Image Deconvolution Research Final Report

Bowen Jiang, Maohao Shen

December 2019, Modified in January, May 2020

1 Introduction

The ultrasonic localization microscopy has been used for medical vascular imaging for years [7], in order to better visualize and understand the geometric shapes of the vascular. Because tissues with different acoustic impedance have different ultrasonic wave reflection properties, we can inject certain density of inert gas micro bubbles into the vascular, to increase the resulted imaging contrast, and we can use these locations of bubbles in received images to estimate the shapes of the vascular.

However, there are some issues to localize these bubbles. The ultrasonic waves travelling through the tissues experience various kinds of losses, such as attenuation, diffraction, absorption, and so on. These losses distort and blur the received signals, and the blurring kernel can be mathematically approximated by the point spread functions (PSF). With the assumptions that the PSF is known, the PSF can be either spatially invariant and spatially varying. Hereby, what we want to do is to recover the clean deconvolved images to better estimate the true geometric shapes of the vascular.

Throughout the semester, we have studies and experimented on both spatially invariant and varying image deconvolution with known PSF, and explored various methods especially the Hogbom Clean Algorithm [1], Matching Pursuit Algorithm (MP) [6], and Orthogonal Matching Pursuit Algorithm (OMP) [2,3] to recover the clean image. Their objective functions can be kind of different, which we will show in the **section 4.1**.

In this final report, we will review these algorithms in **Section 2 and 3**, including experiments, and focus on comparisons. We will mathematically proof that the CLEAN Algorithm and MP are theoretically identical, and OMP behaves as an advanced version of MP with faster convergence but more computations in **Section 4**. The future plans will be briefly shown in **Section 5**.

2 Spatially Invariant Known PSF

2.1 The Hogbom Clean Algorithm

The Hogbom CLEAN algorithm is proposed by J.A. Hogbom in 1974 at Stockholm Observatory to solve for high resolution astronomical images. Similarly, we are going to solve the deconvolution problem for vascular bubble images, which also has sparse features as astronomical images. The clean beam is used by the original CLEAN algorithm to achieve astrophysically reasonable images without spurious high frequencies, but is not used in our experiment, because our propose is micro bubble localization for vascular imaging, where only locations instead of shapes of bubbles are important.

Define the notation $\circledast$ as circular convolution, the problem can be defined as below

$$\min \|\mathbf{x}\|_0 \qquad \text{s.t. } \|\mathbf{y} - \mathbf{b} \circledast \mathbf{x}\|_2 \leq \epsilon \tag{1}$$

where $\mathbf{y}$ is the dirty map observed by the ultrasonic localization microscopy, $\mathbf{b}$ is the known spatially-invariant point spread function (PSF), or called the dirty beam, experienced through the imaging, and $\mathbf{x}$ is the clean map with high resolution that we want to reconstruct.

2.1.1 Theory of Hogbom CLEAN Algorithm

The CLEAN algorithm has an iterative procedure similar to the general greedy algorithm. A constant scaling factor called loop gain $\gamma \in (0, 1]$ is used for updating both $\mathbf{x}$ and $\mathbf{y}$ in each iteration. A constant threshold ϵ is set as the stop condition to discriminate between expected noise and signals, dependent to the experiment. Define the notation $\star$ as cross-correlation, and w_1, w_2 as coordinates in maps. The $\mathbf{b}(w_1, w_2)$ is the shifted dirty beam $\mathbf{b}$ centered at location (w_1, w_2), and $\delta[w_1, w_2]$ is the shifted Kronecker delta at location (w_1, w_2).

In general,CLEAN algorithm is to find the coordinates and values of the highest dirty beam from dirty map $\mathbf{y}$, subtract it from $\mathbf{y}$, and re-build the peak on clean map $\mathbf{x}$. Iterate until the highest peak is below some threshold.

Input: dirty map $\mathbf{y}$, dirty beam $\mathbf{b}$, loop gain γ, threshold ϵ;
Output: clean map $\mathbf{x}$;
Initialize $\mathbf{x} = 0 \cdot \mathbf{y}$ *empty clean map*;
while $\max\{\mathbf{y}\} > \epsilon$ **do**
  $(w_1, w_2) = \text{argmax}_{w_1, w_2}\{\mathbf{b} \star \mathbf{y}\}$ *find coordinates of peak*;
  $\text{amplitude} = \gamma \cdot \max\{\mathbf{b} \star \mathbf{y}\}$ *find amplitude of peak*;
  $\mathbf{y} = \mathbf{y} - \text{amplitude} \cdot \mathbf{b}(w_1, w_2)$ *update dirty map*;
  $\mathbf{x} = \mathbf{x} + \text{amplitude} \cdot \delta[w_1, w_2]$ *update clean map*;
end
return x

Algorithm 1: Pseudo-code for the CLEAN Algorithm

2.1.2 Experiment Results

In this experiment, we use the software Field-II version 3.24 Windows to generate spatially invariant dirty beam **b** under linear array scenario in Figure 1, and ground truth map in Figure 2. The rest of experiment is run in Python 3.

Field II is an ultrasonic simulation program, which generates micro bubbles and simulates point spread functions experienced by these bubbles in practice (link: *field-ii.dk*). In spatially invariant case, a single **b** is generated in the center of a 900x900-pixel image by Field II "PSF.m" function file in Matlab, and is applied to all bubbles in ground truth image. Detailed parameter settings can be found on code in Section 7.4 'PSF.m'. Zoomed in (366x366 $<$ actual 900x900 pixels) of **b** is shown in Figure 1 for display convenience.

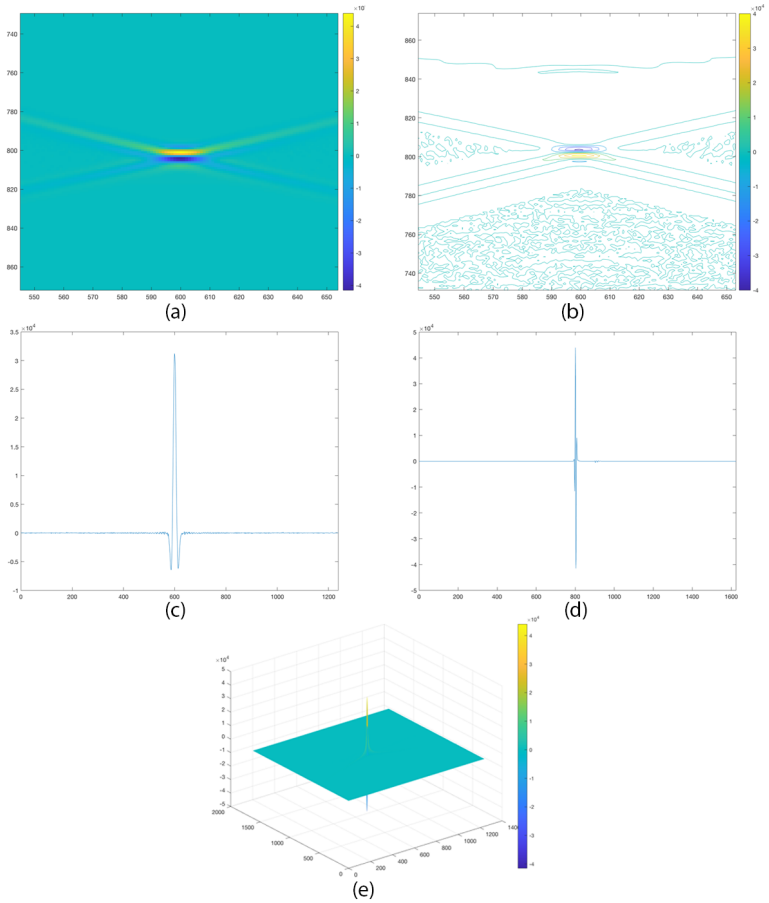

Figure 1: Zoomed in dirty beam **b** (a) original image (b) 2D contour map (c) 1D horizontal cut (d) 1D vertical cut (e) 3D surface plot

We normalize the PSF as $\mathbf{b} = 255 * \mathbf{b}/(\max(\mathbf{b}) - \min(\mathbf{b}))$ for better numerical properties, while keeping 0 at 0. After PSF is available, we do 2D circular convolution to obtain dirty map $\mathbf{y}$, with additional Gaussian random noise.

$$\mathbf{y} = \begin{cases} \text{ground truth} \circledast \mathbf{b} + \text{Gaussian}\,(\mu = 20, \sigma = 10) & \text{if value} \in [0, 255] \\ 255 & \text{if value} > 255 \\ 0 & \text{if value} < 0 \end{cases} \tag{2}$$

Then we implement the CLEAN Algorithm (Algorithm 1) to obtain clean map $\mathbf{x}$, using parameter $\gamma = 0.8$ and $\epsilon = 50$. Figure 2 is our result.

