have deliberately ignored calWeight factor already above.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)
xiACD/=eqBlockSize

# err=f(x)-b: Array-mean T_B part.
# Here, the term J x dX = M x sum(CC_i) = [sum(dTxx_i), sum(dTyy_i)]^T =
# nS*[dBxx.mean(),dByy.mean()]^T is zero because dB.mmean() converged to 0.
errVecMeanTb=-CCM
# Normalized to one equation - we have mWeight*2 equations, but the same
# holds for mWeight factor as for the calWeight above.
xiTbM=np.sum(errVecMeanTb*errVecMeanTb)/2

# Grand-total Xi-square normalized to the number of equations for
# the natural/science, ACD, and mean-array sub-blocks.
xiSquare=xiSci+xiACD+xiTbM

#- All variations should tend to zero means.

dW_mean=dW.mean()
dBxx_mean=dBxx.mean()
dByy_mean=dByy.mean()

dB_mean=np.sqrt(dBxx_mean**2+dByy_mean**2)

# Update stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0xx+=dBxx_mean
Tsun0yy+=dByy_mean

# Update also the P_inst and G by the found differences.
# Find the maximum inter-iteration change for convergence diagnostics.
dT_inst_sup=-1.0
dG_rel_sup=-1.0

# Start with XX pol.
for sciSPW,eqCount in spw2rowXX.items():
    # Update the P_inst and G.
    P1xx[sciSPW]+=dPinst[eqCount]
    Gxx[sciSPW]+=dGinst[eqCount]

```

```

# Change of inst. power expressed in kiloKelvins.
dT_inst=abs(dPinst[eqCount]/ Gxx[sciSPW])
if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

# Change of gain in the relative scale.
dG_rel=abs(dGinst[eqCount]/ Gxx[sciSPW])
if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(SPWs with valid XX pol)

# YY pol.
for sciSPW,eqCount in spw2rowYY.items():
    # Update the P_inst and G.
    PIyy[sciSPW]+=dPinst[eqCount]
    Gyy[sciSPW]+=dGinst[eqCount]

# Change of inst. power expressed in kiloKelvins.
dT_inst=abs(dPinst[eqCount]/ Gyy[sciSPW])
if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

# Change of gain in the relative scale.
dG_rel=abs(dGinst[eqCount]/ Gyy[sciSPW])
if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(SPWs with valid YY pol)

#-- Convergence diagnostics / Iteration control.

# Calculate the normalized inter-iteration change of operation point.
# The factor 0.5 comes from sqrt(1/4) as we have 4 'orthogonal' variables.
epsilon=0.5*np.sqrt((dW_mean/ dwMean_max)**2+(dB_mean/ dbMean_max)**2+
                    (dG_rel_sup/ dG_max)**2+(dT_inst_sup/ dTinst_max)**2)

# Increment the iteration counter
itCount+=1

#- Diagnostic print-outs per-iteration.

# Updated scan-averaged PWV and Tsun after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('\nMean PWV for the scan,',tgtScan,' w0 = ',w0,'mm',
      ', mean Tsun (XX/YY) = ',1000.0*Tsun0xx,'/',1000.0*Tsun0yy,'K')

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale back from kiloKelvins to Kelvins.
resTxxArr=1000.0*np.array(list(resTxx.values()))
resTyyArr=1000.0*np.array(list(resTyy.values()))

print("\nres Txx: ", resTxxArr,"\nmean +/- RMS: ",
      resTxxArr.mean(), "+/-",resTxxArr.std())
print("\nres Tyy: ", resTyyArr,"\nmean +/- RMS: ",
      resTyyArr.mean(), "+/-",resTyyArr.std())

print("\nInstrumental characteristics shifts during the iteration:")
print("--- Pol XX--- ")

```

```

for sciSPW,eqCount in spw2rowXX.items():
    print("SPW =",sciSPW,"\tdT_inst =",dPinst[eqCount]/Gxx[sciSPW]*1000.0,"K",
          "\tdG/G =",dGinst[eqCount]/Gxx[sciSPW]*100,"%")

print("--- Pol YY--- ")
for sciSPW,eqCount in spw2rowYY.items():
    print("SPW =",sciSPW,"\tdT_inst =",dPinst[eqCount]/Gyy[sciSPW]*1000.0,"K",
          "\tdG/G =",dGinst[eqCount]/Gyy[sciSPW]*100,"%")

print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / end of the main solution-iteration loop.

#-- Diagnostic plots

# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

#-Set the plotting device.

plt.rcParams["figure.figsize"] = (30,20)
# Make XX and YY plots as subplots (horizontally)
ff,sps=plt.subplots(1,2)

# Antenna ID formatted to 'NN' string comes to the PNG filename.
antIDstr='{0:02d}'.format(antID)

# X-axis label just for the bottom sub-plot.
sps[0].set_xlabel('Time [MJDsec]')
sps[1].set_xlabel('Time [MJDsec]')

# Add the antenna ID/name as a plot title.
plt.suptitle('Antenna '+antIDstr+'/'+myAntName)

# Color-map for SPWs, incl. line-style.
# We have minimum 2 and maximum 12 SPWs in the spwList argument.
# If we use all 12 spws, linestyle correspondence to the spw type follows:
csline={5: '-r', 7: '-g', 9: '-b', 11: '-k',          # AC resolved
        6: '--r', 8: '--g', 10: '--b', 12: '--k',     # AC integrated
        0: ':r', 1: ':g', 2: ':b', 3: ':k'}          # SQLD

#- Make the *analytical* plots -least-square residuals etc.

# XX pol

for spw,eqRow in spw2rowXX.items():
    # Panel row 1: Residuals/errors scaled to temperature in Kelvins.
    # Note: Be carefull about scaling if we use the WF weight factor above
    # other than the 'natural' WF=1/Gxx[spw]
    sps[0].plot(time,1000.0*errVec[eqRow],csline[spw])

# YY pol

for spw,eqRow in spw2rowYY.items():
    # Panel row 1: Residuals/errors scaled to temperature in Kelvins.
    # Note: Be carefull about scaling if we use the WF weight factor above
    # other than the natural WF=1/Gyy[spw]
    sps[1].plot(time,1000.0*errVec[eqRow],csline[spw])

```

```

#- Save the plots to a file and close.
plt.savefig(plotDir+'/errs_ant_'+antIDstr+'.png')
plt.show()
plt.close()

#- Make the full-solution plots: PWV(t) and T_b(t).

# Add variations to their time-averages. Temperatures are scaled back
# from kiloKelvins to Kelvins.
pwv=w0+dW
TsunXX=1000.0* (Tsun0xx+dBxx)
TsunYY=1000.0* (Tsun0yy+dByy)

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)
# Add the antenna ID/name as a plot title.
plt.suptitle('Antenna '+antIDstr+'/'+myAntName)

# Left panel: Apparent target brightness per polarization.
sp1.plot(time,TsunXX,'-r')
sp1.plot(time,TsunYY,'-b')

# Right panel: PWV.
sp2.plot(time,pwv,'-g')

plt.savefig(plotDir+'/dWdB/dWdB_ant_'+antIDstr+'.png')
plt.show()
plt.close()

#-- Final diagnostics print-outs.

#- Show, how much the instrumental variables G and P_inst have changed from
# their nominal ACD-based values.

print("--- Accumulated instrumental characteristics changes ---")

for spw in spw2rowXX.keys():
    # Gain as a relative change in percents.
    dGxx=100.0* (Gxx[spw]-GxxOrig[spw])/ GxxOrig[spw]
    # Instrumental power expressed as T_inst change in Kelvins.
    dPxx=1000.0* (PIxx[spw]-PIxxOrig[spw])/ Gxx[spw]

    print('\t',spw,':XX: dGxx =',dGxx,'%','\tdP_Ixx =',dPxx,'K')

for spw in spw2rowYY.keys():
    # Gain as a relative change in percents.
    dGyy=100.0* (Gyy[spw]-GyyOrig[spw])/ GyyOrig[spw]
    # Instrumental power expressed as T_inst change in Kelvins.
    dPyy=1000.0* (PIyy[spw]-PIyyOrig[spw])/ Gyy[spw]

    print('\t',spw,':YY: dGyy =',dGyy,'%','\tdP_Iyy =',dPyy,'K')

#- Print total quadratic error for the variables.
print("\n===== Xi-squares ===== SCI: ",xiSci,
      ", ACD:",xiACD,", Array-mean: ",xiTbM,

```

```

" | Total-per-equation:", xiSquare,
"\n-----> Estimated total error per equation:",
np.sqrt(xiSquare)*1000.0,"K -----\\n\\n")

```

```
#-- Export the results.
```

```
# 'Compactify' the per-polarization T_B arrays.
```

```
Tsun=np.array([TsunXX,TsunYY])
```

```
return {'time':time,'pwv':pwv,'T_B':Tsun,'J':J_wb,'C':CC}
```

```
#-----
```

```
def 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().

```
    """
```

```
#--- Static parameters
```

```
#- Dirs
```

```
plotDir='./sciPWVs'
```

```
#- Tools (to be extended by possible global tools as params)
```

```
myMS=mstool()
```

```
myMS.open(vis)
```

```
myMsmd=msmdtool()
```

```
myMsmd.open(vis)
```

```
#- Source
```

```
# Initial guess of the solar brightness temperature.
```

```
Tsun0=6000.0
```

```
# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
```

```
# in Allisandrakis et al. (2020).
```

```
b_2=-1127.0 # = 2x (-563.5K) as per Table 2 in Allisandrakis et al. (2020).
```

```
# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
```

```
f_LO=1.98e11
```

```
#- Antenna
```

```
# Rear-spillover + ohmic antenna efficiency (see Tab. ? in White et al. 2017).
```

```
eta_l=0.97
```

```
# Forward-spillover antenna efficiency (see Tab. ? in White et al. 2017).
```

```

eta_F=0.95

# Extract Tamb from ASDM_CADEVICE.
# NB: Now we use aU.getLoadTemperatures() for all antennas and scans and
# then pick just our antenna and scan. Do it better later.
TLoad = aU.getLoadTemperatures(vis, doplot = False,
                               warnIfNoLoadTemperatures = True,
                               mymsmd=myMsmd)
Tamb = TLoad[antID][atmScan]['amb']

#- SPWs

# Integration limits for resolved AC SPWs.
startChan=7
stopChan=120

#- Atmosphere

# Calculate limiting TbSky for infinite PWV: It is a constant close to
# the tropospheric temperature for the ALMA array (MeteoCentral).
TbsInf=myAtm.getTebbSky(spwid=0,nc=0,
                       wh2o=qa.quantity(1000000.0,'mm'))['value'][0]

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
w0=getPWVforAnt(vis,tgtScan,antID)['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',w0,' @airMass = ',airMass)
print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#--- Read-out the sky brightness temperatures for all SPWs in the spwList.

# Store the frequency-averaged sky temperatures in a map
# spwId -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW
# id, which is sparse in the set of integers.
Txx={}
Tyy={}

for sciSPW in spwList:

    # Read ACD calibrated sky temperatures for the target defined by targetScan.
    time,TxxS,TyyS=tgt_Tb(vis,atmScan,tgtScan,antID,sciSPW,myMS,myMsmd)

```

```

# We have different handling for frequency-resolved and integrated SPWs.
nChans=TxxS.shape[0]

if (nChans > 1):

    # Resolved AC SPWs.
    # Trim the spectral edges by posing the integration limits and average
    # over the remaining channels.

    TxxM=np.mean(TxxS[fChans],axis=0)
    TyyM=np.mean(TyyS[fChans],axis=0)

else:

    # Integrated ACs or SQLDs.
    # The single channel already represents a mean over the band-pass.

    TxxM=TxxS[0]
    TyyM=TyyS[0]

    # The SQLD spws have 12x higher cadence than the ACs. Rebin the SQLD
    # data in time to have a homogeneous series. Do the same for time axis.
    if sciSPW in [0,1,2,3,13,14,15,16]:
        TxxM=rebin(TxxM,12)
        TyyM=rebin(TyyM,12)
        time=rebin(time,12)

    # Append the frequency-averaged time series for each SPW to the Tsky arrays.
    Txx[sciSPW]=TxxM
    Tyy[sciSPW]=TyyM

#-->endfor(sciSpws in list)

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# The data are read-out, close the tools.
myMsmc.done()
myMS.close()

#-- Allocate arrays to be used for matrix A and the RHS-vector C calculations
# from the ATM model. They have just nAtmSPWs components.

# Find the array dims.
nAtmSPWs=myAtm.getNumSpectralWindows()
nChans=myAtm.getNumChan(0)

# Allocate the model sky (source + atm) temperature.
Tsky=np.zeros(nAtmSPWs)

# Allocate sensitivity factors for dW and dB (Jacobian elements).
SW=np.zeros(nAtmSPWs)
SB=np.zeros(nAtmSPWs)

# Allocate and find frequencies of channels in all our ATM SPWs.

```

```

spw_freqs=np.zeros((nAtmSPWs,nChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6 # MHz to Hz

    spw_freqs[atmSpw]=fMin+df*np.arange(nChans)

#--- Improve a guess of the mean solar brightness temperature.

TsolEst=np.zeros(numSpw)

spwCount=0
for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    # Sky brightness in a given SPW for the current PWV estimate.
    TbsEst=myAtm.getAverageTebbSky(atmSPW, qa.quantity(w0, 'mm')) ['value'] [0]

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW) [1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW) [1] ['value'])
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)

    # Mean (spw-averaged) transmissivity
    transM=np.mean(np.exp(- opacity))
    # Frequency-dependent part of solar spectrum weighted by transmissivity.
    SFtrans=np.mean(np.exp(- opacity)* np.log(spw_freqs[atmSPW]/ f_LO))

    # <Tsky> estimation from measurements, averaged over time and pols.
    TskyEst=0.5*(Txx[sciSPW].mean()+Tyy[sciSPW].mean())

    TsolEst[spwCount]=(TskyEst-(1.0-eta_F+eta_l)*TbsEst
                        -eta_l*eta_F*b_2*SFtrans
                        -(1.0-eta_l)*Tamb)/(eta_l*eta_F*transM)

    spwCount+=1

#--> endfor(sci SPWs)

# Make a new guess of Tsun as an average.
Tsun0=TsolEst.mean()
print("Tsun = ",Tsun0,"+/-",TsolEst.std())

#-- Average solar spectrum (dimension-less, normalised to 1.0 at f=f_LO)

# It has the same dimensions as spw_freqs, i.e. nAtmSPWs x nChans
SF=1.0+b_2/ Tsun0*np.log(spw_freqs/ f_LO)

#--- Solve the set of non-linear equations for full PWV and full T_B.
# Use the least square minimizer, iterate until we are not close to zero.

```

```

# Do minimisation for each time instant separately.

#-- Iteration static setup.

# Half dw for the estimation of d/dw0 derivatives
dws=0.05 #mm
# Half dTb for the estimation of d/dT_B derivatives
dTs=5.0 # K

# Stopper based on gradient of the least-square functional.
grad_max=0.01

#-- Time loop over all time samples.

# Initial estimation pf pwv and T_B
pwv=w0
T_B=Tsun0

# Resulting time series.
pwv_TS=np.zeros(nSamples)
Tb_TS=np.zeros(nSamples)

for ts in range(nSamples):

    #-- Newton-Raphson iterations for least-square residuum minimiser.

    # Initial value of the minimizer gradient - any bigger then the stopper.
    # We calculate the gradient inside the loop.
    grad_curr=0.5
    # Iteration counter.
    itCount=0

    while((grad_curr > grad_max) and (itCount < 20)):

        # 'Operation points': We evaluate the minimizer gradient in the current
        # pwv and T_B, and the surrounding points.
        op_points=[ [pwv,T_B],
                    [pwv+dws,T_B],
                    [pwv-dws,T_B],
                    [pwv,T_B+dTs],
                    [pwv,T_B-dTs]]

        GGP=np.zeros((5,2))

        opIndex=0

        for opp in op_points:

            JJ=np.array([])
            BB=np.array([])

            wC=opp[0]
            T_Bc=opp[1]

            # Set current estimation of PWV to the ATM model.

```

```

myAtm.setUserWH2O(qa.quantity(wC, 'mm'))

#- Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
# Some factors depend only on the ATM mmodel, not the sciSPW data.
for atmSPW in range(nAtmSPWs):

    #- Model value of the atm brightness with current pwv.
    TbsW0=myAtm.getTebbSkySpec(atmSPW, qa.quantity(wC, 'mm'))[1]['value']

    #- Slope of the atm brightness with current pwv.
    TbsW0H=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(wC+dws, 'mm'))[1]['value']
    TbsW0L=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(wC-dws, 'mm'))[1]['value']

    dTdW=(TbsW0H- TbsW0L)/(2.0*dws)

    #- Opacities & transmissivity (spectral)

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_l=airMass*tau_wet/wC
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)
    # Transmissivity (spectral).
    trans=np.exp(- opacity)

    #- Tsky model. Weighted averages over the SPW.
    Tsky[atmSPW]=(eta_l*eta_F*T_Bc*np.mean(trans*SF[atmSPW])
                  +(1.0-eta_F+eta_l)*np.mean(TbsW0)
                  +(1.0-eta_l)*Tamb)

    #- Sensitivities = Jacobian elements. Weighted averages over the SPW.

    # Sensitivity factor for PWV.
    SW[atmSPW]=(- eta_l*eta_F*T_Bc*np.mean(trans*tau_l*SF[atmSPW])
                 +(1.0-eta_F+eta_l)*np.mean(dTdW))

    # Sensitivity factor for the (apparent) solar-brightness.
    SB[atmSPW]=eta_l*eta_F*np.mean(trans*SF[atmSPW])

#-->endfor(ATM spws)

#-- Loop over all sciSPWS in the list.
for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    # Set the sensitivity factors for the current SPW as two rows per pol.
    # Here, the rows are repeating in the Jacobian matrix as the sens.
    # factors calculated from the model do not depend on XX/YY.
    JJ=np.append(JJ, [SW[atmSPW], SB[atmSPW]])
    JJ=np.append(JJ, [SW[atmSPW], SB[atmSPW]])

```

```

    # RHS: Measured Txx and Tyy, subtracted from the model Tsky
    BB=np.append(BB,Tsky[atmSPW]-Txx[sciSPW][ts])
    BB=np.append(BB,Tsky[atmSPW]-Tyy[sciSPW][ts])
    #--> endfor(sci spws in list)

    # Re-shape the Jacobian matrix to be nVars x (nPols*numSpw) array
    JJ=np.reshape(JJ,(2*numSpw,2))

    # Re-shape the RHS to be a column vector of (nPols*numSpw) dim.
    BB=np.reshape(BB,(2*numSpw,1))

    # Transposed Jacobian
    JJT=np.transpose(JJ)

    # Minimizer: Gradient of the merit function. We look for its null point.
    GGP[opIndex]=np.transpose(np.matmul(JJT,BB))

    opIndex+=1
    #-->endfor(opIndex)

    #-- Minimizer gradient and its Jacobian

    GG=GGP[0]

    GG_wPlus=GGP[1]
    GG_wMinus=GGP[2]

    GG_TbPlus=GGP[3]
    GG_TbMinus=GGP[4]

    JG11=(GG_wPlus[0]-GG_wMinus[0])/(2.0*dws)
    JG21=(GG_wPlus[1]-GG_wMinus[1])/(2.0*dws)

    JG12=(GG_TbPlus[0]-GG_TbMinus[0])/(2.0*dTs)
    JG22=(GG_TbPlus[1]-GG_TbMinus[1])/(2.0*dTs)

    # Jacobian inversion.

    iDet=1.0/(JG11*JG22-JG21*JG12)

    iG11=iDet*JG22
    iG12=-iDet*JG21
    iG21=-iDet*JG12
    iG22=iDet*JG11

    # N-R shift
    dWnr=-(iG11*GG[0]+iG12*GG[1])
    dTnr=-(iG21*GG[0]+iG22*GG[1])

    # N-R update of variables.
    pwv+=dWnr
    T_B+=dTnr

    # Gradient current magnitude
    grad_curr=np.sqrt(GG[0]*GG[0]+GG[1]*GG[1])

    itCount+=1
    print("After iteration no. ",itCount," ||grad|| = ",grad_curr)

```

```

    #-->endwhile(grad_curr > grad_max)

    print("Finished for time sample no. ",ts, ", PWV = ",pwv," , T_Sun = ",T_B)

    pwv_TS[ts]=pwv
    Tb_TS[ts]=T_B

#--> endfor(time samples)

return time,pwv_TS,Tb_TS

#-----

def dWdBdG2_weights(vis,atmScan,tgtScan,myAtm,antID,spwList,T_tgt=-1.0e6,
                    flaggedData=[],myMS=None,myMsmd=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 then 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 wltSize.

    Results: It has been found that the optimum value for calWeight lies between
    100 and 300. At low values, the  $\chi^2$  for instrumental variables is high and
    is improving with increasing calWeight. Nevertheless, around 200 the
    improvement of  $\chi^2$  inst. slows down, while, at the same moment, the 2nd part
    of the error vector, the  $\chi^2$  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 catWeght=200. This is actually the value eventually used in production version of dWdBdG2().

"""

#-- Static parameters

#- Tools, global data

Flags used to check, whether we should close the tool later.

ownMS=False

ownMsmd=False

Open own tools if 'global' not provided

if (not myMS):

Open the ms tool

myMS=msstool()

myMS.open(vis)

ownMS=True

if (not myMsmd):

Open the msmd tool.

myMsmd=msmdtool()

myMsmd.open(vis)

ownMsmd=True

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

if (not acdLoads):

acdLoads=aU.getLoadTemperatures(vis, doplot = False,
warnIfNoLoadTemperatures = True,
mymsmd=myMsmd)

#- Antenna

antNames=myMsmd.antennanames()

myAntName=antNames[antID]

antType=myAntName[:2]

*# Map the antenna efficiencies by antenna type. As per investigation from
merged T_sun temperatures form 2023 and 2026 data (project_22_25_Tb.ods).*

eta_l_map={'CM':0.941, 'DA':0.947, 'DV':0.956}

eta_F_map={'CM':0.891, 'DA':0.897, 'DV':0.906}

Rear-spillover + ohmic antenna efficiency - see project_22_25_Tb.ods

eta_l=eta_l_map[antType]

Forward-spillover antenna efficiency - see project_22_25_Tb.ods

eta_F=eta_F_map[antType]

We have two exception that seem to have systematically lower

antenna efficiencies: DA64 and DV08 - see project_22_25_Tb.ods

if myAntName=='DA64':

eta_l=0.929

eta_F=0.880

if myAntName=='DV08':

eta_l=0.943

eta_F=0.894

Extract ambient & hot temperatures for the current antenna and scan.

```

# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# Optimized weight of the ACD calibration equations in the system.
# Search mark in dWdBdG2_weights(): CW_1W
calWeight=100000.0

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.300 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

"""
# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff(arcsec, incl. cos(delta)): 822   155.8
'''
x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.
"""

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!

```

```

b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temoerature corrected to limb brightenning - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu)

# Weight of the array-mean equation <T_B(t)>=T_tgt_arrayMean in the system.
# If T_tgt is not explicitly set (the default value is -1MK), do not apply
# this equation at all. This can be achieved by setting its weight to 0.
if (T_tgt < -10000.0):
    # Do not apply T_tgt bond. If mWeight is zero, T_tgt is irrelevant.
    T_tgt=0.0
    mWeight=0.0
else:
    # We use kiloKelvins in calculations, re-scale T_tgt to kK, too.
    T_tgt*=0.001
    # Set the experimentally found optimized mWeight.
    # Search mark in dWdBdG2_weights(): MW_1W
    mWeight=5000.0

#- SPWs

# LO frequency. We take it as a reference frequency for the solar spectrum.
f_LO=1.98e11 # in Hz

# Integration limits for resolved AC SPWs.
startChan=8
stopChan=120

fChans=range(startChan,stopChan)

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmid) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,' @airMass = ',airMass)
print('Estimated T_B = ',1000.0*Tsun00)
print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#--- Read-out the sky+src powers for all SPWs in the spwList.

# Store the frequency-averaged sky+src powers in a map.
# spwId -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW

```

```

# id, which is sparse in the set of integers.
Pxx={}
Pyy={}

#- Similar maps for the gains and instrumental powers P_inst=G*T_inst.
# Here, all gains and P_inst are, in fact, the G* and P_ints* values for the
# higher-attenuation solar regime. In order to append the ACD calibration
# equations, whose measurements are performed in the norma-attenuation regime
# in the preceding ATM scan, we determine also the attenuation factors
# AF=G*/G and related power offsets AO.

# Frequency-averaged gains. The scale for the spectral-resolved norm. gains.
Gxx={}
Gyy={}

# Attenuation factors G/G*=(P_sky-P0)/(P_off-Po). Attributed to every-spw
# Gxx & Gyy, but actually calculated just in SQLD spws, the rest is mapped.
AFxx={}
AFyy={}

afMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

# Attenuation-based power offsets AO=-Po*(P_sky-P_off)/(P_off-Po).
# They are non-zero only in SQLD spws.
AOxx={}
AOyy={}

# Spectral gain normalized to 1.0. Actual ACD-based values in spectral AC spws,
# a trivial 1.0 ndarray for integrated-AC and SQLD spws.
gXX={}
gYY={}

# Frequency-averaged instrumental powers.
PIxx={}
PIyy={}

#- Finally, the raw ACD-calibration powers.

# Ambient-load powers.
PxxAmb={}
PyyAmb={}

# Hot-load powers.
PxxHot={}
PyyHot={}

#-- Read the data and ACD calibration tables in a loop over all sciSPWs.

for sciSPW in spwList:

    #- Raw data & ACD tables

    # Read raw science powers for the target defined by targetScan.
    time,P_src=tgt_pow(vis,tgtScan,antID,sciSPW,myMS=myMS,myMsmid=myMsmid,
                        subScans='on')

    PxxS=P_src[0]
    PyyS=P_src[1]

```

```

# Read ACD calibrations - gains, Tinst, and calibration powers.
calDict=acd_calTabs(vis, atmScan, antID, sciSPW, myMS=myMS, myMsmc=myMsmc,
                    acdLoads=acdLoads)

Pcals=calDict['calPows']

PxxAmbS=Pcals['P_amb'][0]
PyyAmbS=Pcals['P_amb'][1]

PxxHotS=Pcals['P_hot'][0]
PyyHotS=Pcals['P_hot'][1]

# We have different handling for frequency-resolved and integrated SPWs.
# Find the number of channels in the observed SPW.
nChansO=PxxS.shape[0]

if (nChansO > 1):

    #- Resolved AC SPWs.
    # Trim the spectral edges by posing the integration limits and average
    # over the remaining fChans channels.

    # Science-target powers.
    PxxM=np.mean(PxxS[fChans], axis=0)
    PyyM=np.mean(PyyS[fChans], axis=0)

    # Calibration (ACD) powers.
    PxxAmbM=np.mean(PxxAmbS[fChans], axis=0)
    PyyAmbM=np.mean(PyyAmbS[fChans], axis=0)

    PxxHotM=np.mean(PxxHotS[fChans], axis=0)
    PyyHotM=np.mean(PyyHotS[fChans], axis=0)

    #- Instrumental factors.
    # Store the normalized (to 1.0) spectral gains (arrays, constant) and
    # their scales (numbers, to be re-fined by the solver) separately.

    # Raw G* gains in a full SPW. We may flip the array (see below) so better
    # to do so before its limiting to fChans - we need not to think whether
    # the fChans central range is placed symmetrically within its SPW.

    # Gain has shape 2 x nPols x nChans. The 1st index 0 means we pick the
    # *leading* P_off subscan - see the note in tgt_Tb().
    GxxS=1.0/ (calDict['iG*'][0,0])
    GyyS=1.0/ (calDict['iG*'][0,1])

    # T_inst in full SPW. The same holds for flipped/reversed SPWs as for G*.
    TinstXX=calDict['T_inst'][0]
    TinstYY=calDict['T_inst'][1]

    # Check, whether we need flip them - this is the case of the LSB spws,
    # which have decreasing/reversed frequency channels.
    spwFreqs=calDict['freqs']

    if (spwFreqs[0] > spwFreqs[1]):

```

```

    # Reversed SPW, flip the gain and T_inst arrays.

    GxxS=np.flip(GxxS)
    GyyS=np.flip(GyyS)

    TinstXX=np.flip(TinstXX)
    TinstYY=np.flip(TinstYY)

    # Now it is safe to select the central fChans band-pass.
    GxxS=GxxS[fChans]
    GyyS=GyyS[fChans]

    TinstXX=TinstXX[fChans]
    TinstYY=TinstYY[fChans]

    # Calculate the unity-normalized gains by dividing out the mean value.
    GxxM=GxxS.mean()
    GyyM=GyyS.mean()

    gXX[sciSPW]=GxxS/ GxxM
    gYY[sciSPW]=GyyS/ GyyM

    # Instrumental power averaged over our fChans range.
    PIxxM=np.mean(GxxS*TinstXX)
    PIyyM=np.mean(GyyS*TinstYY)

    # Keep the time-axis of the last resolved SPW as a reference timebase,
    # to which we intepolate the SQLD signals.
    timeBase=time

    # SQLD spws have an instrumental offset ralated to Pzero. It is calculated
    # below, in the single-channel SPWs branch. Here we set it to zero
    # because Pzero for AC-resolved spws is zero.
    AOxx[sciSPW]=0.0
    AOyy[sciSPW]=0.0

else:
    #- Integrated ACs or SQLDs.
    # The single channel already represents a mean over the spw band-pass.

    # Single-channel science data.
    PxxM=PxxS[0]
    PyyM=Pyys[0]

    # Channel-integrated SPWs may have higher cadence. For SQLDs it is true
    # always, but it may be the case of integrated ACs, too. Let us
    # put all data to the same, slowest, timebase (as per resoved AC spws).

    # Compare the cadence (elementary time-step, a.k.a. integration).
    if ((timeBase[1]-timeBase[0]) > (time[1]-time[0])):

        # Indeed, we have higher cadence. Re-bin in time.
        # Use the timeBase defined as a time-axis for resolved SPW.

        print('---> SPW',sciSPW,
              'has higher cadence: Rebinning to time-line of AC-resolved SPW.')

```

```

PxxM=rebin2(PxxM,time,timeBase)
PyyM=rebin2(PyyM,time,timeBase)
time=timeBase

#--> endif(higher cadence)

# The attenuation factors  $AF=(P_{sky}-P_o)/(P_{off}-P_o)$  and offsets  $AO$  should
# be, moreover, extracted from the SQLDss (ACs use mapping afMap
# defined above).
if sciSPW in [0,1,2,3,13,14,15,16]:

    # Extract the caliibration powers relevant for AF calculation
    Patm=Pcals['P_sky']
    # Poff is the only calPow that has a structure 2 x nPols x nChans,
    # similarly like gain. The others are just nPols x nChans 2D arrays.
    # Here, the first dimension '2' is for leading/trailing Poff subscans
    # and we pick the leading one (see the gain extraction above).
    Poff=Pcals['P_off'][0]
    Pzero=Pcals['P_zero']

    # Calculate AFs.
    # The 1st index in powers is pol, the 2nd means (the single) channel.
    AFxx[sciSPW]=(Patm[0,0]-Pzero[0,0])/(Poff[0,0]-Pzero[0,0])
    AFyy[sciSPW]=(Patm[1,0]-Pzero[1,0])/(Poff[1,0]-Pzero[1,0])

    # Calculates AOs.
    AOxx[sciSPW]=Pzero[0,0]*(AFxx[sciSPW]-1.0)
    AOyy[sciSPW]=Pzero[1,0]*(AFyy[sciSPW]-1.0)
else:
    # For non-SQLD spws the offsets are zero because Pzero is zero.
    AOxx[sciSPW]=0.0
    AOyy[sciSPW]=0.0

#--> endif(SQLD spws)

# Calibration data.
PxxAmbM=PxxAmbS[0]
PyyAmbM=PyyAmbS[0]

PxxHotM=PxxHotS[0]
PyyHotM=PyyHotS[0]

#- Instrumental factors.

# Gains (solar-attenuation): Take the leading-subscan based result.
GxxM=1.0/(calDict['iG*'][0,0,0])
GyyM=1.0/(calDict['iG*'][0,1,0])

# Create a fake spectral normalized gain for integrated SPWs.
# We use it in products with the ATM arrays (e.g. transmissivity), which
# are always spectral.
gXX[sciSPW]=np.ones(len(fChans))
gYY[sciSPW]=np.ones(len(fChans))

# Instrumental power: The channel 0 is the average already

```

```

    PIxxM=GxxM* calDict['T_inst'][0,0]
    PIyyM=GyyM* calDict['T_inst'][1,0]
#--> endif(resolved SPW)

# Append the frequency-averaged time series for each SPW to the maps.
Pxx[sciSPW]=PxxM
Pyy[sciSPW]=PyyM

# The calibration powers are already time-averaged.
PxxAmb[sciSPW]=PxxAmbM
PyyAmb[sciSPW]=PyyAmbM

PxxHot[sciSPW]=PxxHotM
PyyHot[sciSPW]=PyyHotM

# Set the ACD-based gains (scales) and instrumental contributions.
Gxx[sciSPW]=GxxM
Gyy[sciSPW]=GyyM

PIxx[sciSPW]=PIxxM
PIyy[sciSPW]=PIyyM

print('SPW ',sciSPW,': <Tsky+src_XX> = ',
      (Pxx[sciSPW].mean()-PIxx[sciSPW])/Gxx[sciSPW],'+/-',
      Pxx[sciSPW].std()/Gxx[sciSPW])
print('      <Tsky+src_YY> = ',
      (Pyy[sciSPW].mean()-PIyy[sciSPW])/Gyy[sciSPW],'+/-',
      Pyy[sciSPW].std()/Gyy[sciSPW])

#-->endfor(all sciSPWs in the list)

# Save the original gains and instrumental powers for comparison with
# the updated values after the solution.
# Note: We have to use copy.deepcopy(), the simple assignment operator
# copies just the reference!

