# Final Project: Progress Report 1

CSE 597

**Amatur Rahman**

Friday, December 7, 2018

## Abstract

The idea of relating Poisson equation into image processing was motivated from [10]. Image blending is one of the important techniques in image processing. In this project, we want to seamlessly cut and paste a source image on a target image, where the source and target are visually different [11]. This problem falls in the category of gradient domain processing of image, where a gradient vector corresponds to the change of pixel value. The process is computationally expensive, more so in case of gigabyte sized images. So sequential algorithms do not scale well to solve these problems.

In this report, a method based on solving the Poisson equation using Jacobi iterative method and direct solver is proposed. Our aim is to compare iterative method with direct method and analyze their effectiveness in solving the image blending problem. The evaluation criteria are based on the number of iterations and computational time to solve the Poisson equation. It is shown that Jacobi iterative method is better than LU Decomposition in terms of both memory and computational time for larger problem size.

## 1 Problem of Interest

To solve this problem, first we formulate the seamless image blending problem using a 2d Poisson equation. Then we discretize it using finite difference to get Ax=B problem formulation. Then we apply *Jacobi* and *LU Decomposition* solver to solve the Ax=B problem. Among these two, Jacobi for sparse matrices performs better in terms of time and space complexity.

- ~~What the problem is:~~ **What the problem is:**

  The problem is to find out the blended image using Poisson solver.

- ~~General overview:~~ **General overview:**

  Poisson image blending is a process of inserting a source image into a target image [7].

- ~~Scientific merit (Why is this worth doing?):~~ **Scientific merit (Why is this worth doing?):**

  Poisson image processing and image blending has wide scale application in motion and object detection. Whenever a pattern matching or machine learning algorithm is used, a large number of data is required to train the machine learning algorithm. Large scale image processing is also required in X-ray image stitching [9]. When the image size gets larger, say around 1000000 pixels, it gets computationally unfeasible to solve using sequential methods. So a numerical parallel approach is worthy of doing.

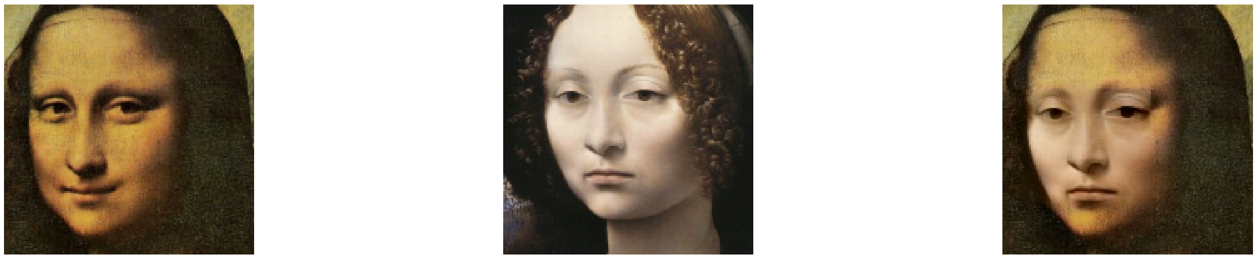

Figure 1: Overview of image blending problem [2]

- ~~Fields where relevant:~~ **Fields where relevant:**

  Digital Image Processing, Data augmentation in deep neural networks, Medical Imaging

- ~~What other methods are~~ **What other methods are used to solve this type of problem:**

  Typically Fast Fourier Transform is used to solve ~~this type of problem: Discussion of known solution (analytic/published) :~~ image blinding [6]. While there are many sequential algorithms implemented in practice, parallel implementation studies is quite limited. In [12], a simple framework for the parallel gradient domain processing is given for gigapixel images.

- **Discussion of known solution(analytic/published)** Conjugate Gradient (CG) algorithms typically exhibit much faster convergence, so it is almost always used with a preconditioner, and preconditioned CG (PCG) usually requires many fewer iterations than Jacobi to reach the same accuracy. But the preconditioning is hard to do.

  Typically Fast Fourier Transform is used to solve image blinding [6]. While there are many sequential algorithms implemented in practice, parallel implementation studies is quite limited. In [12], a simple framework for the parallel gradient domain processing is given for gigapixel images. Methods exist to find a direct Poisson solution using Fast Fourier Transforms (FFT) [8]. FFT methods can make use of parallel computation. However, these methods have not yet been tested in modern out-of-core gradient domain image processing.

  Direct techniques have a much higher cost. For example, Cholesky decomposition and Gaussian elimination has runtime complexity $O(n^3)$, where n is the number of pixels. When n becomes large, such direct solvers require too much memory and computational becomes impractical.

  Often the Poisson problem is simplified by discretization into a large linear system whose dimension is typically the number of pixels in an image. Methods exist to find a direct solution to this linear system.

  Often, especially in distributed systems, it is simpler to implement an iterative method to find a solution [13, 5]. However for larger linear systems, memory consumption is a limiting factor and iterative methods such as Jacobi is preferred.

## 1.1 Numerical Set-up

Now this yields a system of linear equations The boundary conditions are Dirichlet.

- ~~How the A-matrix and b-vector are formed~~ **How the A-matrix and b-vector are formed**

  We have Source image S, and a target image T, we want to seamlessly paste the source image to target so that it matches its surroundings and does not stand out. Say f(x, y) is the pixel value at (x, y) coordinate of the pasted image.

  Now we want to optimize

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ with } f|_{\partial\Omega} = f^*|\partial\Omega$$

  Where $v$ is the gradient of a region in an image, $g$ is a selected region of source, which for simplification we assumed the whole picture, $f*$ is a set of known functions that exist in domain S (source image), f is an unknown function that exist in domain $\Omega$, $\Omega$ is the region g after placing it on source image. $\partial\Omega$ are the boundaries between the source and target regions.

