BIPP uniform weighting implementation summary

Rémi Poitevineau

October 2024

1 Uniform weighting implementation in BIPP

1.1 Algorithm

The uniform weighting scheme requires placing the UV coordinates on a grid and count how many visibilities fall into each cell. Then, for each cell, assign a weight to each visibility by dividing 1 by the total number of visibilities in that cell. Here's the resume of how works my implementation of uniform weighting in BIPP.

1. Gather all UV coordinates.
2. Define a grid and count the number of visibilities per cell.
3. Begin the BIPP imaging loop. For each time step:
   - Retrieve the visibilities and identify their corresponding cells.
   - Calculate the weight for each visibility as $\frac{1}{\text{number of visibilities in the cell}}$.
   - Set the weight to zero if the visibility falls outside the grid.
4. Compute the total sum of weights for normalization and to verify the gridding process. If correct, the number of non-zero cells should match the total sum of weights.

The next sections present shortly the code used. It can be found here

1.2 UV collection

The first step is to gather all the UV data. To achieve this, I duplicated the Imaging loop (see Section 1.6) in order to extract the UV values at each time step, storing them in a list. After the loop completes, I convert the list into a NumPy array for further use.

```
###################### extract uv
print('getting uv')
uu = []
vv = []
for t, f, S in ProgressBar(
        ms.visibilities(channel_id=channel_id, time_id=slice(timeStart, timeEnd, 1)
        , column=args.column)):
    wl = constants.speed_of_light / f.to_value(u.Hz)
    UVW_baselines_t = ms.instrument.baselines(t, uvw=True, field_center=ms.field_center)
    uvw = frame.reshape_and_scale_uvw(wl, UVW_baselines_t)
    ut, vt, wt = uvw.T
    uu.extend(ut)
    vv.extend(vt)

uu = np.array(uu)
vv = np.array(vv)
```

1.3 Grinding function

The gridding function is designed to create a two-dimensional histogram grid from a set of data points represented by their x and y coordinates. It returns the bin edges, the counts of points in each bin, and the corresponding inverse counts.

Here's the algorithm of the gridding function follow by the python code:

1. **Grid Boundaries Calculation**: The function first calculates the start and end points of the grid based on the number of bins (`N`) and the cell size (`du`):

$$\texttt{grid\_start} = -\frac{N}{2} \times \texttt{du}$$
$$\texttt{grid\_end} = \frac{N}{2} \times \texttt{du}$$

2. **Grid Edges Definition**: Using `np.linspace`, the function creates arrays `xedges` and `yedges`, which define the edges of the grid cells along the x and y dimensions. Each array contains `N+1` equally spaced values from `grid_start` to `grid_end`.

3. **2D Histogram Calculation**: The function uses `np.histogram2d` to compute the 2D histogram of the data points (`u` and `v`). This step produces:

   - `counts`: A 2D array containing the counts of points that fall into each grid cell defined by `xedges` and `yedges`.
   - Two additional arrays (not used) that contain the bin edges.

4. **Inverse Counts Calculation**: The function then computes the inverse of the `counts` array using `np.true_divide`. This is done inside a `with np.errstate(divide='ignore', invalid='ignore')` block to handle any potential division by zero or invalid operations gracefully:

$$\texttt{inv\_counts} = \texttt{np.true\_divide}(1, \texttt{counts})$$
$$\texttt{inv\_counts[np.isfinite(inv\_counts)]} = 0$$

5. **Return Values**: Finally, the function returns the following:

   - `xedges`: The x-coordinates of the grid edges.
   - `yedges`: The y-coordinates of the grid edges.

- `counts`: The 2D histogram counts.
- `inv_counts`: The inverse counts for each grid cell.

```
def gridding(N, du, u, v):
    grid_start = -N//2 * du
    grid_end = N//2 * du
    xedges = np.linspace(grid_start, grid_end, N+1)
    yedges = np.linspace(grid_start, grid_end, N+1)
    counts, _, _ = np.histogram2d(u, v, bins=[xedges, yedges])
    with np.errstate(divide='ignore', invalid='ignore'):
        inv_counts = np.true_divide(1, counts)
        inv_counts[~np.isfinite(inv_counts)] = 0
    return xedges, yedges, counts, inv_counts
```

