CLEANing Up Image Deconvolution for Radio Astronomy on the GPU

Nathan Sanders & Katherine Rosenfeld

CS205 Final Project

Overview:

Observational data in radio astronomy requires significant pre-processing before it can be interpreted scientifically. The most commonly-used procedures in this field are gridding, transforming the signal recorded in frequency space by the radio antenna array to an image in the plane of the sky, and CLEANing, deconvolving the image with the point spread function of the instrument. We have implemented standard algorithms for both these tasks on the GPU using pyCUDA, achieving speedups of ~5 for gridding and ~50 for CLEANing.

System requirements:

- CUDA enabled NVIDIA GPU
- pycuda
- scikits.cuda
- pyfits

Gridding:

Synthesis aperture arrays sample the Fourier transform of the sky brightness towards the source. In order to make a science image you must first invert these complex valued measurements, V(u,v), by taking a Fourier transform. The common technique is to grid the sampled data points and evaluate the discrete Fourier transform via an FFT. We grid using a convolution with a standard spheroidal function and use the CUDA SciKit wrapper for cuFFT to calculate the sky image and point source response function. We also include a rudimentary weighting scheme that can be used to accentuate different aspects of the data. We use the following kernels:

1. gridVis_wBM_kernel: Each pixel in the uv plane goes through the data and check to see whether the pixel is included in the convolution. This kernel also calculates the point spread function and the local sampling from the data (for applying the weights later).
2. wgtGrid_kernel: This will apply the weights.
3. dblGrid_kernel: To improve speed, the previous kernel is only evaluated for half the grid. Since the data is Hermitian (the sky brightness is purely real) we can figure out what the other half looks like afterwards.

4. nrmGrid_kernel/nrmBeam_kernel: This normalizes an array.
5. corrGrid_kernel: The corrects for the convolution we applied when gridding.
6. shiftGrid_kernel: This shifts a grid for computing an FFT.
7. trimIm_kernel: We pad the uv grid to improve image quality and this kernel trims it to the requested image size.

Hogbom CLEAN algorithm:

Radio antenna arrays typically have non-uniform spacing, which gives them access to low and high frequencies simultaneously, but with sparse sampling in the UV plane.   A Fourier tranform of this sparsely sampled data yields significant artifacts in the resulting image. For example, an image of a point source will have many sidelobes throughout the 2D image, which can reach intensities within an order of magnitude of the primary lobe.

This point source response pattern is called the **dirty beam**. The mage produced by convolving an astronomical source with the dirty beam is called the **dirty image** or dirty map. A **clean beam** is typically constructed by fitting a 2D Gaussian to the primary lobe of the dirty beam. If the locations of all astrophysical sources in the dirty image are known, a **clean image** can be approximated by convolving this model with the clean beam.

Hogbom (1974, Astronomy & Astrophysics, 15, 417) presented a classic image deconvolution algorithm used in radio astronomy to remove these instrumental artifacts. We implement the Hogbom algorithm on the GPU using pyCUDA ("**cuda_hogbom**" function). The algorithm uses 3 CUDA kernels:

1. find_max: The Hogbom algoritm begins by finding the highest intensity point (in terms of absolute value) on the dirty image. Our implementation uses the gpuarray.max an atomic exchange operator to find the maximum, and then the find_max kernel is called to identify the array index of the maximum and record the maximum value at this position on the image model.
2. sub_beam: In each iteration, the dirty beam is scaled, positioned at the maximum point found in step 1, and subtracted from the dirty image. Our sub_beam kernel performs this subtraction on the gpu, and simultaneously adds the scaled and shifted clean beam to the clean image.
3. add_noise: The final step in the Hogbom algorithm is to add the residuals from the dirty image, after iterative beam subtraction, to the model cleaned image as 'noise.' We do this using an element wise pyCUDA kernel.