GxxOrig=deepcopy(Gxx)
GyyOrig=deepcopy(Gyy)

PIxxOrig=deepcopy(PIxx)
PIyyOrig=deepcopy(PIyy)

#-- Dimensions of the system.

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# How many net equations do we have per block of 1 time-sample.
# Be carefull: Do not flag the data that are *not* present in the spwList!
eqBlockSize=2*numSpw-len(flaggedData)

# The data has been read-out, we may close the (local) tools.
if (ownMsmid): myMsmid.done()

```

```

if (ownMS): myMS.close()

#-- Arrays for calculations of the matrix and the RHS-vector components
#   of the linearized system that depend just on the models (ATM + quiet Sun).

# Pre-calculated frequently repeating factor: Inverse number of time samples.
nsInv=1.0/nSamples

# Number of the model spectral windows.
nAtmSPWs=myAtm.getNumSpectralWindows()
# Number of all channels in the model SPWs. They have all the same structure.
nChans=myAtm.getNumChan(0)
# Number of central band-pass (with edges trimmed) channels.
cChans=len(fChans)

# Frequencies of channels in our model SPWs (central band-pass only).
spw_freqs=np.zeros((nAtmSPWs,cChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6 # MHz to Hz

    spw_freqs[atmSpw]=(fMin+df*np.arange(nChans))[fChans]

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,cChans))
dTdW=np.zeros((nAtmSPWs,cChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,cChans))
tau_l=np.zeros((nAtmSPWs,cChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(spw_freqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
#   Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
# We used those as weights in calculation of epsilon - the normalised
# inter-iteration change of operation-point at the very end of each iteration.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin
dG_max=0.001 # Relative scale dG/G
dTinst_max=0.001 # dPinst/G in kiloKelvins

+++++++ Loop over reasonable cal/mWeights ++++++++

# Define the calWeight/mWeight iteration ranges.

```

```

wItSize=101

# In semi-log scale. Use plt.semilogx(xAxis,yAxis) for plot.
#exponents=0.05*(np.arange(wItSize)-(wItSize-1)/2)
exponents=0.05*(np.arange(wItSize))
weights=pow(10,exponents)

'''
wLow=100.0
wHigh=500.0
weights=wLow+(wHigh-wLow)/(wItSize-1)*np.arange(wItSize)
'''

#- We study influence of selected (iterated) calWeight/mWeight to:

# 1. Time-averaged PWV
iPWV=np.zeros(wItSize)

# 2. Time-averaged (solar) brightness temperatures in XX and YY pols.
iTSxx=np.zeros(wItSize)
iTSyy=np.zeros(wItSize)

# 3. Number of iterations required for convergence.
iNIT=np.zeros(wItSize)

# 4. Errors separated by 'sub-systems' of equations: science (s), ACD cal (a),
# and mean-array T_tgt (m).
iXis=np.zeros(wItSize)
iXia=np.zeros(wItSize)
iXim=np.zeros(wItSize)

# 5. Relative updates of ACD gains and P_inst. List of dictionaries as it is
# easier than np.arrays of dictionaries.
iGG=[]
iPP=[]

#-- The main weight-iteration loop.
for wi in range(wItSize):

    theWeight=float(weights[wi])

    # Which weight do we iterate: calWeight or mWeight? The 2nd remains static.
    # Select by comment/uncomment.

    #calWeight=theWeight
    mWeight=theWeight

    # Reset the starting values for iterations.
    w0=pwv0
    Tsun0xx=Tsun00
    Tsun0yy=Tsun00

    # Reset gains/P_inst back to their pure-ACD values.
    Gxx=deepcopy(GxxOrig)
    Gyy=deepcopy(GyyOrig)

```

```

PIxx=deepcopy(PIxxOrig)
PIyy=deepcopy(PIyyOrig)

# Iteration counter.
itCount=0
# Normalized inter-iteration change of operation point - its initial value.
# Anything bigger than 1.0
epsilon=10.0

#-- Solution iterations: Minimize the mean value of variations.
while ((epsilon>1.0) and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] 'natural' variables
    J_wb=np.zeros((eqBlockSize,3)) # 3 vars repeating in all spws
    # G-block matrix: Jacobian part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each SPW:Pol.
    # A-block matrix: Jacobian part for [dG,dP_inst] for ACD calibration power
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, ambient
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, hot.
    # M-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] in array-mean eqs.
    # This matrix is constant and can be defined already here.
    J_M=np.array([[0.0, nsInv, 0.0],
                  [0.0, 0.0, nsInv]])

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))
    # RHS-vector 'appendix' - amb. & hot ACD calibration powers (time-avgd).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))
    # RHS-vector 'appendix 2' - array-mean values of target T_B
    CCM=np.zeros((2,1))

    # Residual mean sky temperature - after subtracted brightness of the
    # atmosphere (+CMB) and (attenuated) source. We keep it for all sci SPWs
    # and both pols and print it out at the end of every iteration cycle
    # as a diagnostics of convergence and solution quality.

    resTxx={}
    resTyy={}

    # Mapping between spw IDs and the row number in the system of equations.
    spw2rowXX={}
    spw2rowYY={}

    #- Re-calculate ATM model with updated <PWV> and <Tsun>.

    # Set current estimation of PWV to the ATM model.
    myAtm.setUserWH2O(qa.quantity(w0,'mm'))

    #- Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
    # Calculate factors dependent only on the mmodel, not the data.

```

```

for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0,'mm'))[1]['value']
    # Limit the spectrum to the central fChans region as we have trimmed
    # the edge channels in ACD-based spectral normalized gain gXX/gYY.
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001* Tebb[fChans]

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws,'mm'))[1]['value'][fChans]
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0-dws,'mm'))[1]['value'][fChans]

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001* (TebbH- TebbL)/ (2.0* dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass* (tau_wet+tau_dry)

    # Transmissivity (spectral). Limited to fChans.
    trans[atmSPW]=np.exp(- opacity[fChans])
    # Wet opacity per 1mm PWV scaled by airmass. Limited to fChans.
    tau_1[atmSPW]=(airMass/ w0)* tau_wet[fChans]

#--> endfor(modelSPWs)

#- Loop over all SCI spws - combine model-only dependent factors with
# sciSPW-dependent data and instrumental factors and build the system.

# Count the actual number of equations - some rows may be flagged.
eqCount=0

for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

```

```

#- Separate procedure by polarizations for combination with real data.

#- Pol XX

# Full identifier of data: sciSPW:pol
sciDataID=str(sciSPW)+' :XX'

if (sciDataID not in flaggedData):

    #- Model + instrumental factors combined.

    # We do weighting of equations by inverse gain in order to compensate
    # different powers (and gains) in AC and SQLD SPWs. Consequently,
    # the equations are expressed in equivalent temperatures, which are
    # (approximatelly) the same for all SPWs.
    # Weight factor
    WF=1.0/ Gxx[sciSPW]

    # Sensitivity factor for PWV.
    SWxx=(- eta_l* eta_F* Tsun0xx* np.mean(gXX[sciSPW]* trans_M* tau_1M* SF_M)
          +(1.0- eta_F+eta_F* eta_l)
          * np.mean(gXX[sciSPW]* dTdW_M) ) * Gxx[sciSPW]

    # Sensitivity factor for the (apparent) solar-brighness temperature.
    SBxx=Gxx[sciSPW]* eta_l* eta_F* np.mean(gXX[sciSPW]* trans_M* SF_M)

    # Model T_sky as viewed by the receiver with spectral gain gXX.
    TskyXX=(eta_l* eta_F* Tsun0xx* np.mean(trans_M* SF_M* gXX[sciSPW])
            +(1.0- eta_F+eta_F* eta_l) * np.mean(TbsW0_M* gXX[sciSPW])
            +eta_F* (1.0- eta_l) * Tamb)

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the (weighted) sensitivity factors for the current SPW:pol as
    # a row in the N-block/tile for the linearized-system Jacobian matrix.
    J_wb[eqCount]=WF* np.array([SWxx, SBxx, 0.0])

    #- Build the matrix: Instrumental (G & P_inst sensitivities) block.

    # We have to prepend a proper offset for the G and P_inst vars as
    # they are independendt for each SPW and polarization.
    bOffset=2*eqCount

    # G-block scaled by the inverse gain.
    J_gp[eqCount, bOffset]=WF
    J_gp[eqCount, bOffset+1]=TskyXX* WF

    #- Build the matrix: ACD powers (amb & hot) 2-block.

    # Ambient-power sub-block. Scale between the G* / P_inst* used in
    # J_wb and J_gp, and G / P_inst used in the ACD cal equations by the
    # attenuation factor AF=G*/G=(P_sky-Po)/(P_off-Po).

    # Attenuation factor (af>1) is calculated only in SQLD and mapped.
    af=AFxx[afMap[sciSPW]]

    # A-ambient sub-block Jacobian elements weighted by inverse gain.
    J_cdA[eqCount, bOffset]=af* WF

```

```

J_cdA[eqCount,bOffset+1]=af*Tamb*WF

# A-hot-power sub-block weighetd by inverse gain.
J_cdH[eqCount,bOffset]=af*WF
J_cdH[eqCount,bOffset+1]=af*Thot*WF

#- Build the RHS vector: Target data

# RHS: Pxx(t) with their stationary (in the working point) instr.
# and ATM-based parts subtracted.
dPxx=Pxx[sciSPW]-PIxx[sciSPW]-Gxx[sciSPW]*TskyXX

# Set it as the RHS component, weighted by inverse gain.
CC[eqCount]=dPxx*WF

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset. We have it defined for all SPWs,
# however, non-zero it is only in SQLDs.
aoff=AOxx[sciSPW]

# Ambient-load power with its working-point model part subtracted.
CCA[eqCount]=WF*(PxxAmb[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Tamb
+aoff)
# Hot-load power with its working-point model part subtracted.
CCH[eqCount]=WF*(PxxHot[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Thot
+aoff)

#- Diagnostics

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTxx[sciSPW]=dPxx.mean()/Gxx[sciSPW]

# Mapping between SPW:XX and equation number.
spw2rowXX[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#- Pol YY

# Full identicator of data: sciSPW:pol
sciDataID=str(sciSPW)+' :YY'

if (sciDataID not in flaggedData):

#- Model + instrumental factors combined.

# Weight factor - see explanation at XX pol above.
WF=1.0/Gyy[sciSPW]

# Sensitivity factor for PWV (weighted by 1/G).
SWyy=(-eta_l*eta_F*Tsun0yy*np.mean(gYY[sciSPW]*trans_M*tau_1M*SF_M)
+(1.0-eta_F+eta_F*eta_l)
*np.mean(gYY[sciSPW]*dTdW_M))*Gyy[sciSPW]

# Sensitivity factor for the (apparent) solar T_B (weighted by 1/G).

```

```

SByy=Gyy[sciSPW]*eta_l*eta_F*np.mean(gYY[sciSPW]*trans_M*SF_M)

# Model T_sky as viewed by the receiver with spectral gain gYY.
TskyYY=(eta_l*eta_F*Tsun0yy*np.mean(trans_M*SF_M*gYY[sciSPW])
        +(1.0-eta_F+eta_F*eta_l)*np.mean(TbsW0_M*gYY[sciSPW])
        +eta_F*(1.0-eta_l)*Tamb)

#- Build the matrix: Natural (dW & dB sensitivities) block.

# Set the (weighted) sensitivity factors for the current SPW:pol as
# a row in the block/tile for the linearized-system Jacobian matrix.
J_wb[eqCount]=WF*np.array([SWyy,0.0,SByy])

#- Build the matrix: Instrumental (G & P_inst sensitivities) block.

# We have to prepend a proper offset for the G and P_inst vars as
# they are independent for each SPW and polarization.
bOffset=2*eqCount

J_gp[eqCount,bOffset]=WF
J_gp[eqCount,bOffset+1]=TskyYY*WF

#- Build the matrix: ACD powers 2-block.

# Attenuation factor G*/G. Mapped from the corresponding SQLD.
af=AFyy[afMap[sciSPW]]

# Ambient-power sub-block.
J_cdA[eqCount,bOffset]=af*WF
J_cdA[eqCount,bOffset+1]=af*Tamb*WF

# Hot-power sub-block.
J_cdH[eqCount,bOffset]=af*WF
J_cdH[eqCount,bOffset+1]=af*Thot*WF

#- Build the RHS vector: Target data.

# RHS: Pyy(t) with their stationary (in the working point) inst.
# and ATM-based parts subtracted.
dPyy=Pyy[sciSPW]-PIyy[sciSPW]-Gyy[sciSPW]*TskyYY

# Set it as the RHS component.
CC[eqCount]=dPyy*WF

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOyy[sciSPW]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=WF*(PyyAmb[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Tamb
               +aoff)

# Hot-load power with its working-point part subtracted.
CCH[eqCount]=WF*(PyyHot[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Thot
               +aoff)

#- Diagnostics

```

```

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTyy[sciSPW]=dPy.mean()/ Gyy[sciSPW]

# Mapping between SPW:YY and equation number.
spw2rowYY[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#--> endfor(all sci SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPW:pol channels.")
    return False

#-- Grand system assembly and inversion by parts. We use the approach from
# extMatInv() for inversion of extended-system matrix.

# Ammend the RHS vector: Array-mean target brightness temperatures.
# Here we work already with temperatures, thus no 1/G scaling applies.

# Target T_B with the model at work-point subtracted.
CCM[0]=T_tgt-Tsun0xx
CCM[1]=T_tgt-Tsun0yy

#- Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# 1. Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)          # N^T
J_K=np.matmul(J_wb_T,J_wb)         # (N^T x N)
J_K_inv=np.linalg.inv(J_K)          # (N^T x N)^{-1}

# 2. Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)          # G^T

# 3. ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
# Attention!!! NumPy's transpose() makes just another view of a matrix,
# we have to make a copy, otherwise subsequent scaling would scale the
# original J_cdA and J_cdH matrices, too!

J_cdA_T=np.transpose(J_cdA).copy()  # A^T
J_cdH_T=np.transpose(J_cdH).copy()  # H^T

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition.

```

```

J_cd=np.vstack([J_cdA,J_cdH]).copy()
J_cd_T=calWeight*np.transpose(J_cd)      # calWeight * (A | H )^T

# 4. Array-mean (M-block) related matrices. Incorporate mWeight into M^T,
# as with what we have done with calWeight for the A^T and H^T matrices.
M_T=np.array([[0.0, 0.0],
               [nsInv*mWeight, 0.0],
               [0.0,nsInv* mWeight]])      # mWeight*M^T

# Pre-calculated nSamples*mWeight*(M^T x M) matrix.
# Like with A^T x A, the M^T x M product is scaled-up by mWeight just once
# (no mWeight squared!). Moreover, we scale it up here yet by nSamples,
# in order to avoid trivial product nsInv*nSamples=1 in the following calc
nsMTM=np.array([[0.0, 0.0, 0.0],
                 [0.0, nsInv*mWeight, 0.0],
                 [0.0, 0.0, nsInv*mWeight]])

# Frequently repeating blocks. NM_inv plays a role of modified J_K_inv.
NM=J_K+nsMTM                        # J_K+nSamples*mWeight*M^T x M
NM_inv=np.linalg.inv(NM)            # NM^(-1)

# Off-diagonal blocks of the inverted natural sub-matrix (Ainv).
J_M_inv=nsInv*(NM_inv-J_K_inv)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T,J_gp)          # (N^T x G)
Lb=np.transpose(Rb)                 # (G^T x N)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T,J_gp)+np.matmul(J_cd_T,J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(NM_inv,Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc- nSamples*np.matmul(Lb,AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv,LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=J_M_inv- np.matmul(AinvRb,Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

```

```

# Lower-right block.
Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e, multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)+np.matmul(M_T,CCM) # 3 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)+nSamples*np.matmul(M_T,CCM) # 3 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum)          # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (3 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
# NB: Actually, with M-blocks included, it contains a small constant part,
# too, equal to J_K_inv x M_T x CCM, but we keep it here (hidden in
# C_K_wb).
nvVar=np.matmul(J_K_inv,C_K_wb) # 3 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Separate the natural-vars components.
dW = varNat[0]
dBxx=varNat[1]
dByy=varNat[2]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per SPW:POL.
dPG=np.transpose(np.reshape(varInst, (eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

```

```

#- Analytics: Calculate the errors for the over-determined set of eqs.

# err=f(X)-b =approx. f(X_0)+J(X_0) x (X -X_0) -b = J x dX - [b-f(X_0)]
# In our notation, dX is the [varNat,varInst]^T vector and the
# difference [b-f(X_0)] is [CC, (CCA | CCH), CCM]^T
# Now we express the err vector parts for sci, acd, and array-mean eqs.

# err=f(x)-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part. Sum over all SPW:Pol channels & times.
# Normalize it to the number of equations.
xiSci=np.sum(errVec*errVec)/(nSamples*eqBlockSize)

# Xi Squares per SPW:Pol channel. Sum/average only over times.
xiSciPerChan=np.sum(errVec*errVec,axis=1)/nSamples
# Its mean and RMS over the SPW:POL channels.
xscMean=xiSciPerChan.mean()
xscRMS=xiSciPerChan.std()
# Find outliers (> 3 sigma). The badData array contains indices of
# those equations, where the agreement between model and observation is
# exceptionally weak.
badData=np.flatnonzero(abs(xiSciPerChan-xscMean)>3.0*xscRMS)

# Show all per-channel errors as temperatures (K) and mark (auto-flag
# later) the SPW:Pol channels with bad data, if we have some.
# We do not have an inverse map to spw2rowXX/YY, so we have to traverse.

# XX pol
for spw,eqRow in spw2rowXX.items():
    print(str(spw)+':XX: Error:',np.sqrt(xiSciPerChan[eqRow])*1000.0,'K')
    if eqRow in badData: print('\t\t\t!!! Bad data in',str(spw)+':XX',
                                (xiSciPerChan[eqRow]-xscMean)/xscRMS,
                                'sigma from the mean.')

# YY pol
for spw,eqRow in spw2rowYY.items():
    print(str(spw)+':YY: Error:',np.sqrt(xiSciPerChan[eqRow])*1000.0,'K')
    if eqRow in badData: print('\t\t\t!!! Bad data in',str(spw)+':YY',
                                (xiSciPerChan[eqRow]-xscMean)/xscRMS,
                                'sigma from the mean.')

# err=f(x)-b: ACD-data part.
# Actually, this should be scaled by calWeight factor. On the other hand,
# we are interested in Xi2 normalised per one equation, so we would
# down-scale in the next calculation by the same factor. In order to avoid
# useless product calWeight*1/calWeight, we omit it already now.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the acd-data part.
# Normalize it to the number of ACD equations. Scale by just single
# block-size - we have ignored calWeight factor already above.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)
xiACD/=eqBlockSize

# err=f(x)-b: Array-mean T_B part.
# Here, the term J x dX = M x sum(CC_i) = [sum(dTxx_i), sum(dTyy_i)]^T =
# nS*[dBxx.mean(),dByy.mean()]^T is zero because dB.mmean() converged to 0
errVecMeanTb=-CCM
# Normalized to one equation - we have mWeight*2 equations, but the same

```

```

# holds for mWeight factor as for the calWeight above.
xiTbM=np.sum(errVecMeanTb*errVecMeanTb)/ 2

# Grand-total Xi-square normalized to the number of equations for
# the natural/science, ACD, and mean-array sub-blocks.
xiSquare=xiSci+xiACD+xiTbM

#- All variations should tend to zero means.

dW_mean=dW.mean()
dBxx_mean=dBxx.mean()
dByy_mean=dByy.mean()

dB_mean=np.sqrt(dBxx_mean**2+dByy_mean**2)

# Update stationary part of PWV and solar T_B by means of their vars.
w0+=dW_mean
Tsun0xx+=dBxx_mean
Tsun0yy+=dByy_mean

# Update also the P_inst and G by the found differences.
# Find the maximum inter-iteration change for convergence diagnostics.
dT_inst_sup=-1.0
dG_rel_sup=-1.0

# Start with XX pol.
for sciSPW,eqCount in spw2rowXX.items():
    # Update the P_inst and G.
    PIxx[sciSPW]+=dPinst[eqCount]
    Gxx[sciSPW]+=dGinst[eqCount]

    # Change of inst. power expressed in kiloKelvins.
    dT_inst=abs(dPinst[eqCount]/Gxx[sciSPW])
    if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

    # Change of gain in the relative scale.
    dG_rel=abs(dGinst[eqCount]/Gxx[sciSPW])
    if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(SPWs with valid XX pol)

# YY pol.
for sciSPW,eqCount in spw2rowYY.items():
    # Update the P_inst and G.
    PIyy[sciSPW]+=dPinst[eqCount]
    Gyy[sciSPW]+=dGinst[eqCount]

    # Change of inst. power expressed in kiloKelvins.
    dT_inst=abs(dPinst[eqCount]/Gyy[sciSPW])
    if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

    # Change of gain in the relative scale.
    dG_rel=abs(dGinst[eqCount]/Gyy[sciSPW])
    if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(SPWs with valid YY pol)

#- Convergence diagnostics / Iteration control.

# Calculate the normalized inter-iteration change of operation point.

```

```

# The factor 0.5 comes from sqrt(1/4) as we have 4 'orthogonal' variables.
epsilon=0.5*np.sqrt((dW_mean/ dwMean_max)**2+(dB_mean/ dbMean_max)**2+
                    (dG_rel_sup/ dG_max)**2+(dT_inst_sup/ dTinst_max)**2)

# Increment the iteration counter
itCount+=1

#- Diagnostic print-outs per-iteration.

# Updated scan-averaged PWV and Tsun after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('\nMean PWV for the scan,',tgtScan,' w0 = ',w0,
      ", mean Tsun (XX/YY) = ",Tsun0xx,'/',Tsun0yy)

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale back form kiloKelvins to Kelvins.
resTxxArr=1000.0*np.array(list(resTxx.values()))
resTyyArr=1000.0*np.array(list(resTyy.values()))

print("\nres Txx: ", resTxxArr,"\nmean +/- RMS: ",
      resTxxArr.mean(),"+/-",resTxxArr.std())
print("\nres Tyy: ", resTyyArr,"\nmean +/- RMS: ",
      resTyyArr.mean(),"+/-",resTyyArr.std())

print("\nInstrumental characteristics shifts during the iteration:")
print("--- Pol XX--- ")
for sciSPW,eqCount in spw2rowXX.items():
    print("SPW =",sciSPW,"\tdT_inst =",dPinst[eqCount]/ Gxx[sciSPW]*1000.0,
          "K","\tdG/G =",dGinst[eqCount]/ Gxx[sciSPW]*100,"%")

print("--- Pol YY--- ")
for sciSPW,eqCount in spw2rowYY.items():
    print("SPW =",sciSPW,"\tdT_inst =",dPinst[eqCount]/ Gyy[sciSPW]*1000.0,
          "K","\tdG/G =",dGinst[eqCount]/ Gyy[sciSPW]*100,"%")

print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / end of the main solution-iteration loop.

#- Show, how much the instrumental variables G and P_inst have changed.

# The instrumental characteristics are maps, cannot process them as arrays.

GxxChange={}
GyyChange={}

PIxxChange={}
PIyyChange={}

for spw in spw2rowXX.keys():

    GxxChange[spw]=(Gxx[spw]-GxxOrig[spw])/GxxOrig[spw]

```

```

    PIxxChange[spw]=(PIxx[spw]-PIxxOrig[spw])/PIxxOrig[spw]

    for spw in spw2rowYY.keys():

        GyyChange[spw]=(Gyy[spw]-GyyOrig[spw])/GyyOrig[spw]
        PIyyChange[spw]=(PIyy[spw]-PIyyOrig[spw])/PIyyOrig[spw]

# Store the watched quantities in the array per-weight elements.

    iPWV[wi]=w0

# Scale the result back to Kelvins.
    iTSxx[wi]=1000.0*Tsun0xx
    iTSyy[wi]=1000.0*Tsun0yy

    iNIT[wi]=itCount
    iXis[wi]=xiSci
    iXia[wi]=xiACD
    iXim[wi]=xiTbM

    iGG.append([GxxChange,GyyChange])
    iPP.append([PIxxChange,PIyyChange])

#--> endfor(range-of-weights)

#-- Return the arrays of interest in dependence on calWeight.

    return {'NS':nSamples,'CW':weights,"PWV":iPWV,"Tx":iTSxx,"Ty":iTSyy,
            "XiS":iXis,"XiA":iXia,"XiM":iXim,"NIT":iNIT,"dG":iGG,"dP":iPP}

#-----

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

    '''
    --- OBSOLETE---

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

    #-- Static parameters

    #- Dirs

    # Output/diagnostics plots.
    plotDir='./sciPWVs_GP'

    #- Tools, global data

    # Flags used to check, whether we should close the tool later.
    ownMS=False
    ownMsmd=False

```

```

# Open own tools if 'global' not provided
if (not myMS) :
    # Open the ms tool
    myMS=mstool()
    myMS.open(vis)
    ownMS=True

if (not myMsmd) :
    # Open the ms tool.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    ownMsmd=True

# Create our dictionary with ACD load temperatures (ambient, hot) if needed.
if (not acdLoads) :
    acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                     warnIfNoLoadTemperatures = True,
                                     mymsmd=myMsmd)

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.3 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temoerature corrected to limb brightening - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu)

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

```

```

#- SPWs

# Integration limits for resolved AC SPWs.
startChan=8
stopChan=120

fChans=range(startChan, stopChan)

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
eta_F=0.90

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# Weight of the ACD calibration equations in the system. Essentially the
# number, with which appended block of the ACD equations is repeated in order
# to balance the much larger sciData-equations block. Infinity means that
# the instrumental parameters G and P_inst are essentially fixed and given
# by results of the acd_calTabs(). Low weight, on the other hand, leads to
# an ill-coinditioned system. We set in in order of nTimeSmamples for science
# data so that the target and calibration scan are balanced.
calWeight=200.0

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmid)['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,' @airMass = ',airMass)
print('Estimated T_B = ',Tsun00)
print('.....')

#- SciSPW to AtmSPW mapping.

# How science SPWs map to the SPW 0 - 3 in the ATM tool.
sciAtmMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

#--- Read-out the sky+src powers for all SPWs in the spwList.

# Store the frequency-averaged sky+src powers in a map.
# spwld -> np.array(nTimeSamples). Do not use 2D array as we index by the SPW
# id, which is sparse in the set of integers.
Pxx={}

```

```

Pyy={}

#- Similar maps for the gains and instrumental powers P_inst=G*T_inst.
# Here, all gains and P_inst are, in fact, the G* and P_inst* values for the
# higher-attenuation solar regime. In order to append the ACD calibration
# equations, whose measurements are performed in the normal-attenuation regime
# in the preceding ATM scan, we determine also the attenuation factors
# AF=G*/G and related power offsets AO.

# Frequency-averaged gains. The scale for the spectral-resolved norm. gains.
Gxx={}
Gyy={}

# Attenuation factors G/G*=(P_sky-P0)/(P_off-P0). Attributed to every-spw
# Gxx & Gyy, but actually calculated just in SQLD spws, the rest is mapped.
AFxx={}
AFyy={}

afMap={0:0,1:1,2:2,3:3,5:0,6:0,7:1,8:1,9:2,10:2,11:3,12:3}

# Attenuation-based power offsets AO=-Po*(P_sky-P_off)/(P_off-P0).
# The same holds as for the AFs - we use afMap.
AOxx={}
AOyy={}

# Spectral gain normalized to 1.0. Actual ACD-based values in spectral AC spws,
# a trivial 1.0 ndarray for integrated-AC and SQLD spws.
gXX={}
gYY={}

# Frequency-averaged instrumental powers.
PIxx={}
PIyy={}

#- Finally, the raw ACD-calibration powers.

# Ambient-load powers.
PxxAmb={}
PyyAmb={}

# Hot-load powers.
PxxHot={}
PyyHot={}

#-- Read the data and ACD calibration tables in a loop over all sciSPWs.

for sciSPW in spwList:

    #- Raw data & ACD tables

    # Read raw science powers for the target defined by targetScan.
    time,P_src=tgt_pow(vis,tgtScan,antID,sciSPW,myMS=myMS,myMsmid=myMsmid,
                       subScans='on')

    PxxS=P_src[0]
    PyyS=P_src[1]

    # Read ACD calibrations - gains,Tinst, and calibration powers.

```

```

calDict=acd_calTabs(vis,atmScan,antID,sciSPW,myMS=myMS,myMsmc=myMsmc,
                    acdLoads=acdLoads)

Pcals=calDict['calPows']

PxxAmbS=Pcals['P_amb'][0]
PyyAmbS=Pcals['P_amb'][1]

PxxHotS=Pcals['P_hot'][0]
PyyHotS=Pcals['P_hot'][1]

# We have different handling for frequency-resolved and integrated SPWs.
nChans=PxxS.shape[0]

if (nChans > 1):

    #- Resolved AC SPWs.
    # Trim the spectral edges by posing the integration limits and average
    # over the remaining fChans channels.

    # Science-target powers.
    PxxM=np.mean(PxxS[fChans],axis=0)
    PyyM=np.mean(PyyS[fChans],axis=0)

    # Calibration (ACD) powers.
    PxxAmbM=np.mean(PxxAmbS[fChans],axis=0)
    PyyAmbM=np.mean(PyyAmbS[fChans],axis=0)

    PxxHotM=np.mean(PxxHotS[fChans],axis=0)
    PyyHotM=np.mean(PyyHotS[fChans],axis=0)

    #- Instrumental factors.
    # Store the normalized (to 1.0) spectral gains (arrays, constant) and
    # their scales (numbers, to be re-fined by the solver) separately.

    # Raw G* gains in a full SPW. We may flip the array (see below) so better
    # to do so before its limiting to fChans - we need not to think whether
    # the fChans central range is placed symmetrically within its SPW.
    # The 1st index 0 means we pick the leading P_off subscan - see the note
    # in acd_calTabs2().
    GxxS=1.0/(calDict['iG*'][0,0])
    GyyS=1.0/(calDict['iG*'][0,1])

    # T_inst in full SPW. The same holds as for G*.
    TinstXX=calDict['T_inst'][0]
    TinstYY=calDict['T_inst'][1]

    # Check, whether we need flip them - this is the case of the LSB spws,
    # which have decreasing frequency channels.
    spwFreqs=calDict['freqs']

    if (spwFreqs[0] > spwFreqs[1]):
        # Reversed SPW, flip the gain and T_inst arrays.

        GxxS=np.flip(GxxS)

```

```

    GyyS=np.flip(GyyS)

    TinstXX=np.flip(TinstXX)
    TinstYY=np.flip(TinstYY)

# Now it is safe to select the central fChans band-pass.
    GxxS=GxxS[fChans]
    GyyS=GyyS[fChans]

    TinstXX=TinstXX[fChans]
    TinstYY=TinstYY[fChans]

# Calculate the unity-normalized gains by dividing out the mean value.
    GxxM=GxxS.mean()
    GyyM=GyyS.mean()

    gXX[sciSPW]=GxxS/ GxxM
    gYY[sciSPW]=GyyS/ GyyM

# Instrumental power averaged over our fChans range.
    PIxxM=np.mean(GxxS*TinstXX)
    PIyyM=np.mean(GyyS*TinstYY)

else :
    #- Integrated ACs or SQLDs.
    # The single channel already represents a mean over the band-pass.

    # Science data.
    PxxM=PxxS[0]
    PyyM=Pyys[0]

    # The SQLD spws have 12x higher cadence than the ACs. Rebin the SQLD
    # data in time to have a homogeneous series. Do the same for time axis.
    # Moreover, extract the attenuation factor (P_sky-Po)/(P_off-Po) and
    # offsets AO from the SQLD spws (ACs use mapping afMap defined above).
    if sciSPW in [0,1,2,3,13,14,15,16]:
        # Re-bin in time.
        PxxM=rebin(PxxM,12)
        PyyM=rebin(PyyM,12)
        time=rebin(time,12)

        # Extract the caliibration powers relevant for AF calculation
        Patm=Pcals['P_sky']
        Poff=Pcals['P_off']
        Pzero=Pcals['P_zero']

        # Calculate AFs.
        # The 1st index in powers is pol, the 2nd means (the single) channel.
        AFxx[sciSPW]=(Patm[0,0]-Pzero[0,0])/(Poff[0,0]-Pzero[0,0])
        AFyy[sciSPW]=(Patm[1,0]-Pzero[1,0])/(Poff[1,0]-Pzero[1,0])

        # Calculates AOs.
        AOxx[sciSPW]=Pzero[0,0]*(AFxx[sciSPW]-1.0)
        AOyy[sciSPW]=Pzero[1,0]*(AFyy[sciSPW]-1.0)

    #--> endfor(SQLD spws)

```

```

# Calibration data.
PxxAmbM=PxxAmbS[0]
PyyAmbM=PyyAmbS[0]

PxxHotM=PxxHotS[0]
PyyHotM=PyyHotS[0]

#- Instrumental factors.
GxxM=1.0/ (calDict['iG*'][0,0,0])
GyyM=1.0/ (calDict['iG*'][0,1,0])

# Create a fake spectral normalized gain for integrated SPWs.
# We use it in products with the ATM arrays (e.g. transmissivity), which
# are always spectral.
gXX[sciSPW]=np.ones(len(fChans))
gYY[sciSPW]=np.ones(len(fChans))

# Instrumental power: The channel 0 is the average already
PIxxM=GxxM*calDict['T_inst'][0,0]
PIyyM=GyyM*calDict['T_inst'][1,0]
#--> endif(resolved SPW)

# Append the frequency-averaged time series for each SPW to the maps.
Pxx[sciSPW]=PxxM
Pyy[sciSPW]=PyyM

# The calibration powers are already time-averaged.
PxxAmb[sciSPW]=PxxAmbM
PyyAmb[sciSPW]=PyyAmbM

PxxHot[sciSPW]=PxxHotM
PyyHot[sciSPW]=PyyHotM

# Set the ACD-based gains (scales) and instrumental contributions.
Gxx[sciSPW]=GxxM
Gyy[sciSPW]=GyyM

PIxx[sciSPW]=PIxxM
PIyy[sciSPW]=PIyyM

print('SPW ',sciSPW,': <Tsky+src_XX> = ',
      (Pxx[sciSPW].mean()-PIxx[sciSPW])/Gxx[sciSPW],'+/-',
      Pxx[sciSPW].std()/Gxx[sciSPW])
print('      <Tsky+src_YY> = ',
      (Pyy[sciSPW].mean()-PIyy[sciSPW])/Gyy[sciSPW],'+/-',
      Pyy[sciSPW].std()/Gyy[sciSPW])

#-->endfor(all sciSPWs in the list)

# Save the original gains and instrumental powers for comparison with
# the updated values after the solution.
# Note: We have to use copy.deepcopy(), the simple assignment operator

```

```

# copies just the reference!

GxxOrig=deepcopy(Gxx)
GyyOrig=deepcopy(Gyy)

PIxxOrig=deepcopy(PIxx)
PIyyOrig=deepcopy(PIyy)

#-- Dimensions of the system.