1.4 Grid definition

Here, we define our grid. We begin by calculating the UV cell size, du, based on the field of view and wavelength. Note that the UV size is measured in wavelengths, not meters. Next, we define the grid for storing the UV values and apply the gridding function mentioned earlier. We also print the number of non-empty cells to verify later if the total weight sum is accurate.

```
###################### gridding
# Number of pixels of the dirty image (NxN)
N = args.npix
# Field of fiew in rad
fov = args.fov
# WWavelength
wl = constants.speed_of_light / frequency.to_value(u.Hz)
# Size of a uv cell in the UV space
du = 1/fov*wl

print('gridding')
xedges, yedges, counts, inv_counts  = gridding(N, du, uu, vv)

# Print check
n_factor = float(np.count_nonzero(counts))
print("non zero cells=",n_factor)
```

1.5 Weighting function

The weighting function is designed to assign weights to data points based on their position in a predefined 2D grid. The function returns an array of assigned weights for each data.

It takes the following parameters:

- `u`: An array of x-coordinates for the data points.
- `v`: An array of y-coordinates for the data points.
- `xedges`: The edges of the grid along the x-axis, typically generated from a previous gridding step.
- `yedges`: The edges of the grid along the y-axis, also generated from the gridding step.
- `counts`: A 2D array containing the counts of points in each grid cell.

The function follows this algorithm:

1. **Determine Grid Cell Indices:** The function uses `np.digitize` to find the indices of the grid cells that correspond to the coordinates `u` and `v`. This results in:
   - `x_idx`: The index of the grid cell for each value in `u`.

- `y_idx`: The index of the grid cell for each value in `v`.

The indices are adjusted by subtracting 1 to match Python's zero-based indexing.

2. **Clipping Indices:** The function applies `np.clip` to ensure that `x_idx` and `y_idx` stay within valid bounds:
   - The indices are clipped to be within the range $[0, \texttt{len(xedges)} - 2]$ and $[0, \texttt{len(yedges)} - 2]$, respectively. This prevents indexing errors that would occur if the coordinates are outside the grid boundaries.
3. **Assign Weights:** The function creates an array, `assigned_weights`, that pulls the corresponding counts from the `counts` array based on the computed indices `x_idx` and `y_idx`. This means that each data point in `u` and `v` is assigned a weight based on the number of points in the grid cell it falls into.
4. **Handle Out-of-Bounds Points:** A loop checks if any data points fall outside the grid boundaries defined by `xedges` and `yedges`. If a point is out of bounds, its corresponding weight in `assigned_weights` is set to 0.0.
5. **Return Weights:** Finally, the function returns the `assigned_weights`, which provides a weighted representation of the input data points based on their grid cell counts.

```
def weighting(u, v, xedges, yedges, counts):
    x_idx = np.digitize(u, xedges) - 1
    y_idx = np.digitize(v, yedges) - 1
    x_idx = np.clip(x_idx, 0, len(xedges)-2)# N-1)
    y_idx = np.clip(y_idx, 0, len(yedges)-2)#N-1)
    assigned_weights = counts[x_idx, y_idx]
    for i in range(len(u)):
        if u[i]<xedges[0] or u[i]>xedges[-1] or v[i]<yedges[0] or v[i]>yedges[-1]:
            assigned_weights[i] = 0.0
    return assigned_weights
```