Data input

gICLEAN.py is executed as a python script from the command line:

```
python gICLEAN.py [example] [image size] [make figures]
```

The command line options can also edited within the main routine:

- [example]: Choose a dataset to image. We provide the options of a ‘gaussian’, ‘ring’, ‘mouse’, or ‘hd163296’. All except the last are simulated datasets. The default is ‘gaussian’.
- [image size]: The number of pixels on an image side. The default is 1024.
- [make figures]: 0/1 toggles whether or not summary images are created. The default is 0=False.

This will compile and run the CUDA kernels and generate final images. It requires an ALMA style visibility dataset in FITS format that is set by the *vfile* keyword. This version is good for single channel images only (although it is in principle scalable for spectral lines or multi-frequency synthesis). We have supplied several examples along with reasonable imaging parameters in our code with the data available on the website. The gaussian, ring, and mickey examples are noiseless, simulated datasets while hd163296 is a single channel of the CO J=3-2 line observed by the sub-millimeter telescope ALMA.

Usage

First, set up the environment:

```
gpu-login
module load courses/cs205/2012
```

To perform gridding and CLEANing for the Gaussian source example, simply run:

```
python gICLEAN.py
```

This script takes a few command line options:

```
python gICLEAN.py [EXAMPLE] [ISIZE] [PLOTME]
```

- *example*: Use this to select which example dataset to operate on. Options are “gaussian,” “ring,” “mouse,” and “hd163296.” **FITS files containing visibilities (raw data) for each example to be gridded and CLEANed are stored in /scratch/global/NSKR.**
- *ISIZE:* The output image size, in pixels. This is ‘1024’ by default; to make e.g. an image four times larger on each size, choose ‘4096’.
- *PLOTME*: Turn this on to generate various png plots using matplotlib. This slows down the code, because it needs to grab data from the GPU and do the plotting, but is useful for debugging and data display.

Within this module, there are several functions that have their own configurable options, although these are not given command line switches.

cuda_gridvis parameters:

- settings: This is a Python dictionary containing the following imaging parameters:
    - *vfile*: A string with the location of the dataset (see Data Input section)
    - *cell*: The size of a single pixel in arcseconds.
    - *imsize*: The number of pixels on an edge. Images are square and must be a power of 2.
    - *briggs*: This controls the visibility weights. A larger number corresponds to better sensitivity while a lower number will improve resolution.

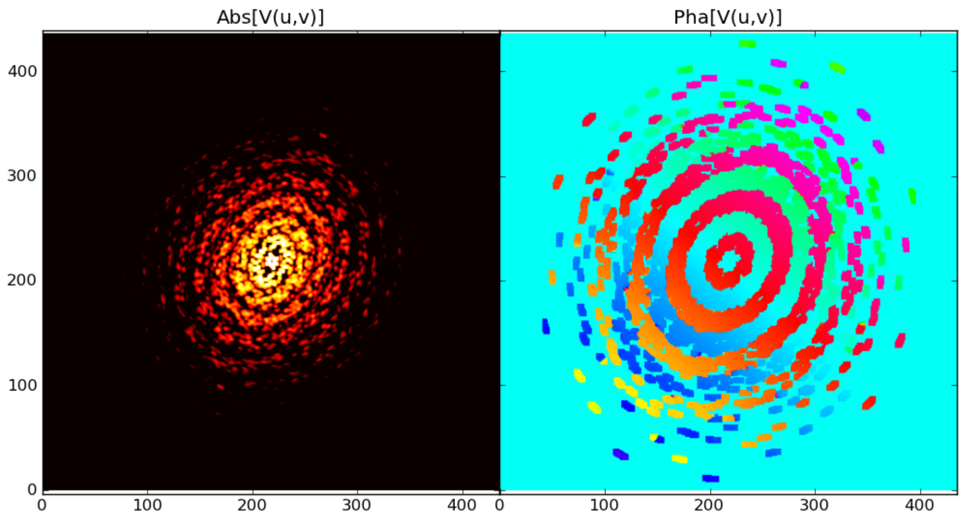
- plan: This is the plan for cuFFT to follow.