# Count the SPW in the list.
numSpw=len(spwList)
# Count the time samples in data.
nSamples=len(time)

# How many net equations do we have per block of 1 time-sample.
# Be carefull: Do not flag the data that are *not* present in the spwList!
eqBlockSize=2*numSpw-len(flaggedData)

# The data has been read-out, we may close the (local) tools.
if (ownMsmd): myMsmd.done()
if (ownMS): myMS.close()

#-- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).

# Number of the model spectral windows.
nAtmSPWs=myAtm.getNumSpectralWindows()
# Number of all channels in the model SPWs.
nChans=myAtm.getNumChan(0)
# Number of central band-pass (with edges trimmed) channels.
cChans=len(fChans)

# Frequencies of channels in our model SPWs (central band-pass only).
spw_freqs=np.zeros((nAtmSPWs,cChans))

for atmSpw in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(atmSpw) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(atmSpw) ['value'] [0]*1.0e6 # MHz to Hz

    spw_freqs[atmSpw]=(fMin+df*np.arange(nChans))[fChans]

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,cChans))
dTdW=np.zeros((nAtmSPWs,cChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,cChans))
tau_1=np.zeros((nAtmSPWs,cChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(spw_freqs/ f_LO)

```

```

#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin

#+++++ Loop over reasonable calWeights - experimental ++++++

PWV=np.zeros(900)

TSxx=np.zeros(900)
TSyy=np.zeros(900)

NIT=np.zeros(900)
xis=np.zeros(900)
xia=np.zeros(900)

for cwi in range(900):

    # Set calWeight.
    calWeight=100.0+cwi

    # Reset the starting values for iterations.
    w0=pwv0
    Tsun0xx=Tsun00
    Tsun0yy=Tsun00

    Gxx=deepcopy(GxxOrig)
    Gyy=deepcopy(GyyOrig)

    PIdx=deepcopy(PIdxOrig)
    PIyy=deepcopy(PIyyOrig)

    # Initial values for the dW and dL means - anything bigger then the stoppers.
    dW_mean=0.5
    dB_mean=0.5

    # Iteration counter.
    itCount=0

    #-- Iterations: Minimize the mean value of variations.

    while ((abs(dW_mean) > dwMean_max) or (abs(dB_mean) > dbMean_max))

```

```

    and (itCount< 20)):

    ## Prepare/reset the linearized system of equations.

    ## Repeating blocks/tiles for each time-sample

    # N-block matrix: Jacobian part for [dW,dT_Bxx,dT_Byy] 'natural' variables
    J_wb=np.zeros((eqBlockSize,3)) # 3 vars repeating in all spws
    # G-block matrix: Jacobian part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each SPW:Pol.
    # A-block matrix: Jacobian part for [dG,dP_inst] for ACD powers.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, ambient
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per SPW:Pol, hot.

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))
    # RHS-vector 'appendix' - amb. & ACD calibration powers (time-averaged).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

    resTxx={}
    resTyy={}

    spw2rowXX={}
    spw2rowYY={}

    ## Re-calculate ATM model with updated <PWV> and <Tsun>.
    ## Set current estimation of PWV to the ATM model.
    myAtm.setUserWH20(qa.quantity(w0,'mm'))

    ## Loop over 4 ATM spws - these correspond to the channel-resolved ACs.
    ## Calculate factors dependent only on the mmodel, not the data.
    for atmSPW in range(nAtmSPWs):

        ## Model value of the sky/atm brightness with current pwv.
        Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0,'mm'))[1]['value']
        ## Limit the spectrum to the central fChans region as we have trimmed
        ## the edge channels in ACD-based spectral normalized gain gXX/gYY.
        ## Scale to kiloKelvins!
        TbsW0[atmSPW]=0.001*Tebb[fChans]

        ## Slope (centered difference) of the atm brightness with current pwv.
        TebbH=myAtm.getTebbSkySpec(atmSPW,
                                   qa.quantity(w0+dws,'mm'))[1]['value'][fChans]
        TebbL=myAtm.getTebbSkySpec(atmSPW,
                                   qa.quantity(w0-dws,'mm'))[1]['value'][fChans]

        ## In kiloKelvins/mm!
        dTdW[atmSPW]=0.001*(TebbH- TebbL)/(2.0*dws)

        ## Opacities & transmissivity.

        ## Dry zenith opacity
        tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
        ## Water-vapour related 'wet' opacity
        tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
        ## Total opacity (spectral).

```

```

opacity=airMass* (tau_wet+tau_dry)

# Transmissivity (spectral). Limited to fChans.
trans[atmSPW]=np.exp(- opacity[fChans])
# Wet opacity per 1mm PWV scaled by airmass. Limited to fChans.
tau_1[atmSPW]=(airMass/ w0)* tau_wet[fChans]

#--> endfor(modelSPWs)

#- Loop over all SCI spws - combine model-only dependent factors with
# sciSPW-dependent data and instrumental factors and build the system.

# Count the actual number of equations - some rows may be flagged.
eqCount=0

for sciSPW in spwList:

    # Map the sci SPW to its corresponding ATM counterpart
    atmSPW=sciAtmMap[sciSPW]

    #- Model factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Separate procedure by polarizations for combination with real data.

    #- Pol XX

    # Full identifier of data: sciSPW:pol
    sciDataID=str(sciSPW)+' :XX'

    if (sciDataID not in flaggedData):

        #- Model + instrumental factors combined.

        # Sensitivity factor for PWV.
        SWxx=(- eta_1* eta_F* Tsun0xx* np.mean(gXX[sciSPW]* trans_M* tau_1M* SF_M)
              +(1.0- eta_F+eta_1)* np.mean(gXX[sciSPW]* dTdW_M))* Gxx[sciSPW]

        # Sensitivity factor for the (apparent) solar-brighthness temperature.
        SBxx=Gxx[sciSPW]* eta_1* eta_F* np.mean(gXX[sciSPW]* trans_M* SF_M)

        # Model T_sky as viewed by the receiver with spectral gain gXX.
        TskyXX=(eta_1* eta_F* Tsun0xx* np.mean(trans_M* SF_M* gXX[sciSPW])
                +(1.0- eta_F+eta_1)* np.mean(TbsW0_M* gXX[sciSPW])
                +(1.0- eta_1)* Tamb)

```

```

#- Build the matrix: Natural (dW & dB sensitivities) block.

# Set the sensitivity factors for the current SPW and pol as a row
# in the block/tile for the linearized-system Jacobian matrix.
J_wb[eqCount]=np.array([SWxx,SBxx,0.0])

#- Build the matrix: Instrumental (G & P_inst sensitivities) block.

# We have to prepend a proper offset for the G and P_inst vars as
# they are independent for each SPW and polarization.
bOffset=2*eqCount

J_gp[eqCount,bOffset]=1.0
J_gp[eqCount,bOffset+1]=TskyXX

#- Build the matrix: ACD powers (amb & hot) 2-block.

af=AFxx[afMap[sciSPW]]

J_cdA[eqCount,bOffset]=af
J_cdA[eqCount,bOffset+1]=af*Tamb

# Hot-power sub-block.
J_cdH[eqCount,bOffset]=af
J_cdH[eqCount,bOffset+1]=af*Thot

#- Build the RHS vector: Target data

# RHS: Pxx(t) with their stationary (in the working point) instrumen
# and ATM-based parts subtracted.
dPxx=Pxx[sciSPW]-PIxx[sciSPW]-Gxx[sciSPW]*TskyXX

# Set it as the RHS component.
CC[eqCount]=dPxx

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOxx[afMap[sciSPW]]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=PxxAmb[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Tamb+aoff
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=PxxHot[sciSPW]-af*PIxx[sciSPW]-af*Gxx[sciSPW]*Thot+aoff

#- Diagnostics

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTxx[sciSPW]=dPxx.mean()/Gxx[sciSPW]

# Mapping between SPW:XX and equation number.
spw2rowXX[sciSPW]=eqCount

eqCount+=1

```

```

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#- Pol YY

# Full identifier of data: sciSPW:pol
sciDataID=str(sciSPW)+' :YY'

if (sciDataID not in flaggedData):

    #- Model + instrumental factors combined.

    # Sensitivity factor for PWV.
    SWyy=(- eta_l* eta_F* Tsun0yy* np.mean(gYY[sciSPW]* trans_M* tau_lM* SF_M)
          +(1.0- eta_F+eta_l)* np.mean(gYY[sciSPW]* dTdW_M) ) * Gyy[sciSPW]

    # Sensitivity factor for the (apparent) solar-brighthness temperature.
    SByy=Gyy[sciSPW]* eta_l* eta_F* np.mean(gYY[sciSPW]* trans_M* SF_M)

    # Model T_sky as viewed by the receiver with spectral gain gYY.
    TskyYY=(eta_l* eta_F* Tsun0yy* np.mean(trans_M* SF_M* gYY[sciSPW])
            +(1.0- eta_F+eta_l)* np.mean(TbsW0_M* gYY[sciSPW])
            +(1.0- eta_l)* Tamb)

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the sensitivity factors for the current SPW and pol as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    J_wb[eqCount]=np.array([SWyy,0.0,SByy])

    #- Build the matrix: Instrumental (G & P_inst sensitivities) block.

    # We have to prepend a proper offset for the G and P_inst vars as
    # they are independnt for each SPW and polarization.
    bOffset=2* eqCount

    J_gp[eqCount,bOffset]=1.0
    J_gp[eqCount,bOffset+1]=TskyYY

    #- Build the matrix: ACD powers 2-block.

    # Attenuation factor G*/G.
    af=AFyy[afMap[sciSPW]]

    # Ambient-power sub-block.
    J_cdA[eqCount,bOffset]=af
    J_cdA[eqCount,bOffset+1]=af* Tamb

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=af
    J_cdH[eqCount,bOffset+1]=af* Thot

    #- Build the RHS vector: Target data.

```

```

# RHS: Pyy(t) with their sonary (in the working point) instrumental
# and ATM-based parts subtracted.
dPyy=Pyy[sciSPW]-PIyy[sciSPW]-Gyy[sciSPW]*TskyYY

# Set it as the RHS component.
CC[eqCount]=dPyy

#- Ammend the RHS vector: Calibration data.

# Attenuation-based power offset.
aoff=AOyy[afMap[sciSPW]]

# Ambient-load power with its working-point part subtracted.
CCA[eqCount]=PyyAmb[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Tamb+aoff
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=PyyHot[sciSPW]-af*PIyy[sciSPW]-af*Gyy[sciSPW]*Thot+aoff

#- Diagnostics

# Residua of sky tempereture for mean PWV and Tsun (operation point).
resTyy[sciSPW]=dPyy.mean()/Gyy[sciSPW]

# Mapping between SPW:YY and equation number.
spw2rowYY[sciSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',sciDataID,' data.')
#--> endif(not flagged)

#--> endfor(all sci SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPWs/polarization chans.")
    return False

#-- Grand system assembly and inversion by parts. We use the approach from
# extMatInv() for inversion of extended-system matrix. The procedure
# performed here looks mysterious (sometimes a factor nSamples emergeor
# the RHS vector summed over time-samples), but all directly implies fro
# application of the inverted matrix of extended grand-system (preserved
# in the commented section below) to the RHS vector. We use advantage of
# the symmetry of the original grand-system matrix (JJ_K^T=JJ_K).
# Advantage: This block-decomposed solution is O(ns^1) in the number of
# time-samples, while the 'brutal-force' direct method in the commented
# section below scales as O(ns^3) and takes ages...

#- Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# Natural (N-block) matrices.

```

```

J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
J_cdA_T=np.transpose(J_cdA).copy()
J_cdH_T=np.transpose(J_cdH).copy()

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!

J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T,J_cdH_T]) from aly scaled-up transposed sub-blocks.
J_cd=np.vstack([J_cdA,J_cdH])
J_cd_T=calWeight*np.transpose(J_cd)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T,J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T,J_gp)+np.matmul(J_cd_T,J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(J_K_inv,Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc-nSamples*np.matmul(Lb,AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv,LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=-np.matmul(AinvRb,Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

```

```

##- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e, multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)          # 3 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)    # 3 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum)        # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

##- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (3 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv,C_K_wb)      # 3 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

##- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
dW = varNat[0]
dBxx=varNat[1]
dByy=varNat[2]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per SPW:POL.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

##- Analytics: Calculate the errors for the over-determined set of eqs.

# err=A.x-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part.
xiSci=np.sum(errVec*errVec)

```

```

# err=A.x-b: ACD-data part.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the acd-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)

# Grand-total Xi-square normalized to the number of equations and relative
# Tsky+src/Thot ratio(= approx. 3500K/350K=10)
xiSquare=xiSci/ nSamples+xiACD/ calWeight*10.0

#- All variations should tend to zero means.

dW_mean=dW.mean()
dBxx_mean=dBxx.mean()
dByy_mean=dByy.mean()

dB_mean=np.sqrt(dBxx_mean**2+dByy_mean**2)

# Upde stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0xx+=dBxx_mean
Tsun0yy+=dByy_mean

# Update also the P_inst and G by the found differences.
# Start with XX pol.
for sciSPW,eqCount in spw2rowXX.items():
    # Update the P_inst and G.
    PIxx[sciSPW]+=dPinst[eqCount]
    Gxx[sciSPW]+=dGinst[eqCount]
#--> endfor(SPWs with valid XX pol)

# YY pol.
for sciSPW,eqCount in spw2rowYY.items():
    # Update the P_inst and G.
    PIyy[sciSPW]+=dPinst[eqCount]
    Gyy[sciSPW]+=dGinst[eqCount]
#--> endfor(SPWs with valid YY pol)

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and Tsun after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('\nMean PWV for the scan,',tgtScan,' w0 = ',w0,
      ', mean Tsun (XX/YY) = ',Tsun0xx,'/',Tsun0yy)

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale back from kiloKelvins to Kelvins.
resTxxArr=1000.0*np.array(list(resTxx.values()))
resTyyArr=1000.0*np.array(list(resTyy.values()))

```

```

    print("\nres Txx: ", resTxxArr, "\nmean +/- RMS: ",
          resTxxArr.mean(), "+/-", resTxxArr.std())
    print("\nres Tyy: ", resTyyArr, "\nmean +/- RMS: ",
          resTyyArr.mean(), "+/-", resTyyArr.std())

    print("\nInstrumental characteristics shift [dP_inst, dG]:", dPinst, dGinst)

    print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / end of main iteration loop.

    PWV[cwi]=w0

    TSxx[cwi]=Tsun0xx
    TSyy[cwi]=Tsun0yy

    NIT[cwi]=itCount
    xis[cwi]=xiSci
    xia[cwi]=xiACD

#--> endfor(range-of-calWeights)

# Range of calWeights
    CWS=100.0+np.arange(900.0)

#-- Return the arrays of interest in dependence on calWeight.

    return {'CW':CWS, "PWV":PWV, "Tx":TSxx, "Ty":TSyy, "XiS":xis, "XiA":xia, "NIT":NIT}

#-----

def 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().
    """

    #-- Create the atmosphere model that covers the frequency domain of all SPWs.
    # It is common for all antennas - we vary the PWV per antenna inside dWdB().
    atmModel=setAtmosSciSPWs(vis, tgtScan)

    #-- Get antenna list.
    myMsmd=msmdtool()
    myMsmd.open(vis)
    antIDs=myMsmd.antennaid()
    antNames=myMsmd.antennanames()
    myMsmd.done()

    #-- Loop over all antennas.
    for myAnt in antIDs:

        # Report progress.

        print('\n=====')
```

```

print('Calculating antenna ',int(myAnt),'<-->',antNames[myAnt])
print('=====\\n')

dwdb=dWdB(vis,atmScan,tgtScan,atmModel,int(myAnt),spwList)

#- For using binary np.save() we need everything as an np.ndarray.

# Convert the antenna ID to a single-element array.
antIDa=np.array([int(myAnt)])

# Extract the arrays from the map.
tt=dwdb['time']
dW=dwdb['dW']
dB=dwdb['dB']
AA=dwdb['A']
CC=dwdb['C']

# Open our NPY-binary file in the append mode and save the arrays for
# the current antenna.
with open('./dWdB_DATA.npy', 'ab') as f:
    np.save(f,antIDa)
    np.save(f,tt)
    np.save(f,dW)
    np.save(f,dB)
    np.save(f,AA)
    np.save(f,CC)
# --> end(save block)

# --> endfor(all antennas)

return

#-----

def 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().
    """

    #-- Global tools & data

    myMS=mstool()
    myMS.open(vis)

    myMsmd=msmdtool()
    myMsmd.open(vis)

    acdLoads=aU.getLoadTemperatures(vis,doplot = False,
                                     warnIfNoLoadTemperatures = True,
                                     mymsmd=myMsmd)

```

```

#-- Atmosphere model that covers the frequency domain of all SPWs.

myAtm=attool()
setAtmTool4SPWs(myAtm,vis,tgtScan)

#-- Get antenna list.
antIDs=myMsmid.annennaidids()
antNames=myMsmid.annennanames()

#-- Redirect output to a text file.

sys.stdout=open("./1_STDOUT.txt", "w")

#-- Loop over all antennas.
for myAnt in antIDs:

    # Report progress.

    print('\n=====')
    print('Calculating antenna ',int(myAnt),'<-->',antNames[myAnt])
    print('=====')

    # Select by (un)commenting whether dWdB_Tbs() or dWdBdG() should be used
    # for actual per-antenna calculations.

    '''
dwdB=dWdB_Tbs(vis,atmScan,tgtScan,myAtm,int(myAnt),spwList,flaggedData=[],
              myMS=myMS,myMsmid=myMsmid,acdLoads=acdLoads)
    '''

    dwdB=dWdBdG(vis,atmScan,tgtScan,myAtm,int(myAnt),spwList,flaggedData=[],
                myMS=myMS,myMsmid=myMsmid,acdLoads=acdLoads)

    # For utilisation of binary np.save() we need everything as an np.ndarray.
    # Convert the antenna ID to a single-element array.
    antIDa=np.array([int(myAnt)])

    # Extract the arrays from the map.
    tt=dwdB['time']
    dW=dwdB['pwv']
    dB=dwdB['T_B']
    AA=dwdB['J']
    CC=dwdB['C']

    # Open our NPY-binary file in the append mode and save the arrays for
    # the current antenna.
    with open('./PWV_TB_GP_DATA.npy', 'ab') as f:
        np.save(f,antIDa)
        np.save(f,tt)
        np.save(f,dW)
        np.save(f,dB)
        np.save(f,AA)

```

```

    np.save(f, CC)
    # --> end(save block)

    # Flush the redirected stdout stream.
    sys.stdout.flush()
    # --> endfor(all antennas)

    #-- Restore & clean-up.

    # Restore stdout back to console.
    sys.stdout = sys.__stdout__

    # Close the tools.
    myAtm.done()
    myMsmc.done()
    myMS.close()

    return True

#-----

def 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().
    """

    #-- List all antennas in the array.
    myMsmc=msmctool()
    myMsmc.open(vis)
    antIDs=myMsmc.antennas()
    myMsmc.done()

    # So far empty map [antID -> dWdB_record].
    dWdB_map={}

    # Init the record counter.
    rCount=0

    # Open the binary dWdB data.
    with open(filePath, 'rb') as f:

```

```

#-- Loop over the stack of records.
while (rCount<runID):

    #-- Loop over all antennas.
    for myAnt in antIDs:

        # We may be aiming at the last record (the default) - check the EOF.
        # The np.load() method needs to be checked via exceptions.
        try:

            antIDa=np.load(f)

        except (EOFError,ValueError, OSError): # EOF: new/old NumPy style
            print("EOF reached beyond record no.",rCount,".\n")
            # Prevent further reading attempts.
            rCount=runID
            # Break the antenna for-loop.
            break

        #--> end try/catch block.

        # The antID has been encoded as a single-element np array in order
        # to use save()/load() infrastructure.
        antID=antIDa[0]

        # Load the arrays.
        tt=np.load(f)
        dW=np.load(f)
        dB=np.load(f)
        AA=np.load(f)
        CC=np.load(f)

        # Organise them into the record (a map) as they were returned by dWdB().
        dWdB_record={}

        dWdB_record['time']=tt
        dWdB_record['dW']=dW
        dWdB_record['dB']=dB
        dWdB_record['A']=AA
        dWdB_record['C']=CC

        # Append the record to the collection.
        dWdB_map[antID]=dWdB_record

    #--> endfor(all antennas)

    # Increment the record counter.
    rCount+=1

    #-->endwhile(recordCountReached)
#--> end with-block

# Return the entire collection.
return dWdB_map

```

```

#-----

```

```

#                               5.1 Consistency checks of the dPWV proxies
#-----

def 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.
    """

    # Restore all-antenna records, get the size of the collection.
    dwdbMap=restoreSol_dWdB()
    nAnts=len(dwdbMap)

    # Extract time-axis from the 1st antenna - it should be the same for all.
    tt=dwdbMap[0]['time']

    # Summed arrays of dB for XX and YY. Init by the 1st antenna.
    dBxx=dwdbMap[0]['dB'][0]
    dByy=dwdbMap[0]['dB'][1]

    for ant in range(1,nAnts):

        dB=dwdbMap[ant]['dB']

        dBxx+=dB[0]
        dByy+=dB[1]

    #--> endfor(antennas)

    # Do the average - scale the sum by the count of antennas.
    dBxx/=nAnts
    dByy/=nAnts

    plt.plot(tt,dBxx,'b-')
    plt.plot(tt,dByy,'r-')

    return

#-----

def 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.

"""

Number fo records (num. of antennas)

nAnts=len(resCG.keys())

Allocate the arrays.

Tbxx_c=np.zeros(nAnts)

Tbyy_c=np.zeros(nAnts)

Tbxx_v=np.zeros(nAnts)

Tbyy_v=np.zeros(nAnts)

ind=0

Loop over all antennas.

for antID in resCG.keys():

Extract the data. Make the time averaging.

Tbxx_c[ind]=resCG[antID]['dB'][0].mean()

Tbyy_c[ind]=resCG[antID]['dB'][1].mean()

Tbxx_v[ind]=resVG[antID]['dB'][0].mean()

Tbyy_v[ind]=resVG[antID]['dB'][1].mean()

ind+=1

#--> endfor(all antennas)

Make and print the stats.

print("\n----- Constant gains:\n")

print("T_Bxx mean = ",Tbxx_c.mean()," , STD = ",Tbxx_c.std())

print("T_Byy mean = ",Tbyy_c.mean()," , STD = ",Tbyy_c.std())

print("\n----- Variable gains:\n")

print("T_Bxx mean = ",Tbxx_v.mean()," , STD = ",Tbxx_v.std())

print("T_Byy mean = ",Tbyy_v.mean()," , STD = ",Tbyy_v.std())

return {"C": [Tbxx_c,Tbyy_c], "V": [Tbxx_v,Tbyy_v]}

#-----
6. Analysis: Compare the dPWV proxies vs. phase variations
#-----

def dpwvXphi_plots(doSun=False,useSQLD=False):

"""

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

Plot the phase variations opverplotted 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*.*

"""

```

#-- 'Static' settings.

# Our input MS.
vis='uid__A002_X110fd13_X1099e.ms'

# Find the antenna list (the IDs and corresponding names)
msmd.open(vis)
antIDs=myMsmd.antennaid()
antNames=myMsmd.antennaname()
msmd.done()

#-- Do we analyse the BP calibrator or the Sun? It controls the atm/tgtScan.
if (doSun):

    # The Sun's target-scan and its preceding ACD-calibration ATM scan.
    atmScan=8
    tgtScan=9

    srcName='sun'
    # Self-calibrated solar phase solution
    gTable='../calibrated/lowres-sc.phase_int'

else:

    # BP calibrator: The target and ACD-calibration scans.
    atmScan=4
    tgtScan=5

    srcName='BPcal'
    # Band-pass calibrator phase gaintable - raw, i.e. without WVR correction.
    gTable=vis+'.ap_pre_bandpass'

#-- Do we use ACs or SQLD for dPWV calculations?

if (useSQLD):
    # SPW type - part of the output-plots directory name.
    spwType='sqld'

else: # ACs
    # SPW type - part of the output-plots directory name.
    spwType='ac'

#-- Directory, where to save the output plots.
pngDir+='./dwDphi_'+spwType+'/' +srcName+'/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

# In the MS calibration script, DV10 = id37 has been picked as the reference.
refAnt=37

#-- Calculate the dPWV proxy for the *reference* antenna.
print('Calculating reference antenna.')
mjd_ref, dpw_refXX, dpw_refYY=dPWV(vis, atmScan, tgtScan, refAnt, useSQLD)

#-- Loop over all antennas.

for myant in antIDs:

```

```

# Report progress.
print('Calculating antenna ', int(myant), '<-->', antNames[myant])

#- Calculate the dPWV proxy for antenna ant.
time_pwv, dpw_XX, dpw_YY=dPWV(vis, atmScan, tgtScan, int(myant), useSQLD)

# Subtract the dPWV at the reference antenna.
dwx=dpw_XX-dpw_refXX
dwy=dpw_YY-dpw_refYY

#- Time->Phase map/dict.
tpDict=getPhase(gTable, myant, 5) # SPW 5

time_phi=tpDict['mjd']
phx=tpDict['phiXX']
phy=tpDict['phiYY']

#- Make a plot with an overplot: Phases vs. dPWV

# Plot phases in red: XX solid line, YY dashed.
plt.plot(time_phi, phx, '-r')
plt.plot(time_phi, phy, '--r')

# Plot our dPWV-proxy, with refAnt-offset subtracted and scaled by -25.0
# Blue color, XX solid, YY dashed
plt.plot(time_pwv, -25.0*dwx, '-b')
plt.plot(time_pwv, -25.0*dwy, '--b')

# Add the antenna name as a plot title.
plt.title(antNames[myant])

# Save to a file and close.
plt.savefig(pngDir+'dwDp_'+antNames[myant]+' .png')
plt.show()
plt.close()

return

#-----

def 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*.

```

```

"""

#-- 'Static' settings.

# Our input MS.
vis='uid____A002_X110fd13_X1099e.ms'

# Find the antenna list (the IDs and corresponding names)
msmd.open(vis)
antIDs=myMsmd.antennasids()
antNames=myMsmd.antennanames()
msmd.done()

#-- Do we analyse the BP calibrator or the Sun? It controls the atm/tgtScan.
if (doSun):

    # The Sun's target-scan and its preceding ACD-calibration ATM scan.
    atmScan=8
    tgtScan=9

    srcName='sun'
    # Self-calibrated solar phase solution
    gTable='../calibrated/lowres-sc.phase_int'

else:

    # BP calibrator: The target and ACD-calibration scans.
    atmScan=4
    tgtScan=5

    srcName='BPcal'
    # Band-pass calibrator phase gaintable - raw, i.e. without WVR correction.
    gTable=vis+'.ap_pre_bandpass'

#-- Do we use ACs or SQLD for dPWV calculations? We have different time
# resolutions: Phase solution may be integration based (for the Sun), so
# of the same time cadence as the AC data, or 10x undersampled (for the BP
# calibrator, where we use solint='1.92s' in gaincal()). The SQLD is 12x
# faster than ACs.

if (useSQLD):
    # SPW type - part of the output-plots directory name.
    spwType='sqld'
else: # ACs
    # SPW type - part of the output-plots directory name.
    spwType='ac'

#-- Directory, where to save the output plots.
pngDir='../1corr_dwDphi_'+spwType+'/' + srcName + '/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

# In the MS calibration script, DV10 = id37 has been picked as the reference.
refAnt=37

# Calculate the dPWV proxy for the reference antenna.
print('Calculating reference antenna.')
mjd_ref, dpw_refXX, dpw_refYY=dPWV(vis, atmScan, tgtScan, refAnt, useSQLD)

```

```
#-- Loop over all antennas
```

```
for myant in antIDs:
```

```
# Report progress.
```

```
print ('Calculating antenna ', int(myant), '<-->', antNames[myant])
```

```
# Extract phases as the time -> phase map/dictionary.
```

```
tpDict=getPhase(gTable, int(myant), 5) # use SPW 5
```

```
time_phi=tpDict['mjd']
```

```
phx=tpDict['phiXX']
```

```
phy=tpDict['phiYY']
```

```
# Calculate the dPWV proxy for antenna ant.
```

```
time_pwv, dpw_XX, dpw_YY=dPWV(vis, atmScan, tgtScan, int(myant), useSQLD)
```

```
# Subtract the dPWV at the reference antenna.
```

```
ddpwXX=dpw_XX- dpw_refXX
```

```
ddpwYY=dpw_YY- dpw_refYY
```

```
# We have different time resolutions/cadences: Phase solution may be  
# integration-based (for the Sun), so of the same time cadence as the AC  
# dPWV data, or 10x undersampled (for the BP calibrator, where we use  
# solint='1.92s' in gaincal()). The SQLD is yet 12x faster than ACs, so if  
# we use it, our dPWV proxy can have even 120x better resolution than the  
# phase solutions (if we use 10 integrations as in the BP calibrator case).  
# Solution: Make the resolution/cadence conforming to that of the *phases*.
```

```
dwx=np.interp(time_phi, time_pwv, ddpwXX)
```

```
dwy=np.interp(time_phi, time_pwv, ddpwYY)
```

```
#-- Calculate the correlations between dPWV and phase with optimised
```

```
# removal of the time-trend.
```

```
rcXX=paregression(dwx, phx)
```

```
rcYY=paregression(dwy, phy)
```

```
# Extract the time-trend.
```

```
ttXX=rcXX['tTrend']
```

```
ttYY=rcYY['tTrend']
```

```
# Extract the coefficient of linear correlation.
```

```
KXX=rcXX['K']
```

```
KYY=rcYY['K']
```

```
# Extract the offset of linear correlation.
```

```
QXX=rcXX['Q']
```

```
QYY=rcYY['Q']
```

```
# Extract fit-quality measure (Xi-square) of the linear correlation.
```

```
Xi2XX=rcXX['Xi2']
```

```
Xi2YY=rcYY['Xi2']
```

```
# Print the extracted values
```

```
print ('T-trend = ', ttXX, ttYY)
```

```
print ('K = ', KXX, KYY)
```

```
print ('Q = ', QXX, QYY)
```

```
print ('Xi2 = ', Xi2XX, Xi2YY)
```

```

#- Make scatter plots: dPWW over time-detrended phase. Eeach pol. extra

# Time vector, start with 1.
tVec=np.arange(phiSize)+1.0

# Detrended phases.
pdXX=phx-ttXX*tVec
pdYY=phy-ttYY*tVec

# Make XX and YY plots as subplots (horizontally)
ff, (sp1, sp2)=plt.subplots(1,2)

# Plot the correlations as red dots.
sp1.plot(dwX,pdXX, '.r')
sp2.plot(dwY,pdYY, '.r')

# Add the antenna name as a plot title.
plt.title(antNames[myant])

# Save to a file and close.
plt.savefig(pngDir+'dwDp_'+antNames[myant]+'.png')
plt.show()
plt.close()

return

#-----

def 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.
    """

    #- 'Static' settings.

    # Our input MS.
    vis='uid__A002_X110fd13_X1099e.ms'

    # Find the antenna list (the IDs and corresponding names)
    msmd.open(vis)
    antIDs=myMsmd.antennasids()
    antNames=myMsmd.antennanames()
    msmd.done()

```

```

#-- Do we analyse the BP calibrator or the Sun? It controls the atm/tgtScan.
if (doSun) :

    # The Sun's target-scan and its preceding ACD-calibration ATM scan.
    atmScan=8
    tgtScan=9

    srcName='sun'
    # Self-calibrated solar phase solution
    gTable='../calibrated/lowres-sc.phase_int'

else :

    # BP calibrator: The target and ACD-calibration scans.
    atmScan=4
    tgtScan=5

    srcName='BPcal'
    # Band-pass calibrator phase gaintable - raw, i.e. without WVR correction.
    gTable=vis+'..ap_pre_bandpass'

#-- Do we use ACs or SQLD for dPWV calculations? We have different time
# resolutions: Phase solution may be integration based (for the Sun), so
# of the same time cadence as the AC data, or 10x undersampled (for the BP
# calibrator, where we use solint='1.92s' in gaincal()). The SQLD is 12x
# faster than ACs.

if (useSQLD) :
    # SPW type - part of the output-plots directory name.
    spwType='sqld'
else : # ACs
    # SPW type - part of the output-plots directory name.
    spwType='ac'

#-- Directory, where to save the output plots.
pngDir='../blindFix_'+spwType+'/' + srcName + '/'
if (not os.path.exists(pngDir)) : os.makedirs(pngDir)

# Diagnostic output to a text file in the same dir as the plots.
diagTxt=open(pngDir+'diag_outs.txt', 'w')

# In the MS calibration script, DV10 = id37 has been picked as the reference.
refAnt=37

# Calculate the dPWV proxy for the reference antenna.
print('Calculating reference antenna.')
mjd_ref, dpw_refXX, dpw_refYY=dPWV(vis, atmScan, tgtScan, refAnt, useSQLD)

#-- Loop over all antennas

for myant in antIDs:

    # Report progress.