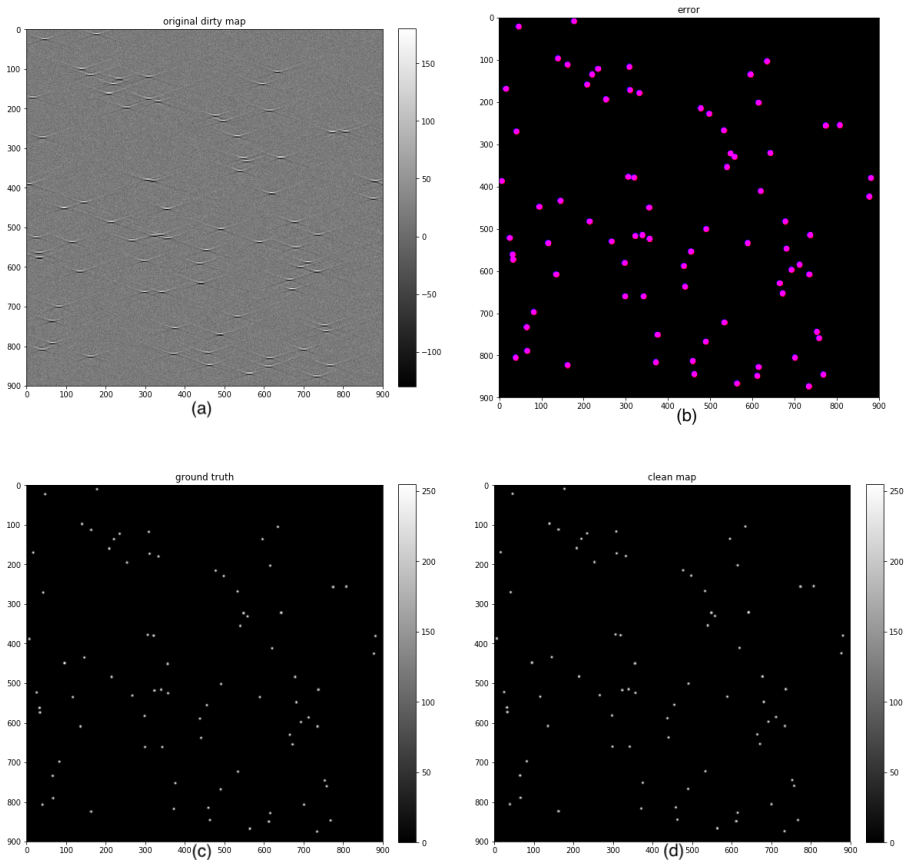

Figure 2: (a) dirty map $\mathbf{y}$ (b) ground truth is colored red, detected clean map is colored blue (c) ground truth, convolved with clean Gaussian beam only for display purpose (d) final clean map $\mathbf{x}$ convolved with clean beam

Accuracy ≈ 100. Note that it is accuracy of this sample. It can be seen that in Figure2(c), all blue and red dots are overlapped, so only purple dots are visible.

2.2 Matching Pursuit Algorithm

Define the same notations for $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{b}$ as in Section 2.1. If vectoring dirty beams $\mathbf{b}$ as columns, we can store every shifted $\mathbf{b}$ at all possible N locations in a dictionary $\mathbf{D} \in \mathbb{R}^{N \times N}$, where N is total number of pixels in $\mathbf{y}$. Each column of $\mathbf{D}$ is also called an atom $\mathbf{d}$.

Matching Pursuit is also a greedy algorithm. The problem can be defined as

$$\|\mathbf{y} - \mathbf{D}\mathbf{x}\|_2 \approx 0 \qquad \text{s.t. } \|\mathbf{x}\|_0 = m \tag{3}$$

where m is the sparsity, defined as the number of atoms selected from $\mathbf{D}$, and it is enforced that $m << N$. The stopping condition we used in this experiment is sparsity m, it is also reasonable to choose threshold ϵ as stopping condition.

2.2.1 Theory of Matching Pursuit Algorithm

Matching Pursuit is a greedy algorithm to discover the most correlated atom $\mathbf{d}$ from $\mathbf{D}$ one at a time iteratively, calculate the residual $\mathbf{r}$ by subtracting $\mathbf{d}$, and re-build the clean map $\mathbf{x}$ from index set in the end. The index set $\mathbf{\Lambda}$ contains indices of all atoms already selected from ϕ, the corresponding atoms in the atom set $\mathbf{\Phi}$, and corresponding amplitudes in amplitude set $\mathbf{a}$. Denote the subscript t as the time step, and $\delta(i)$ as the shifted Dirac delta function at position i.

The idea behind this algorithm seems similar to the Hogbom Clean algorithm in Section 2.1, and we will have a more detailed comparison between them and proof their equivalence in Sections 4.1.

Input: dirty map $\mathbf{y}$, dictionary $\mathbf{D}$, sparsity m ;
Output: clean map $\mathbf{x}$;
Initialize residual $\mathbf{r}_0 = \mathbf{y}$, index set $\mathbf{\Lambda}_0 = \emptyset$, atom set $\mathbf{\Phi}_0 = \emptyset$, amplitude set $\mathbf{a}_0 = \emptyset$, $t = 0$;
while $t < m$ **do**
  $\lambda_t = \arg\max_{j=1,...,N} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle|$ *index of most correlated atom*;
  $\mathbf{a}_t = \mathbf{a}_{t-1} \cup \{\max_{j=1,...,N} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle|\}$;
  $\mathbf{d}_t = \mathbf{d}_{\lambda_t}$ *find most correlated atom*;
  $\mathbf{\Lambda}_t = \mathbf{\Lambda}_{t-1} \cup \{\lambda_t\}$ *update index set*;
  $\mathbf{\Phi}_t = [\mathbf{\Phi}_{t-1} \quad \mathbf{d}_t]$ *update atom set*;
  $\mathbf{r}_t = \mathbf{r}_{t-1} - |\langle \mathbf{r}_{t-1}, d_t \rangle| \cdot \mathbf{d}_t$ *update residual*;
  $t = t + 1$;
end
for $j = 1, ..., length\{\mathbf{\Lambda}_t\}$ **do**
  $\mathbf{x} = \mathbf{x} + \mathbf{a}_t(j) \cdot \delta(\mathbf{\Lambda}_t(j))$ *build clean map*;
end
return $\mathbf{x}$;

Algorithm 2: Pseudo-code for Matching Pursuit

2.3 Orthogonal Matching Pursuit Algorithm

Define the same notations as in Section 2.2. The deconvolution problem we are going to solve is the same as Matching Pursuit.

$$\|\mathbf{y} - \mathbf{D}\mathbf{x}\|_2 \approx 0 \qquad \text{s.t. } \|\mathbf{x}\|_0 = m \tag{4}$$

2.3.1 Theory of Orthogonal Matching Pursuit Algorithm

The Orthogonal Matching Pursuit can be regarded as an advanced version of the Matching Pursuit. It is also an greedy algorithm to discover the best matching atom $\mathbf{d}$ one at a time iteratively from $\mathbf{D}$, but calculate the residual $\mathbf{r}$ in each iteration by solving a least-squares problem.

The intuition of Orthogonal Matching Pursuit is to find the atom that is most correlated with residual, and use all atoms selected so far to estimate the dirty map. Sparse coding $\tilde{\mathbf{x}}$ stores coefficients of all selected atoms. Then the current residual $\mathbf{r}$ is the difference between the true dirty map $\mathbf{y}$ and the current estimated dirty map $\hat{\mathbf{y}}$. Detailed advantages compared with MP can be found in Section 4.2.

Input: dirty map $\mathbf{y}$, dictionary $\mathbf{D}$, sparsity m ;
Output: clean map $\mathbf{x}$;
Initialize residual $\mathbf{r}_0 = \mathbf{y}$, index set $\mathbf{\Lambda}_0 = \emptyset$, atom set $\mathbf{\Phi}_0 = \emptyset$, $t = 0$;
while $t < m$ **do**
  $\lambda_t = \arg\max_{j=1,\dots,N} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle|$ *index of most correlated atom*;
  $\mathbf{d}_t = \mathbf{d}_{\lambda_t}$ *find most correlated atom*;
  $\mathbf{\Lambda}_t = \mathbf{\Lambda}_{t-1} \cup \{\lambda_t\}$ *update index set*;
  $\mathbf{\Phi}_t = [\mathbf{\Phi}_{t-1} \quad \mathbf{d}_t]$ *update atom set*;
  $\tilde{\mathbf{x}}_t = \text{argmin}_{\tilde{\mathbf{x}}} \|\mathbf{\Phi}_t \tilde{\mathbf{x}} - \mathbf{y}\|_2$ *find sparse coding* $\tilde{\mathbf{x}}$ *by* $\mathbf{\Phi}_t$;
  $\hat{\mathbf{y}}_t = \mathbf{\Phi}_t \tilde{\mathbf{x}}_t$ *estimate dirty map* $\hat{\mathbf{y}}$ *by* $\mathbf{\Phi}_t$;
  $\mathbf{r}_t = \mathbf{y} - \hat{\mathbf{y}}_t$ *update residual*;
  $t = t + 1$;
end
for $j = 1, \dots, \mathit{length}\{\mathbf{\Lambda}_t\}$ **do**
  $\mathbf{x} = \mathbf{x} + \tilde{\mathbf{x}}_{t-1} \cdot \delta(\mathbf{\Lambda}_t(j))$ *build clean map*;
end
return $\mathbf{x}$;

Algorithm 3: Pseudo-code for Orthogonal Matching Pursuit

Note that in the above algorithm, we have the equation

$$\tilde{\mathbf{x}}_t = \text{argmin}_{\tilde{\mathbf{x}}} \|\mathbf{\Phi}_t \tilde{\mathbf{x}} - \mathbf{y}\|_2 \tag{5}$$

This can be solved by the left inverse of $\mathbf{\Phi}_t$, denoted by $\mathbf{\Phi}_t^\dagger$, as below

$$\tilde{\mathbf{x}}_t = \text{solution of } \mathbf{\Phi}_t^\dagger \mathbf{\Phi}_t \tilde{\mathbf{x}} = \mathbf{\Phi}_t^\dagger \mathbf{y} \tag{6}$$

2.3.2 Experiment Results

Only results for OMP is shown, because MP is identical to CLEAN algorithm, and we will prove their equivalence in Section 4.1. We generate the spatially invariant dirty beam $\mathbf{b}$ and 300x300-pixel ground truth image by Field II, the same procedure in Section 2.1.2. We set the parameter $m = 36$, approximately the number of bubbles in ground truth. The experiment is run in Python 3.