cuda_hogbom parameters:

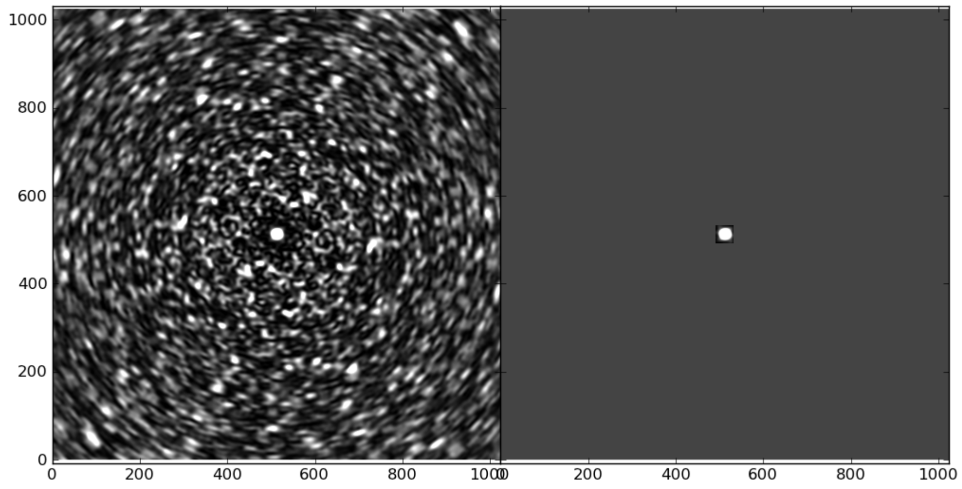
- *thresh*: This defines the stopping threshold for the Hogbom algoirthm, Iterations continue until the maximum intensity point on the dirty image is a factor less than *thresh* of the maximum in the original dirty image. The default value is *thresh*=0.2.
- *gain:* This defines the "strength" of each iteration. At each iteration, the dirty beam is multiplied by *gain* before subtraction from the ditty image. By default, *gain*=0.1, meaning that 10 iterations are required to fully clean an image with a single point source.

Example

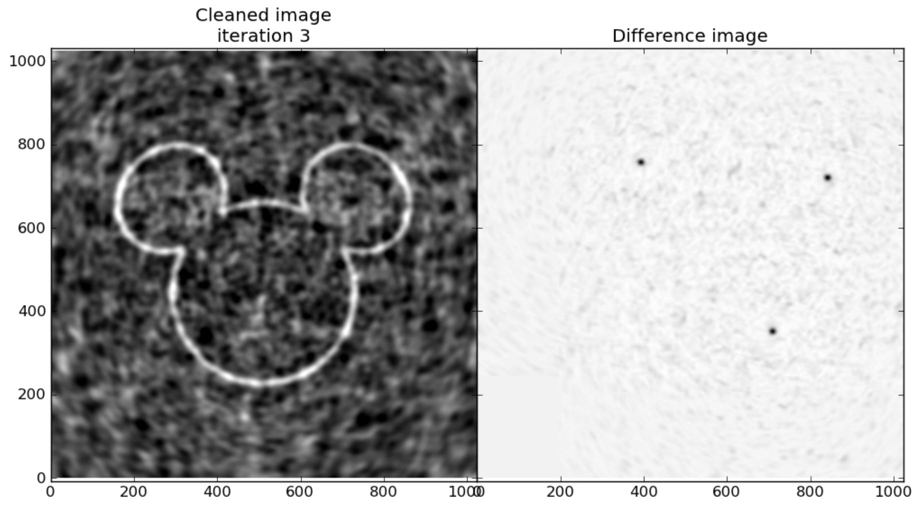
The input data are complex valued and must be gridded (this example is an offset, inclined