```

```

print('Calculating antenna ',int(myant),'<-->',antNames[myant])

# Extract phases as the time -> phase map/dictionary.
tpDict=getPhase(gTable,int(myant),5) # use SPW 5

time_phi=tpDict['mjd']
phx=tpDict['phiXX']
phy=tpDict['phiYY']

# Calculate the dPWV proxy for antenna ant.
time_pwv,dpw_XX,dpw_YY=dPWV(vis,atmScan,tgtScan,int(myant),useSQLD)

# Subtract the dPWV at the reference antenna.
ddpwXX=dpw_XX-dpw_refXX
ddpwYY=dpw_YY-dpw_refYY

# Make the resolutions conforming to that of the phase solution.
# See explanation at the same point of dpwvXphi_corrs().
dwx=np.interp(time_phi,time_pwv,ddpwXX)
dwy=np.interp(time_phi,time_pwv,ddpwYY)

#-- Calculate the correlations between dPWV and phase with optimised
# removal of the time-trend in phases.

rcXX=paregression(dwx,phx)
rcYY=paregression(dwy,phy)

# Extract the optimized time-trend coefficients (dPhi=t_phi*i) for phases.
ttXX=rcXX['tTrend']
ttYY=rcYY['tTrend']

# Extract the coefficient of linear correlation between dPWV and dPhi.
KXX=rcXX['K']
KYY=rcYY['K']

# Extract the offset of linear correlation between dPWV and dPhi.

QXX=rcXX['Q']
QYY=rcYY['Q']

# Extract fit-quality measure (Xi-square) of the linear correlation.
Xi2XX=rcXX['Xi2']
Xi2YY=rcYY['Xi2']

# Print the extracted values
print('T-trend = ',ttXX,ttYY)
print('K = ',KXX,KYY)
print('Q = ',QXX,QYY)
print('Xi2 = ',Xi2XX,Xi2YY)

#-- Compare the optimized detrend in phases with just the blind detrending
# the dPWV.

# Time measured as an integration index.
iVect=np.arange(phiSize)

# Time-trends solely in dPWV (blind = no phase knowledge)

```

```

wtx=np.polyfit(iVect,dwx,1)[0]
wty=np.polyfit(iVect,dwy,1)[0]

# How much they are different (by what factor) from the optimized detrend?
# We use scaling  $t_{\phi} = -KK \cdot t_{pwv}$  between the linear time-trend equivalently
# expressed in terms of dPWV series (instead of dPhi one).
trFactorXX=-KXX*wtx/ ttXX
trFactorYY=-KYY*wty/ ttYY

#-- Blindly fix the phases and plot the residuals.

# Scatter-ranges for the uncorrected phases. We compare later with corrected
# and discard those that worsen the situation from plotting

px0Range=np.max(phx)-np.min(phx)
py0Range=np.max(phy)-np.min(phy)

#- Blind correction
# Method 1: Fixed-value scale factor of -25.

bc_pxx1=phx+25.0*dwx
bc_pyy1=phy+25.0*dwy

# Scatter-ranges after correction by method 1
px1Range=np.max(bc_pxx1)-np.min(bc_pxx1)
py1Range=np.max(bc_pyy1)-np.min(bc_pyy1)

# Method 2: Use the correlation factor found by paregression()

bc_pxx2=phx- KXX*dwx
bc_pyy2=phy- KYY*dwy

# Scatter-ranges after correction by method 2
px2Range=np.max(bc_pxx2)-np.min(bc_pxx2)
py2Range=np.max(bc_pyy2)-np.min(bc_pyy2)

# For those antennas, where the blind-correction is not really bad, print
# the ratio between the optimized (using knowledge of phases) and
# and the blind time-trend coefficients for dPWV to a text file.
if (px1Range < 1.5*px0Range):
    print('De-trending factor XX for antenna: ',antNames[myant],': ',
          trFactorXX,file=diagTxt)

if (py1Range < 1.5*py0Range):
    print('De-trending factor YY for antenna: ',antNames[myant],': ',
          trFactorYY,file=diagTxt)

#- Init plots

# Set the sizes of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make XX and YY plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)

```

```

# Add the antenna name as a plot title.
plt.title(antNames[myant])

#- XX: left sub-plot
# Plot the method 1 results as a red line, if the scatter is not too big.
if (px1Range < 1.5*px0Range): sp1.plot(time_phi,bc_pxx1,'-r')
# Overplot the method 2 results as a blue line, if scatter is not too big.
if (px2Range < 1.5*px0Range): sp1.plot(time_phi,bc_pxx2,'-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp1.plot(time_phi,phx,'-g')

#- YY: right sub-plot

# Plot the method 1 results as a red line, if the scatter is not too big.
if (py1Range < 1.5*py0Range): sp2.plot(time_phi,bc_pyy1,'-r')
# Overplot the method 2 results as a blue line, if scatter is not too big.
if (py2Range < 1.5*py0Range): sp2.plot(time_phi,bc_pyy2,'-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp2.plot(time_phi,phy,'-g')

#- Save to a file and close.
plt.savefig(pngDir+'phi_residua_'+antNames[myant]+'-g.png')
plt.show()
plt.close()

# Flush the diagTxt buffer.
diagTxt.flush()

# Close the (single) diagnostic output file.
diagTxt.close()
return

#-----
# 7. Compare phases by dPWV proxies with those from wvrgcal() - BP calibrator
#-----

def 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 tp 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).
"""

# Create the table tool and use it to open the WVR caltable.
cTab=tbtool()
cTab.open('wvr_gcal.W')

# Construct a query string: Selection by antenna, spw, and field.
selection="ANTENNA1==" + str(antID) + " && SPECTRAL_WINDOW_ID==" + str(spw)
selection+=" && FIELD_ID==1" # Field 1 <--> BP calibrator

# Extract the sub-table that correspond to the selection criteria.
# We need just the time and gain columns.
ctSel=cTab.query(selection,sortlist="TIME",columns="TIME,CPARAM")

# Extract all the column data for the above selection.
times_all=ctSel.getcol("TIME")
gains_all=ctSel.getcol("CPARAM")

# Data are extracted, we may close the tab tool.
cTab.done()

# The gains array has a shape (1 x 1 x nTimeSamples), there are no separate
# polarization values, just I-component of Stokes. Get rid of the two
# degenerated axes.
gI_all=gains_all[0][0]

# FIELD_ID is not the ultimate selector - we still have the data from other
# scans than the BP scan. Call to getPhase() is the easiest way how to get the
# times of the BP scan. Alternatively, msmdtool() with getTimesForScan() may
# be used.
gTable=vis+'.ap_pre_bandpass'
timesBP=getPhase(gTable,antID,spw) ["mjd"]

tScanBeg=np.min(timesBP)
tScanEnd=np.max(timesBP)

# Sub-select just the BP-scan data.
# Note: Pytom requires using '&' operator and not 'and' here.
scanRange=np.where((times_all >= tScanBeg) & (times_all <= tScanEnd))

times=times_all[scanRange]
gI=gI_all[scanRange]

# We are interested just in the phases.
phase=np.angle(gI)

# Use np.unwrap in order to get rid of the phase jumps at +/- PI.
uPhase=np.unwrap(phase)

# Form the returned result as a dictionary.
# Return the phases of XX and YY (the same value) separatelly in order to have
# the result conforming to those of getPhase() and dPWV()
pDict={'mjd':times, 'phiXX':uPhase, 'phiYY':uPhase}

return pDict

```

```
#-----

def 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. wvgcal()-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 wvgcal() calculation in blue.
    """

    # Input MS.
    vis='uid___A002_X110fd13_X1099e.ms'

    # Output directory, where to save the plots.
    pngDir='./PWV_compare_AC/'
    if (not os.path.exists(pngDir)): os.makedirs(pngDir)

    # Calculate the dPWV proxy for the reference antenna (DV10 = id 37).
    print('Calculating the reference antenna.')
    mjd_ref, dpw_refXX, dpw_refYY=dPWV(vis,4,5,37)

    #-- Loop over all antennas

    # Find the antenna list (the IDs and corresponding names)
    msmd.open(vis)
    antIDs=myMsmd.antennasids()
    antNames=myMsmd.antennanames()
    msmd.done()

    for myAnt in antIDs:

        # Report progress.
        print('Processing antenna ',int(myAnt),'<-->',antNames[myAnt])

        # Calculate our dPWV proxy for the BP-scan=5 using DL-cals from atmScan=4
        time_d,pwvXX,pwvYY=dPWV(vis,4,5,int(myAnt))

        # Subtract the reference-antenna dPWV
        pwvXX=pwvXX-dpw_refXX
        pwvYY=pwvYY-dpw_refYY

        # Extract wvgcal()-produced phases for SPW 5.
        phiWVR=wvrPhase(vis,myAnt,5)

        # Extract times for the W-caltable.
        time_w=phiWVR['mjd']

        # Phases for XX and YY extra.
        phi_w_XX=phiWVR['phiXX']
```

```

phi_w_YY=phiWVR['phiYY']

#- Init plots

# Set the sizes of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make XX and YY plots as subplots (horizontally)
ff, (sp1, sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.title(antNames[myAnt])

#- XX: left sub-plot
# Plot the dPWV() results scaled by -25.0 as a red line.
sp1.plot(time_d, -25.0*pwvXX, '-r')
# Overplot the wvrgcal() results as a blue line.
sp1.plot(time_w, phi_w_XX, '-b')

#- YY: right sub-plot
# Plot the dPWV() results scaled by -25.0 as a red line.
sp2.plot(time_d, -25.0*pwvYY, '-r')
# Overplot the wvrgcal() results as a blue line.
sp2.plot(time_w, phi_w_YY, '-b')

#- Save to a file and close.
plt.savefig(pngDir+'pwv_compare_'+antNames[myAnt]+' .png')
plt.show()
plt.close()

```

return

```

#-----

def 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 (1) 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 comaprison between ucorrected phase and its improved counterpart fixed using our dPWV() proxy and the theoretical factor -25.0 (called "Method 1" in blindPhiFox())

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.

"""

```
#-- What type of data we are using for analysis? Check the spw definition.
myMsm = msmdtool()
myMsm.open(vis)

freqs=myMsm.chanfreqs(spwID)*1.0e-9

# Extract also the list of all antennas once we have the msmd tool ready.
antIDs=myMsm.antennas()
antNames=myMsm.antennanames()

myMsm.done()

nChans=len(freqs)

# Corresponding high-frequency SPW for subtraction of the dry component.
# To be set below in line with the spwID type.
spwHigh=-1

if (spwID in [0,1,2,3,13,14,15,16]):
    # SQLD SPW.
    spwType='sqld'
    # SQLDs follow consecutively in line.
    spwHigh=spwID+3
```

```

# Must have corresponding spwRes defined, we cannot guess it.
if (spwRes==-1):
    print('Parameter *spwRes* MUST be defined for SQLD spw.')
    return

else:
    # ACs are interleaved (appear in pairs: resolved/integrated).
    spwHigh=spwID+6

    if (nChans>1):
        # Channel-resolved AC SPW
        spwType='ac-res'
        # We are the resolved SPW
        spwRes=spwID

    else:
        # Channel-integrated AC SPW
        spwType='ac-int'
        # An obvious guess for resolved SPW.
        if (spwRes==-1): spwRes=spwRes=spwID-1

# Output figures directory - separate for each SPW type.
pngDir='./2wvrs_'+spwType+'/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

#-- Calculate the dPWV proxy for the reference antenna.
print('Calculating the reference antenna.')

if (spwType=='ac-res'):
    # Chan-resolved SPW: Try to integrate over ~half of it.
    mjd_ref, dpw_refXX, dpw_refYY=dPWV_spwPart(vis, atmScan, tgtScan, refAnt, spwID,
                                                startChan=10, stopChan=nChans//2)
    mjd_ref, dpw_refXX_hi, dpw_refYY_hi=dPWV_spwPart(vis, atmScan, tgtScan, refAnt,
                                                spwHigh, startChan=10,
                                                stopChan=nChans//2)

else:
    mjd_ref, dpw_refXX, dpw_refYY=dPWV(vis, atmScan, tgtScan, refAnt, spwID, spwRes)
    mjd_ref, dpw_refXX_hi, dpw_refYY_hi=dPWV(vis, atmScan, tgtScan, refAnt, spwHigh,
                                                spwRes)

#-- Loop over all antennas

for myant in antIDs:

    # Report progress.
    print('Calculating antenna ', int(myant), '<-->', antNames[myant])

    # Calculate the dPWV proxy for the antenna *myant*.
    if (spwType=='ac-res'):
        # Chan-resolved SPW: Try to integrate over ~half of it.
        time_dpw, dpwXX, dpwYY=dPWV_spwPart(vis, atmScan, tgtScan, int(myant), spwID,

```

```

        startChan=10, stopChan=nChans//2)
    time_dpw, dpwXX_hi, dpwYY_hi=dPWV_spwPart(vis, atmScan, tgtScan, int(myant),
                                              spwHigh, startChan=10,
                                              stopChan=nChans//2)

else:
    time_dpw, dpwXX, dpwYY=dPWV(vis, atmScan, tgtScan, int(myant), spwID, spwRes)
    time_dpw, dpwXX_hi, dpwYY_hi=dPWV(vis, atmScan, tgtScan, int(myant), spwHigh,
                                       spwRes)

# Subtract the dPWV at the reference antenna.
ddpwXX=dpwXX-dpw_refXX-dpwXX_hi+dpw_refXX_hi
ddpwYY=dpwYY-dpw_refYY-dpwYY_hi+dpw_refYY_hi

# Extract the *uncorrected* phases from the ap-prebandpass caltable.
gTable=vis+'.ap_pre_bandpass'
tpDict=getPhase(gTable, myant, spwRes)

time_ph=tpDict['mjd']
phx=tpDict['phiXX']
phy=tpDict['phiYY']

# Make dPWV's resolution & range conforming to that of the phase solution.
dwx=np.interp(time_ph, time_dpw, ddpwXX)
dwy=np.interp(time_ph, time_dpw, ddpwYY)

#-- Blindly fix the raw phases and plot the residuals.

# Scatter-ranges for the uncorrected phases. We compare later with corrected
# and discard those that worsen the situation from plotting

px0Range=np.max(phx)-np.min(phx)
py0Range=np.max(phy)-np.min(phy)

#- Blind correction by the fixed-value scale factor of -25.

bc_pxx=phx+25.0*dwx
bc_pyy=phy+25.0*dwy

# Scatter-ranges after correction.
px1Range=np.max(bc_pxx)-np.min(bc_pxx)
py1Range=np.max(bc_pyy)-np.min(bc_pyy)

#-- Extract *WVR-corrected phases* from the WVR-fixed ap-pre-BP caltable.
gTableWVRfix=vis+'.ap_pre_bandpass-wvrFix'
tpcDict=getPhase(gTableWVRfix, myant, spwRes)

time_phc=tpcDict['mjd']
phx_c=tpcDict['phiXX']
phy_c=tpcDict['phiYY']

#- Init plots

# Set the sizes of the plot

```

```

plt.rcParams["figure.figsize"] = (10,5)
# Make XX and YY plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.title(antNames[myant])

#- XX: left sub-plot
# Plot the dPWV-fix results as a red line, if the scatter is not too big.
sp1.plot(time_ph,bc_pxx, '-r')
# Overplot the WVR-fixed phases as blue
sp1.plot(time_phc,phx_c, '-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp1.plot(time_ph,phx, '-g')

#- YY: right sub-plot

# Plot the method 1 results as a red line, if the scatter is not too big.
sp2.plot(time_ph,bc_pyy, '-r')
# Overplot the WVR-fixed phases in blue
sp2.plot(time_phc,phy_c, '-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp2.plot(time_ph,phy, '-g')

#- Save to a file and close.

plt.savefig(pngDir+'phi_residua_'+antNames[myant]+' .png')
plt.show()
plt.close()

```

return

```

#-----
def 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 <Tsky(f)> and tau_1(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.
    """

    #-- What type of data we are using for analysis? Check the spw definition.
    myMsm = msmdtool()
    myMsm.open(vis)

    freqs=myMsm.chanfreqs(spwLow)*1.0e-9

    # Extract also the list of all antennas once we have the msmd tool ready.
    antIDs=myMsm.annenaids()

```

```

antNames=myMsmc.antennanames()

myMsmc.done()

nChans=len(freqs)

# Corresponding high-frequency SPW for subtraction of the dry component.
# To be set below in line with the spwLow type.
spwHigh=-1

if (spwLow in [0,1,2,3,13,14,15,16]):
    # SQLD SPW.
    spwType='sqld'
    # SQLDs follow consecutively in line.
    spwHigh=spwLow+3

    # Must have corresponding spwRes defined, we cannot guess it.
    if (spwRes==-1):
        print('Parameter *spwRes* MUST be defined for SQLD spw.')
        return

else:
    # ACs are interleaved (appear in pairs: resolved/integrated).
    spwHigh=spwLow+6

    if (nChans>1):
        # Channel-resolved AC SPW
        spwType='ac-res'
        # We are the resolved SPW
        spwRes=spwLow

        print('dWdP() inside this function is not ready yet for resolved SPWs!')
        return

    else:
        # Channel-integrated AC SPW
        spwType='ac-int'
        # An obvious guess for resolved SPW.
        if (spwRes==-1): spwRes=spwRes=spwLow-1

# Output figures directory - separate for each SPW type.
pngDir='./2wvrs_3_'+spwType+'/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

#-- Create the atmosphere model for our SPWs (Lo/Hi):
# Use the *resolved* swps corresponding to our integrated spwHigh/spwLow

atmLow=wopacity(vis, spwRes, tgtScan)
atmHigh=wopacity(vis, spwRes+6, tgtScan)

# Collect the sub-models for spws as a map.
atmModel={spwLow:atmLow, spwHigh:atmHigh}

#-- Calculate the dPWV proxy for the reference antenna.
print('Calculating the reference antenna.')

```

```

mjd_ref,dwXX_ref,dwYY_ref,dpXX_ref,dpYY_ref=dWdP(vis,atmScan,tgtScan,
                                                atmModel,refAnt,spwLow)

#-- Loop over all antennas

for myant in antIDs:

    # Report progress.
    print('Calculating antenna ',int(myant),'<-->',antNames[myant])

    # Calculate the dW (wet) and dP (dry) atm. variations.
    time_dpw,dwXX,dwYY,dpXX,dpYY=dWdP(vis,atmScan,tgtScan,
                                        atmModel,int(myant),spwLow)

    # Subtract the dPWV at the reference antenna.
    ddpwXX=dwXX-dwXX_ref
    ddpwYY=dwYY-dwYY_ref

    # Extract the *uncorrected phases* from the ap-prebandpass caltable.
    gTable=vis+'.ap_pre_bandpass'
    tpDict=getPhase(gTable,myant,spwRes)

    time_ph=tpDict['mjd']
    phx=tpDict['phiXX']
    phy=tpDict['phiYY']

    # Make dPWV's resolution & range conforming to that of the phase solution.
    dwx=np.interp(time_ph,time_dpw,ddpwXX)
    dwy=np.interp(time_ph,time_dpw,ddpwYY)

    #-- Blindly fix the raw phases and plot the residuals.

    # Scatter-ranges for the uncorrected phases. We compare later with corrected
    # and discard those that worsen the situation from plotting

    px0Range=np.max(phx)-np.min(phx)
    py0Range=np.max(phy)-np.min(phy)

    #- Blind correction by the fixed-value scale factor of -25.

    bc_pxx=phx+25.0*dwx
    bc_pyy=phy+25.0*dwy

    # Scatter-ranges after correction.
    px1Range=np.max(bc_pxx)-np.min(bc_pxx)
    py1Range=np.max(bc_pyy)-np.min(bc_pyy)

    #-- Extract the *WVR-corrected phases* from WVR-fixed ap-pre-BP caltable
    gTableWVRfix=vis+'.ap_pre_bandpass-wvrFix'
    tpcDict=getPhase(gTableWVRfix,myant,spwRes)

    time_phc=tpcDict['mjd']
    phx_c=tpcDict['phiXX']
    phy_c=tpcDict['phiYY']

```

```

#- Init plots

# Set the sizes of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make XX and YY plots as subplots (horizontally)
ff, (sp1, sp2) = plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.title(antNames[myant])

#- XX: left sub-plot
# Plot the dPWV-fix results as a red line, if the scatter is not too big.
sp1.plot(time_ph, bc_pxx, '-r')
# Overplot the WVR-fixed phases as blue
sp1.plot(time_phc, phx_c, '-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp1.plot(time_ph, phx, '-g')

#- YY: right sub-plot

# Plot the method 1 results as a red line, if the scatter is not too big.
sp2.plot(time_ph, bc_pyy, '-r')
# Overplot the WVR-fixed phases in blue
sp2.plot(time_phc, phy_c, '-b')
# Overplot the uncorrected phase for comparison as a green line (always).
sp2.plot(time_ph, phy, '-g')

#- Save to a file and close.

plt.savefig(pngDir+'phi_residua_'+antNames[myant]+'.png')
plt.show()
plt.close()

return

#-----

def 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).
    """

    #-- 'Static' settings.

    # Our MS
    vis='uid____A002_X110fd13_X1099e.ms'

    # BP calibrator: The target and ACD-calibration scans.

```

```

atmScan=4
tgtScan=5

# In order to create the homo-located (with the same support) time series
# we use np.interp() in this case - see below. Hence, no rebin factors nor
# data trimming comes here, just the suffix of the out-plot directory.
if (useSQLD) :

    # SPW type - part of the output-plots directory name.
    spwType='sqli'

else: # ACs

    # SPW type - part of the output-plots directory name.
    spwType='ac'

# In the MS calibration script, DV10 = id37 has been picked as the reference.
refAnt=37

# Calculate the dPWV proxy for the reference antenna.
print('Calculating dPWV for the reference antenna.')
#--- Try the partial spw integration over SPW 5 instead of using SPW 6
#mjd_ref,dpw_refXX,dpw_refYY=dPWV(vis,atmScan,tgtScan,refAnt,useSQLD)
spwType='spw5'
mjd_ref,dpw_refXX,dpw_refYY=dPWV_spwPart (vis,atmScan,tgtScan,refAnt,
                                           spwRes=5,startChan=10,stopChan=60)

# Directory, where to save the plots.
pngDir='/scratch/barta/LB_Solar/Project_2022.1.00031.CSV/'
pngDir+='Sun_10_a_05_INT/X1099e/2wvrs_'+spwType+'/'
if (not os.path.exists(pngDir)): os.makedirs(pngDir)

# Diagnostic output to a text file in the same dir as the plots.
diagTxt=open(pngDir+'diag_outs.txt','w')

#-- Loop over all antennas

# Find the antenna list (the IDs and corresponding names)
msmd.open(vis)
antIDs=myMsmd.antennasids()
antNames=myMsmd.antennanames()
msmd.done()

for myant in antIDs:

    # Report progress.
    print('Calculating antenna ',int(myant),'<-->',antNames[myant])

    # Calculate the dPWV proxy for antenna ant.
    #--- Try the partial SPW5 integration instead of SPW6.
    #mjd_ant,dpw_XX,dpw_YY=dPWV(vis,atmScan,tgtScan,int(myant),useSQLD)
    time_dpwv,dpw_XX,dpw_YY=dPWV_spwPart (vis,atmScan,tgtScan,int(myant),
                                           spwRes=5,startChan=10,stopChan=60)

    # Subtract the dPWV at the reference antenna.

```

```

ddpwXX=dpw_XX-dpw_refXX
ddpwYY=dpw_YY-dpw_refYY

#-- 1. dataseries: Re-calculate dPWV to phase corrections by the theoretical
#                scale factor of -25.

dPhiXX=-25.0*ddpwXX
dPhiYY=-25.0*ddpwYY

#-- 2. dataseries: Extract the phase corrections from the wvr cal() table.

# Extract wvr cal()-produced phases for SPW 5.
phiWVR=wvrPhase(vis,myant,5)

# Extract times for the W-caltable.
time_wvr=phiWVR['mjd']

# Phases for XX and YY extra.
phi_w_XX=phiWVR['phiXX']
phi_w_YY=phiWVR['phiYY']

#-- Interpolate the phases as from dPWV to the time-points from WVR.

phi_p_XX=np.interp(time_wvr,time_dpwv,dPhiXX)
phi_p_YY=np.interp(time_wvr,time_dpwv,dPhiYY)

#-- Calculate the correlations between dPWV-based and WVR-based
# phase-corrections with optimised removal of the time-trend.

rcXX=paregression(phi_w_XX,phi_p_XX)
rcYY=paregression(phi_w_YY,phi_p_YY)

# Extract the optimized time-trend coefficients (dPhi=t_phi*i) for phases.
ttXX=rcXX['tTrend']
ttYY=rcYY['tTrend']

# Extract the coefficient of linear correlation between dPWV and dPhi.
KXX=rcXX['K']
KYY=rcYY['K']

# Extract the offset of linear correlation between dPWV and dPhi.

QXX=rcXX['Q']
QYY=rcYY['Q']

# Extract fit-quality measure (Xi-square) of the linear correlation.
Xi2XX=rcXX['Xi2']
Xi2YY=rcYY['Xi2']

# Print the extracted values for quick inspection.
print('T-trend = ',ttXX,ttYY)
print('K = ',KXX,KYY)
print('Q = ',QXX,QYY)
print('Xi2 = ',Xi2XX,Xi2YY)

#-- Print the results one more to the table in the file.
print(antNames[myant],'\t',ttXX,'\t',ttYY,'\t',KXX,'\t',KYY,'\t',

```

```

    QXX, '\t', QYY, '\t', Xi2XX, '\t', Xi2YY, file=diagTxt)

    #- Make scatter plots: dPhi(WVR) vs.dPhi(PWV)

    phiSize=time_wvr.shape[0]

    # Time-index vector, start with 1.
    tVec=np.arange(phiSize)+1.0

    # De-trended PWV-based phases.
    pdXX=phi_p_XX-ttXX*tVec
    pdYY=phi_p_YY-ttYY*tVec

    # Make XX and YY plots as subplots (horizontally)
    ff, (sp1, sp2)=plt.subplots(1,2)

    # Plot the correlations as red dots.
    sp1.plot(phi_w_XX, pdXX, '.r')
    sp2.plot(phi_w_YY, pdYY, '.r')

    # Add the antenna name as a plot title.
    plt.title(antNames[myant])

    # Save to a file and close.
    plt.savefig(pngDir+'dwvrDpww_'+antNames[myant]+' .png')
    plt.show()
    plt.close()

    # --> endfor(over-all-antennas)

    # Close the (single) diagnostic output file.
    diagTxt.close()
    return

#-----
#                               7.1 PWV for the Sun vs. phases
#-----

def 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 pahse-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 2PI 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 worrysome 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 easilly bigger then 360 deg.
    """

```

```

#-- Phase-cal INF solution interpolated over a solar scan.

# The phase-inf solutions have been obtained from the ms.split MS, where the
# SPWs has been meanwhile renumbered. Make a map between raw and split spwlds.
splitMap={5:0,7:1,9:2,11:3}

# The *spw* parameter is in the original notation, find its split counterpart.
spwSplit=splitMap[spw]

# Extract the phases in their time dependence.
phaseDict=getPhase('uid__A002_X110fd13_X1099e.ms.split.phase_inf',
                    antID, spwSplit)

# Skip the very first phase-inf solution, it comes from the BP-cal scan.
phiTime=phaseDict['mjd'][1:]
phiXX=phaseDict['phiXX'][1:]
phiYY=phaseDict['phiYY'][1:]

#-- PWV-based phase delays.

# Reference-antennna values.
refAnt=37
pwvRef=pwvMap[refAnt]['dW']

# Current antID
pwvTime=pwvMap[antID]['time']
pwvCurr=pwvMap[antID]['dW']

# PWV-based phase delay.
# Approximate quick calculation for LO frequency. Use the central frequency
# for each SPW in the next improved version!!!
dPhi=-25.0*(pwvCurr-pwvRef)

#-- Plot comparison.

# Scale the pahses to degrees.
r2d=180.0/math.pi

plt.plot(pwvTime, r2d*dPhi, 'b-')

plt.plot(phiTime, r2d*phiXX, 'r-')
plt.plot(phiTime, r2d*phiYY, 'g-')

return

#-----

def 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.
    '''

    #-- 1. Phase-cal INF solution interpolated over a solar scan.

```

```

# The phase-inf solutions have been obtained from the ms.split MS, where the
# SPWs has been meanwhile renumbered. Make a map between raw and split spwlds.
splitMap={5:0,7:1,9:2,11:3}

# The *spw* parameter is in the original notation, find its split counterpart.
spwSplit=splitMap[spw]

# Extract the phases in their time dependence.
phaseDict=getPhase('uid___A002_X110fd13_X1099e.ms.split.phase_inf',
                    antID, spwSplit)

# Skip the very first phase-inf solution, it comes from the BP-cal scan.
phiTime=phaseDict['mjd'][1:]
phiXX=phaseDict['phiXX'][1:]
phiYY=phaseDict['phiYY'][1:]

#-- 2. Solar phases by PWV proxy.

# Extract the phases in their time dependence.
phaseDictPWV=getPhase('sol_pwvCalTable.W', antID, spwSplit)

pwvTime=phaseDictPWV['mjd']
pwvPhiXX=phaseDictPWV['phiXX']
pwvPhiYY=phaseDictPWV['phiYY']

#-- 3. Solar phases by self-calibration.

# Extract the phases in their time dependence.
phaseDictSC=getPhase('sol_selfCal-phase-int', antID, spwSplit)

scTime=phaseDictSC['mjd']
scPhiXX=phaseDictSC['phiXX']
scPhiYY=phaseDictSC['phiYY']

#-- Plot comparison.

# Scale the phases to degrees.
r2d=180.0/math.pi

# Make a difference between selfcal and PWV-based solutions.
# Wrap-back the artificial jumps - they are failed self-cal solutions.
diffXX=r2d*wrapBack(pwvPhiXX-scPhiXX)
diffYY=r2d*wrapBack(pwvPhiYY-scPhiYY)

plt.plot(pwvTime,diffXX,'r-')
plt.plot(pwvTime,diffYY,'b-')

plt.plot(phiTime,r2d*phiXX,'r-')
plt.plot(phiTime,r2d*phiYY,'b-')

return

#-----
def 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.

```

"""

# This is to be generalized. In case of multiple solar scans, also the *pwvMap*
# parameter shall have a bit different structure (additional scan dimension).
solScan=9

#-- Prepare the empty pwvCalTable with the structure of phase gaintable.

# Remove the old version first.
os.system('rm -rf ' + pwvCalTableName)

# Copy the structure of the phase-inf gaintable as an empty 'template'.
gCalTable=tbtool()
gCalTable.open(vis+'.split.phase_inf')
pwvCalTable=gCalTable.copy(pwvCalTableName,deep=True,norows=True)
pwvCalTable.flush()

# Close both (original & copied template) tables.
pwvCalTable.done()
gCalTable.done()

#-- Find the per-SPW Phase Factors  $PF=d\Phi(f_{spw})/dPWV$  from ATM model.

myAtm=attool()
setAtmTool4SPWs(myAtm,vis,solScan,wvHeightScale=1.5,tempDropPerKm=-5.6)

# Set approximate PWV, the path length should not actually depend on that.
pwvMean=aU.getMedianPWV(vis)[0]
myAtm.setUserWH2O(qa.quantity(pwvMean,'mm'))

# Phase delays per 1mm PWV [rad/mm] for each SPW.
nSpws=myAtm.getNumSpectralWindows()
PF=np.zeros(nSpws)

for spw in range(nSpws):
    # Central channel.
    cChan=myAtm.getNumChan(spw)//2

    # Dispersive and non-dispersive delays for central channel.
    dd=myAtm.getDispersivePhaseDelay(cChan,spw)['value'][0]
    nd=myAtm.getNonDispersivePhaseDelay(cChan,spw)['value'][0]

    # Set total delay per 1mm PWV as the PF factor for the current SPW.
    # It is a *delay*, so add the minus. Express in radians/mm.
    PF[spw]=-math.pi*(nd+dd)/(pwvMean*180.0)

#--> endfor(spws)
myAtm.done()

```

```

#-- Fill the pwvCalTable with contents.

# Re-open the calTable in the rw mode.
pwvCalTable.open(pwvCalTableName,nomodify=False)
# Set the original table-write offset (start-row for write) to zero.
tabOffset=0

# PWV(t) for the reference antenna - to be subtraced.
pwvRef=pwvMap[refAnt]['dW']

#-- Loop over all antennas.

for antID in pwvMap.keys():

    time=pwvMap[antID]['time']
    pwv=pwvMap[antID]['dW']

    nSamples=len(time)

    #- Create 'static' columns. Some (e.g., interval, observation_id, etc.) are
    # left in their default values of zero.

    # The 'scalar columns' are easy - just nSamples-sized arrays.
    field_id=np.zeros(nSamples,dtype=int) # The Sun is field 0. Generalize!!!
    antenna_1=int(antID)*np.ones(nSamples,dtype=int)
    antenna_2=int(refAnt)*np.ones(nSamples,dtype=int)
    scan_num=solScan*np.ones(nSamples,dtype=int)

    # For the 'vector columns' we have to keep shape nPol x nChans x nSamples.
    paramErr=np.zeros((1,1,nSamples))
    flag=np.zeros((1,1,nSamples),dtype=int)
    snr=5.0*np.ones((1,1,nSamples)) # SNR better than 3.0 is fine.

    # Differential PWV between the current and reference antenna.
    pwvDiff=pwv-pwvRef

    #- Loop over SPWs. We are working with split MS, so the sci spws are 0 - 3.
    for spw in range(4):

        #- Columns that depend on SPW

        # SPW
        c_spw=spw*np.ones(nSamples,dtype=int)

        # Complex gains (just the phase) - the CPARAM column.
        cGain=np.exp(1j*PF[spw]*pwvDiff)
        # It is a vector column: Reshape it to nPols x nChans x nSamples.
        cParam=np.reshape(cGain,(1,1,nSamples))

        #- Write all the sub-columns to the calTable, starting at tabOffset.

        # Make space for writing the columns.
        pwvCalTable.addrows(nSamples)

        # Append the columns.
        pwvCalTable.putcol('TIME',time,startrow=tabOffset)
        pwvCalTable.putcol('FIELD_ID',field_id,startrow=tabOffset)

```

```

pwvCalTable.putcol('SPECTRAL_WINDOW_ID', c_spw, startrow=tabOffset)
pwvCalTable.putcol('ANTENNA1', antenna_1, startrow=tabOffset)
pwvCalTable.putcol('ANTENNA2', antenna_2, startrow=tabOffset)
# INTERVAL defaults to zeros
pwvCalTable.putcol('SCAN_NUMBER', scan_num, startrow=tabOffset)
# OBSERVATION_ID defaults to zero.
pwvCalTable.putcol('CPARAM', cParam, startrow=tabOffset)
pwvCalTable.putcol('PARAMERR', paramErr, startrow=tabOffset)
pwvCalTable.putcol('FLAG', flag, startrow=tabOffset)
pwvCalTable.putcol('SNR', snr, startrow=tabOffset)
# WEIGHT defaults to an empty array.

# Update the write offset.
tabOffset+=nSamples
#--> endfor(spws)

print("Finished antenna ", antID)

#--> endfor(antennas)

# Sync to SDD/HDD.
pwvCalTable.flush()

# Close the tool.
pwvCalTable.done()

```

return

```

#-----
#               7.2 Direct WVR temperatures vs. dPWV
#-----

```

```

def 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 the 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.*

```
    """
```

```
    #--- Find the time-ranges for the off-source subscans.
```

```
    # Attach the msmd tool to our MS.
```

```
    myMsmd=msmdtool()
    myMsmd.open(vis)
```

```
    # Extract sub-scan information for the solar scan.
```

```
    sunSub_info=aU.computeDurationOfScan(tgtScan, mymsmd=myMsmd,
```

```

        returnSubscanTimes=True)

# The off-source (i.e., hi-attenuated sky/atm) sub-scans are the first
# and the last. So, we need to find the number of sub-scans. It is the 2nd
# item of the info - see computeDurationOfScan() in sun_reduction_util.py.
nSubScans=sunSub_info[1]

# The off-source scans. Note: Keys in subInfo start with 1.
offSubScan1=1
offSubScan2=nSubScans

# Extract their time-ranges - we select the data later by them.
tRange1=sunSub_info[2][offSubScan1] # Time-range for the leading subscan.
tRange2=sunSub_info[2][offSubScan2] # Time-range for the trailing subscan.

