Exercise 2: Interferometric Imaging

Behaviour of Fourier transforms can be kind of unintuitive. Let's code up a little imaging simulator and explore some phenomena. Using whatever programming language you like (a Jupyter is very good for this, but not required).

Part 1: Set up

Step 0: Decide if you want to do this in 1D or 2D. Both are fine: 1D makes it easier to plot and see what's going on; 2D is closer to actual interferometric imaging. I'm writing up the solutions with 1D.

Step 1: define a 1D or 2D array as your 'sky'. To start with, let's just put in a delta function somewhere (all zeros, with one non-zero value). Optionally, define a coordinate array (in arbitrary units). The array size is arbitrary, but can be fairly large. A modern computer can do an FFT on a 10000 element array in no time at all.

Step 2: Find an FFT package in whatever programming language you're using, and read up on how to use it. Note that, for example, scipy's fft does some funny things with ordering in the output results of the FFT.

Take the FFT of your sky array, which gives you the visibilities. Set up a function to plot the amplitude and phase of the visibilities. Check that the results from inputing a delta function look reasonable.

Step 3: Define a sampling/weight function. This is an array (with the same number of elements as the visibilities), which defines which Fourier components we're measuring. Setting the whole array equal to 1 is equivalent to a perfect observation (all possible baselines, equal weighting). Setting elements to zero gives missing baselines. Putting in non-zero measurements acts as a weighting scheme.

Note that every weighting/sampling function must be symmetric (subject to however your FFT package is ordering things)! This is because all baselines also have the reverse baseline measured. If this is not followed then the resulting image (after inverse FFT-ing) will not be real-valued! (This is good check to see if you've done it right.)

Also note that, to give the correct normalization in the end, the sum of the weights must add up to the number of grid points (aka the average must be 1), although this may depend slightly on the FFT algorithm you use. When we calculate the synthesized beam, in the next step, make sure that the amplitude of the peak is always equal to 1.

Make a function to plot the weights (which will be important for confirming that they are symmetric).

Step 4: Apply an inverse FFT to the product of the visibilities and the weights, to get the 'image'. Set up a function to plot the image (and the 'synthesized beam', which can be found by inverse FFT-ing the weight function).

In principle, you now have everything you need to define a sky and a weighting scheme in a few lines of code, do the FFTs, and invoke the plotting functions to show the results. If you're working in Jupyter, it should all convienetly fit into one cell. Let's start looking at some test cases!

(Note that using scipy's fft package, I'm getting weird phase behaviour in some cases (particularly the delta functions). I don't understand it, but everything else, including the inverse transform, work fine. Bonus points to anyone who can figure out why.)

Part 2: Experiments

Let's go through a number of different test cases. For each, generate the figures (sky, amplitude and phase of visiblities, weights, image and synthesized beam) and make a few comments on the behaviour and what you expect based on Fourier transform theory. This table might be helpful.

1. First test case: delta function at the center of the image, with perfect sampling. How does the Fourier transform compare to the theoretical expectation?

2. Shifted delta function: as with the above example, but move the delta function a few pixels in some direction (again, perfect uniform sampling). How does this change things, and does it match Fourier transform theory? Play with moving the delta function by different amounts and in different directions. Any thoughts about what this implies for sources very far from the center of an observation?

3. Single delta-function source, with some limited sampling. Put in a delta function source somewhere (position shouldn't matter). Change the weight function to remove some parts of the *u,v* plane. Try removing long baselines (high frequencies) and see how that affects the resulting synthesized beam. Same with removing short baselines, or the mid-length baselines. Try a very sparse measurement (very few parts sampled). Make a few comments on the behaviour of the synthesized beam.

4. Let's look at extended sources now. Put a top-hat function in your image, and play with the weighting again as before. How do the visibilities look, and how do they change as a function of the width of the tophat (and why?)? How does the image of the top-hat look depending on what information is present/missing?

5. Let's start looking at the effects of noise in the visibilities. Create a sky with a delta function, and use a weighting that removes some of the shortest and longest baselines. Inject some noise into the visibilities by adding Gaussian random numbers (with some amplitude/width) at each grid point (note that noise should be hermitian, just like the data!). Comment on how the resulting image changes from the addition of noise. Also, commend on how the noise looks on small scales (adjacent pixels).

6. Finally, let's look at phase errors (from imperfect calibration). Again, let's start with a delta-function source and weights that remove the shortest and longest baselines. Without changing the amplitudes, introduce a random phase error (by multiplying by exp(i*phase_error)), which can be a uniform distribution of some width (say, +-10 degrees to start with). Again, make sure the visibilities are still Hermitian. Look at how the image changes as a result. Experiment with different widths of the phase error distribution to see how things change.