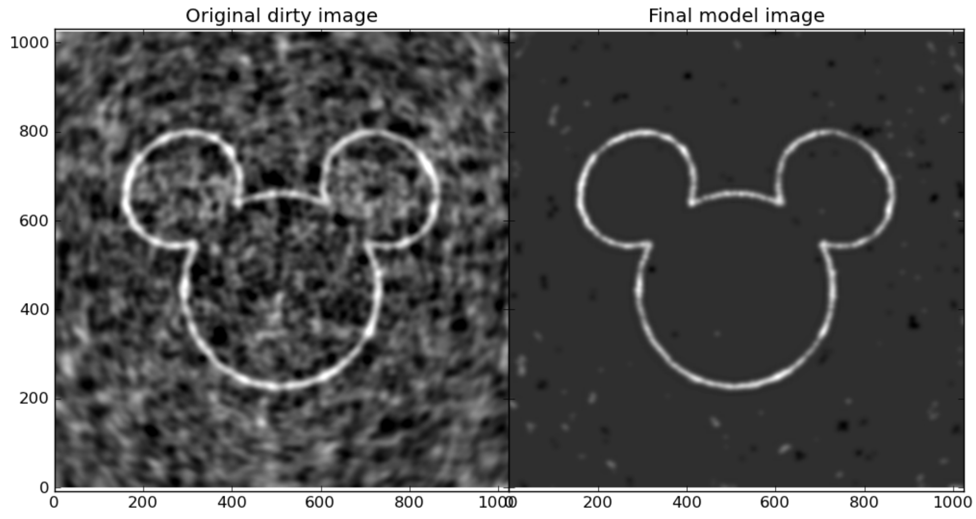
ring):
The sampling pattern in the complex plane determines the shape of the dirty beam shown below (left) and is approximated by a clean beam (right), which preserves only the primary lobe.



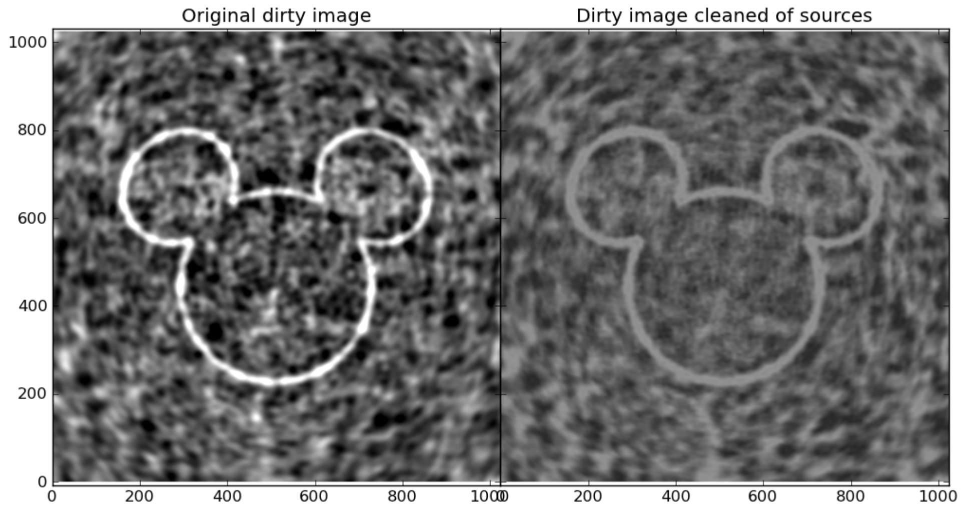
The Hogbom algorithm iteratively finds the pixel with maximum absolute intensity on the dirty image and subtracts the dirty beam shifted and scaled to that location/intensity. Below we show a simulated dirty image of the fictional astrophysical sources ϵ Mouse (left) and the first three iterations of the dirty beam subtraction (right).