1.6 Imaging with uniform weighting

To apply uniform weighting in BIPP, a few additional steps must be added inside the imaging loop. First, the shape of S.data needs to match that of the rescaled UVW so that the gridding function can return an array of weights. This allows S.data to be multiplied directly by the array returned by the weighting function. Afterward, S.data must be returned to its original shape so it can be passed to imager.collect(). Additionally, we need to add the sum of visibility weights for this epoch to the global weight sum (which is initialized to 0 before the loop begins). This total weight sum is later used to normalize the data after the dirty image has been computed.

```
W_glob = 0
############################
##########################
for t, f, S in ProgressBar(
        ms.visibilities(channel_id=channel_id, time_id=slice(timeStart, timeEnd, 1), column=
            args.column)
):

    wl = constants.speed_of_light / f.to_value(u.Hz)
    XYZ = ms.instrument(t)
    W = ms.beamformer(XYZ, wl)
    S, W = measurement_set.filter_data(S, W)

    UVW_baselines_t = ms.instrument.baselines(t, uvw=True, field_center=ms.field_center)
    uvw = frame.reshape_and_scale_uvw(wl, UVW_baselines_t)
    if np.allclose(S.data, np.zeros(S.data.shape)):
```

```
        continue
    ##### un comment for natural weighting
    #imager.collect(wl, fi, S.data, W.data, XYZ.data, uvw)

    ###################### weighting the visibility with the grid
    new_S = S.data.T.reshape(-1, order="F")
    ut, vt, wt = uvw.T
    w = weighting(ut, vt, xedges, yedges, inv_counts)
    W_glob += np.sum(w)
    #### un comment for getting the psf
    #new_S = np.full(new_S.shape, 1 + 1j)*w#new_S*w
    new_S = new_S*w
    new_S = new_S.reshape((S.data.shape[0], S.data.shape[0]), order="F").T
    imager.collect(wl, fi, new_S, W.data, XYZ.data, uvw)

print("sum of w=",W_glob)


images = imager.get().reshape((-1, args.npix, args.npix))


lsq_image = fi.get_filter_images("lsq", images)
std_image = fi.get_filter_images("std", images)

I_lsq_eq = s2image.Image(lsq_image, xyz_grid)
I_std_eq = s2image.Image(std_image, xyz_grid)

print("lsq_image.shape =", lsq_image.shape)

###############################################
I_lsq_eq_summed = s2image.Image(lsq_image.reshape(args.nlevel,lsq_image.shape[-2], lsq_image.
    shape[-1]).sum(axis = 0), xyz_grid)

# I_lsq_eq_summed should be divided b W_glob
# I_lsq_eq_summed = I_lsq_eq_summed / W_glob
# but it returns the error:
# TypeError: unsupported operand type(s) for /: 'Image' and 'float'
 # Instead do it manually after
```

2 Results on the gauss4_ t201806301100_ SBL180.MS file

2.1 PSF results

To verify if the gridding is accurate, we examine the shape of the PSF. In Fig. 1, we observe that the uniform PSF shape from BIPP matches that of WSclean. Additionally, by comparing the pixel value distributions of both PSFs (Fig. 2), we see that the two distributions overlap closely. When examining the difference between the distributions, we find a sharply peaked Gaussian centered around 0, with tails extending to 0.2. This indicates a maximum difference of 2% between the BIPP and WSclean uniform PSFs. Since the PSF is influenced by the gridding of the UV space, we can conclude that the earlier gridding process is correct

2.2 Dirty image results

Next, we compare the resulting dirty images from BIPP, CASA, and WSclean using the $gauss4_t201806301100_SBL180.MS$ file. We begin with a visual comparison of the three images (Fig. 3). All three dirty images display a similar overall shape and pixel value magnitudes. While there are some variations in the shapes, these differences are expected.

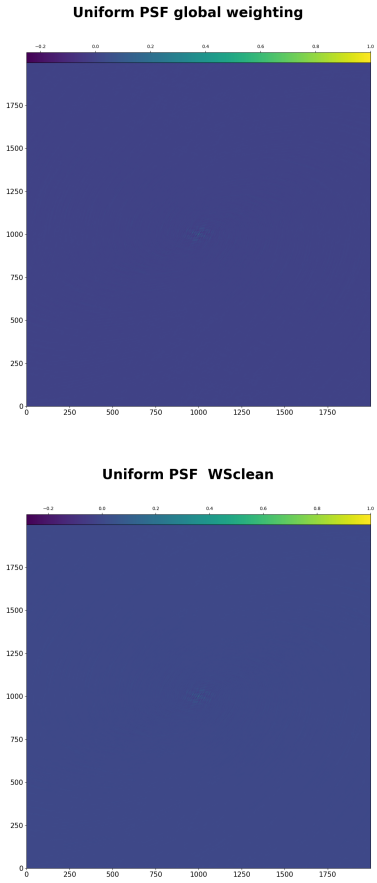

Figure 1: Uniform PSF of BIPP (bottom figure) and WSclean (top figure)

For a deeper analysis, we examine the pixel value distributions of the dirty images (Fig. 4). The BIPP dirty image shows higher values in both the negative and positive ranges, with a difference of about 5% compared to the other two images. There is also a slight variation in shape, with the second and third bumps being less pronounced in BIPP. When comparing BIPP to the other two software programs, the distributions are centered around zero. Both comparisons reveal a negative tail in the distributions.