There are some simplifications to enhance the efficiency. With the assumption that PSF is spatially invariant, and the only difference between different PSF is their center location. It is unnecessary to store all PSFs into a large dictionary $\mathbf{D}$. Instead, once we have already generate a single $\mathbf{b}$ by Field II, we can shift that PSF to different locations when needed in Matlab. Therefore, for the following step in Algorithm 3,

$$\lambda_t = \arg \max_{j=1,\dots,N} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle| \quad \forall\ \mathbf{d}_j \in \mathbf{D} \tag{7}$$

we don't generate all distinct $\mathbf{d}_j$ by Field II and form $\mathbf{D}$. It is modified as

Generate a single $\mathbf{b}$ in image center by Field II ;
result = array[N] ;
for *j=1,...,N* **do**
  shift $\mathbf{b}$ to the location j, denoted as $\mathbf{b}_j$ $\quad$ (the generated $\mathbf{b} = \mathbf{b}_{N/2}$);
  result$[j] = |\langle \mathbf{r}_{t-1}, \mathbf{b}_j \rangle|$;
end
$\lambda_t = \max_j \{\text{result}\}$

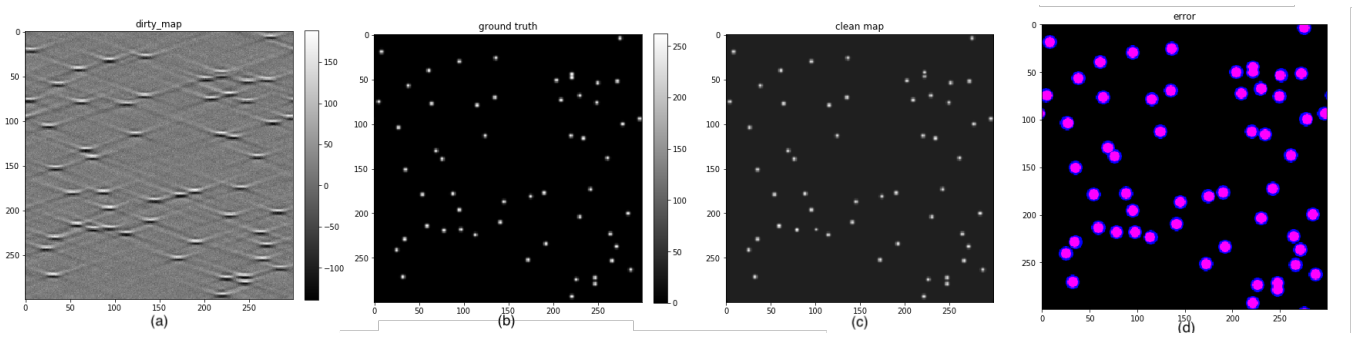

Figure 3: (a) dirty map $\mathbf{y}$ (b) ground truth,convolved with clean Gaussian beam only for display purpose (c) final clean map $\mathbf{x}$ convolved with clean beam (d) ground truth is colored red, detected clean map is colored blue

Accuracy $\approx 100\%$ Note that it is accuracy of this sample. It can be seen that in Figure3(d), all blue and red dots are overlapped, and almost all are concentric as purple dots. Empirically OMP and MP(CLEAN) have similar performance in this experiment. Comparison in theory of them will be in Section 4.2.

3 Spatially Varying Known PSF

In Section 2, we assumed that the PSF is spatially invariant. However, in practice, bubbles at different spatially locations will have different blurring and distortions. Therefore, we need to access the various shapes of the PSF at all possible different spatially locations to solve the deconvolution problem.

Theoretically, all three methods mentioned before work for the spatially varying case, but they have different computational efficiency in practice. Either MP or OMP algorithm has a large dictionary to store all spatially varying PSFs at all different possible locations, which is expensive, even for spatially invariant case. Meanwhile, even if we don't store the PSFs but generate them when needed, the equation in Section 2.1.1

$$\lambda_t = \arg \max_{j=1,\dots,d} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle| \tag{8}$$

requires correlations between the residual and every atom in every iteration, which is huge amount of work. Given one PSF available, generating shifted PSFs with same shape at different location is fast enough. However, generating spatially varying PSF requires function call by Field II, which takes too much time so that the work is impractical in this case. Similarly, storing all PSFs in local disks also requires long accessing time and huge memory space.

Therefore, we want to use the CLEAN algorithm (Algorithm 4). It finds the peak location (y_1, y_2) by correlation with an "cropped" dirty beam $\tilde{\mathbf{b}}$ without side lobes (9) in Figure 5, because doing correlation with all different varying PSFs at different locations in each iteration is computationally costing. We assume that $\tilde{\mathbf{b}}$ is able to detect most bubbles. Then we use Field II while running the algorithm to generate the exact shaped spatially varying PSF $\mathbf{b}(y_1, y_2)$ at that location, which will be subtracted from the dirty map $\mathbf{y}$. It can be seen that we only need to generate PSF once in each iteration, instead of N times. Everything else is the same as spatially invariant case. Parameter used are $\gamma = 1.2, m = 100$, and the experiment is run in Matlab R2019b.

Define maximum value of side lobes of $\mathbf{b}$ to be $\beta \approx 7000$. A single $\mathbf{b}$ is generated by Field II at the image center, and then the value of $\tilde{\mathbf{b}}$ at index i, j can be defined as

$$\tilde{b}_{ij} = \begin{cases} b_{ij} & \text{if } |\tilde{b}_{ij}| > \beta \\ 0 & \text{else} \end{cases} \tag{9}$$

Intuitively, $\tilde{\mathbf{b}}$ is the cropped version of a single PSF $\mathbf{b}$ among all spatially varying PSFs, remaining only the center elliptical part without side lobes. It can be roughly used in computing cross-correlations to represent all kinds of different PSFs, and works well empirically in experiments.

The residual in Figure 3 (c) is defined as

$$\mathbf{r} = \mathbf{y} - \sum_{(y_1,y_2),\text{amplitude}} \text{amplitude} \cdot \mathbf{b}\,(y_1, y_2) \tag{10}$$

with the same notation in Section 2.1.1, and the 'amplitude' is the amplitude of each shifted $\mathbf{b}$ same as in Section 2.1.1. In other words, it is the dirty map after subtracting all detected dirty beams.

Input: dirty map $\mathbf{y}$, "cropped" dirty beam $\tilde{\mathbf{b}}$, loop gain γ, threshold ϵ;
Output: clean map $\mathbf{x}$;
Initialize $\mathbf{x} = 0 \cdot \mathbf{y}$, open access to *Field II*;
while $\max\{\mathbf{y}\} > \epsilon$ **do**
  $(m_1, m_2) = \text{argmax}\left\{\tilde{\mathbf{b}} \star \mathbf{y}\right\}$;
  $\text{amplitude} = \gamma \cdot \max\left\{\tilde{\mathbf{b}} \star \mathbf{y}\right\}$;
  generate exact shaped $\mathbf{b}\,(m_1, m_2)$ by *Field II*;
  $\mathbf{y} = \mathbf{y} - \text{amplitude} \cdot \mathbf{b}\,(m_1, m_2)$;
  $\mathbf{x} = \mathbf{x} + \text{amplitude} \cdot \delta\,[m_1, m_2]$;
end
return x

Algorithm 4: Pseudo-code in spatially varying case

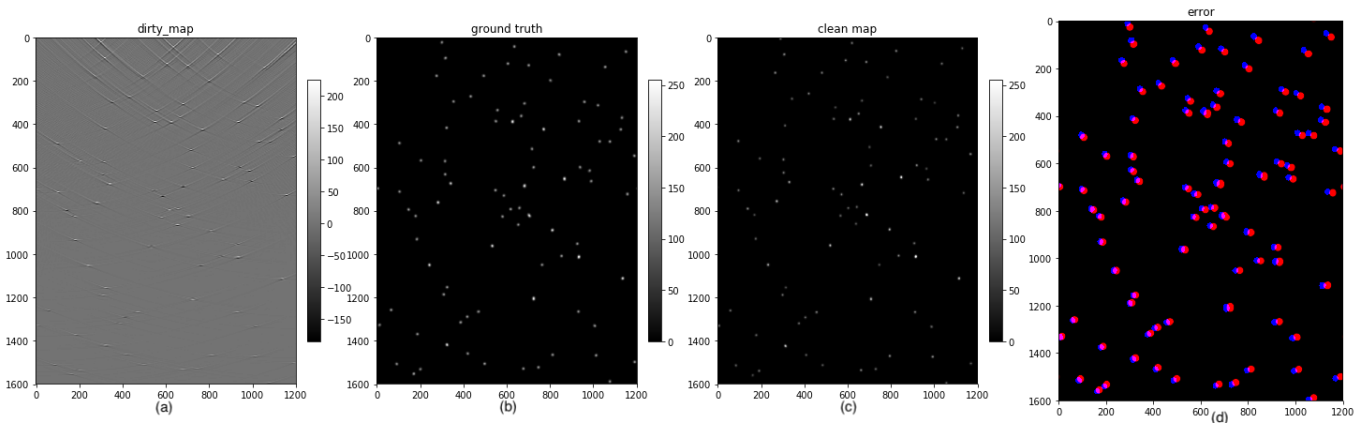

Figure 4: (a) dirty map $\mathbf{y}$, (b) ground truth, convolved with clean Gaussian beam only for display purpose (c) clean map $\mathbf{x}$, convolved with clean beam (d) ground truth is colored red, detected clean map is colored blue

In the error image in Figure4(d), bubbles in the image center mostly overlapped, but some bubbles in the outer regions are partially overlapped. The clean Gaussian beam used for display purpose has radius $\sigma = 3$, so half-overlapped bubbles have peaks misaligned by 3. We believe that these error

comes from the usage of cropped dirty beam as in (9), which is the balance between accuracy and computational efficiency, as explained earlier in this section.