# Time extracted, close the msmd tool.
myMsmd.done();

# Take the time in the middle of the subscans as representative moment.
t1Rep=0.5*(tRange1[0]+tRange1[1])
t2Rep=0.5*(tRange2[0]+tRange2[1])

# Make the representative times as a 1-d, two-element array.
off_times=np.array([t1Rep,t2Rep])

#--- Find (reference) antenna pointings for those time-ranges.

# Open the pointing table
pTable=tbtool()
pTable.open(vis+'/POINTING')

# Select by antenna
selection="ANTENNA_ID==" +str(refAnt)

ptSel=pTable.query(selection,sortlist="TIME",columns="TIME,DIRECTION")

# Extract the column data: Time and pointing direction as Az/El
times=ptSel.getcol("TIME")
pDir=ptSel.getcol("DIRECTION")

# Data extracted, can close the table tool.
pTable.done()

# Decompose pDir to Az and El - we need 1-D array for subsequent np.interp()
az=pDir[0][0]
el=pDir[1][0]

# Now we can interpolate to the off-source times
az_off=np.interp(off_times, times, az)
el_off=np.interp(off_times, times, el)

#--- Find Az/El of the solar disc center at the same OFF times.

az_sc0,el_sc0=aU.ComputeSolarAzEl(mjdsec=t1Rep)

```

```

az_sc1,el_sc1=aU.ComputeSolarAzEl(mjdsec=t2Rep)

"""
--- This is an alternative approach if the above is too inaccurate ---

# MJD secs to MJD
smjd=qa.convert(qa.quantity(t1Rep,'s'),'d')
# This *does* use the JPL Horizon server query.
sunDict=aU.planet(body='sun',mjd=smjd['value'])

+convert solar Ra/Dec to Az/El using the same approach as in
aU.ComputeSolarAzElLatLong(), just submitting our solar Ra/Dec instead of that
found by aU.ComputeSolarRADec(mjdsec), which does NOT query JPL Horizon server
"""

# Make it a 2-element vector to conform the az/el for off positions.
# Re-scale to radians.
az_sc=np.array([az_sc0,az_sc1])*math.pi/180.0
el_sc=np.array([el_sc0,el_sc1])*math.pi/180.0

# Calculate re-usable radians-to-degs conversion factor.
RAD2DEG=180.0/math.pi

print('OFF position:',az_off*RAD2DEG,el_off*RAD2DEG)
print('SOL position:',az_sc*RAD2DEG,el_sc*RAD2DEG)

#--- Calculate the angular separation

# Unit direction vector to the solar center at beginning and the end.
dirSc1=np.array([math.cos(el_sc[0])*math.cos(az_sc[0]),
                 math.cos(el_sc[0])*math.sin(az_sc[0]),
                 math.sin(el_sc[0])])

dirSc2=np.array([math.cos(el_sc[1])*math.cos(az_sc[1]),
                 math.cos(el_sc[1])*math.sin(az_sc[1]),
                 math.sin(el_sc[1])])

# Unit direction vector to the OFF_SOURCE position at beginning and the end.
dirOff1=np.array([math.cos(el_off[0])*math.cos(az_off[0]),
                  math.cos(el_off[0])*math.sin(az_off[0]),
                  math.sin(el_off[0])])

dirOff2=np.array([math.cos(el_off[1])*math.cos(az_off[1]),
                  math.cos(el_off[1])*math.sin(az_off[1]),
                  math.sin(el_off[1])])

# The dot product of the unit direction vectors is directly cos(delta)
dRad0=math.acos(np.dot(dirSc1,dirOff1))
dRad1=math.acos(np.dot(dirSc2,dirOff2))

# show the result in degs.
print('Offset beg: ',dRad0*RAD2DEG)
print('Offset end: ',dRad1*RAD2DEG)

```

```

    # Return it in radians.
    return (dRad0,dRad1)

#-----

def 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*.
    """

    #-- Find the time-ranges for the entire tgtScan

    # Attach the msmd tool to our MS.
    myMsmd=msmdtool()
    myMsmd.open(vis)

    # Extract sub-scan information for the solar scan.
    sunSub_info=aU.computeDurationOfScan(tgtScan,mymsmd=myMsmd,
                                         returnSubscanTimes=True)

    # The off-source (i.e., hi-attenuated sky/atm) sub-scans are the first
    # and the last. So, we need to find the number of sub-scans. It is the 2nd
    # item of the info - see computeDurationOfScan() in sun_reduction_util.py.
    nSubScans=sunSub_info[1]

    # Extract their time-ranges - we select the data later by them.
    tRangeFirst=sunSub_info[2][1] # Time-range for the leading subscan.
    tRangeLast=sunSub_info[2][nSubScans] # Time-range for the trailing subscan.

    # Subscan time-ranges extracted, close the msmd tool.
    myMsmd.done();

    # The entire scan ranges from beginning of the 1st subscan to the end of
    # the last one.

    tScanBeg=tRangeFirst[0]
    tScanEnd=tRangeLast[1]

    #-- Find (reference) antenna pointings for the full-scan time-range.

    # Open the pointing table
    pTable=tbtool()
    pTable.open(vis+' /POINTING')

    # Select by antenna
    selection="ANTENNA_ID==" +str(refAnt)

    ptSel=pTable.query(selection,sortlist="TIME",columns="TIME,DIRECTION")

    # Extract the column data: Time and pointing direction as Az/EI
    times=ptSel.getcol("TIME")
    pDirs=ptSel.getcol("DIRECTION")

```

```

# Data extracted, can close the table tool.
pTable.done()

# Decompose pDir to Az and El
az=pDirs[0][0]
el=pDirs[1][0]

# Now, select the time-range for the scan only.
# Note that '&' is necessary in the condition, 'and' operator does not work!
scanIdxRange=np.where((times >= tScanBeg) & (times <=tScanEnd))

tScan=times[scanIdxRange]
azTgt=az[scanIdxRange]
elTgt=el[scanIdxRange]

#-- Find Az/El of the solar disc center at the same OFF times.

# We do it more exact way than in sol_offPos(). It has been found that
# aU.ComputeSolarAzEl(mjdsec=mjds) looses precision with time if the analysis
# utils are not updated. So, here, we query the JPL server for RaDec at the
# beginning and the end of the scan and do linear interpolation in RaDec.
# Finally, we convert to AzEl for ALMA location.

#- Scan beginning
# MJD secs to MJD
smjd1=qa.convert(qa.quantity(tScanBeg, 's'), 'd')
# This *does* use the JPL Horizon server query.
sunDict=aU.planet(body='sun', mjd=smjd1['value'], apparent=True)
RaDecRads1=sunDict['directionRadians']

Ra1=RaDecRads1[0]
Dec1=RaDecRads1[1]

#- Scan end
# MJD secs to MJD
smjd2=qa.convert(qa.quantity(tScanEnd, 's'), 'd')
# This *does* use the JPL Horizon server query. 'Apparent' is essential here,
# otherwise we are shifted in AzEl ~10 arcmin w.r.t. the antenna pointings.
sunDict=aU.planet(body='sun', mjd=smjd2['value'], apparent=True)
RaDecRads2=sunDict['directionRadians']

Ra2=RaDecRads2[0]
Dec2=RaDecRads2[1]

#- Interpolate RaDecs

RaSol=np.interp(tScan, [tScanBeg, tScanEnd], [Ra1, Ra2])
DecSol=np.interp(tScan, [tScanBeg, tScanEnd], [Dec1, Dec2])

#- Re-calculate to AzEl for ALMA location. This part has been copied from
# aU.ComputeSolarAzElLatLong() aU.getObservatoryLatLong()

# Conversion constants.
DEG_TO_RAD = math.pi/180.

```

```

RAD_TO_DEG = 180/math.pi
HRS_TO_RAD = math.pi/12.
RAD_TO_HRS = 1.0/HRS_TO_RAD

# ALMA location.
[latitude,longitude,obs] = aU.getObservatoryLatLong(observatory='ALMA')

# Use our JPL-based RaDecs instead of the aU.ComputeSolarRADec(mjdsec)
# Scale them to hours/degs
[RA,Dec] =[RAD_TO_HRS*RaSol,RAD_TO_DEG*DecSol]

# We need Local sidereal time (LST). It cannot be computed as an array
# (there are some conditions/comparisons in aU.ComputeLST()), so we do it
# again at the beginning and at the end of the scan and interpolate.

LST1 = aU.ComputeLST(tScanBeg,longitude)
LST2 = aU.ComputeLST(tScanEnd,longitude)
LST=np.interp(tScan,[tScanBeg,tScanEnd],[LST1,LST2])

phi = latitude*DEG_TO_RAD
hourAngle = HRS_TO_RAD*(LST - RA)

# Python apparently cannot calculate trigonometric functions for np.arrays
# at once, so we must continue in a loop :-()

# This is for angular separations between pointing and solar AzEl
angSeps=np.zeros(tScan.shape[0])

# Loop over times in the scan.
ti=0
for t in tScan:

    azSun = math.atan2(math.sin(hourAngle[ti]),
                       (math.cos(hourAngle[ti])*math.sin(phi)
                        - math.tan(Dec[ti]*DEG_TO_RAD)*math.cos(phi)))+math.pi

    arg=(math.sin(phi)*math.sin(Dec[ti]*DEG_TO_RAD)
         +math.cos(phi)*math.cos(Dec[ti]*DEG_TO_RAD)*math.cos(hourAngle[ti]))

    elSun = math.asin(arg);

#--- Calculate the angular separation

# Unit direction vector to the solar center
dirSun=np.array([math.cos(elSun)*math.cos(azSun),
                 math.cos(elSun)*math.sin(azSun),
                 math.sin(elSun)])

# Unit direction vector to the target
dirTgt=np.array([math.cos(elTgt[ti])*math.cos(azTgt[ti]),

```

```

        math.cos(elTgt[ti])*math.sin(azTgt[ti]),
        math.sin(elTgt[ti]))

    # The dot product of the unit direction vectors is directly cos(delta).
    # Show it in arcmins.
    angSeps[ti]=180.0*60.0/math.pi*math.acos(np.dot(dirSun,dirTgt))

    ti+=1
#-->endfor(all pointing times in the tgtScan)

plt.plot(tScan,angSeps)

return

#-----

def 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.
    """

    #--- Form the MS selectors: The SPW as data_description_id and the time
    # ranges for the solar-target off-source sub-scans.

    myMsm = msmdtool()
    myMsm.open(vis)

    #- 1. Identify the SPW=4 data (WVR channels) by the data_desc_id
    wvrDataDescID=int(myMsm.datadescids(spw=4)[0])

    print('Data desc ID for SWP 4 = ',wvrDataDescID)

    #- 2. Identify the on-source / off-source subscans by their time range.

    # Get sub-scan info for our tgtScan.
    sub_info=aU.computeDurationOfScan(tgtScan,mymsmd=myMsm,
                                     returnSubscanTimes=True)
    # How many sub-scans do we have?
    nSubScans=sub_info[1]

    # The off-source subscans are just the first and last one. The on-source
    # are in the middle.
    # Note: This is a key to the map/dictionary of subscans and it starts at 1.
    offSubScans=[1,nSubScans]
    onSubScans=range(2,nSubScans)

    # Make and fill a list of time-ranges (as MJDs) for the selected sub-scans.

```

```
# We use the string representation of date-time - that is why sub_info[3] is
# extracted here. This is suitable for ms::msselect() used subsequently -
# see the comments below.
```

```
tRanges=[]
```

```
if (subScans=='on') :
    for oss in onSubScans:
        tRanges.append(sub_info[3][oss])
elif (subScans=='off') :
    for offs in offSubScans:
        tRanges.append(sub_info[3][offs])
```

```
# Make the string-list to be a single comma-separated string.
trString=', '.join(tRanges)
```

```
# We are done with the time-range extraction.
myMsmd.done()
```

```
#-- Use the above identifiers/selectors for filtering the WVR data from MS.
```

```
myMs=mstool()
myMs.open(vis)
```

```
# Initialize the MS selector by SPW=4
myMs.selectinit(datadescid=wvrDataDescID)
```

```
# Select WVRs for a given antenna and the scan number.
myMs.select({'antenna1':antID, 'antenna2':antID, 'scan_number':tgtScan})
```

```
# On demand, refine selection to the sub-scans defined by the time-range.
# Note: We use ms::msselect() (and not ms::select()) here as it allows
# for non-contiguous union of time intervals for the sub-scans
if (trString): myMs.msselect({'time':trString})
```

```
#-- Filtration done, now pick-up and process the data.
```

```
# Extract the (complex) data and their 'coordinates' from the filtered MS.
wvrData=myMs.getdata(["data", "axis_info"], ifraxis=True)
```

```
# Clean-up the tool MS tool.
myMs.reset()
myMs.close()
```

```
# Get the data - they are stored as complex amplitudes.
wvr_cAmp=wvrData["data"]
```

```
# Get their corresponding times as MJD.
wvr_mjd=wvrData["axis_info"]["time_axis"]["MJDseconds"]
```

```
# The data are complex by nature, but in fact having just the real part.
wvrPow=wvr_cAmp.real
```

```
# The 'ifraxis=True' in the ms.getdata() above causes expansion of the antenna
```

```

# axis. In our case of WVRs (spw=4) it is degenerated to dimension=1.
# Let us remove it (its is the 3rd axis of the array).
wvr_pow=np.squeeze(wvrPow,axis=2)

# The 'polarization' axis is degenerated as well for WVRs, let us get rid of
# this dimension, too,
wvr_pow=np.squeeze(wvr_pow,axis=0)

print('WVR pow dims: ',wvr_pow.shape)

# Return the time axis as MJD.
# Return the power as a numpy array of nChans x nIntegs dims.
return (wvr_mjd,wvr_pow)

#-----

def 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  $T_{bs}$  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.
    """

    #-- Get the raw WVR temperatures averaged over the scan.
    time,wvrs=wvr_dataRaw(vis,scan,antID)
    Twvr=np.mean(wvrs,axis=1)

    # Count of WVR cahnnels
    wvrChans=wvrs.shape[0]

    #-- Iteration setup.

    # Half dw for the slope (tau_1eff) estimation.
    dws=0.05
    # N-R iteration stopper: WVR temperature is closer to  $T_{bs}$  than this.
    dT_min=0.001
    # Calculate limiting  $T_{bSky}$  for infinite PWV: It is a constant close to
    # the tropospheric temperature for ALMA array (MeteoCentral).
    TbsInf=atmTool.getTebbSky(spwid=0,nc=0,
                              wh2o=qa.quantity(10000.0,'mm'))['value'][0]

    # Array for the results.
    pwvArr=np.zeros(wvrChans)

    #-- Newton-Raphson iterations: Separetelly per each WVR channel, as they
    # need not to converge synchronously. We skip the channel 0 (WV line core)

```

```

# as it gives outlaid value from the rest channels 1 - 3.

for wC in range(wvrChans):

    # Initial estimate for PWV. 3.0 mm and more is typical for the solar obs.
    w0=3.0
    # Initial dT per channel (arbitrary number greater then dw_min).
    dT=0.5
    # Iteration count.
    itc=0

    while (abs(dT)>dT_min):
        # Calculate the TbSky for current estimate of PWV
        TbsW0=atmTool.getAverageTebbSky(wC,qa.quantity(w0,'mm'))['value'][0]

        # Calculate the slope by stepping a bit forward and backward in PWV
        TbsW0H=atmTool.getAverageTebbSky(wC,qa.quantity(w0+dws,'mm'))['value'][0]
        TbsW0L=atmTool.getAverageTebbSky(wC,qa.quantity(w0-dws,'mm'))['value'][0]

        # The slope has a meaning of effective tau_1.
        # Its values are actually close to tau_1 for correct PWV.
        tau_1eff=-np.log((TbsW0H-TbsInf)/(TbsW0L-TbsInf))/(2.0*dws)

        # How far we are to the solution?
        dT=Twvr[wC]-TbsW0

        # Calculate the shift towards the solution.
        dW=dT/(tau_1eff*(TbsInf-TbsW0))
        # Update the current PWV value
        w0+=dW

        itc+=1

    #--> endwhile(N-R iterations)

    # Insert the converged value to shrinked (skip chan 0) array of results.
    pwvArr[wC]=w0

#--> endfor(WVR channels)

print(pwvArr)

# Use just WVR channels 1 - 3 for reasons described above.
pwvMean=pwvArr[1:].mean()
pwvStd=pwvArr[1:].std()

print("PWV: ",pwvMean,"+/-",pwvStd)

return (pwvMean,pwvStd)

#-----

def 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.

"""

Arrays for PWVs calculated by our method pwvCalc() and those in ASDM_CALWVR.

ourPWVs=np.array([])

sysPWVs=np.array([])

MS definition.

vis='uid__A002_X110fd13_X1099e.ms'

Define the scans: All non-solar

theScans=[1,2,3,4,5,6,7,8,10]

Get antenna list.

myMsmd=msmdtool()

myMsmd.open(vis)

antIDs=myMsmd.antennasids()

antNames=myMsmd.antennanames()

myMsmd.done()

#-- Loop over all selected scans.

for scan **in** theScans:

Set the ATM model for the scan.

myAtm=attool()

setAtmTool(myAtm,'uid__A002_X110fd13_X1099e.ms',4,scan)

#-- Loop over all antennas.

for myAnt **in** antIDs:

Skip the CM antennas, these have no WVRs.

if (antNames[myAnt][:2]=='CM'):

 print("Skipping antenna ",antNames[myAnt]," - no WVR data.")

continue

Report progress for WVR-equipped antennas.

print('\n=====')

print('Calculating scan ',scan,' for antenna ',

 int(myAnt),'<-->',antNames[myAnt])

```

print('=====\\n')

# Calculate PWV by our method; the 2nd item in a tuple is stddev.
myPWV, myPWVerr = pwvCalc(vis, myAtm, scan, int(myAnt))
# Append it to the list of all PWV values.
ourPWVs = np.append(ourPWVs, myPWV)

# Extract the PWV from SYS caltable ASDM_CALWWR (+ interpolation to the
# scan mid-time)
sysPWV = getPWVforAnt(vis, scan, int(myAnt)) ['pwvAvg']
# Append it to the list of all sys-PWV values.
sysPWVs = np.append(sysPWVs, sysPWV)

# A quick check.
print('--- ', myPWV, sysPWV)

#--> endfor(antennas)

# Close the atmosphere tool.
myAtm.done()

#--> endfor(scans)

# Return the PWV collections for further analysis.
return (ourPWVs, sysPWVs)

#-----

def 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 $\langle Twvr \rangle$. Since $dW(t)$ and $dL(t)$ variations should have (by definition) zero time average, three equations

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

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 $\langle L_{sun} \rangle$ 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 $\langle L_{sun} \rangle$ dispersion over the WVR channels.

2. Once optimized $\langle PWV \rangle$ and $\langle L_{sun} \rangle$ 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 dL_{sun}) and use the standard quadratic-form minimizing to find the solution (see the same procedure in `dWdB()` for details). The solutions $dW(t)$ and $dL_{sun}(t)$ 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 $\langle L_{sun} \rangle$, and Step 2 is repeated until near-zero averages are reached.

"""

#-- Static parameters

#- Solar spectrum - in theory it is R-J.

Reference frequency for the solar spectrum - central SPW 4 frequency (in Hz)

`f_wvr=1.8755e11`

Spectral index for the source. `Spix=2.0` corresponds to R-J law.

`spix=2.0`

#- Source-data definition.

`vis='uid__A002_X110fd13_X1099e.ms'`

`solScan=9`

#- Atmosphere static parameters.

Calculate limiting $TbSky$ for infinite PWV: It is a constant close to

the tropospheric temperature for the ALMA array (MeteoCentral).

`TbsInf=myAtm.getTebbSky(spwid=0,nc=0,`
`wh2o=qa.quantity(10000.0,'mm'))['value'][0]`

Initial guess of PWV: Scan-center interpolated ASDM_CALWVR value.

`w0=getPWVforAnt(vis,solScan,antID) ['pwvAvg']`

`print('..... Initial estimate')`

```

print('PWV=',w0)
print('.....')

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

#-- Extract WVR raw temperatures.

# Scan-averaged WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,solScan,antID,'on') # on,off or all
# Skip the saturated WVR channel 0.
Twvr=Twvr[1:,:]
# Average over a scan.
TwvrMean=np.mean(Twvr,axis=1)

# How many time samples are in this scan?
nSamples=Twvr.shape[1]

# How many ATM spws do we have?
nAtmSPWs=myAtm.getNumSpectralWindows()

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumantion spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :- (
nChans=myAtm.getNumChan(0)

# Allocate the per-SPW arrays.

# Prepare arrays for brightness temperature, opacities, etc.
TbsW0=np.zeros(nAtmSPWs)
tau_l=np.zeros(nAtmSPWs)
tau_leff=np.zeros(nAtmSPWs)
trans=np.zeros(nAtmSPWs)

# Sensitivity factors for dW and dL
SW=np.zeros(nAtmSPWs)
SL=np.zeros(nAtmSPWs)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC) ['value'] [0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#-- Step 1: Optimize <PWV> by finding minimum dispersion of <Lsun>

#-General N-R iteration setup for Step 1.

```

```

# Half dw for the estimation of derivatives in the N-R iterations.
dws=0.05
# N-R iteration stopper: We move in PWV space by less than this,
ddW_min=0.01
# Initial displacement in the PWV space. Anything above ddW_min.
ddW=0.5

while (abs(ddW)>ddW_min):

    # Set current estimation of PWV to the ATM model.
    myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

    #-- Loop over ATM spws <--> all WVR channels
    for wC in range(nAtmSPWs):

        #- Sky brightness for current PWV estimate and its logarithmic slope.
        TbsW0[wC]=myAtm.getAverageTebbSky(wC, qa.quantity(w0, 'mm'))['value'][0]

        #- Tbs logarithmic slope w.r.t. changing PWV.
        TbsW0H=myAtm.getAverageTebbSky(wC, qa.quantity(w0+dws, 'mm'))['value'][0]
        TbsW0L=myAtm.getAverageTebbSky(wC, qa.quantity(w0-dws, 'mm'))['value'][0]
        # The slope has a meaning of effective tau_1.
        # Its values are actually close to tau_1 scaled by airmass for correct PWV
        tau_1eff[wC]=-np.log((TbsW0H-TbsInf)/(TbsW0L-TbsInf))/(2.0*dws)

        #- Transmissivity & tau_1

        # Dry zenith opacity - an array per sub-channel.
        tau_dry = np.array(myAtm.getDryOpacitySpec(wC)[1])
        # Water-vapour related 'wet' opacity - an array per sub-channel.
        tau_wet = np.array(myAtm.getWetOpacitySpec(wC)[1]['value'])

        # Wet opacity per 1mm PWV scaled by airmass - mean per SPW.
        tau_1[wC]=airMass*tau_wet.mean()/w0

        # Total opacity - an array per sub-channel.
        opacity=airMass*(tau_wet+tau_dry)

        # Mean transmissivity per SPW, scaled by the solar spectrum
        trans[wC]=np.mean(((wvrFreqs[wC]/f_wvr)**spix)*np.exp(-opacity))

    #-->endfor(ATM SPWs)

    #- Skip the WVR channel 0 - it is saturated. The _S arrays are _S_horten
    TbsW0_S=TbsW0[1:]
    tau_1_S=tau_1[1:]
    tau_1eff_S=tau_1eff[1:]
    trans_S=trans[1:]

    #- Calculate <Lsun> in each WVR channel and its dispersion (stddev).

    # Mean solar illumination <Lsun>=<r*Tsun> per WVR channel.
    Lsun=(TwvrMean-TbsW0_S)/trans_S

```

```

# Calculate slope dLsun/dw
dLdW=-tau_1eff_S* (TbsInf-TbsW0_S)/ trans_S+Lsun*tau_1_S

# Estimate the <Lsun> (a.k.a <r*Tsun>) in case of w0 displaced by +/- dws.
LsunPlus=Lsun+dLdW*dws
LsunMinus=Lsun-dLdW*dws

# Calculate the solar-illumination temperatures dispersions over WVR
# channels: For current w0 and w0 displaced by +/- dws.
Td0=Lsun.std()
TdP=LsunPlus.std()
TdM=LsunMinus.std()

# We expect the temperature dispersion is quadratic in w around its minimum.
# In such case, the distance to its minimum can be calculated as -(y'/y'')
# We calculate y' as a difference, y'' is thresholded by experimentally found
# constant below. Why? The quadratic approximation is actually valid just
# close to the minimum (PWV +/- 0.2 from its optimum value) as found
# by an experiment(Fig. Tsun_disp_vs_w.png), in the larger distances from
# the minimising PWV it is rather linear, with y'' going to zero - see Fig.12
# in PR15, namely the left 'shoulder'.

# First central difference as a proxy to y'.
Td1=(TdP-TdM)/(2.0*dws)
# Second central difference as a proxy to y''.
Td2=(TdP+TdM-2.0*Td0)/(dws*dws)
# Thresholding for the linear 'shoulder' of Tdisp(w)
if (Td2 < 50.0): Td2=147.5

# Calculate the displacement towards the optimum.
ddW=-Td1/Td2

# Update PWV.
w0+=ddW

#--> endwhile(N-R iterations for Step 1)

# Static part (scan-average) of solar illumination averaged over WVR channels.
LsunMean=Lsun.mean()

print('----- After Step 1')
print('PWV=',w0,'L_sun=',LsunMean)
print('LS spec:',Lsun,'LS disp:',Td0)
print('-----')

#-- Step 2: Solve the linear set of equations for dW and dL variations
# Iterate until they have negligible mean value (scan-average).

#- Iteration setup

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01
dLMean_max=0.01

# Initial values for the dW and dL means - anything bigger then the stoppers.
dWtMean=0.5
dLtMean=1.0

```

```

#- Iterations: Minimize the mean value of variations.
while ((abs(dWtMean) > dwMean_max) or (abs(dLtMean) > dlMean_max)):

    #- Recalculate ATM model with updated <PWV> and <Lsun>.

    # Set current estimation of PWV to the ATM model.
    myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

    #-- Loop over ATM spws <--> all WVR channels
    for wC in range(nAtmSPWs):

        #-- All quantities per WVR-channel, spectrally resolved by sub-channels.
        # The _C values are _C_hannelised (spectral). We do not specifically
        # indicate this in case of variables local within the loop.

        #- Sky brightness for current PWV estimate and its logarithmic slope.
        TbsW0_C=myAtm.getTebbSkySpec(wC, qa.quantity(w0, 'mm'))[1]['value']

        TbsW0H=myAtm.getTebbSkySpec(wC, qa.quantity(w0+dws, 'mm'))[1]['value']
        TbsW0L=myAtm.getTebbSkySpec(wC, qa.quantity(w0-dws, 'mm'))[1]['value']
        # The slope has a meaning of effective tau_1, see above.
        tau_1eff_C=-np.log((TbsW0H-TbsInf)/(TbsW0L-TbsInf))/(2.0*dws)

        #- Transmissivity & tau_1

        # Dry zenith opacity
        tau_dry = np.array(myAtm.getDryOpacitySpec(wC)[1])
        # Water-vapour related 'wet' opacity
        tau_wet = np.array(myAtm.getWetOpacitySpec(wC)[1]['value'])

        # Wet opacity per 1mm PWV scaled by airmass.
        tau_1_C=airMass*tau_wet/w0

        # Total opacity (spectral).
        opacity=airMass*(tau_wet+tau_dry)

        # Transmissivity scaled by solar spectrum
        trans_C=((wvrFreqs[wC]/f_wvr)**spix)*np.exp(-opacity)

        #-- Making the SPW averages for the ATM model quantities.

        # Mean sky-brightness temperature for the current PWV=w0.
        TbsW0[wC]=TbsW0_C.mean()
        trans[wC]=trans_C.mean()
        tau_1[wC]=tau_1_C.mean()
        tau_1eff[wC]=tau_1eff_C.mean()

        # Sensitivity factor for PWV variations.
        SW[wC]=np.mean(tau_1eff_C*(TbsInf-TbsW0_C)-LsunMean*trans_C*tau_1_C)
        # Sensitivity factor for solar illumination.
        SL[wC]=trans[wC]

    #-->endfor(wvr channel)

    #- Skip the WVR channel 0 - it is saturated.
    TbsW0_S=TbsW0[1:]

```

```

tau_1_S=tau_1[1:]
tau_1eff_S=tau_1eff[1:]
trans_S=trans[1:]
SW_S=SW[1:]
SL_S=SL[1:]

#- Calculate <Lsun> in each WVR channel and its dispersion (stddev).
# Now just for inspection.

# Scan-averaged solar illumination <Lsun> per WVR channel.
Lsun=(TwvrMean-TbsW0_S)/trans_S
# Its dispersion over the 3 used WVR channels
Td0=Lsun.std()

#- Calculate coefficients of the system in line with Eq. (16) in PR 15.

# RHS: The Twvr with the stationary (scan-averaged) part subtracted.
# Twvr is a matrix nTimeSamples x 3 (=nChans), while the subtracted static
# part is just a 3-vector. Make both to be the same shape using tensor dot.

# A 'unity'-vector in time dimension.
uTime=np.ones(nSamples)

TwvrStatic=TbsW0_S+LsunMean*trans_S

dTwvr=Twvr-np.tensordot(TwvrStatic,uTime,0)
# An alternative setup
#dTwvr=Twvr-np.tensordot(TwvrMean,uTime,0)

#- Set of linear equations for dW and dB: Assembly the system.

# Set the 3-vector of sensitivity factors as rows in the transposed matrix.
AwlT=np.array([SW_S,SL_S])
# Transpose to get the 'direct' matrix with sensitivity factors as columns.
Awl=np.transpose(AwlT)

# Set the WVR variations directly as the RHS vector.
Cwl=dTwvr

#- The system is overdetermined, make the least-square minimizer

# Quadratic-form matrix 2x2 (we have two variables).
AK=np.matmul(AwlT,Awl)

# Modified RHS (nTimeSamples x 2)
CK=np.matmul(AwlT,Cwl)

#- Solve the nVars x nVars set of equations, using Cramer's rule.

A11=AK[0,0]
A12=AK[0,1]

A21=AK[1,0]
A22=AK[1,1]

```

```

C1=CK[0]
C2=CK[1]

iD=1.0/ (A11*A22-A12*A21) # Inverse determinant of the system.

# Time series of the dPWV variations and dLsol variations as a result.
dWt=iD* (A22*C1-A12*C2)
dLt=iD* (A11*C2-A21*C1)

# Calculate the time-averages of the above variations - these control the
# iteration stop.
dWtMean=dWt.mean()
dLtMean=dLt.mean()

print('dW mean: ', dWtMean, 'dLsun mean:', dLtMean)

# Update the static <PWV> and <Lsun>
w0+=dWtMean
LsunMean+=dLtMean

#--> endwhile(non-zero-means)

print('==== After Step 2')
print('PWV=', w0, 'L_sun=', LsunMean)
print('LS spec:', Lsun, 'LS spec_mean +/- disp:', Lsun.mean(), ' +/- ', Td0)
print('====')

#-- Prepared data for export.

# For Lsun add the mean value to the variations always
LsunFull=LsunMean+dLt

# For PWV add the mean just on request.
if (fullPWV): dWt+=w0

# Return the PWV and Lsun variations in time as a tuple.
return (wTime, dWt, LsunFull)

#-----

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

- We do not have instrumental vars (gains/ P_{inst}), 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.
- $T_{sun0}(t)$, representing just the part of solar radiation scattered to WVR, is typically 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.

"""

#--- Static parameters

#- Dirs

Source-data definition. Set static because we are MS-specific routine.

`vis='uid__A002_X110fd13_X1099e.ms'`

ATM scan preceding the solar scan 9 is used always.

`atmScan=8`

Output/diagnostics plots.