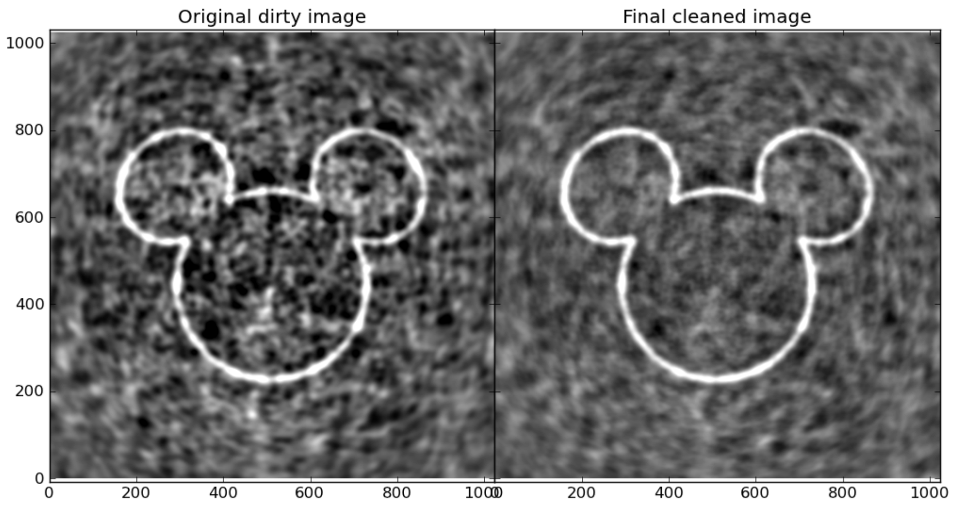
The Hogbom algorithm converges to within the specified threshold (0.2 times the initial maximum intensity, in this case) after 2700 iterations. At each iteration, it has added a point source to its model for the image, shown below at right. The model closely resembles the astrophysical source ϵ Mouse:



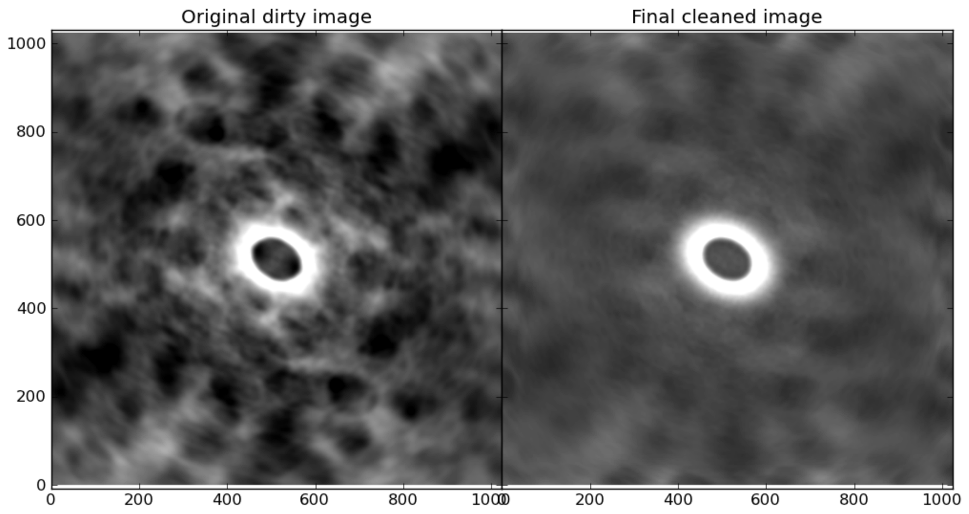
After removing these 2700 “point sources” from the dirty image, the source ϵ Mouse is essentially completely removed (flux is zero, shown as gray below) and the variance in the background, produced by the sidelobes of the dirty beam convolved with the source, is much lower (less negative/positive; more gray).



Finally, the remaining background variance is added to the model image, convolved with the clean beam, to produce the final, CLEANed image (right). In this image, the effects of the sidelobes of the dirty beam are largely removed. When compared with the original dirty image (left), the source ϵ Mouse is more clear and the variance in the background is significantly smaller.