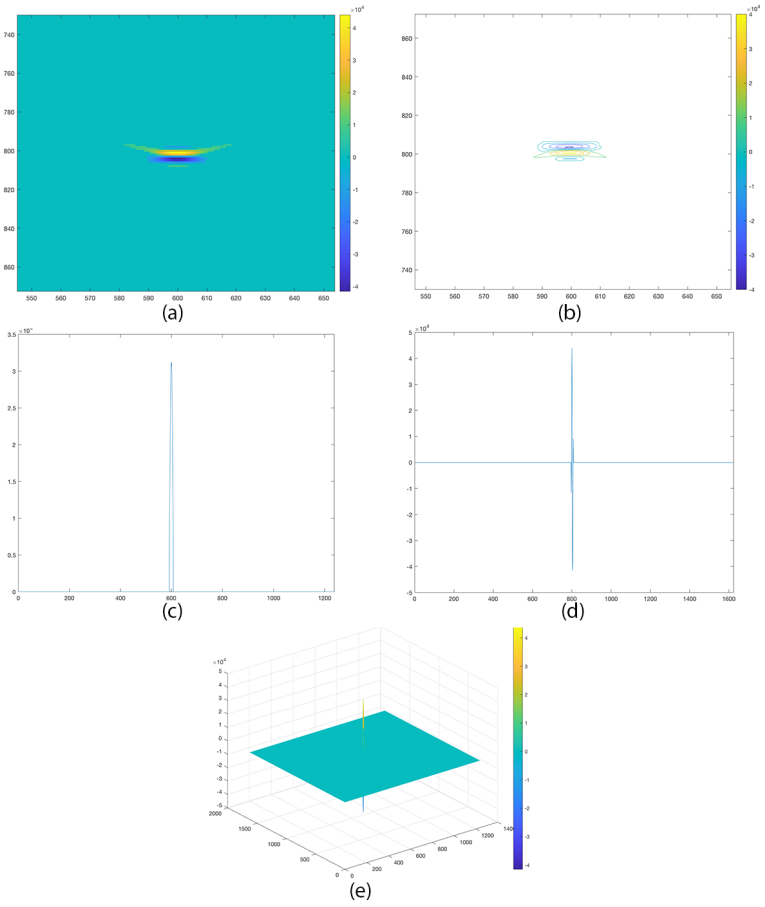

Figure 5: Zoomed in cropped dirty beam **b** (a) original image (b) 2D contour map (c) 1D horizontal cut (d) 1D vertical cut (e) 3D surface plot

4 Comparison of Algorithms

4.1 Proof the equivalence of the Clean and MP Algorithm

Comparing the Hogbom Clean Algorithm and the Matching Pursuit Algorithm, we found they have differences in practical implementation efficiencies as mentioned in section 3, but in this section we will prove that they are identical in terms of the mathematical theory.

• First, we can use a objective function to represent the problem that MP and CLEAN will solve. Here we suppose the stopping condition for MP and CLEAN are same (both use threshold ϵ not sparsity m) for comparison convenience, though both ways are acceptable in experiment:
For matching pursuit:

$$\|\mathbf{y} - \mathbf{D}\mathbf{x}\|_2 \leq \epsilon \qquad \text{s.t. } \min \|\mathbf{x}\|_0 \tag{11}$$

where $\mathbf{y}$ is vectorized dirty map image, $\mathbf{D}$ is the dictionary with shifted PSF as each column. $\mathbf{x}$ is the clean map, or we can call it sparse coefficients in sparse coding.

For CLEAN:

$$\|\mathbf{y} - \mathbf{b} \circledast \mathbf{x}\|_2 \leq \epsilon \qquad \text{s.t. } \min \|\mathbf{x}\|_0 \tag{12}$$

where $\mathbf{y}$ is vectorized dirty map image, b is the PSF function, or dirty beam. $\mathbf{x}$ is the clean map we want to reconstruct.

For these two algorithms, we want to solve the corresponding optimization problems (11) and (12) to find a clean image $\mathbf{x}$ (or sparse coefficients), providing a sparse representation as close as possible to original clean image $\mathbf{y}$, while requiring the number of non zero elements in $\mathbf{x}$ to be small(sparsity constraints). Besides, representing the i-th column of dictionary $\mathbf{D}$ as shifted PSF function $\mathbf{d}_i = \mathbf{b}[n-i]$, then we can rewrite $\mathbf{D}\mathbf{x}$ as :

$$\mathbf{D}\mathbf{x} = \sum_{i=1}^{d} \mathbf{x}[i]\mathbf{b}[n-i] \tag{13}$$

$$= \mathbf{b} \circledast \mathbf{x} \tag{14}$$

Thus, the optimization problems that Matching Pursuit and the CLEAN are solving identical.

• Second, let's compare the details of algorithms for solving above objective functions:
[1] compare the step for finding the maximum value index λ_t:

For Matching Pursuit using inner product:

$$\lambda_t = \arg\max_{j=1,\ldots,d} |\langle \mathbf{r}_{t-1}, \mathbf{d}_j \rangle| \tag{15}$$

$$= \arg\max_{j=1,\ldots,d} |\langle \mathbf{r}_{t-1}, \mathbf{b}[n-j] \rangle| \tag{16}$$

For CLEAN using correlation:

$$\lambda_t = \arg\max_{j=1,\ldots,d} |\mathbf{r}_{t-1} \star \mathbf{b}| \tag{17}$$

$$= \arg\max_{j=1,\ldots,d} \left| \sum_{n=1}^{d} \mathbf{r}_{t-1}[n]\mathbf{b}[n-j] \right| \tag{18}$$

$$= \arg\max_{j=1,\ldots,d} |\langle \mathbf{r}_{t-1}, \mathbf{b}[n-j] \rangle| \tag{19}$$

Thus, Matching Pursuit and the CLEAN algorithm use the same procedure to find the maximum value as well as the micro bubble peak index.

[2] compare the step for updating residual:
For Matching Pursuit, assume $\gamma = 1$:

$$\mathbf{r}_t = \mathbf{r}_{t-1} - |\langle \mathbf{r}_{t-1}, \mathbf{d}_{\lambda_t} \rangle| \, \mathbf{D}_{\lambda_t} \tag{20}$$

For CLEAN:

$$\mathbf{r}_t = \mathbf{r}_{t-1} - |\langle \mathbf{r}_{t-1}, \mathbf{b}[n-\lambda_t] \rangle| \, \mathbf{b}[n-\lambda_t] \tag{21}$$

Because we define $\mathbf{D}_i = \mathbf{b}[n-i]$ this means these two equations are same.
Thus, the residual updating stage is also identical for matching pursuit and CLEAN algorithm.

Therefore, we have proved that matching pursuit and CLEAN algorithm are actually mathematically identical to each other, except for the practical implementation difference, that MP specifically uses a dictionary to store the information of PSF functions, while CLEAN directly uses convolution of PSF with images.

4.2 Comparison between OMP and MP

Basically, the Matching Pursuit and the Orthogonal Matching Pursuit algorithms differs in their ways to update the residual in each iteration. The OMP algorithm introduces the orthogonalization for faster convergence, but with more computation.