`plotDir='./wvr_decomp'`

#- Tools, global data

Flags used to check, whether we should close the tool later.

`ownMsmd=False`

if (not myMsmd):

 # Open the msmd tool.

`myMsmd=msmdtool()`

`myMsmd.open(vis)`

`ownMsmd=True`

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

if (not acdLoads):

`acdLoads=aU.getLoadTemperatures(vis, doplot = False,`
 `warnIfNoLoadTemperatures = True,`
 `mymsmd=myMsmd)`

#- Target

```

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.3 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightenning correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature corrected to limb brightenning - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu) # kK

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
# WVR has, unlike the sci-receiver, on-axis recording, may have better eta_F.
eta_F=0.9

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0, 'mm'))

# Airmass for the solar scan.

```

```

airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,'mm @airMass = ',airMass)
print('Estimated T_B(full Sun) = ',1000.0*Tsun00,'K') # scale to Kelvins.
print('.....')

# The MS metadata has been read-out, we may close the (local) tools.
if (ownMsmd): myMsmd.done()

#-- Extract WVR raw temperatures.

# Raw WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,tgtScan,antID,ssType)

# Average over time.
TwvrMean=np.mean(Twvr,axis=1)

# How many time samples are in this scan?
nSamples=Twvr.shape[1]

# How many ATM (i.e. in the model) spws do we have? In this case it is
# the same as a number of WVR channels.
nAtmSPWs=myAtm.getNumSpectralWindows()

# How many equations do we actually have per block of 1 time-sample?
# Some WVR channels may be flagged (typically, the channel 0 is saturated).
eqBlockSize=nAtmSPWs-len(flaggedWVRchannels)

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumantion spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :-(
nChans=myAtm.getNumChan(0)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC) ['value'][0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC) ['value'][0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#-- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).
# Note: In this case, when we have 1-to-1 correspondence between the model
# and data SPWs, we actually need not to store the per-spw values to arrays.
# But we keep the structure of sci-SPW approach in dWdB_TBs() and dWdBdG().

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.

```

```

# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,nChans))
dTdW=np.zeros((nAtmSPWs,nChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,nChans))
tau_l=np.zeros((nAtmSPWs,nChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(wvrFreqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
#   Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01    #mm
dbMean_max=0.001   # kiloKelvin

# Initial values for the dW and dL means - anything bigger then the stoppers.
dW_mean=0.5
dB_mean=0.5

# Iteration counter.
itCount=0

# Set the starting values for iterations.
w0=pwv0
# From dWdLs() we estimate the solar contamination ~100K = 0.1kK in the
# on-source subscans. The off-source should be contamination-free.
if (ssType=='on'):
    Tsun0=0.1 # kiloKelvins
else:
    Tsun0=0.0 # no illumination

#-- Iterations: Minimize the mean value of variations.

while(((abs(dW_mean) > dwMean_max) or (abs(dB_mean) > dbMean_max))
      and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # Our Jacobian contains just N-blocks for [dW,dT] 'natural' variables.
    JJ=np.zeros((eqBlockSize,2)) # 2 vars repeating in non-flagged wvrChans
    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))

    # Solution residua for each used WVR channel.
    resT={}
    # Mapping the WVR channel to the index of equation in the system.
    spw2row={}

```

```

#- Re-calculate ATM model with updated <PWV>

# Set current estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

# Count the actual number of equations - some WVR channels may be flagged.
eqCount=0

#- Loop over 4 ATM spws - these directly correspond to 4 WVR channels
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001* Tebb

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value']
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0- dws, 'mm'))[1]['value']

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001* (TebbH- TebbL)/ (2.0* dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)

    # Transmissivity (spectral).
    trans[atmSPW]=np.exp(- opacity)
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_1[atmSPW]=(airMass/ w0)*tau_wet

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Model + antenna efficiencies combined.

    # Sensitivity factor for PWV.
    SW=(- eta_1*eta_F* Tsun0* np.mean(trans_M* tau_1M* SF_M)

```

```

    +(1.0-eta_F+eta_F*eta_l)*np.mean(dTdW_M)

# Sensitivity factor for the (apparent) solar illumination temperature.
SB=eta_l*eta_F*np.mean(trans_M*SF_M)

# Model T_sky as viewed by the receiver with uniform spectral gain.
Tsky=(eta_l*eta_F*Tsun0*np.mean(trans_M*SF_M)
      +(1.0-eta_F+eta_F*eta_l)*np.mean(TbsW0_M)
      +eta_F*(1.0-eta_l)*Tamb)

#- Build the equations only for the non-flagged channels.
if (atmSPW not in flaggedWVRchannels):

    #- Build the matrix: Only the natural (dW & dB sensitivities) block.

    # Set the sensitivity factors for the current ATM SPW as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    JJ[eqCount]=np.array([SW,SB])

    #- Build the RHS vector: Target data

    # RHS: Twvr(t) with the stationary (in the working point) ATM-based part
    # subtracted. Rescale Twvr to kiloKelvin.
    dTw=0.001*Twvr[atmSPW]-Tsky

    # Set it as the RHS component.
    CC[eqCount]=dTw

    # Residua of sky temperature for mean PWV and Tsun (operation point).
    resT[atmSPW]=dTw.mean()

    # Mapping between ATM SPW and equation number.
    spw2row[atmSPW]=eqCount
    eqCount+=1

else: print('Ignoring flagged ',atmSPW,' WVR channel.')
#--> endif(not flaggged)

#--> endfor(ATM SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Ooops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPWs/polarization channels.")
    return False

#-- Solution of the system of linearised equations.

# Transposed matrices.
JJ_T=np.transpose(JJ)

# Quadratic-form matrix 2x2 (we have two variables).
J_KK=np.matmul(JJ_T,JJ)

# Modified RHS (nVars x nTimeSamples)

```

```

C_KK=np.matmul(JJ_T,CC)

if (solIllumination):
    # We allow for variable solar illumination and need to solve it, too.
    # Solve the nVarsx x nVars set of so called normal equations.
    # C_KK represents the nTimeSamples column vectors, the solution is
    # of the form 2 x nTimeSamples.
    varVec=np.linalg.solve(J_KK,C_KK)

else:
    # The off-subscans are 2 degs of the Sun, the illumination should be zero.
    C_KK[1]=0.0
    varVec=C_KK/ J_KK[0,0]
#--> endif(on/off subscan)

# Decompose the solution vector to our components dW, dBxx, and dByy.

dW=varVec[0]
dB=varVec[1]

#- All variations should tend to zero means.

dW_mean=dW.mean()
dB_mean=dB.mean()

# Update stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0+=dB_mean

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and sol. illumination after completed iteration.
print('..... After iteration no. ',itCount)
print('Mean PWV for the scan,',tgtScan,' w0 = ',w0,"mm"
      ", mean Tsun(illumination) = ",1000.0*Tsun0,"K") # scale to Kelvins

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
resTArr=np.array(list(resT.values()))

print("res T: ", resT,"mean +/- RMS: ",resTArr.mean(),"+/-",resTArr.std())

print('.....')

#--> endwhile(non-zero-means)

#-- Diagnostic plots.
# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

#-Set the plotting device.

```

```

plt.rcParams["figure.figsize"] = (15,15)
# X-axis label
plt.xlabel('Time [MJDsec]')
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Color-map for WVR-channels/ATM-SPWs 0 - 3.
csline={0: '-r', 1: '-g', 2: '-b', 3: '-k'}

#- Make the *analytical* plots - least-square residuals etc.

# Subtract 'theoretical' and real sky+src temperature variations.
errVec=np.matmul(JJ,varVec)-CC

for spw,eqRow in spw2row.items():
    # Residuals/errors as temperature in Kelvins.
    plt.plot(wTime,1000.0*errVec[eqRow],csline[spw])

#- Save the plots to a file and close.
plt.savefig(plotDir+'/errs_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Make the full *solution* plots: PWV(t) and Tsun(t).

# Add variations to the mean (scan-averaged) values.
# Scale the solar illumination temperature from kiloKelvins to Kelvins.
pwv=w0+dW #mm
Tsun=1000.0*(Tsun0+dB) # K

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff,(sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID))

# Left panel: Fast variations - apparent target brightness
sp1.plot(wTime,Tsun,'-r')

# Right panel: Slow variations - total PWV (compression + humidity).
sp2.plot(wTime,pwv,'-g')

plt.savefig(plotDir+'/dWdB/dWdB_ant_'+str(antID)+'.png')
plt.show()
plt.close()

#-- Export resulting data in 'conforming' format.

return {'time':wTime,'pwv':pwv,'T_B':Tsun,'J':JJ,'C':CC}

#-----
def 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().

```
"""
```

```
#--- Static parameters
```

```
#- I/O
```

```
# Input data from the ATM scan preceding the solar scan.
```

```
atmScan=tgtScan-1
```

```
# Output/diagnostics plots.
```

```
plotDir='./wvr_decomp_G'
```

```
if (not os.path.exists(plotDir)): os.makedirs(plotDir)
```

```
if (not os.path.exists(plotDir+'dWdB')): os.makedirs(plotDir+'dWdB')
```

```
#- Tools, global data
```

```
# Flags used to check, whether we should close the tool later.
```

```
ownMsmd=False
```

```
if (not myMsmd) :
```

```
    # Open the msmd tool.
```

```
    myMsmd=msmdtool()
```

```
    myMsmd.open(vis)
```

```
    ownMsmd=True
```

```
# Create our dictionary with ACD load temperatures (ambient, hot) if needed.
```

```
if (not acdLoads) :
```

```
    acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                     warnIfNoLoadTemperatures = True,
                                     mymsmd=myMsmd)
```

```
#- Target
```

```

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.3 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff(arcsec, incl. cos(delta)): 822 155.8
'''
x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/(Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0-(x_h**2+y_h**2)/r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature corrected to limb brightening - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu) # kK

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
# WVR has, unlike the sci-receiver, on-axis recording, may have better eta_F.
eta_F=0.9

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# How strongly we insist on calibration solution G=1.0, P_inst=0.0.
# See comment in dWdBdG().

```

```

# The used value has been found experimentally via dWdLs3_calWeight().
calWeight=7.0

#- Atmosphere

# How many ATM (i.e. in the model) spws do we have? In this case it is
# the same as the number of the WVR channels.
nAtmSPWs=myAtm.getNumSpectralWindows()

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0, 'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,'mm @airMass = ',airMass)
print('Estimated T_B(full Sun) = ',1000.0*Tsun00,'K') # scale to Kelvins.
print('.....')

# We do not need the msmd tool anymore.
if (ownMsmd): myMsmd.done()

#-- Extract WVR raw temperatures.

# Raw WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,tgtScan,antID,ssType)

# Average over time.
TwvrMean=np.mean(Twvr,axis=1)
# Std devs per channel.
TwvrRMS=np.std(Twvr,axis=1)

#-- Flag and remove clear outliers.

# Find all array indices, for which data in at least one of the valid
# WVR channels are out of 3 sigma interval.
validWvrChans=np.delete(range(nAtmSPWs),flaggedWVRchannels)

for SPW in validWvrChans:
    # Outliers ( > 3 sigma) in the current SPW.
    badData=np.flatnonzero(abs(Twvr[SPW]-TwvrMean[SPW])>3.0*TwvrRMS[SPW])
    if (badData.size):
        print('Flagged in WVR channel no.',SPW,': Data in times ',wTime[badData],
              ', bad temperatures T=',Twvr[SPW,badData])

    # Delete data at bad-data indices in *all* WVR channels, so that the data
    # remain homogeneous in time. Make the time axis conformant.
    Twvr=np.delete(Twvr,badData,axis=1)
    wTime=np.delete(wTime,badData)
#--> endfor(valid WVR chans)

#-- Find dimensions of the system.

```

```

# How many time samples are in this scan (after flagging)?
nSamples=Twvr.shape[1]

# How many equations do we actually have per block of 1 time-sample?
# Some WVR channels may be flagged (typically, the channel 0 is saturated).
eqBlockSize=nAtmSPWs-len(flaggedWVRchannels)

#- Relative gains and instr. offsets - one per WVR channel, no polarization.

# Gains to be initialised to their expected value of 1.0.
Grw=np.ones(nAtmSPWs)
# Offsets should be close to 0.0.
Piw=np.zeros(nAtmSPWs)

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumination spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :-(
nChans=myAtm.getNumChan(0)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC) ['value'] [0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#-- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).
# Note: In this case, when we have 1-to-1 correspondence between the model
# and data SPWs, we actually need not to store the per-spw values to arrays.
# But we keep the structure of sci-SPW approach in dWdB_TBs() and dWdBdG().

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,nChans))
dTdW=np.zeros((nAtmSPWs,nChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,nChans))
tau_1=np.zeros((nAtmSPWs,nChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(wvrFreqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.

```

```

dws=0.05

# Stoppers for iterative solution: We should get less then these values.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin

# Initial values for the dW and dL means - anything bigger then the stoppers.
dW_mean=0.5
dB_mean=0.5

# Iteration counter.
itCount=0

# Set the starting values for iterations.
w0=pwv0
# From dWdLs() we estimate the solar contamination ~100K = 0.1kK in the
# on-source subscans. The off-source should be contamination-free.
if (ssType=='on'):
    Tsun0=0.1 # kiloKelvins
else:
    Tsun0=0.0 # no illumination

#-- Iterations: Minimize the mean value of variations.

while ((abs(dW_mean) > dwMean_max) or (abs(dB_mean) > dbMean_max))
    and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-blocks: Jacobian-part for the [dW,dT] 'natural' variables.
    if (solIllumination):
        # Solve for both dW and dB - we allow for varying solar illumination.
        J_wb=np.zeros((eqBlockSize,2)) # 2 vars repeating in non-flagged wvrChans
    else:
        # Solar illumination should be zero, solve for dW solely.
        J_wb=np.zeros((eqBlockSize,1)) # 1 var repeating in non-flagged wvrChans

    # G-block matrix: J-part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each WVR channel.

    # A-block matrix: J-part for [dG,dP_inst] for WVR "calibration" equations.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, amb.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, hot.

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))

    # RHS-vector 'appendix' - amb. & hot 'calibration powers' (time-averaged).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

    # Solution residua for each used WVR channel.
    resT={}
    # Mapping the WVR channel to the index of equation in the system.

```

```

spw2row={}

#- Re-calculate ATM model with updated <PWV>

# Set current estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

# Count the actual number of equations - some WVR channels may be flagged.
eqCount=0

#- Loop over 4 ATM spws - these directly correspond to 4 WVR channels
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001*Tebb

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value']
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0-dws, 'mm'))[1]['value']

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001*(TebbH- TebbL)/(2.0*dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)

    # Transmissivity (spectral).
    trans[atmSPW]=np.exp(- opacity)
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_1[atmSPW]=(airMass/ w0)*tau_wet

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Model + antenna efficiencies combined.

    # Sensitivity factor for PWV.

```

```

SW=(- eta_l* eta_F* Tsun0* np.mean(trans_M* tau_1M* SF_M)
    +(1.0- eta_F+ eta_F* eta_l)* np.mean(dTdW_M))* Grw[atmSPW]

# Sensitivity factor for the (apparent) solar illumination temperature.
SB=eta_l* eta_F* np.mean(trans_M* SF_M)* Grw[atmSPW]

# Model T_sky as viewed by the receiver with uniform spectral gain.
Tsky=(eta_l* eta_F* Tsun0* np.mean(trans_M* SF_M)
      +(1.0- eta_F+ eta_F* eta_l)* np.mean(TbsW0_M)
      +eta_F* (1.0- eta_l)* Tamb)

#- Build the equations only for the non-flagged channels.
if (atmSPW not in flaggedWVRchannels):

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the sensitivity factor(s) for the current ATM SPW as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    if (solIllumination):
        # Solve for both dW & dB independently.
        J_wb[eqCount]=np.array([SW,SB])
    else:
        # Only dW is allowed to vary.
        J_wb[eqCount]=np.array([SW])

    #- Build the matrix: Instrumental (Grw & Piw sensitivities) block.

    # We have to prepend a proper offset for the Grw and Piw vars as
    # they are independendt for each SPW and polarization.
    bOffset=2*eqCount

    J_gp[eqCount,bOffset]=1.0
    J_gp[eqCount,bOffset+1]=Tsky

    #- Build the matrix: ACD powers (amb & hot) 2-block.

    J_cdA[eqCount,bOffset]=1.0
    J_cdA[eqCount,bOffset+1]=Tamb

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=1.0
    J_cdH[eqCount,bOffset+1]=Thot

    #- Build the RHS vector: WVR data

    # RHS: Twvr(t) with the stationary (in the working point) ATM-based part
    # subtracted. Rescale Twvr to kiloKelvin.
    dTw=0.001* Twvr[atmSPW]- Tsky

    # Set it as the RHS component.
    CC[eqCount]=dTw

    #- Ammend the RHS vector: Calibration data.

    # Ambient-load power with its working-point part subtracted.

```

```

CCA[eqCount]=Tamb-Piw[atmSPW]-Grw[atmSPW]*Tamb
# Hot-load power with its working-point part subtracted.
CCH[eqCount]=Thot-Piw[atmSPW]-Grw[atmSPW]*Thot

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resT[atmSPW]=dTtw.mean()

# Mapping between ATM SPW and equation number.
spw2row[atmSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',atmSPW,' WVR channel.')
#--> endif(not flagged)

#--> endfor(ATM SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPWs/polarization channels.")
    return False

#- Solution of the system of linearised equations. See dWdBdG() for
#  explanation.

#- Calculate the small-block matrices and their inversions needed for
#  grand-system inversion.

# Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
J_cdA_T=np.transpose(J_cdA).copy()
J_cdH_T=np.transpose(J_cdH).copy()

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T,J_cdH_T]) from already scaled-up transposed sub-blocks.
J_cd=np.vstack([J_cdA_T,J_cdH_T]).copy()
J_cd_T=calWeight*np.transpose(J_cd)

#- Base-matrix extensions.

```

```

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T,J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T,J_gp)+np.matmul(J_cd_T,J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(J_K_inv,Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc-nSamples*np.matmul(Lb,AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv,LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=-np.matmul(AinvRb,Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1),(-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e, multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)      # 2 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)  # 2 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum)      # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (2 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

```

```

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv,C_K_wb) # 2 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
# PWV variations present always.
dW=varNat[0]

# dB variations only in case of solar illumination, otherwise they are zeros
if (solIllumination):
    dB=varNat[1]
else:
    dB=np.zeros(nSamples)

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per WVR channel.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errors for the over-determined set of eqs.

# err=A.x-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part.
xiSci=np.sum(errVec*errVec)

# err=A.x-b: Calibration-data part.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the cal-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)

# Grand-total Xi-square normalized to the number of equations (sci/acd).
xiSquare=xiSci/nSamples+xiACD/calWeight

#- All variations should tend to zero means.

dW_mean=dW.mean()
dB_mean=dB.mean()

# Update stationary part of PWV and solar T_B by means of their variations.
w0+=dW_mean
Tsun0+=dB_mean

```

```

# Update also the Piw and Grw by the found differences.
for atmSPW,eqCount in spw2row.items():
    # Update the P_inst and G.
    Piw[atmSPW]+=dPinst[eqCount]
    Grw[atmSPW]+=dGinst[eqCount]
#--> endfor(SPWs with valid XX pol)

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and sol. illumination after completed iteration.
print('..... After iteration no. ',itCount)
print('Mean PWV for the scan,',tgtScan,' w0 = ',w0,"mm"
      ", mean Tsun(illumination) = ",1000.0*Tsun0,"K") # scale to Kelvins

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
resTArr=np.array(list(resT.values()))

print("res T: ", resT,"mean +/- RMS: ",resTArr.mean(),"+/-",resTArr.std())

print('.....')

#--> endwhile(non-zero-means)

#-- Diagnostic plots.
# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

# We may have a set of per-scan figures: Distinguish by scan ID in filename.
scID='_{:02d}'.format(tgtScan)

#-Set the plotting device.
plt.rcParams["figure.figsize"] = (15,15)
# X-axis label
plt.xlabel('Time [MJDsec]')
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID)+' , scan '+str(tgtScan))

# Color-map for WVR-channels/ATM-SPWs 0 - 3.
csline={0: '-r', 1: '-g', 2: '-b', 3: '-k'}

#- Make the *analytical* plots - least-square residuals etc.

for spw,eqRow in spw2row.items():
    # Residuals/errors as temperature in Kelvins.
    plt.plot(wTime,1000.0*errVec[eqRow],csline[spw])

#- Save the plots to a file and close.

plt.savefig(plotDir+'/errs_ant_'+str(antID)+'_scan'+scID+'.png')

```

```

plt.show()
plt.close()

#-- Make the full *solution* plots: PWV(t) and Tsun(t).

# Add variations to the mean (scan-averaged) values.
# Scale the solar illumination temperature from kiloKelvins back to Kelvins.
pwv=w0+dW #mm
Tsun=1000.0*(Tsun0+dB) # K

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID)+' , scan '+str(tgtScan))

# Left panel: Fast variations - apparent target brightness
sp1.plot(wTime,Tsun,'-r')

# Right panel: Slow variations - total PWV (compression + humidity).
sp2.plot(wTime,pwv,'-g')

plt.savefig(plotDir+'dWdB/ant_'+str(antID)+'_scan_'+scID+'.png')
plt.show()
plt.close()

#-- Show, how much the instrumental variables G and P_inst have changed.

print("Corrected relative gains [dimension-less]: ", Grw)
print("Corrected instrumental offsets [K]:", 1000.0*Piw)

# Print total quadratic error for the variables.
print("\n===== Xi-squares ===== SCI: ",xiSci,
      ", ACD:",xiACD,
      ", Total:", xiSquare,"\n")

#-- Export the resulting data in the 'conforming' format.

return {'time':wTime,'pwv':pwv,'T_B':Tsun,'J':J_wb,'C':CC}

#-----
def dWdLs4(vis,myAtm,antID,tgtScan,
          flaggedWVRchannels=[],ssType='',solIllumination=True,
          myMsmd=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 Tsun=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 T_{sun} 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 T_{sun} 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 $m\text{Weight}$.

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).

"""

#--- Static parameters

#- I/O

*# Input data from the ATM scan preceding the target scan.
Note: Here it has a little sense so far - we use calibrated T_{wvr} in Kelvins
and the nominal gain G_{wr} thus $G_{\text{wr}}=1.0$. We use it only for finding T_{amb}
and T_{hot} temperatures in order to keep the structure of equations conformant
the that in `sWdBdG2()`. However, we may later try expolarion of the real
ATM scans, which have also `CALIBRATE_WVR#AMBIENT/HOT` intents. In those
cases, we have to be more carefull and identify the actual ATM scan
relevant to the **tgtScan***

`atmScan=tgtScan-1`

Output/diagnostics plots.

`plotDir='./wvr_decomp_G'`
`if (not os.path.exists(plotDir)): os.makedirs(plotDir)`
`if (not os.path.exists(plotDir+'dWdB')): os.makedirs(plotDir+'dWdB')`

#- Tools, global data

Flags used to check, whether we should close the tool later.

`ownMsmd=False`

Open own tools if 'global' not provided

`if (not myMsmd):`
 # Open the msmd tool.
 `myMsmd=msmdtool()`
 `myMsmd.open(vis)`
 `ownMsmd=True`

Create our dictionary with ACD load temperatures (ambient, hot) if needed.

`if (not acdLoads):`
 `acdLoads=aU.getLoadTemperatures(vis, doplot = False,`
 `warnIfNoLoadTemperatures = True,`
 `mymsmd=myMsmd)`

#- Antenna

`antNames=myMsmd.antennanames()`

```

myAntName=antNames[antID]
antType=myAntName[:2]

# Map the antenna efficiencies by antenna type. As per investigation from
# merged T_sun temperatures form 2023 and 2026 data (project_22_25_Tb.ods).
eta_l_map={'CM':0.941, 'DA':0.947, 'DV':0.956}
eta_F_map={'CM':0.891, 'DA':0.897, 'DV':0.906}

# Rear-spillover + ohmic antenna efficiency - see project_22_25_Tb.ods
eta_l=eta_l_map[antType]
# Forward-spillover antenna efficiency - see project_22_25_Tb.ods
eta_F=eta_F_map[antType]

# We have two exception that seem to have systematically lower
# antenna efficiencies: DA64 and DV08 - see project_22_25_Tb.ods
if myAntName=='DA64':
    eta_l=0.929
    eta_F=0.880
if myAntName=='DV08':
    eta_l=0.943
    eta_F=0.894

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# How strongly we insist on calibration solution G=1.0, P_inst=0.0.
# See comment in dWdBdG2().
# The used value has been found experimentally via dWdLs4_weights().
# Search mark in dWdLs4(): CW_2
calWeight=20.0

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.300 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

"""
# *** This part is observation specific - need to read ephemeris !!!

```

```

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff(arcsec, incl. cos(delta)): 822   155.8
'''

x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.
'''

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature corrected to limb brightening - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu) # kK

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

# Solar illumination. May be either zero or unbound, depending on the
# solIllumination parameter setting (T/F).
T_tgt=0.0

if (solIllumination):
    # Do not apply Tsun=0 condition, we allow for unbound illumination.
    # If mWeight is set to zero, the T_tgt value itself is irrelevant.
    mWeight=0.0
else:
    # We are pointing far off the Sun. Enforce zero illumination by adding
    # the T_tgt=0.0 equation with the appropriate weight.
    # Search mark in dWdLs4(): MW_2
    mWeight=1.0

#- Atmosphere

# How many ATM (i.e. in the model) spws do we have? In this case it is
# the same as the number of the WVR channels.
nAtmSPWs=myAtm.getNumSpectralWindows()

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

# Set initial estimation of PWV to the ATM model.

```

```

myAtm.setUserWH2O(qa.quantity(pwv0, 'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan, ',tgtScan, ' w0 = ',pwv0, 'mm @airMass = ',airMass)
print('Estimated T_B(full Sun) = ',1000.0*Tsun00, 'K') # scale to Kelvins.
print('.....')

# The MS metadata has been read-out, we may close the (local) tools.
if (ownMsmid): myMsmid.done()

#-- Extract WVR raw temperatures.

# Raw WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,tgtScan,antID,ssType)

# Average over time.
TwvrMean=np.mean(Twvr,axis=1)
# Stddevs per channel.
TwvrRMS=np.std(Twvr,axis=1)

#-- Flag and remove clear outliers.

# Find all array indices, for which data in at least one of the valid
# WVR channels are out of 3 sigma interval.
validWvrChans=np.delete(range(nAtmSPWs),flaggedWVRchannels)

for SPW in validWvrChans:
    # Outliers ( > 3 sigma) in the current SPW.
    badData=np.flatnonzero(abs(Twvr[SPW]-TwvrMean[SPW])>3.0*TwvrRMS[SPW])
    if (badData.size):
        print('Flagged in WVR channel no.',SPW,': Data in times ',wTime[badData],
              ', bad temperatures T=',Twvr[SPW,badData])

        # Delete data at bad-data indices in *all* WVR channels, so that the data
        # remain homogeneous in time. Make the time axis conformant.
        Twvr=np.delete(Twvr,badData,axis=1)
        wTime=np.delete(wTime,badData)
#--> endfor(valid WVR chans)

#-- Find dimensions of the system.

# How many time samples are in this scan (after flagging)?
nSamples=Twvr.shape[1]

# How many equations do we actually have per block of 1 time-sample?
# Some WVR channels may be flagged (typically, the channel 0 is saturated).
eqBlockSize=nAtmSPWs-len(flaggedWVRchannels)

# Pre-calculated frequently repeating factor: Inverse number of time samples.
nsInv=1.0/nSamples

#- Relative gains and instr. offsets - one per WVR channel, no polarization.

# Gains to be initialised to their expected value of 1.0.

```

```

Grw=np.ones(nAtmSPWs)
# Offsets should be close to 0.0.
Piw=np.zeros(nAtmSPWs)

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumantion spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :- (
nChans=myAtm.getNumChan(0)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC) ['value'] [0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#-- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).
# Note: In this case, when we have 1-to-1 correspondence between the model
# and data SPWs, we actually need not to store the per-spw values to arrays.
# But we keep the structure of sci-SPW approach in dWdB_TBs() and dWdBdG().

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,nChans))
dTdW=np.zeros((nAtmSPWs,nChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,nChans))
tau_1=np.zeros((nAtmSPWs,nChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(wvrFreqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#-- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less then these values.
# We used those as weights in calculation of epsilon - the normalised
# inter-iteration change of operation-point at the very end of each iteration.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin
dG_max=0.001 # Relative scale dG/G
dTinst_max=0.001 # dPinst/G in kiloKelvins

# Iteration counter.

```

```

itCount=0

# Normalized inter-iteration change of operation point - its initial value.
# Anything bigger than 1.0
epsilon=10.0

# Set the starting values / an initial guess for iterations.
w0=pwv0
# From dWdLs() we estimate the solar contamination ~100K = 0.1kK in the
# on-source subscans. The off-source should be contamination-free.
if (ssType=='on'):
    Tsun0=0.1 # kiloKelvins
else:
    Tsun0=0.0 # no illumination

#-- Iterations: Minimize the mean value of variations.

while( (epsilon>1.0) and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-blocks: Jacobian-part for the [dW,dT] 'natural' variables.
    J_wb=np.zeros((eqBlockSize,2)) # 2 vars repeating in non-flagged wvrChans
    # G-block matrix: J-part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each WVR channel.

    # A-block matrix: J-part for [dG,dP_inst] for WVR "calibration" equations.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, amb.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, hot.
    # M-block matrix: Jacobian part for [dW,dT] in array-mean eqs.
    # This matrix is constant and can be defined already here.
    J_M=np.array([[0.0, nsInv]])

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))

    # RHS-vector 'appendix 1' - amb. & hot 'calibration powers' (time-averaged).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

    # RHS-vector 'appendix 2' - array-mean values of the target T_B
    CCM=np.zeros((1,1))

    # Solution residua for each used WVR channel.
    resT={}
    # Mapping the WVR channel to the index of equation in the system.
    spw2row={}

    #- Re-calculate ATM model with updated <PWV> and solar illumination.

    # Set current estimation of PWV to the ATM model.
    myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

```

```

# Count the actual number of equations - some WVR channels may be flagged.
eqCount=0

#- Loop over 4 ATM spws - these directly correspond to 4 WVR channels
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001* Tebb

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value']
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0- dws, 'mm'))[1]['value']

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001* (TebbH- TebbL)/ (2.0* dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass* (tau_wet+tau_dry)

    # Transmissivity (spectral).
    trans[atmSPW]=np.exp(- opacity)
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_1[atmSPW]=(airMass/ w0)* tau_wet

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Model + antenna efficiencies combined.

    # Sensitivity factor for PWV.
    SW=(- eta_1* eta_F* Tsun0* np.mean(trans_M* tau_1M* SF_M)
        +(1.0- eta_F+eta_F* eta_1)* np.mean(dTdW_M)) * Grw[atmSPW]

    # Sensitivity factor for the (apparent) solar illumination temperature.
    SB=eta_1* eta_F* np.mean(trans_M* SF_M) * Grw[atmSPW]

    # Model T_sky as viewed by the receiver with uniform spectral gain.
    Tsky=(eta_1* eta_F* Tsun0* np.mean(trans_M* SF_M)

```

```

+ (1.0-eta_F+eta_F*eta_l)*np.mean(TbsW0_M)
+eta_F*(1.0-eta_l)*Tamb)

# Build the equations only for the non-flagged channels.
if (atmSPW not in flaggedWVRchannels):

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the sensitivity factor(s) for the current ATM SPW as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    J_wb[eqCount]=np.array([SW,SB])

    #- Build the matrix: Instrumental (Grw & Piw sensitivities) block.

    # We have to prepend a proper offset for the Grw and Piw vars as
    # they are independentt for each SPW and polarization.
    bOffset=2*eqCount

    J_gp[eqCount,bOffset]=1.0
    J_gp[eqCount,bOffset+1]=Tsky

    #- Build the matrix: ACD powers (amb & hot) 2-block.

    J_cdA[eqCount,bOffset]=1.0
    J_cdA[eqCount,bOffset+1]=Tamb

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=1.0
    J_cdH[eqCount,bOffset+1]=Thot

    #- Build the RHS vector: WVR data

    # RHS: Twvr(t) with the stationary (in the working point) ATM-based part
    # subtracted. Rescale measured WVR temperatures to kiloKelvin.
    dTw=0.001*Twvr[atmSPW]-Piw[atmSPW]-Grw[atmSPW]*Tsky

    # Set it as the RHS component.
    CC[eqCount]=dTw

    #- Ammend the RHS vector: Calibration data.

    # This is a sort of trick. It would be natural to add equations
    # Grw[atmSPW]=1.0, Piw[atmSPW]=0.0. But we like to keep the structure
    # of dWdBdG2() equations (with all the lin-algebra apparatus for its
    # 'pain-less' solution), so we encoded the trivial eqs. above into
    # form Tamb=Grw*Tamb+Piw, Thot=Grw*Thot+Piw, which has exactly the
    # same solution Grw=1.0, Piw=0.0. This form is co-variant with that
    # used in PWV proxy calculation in dWdBdG2().

    # Ambient-load power with its working-point part subtracted.
    # Note: Unlike Twvr, the Tamb and Thot are in kiloKelvins already.
    CCA[eqCount]=Tamb-Piw[atmSPW]-Grw[atmSPW]*Tamb
    # Hot-load power with its working-point part subtracted.
    CCH[eqCount]=Thot-Piw[atmSPW]-Grw[atmSPW]*Thot

```

```

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resT[atmSPW]=dT_w.mean()

# Mapping between ATM SPW and equation number.
spw2row[atmSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',atmSPW,' WVR channel.')
#--> endif(not flagged)

#--> endfor(ATM SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPWs/polarization channels.")
    return False

#-- Solution of the grand system of linearised equations. See dWdBdG2() for
# explanation.

#- Ammend the RHS vector: Array-mean target brightness temperatures.

# Effective expected target T_B with the model at work-point subtracted.
# In our case, effective T_B is just the part of 'parasitic' illumination.
# For far-off-Sun pointings T_tgt is set to zero. For near-Sun pointings
# we use solIllumination=True and Tsun0 is unbound because mWeight=0.
CCM[0]=T_tgt-Tsun0

#- Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# 1. Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# 2. Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# 3. ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
# We need to copy() here - see the same place in dWdBdG2() for explanation!
J_cdA_T=np.transpose(J_cdA).copy()
J_cdH_T=np.transpose(J_cdH).copy()

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

```

```

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T, J_cdH_T]) from already scaled-up transposed sub-blocks.
J_cd=np.vstack([J_cdA, J_cdH]).copy()
J_cd_T=calWeight*np.transpose(J_cd)

# 4. Array-mean (M-block) related matrices. Incorporate mWeight into M^T, in
# line with what we have done with calWeight for the A^T and H^T matrices.
M_T=np.array([[0.0],
               [nsInv*mWeight]])    # mWeight*M^T

# Pre-calculated nSamples*mWeight*(M^T x M) matrix.
# Like with A^T x A, the M^T x M product is scaled-up by mWeight just once
# (no mWeight squared!). Moreover, we scale it up here yet by nSamples,
# in order to avoid trivial product nsInv*nSamples=1 in the following calcs.
nsMTM=np.array([[0.0, 0.0],
                 [0.0, nsInv*mWeight]])

# Frequently repeating blocks. NM_inv plays a role of modified J_K_inv.
NM=J_K+nsMTM                # J_K+nSamples*mWeight*M^T x M
NM_inv=np.linalg.inv(NM)    # NM^(-1)

# Off-diagonal blocks of the inverted natural sub-matrix (Ainv).
J_M_inv=nsInv*(NM_inv-J_K_inv)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 2) and (2 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T, J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T, J_gp)+np.matmul(J_cd_T, J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(NM_inv, Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc-nSamples*np.matmul(Lb, AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv, LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=J_M_inv-np.matmul(AinvRb, Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.