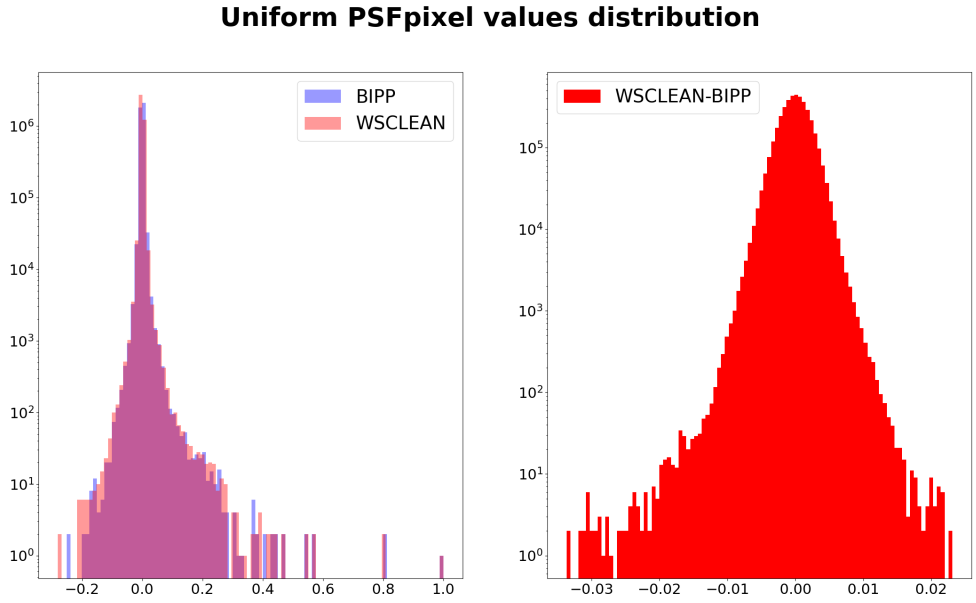

Figure 2: Comparison of the uniform PSF pixels value distribution of BIPP and WSclean. On the left, we plotted the pixels value distributions of both PSF and on the right, we plotted the difference between the two. The x-axis represent the pixels values and the y-axis the number of pixels with this value.

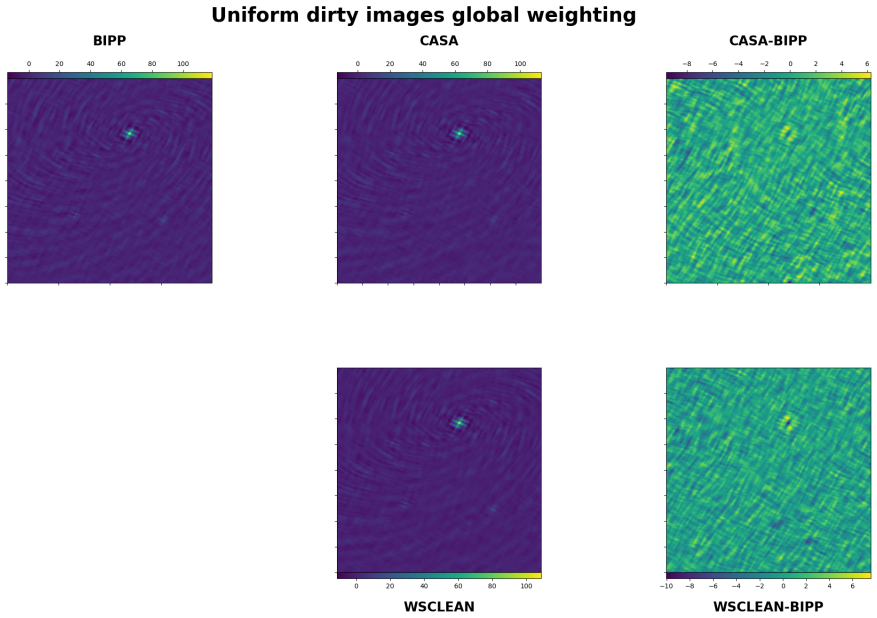

Figure 3: Dirty image obtain with BIPP, CASA and WSclean from the $gauss4\_t201806301100\_SBL180.MS$. From left to right of the top row, we show the dirty image made by BIPP, the dirty image obtained with CASA and the difference between the two. From left to right on the bottom row, we find the WSclean dirty image and then its difference with the BIPP dirty image. The colorbars represent the pixel values.