In Matching Pursuit, we update the current residual by subtracting the best

correlated atom with its coefficient, so the updated residual is orthogonal to the current best atom $\mathbf{b}_t$.

$$\mathbf{r}_t = \mathbf{r}_{t-1} - |\langle \mathbf{r}_{t-1}, \mathbf{b}_t \rangle| \, \mathbf{b}_t \tag{22}$$

It can be seen that the updated residual is orthogonal to the current most correlated dictionary atom.

$$\langle \mathbf{r}_t, \mathbf{b}_t \rangle = 0 \tag{23}$$

In contrast, the OMP algorithm estimates the clean map using all dictionary atoms selected so far, instead of one single atom, and then calculate the residual in each iteration.

$$\mathbf{r}_t = \mathbf{v} - \mathbf{\Phi}_t \mathbf{x}_t \tag{24}$$

In the above equation, we re-normalize the coefficients of each selected atoms by solving the least-squares problem, and the residual is always orthogonal to the span of all atoms already selected in every iteration, which is the key difference to the MP.

$$\langle \mathbf{r}_t, \text{span}\{\mathbf{\Phi}_t \mathbf{x}_t\} \rangle = 0 \tag{25}$$

Therefore, in OMP, all atoms already selected have no influence on the future residuals. However, in MP, we can never guarantee this, because we only ensure the orthogonality between current residual $\mathbf{r}_t$ and current single atom selected ϕ_{λ_t} in each iteration. Therefore, throughout iterations, subtracting subsequent residuals from the previous one can introduce components that are not orthogonal to the span of all atoms already selected, and the new residuals may be not orthogonal to atoms previous selected.

$$\text{It's possible that } \langle \mathbf{r}_t, \mathbf{b}_{t-\tau} \rangle \neq 0 \text{ for } \tau \in [1, t-1] \tag{26}$$

Because we find sparse coding based on the residual, we want the residual independent of all atoms already selected. Therefore, in MP, with additional non-orthogonal components from the past, we fail to obtain exact future residuals. Then we may find the same atoms already selected again, or get coefficients of atoms not efficient enough to reflect the true residual. Even though the convergence can be guaranteed, because the residual continues to decrease as in (20), and is always non-negative if assume positive $\mathbf{D}$, it takes longer for $\|\mathbf{r}_t\|_2$ to convergence on a stable value, so that the stopping condition $\|\mathbf{r}_t\|_2 < \epsilon$ for solving $\mathbf{x}$ is met.

In OMP, the residuals are more efficient and exclude influences from the past, so the convergence can be faster than MP, although solving the least square problem in each iteration involves matrix inversion, which requires more computations, and this can slow down the program quite a bit in practice. Furthermore, because residuals $\mathbf{r}_t$ are different for MP and OMP in each iteration even for the same initial $\mathbf{y}$, and we rely on $\mathbf{r}_t$ to select $\mathbf{d}$ from $\mathbf{D}$, there is no guarantee that MP and OMP have the same outputs $\mathbf{x}$. In the experiment, their accuracies are comparable.

5 Conclusion and Future Works

In summary, the Hogbom Clean Algorithm, the Matching Pursuit Algorithm, and the Orthogonal Matching Pursuit Algorithm are effective for spatially invariant and spatially varying deconvolution with known PSF. Most blurred bubbles can be recovered at positions almost the same as ground truth. Even for some overlapped bubbles, because their merged shape is larger than a single PSF, after we subtract one PSF, as long as the remaining peak is still high enough, we can distinguish two bubbles. Also, most noise and side lobes can be distinguished from bubbles by these algorithms with appropriate stopping conditions, for example, setting the sparsity equal to the approximated total number of bubbles, or setting the threshold higher than the estimated noise level.

Compared with the Clean Algorithm and the Matching Pursuit, even though their practical implementations and efficiencies are different, we have proved that their mathematical equivalence. Meanwhile, the Orthogonal Matching Pursuit enforces orthogonality between residual and all atoms already selected in every iteration, so it leads to faster convergence than the Matching Pursuit, but solving the least square equation requires more computation costs.

In next stage, we plan to focus on solving blind deconvolution for micro bubble localization. Besides studying and implementing some traditional blind deconvolution methods, we find that some neural network methods, especially some novel methods with generative adversarial networks (GAN) that have been shown empirically to be very robust to solve blind deconvolution problem. In [4], they proposed an idea that trains two generators both to generate latent sharp image and blur kernel to make their convolution result as close as to the blur image. In [5], in order to solve blind deconvolution on micro tubule structure, they use the cycleGAN with one generator for generating high resolution image based on low resolution input image, as well as another blur kernel generator for generating low resolution image based on high resolution input image. Then, they use two different discriminators both applied on shape image domain, and blur image domain to optimize two generators. Both two ideas inspire us, however, the problem they tried to solve is a little different from our research problem: the PSF, or blur kernel in their research problem, is spatially invariant with respect to the whole image. In contrast, micro bubble image has spatially varying PSF functions, that means the process of estimating PSF will depend on local patch within a single image. This problem may not have been solved by someone else so far, and we consider this should be a potential interesting research topic.

6 Reference

[1] Hogbom, J. A. "Aperture synthesis with a non-regular distribution of interferometer baselines." Astronomy and Astrophysics Supplement Series 15 (1974): 417.

[2] Tropp, Joel A., and Anna C. Gilbert. "Signal recovery from random measurements via orthogonal matching pursuit." IEEE Transactions on information theory 53, no. 12 (2007): 4655-4666.

[3] Pati, Yagyensh Chandra, Ramin Rezaiifar, and Perinkulam Sambamurthy Krishnaprasad. "Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition." In Proceedings of 27th Asilomar conference on signals, systems and computers, pp. 40-44. IEEE, 1993.

[4] Asim, Muhammad, Fahad Shamshad, and Ali Ahmed. "Blind image deconvolution using deep generative priors." arXiv preprint arXiv:1802.04073 (2018).

[5] Lim, Sungjun, Sang-Eun Lee, Sunghoe Chang, and Jong Chul Ye. "CycleGAN with a Blur Kernel for Deconvolution Microscopy: Optimal Transport Geometry." arXiv preprint arXiv:1908.09414 (2019).

[6] Mallat, Stéphane G., and Zhifeng Zhang. "Matching pursuits with time-frequency dictionaries." IEEE Transactions on signal processing 41, no. 12 (1993): 3397-3415.

[7] Errico, Claudia, Juliette Pierre, Sophie Pezet, Yann Desailly, Zsolt Lenkei, Olivier Couture, and Mickael Tanter. "Ultrafast ultrasound localization microscopy for deep super-resolution vascular imaging." Nature 527, no. 7579 (2015): 499.

7 Appendix of Code Files

7.1 CLEAN, Spatially Invariant, Python 3

```

# coding: utf-8

# # Image Deconvolution with Hogbom CLEAN Algorithm
# ### Bowen Jiang, Maohao Shen

# In[2]:


import cv2
import numpy as np
import scipy
from scipy import signal
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter
import scipy.io as sio


# ## Ground Truth

# In[31]:


# This is a frame of original clean microbubble data # 78
bubble = sio.loadmat('groundtruth.mat')
bubble = bubble['groundtruth']
bubble = bubble[0:900,0:900]/255
#bubble = cv2.cvtColor(cv2.imread('bubble.png'), cv2.COLOR_BGR2RGB)
#bubble = bubble[0:900, 0:900, 0]
plt.figure(figsize = (10,10))
plt.imshow(bubble, cmap='gray')
plt.title('ground truth - original bubble');


# ## Dirty Beam Generated in Field II

# In[33]:


dirty_beam = sio.loadmat('dirty_beam.mat')
dirty_beam = dirty_beam['data']
dirty_beam = dirty_beam[350:1250,150:1050]

dirty_beam = 255 * (dirty_beam)/(np.max(dirty_beam)-np.min(
    dirty_beam))  # normalize

# dirty beam: a PSF function generated by Field II simulation
flip_dirty_beam = np.flipud(np.fliplr(dirty_beam))  # flip to turn
    corr to fft

f = plt.figure(figsize=(20,20))
f.add_subplot(1,3,1)
```

```
plt.imshow(dirty_beam, cmap='gray')
plt.title('dirty_beam')
# maginify the psf function for visualize
sample2 = dirty_beam[390:510, 390:510].copy()
#calculate the frequency domain of PSF function
dirty_freq = np.fft.fftshift(np.fft.fft2(dirty_beam))
dirty_freq_flip = np.fft.fftshift(np.fft.fft2(flip_dirty_beam))

f.add_subplot(1,3,2)
plt.imshow(sample2, cmap='gray')
plt.title('dirty_beam_magnified')


# In[34]:


print(np.min(dirty_beam),np.max(dirty_beam),np.mean(dirty_beam))


# ## Generate the clean beam (delta function)

# In[41]:


# clean beam: a Gaussian function for approximation of clean
    microbubble
x = np.arange(0, 900, 1)
y = np.arange(0, 900, 1)

gauss_x = np.exp(-(x - 450)**2 / (2 * 2**2))
gauss_y = np.exp(-(y - 450)**2 / (2 * 2**2))
# 2-D Gaussian function
clean_beam = np.outer(gauss_x, gauss_y)

maxi = np.max(clean_beam)
for i in range(clean_beam.shape[0]):
    for j in range(clean_beam.shape[1]):
        clean_beam[i,j] = (clean_beam[i,j]/maxi) * 255

clean_freq = np.fft.fftshift(np.fft.fft2(clean_beam))

f = plt.figure(figsize=(10,10))
f.add_subplot(1,2,1)
plt.imshow(clean_beam, cmap='gray')
plt.title('clean beam');


# ## Generate the dirty map with blurred bubbles and Gaussian
    random noise

# In[53]:


# Concolve the original microbubble data with dirty beam to mimic
    the dirty map image
fft_bubble = np.fft.fftshift(np.fft.fft2(bubble))
```

```
dirty_map = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift(
    fft_bubble * dirty_freq))))
ground_truth = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift
    (fft_bubble * clean_freq))))

#dirty_map = np.abs((dirty_map / np.max(dirty_map)) * 255)  #
    normalize
# add some gaussian random noise
for i in range(dirty_map.shape[0]):
    for j in range(dirty_map.shape[1]):
        noise = np.random.normal(20, 10)
        if noise > 0:
            dirty_map[i, j] += noise
            if dirty_map[i, j] > 255:
                dirty_map[i, j] = 255

f = plt.figure(figsize=(25,25))
f.add_subplot(1,2,1)
plt.imshow(dirty_map, cmap='gray')
plt.title('dirty map')
f.add_subplot(1,2,2)
plt.imshow(ground_truth, cmap='gray')
plt.title('ground truth');


# In[44]:


print(np.max(dirty_map), np.min(dirty_map))


# # Implementation of the Hogbom CLEAN Algorithm

# In[45]:


# Help function to generate PSF function at different spatial
    position
def shift_dirty_beam(dirty_beam, weight, coord_x, coord_y):
    new_dirty_beam = np.zeros((dirty_beam.shape[0]*2, dirty_beam.
    shape[1]*2))

    dirty_beam = (dirty_beam * weight)

    new_dirty_beam[int(0.5*dirty_map.shape[0]):int(1.5*dirty_map.
    shape[0]), int(0.5*dirty_map.shape[1]):int(1.5*dirty_map.shape
    [1])] = dirty_beam
    new_dirty_beam = new_dirty_beam[int(dirty_map.shape[0]-coord_x)
    :int(2*dirty_map.shape[0]-coord_x), int(dirty_map.shape[1]-
    coord_y):int(2*dirty_map.shape[1]-coord_y)]

    return new_dirty_beam


# In[46]:


```

```
# Help function to generate clean beam function at different spatial position
def shift_clean_beam(weight, coord_x, coord_y):
    new_clean_beam = np.zeros((bubble.shape[0]*2,bubble.shape[1]*2))

    x = np.arange(0, 900, 1)
    y = np.arange(0, 900, 1)

    gauss_x = np.exp(-(x - 450)**2 / (2 * 2**2))
    gauss_y = np.exp(-(y - 450)**2 / (2 * 2**2))
    # 2-D Gaussian function
    clean_beam = np.outer(gauss_x, gauss_y)

    maxi = np.max(clean_beam)
    for i in range(clean_beam.shape[0]):
        for j in range(clean_beam.shape[1]):
            clean_beam[i,j] = (clean_beam[i,j]/maxi) * 255

    clean_beam *= weight

    new_clean_beam[int(0.5*bubble.shape[0]):int(1.5*bubble.shape[0]), int(0.5*bubble.shape[1]):int(1.5*bubble.shape[1])] = clean_beam
    new_clean_beam = new_clean_beam[int(bubble.shape[0]-coord_x):int(2*bubble.shape[0]-coord_x), int(bubble.shape[1]-coord_y):int(2*bubble.shape[1]-coord_y)]

    return new_clean_beam


# In[54]:


# parameters and initialization
thresh_hold = 50                # the stop criteria
loop_gain = 1                # the ratio apply on substraction each time
clean_map = np.zeros((900, 900))

# before reruning the following code, must rerun the dirty map again first
f = plt.figure(figsize=(20,20))
f.add_subplot(2,2,1)
plt.imshow(dirty_map, cmap='gray')
plt.title('original dirty map')
plt.colorbar(fraction=0.046, pad=0.04)

count = 0
#while peak >= thresh_hold:
while count < 78:
    #print(count)
    # do cross-correlation to find peak value
    fft_dirtymap = np.fft.fftshift(np.fft.fft2(dirty_map))
    corr = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift(fft_dirtymap * dirty_freq_flip))))

```

```
    coord_x = np.argmax(corr) // corr.shape[1]
    coord_y = np.mod(np.argmax(corr), corr.shape[1])
    peak = dirty_map[coord_x, coord_y]
    #print(coord_x, coord_y, peak, np.max(corr))

    # subtract dirty beam at peak location from dirty map
    #weight = loop_gain * (peak / 255)
    weight = loop_gain
    dirty_map -= shift_dirty_beam(dirty_beam, weight, coord_x,
    coord_y)

    # add clean beam at peak location on clean map
    clean_map += shift_clean_beam(weight, coord_x, coord_y)
    #clean_map[coord_x, coord_y] = weight * 255
    count += 1

error = np.zeros((900,900,3))
error[:,:,0] = ground_truth
error[:,:,2] = clean_map
f.add_subplot(2,2,2)
plt.imshow(error, cmap='gray')
plt.title('error')

f.add_subplot(2,2,3)
plt.imshow(ground_truth, cmap='gray')
plt.title('original map')
plt.colorbar(fraction=0.046, pad=0.04)
f.add_subplot(2,2,4)
plt.imshow(clean_map, cmap='gray')
plt.title('clean map')
plt.colorbar(fraction=0.046, pad=0.04);
```