```

```

Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e, multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)+np.matmul(M_T,CCM) # 2 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)+nSamples*np.matmul(M_T,CCM) # 2 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum) # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (2 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv,C_K_wb) # 2 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
# PWV variations.
dW=varNat[0]
# Illumination/brightness variations: In this approach we may have them, too
dB=varNat[1]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per WVR channel.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errors for the over-determined set of eqs.
# See corresponding part of dWdBdG2() for explanation of error estimation.

```

```

# err=f(x)-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part, normalized per one equation.
xiSci=np.sum(errVec*errVec)/(nSamples*eqBlockSize)

# Xi Squares per SPW:Pol channel. Sum/average only over times.
xiSciPerChan=np.sum(errVec*errVec,axis=1)/nSamples
# Its mean and RMS.
xscMean=xiSciPerChan.mean()
xscRMS=xiSciPerChan.std()
# Find outliers (> 3 sigma). The badData array contains indices of
# those equations, where the agreement between model and observation is
# exceptionally weak.
badData=np.flatnonzero(abs(xiSciPerChan-xscMean)>3.0*xscRMS)

# Show solution errors per WVR channel and mark (later possibly auto-flag)
# the WVR channels with bad data, if we have some.
# We do not have an inverse map to spw2row, so we have to traverse.
for wvrChan,eqRow in spw2row.items():
    print('WVR chan '+str(wvrChan)+' Error: ',
          np.sqrt(xiSciPerChan[eqRow])*1000.0,'K')
    if eqRow in badData: print('\t\t!!! Bad data in WVR chan',str(wvrChan),
                              ':', (xiSciPerChan[eqRow]-xscMean)/xscRMS,
                              'sigma from the mean.')

# err=f(x)-b: Calibration-data part.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the cal-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)
# Normalized per equation - see corresponding-place comment in dWdBdG2().
xiACD/=eqBlockSize

# err=f(x)-b: Array-mean T_B (illumination) part.
errVecMeanTb=-CCM
# Xi square for the averaged illumination part.
# We have just a single equation, thus no normalization is required.
xiTbM=np.sum(errVecMeanTb*errVecMeanTb)

# Grand-total Xi-square normalized to the number of equations for
# the natural/science, ACD, and mean-array sub-blocks.
xiSquare=xiSci+xiACD+xiTbM

#- All variations should tend to zero means.

dW_mean=dW.mean()
dB_mean=dB.mean()

# Update stationary part of PWV and illumination by avgs of their variations
w0+=dW_mean
Tsun0+=dB_mean

# Update also the Piw and Grw by the found differences.
# Find the maximum inter-iteration change for convergence diagnostics.
dT_inst_sup=-1.0
dG_rel_sup=-1.0

```

```

for atmSPW,eqCount in spw2row.items():
    # Update the P_inst and G.
    Piw[atmSPW]+=dPinst[eqCount]
    Grw[atmSPW]+=dGinst[eqCount]

    # Change of inst. power expressed in kiloKelvins.
    dT_inst=abs(dPinst[eqCount]/ Grw[wvrChan])
    if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

    # Change of gain in the relative scale.
    dG_rel=abs(dGinst[eqCount]/ Grw[wvrChan])
    if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(non-flagged WVRchannels/atmSPWs)

#-- Convergence diagnostics / Iteration control.

# Calculate the normalized inter-iteration change of operation point.
# The factor 0.5 comes from sqrt(1/4) as we have 4 'orthogonal' variables.
epsilon=0.5*np.sqrt((dW_mean/ dWMean_max)**2+(dB_mean/ dBMean_max)**2+
                    (dG_rel_sup/ dG_max)**2+(dT_inst_sup/ dTinst_max)**2)

# Increment the iteration counter.
itCount+=1

#- Diagnostic print-outs.

# Updated scan-averaged PWV and sol. illumination after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('Mean PWV for the scan,',tgtScan,' w0 = ',w0,"mm"
      ", mean Tsun(illumination) = ",1000.0*Tsun0,"K") # scale to Kelvins

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale-up back to Kelvins (we calculate in kiloKelvins).
resTArr=1000.0*np.array(list(resT.values()))

print("res T: ", resT,"mean +/- RMS: ",resTArr.mean(),"+/-",resTArr.std())

# Note: Here we measure the Pinst change directly in Kelvin(!), as nominal
# Piw is zero. For gain, we need not to divide by Grw as it is nominally 1.
print("\nInstrumental characteristics shifts during the iteration:")
for atmSPW,eqCount in spw2row.items():
    print("WVR chanNum = ",atmSPW,"\tdP_I =",1000.0*dPinst[eqCount],"K",
          "\tdG/G =",dGinst[eqCount]*100,"%")

print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / End of solution iteration loop.

#-- Diagnostic plots.
# Make a plot comparing the measured fluctuations, their part attributed to
# the intrinsic brightness changes, and error in eqs-system solution.

```

```

# We may have a set of per-scan figures: Distinguish by scan ID in filename.
scID='_{:02d}'.format(tgtScan)
# Antenna ID as a 2-digit string.
antIDstr='{:02d}'.format(int(antID))

#-Set the plotting device.
plt.rcParams["figure.figsize"] = (15,15)
# X-axis label
plt.xlabel('Time [MJDsec]')
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID)+' , scan '+str(tgtScan))

# Color-map for WVR-channels/ATM-SPWs 0 - 3.
csline={0: '-r', 1: '-g', 2: '-b', 3: '-k'}

#- Make the *analytical* plots - least-square residuals etc.

for spw,eqRow in spw2row.items():
    # Residuals/errors as temperature in Kelvins.
    plt.plot(wTime,1000.0*errVec[eqRow],csline[spw])

#- Save the plots to a file and close.

plt.savefig(plotDir+'/errs-scan_'+scID+'_ant_'+antIDstr+'.png')
plt.show()
plt.close()

#-- Make the full *solution* plots: PWV(t) and Tsun_illumination(t).

# Add variations to the mean (scan-averaged) values.
# Scale the solar illumination temperature from kiloKelvins back to Kelvins.
pwv=w0+dW #mm
Tsun=1000.0*(Tsun0+dB) # K

# Set the size of the plot
plt.rcParams["figure.figsize"] = (10,5)
# Make slow-WV and fast-WV plots as subplots (horizontally)
ff, (sp1,sp2)=plt.subplots(1,2)
# Add the antenna name as a plot title.
plt.suptitle('Antenna '+str(antID)+' , scan '+str(tgtScan))

# Left panel: Fast variations - apparent target brightness
sp1.plot(wTime,Tsun, '-r')

# Right panel: Slow variations - total PWV (compression + humidity).
sp2.plot(wTime,pwv, '-g')

plt.savefig(plotDir+'/dWdB/scan_'+scID+'_ant_'+antIDstr+'.png')
plt.show()
plt.close()

#-- Show, how much the instrumental variables G and P_inst have changed.

print("Corrected relative gains [dimension-less]: ", Grw)
print("Corrected instrumental offsets [K]:", 1000.0*Piw)

```

```

# Print total quadratic error for the variables.
print("\n===== Xi-squares =====      SCI: ",xiSci,
      ", ACD:",xiACD," , Array-mean: ",xiTbM,
      " | Total-per-equation:", xiSquare,
      "\n-----> Estimated total error per equation:",
      np.sqrt(xiSquare)*1000.0,"K -----\\n\\n")

#-- Export the resulting data in the 'conforming' format.

return {'time':wTime,'pwv':pwv,'T_B':Tsun,'J':J_wb,'C':CC}

#-----
def 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 (residuuum) 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 quantties are omitted.
    """

    #--- Static parameters

    #- I/O

    # Input data from the ATM scan preceding the target scan.
    atmScan=tgtScan-1

    #- Tools, global data

    # Flags used to check, whether we should close the tool later.
    ownMsmd=False

    # Open own tools if 'global' not provided
    if (not myMsmd):
        # Open the msmd tool.
        myMsmd=msmdtool()
        myMsmd.open(vis)
        ownMsmd=True

    # Create our dictionary with ACD load temperatures (ambient, hot) if needed.
    if (not acdLoads):
        acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                         warnIfNoLoadTemperatures = True,
                                         mymsmd=myMsmd)

    #- Antenna

```

```

antNames=myMsmc.antennanames()
myAntName=antNames[antID]
antType=myAntName[:2]

# Map the antenna efficiencies by antenna type. As per investigation from
# merged T_sun temperatures form 2023 and 2026 data (project_22_25_Tb.ods).
eta_l_map={'CM':0.941, 'DA':0.947, 'DV':0.956}
eta_F_map={'CM':0.891, 'DA':0.897, 'DV':0.906}

# Rear-spillover + ohmic antenna efficiency - see project_22_25_Tb.ods
eta_l=eta_l_map[antType]
# Forward-spillover antenna efficiency - see project_22_25_Tb.ods
eta_F=eta_F_map[antType]

# We have two exception that seem to have systematically lower
# antenna efficiencies: DA64 and DV08 - see project_22_25_Tb.ods
if myAntName=='DA64':
    eta_l=0.929
    eta_F=0.880
if myAntName=='DV08':
    eta_l=0.943
    eta_F=0.894

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']
Thot = 0.001*acdLoads[antID][atmScan]['hot']

# How strongly we insist on calibration solution G=1.0, P_inst=0.0.
# See comment in dWdBdG2().
# The used value has been found experimentally via dWdLs4_weights().
# Search mark in dWdLs4_weights(): CW_2W
calWeight=20.0

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.300 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

```

```

"""
# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for July 30 2026 Band 5 observation.
'''
Target:      RA=08 37 18.01 DEC=+18 31 23.0
Sun center:  RA=08 38 15.79 DEC=+18 28 47.2
-----
diff (arcsec, incl. cos(delta)): 822  155.8
'''

x_h=822.0 # arcsec
y_h=155.8 # arcsec

# Earth-Sun distance for July 30 2026 Band 5 observation.
Dsun=1.01508611721886*149597870.700 # km

# *** End of observation-specific part.
"""

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightenning correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature corrected to limb brightenning - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu) # kK

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

# Solar illumination. May be either zero or unbound, depending on the
# solIllumination parameter setting (T/F).
T_tgt=0.0

if (solIllumination):
    # Do not apply Tsun=0 condition, we allow for unbound illumination.
    # If mWeight is set to zero, the T_tgt value itself is irrelevant.
    mWeight=0.0
else:
    # We are pointing far off the Sun. Enforce zero illumination by adding
    # the T_tgt=0.0 equation with the appropriate weight.
    # Search mark in dWdLs4_weights(): MW_2W
    mWeight=1.0

#- Atmosphere

# How many ATM (i.e. in the model) spws do we have? In this case it is
# the same as the number of the WVR channels.
nAtmSPWs=myAtm.getNumSpectralWindows()

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmd) ['pwvAvg']

```

```

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0, 'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,'mm @airMass = ',airMass)
print('Estimated T_B(full Sun) = ',1000.0*Tsun00,'K') # scale to Kelvins.
print('.....')

# The MS metadata has been read-out, we may close the (local) tools.
if (ownMsmid) : myMsmid.done()

#-- Extract WVR raw temperatures.

# Raw WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,tgtScan,antID,ssType)

# Average over time.
TwvrMean=np.mean(Twvr,axis=1)
# Stddevs per channel.
TwvrRMS=np.std(Twvr,axis=1)

#-- Flag and remove clear outliers.

# Find all array indices, for which data in at least one of the valid
# WVR channels are out of 3 sigma interval.
validWvrChans=np.delete(range(nAtmSPWs),flaggedWVRchannels)

for SPW in validWvrChans:
    # Outliers (> 3 sigma) in the current SPW.
    badData=np.flatnonzero(abs(Twvr[SPW]-TwvrMean[SPW])>3.0*TwvrRMS[SPW])
    if (badData.size):
        print('Flagged in WVR channel no.',SPW,': Data in times ',wTime[badData],
              ', bad temperatures T=',Twvr[SPW,badData])

        # Delete data at bad-data indices in *all* WVR channels, so that the data
        # remain homogeneous in time. Make the time axis conformant.
        Twvr=np.delete(Twvr,badData,axis=1)
        wTime=np.delete(wTime,badData)
#--> endfor(valid WVR chans)

#-- Find dimensions of the system.

# How many time samples are in this scan (after flagging)?
nSamples=Twvr.shape[1]

# How many equations do we actually have per block of 1 time-sample?
# Some WVR channels may be flagged (typically, the channel 0 is saturated).
eqBlockSize=nAtmSPWs-len(flaggedWVRchannels)

# Pre-calculated frequently repeating factor: Inverse number of time samples.
nsInv=1.0/nSamples

#- Relative gains and instr. offsets - one per WVR channel, no polarization.

```

```

# Gains to be initialised to their expected value of 1.0.
Grw=np.ones(nAtmSPWs)
# Offsets should be close to 0.0.
Piw=np.zeros(nAtmSPWs)

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumination spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :- (
nChans=myAtm.getNumChan(0)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC) ['value'] [0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC) ['value'] [0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#- Arrays for calculations of the matrix and the RHS-vector components
# of the linearized system that depend just on the models (ATM + quiet Sun).
# Note: In this case, when we have 1-to-1 correspondence between the model
# and data SPWs, we actually need not to store the per-spw values to arrays.
# But we keep the structure of sci-SPW approach in dWdB_TBs() and dWBdG().

# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
TbsW0=np.zeros((nAtmSPWs,nChans))
dTdW=np.zeros((nAtmSPWs,nChans))

# Transmissivity and unit-PWV opacity. Place-holders.
trans=np.zeros((nAtmSPWs,nChans))
tau_1=np.zeros((nAtmSPWs,nChans))

# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
SF=1.0+b_2/ Tsun00*np.log(wvrFreqs/ f_LO)

#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).

#- Iteration setup

# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05

# Stoppers for iterative solution: We should get less than these values.
# We used those as weights in calculation of epsilon - the normalised
# inter-iteration change of operation-point at the very end of each iteration.
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin
dG_max=0.001 # Relative scale dG/G
dTinst_max=0.001 # dPinst/G in kiloKelvins

```

```

##### Loop over reasonable cal/mWeights #####

# Define the calWeight/mWeight iteration ranges.
wItSize=101

# In semi-log scale. Use plt.semilogx(xAxis,yAxis) for plot.
#exponents=0.05*(np.arange(wItSize)-(wItSize-1)/2)
exponents=0.05*(np.arange(wItSize))-2
weights=pow(10,exponents)

'''
wLow=3.0
wHigh=33.0
weights=wLow+(wHigh-wLow)/(wItSize-1)*np.arange(wItSize)
'''

#- We study influence of selected (iterated) calWeight/mWeight to:

# 1. Time-averaged PWV
iPWV=np.zeros(wItSize)

# 2. Time-averaged solar-illumination brightness temperature.
iTSun=np.zeros(wItSize)

# 3. Number of iterations required for convergence.
iNIT=np.zeros(wItSize)

# 4. Errors separated by 'sub-systems' of equations: science (s), ACD cal (a),
#    and mean-array T_tgt (m).
iXis=np.zeros(wItSize)
iXia=np.zeros(wItSize)
iXim=np.zeros(wItSize)

# 5. Relative updates of gains and P_inst.
iGG=np.zeros((wItSize,nAtmSPWs))
iPP=np.zeros((wItSize,nAtmSPWs))

#- The main weight-iteration loop.
for wi in range(wItSize):

    theWeight=float(weights[wi])

    # Which weight do we iterate: calWeight or mWeight? The 2nd remains static.
    # Select by comment/uncomment.

    #calWeight=theWeight
    mWeight=theWeight

    # Reset the starting values for iterations.
    w0=pwv0

```

```

# From dWdLs4() we estimate the solar contamination  $\sim 100\text{K} = 0.1\text{kK}$  in the
# on-source subscans. The off-source should be contamination-free.
if (ssType=='on'):
    Tsun0=0.1 # kiloKelvins
else:
    Tsun0=0.0 # no illumination

#- Reset gains/P_inst back to their pure-ACD values.

# Gains to be initialised to their expected value of 1.0.
Grw=np.ones(nAtmSPWs)
# Offsets should be close to 0.0.
Piw=np.zeros(nAtmSPWs)

# Iteration counter.
itCount=0
# Normalized inter-iteration change of operation point - its initial value.
# Anything bigger than 1.0
epsilon=10.0

#-- Solution-finding iterations: Minimize the mean value of variations.

while( (epsilon>1.0) and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-blocks: Jacobian-part for the [dW,dT] 'natural' variables.
    J_wb=np.zeros((eqBlockSize,2)) # 2 vars repeating in non-flagged wvrChans
    # G-block matrix: J-part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each WVR channel.

    # A-block matrix: J-part for [dG,dP_inst] for WVR "calibration" equations.
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, amb.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR channel, hot.
    # M-block matrix: Jacobian part for [dW,dT] in array-mean eqs.
    # This matrix is constant and can be defined already here.
    J_M=np.array([[0.0, nsInv]])

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))

    # RHS-vector 'appendix 1' - amb. & hot 'calibration powers' (time-avgd).
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

    # RHS-vector 'appendix 2' - array-mean values of the target  $T_B$ 
    CCM=np.zeros((1,1))

    # Solution residua for each used WVR channel.
    resT={}
    # Mapping the WVR channel to the index of equation in the system.
    spw2row={}

```

```

#- Re-calculate ATM model with updated <PWV> and solar illumination.

# Set current estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

# Count the actual number of equations - some WVR channels may be flagged.
eqCount=0

#- Loop over 4 ATM spws - these directly correspond to 4 WVR channels
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001* Tebb

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value']
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0-dws, 'mm'))[1]['value']

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001* (TebbH- TebbL)/ (2.0*dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass*(tau_wet+tau_dry)

    # Transmissivity (spectral).
    trans[atmSPW]=np.exp(- opacity)
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_1[atmSPW]=(airMass/ w0)*tau_wet

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Model + antenna efficiencies combined.

    # Sensitivity factor for PWV.
    SW=(- eta_1*eta_F*Tsun0*np.mean(trans_M*tau_1M*SF_M)

```

```

    +(1.0-eta_F+eta_F*eta_l)*np.mean(dTdW_M))*Grw[atmSPW]

# Sensitivity factor for the (apparent) solar illumination temperature.
SB=eta_l*eta_F*np.mean(trans_M*SF_M)*Grw[atmSPW]

# Model T_sky as viewed by the receiver with uniform spectral gain.
Tsky=(eta_l*eta_F*Tsun0*np.mean(trans_M*SF_M)
      +(1.0-eta_F+eta_F*eta_l)*np.mean(TbsW0_M)
      +eta_F*(1.0-eta_l)*Tamb)

# Build the equations only for the non-flagged channels.
if (atmSPW not in flaggedWVRchannels):

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the sensitivity factor(s) for the current ATM SPW as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    J_wb[eqCount]=np.array([SW,SB])

    #- Build the matrix: Instrumental (Grw & Piw sensitivities) block.

    # We have to prepend a proper offset for the Grw and Piw vars as
    # they are independent for each SPW and polarization.
    bOffset=2*eqCount

    J_gp[eqCount,bOffset]=1.0
    J_gp[eqCount,bOffset+1]=Tsky

    #- Build the matrix: ACD powers (amb & hot) 2-block.

    J_cdA[eqCount,bOffset]=1.0
    J_cdA[eqCount,bOffset+1]=Tamb

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=1.0
    J_cdH[eqCount,bOffset+1]=Thot

    #- Build the RHS vector: WVR data

    # RHS: Twvr(t) with the stationary ATM-based part
    # subtracted. Rescale measured WVR temperatures to kiloKelvin.
    dTw=0.001*Twvr[atmSPW]-Piw[atmSPW]-Grw[atmSPW]*Tsky

    # Set it as the RHS component.
    CC[eqCount]=dTw

    #- Amend the RHS vector: Calibration data.

    # Ambient-load power with its working-point part subtracted.
    # Note: Unlike Twvr, the Tamb and Thot are in kiloKelvins already.
    CCA[eqCount]=Tamb-Piw[atmSPW]-Grw[atmSPW]*Tamb
    # Hot-load power with its working-point part subtracted.
    CCH[eqCount]=Thot-Piw[atmSPW]-Grw[atmSPW]*Thot

```

```

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resT[atmSPW]=dT_w.mean()

# Mapping between ATM SPW and equation number.
spw2row[atmSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',atmSPW,' WVR channel.')
#--> endif(not flagged)

#--> endfor(ATM SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged SPW:pol channels.")
    return False

#- Solution of the grand system of linearised equations. See dWdBdG2()
# for explanation.

#- Ammend the RHS vector: Array-mean target brightness temperatures.

# Effective expected target T_B with the model at work-point subtracted.
# In our case, effective T_B is just the part of 'parasitic' illumination.
# For far-off-Sun pointings T_tgt is set to zero. For near-Sun pointings
# we use solillumantion=True and Tsun0 is unbound because mWeight=0.
CCM[0]=T_tgt-Tsun0

#- Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# 1. Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# 2. Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# 3. ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
# Need copy() here - see the same place in dWdBdG2().
J_cdA_T=np.transpose(J_cdA).copy()
J_cdH_T=np.transpose(J_cdH).copy()

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as

```

```

# np.hstack([J_cdA_T,J_cdH_T]) from already scaled-up transp. sub-blocks.
J_cd=np.vstack([J_cdA,J_cdH]).copy()
J_cd_T=calWeight*np.transpose(J_cd)

# 4. Array-mean (M-block) related matrices. Incorporate mWeight into M^T,
# like what we have done with calWeight for the A^T and H^T matrices.
M_T=np.array([[0.0],
               [nsInv*mWeight]])    # mWeight*M^T

# Pre-calculated nSamples*mWeight*(M^T x M) matrix.
# Like with A^T x A, the M^T x M product is scaled-up by mWeight just once
# (no mWeight squared!). Moreover, we scale it up here yet by nSamples,
# in order to avoid trivial nsInv*nSamples=1 in the following calcs.
nsMTM=np.array([[0.0, 0.0],
                 [0.0, nsInv*mWeight]])

# Frequently repeating blocks. NM_inv plays a role of modified J_K_inv.
NM=J_K+nsMTM                # J_K+nSamples*mWeight*M^T x M
NM_inv=np.linalg.inv(NM)    # NM^(-1)

# Off-diagonal blocks of the inverted natural sub-matrix (Ainv).
J_M_inv=nsInv*(NM_inv-J_K_inv)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 2) and (2 x nEqs) repeating blocks.
Rb=np.matmul(J_wb_T,J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T,J_gp)+np.matmul(J_cd_T,J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(NM_inv,Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc-nSamples*np.matmul(Lb,AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv,LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=J_M_inv-np.matmul(AinvRb,Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

```

```

#- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1), (-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e, multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)+np.matmul(M_T,CCM) # 2 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)+nSamples*np.matmul(M_T,CCM) # 2 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum) # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (2 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the
# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv,C_K_wb) # 2 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
# PWV variations.
dW=varNat[0]
# Illumination/brightness vars: In this approach we may have them, too.
dB=varNat[1]

# Reshape instrumental parameters to two rows (transpose): 1st is dP_inst,
# the 2nd dG, one component per WVR channel.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errs for the over-determined set of eqs.
# See corresponding part of dWdBdG2() for explanation of err estimation.

# err=f(x)-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC

```

```

# Xi square for the sci-data part, normalized per one equation.
xiSci=np.sum(errVec*errVec)/ (nSamples*eqBlockSize)

# Xi Squares per SPW:Pol channel. Sum only over times.
xiSciPerChan=np.sum(errVec*errVec,axis=1)
# Its mean and RMS.
xscMean=xiSciPerChan.mean()
xscRMS=xiSciPerChan.std()
# Find outliers ( > 3 sigma). The badData array contains indices of
# those equations, where the agreement between model and observation is
# exceptionally weak.
badData=np.flatnonzero(abs(xiSciPerChan-xscMean)>3.0*xscRMS)

# Extract the WVR channels with bad data, if we have some.
# We do not have an inverse map to spw2row, so we have to traverse.
for wvrChan,eqRow in spw2row.items():
    print('WVR chan '+str(wvrChan)+' : Error:',xiSciPerChan[eqRow])
    if eqRow in badData: print('\t\t!!! Bad data in WVR chan',str(wvrChan),
                                ':', (xiSciPerChan[eqRow]-xscMean)/ xscRMS,
                                'sigma from the mean.')

# err=f(x)-b: Calibration-data part.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the cal-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)
# Normalized per one equation.
xiACD/=eqBlockSize

# err=f(x)-b: Array-mean T_B (illumination) part.
errVecMeanTb=-CCM
# Xi square for the averaged illumination part.
xiTbM=np.sum(errVecMeanTb*errVecMeanTb)

# Grand-total Xi-square normalized to the number of equations for
# the natural/science, ACD, and mean-array sub-blocks.
xiSquare=xiSci+xiACD+xiTbM

#- All variations should tend to zero means.

dW_mean=dW.mean()
dB_mean=dB.mean()

# Update stat. part of PWV and illumination by avgs of their variations.
w0+=dW_mean
Tsun0+=dB_mean

# Update also the Piw and Grw by the found differences.
# Find the maximum inter-iteration change for convergence diagnostics.
dT_inst_sup=-1.0
dG_rel_sup=-1.0

for wvrChan,eqCount in spw2row.items():
    # Update the P_inst and G.
    Piw[wvrChan]+=dPinst[eqCount]
    Grw[wvrChan]+=dGinst[eqCount]

# Change of inst. power expressed in kiloKelvins.

```

```

dT_inst=abs(dPinst[eqCount]/ Grw[wvrChan])
if (dT_inst > dT_inst_sup): dT_inst_sup=dT_inst

# Change of gain in the relative scale.
dG_rel=abs(dGinst[eqCount]/ Grw[wvrChan])
if (dG_rel > dG_rel_sup): dG_rel_sup=dG_rel
#--> endfor(non-flagged WVRchannels/atmSPWs)

#-- Convergence diagnostics / Iteration control.

# Calculate the normalized inter-iteration change of operation point.
# The factor 0.5 comes from sqrt(1/4) as we have 4 'orthogonal' variables.
epsilon=0.5*np.sqrt((dW_mean/ dwMean_max)**2+(dB_mean/ dbMean_max)**2+
                    (dG_rel_sup/ dG_max)**2+(dT_inst_sup/ dTinst_max)**2)

# Increment the iteration counter
itCount+=1

#- Diagnostic print-outs.

# Updated scan-avgd PWV and sol. illumination after completed iteration.
print('vvvvvvv\n After iteration no. ',itCount)
print('Mean PWV for the scan,',tgtScan,' w0 = ',w0,"mm"
      ", mean Tsun(illumination) = ",1000.0*Tsun0,"K") # scale to Kelvins

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
# Scale-up back to Kelvins (we calculate in kiloKelvins).
resTArr=1000.0*np.array(list(resT.values()))

print("res T: ", resT,"mean +/- RMS: ",resTArr.mean(),"+/-",resTArr.std())

# Note: Here we measure the Pinst change directly in Kelvin(!), as nominal
# Piw is 0. For gain, we need not to divide by Grw as it is nominally 1.
print("\nInstrumental characteristics shifts during the iteration:")
for wvrChan,eqCount in spw2row.items():
    print("WVR chanNum = ",wvrChan,"\tdP_I =",1000.0*dPinst[eqCount],"K",
          "\tdG/G =",dGinst[eqCount]*100,"%")

print("Epsilon:",epsilon)

print('^^^^^^\n\n')

#--> endwhile(non-zero-means) / End of solution-iterations loop.

# Store the watched quantities in the array per-weight elements.

iPWV[wi]=w0
iTsun[wi]=1000.0*Tsun0 # Scale the result back to Kelvins.

iNIT[wi]=itCount

```

```

iXis[wi]=xiSci
iXia[wi]=xiACD
iXim[wi]=xiTbM

iGG[wi]=100.0*(Grw-1.0)
iPP[wi]=1000.0*Piw

#--> endfor(range-of-weights) / End of the main loop.

#-- Return the arrays of interest in dependence on calWeight.

return {'NS':nSamples, 'CW':weights, "PWV":iPWV, "Tsun":iTsun,
        "XiS":iXis, "XiA":iXia, "XiM":iXim, "NIT":iNIT, "dG":iGG, "dP":iPP}

#-----

def dWdLs3_calWeight(myAtm, antID, tgtScan,
                    flaggedWVRchannels=[], ssType='', solIllumination=True,
                    myMS=None, myMsmd=None, acdLoads=None):

    """
    --- OBSOLETE ---

    Replaced/superseded by a new dWdLs4_weights() function.

    """

    #--- Static parameters

    #- Dirs

    # Source-data definition. Set static because we are MS-specific routine.
    vis='uid__A002_X110fd13_X1099e.ms'
    # ATM scan preceding the solar scan 9 is used always.
    atmScan=8

    # Output/diagnostics plots.
    plotDir='./wvr_decomp_G'

    #- Tools, global data

    # Flags used to check, whether we should close the tool later.
    ownMS=False
    ownMsmd=False

    # Open own tools if 'global' not provided
    if (not myMS):
        # Open the ms tool
        myMS=mstool()
        myMS.open(vis)
        ownMS=True

    if (not myMsmd):
        # Open the ms tool.
        myMsmd=msmdtool()

```

```

myMsmd.open(vis)
ownMsmd=True

# Create our dictionary with ACD load temperatures (ambient, hot) if needed.
if (not acdLoads):
    acdLoads=aU.getLoadTemperatures(vis, doplot = False,
                                     warnIfNoLoadTemperatures = True,
                                     mymsmd=myMsmd)

#- Target

# Initial guess of the solar brightness temperature.
# SD calibration script: Reference T_b for solar disc center @198GHz
TsunCenter=6.3 # kiloKelvins!!!

# Solar radius.
Rsun=696000 # km

# *** This part is observation specific - need to read ephemeris !!!

# Heliocentric coordinates for Dec 2023 Band 5 observation.
x_h=928.0 # arcsec
y_h=87.0 # arcsec

# Earth-Sun distance for Dec 2023 Band 5 observation.
Dsun=1.4723338439337283e8 # km

# *** End of observation-specific part.

# Apparent angular solar radius in arcsec.
r_sol=180.0*3600.0*Rsun/ (Dsun*np.pi)

# mu=cos(theta). Used for limb-brightening correction below.
mu=np.sqrt(1.0- (x_h**2+y_h**2)/ r_sol**2)

# b_2=2.0*a_2, where a_2 is the temperature drop with frequency in Eq. (4)
# in Alissandrakis et al. (2020). Scaled to kiloKelvins!
b_2=-1.127 # = 2x (-563.5K) as per Table 2 in Alissandrakis et al. (2020).

# Brightness temperature corrected to limb brightening - see
# Alissandrakis et al. (2020).
Tsun00=TsunCenter+0.5*b_2*np.log(mu) # kK

# Reference frequency for the solar spectrum: We take the LO frequency (in Hz)
f_LO=1.98e11

#- Antenna

# Rear-spillover + ohmic antenna efficiency (see Tab. 3 in White et al. 2017).
eta_l=0.95
# Forward-spillover antenna efficiency (Tab. 3 in White et al. 2017).
# WVR has, unlike the sci-receiver, on-axis recording, may have better eta_F.
eta_F=0.9

# Extract ambient & hot temperatures for the current antenna and scan.
# It determines the rear-spillover contribution to noise.
# Scaled to kiloKelvins!!!
Tamb = 0.001*acdLoads[antID][atmScan]['amb']