As an additional example of a more realistic simulated astrophysical source, see the dirty (left) and final, CLEANed version (right) of a simulated disk, below.



Performance:

Here are timings output by our GPU code for the source ϵ Mouse :

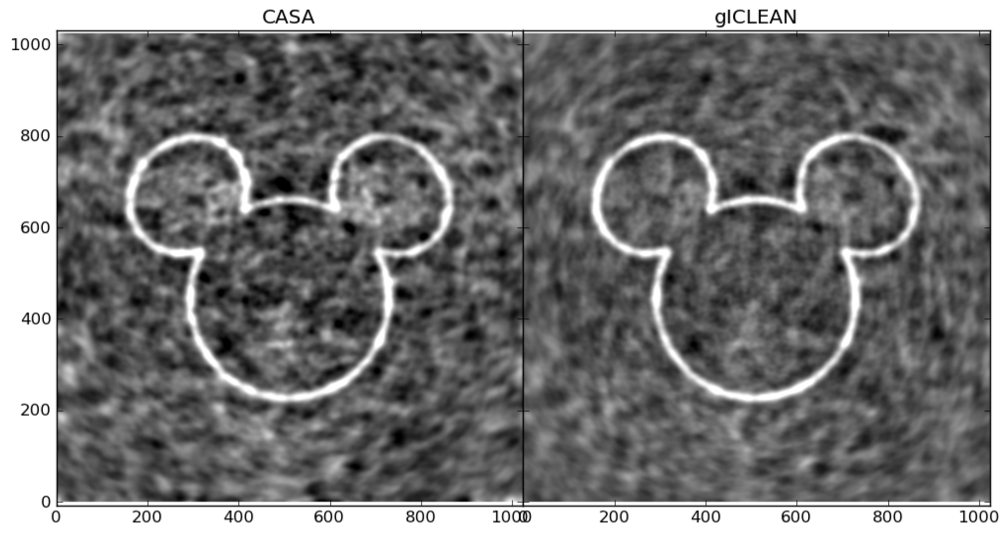
Gridding execution time 2.12 ±0.06s
0.0000096 s per visibility
Hogbom execution time 3.27±0.02 s
0.0013 s per iteration

A reference implementation of the Hogbom algorithm in numpy took:

```
Serial execution time 251.6380  s
        0.0908  s per iteration
```

The standard radio astronomy software CASA took 14.4$\pm$0.07 s for both gridding and CLEANing.

We therefore achieve a Hogbom speedup of 70x compared to the reference numpy implementation, which is an identical CLEAN algorithm. Our routine has a 2.7x speedup compared to the CASA implementation, which apparently is already significantly optimized compared to the reference implementation. The final CLEANed images from our implementation and CASA are compared below:



The execution time per visibility is independent of the source's structure but does scale with the number of samples as well as the sampling coverage. The time per Hogbom iteration should be independent of all characteristics except image dimensions. Indeed, with we find similar timings with all our examples using 1024x1024 image dimensions. Increasing the image dimensions, we find that the gridding execution time remains approximately the same, while the Hogbom execution time grows approximately linearly with the number of pixels.

See also:

- A description of the Hogbom algorithm by the National Radio Astronomy Observatory: http://www.cv.nrao.edu/~abridle/deconvol/node8.html
- Hogbom's original 1974 paper: http://adsabs.harvard.edu/abs/1974A%26AS...15..417H
- The standard radio astronomy data reduction package, CASA: http://casa.nrao.edu/
- A reference implementation of the Hogbom algorithm in numpy: http://www.mrao.cam.ac.uk/~bn204/alma/python-clean.html

Finally, note that the following is printed at the end of each run of the code:

```
Exception scikits.cuda.cufft.cufftInvalidPlan: cufftInvalidPlan() in <bound method
Plan.__del__ of <scikits.cuda.fft.Plan instance at 0xd56dcf8>> ignored
```

This exception does not seem to affect the execution of the code in any way.