7.2 OMP, Spatially Invariant, Python 3

```
1
2  # coding: utf-8
3
4  # # Image Deconvolution with Orthogonal Matching Pursuit
5  # ### Bowen Jiang, Maohao Shen
6
7  # In[41]:
8
9
10 import cv2
11 import numpy as np
12 import scipy
13 from scipy import signal
14 get_ipython().run_line_magic('matplotlib', 'inline')
15 import matplotlib.pyplot as plt
16 from scipy.ndimage import gaussian_filter
17 import scipy.io as sio
18
19
20 # ## Ground Truth and Dirty Beam
21
22 # In[118]:
23
24
25 # This is a frame of original clean microbubble data   #55
26 bubble = sio.loadmat('groundtruth.mat')
27 bubble = bubble['groundtruth']
28 bubble = bubble[800:1100,100:400]/255
29 #bubble = cv2.cvtColor(cv2.imread('bubble.png'), cv2.COLOR_BGR2RGB)
30 #bubble = bubble[0:900, 0:900, 0]
31 plt.figure(figsize = (10,10))
32 plt.imshow(bubble, cmap='gray')
33 plt.title('ground truth - original bubble');
34
35
36 # In[121]:
37
38
39 # dirty beam: a PSF function generated by Field II simulation
40 dirty_beam = sio.loadmat('data.mat')
41 dirty_beam = dirty_beam['data']
42 dirty_beam = dirty_beam[650:950,450:750]
43
44 dirty_beam = 255 * (dirty_beam)/(np.max(dirty_beam)-np.min(
       dirty_beam))  # normalize
45
46 flip_dirty_beam = np.flipud(np.fliplr(dirty_beam))  # flip to turn
       corr to fft
47
48 f = plt.figure(figsize=(20,20))
49 f.add_subplot(1,3,1)
50 plt.imshow(dirty_beam, cmap='gray')
51 plt.title('dirty_beam')
52
53 #calculate the frequency domain of PSF function
```

```
dirty_freq = np.fft.fftshift(np.fft.fft2(dirty_beam))


# In[132]:


# clean beam: a Gaussian function for approximation of clean microbubble
x = np.arange(0, 300, 1)
y = np.arange(0, 300, 1)

gauss_x = np.exp(-(x - 150)**2 / (2 * 1.5**2))
gauss_y = np.exp(-(y - 150)**2 / (2 * 1.5**2))
# 2-D Gaussian function
clean_beam = np.outer(gauss_x, gauss_y)

maxi = np.max(clean_beam)
for i in range(clean_beam.shape[0]):
    for j in range(clean_beam.shape[1]):
        clean_beam[i,j] = (clean_beam[i,j]/maxi) * 255

clean_freq = np.fft.fftshift(np.fft.fft2(clean_beam))

f = plt.figure(figsize=(10,10))
f.add_subplot(1,2,1)
plt.imshow(clean_beam, cmap='gray')
plt.title('clean beam');


# ## Dirty Map

# In[140]:


# Concolve the original microbubble data with dirty beam to mimic the dirty map image
fft_bubble = np.fft.fftshift(np.fft.fft2(bubble))
dirty_map = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift(fft_bubble * dirty_freq))))
ground_truth = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift(fft_bubble * clean_freq))))

#dirty_map = np.abs((dirty_map / np.max(dirty_map)) * 255)  # normalize
# add some gaussian random noise
for i in range(dirty_map.shape[0]):
    for j in range(dirty_map.shape[1]):
        noise = np.random.normal(20, 10)
        if noise > 0:
            dirty_map[i, j] += noise
            if dirty_map[i, j] > 255:
                dirty_map[i, j] = 255

f = plt.figure(figsize=(25,25))
f.add_subplot(1,2,1)
plt.imshow(dirty_map, cmap='gray')
plt.title('dirty map')
```

```
f.add_subplot(1,2,2)
plt.imshow(ground_truth, cmap='gray')
plt.title('ground truth');


# ## Shifted Dirty Beam

# In[124]:


# Help function to generate PSF function at different spatial
    position
def shift_dirty_beam(dirty_beam, coord_x, coord_y):
    new_dirty_beam = np.zeros((dirty_beam.shape[0]*2, dirty_beam.
    shape[1]*2))

    new_dirty_beam[int(0.5*dirty_map.shape[0]):int(1.5*dirty_map.
    shape[0]), int(0.5*dirty_map.shape[1]):int(1.5*dirty_map.shape
    [1])] = dirty_beam
    new_dirty_beam = new_dirty_beam[int(dirty_map.shape[0]-coord_x)
    :int(2*dirty_map.shape[0]-coord_x), int(dirty_map.shape[1]-
    coord_y):int(2*dirty_map.shape[1]-coord_y)]

    new_dirty_beam = new_dirty_beam

    return new_dirty_beam


# ## Shifted Clean Beam (delta function)

# In[133]:


# Help function to generate clean beam function at different
    spatial position
def shift_clean_beam(weight, coord_x, coord_y):
    new_clean_beam = np.zeros((bubble.shape[0]*2,bubble.shape[1]*2)
    )

    #x = 0*np.arange(0, 900, 1)
    #y = 0*np.arange(0, 900, 1)

    x = np.arange(0, 300, 1)
    y = np.arange(0, 300, 1)

    gauss_x = np.exp(-(x - 150)**2 / (2 * 1.5**2))
    gauss_y = np.exp(-(y - 150)**2 / (2 * 1.5**2))
    # 2-D Gaussian function
    clean_beam = np.outer(gauss_x, gauss_y)

    maxi = np.max(clean_beam)
    for i in range(clean_beam.shape[0]):
        for j in range(clean_beam.shape[1]):
            clean_beam[i,j] = (clean_beam[i,j]/maxi) * 255

    #clean_beam = np.outer(x, y).astype('float64')
    #clean_beam[448:452,448:452] = 255
```

```
155
156     clean_beam *= weight
157
158     new_clean_beam[int(0.5*bubble.shape[0]):int(1.5*bubble.shape
        [0]), int(0.5*bubble.shape[1]):int(1.5*bubble.shape[1])] =
        clean_beam
159     new_clean_beam = new_clean_beam[int(bubble.shape[0]-coord_x):
        int(2*bubble.shape[0]-coord_x), int(bubble.shape[1]-coord_y):
        int(2*bubble.shape[1]-coord_y)]
160
161     return new_clean_beam
162
163
164 # ## Implementation of Orthogonal Matching Pursuit
165
166 # In[58]:
167
168
169 """ # old version using the dictionary
170 def OMP_dic(D, v, m, epsilon):
171     #
172     #inputs:
173     #D is the dictionary
174     #v is the dirty map
175     #m is the maximum sparsity of output x
176     #epsilon is the stopping condition
177
178     #outputs:
179     #x is the reconstructed clean map
180     #
181
182     x = np.zeros(D.shape[1])
183     r = v.copy()    # initialize residual same as dirty map
184     indices = []    # collection of selected indices
185
186     for i in range(m):
187         print(i)
188
189         if (np.linalg.norm(r) / np.linalg.norm(v)) < epsilon:
190             break
191
192         projection = np.dot(D.T, r)
193
194         index = np.argmax(abs(projection))
195
196         if index in indices:
197             break
198         indices.append(index)
199
200         if len(indices) == 1:
201             phis = D[:,index]
202             x_i = projection[index] / np.dot(phis.T, phis)
203
204         else:
205             phis = np.vstack([phis, D[:,index]])
206             x_i = scipy.linalg.solve(np.dot(phis, phis.T), np.dot(
    phis, v), assume_a='sym')
```

```
207
208         a = np.dot(phis.T, x_i)
209         r = v - a
210
211     for i, index in enumerate(indices):
212         x[index] += x_i[i]
213
214     return x
215
216
217 #sparse = OMP_dic(dictionary, dirty_map.ravel(), 3, 1E-4)
218 """
219
220
221 # In[141]:
222
223
224 def OMP(dirty_beam, v, m, epsilon): # new version without the
        dictionary
225     """
226     inputs:
227     dirty_beam
228     v is the dirty map
229     m is the maximum sparsity of output x
230     epsilon is the stopping condition
231
232     outputs:
233     x is the reconstructed clean map
234     """
235
236     length = v.shape[1]
237     width = v.shape[0]
238     v = v.ravel()
239
240     x = np.zeros(length*width)
241     r = v.copy()     # initialize residual same as dirty map
242     indices = []     # collection of selected indices
243
244     for i in range(m):
245         print(i)
246
247         #if (np.linalg.norm(r) / np.linalg.norm(v)) < epsilon:
248         #    break
249
250         projection = np.zeros((width,length))
251         for j in range(width):
252             for k in range(length):
253                 shifted_b = shift_dirty_beam(dirty_beam, j, k).
        ravel()
254                 if np.max(shifted_b + r) < 280:
255                     projection[j,k] = 0  # increase speed
256                 else:
257                     projection[j,k] = np.dot(shifted_b, r)
258         print('proj done')
259
260         index = np.argmax(projection)
261
```

```
        if index in indices:
            break
        indices.append(index)

        if len(indices) == 1:
            shifted_b = shift_dirty_beam(dirty_beam, index//length, index%length).ravel()
            phis = shifted_b
            x_i = projection[index//length, index%length] / np.dot(phis.T, phis)

        else:
            shifted_b = shift_dirty_beam(dirty_beam, index//length, index%length).ravel()
            phis = np.vstack([phis, shifted_b])

            x_i = np.linalg.pinv(phis).T @ v
            print('pinv done')

        a = np.dot(phis.T, x_i)
        r = v - a

    for i, index in enumerate(indices):
        x[index] += x_i[i]

    return x


# In[142]:


sparse = OMP(dirty_beam, dirty_map, 57, 1E-4)  #57


# In[143]:


sparse_copy = sparse.copy()
clean_map = np.zeros((dirty_map.shape[0], dirty_map.shape[1]))
for i in range(len(sparse_copy)):
    if sparse_copy[i] != 0:
        clean_map[i//dirty_map.shape[1], i%dirty_map.shape[1]] += sparse_copy[i] * 255

fft_clean = np.fft.fftshift(np.fft.fft2(clean_map))
clean_map = np.fft.fftshift(np.real(np.fft.ifft2(np.fft.fftshift(fft_clean * clean_freq))))

# only for old version
#clean_map = dictionary_clean @ sparse_copy
#clean_map = clean_map.reshape(300,300)

f = plt.figure(figsize=(15,15))
f.add_subplot(2,2,1)
plt.imshow(dirty_map, cmap='gray')
plt.title('dirty_map')
plt.colorbar(fraction=0.046, pad=0.04)
```

```
314
315 error = np.zeros((300,300,3))
316 error[:,:,0] = ground_truth
317 error[:,:,2] = clean_map
318 f.add_subplot(2,2,2)
319 plt.imshow(error, cmap='gray')
320 plt.title('error')
321
322 f.add_subplot(2,2,3)
323 plt.imshow(ground_truth, cmap='gray')
324 plt.title('ground truth')
325 plt.colorbar(fraction=0.046, pad=0.04)
326
327 f.add_subplot(2,2,4)
328 plt.imshow(clean_map, cmap='gray')
329 plt.title('clean map');
330 plt.colorbar(fraction=0.046, pad=0.04)
```

7.3 CLEAN, Spatially Varying, Matlab 2019b

```
% CLEAN Algorithm, Spatially Varying
% Bowen Jiang, Maohao Shen

dirty_map = double(imread('dirty_map.jpg'));
dirty_map = dirty_map(:,:,1);
residual = dirty_map;
clean_map = zeros(1600,1200);

dirty_beam = double(imread('dirty_beam.jpg'));
dirty_beam = dirty_beam(:,:,1);
dirty_beam_flip = flip(dirty_beam);
dirty_beam_flip = flip(dirty_beam_flip,2);

loop_gain = 1.2;

num_ite = 100;
for ite = 1:num_ite
    find_peak = conv2(residual, dirty_beam_flip, 'same');
    [peak, x_coords] = max(find_peak);
    [peak, y_coord] = max(max(find_peak));
    x_coord = x_coords(y_coord);

    peak_coord = [x_coord, y_coord];
    peak = dirty_map(x_coord, y_coord);

    psf = PSF(x_coord, y_coord);

    residual = residual - loop_gain*(peak/255)*psf;
    clean_map(x_coord, y_coord) = 1;
end
```