```

```

Thot = 0.001*acdLoads[antID][atmScan]['hot']

#- Atmosphere

# Initial guess of mean PWV: Scan-center interpolated ASDM_CALWVR value.
pwv0=getPWVforAnt(vis,tgtScan,antID,myMsmc)['pwvAvg']

# Set initial estimation of PWV to the ATM model.
myAtm.setUserWH2O(qa.quantity(pwv0,'mm'))

# Airmass for the solar scan.
airMass=myAtm.getAirMass()

# Print the initial estimates. Temperature to be scaled from kiloKelvins to K.
print('..... Initial estimate')
print('Mean PWV for scan,',tgtScan,' w0 = ',pwv0,'mm @airMass = ',airMass)
print('Estimated T_B(full Sun) = ',1000.0*Tsun00,'K') # scale to Kelvins.
print('.....')

#-- Extract WVR raw temperatures.

# Raw WVR Tbs
wTime,Twvr=wvr_dataRaw(vis,tgtScan,antID,ssType)

# Average over time.
TwvrMean=np.mean(Twvr,axis=1)

# How many time samples are in this scan?
nSamples=Twvr.shape[1]

# How many ATM (i.e. in the model) spws do we have? In this case it is
# the same as the number of the WVR channels.
nAtmSPWs=myAtm.getNumSpectralWindows()

# How many equations do we actually have per block of 1 time-sample?
# Some WVR channels may be flagged (typically, the channel 0 is saturated).
eqBlockSize=nAtmSPWs-len(flaggedWVRchannels)

#- Reconstruct the sub-channel frequencies for our ATM SPWs.
# We need them for proper scaling of the solar-illumination spectrum.

# We have the same number of sub-channels for all WVR-channel ATM SPWs.
# Fortunately, because Python is incapable of creating a vector of vectors,
# which would have different sizes, only rectangular matrix is OK :- (
nChans=myAtm.getNumChan(0)

# Frequencies for sub-channels in our ATM SPWs.
wvrFreqs=np.zeros((nAtmSPWs,nChans))

for wC in range(nAtmSPWs):

    fMin=myAtm.getMinFreq(wC)['value'][0]*1.0e9 # GHz to Hz
    df=myAtm.getChanSep(wC)['value'][0]*1.0e6 # MHz to Hz

    wvrFreqs[wC]=fMin+df*np.arange(nChans)

#-- Arrays for calculations of the matrix and the RHS-vector components

```

```
# of the linearized system that depend just on the models (ATM + quiet Sun).
# Note: In this case, when we have 1-to-1 correspondence between the model
# and data SPWs, we actually need not to store the per-spw values to arrays.
# But we keep the structure of sci-SPW approach in dWdB_TBs() and dWdBdG().
```

```
# Atmosphere (+CMB) brightness temperature for PWV=w0 and its slope.
# Place-holders, to be calculated inside iterations.
```

```
TbsW0=np.zeros((nAtmSPWs,nChans))
dTdW=np.zeros((nAtmSPWs,nChans))
```

```
# Transmissivity and unit-PWV opacity. Place-holders.
```

```
trans=np.zeros((nAtmSPWs,nChans))
tau_1=np.zeros((nAtmSPWs,nChans))
```

```
# Average Solar-Flux spectrum: QSR model normalised to 1.0 at f=f_LO.
```

```
SF=1.0+b_2/ Tsun00*np.log(wvrFreqs/ f_LO)
```

```
#-- Solve the linearised set of equations for dW and dB variations.
# Iterate until they have negligible mean value (scan-average).
```

```
#-- Iteration setup
```

```
# Half dw for the estimation of the dTsky/dw by centered difference.
dws=0.05
```

```
# Stoppers for iterative solution: We should get less then these values.
```

```
dwMean_max=0.01 #mm
dbMean_max=0.001 # kiloKelvin
```

```
+++++ Loop over reasonable calWeights - experimental +++++
```

```
# Scan-averaged PWV for a given calWeight
```

```
PWV=np.zeros(900)
```

```
# Scan-averaged solar illumination T_B for a given calWeight
```

```
TSil=np.zeros(900)
```

```
# Number of iterations required for convergence
```

```
NIT=np.zeros(900)
```

```
# Xi-square: Natural part
```

```
xis=np.zeros(900)
```

```
# Xi-square: Instrumental part
```

```
xia=np.zeros(900)
```

```
for cwi in range(900):
```

```
    # Set calWeight.
```

```
    calWeight=1.0+0.05*cwi
```

```
    #- Relative gains and instr. offsets - one per WVR channel, no polarization.
```

```
    # Gains to be initialised to their expected value of 1.0.
```

```
    Grw=np.ones(nAtmSPWs)
```

```
    # Offsets should be close to 0.0.
```

```
    Piw=np.zeros(nAtmSPWs)
```

```
    # Initial values for the dW and dL means - anything bigger then the stoppers.
```

```

dW_mean=0.5
dB_mean=0.5

# Iteration counter.
itCount=0

# Set the starting values for iterations.
w0=pwv0
# From dWdLs() we estimate the solar contamination ~100K = 0.1kK in the
# on-source subscans. The off-source should be contamination-free.
if (ssType=='on'):
    Tsun0=0.1 # kiloKelvins
else:
    Tsun0=0.0 # no illumination

#-- Iterations: Minimize the mean value of variations.

while ((abs(dW_mean) > dwMean_max) or (abs(dB_mean) > dbMean_max))
    and (itCount< 20)):

    #-- Prepare/reset the linearized system of equations.

    #- Repeating blocks/tiles for each time-sample

    # N-blocks: Jacobian-part for the [dW,dT] 'natural' variables.
    if (solIllumination):
        # Solve for both dW and dB - we allow for varying solar illumination.
        J_wb=np.zeros((eqBlockSize,2)) # 2 vars repeating in non-flagged wvrc
    else:
        # Solar illumination should be zero, solve for dW solely.
        J_wb=np.zeros((eqBlockSize,1)) # 1 var repeating in non-flagged wvrc

    # G-block matrix: J-part for [dG,dP_inst] 'instrumental' variables.
    J_gp=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per each WVR chan

    # A-block matrix: J-part for [dG,dP_inst] for WVR "calibration" eqs
    J_cdA=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR c, amb.
    J_cdH=np.zeros((eqBlockSize,2*eqBlockSize)) # 2 vars per WVR c, hot.

    # RHS vector for all time samples stacked as a matrix.
    CC=np.zeros((eqBlockSize,nSamples))

    # RHS-vector 'appendix' - amb. & hot 'calibration powers' (time-avg)
    CCA=np.zeros((eqBlockSize,1))
    CCH=np.zeros((eqBlockSize,1))

    # Solution residua for each used WVR channel.
    resT={}
    # Mapping the WVR channel to the index of equation in the system.
    spw2row={}

    #- Re-calculate ATM model with updated <PWV>

    # Set current estimation of PWV to the ATM model.

```

```

myAtm.setUserWH2O(qa.quantity(w0, 'mm'))

# Count the actual number of equations - some WVR channels may be flagged.
eqCount=0

#- Loop over 4 ATM spws - these directly correspond to 4 WVR channels
for atmSPW in range(nAtmSPWs):

    #- Model value of the sky/atm brightness with current pwv.
    Tebb=myAtm.getTebbSkySpec(atmSPW, qa.quantity(w0, 'mm'))[1]['value']
    # Scale to kiloKelvins.
    TbsW0[atmSPW]=0.001* Tebb

    #- Slope (centered difference) of the atm brightness with current pwv.
    TebbH=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0+dws, 'mm'))[1]['value']
    TebbL=myAtm.getTebbSkySpec(atmSPW,
                                qa.quantity(w0- dws, 'mm'))[1]['value']

    # In kiloKelvins/mm
    dTdW[atmSPW]=0.001* (TebbH- TebbL)/ (2.0* dws)

    #- Opacities & transmissivity.

    # Dry zenith opacity
    tau_dry = np.array(myAtm.getDryOpacitySpec(atmSPW)[1])
    # Water-vapour related 'wet' opacity
    tau_wet = np.array(myAtm.getWetOpacitySpec(atmSPW)[1]['value'])
    # Total opacity (spectral).
    opacity=airMass* (tau_wet+tau_dry)

    # Transmissivity (spectral).
    trans[atmSPW]=np.exp(- opacity)
    # Wet opacity per 1mm PWV scaled by airmass.
    tau_1[atmSPW]=(airMass/ w0)* tau_wet

    #- Model (_M) factors.

    # Tebb and its slope.
    TbsW0_M=TbsW0[atmSPW]
    dTdW_M=dTdW[atmSPW]

    # Transmissivity and unit-PWV opacity.
    trans_M=trans[atmSPW]
    tau_1M=tau_1[atmSPW]

    # Solar-flux model spectrum (normalized).
    SF_M=SF[atmSPW]

    #- Model + antenna efficiencies combined.

    # Sensitivity factor for PWV.
    SW=(- eta_1* eta_F* Tsun0* np.mean(trans_M* tau_1M* SF_M)
        + (1.0- eta_F+eta_F* eta_1) * np.mean(dTdW_M) ) * Grw[atmSPW]

    # Sensitivity factor for the (apparent) solar illumination temperature.
    SB=eta_1* eta_F* np.mean(trans_M* SF_M) * Grw[atmSPW]

```

```

# Model T_sky as viewed by the receiver with uniform spectral gain.
Tsky=(eta_l*eta_F*Tsun0*np.mean(trans_M*SF_M)
      +(1.0-eta_F+eta_F*eta_l)*np.mean(TbsW0_M)
      +eta_F*(1.0-eta_l)*Tamb)

#- Build the equations only for the non-flagged channels.
if (atmSPW not in flaggedWVRchannels):

    #- Build the matrix: Natural (dW & dB sensitivities) block.

    # Set the sensitivity factor(s) for the current ATM SPW as a row
    # in the block/tile for the linearized-system Jacobian matrix.
    if (solIllumination):
        # Solve for both dW & dB independently.
        J_wb[eqCount]=np.array([SW,SB])
    else:
        # Only dW is allowed to vary.
        J_wb[eqCount]=np.array([SW])

    #- Build the matrix: Instrumental (Grw & Piw sensitivities) block.

    # We have to prepend a proper offset for the Grw and Piw vars as
    # they are independendt for each SPW and polarization.
    bOffset=2*eqCount

    J_gp[eqCount,bOffset]=1.0
    J_gp[eqCount,bOffset+1]=Tsky

    #- Build the matrix: ACD powers (amb & hot) 2-block.

    J_cdA[eqCount,bOffset]=1.0
    J_cdA[eqCount,bOffset+1]=Tamb

    # Hot-power sub-block.
    J_cdH[eqCount,bOffset]=1.0
    J_cdH[eqCount,bOffset+1]=Thot

    #- Build the RHS vector: WVR data

    # RHS: Twvr(t) with the stationary ATM-based part
    # subtracted. Rescale Twvr to kiloKelvin.
    dTw=0.001*Twvr[atmSPW]-Tsky

    # Set it as the RHS component.
    CC[eqCount]=dTw

    #- Ammend the RHS vector: Calibration data.

    # Ambient-load power with its working-point part subtracted.
    CCA[eqCount]=Tamb-Piw[atmSPW]-Grw[atmSPW]*Tamb
    # Hot-load power with its working-point part subtracted.
    CCH[eqCount]=Thot-Piw[atmSPW]-Grw[atmSPW]*Thot

```

```

#- Diagnostics

# Residua of sky temperature for mean PWV and Tsun (operation point).
resT[atmSPW]=dT_w.mean()

# Mapping between ATM SPW and equation number.
spw2row[atmSPW]=eqCount

eqCount+=1

else: print('Ignoring flagged ',atmSPW,' WVR channel.')
#--> endif(not flagged)

#--> endfor(ATM SPWs)

# Now, the eqCount should be the same as eqBlockSize. Just check it.
if (eqCount != eqBlockSize):
    print("***** Oops, something is wrong !*****")
    print("    The expected number of equations does not correspond to the")
    print("    number of measured and non-flagged WVR channels.")
    return False

#- Solution of the system of linearised equations. See dWdBdG() for
# explanation.

# Calculate the small-block matrices and their inversions needed for
# grand-system inversion.

# Natural (N-block) matrices.
J_wb_T=np.transpose(J_wb)
J_K=np.matmul(J_wb_T,J_wb)
J_K_inv=np.linalg.inv(J_K)

# Gain/P_inst (G-block) matrices.
J_gp_T=np.transpose(J_gp)

# ACD rows (A-block) matrices.
# Calculate transpositions per sub-block (ambient/hot).
J_cdA_T=np.transpose(J_cdA)
J_cdH_T=np.transpose(J_cdH)

# Increase importance of the ACD rows by calibration weight.
# Only the transposed A-block matrices should be scaled up!
J_cdA_T*=calWeight
J_cdH_T*=calWeight

# Compose the full block and its transposition. The transposition to be
# scaled up - see above. Or, alternatively, it can be built as
# np.hstack([J_cdA_T,J_cdH_T]) from already scaled-up transposed sub-blks.
J_cd=np.vstack([J_cdA,J_cdH])
J_cd_T=calWeight*np.transpose(J_cd)

#- Base-matrix extensions.

# Right and lower bands: Their (nEqs x 3) and (3 x nEqs) repeating blocks.

```

```

Rb=np.matmul(J_wb_T,J_gp)
Lb=np.transpose(Rb)

# Lower-right corner
LRc=nSamples*np.matmul(J_gp_T,J_gp)+np.matmul(J_cd_T,J_cd)

#- Frequently repeating matrix products. Cf. with extMatInv() above.

# Ainv x Rb and Lb x Ainv
AinvRb=np.matmul(J_K_inv,Rb)
LbAinv=np.transpose(AinvRb)

#- Schur complement and its inversion.
SC=LRc-nSamples*np.matmul(Lb,AinvRb)
SC_inv=np.linalg.inv(SC)

#- Blocks of inverted grand-system matrix.

# Start with the lower-left block as it repeats in upper-left block, too.
Minv21=-np.matmul(SC_inv,LbAinv)

# Upper-left block: Off-diagonal terms. The diagonal blocks differ just
# by added J_K_inv.
Minv11_OD=-np.matmul(AinvRb,Minv21)

# Upper-right block.
Minv12=np.transpose(Minv21)

# Lower-right block.
Minv22=SC_inv

#-- Apply the inverted matrix components to the RHS vector.

# Sum of the RHS along the time axis, keeping the SPW:POL dimension.
# Preserve the number of matrix rows, just sum their contents, i.e. make
# the single-column matrix.
CCsum=np.reshape(CC.sum(axis=1),(-1,1))

#- Form the RHS vector for the normal equations (K-form minimizer),
# i.e. multiply the raw data by JJ_T

# Start with the natural-variables dW and dB part.
C_K_wb=np.matmul(J_wb_T,CC)      # 2 x nSamples matrix

# Make the same also for its time-sum.
C_K_wb_sum=np.matmul(J_wb_T,CCsum)  # 2 x 1

# Follow with the instrumental part.
C_K_gp=(np.matmul(J_gp_T,CCsum)      # 2*eqCount x 1
        +np.matmul(J_cdA_T,CCA)
        +np.matmul(J_cdH_T,CCH))

#- Apply blocks of the inverted JJ_K^(-1) matrix to the C_K RHS vector.

# Constant part of the natural-variables vector (2 x 1)
nvConst=np.matmul(Minv11_OD,C_K_wb_sum)+np.matmul(Minv12,C_K_gp)

# Time-varying part of the natural-variables vector. It comes from the

```

```

# addition to the Minv11 diagonal-terms (see comment above at Minv11_OD).
nvVar=np.matmul(J_K_inv,C_K_wb) # 2 x nSamples

# Compose the complete natural variables vector.
varNat=nvVar+nvConst

# Instrumental variables. The result is of 2*eqBlockSize x 1 shape.
varInst=np.matmul(Minv21,C_K_wb_sum)+np.matmul(Minv22,C_K_gp)

#- Reshape the results - extract our elementary variables.

# Seaparate the natural-vars components.
# PWV variations present always.
dW=varNat[0]

# dB variations only in case of solar illum., otherwise they are zeros
if (solIllumination):
    dB=varNat[1]
else:
    dB=np.zeros(nSamples)

# Reshape instrumental parameters to two rows (transpose): 1st is dP_i,
# the 2nd dG, one component per WVR channel.
dPG=np.transpose(np.reshape(varInst,(eqBlockSize,2)))

# Extract the shifts in the P_inst and G
dPinst=dPG[0]
dGinst=dPG[1]

#- Analytics: Calculate the errors for the over-determined set of eqs.

# err=A.x-b: Sci-data part.
errVec=np.matmul(J_wb,varNat)+np.matmul(J_gp,varInst)-CC
# Xi square for the sci-data part.
xiSci=np.sum(errVec*errVec)

# err=A.x-b: Calibration-data part.
errVecACDamb=np.matmul(J_cdA,varInst)-CCA
errVecACDhot=np.matmul(J_cdH,varInst)-CCH
# Xi square for the cal-data part.
xiACD=np.sum(errVecACDamb*errVecACDamb)+np.sum(errVecACDhot*errVecACDhot)

# Grand-total Xi-square normalized to the number of equations (sci/acd).
xiSquare=xiSci/nSamples+xiACD/(calWeight**2)

#- All variations should tend to zero means.

dW_mean=dW.mean()
dB_mean=dB.mean()

# Update stationary part of PWV and sol T_B by means of their variations.
w0+=dW_mean
Tsun0+=dB_mean

```

```

# Update also the Piw and Grw by the found differences.
for atmSPW,eqCount in spw2row.items():
    # Update the P_inst and G.
    Piw[atmSPW]+=dPinst[eqCount]
    Grw[atmSPW]+=dGinst[eqCount]
#--> endfor(valid WVR chans)

#- Convergence diagnostics.

itCount+=1

# Updated scan-averaged PWV and sol. illum. after completed iteration.
print('..... After iteration no. ',itCount)
print('Mean PWV for the scan,',tgtScan,' w0 = ',w0,"mm"
      ", mean Tsun(illumination) = ",1000.0*Tsun0,"K") # scale to Kelvins

# Mean values of dW and dB fluctuations should tend to zero.
print("dW mean: ",dW_mean)
print("dB mean: ",dB_mean)

# Sky-brightness residua should tend to zero, too.
# Convert the spwld maps to np.arrays first. We have to go via 'list'.
resTArr=np.array(list(resT.values()))

print("res T: ", resT,"mean +/- RMS: ",resTArr.mean(),"+/-",resTArr.std())

print('.....')

#--> endwhile(non-zero-means) / end of main iteration loop.

PWV[cwi]=w0

TSil[cwi]=Tsun0

NIT[cwi]=itCount
xis[cwi]=xiSci
xia[cwi]=xiACD

#--> endfor(range-of-calWeights)

# Range of calWeights
CWS=1.0+0.05*np.arange(900.0)

#-- Return the arrays of interest in dependence on calWeight.

return {'CW':CWS,"PWV":PWV,"Ts":TSil,"XiS":xis,"XiA":xia,"NIT":NIT}

#-----

def 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.)

'''

#-- I/O

Output-plots directory.

pngDir='./4scanPWVs/'

if (not os.path.exists(pngDir)): os.makedirs(pngDir)

#-- Global tools & data

myMS=mstool()

myMS.open(vis)

myMsmD=msmdtool()

myMsmD.open(vis)

acdLoads=aU.getLoadTemperatures(vis,doplot = False,
warnIfNoLoadTemperatures = True,
mymD=myMsmD)

#-- Prepare ATM models

#- ATMs for WVR

Leading P-cal scan (WVR).

pCallATM=attool()

setAtmTool(pCallATM,vis,4,7,myMsmD=myMsmD)

Pre-solar ATM scan (WVR).

sysCalATM=attool()

setAtmTool(sysCalATM,vis,4,8,myMsmD=myMsmD)

Solar scan

solATM=attool()

setAtmTool(solATM,vis,4,9,myMsmD=myMsmD)

Trailing P-cal scan (WVR).

pCal2ATM=attool()

setAtmTool(pCal2ATM,vis,4,10,myMsmD=myMsmD)

#- ATMs for SCI SPWs.

Sci SPW for leading-phase scan (sci).

p1SciATM=attool()

```

setAtmTool4SPWs (p1SciATM, vis, 7, myMsmnd=myMsmnd)

# Sci SPW for solar scan (sci).
solSciATM=attool()
setAtmTool4SPWs (solSciATM, vis, 9, myMsmnd=myMsmnd)

# Sci SPW for leading-phase scan (sci).
p2SciATM=attool()
setAtmTool4SPWs (p2SciATM, vis, 10, myMsmnd=myMsmnd)

#-- Get antenna list.
antIDs=myMsmnd.antennasids()
antNames=myMsmnd.antennanames()

#-- Redirect output to a text file.

sys.stdout=open("./STDOUT.txt", "w")

#-- Loop over all antennas.
for myAnt in antIDs:

    # Ant ID as int.
    antID=int (myAnt)

    # Report progress.

    print ('\n=====')
    print ('Calculating antenna ', antID, '<-->', antNames [myAnt])
    print ('=====')

#-- Run sci-spw PWV extraction via dWdBdG2() for all antennas.

sciPWV_S=dWdBdG2 (vis, 8, 9, solSciATM, antID,
                  spwList=[5, 7, 9, 11, 6, 8, 10, 12, 0, 1, 2, 3],
                  T_tgt=6905.0, flaggedData=[], myMS=myMS, myMsmnd=myMsmnd,
                  acdLoads=acdLoads)

'''
An attempt (failed) to extract sci-SPW PWV proxy from the phase-calibrator
scans. For some reason - most likely that PC-scan is performed in the non-MD
mode - the powers in the p-cal scan are much higher than expected and
using the gains found by the ATM/ACD scan lead to brightness temperatures
comparable to the Sun, i.e. around 4000K!

sciPWV_P1=dWdBdG2(vis, 8, 7, p1SciATM, antID,
                  spwList=[5, 7, 9, 11, 6, 8, 10, 12, 0, 1, 2, 3],
                  T_tgt=0.0, flaggedData=[], myMS=myMS, myMsmnd=myMsmnd,
                  acdLoads=acdLoads)

sciPWV_P2=dWdBdG2(vis, 8, 10, p2SciATM, antID,
                  spwList=[5, 7, 9, 11, 6, 8, 10, 12, 0, 1, 2, 3],
                  T_tgt=0.0, flaggedData=[], myMS=myMS, myMsmnd=myMsmnd,
                  acdLoads=acdLoads)

```

```

'''
#-- Run the dWdLs4() calculations for each scan. Only for 12m antennas.
if (antNames[myAnt][:2]!='CM'): # Skip CM antennas, no WVR.

    pCal1PWV=dWdLs4(vis,pCal1ATM,antID,7,
                    flaggedWVRchannels=[0],ssType='',solIllumination=False,
                    myMsmD=myMsmD,acdLoads=acdLoads)

    sysCalPWV=dWdLs4(vis,sysCalATM,antID,8,
                    flaggedWVRchannels=[0],ssType='',solIllumination=False,
                    myMsmD=myMsmD,acdLoads=acdLoads)

    solOffPWV=dWdLs4(vis,sysCalATM,antID,9,
                    flaggedWVRchannels=[0],ssType='off',
                    solIllumination=False,
                    myMsmD=myMsmD,acdLoads=acdLoads)

if (solWVR):
    # WVR in solar on-source subscan.
    solOnPWV=dWdLs4(vis,solATM,antID,9,
                    flaggedWVRchannels=[0],ssType='on',solIllumination=True,
                    myMsmD=myMsmD,acdLoads=acdLoads)

    pCal2PWV=dWdLs4(vis,pCal2ATM,antID,10,
                    flaggedWVRchannels=[0],ssType='',solIllumination=False,
                    myMsmD=myMsmD,acdLoads=acdLoads)

#- Comparison plots only for 12m antennas.

# Init plot for antenna.
plt.rcParams["figure.figsize"] = (20,15)
plt.suptitle(antNames[myAnt])

# Plot the WVR-based PWV(t).
t1=pCal1PWV['time']
p1=pCal1PWV['pwv']
plt.plot(t1,p1,'r.')

t2=sysCalPWV['time']
p2=sysCalPWV['pwv']
plt.plot(t2,p2,'g.')

t3=solOffPWV['time']
p3=solOffPWV['pwv']
plt.plot(t3,p3,'b.')

if (solWVR):
    # WVR in solar on-source subscan.
    t4=solOnPWV['time']
    p4=solOnPWV['pwv']
    plt.plot(t4,p4,'k.')

```

```

t5=pCal2PWV['time']
p5=pCal2PWV['pwv']
plt.plot(t5,p5,'m.')

# Add the sciSPW-based PWV(t).
t6=sciPWV_S['time']
p6=sciPWV_S['pwv']
plt.plot(t6,p6,'c.')

'''
t7=sciPWV_P1['time']
p7=sciPWV_P1['pwv']
plt.plot(t7,p7,'c.')

t8=sciPWV_P2['time']
p8=sciPWV_P2['pwv']
plt.plot(t8,p7,'c.')
'''

# Add the SYSCAL-produced PWVs. Select just last three values
# which are at times around our 4 scans.
sTimes,sPWVs=getPWVforAnt(vis,9,antID,myMsmd=myMsmd) ['allPWVs']
tLast=sTimes[-3:]
pLast=1000.0*sPWVs[-3:] # PWV in m -> mm

plt.plot(tLast,pLast,'ro')

# Save to a file and close.
antIndex='_{:02d}'.format(antID)

plt.savefig(pngDir+'ant'+antIndex+'.png')
plt.show()
plt.close()

#- Save WVR/PWV data to binary NPY file.

# Convert the antenna ID to a single-element array.
antIDa=np.array([antID])

# Open our NPY-binary file in the append mode and save the arrays for
# the current antenna.
with open('./pwv4scansW.npy', 'ab') as f:
    np.save(f,antIDa)

# WVR-based PWVs
np.save(f,t1)
np.save(f,p1)
np.save(f,t2)
np.save(f,p2)
np.save(f,t3)
np.save(f,p3)

if (solWVR):

```

```

        # WVR in solar on-source subscan.
        np.save(f,t4)
        np.save(f,p4)

    np.save(f,t5)
    np.save(f,p5)

    # Sci SPW-based PWVs
    np.save(f,t6)
    np.save(f,p6)

    '''
    np.save(f,t7)
    np.save(f,p7)
    np.save(f,t8)
    np.save(f,p8)
    '''
    # --> end(save block)

#--> endif(not-a--CM-antenna)

#-- Save the *full* dWdBdG() output for *all* antennas.

antIDa=np.array([antID])

# Extract the arrays from the map.
tt=sciPWV_S['time']
dW=sciPWV_S['pwv']
dB=sciPWV_S['T_B']
AA=sciPWV_S['J']
CC=sciPWV_S['C']

# Open our NPY-binary file in the append mode and save the arrays for
# the current antenna. Save the matrix AA and the RHS vector CC, too, so
# that the results can be extracted by our restoreSol_dWdB() method.
with open('./PWV.npy', 'ab') as f:
    np.save(f,antIDa)
    np.save(f,tt)
    np.save(f,dW)
    np.save(f,dB)
    np.save(f,AA)
    np.save(f,CC)
# --> end(save block)

# Flush the redirected stdout stream.
sys.stdout.flush()
# --> endfor(all antennas)

#--- Clean-up

# Restore stdout back to console.
sys.stdout = sys.__stdout__

solSciATM.done()
pCal2ATM.done()
solATM.done()

```

```

sysCalATM.done()
pCallATM.done()

myMsmc.done()
myMS.done()

```

```

return

```

```

#-----

```

```

def 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.
    """

```

```

    #-- Static settings

```

```

    filePath='./pwv4scansW.npy'

```

```

    #-- List all antennas in the array.

```

```

    myMsmc=msmctool()
    myMsmc.open(vis)
    antIDs=myMsmc.antennas()
    antNames=myMsmc.antennanames()

```

```

    myMsmc.done()

```

```

    # So far empty map [antID -> pwv_rec].
    pwv_map={}

```

```

    # Init the record counter.
    rCount=0

```

```

    # Open the binary dWdB data.
    with open(filePath, mode='rb') as f:

```

```

        #-- Loop over the stack of records.

```

```

        while (rCount<runID):

```

```

            #-- Loop over all antennas.

```

```

            for myAnt in antIDs:

```

```

                # Skip the CM antennas, they have no WVR.

```

```

                if (antNames[myAnt][:2]=='CM'): continue

```

```
# We may be aiming at the last record (the default) - check the EOF.
# The np.load() method needs to be checked via exceptions.
```

```
try:
```

```
    antIDa=np.load(f)
```

```
except (EOFError, ValueError, OSError): # EOF: new/old NumPy style
    print("EOF reached beyond record no. ", rCount, ".\n")
    # Prevent further reading attempts.
    rCount=runID
    # Break the antenna for-loop.
    break
```

```
#--> end try/catch block.
```

```
# Convert the antenna ID from (single-element) int-array to int.
antID=antIDa[0]
```

```
# Read the times and pwvs per scans/sub-scans
```

```
t1=np.load(f)
p1=np.load(f)
t2=np.load(f)
p2=np.load(f)
t3=np.load(f)
p3=np.load(f)
```

```
if (hasSolWVR):
```

```
    # Read solar on-source subscan data only if they are present.
    t4=np.load(f)
    p4=np.load(f)
```

```
t5=np.load(f)
p5=np.load(f)
t6=np.load(f)
p6=np.load(f)
```

```
#- Organise the data depending whether we have or not the on-source WVRs
```

```
if (hasSolWVR):
```

```
    # All times as a list of arrays
    times=[t1,t2,t3,t4,t5,t6]
    # All corresponding PWSs as a list of arrays
    pwvs=[p1,p2,p3,p4,p5,p6]
```

```
else:
```

```
    # Times except the on-source subscan as a list of arrays
    times=[t1,t2,t3,t5,t6]
    # Corresponding PWSs except the on-source subscans as a list of arrays
    pwvs=[p1,p2,p3,p5,p6]
```

```
# One record per antenna
```

```
pwv_rec={'times':times, 'pwvs':pwvs}
```

```
# Append the record to the collection. If multiple records/runs are
```

```

        # read, the map item is overwritten until the last one.
        pwv_map[antID]=pwv_rec

    #--> endfor(all antennas)

    # Increment the record counter.
    rCount+=1

    #-->endwhile(recordCountReached)
    #--> end with-block

    # Return the entire collection.
    return pwv_map

#-----

def 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.
    """

    #-- Static settings for this specific project/MS.

    # Input data selection
    vis='uid__A002_X110fd13_X1099e.ms'
    tgtScan=9
    atmScan=8

    # Output figures directory - separate for each SPW type.
    pngDir='./sol_WVR+dPWV/'
    if (not os.path.exists(pngDir)): os.makedirs(pngDir)

    # Set the ATM model for the WVR data (SPW 4) and the solar scan
    myAtm=attool()
    setAtmTool(myAtm,vis,4,tgtScan)

    #-- Extract the list of antennas.
    myMsmd=msmdtool()
    myMsmd.open(vis)

    antIDs=myMsmd.annenaids()

```

```

antNames=myMsmc.antennanames()

myMsmc.done()

#-- dPWVs: To be extracted from the pre-calculated file.

dwdbMap=restoreSol_dWdB()

#-- Loop over all antennas

for myAnt in antIDs:

    # Skip all CM antennas, these have no WVRs.
    if (antNames[myAnt][:2]=='CM'):
        print("Skipping antenna ",antNames[myAnt], " - no WVR data.")
        continue

    # Report progress.
    print('Calculating antenna ',int(myAnt),'<-->',antNames[myAnt])

    #- Science SPW: dPWV proxy

    # Time axis for science SPWs.
    timeSci=dwdbMap[int(myAnt)]['time']
    # dPWV proxy from science SPW
    dPWV=dwdbMap[int(myAnt)]['dW']

    #- WVR data from SPW 4

    timeWVR,dW,dL=dWdLs(int(myAnt),myAtm)

    #- Init plots

    # Set the sizes of the plot
    plt.rcParams["figure.figsize"] = (10,5)
    # Make XX and YY plots as subplots (horizontally)
    ff,(sp1,sp2)=plt.subplots(1,2)
    # Add the antenna name as a plot title.
    plt.title(antNames[myAnt])

    #- XX: left sub-plot
    # Plot the dPWV results as a red line.
    sp1.plot(timeSci,dPWV[0],'-r')
    # Overplot the WVR variations as blue.
    sp1.plot(timeWVR,dW,'-b')

    #- YY: right sub-plot
    # Plot the dPWV results as a red line.
    sp2.plot(timeSci,dPWV[1],'-r')
    # Overplot the WVR variations as blue.
    sp2.plot(timeWVR,dW,'-b')

    #- Save to a file and close.

    plt.savefig(pngDir+'wvrXdpwv_'+antNames[myAnt]+' .png')
    plt.show()
    plt.close()

```

```

#--> endfor(all antennas)

# Close the ATM tool.
myAtm.done()

return

#-----

def pwv4ScansCompare(vis):
    """
    More sophisticated comparison 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().
    """

    # Where to store the correlation plots.
    pngDir='./pwvCorrs/'

    # Load the PWV time profiles mapped by antenna ID.
    pwvMap=pwv4scansRestore(vis)

    for antID,pwvProfile in pwvMap.items():

        # Extract times and pwvs for all scans/sub-scans.
        times=pwvProfile['times']
        pwvs=pwvProfile['pwvs']

        # The solar on-source subscan WVR-base PWV is listed at index 3.
        tW=times[3]
        pW=pwvs[3]

        # Its proxy counterpart calculated from sci SPW is at index 5.
        tS=times[5]
        pS=pwvs[5]

        # We need to have collocated data in order to calculate linear regression.
        # Interpolate to times of sci SPW proxy data as they have higher cadence.
        pWi=np.interp(tS,tW,pW)

        # Calculate coefficients of linear correlation between WVR- and sciSPW-based
        # PWV values interpolated to the sci-SPW times.
        k,q=np.polyfit(pWi,pS,1)

        #-- Print & plot the correlation results.
        print(antID,'\t',k,'\t',q)

        # Antenna index with two digits (e.g., '07') as the fig title.
        antIndex='_{:02d}'.format(antID)
        plt.suptitle(antIndex)

        # Scatter plot

```

```
plt.plot(pWi,pS, 'b. ')\n\n# Find the range of WVR-based PWVs\npwmin=np.min(pW)\npwmax=np.max(pW)\n\n# Overplot the scatter-plot with best linfit/\nx=np.array([pwmin,pwmax])\ny=k*x+q\nplt.plot(x,y, 'r-')\n\n# Add the identity line for comparison.\nplt.plot(x,x, 'g-')\n\n# Save the fig to a file and close.\nplt.savefig(pngDir+'pwv_'+antIndex+'.png')\nplt.show()\nplt.close()\n#-->endfor(all recorded antennas)\n\nreturn
```