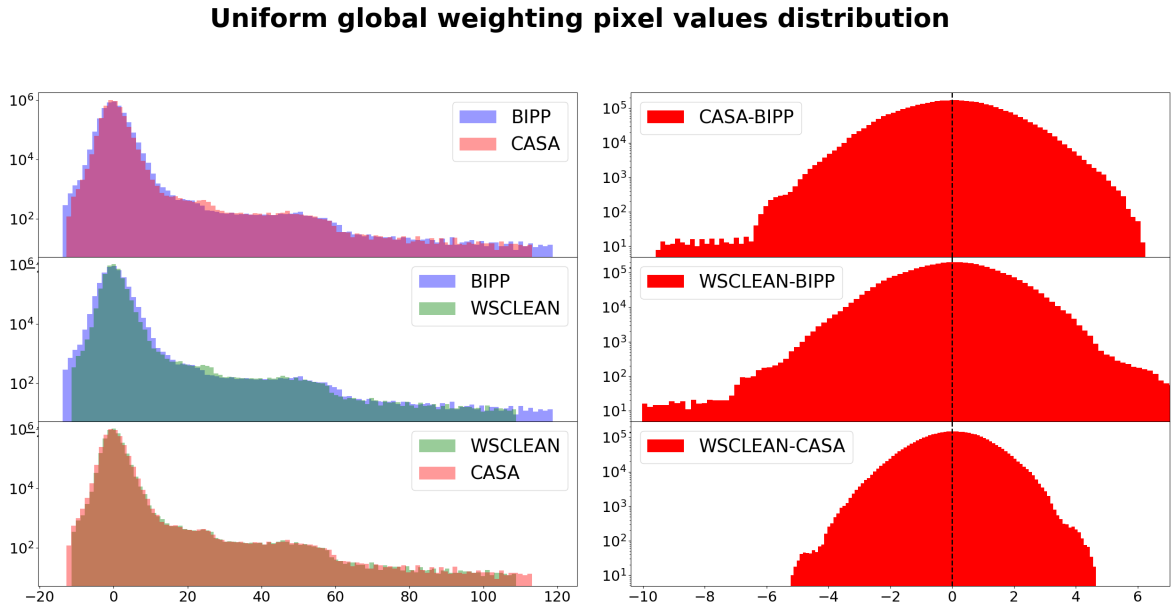

Figure 4: Comparison of the dirty image pixels value distribution between BIPP, CASA and WSclean. Each row compare two dirty images. On the left column, we superimpose both distribution and the right column we plot the difference between those two distributions. The x-axis represent the pixels values and the y-axis the number of pixels with this value.

2.3 BIPP print

Here I show what the BIPP code returns on the terminal where running.

```
(VENV) [poitevin@i24 Bluebild]$ python 4gauss_uniform.py LOFAR gauss4_t201806301100_SBL180.MS
    -o 4Gauss_t1_sigma095 -n 2000 -f 4.26 -p 15
Nvidia GPU detected!
N_station:37 , N_antenna:37
Telescope Name:LOFAR
MS file:gauss4_t201806301100_SBL180.MS
Output Name:4Gauss_t1_sigma095
N_Pix:2000 pixels
FoV:4.26 deg
N_level:4 levels
Clustering Bool:True
Clustering:kmeans
grid:ms
ms_fieldcenter:True
ms channel start:0 channel end: -1
ms timestep start: 0 timestep end: -1
WSClean Path:/work/ska/MWA/WSClean/1133149192-084-085_Sun_10s_cal.ms_WSClean-image.fits
Partitions:15
MS Column Used: DATA
nuFFT tolerance: 0.001
Multi-channel mode with -1channels.
wl:8.527800046654802; f: 35154724.12109375 Hz
Self generated grid used based on ms fieldcenter
N_pix =  2000
precision =  double
Proc =  GPU
Initial set up takes 0.23575592041015625 s
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ PARAMETER ESTIMATION @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

3592it [01:28, 40.73it/s]
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ IMAGING @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

getting uv
3592it [00:49, 72.98it/s]
len uu= 4917448
max u= 63710.829254219025
min u= -63710.829254219025
N= 2000
fov rad= 0.07435102613495843
wl = 8.510069964693551
du =  114.69646741896402
gridding
max xedges =  114696.46741896402
max yedges =  114696.46741896402
non zero cells= 32228.0
sum w= 32227.999999999996
imaging
3592it [28:30,  2.10it/s]
sum w= 32228.000000000033
lsq_image.shape = (5, 2000, 2000)
```