7.4 Generate PSF by Field II, Matlab 2019b

```
% Field II Simulation Program

function [scale_data] = PSF (x_coord, y_coord)

%clear all;close all;clc;
%path(path, 'C:\Research\Field_II_ver_3_24_windows_gcc')
%field_init
% set_field ('att',1.5*100);
% set_field ('Freq_att',0.5*100/1e6);
% set_field ('att_f0',5e6);
% set_field ('use_att',1);
%% Random microbubles & Field II simulation
f0 = 5e6; % Transducer center frequency [Hz]
fs = 100e6; % Sampling frequency [Hz]
c = 1540; % Speed of sound [m/s]
lambda = c/f0; % Wave length [m]
width = 0.27e-3; % Width of element
element_height = 5/1000; % Height of element [m]
kerf = 0.03/1000; % Kerf [m]
focus=[0 0 20]/1000; % Fixed focal point [m]
N_elements = 128; % Number of elements in the transducer
N_active = N_elements; % Active elements in the transducer
% Set the sampling frequency
set_sampling(fs);
% Generate aperture for emission
emit_aperture = xdc_linear_array(N_active, width, element_height,
    kerf, 10, 10, focus);
% Set the impulse response and excitation of the emit aperture
impulse_response = sin(2*pi*f0*(0:1/fs:2/f0));
impulse_response = impulse_response.*hann(max(size(impulse_response
    )))';
xdc_impulse(emit_aperture, impulse_response);
excitation = sin(2*pi*f0*(0:1/fs:1/f0));
excitation = zeros(length(excitation),1)';
excitation(9) = 1;
xdc_excitation(emit_aperture, excitation);
% Generate aperture for reception
% receive_aperture = xdc_convex_array (N_elements, width,
    element_height, kerf, Rconvex, 5, 5, focus);
receive_aperture = xdc_linear_array(N_active, width, element_height
    , kerf, 10, 10, focus);
% Set the impulse response for the receive aperture
xdc_impulse(receive_aperture, impulse_response);

chano = N_elements;
resol = 1*lambda;  %axial resolution
dx = resol/10; % lateral resolution
dz = resol/10;
startdepth = 0; % m
enddepth = 50e-3; %m


x = -(chano-1)/2*(width+kerf):dx:(chano-1)/2*(width+kerf); %
    lateral dimension of the FOV
z = startdepth:dz:enddepth; % axial dimension of the FOV
```

```
51
52 Nz = length(z);
53 Nx = length(x);
54 data = zeros(Nz,Nx);
55 groundtruth = zeros(Nz,Nx);
56
57 %random generatre n number of bubbles
58 numberofbubbles = 1;
59 positions = zeros(numberofbubbles,2);
60 %positions(:,1) = rand(1, numberofbubbles) * 2*(chano-1)/2*(width+
       kerf) -(chano-1)/2*(width+kerf);
61 %positions(:,1) = [((200-600)/600)* (chano-1)/2*(width+kerf)];
62 %positions(:,2) = rand(1, numberofbubbles) * enddepth;
63 %positions(:,2) = [((400)/1600)*50e-3];
64
65
66 positions(:,2) = [(x_coord-0.5)/Nz * enddepth];
67 positions(:,1) = [((y_coord-0.5)/Nx * abs(x(end)-x(1))) - abs(x(end
       )-x(1))/2];
68 %groundtruth(ceil(positions(1,2)/enddepth * Nz),ceil(  (positions
       (1,1) + abs(x(end)-x(1))/2)   / abs(x(end)-x(1))  *  Nx))= 1;
69 groundtruth(x_coord,y_coord)= 1;
70
71
72 bublepositions = zeros(numberofbubbles,3);
73 bublepositions(:,1) = positions(:,1);
74 bublepositions(:,3) = positions(:,2);
75 phantom_positions = bublepositions;
76 amplitude_scaler = abs(rand(numberofbubbles,1) * 1000);
77 phantom_amplitudes = amplitude_scaler.*ones(numberofbubbles,1)*1e25
       ;
78
79
80 arrayx = (-chano/2+.5:chano/2-.5)*(width+kerf);
81 arrayz = zeros(1,length(arrayx));
82 senscutoff = 0.0;
83 Origin = [0,0];
84 Nt = 1;
85 [X,Z] = meshgrid(x,z);
86 [Na,Nl] = size(X);
87
88 X = repmat(X,[1,1,chano]);
89 Z = repmat(Z,[1,1,chano]);
90 arrayX = repmat(reshape(arrayx,[1,1,chano]),[Na,Nl,1]);% Meshgrid
       the array vector
91 arrayZ = repmat(reshape(arrayz,[1,1,chano]),[Na,Nl,1]);% Meshgrid
       the array vector
92 twpeak = 0.50e-6; %time offset for true ultrasound peak arrival
       time.
93
94 scat =[];
95 [scat,start_time] = calc_scat_all(emit_aperture,receive_aperture,
       phantom_positions,phantom_amplitudes,1);
96 scat = [zeros(round(start_time*fs),N_active*N_active);scat];
97 time_samples = size(scat);
98 scat = reshape(scat,[time_samples(1),N_elements,N_elements]);
99 scat = sum(scat,2);
```

```
scat = squeeze(scat);

%% DAS beamforming
if sum(sum(sum(isnan(scat))))~= 0
    disp('NAN. Why?');
else
    disp('No problem.');
    Steer = [0 ] /180 *pi;
    NoAngles = length(Steer);
    N_transmit = N_elements;
    N_receive = N_elements;
    RData = scat;
    chano = N_receive;
    Fs = fs; %sampling frequency
    ft = f0; %transmit frequency
    cycle = 2; % number of transmit cycles
    plength = 1/ft*cycle; % transmit pulse length (sec)
    plengths = ceil(plength*Fs); % transmit pulse length in unit of
     samples
    % twpeak = 3.5781/fc;
    % prf = 500;

    PageEnd = size(RData,1);
    % Reshape the raw channel data to 3D (x,z,t)
    ReShpRData = zeros(PageEnd,chano,NoAngles,Nt);
    Tx_focus = 0e-3; %m
    if Tx_focus > 0 % not plane waves
        FocalPt(1) = Origin(1) + Tx_focus * sin(Steer);
        FocalPt(2) = 0.0;
        FocalPt(3) = Tx_focus * cos(Steer);
        % Compute distance to focal point from each active element.
        X_array = arrayx - FocalPt(1);
        Tx_Delay = sqrt(X_array.*X_array + FocalPt(3)*FocalPt(3))/c
    ;
        Tx_Delay = max(Tx_Delay(:)) - Tx_Delay;
    elseif Tx_focus == 0   %plane wave imaging
        Tx_Delay = (arrayx'- Origin(1)) * sin(Steer)/c;
        Tx_Delay = Tx_Delay - min(Tx_Delay(:));
    else   %diverging waves
        FocalPt(1) = Origin(1) + Tx_focus * sin(Steer);
        FocalPt(2) = 0;
        FocalPt(3) = Tx_focus * cos(Steer);
        % Compute distance to focal point from each active element.
        X_array = arrayx' - FocalPt(1);
        Tx_Delay = sqrt(X_array.*X_array + FocalPt(3)*FocalPt(3))/c
    ;
        Tx_Delay = Tx_Delay - min(Tx_Delay(:));
    end
    for i = 1:Nt
        for j = 1:NoAngles
            dummy = double(squeeze(RData((i-1)*NoAngles*PageEnd+(j
    -1)*PageEnd+1:(i-1)*NoAngles*PageEnd+j*PageEnd,:)));
            ReShpRData(:,:,j,i) = reshape(dummy,PageEnd,N_receive
    ,1,1);
        end
    end
    %ReShpRData = scat for single plane wave imaging
```

```
152
153     % Pixel-oriented beamforming (matrix version)
154
155     clear delaysub delayidx;
156     for n = 1:NoAngles
157         Tdelay = interp1(arrayx,Tx_Delay,x,'linear');
158         Tdelay = repmat(reshape(Tdelay,[1,Nl,1]),[Na,1,chano]); %
    Transmit delay
159 %         delay = round(((Z+sqrt((Z-arrayZ).^2+(X-arrayX).^2))/c+
    Tdelay+lensCorrection/c*2+twpeak)*Fs);% calculate the delay in
    each channel for each pixel in units of samples
160         delay = round(((Z+sqrt((Z-arrayZ).^2+(X-arrayX).^2))/c+
    Tdelay+twpeak)*Fs);
161         delay(delay<=0) = 1; % clear 0 delays
162         delay(delay>PageEnd) = PageEnd; % clear delays beyond the
    sampling range
163         %delaysub(:,:,:,n) = delay;
164         %delayidx(:,:,:,n) = sub2ind([size(ReShpRData,1),chano],
    delay,repmat(reshape(1:chano,[1,1,chano]),[Na,Nl,1]));%find the
     indices of the data for each pixel in each channel
165         delay = sub2ind([size(ReShpRData,1),chano],delay,repmat(
    reshape(1:chano,[1,1,chano]),[Na,Nl,1]));%find the indices of
    the data for each pixel in each channel
166     end
167
168     X =[];
169     Z =[];
170     arrayZ =[];
171     arrayX =[];
172     Tdelay =[];
173     % Beamforming and IQ downmixing
174     beamformedIQ = zeros(Na,Nl,NoAngles,Nt);
175
176     for i = 1:Nt
177         for n = 1:NoAngles
178             dummy = squeeze(ReShpRData(:,:,n,i));
179             %size(dummy) is 6547*128
180             beamformedIQ = sum(dummy(delay),3);
181             %TEMP = sum(dummy(delayidx(:,:,:,n)).*ElementSens,3);
182             %size(TEMP) is 1624*495, size(delayidx is 1624*495*128)
183             %beamformedIQ(:,:,n,i) = TEMP;
184             %env(:,:,n,i) = abs(hilbert(TEMP));
185         end
186     end
187     %env = beamformedIQ;
188     %data(:,:) = beamformedIQ;
189     %figure;imagesc(x*1e3,z*1e3,20*log10(env./max(env(:))),[-60 0])
    ;colormap(gray);
190     %figure;imagesc(x*1e3,z*1e3,beamformedIQ);colormap(gray);
191     beamformedIQ = abs( hilbert(beamformedIQ));
192     %figure;imagesc(x*1e3,z*1e3,beamformedIQ);colormap(gray);
193     data(:,:)=beamformedIQ;
194 end
195
196 %save('PSFrf.mat','data');
197 %%
198 %colormap('gray')
```

```
%data = data(201:600, 1:400);
%imagesc(data)

%%
%data = data(2:401, :);
scale_data = ((data-min(min(data))) ./ (max(max(data))-min(min(data
    )))) * 255;
%scale_data = ((data-min(min(data))) ./ (max(max(data))-min(min(
    data))));
%imwrite(scale_data, 'dirty_map.jpg');
```