  So given a vector field, we want to find the value of f in unknown region that optimize.

  Now, the solution of the above equation is the solution of Poisson equation:

$$\Delta f = \text{ div v over } \Omega, \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{1}$$

  Here,

$$\operatorname{div} \mathbf{v} = \partial v/\partial x + \partial v/\partial y \tag{2}$$

  $\Delta$ is the Laplacian operator:

  So we get the formulation of divergence using

$$divG = -4f(X, y) + f(X-1, y) + f(X, y-1) + f(X+1, y) + f(X, y+1) \tag{3}$$

Now we get the b matrix by,

$$b[i] = div(G(Source(x, y))) + Neighbor(target[i])$$

for all i = 1 to N

where N is the problem size. If we deal with 100x100 pixel image N = 100.

The equations were adopted from [3].

A is a matrix of size NxN, which is obtained by discretizing Poisson 2D equation. It is a sparse matrix having -4 in diagonals, and 1 in off-diagonals.

To get the test problem of B vector from image, In order to generate the test problem, we cut and pasted a source image in grayscale from to the target image, and then applied the blending process using the equation. We tested our solver with images of 100x100 pixels. All the other scalability tests were done on a randomly generated B vector.

- ~~Problem sizes for problems of interest:~~ **Problem sizes for problems of interest:**

  We want to be able to solve the seamless image blending for gigapixel images. So there would $10^6 x 10^6$ pixels in production problem.

- ~~Discretization method and Boundary conditions:~~ **Discretization method and Boundary conditions:**

  Poisson's equation is discretized by mapping it into a finite differenced spatial grid. From that, we obtain the results in the Equation 3. We use Dirichlet condition, where all boundary values are zero.

## 2 Solvers

### 2.1 Direct Solver

- ~~Direct solver being used:~~ **Direct solver being used:**

  LU Decomposition.

- ~~Justification for direct solver being used:~~ **Justification for direct solver being used:**

  Since most of the element of the A matrix is zero, doing the backfill in LU decompostion is computationally inexpensive than other direct solvers like Cramer's method.

  In LU factorization, we can separate the factorization and solving for x in two steps. Since the image processing application we are working on keeps A unchanged and is supposed to apply repetitively to different datasets, this decoupling saves time. Because factorization itself is the most expensive computation (O($N^3$)). This only requires matrix A. Once this is done, this factored A can be used with different B matrices to compute at a faster rate, because the cost of solving for x is only O($N^2$) which is way less than the cost of decompisition as seen by our analysis to follow. For larger problem size, this effect is even more pronounced. [1]

- ~~List of and justification for optimization flags~~ **List of and justification for optimization flags**

  We are using "O2" flags because they needed 0.007 seconds on average, whereas other take 0.015 seconds. It massively improves LU decomposition implementation too.

- ~~Timing for test problem~~ **Timing for test problem**

  - For a problem size of 1000 pixels:
    Total CPU ticks for forward substitution: 1561
    Total CPU ticks for backward substitution: 1625
    Total CPU ticks for LU decomposition: 1287851

- ~~Memory being allocated for test problem :~~ **Projection of time for production problem:** In production, we will be using gigapixels of image. Considering N = 1000000 (that results in N*N = 1gigapixel), we can obtain time for production problem by multiplying the results of problem size of 1000 pixels by 1000. (2/3ncube)

  Eliminating the first column will require n additions and n multiplications for n1 rows. Therefore, the number of operations for the first column is 2n(n1). For the second column, we have n1 additions and n1 multiplications, and we do this for (n2) rows giving us 2(n1)(n2). Therefore, the total number of operations required for the full decomposition can be written as O($\frac{2}{3}n^3$). For N = 1000, time 0.270089 seconds. Let $\frac{2}{3} \times n^3 = 0.270089$. From this we calculate $n$. Now, if this $n$ is multiplied by 1000, we get the projected time for gigapixels image, which is approx. 7716 days! This is not feasible computationally.

- **Memory being allocated for test problem:**

  Maximum resident set size (kbytes): 63896 (memory allocated in RAM)

  Number of page faults: 16016

- ~~Comparison with memory being used by task~~ **Comparison with memory being used by task** Maximum resident set size when no method is running is 32492 kbytes, whereas LU decomposition takes extra 63896 Kbytes in memory.

- ~~Projection of memory required for production problem:~~ **Projection of memory required for production problem:**

  In production, we will be needing gigapixels of image. Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000000, memory for A is 1000000*1000000*2 bytes, for B, x and intermediate matrix is 1000000*8000000*3 bytes. So that sums up to 22400 MBytes.

## 2.2 Iterative Solver

In iterative method error is decreased in each iteration. We ran the solver for maximum 2*$N^2$ iteration, or when the error is significantly small (error tolerance 1e-.04).

- ~~Iterative solver being used:~~ **Iterative solver being used:** Jacobi

- ~~Justification for iterative solver being used~~ **Justification for iterative solver being used** The linear system in discussion has a strictly diagonally dominant A matrix. Jacobi method converges when we have a strictly diagonally dominant matrix. In this case, since we are discretizing 2D Poisson's equation, we get a diagonally dominant A matrix. That's why Jacobi method was chosen. The matrix is also positive definite. There are other methods such as Gauss-Siedel, Successive Over-relaxation which are also applicable in this context. However, our final goal in this project is efficient parallelization, and it is achievable quickly and easily using Jacobi. This is because Jacobi algorithm does calculations of rows separately. This can be exploited in both multicore-programming and accelerators. Gauss-Siedel performs poorly in such case because of the row dependencies.

- ~~Justification for convergence criteria, and residual/norm being used:~~ **Justification for convergence criteria, and residual/norm being used:**

  From the analysis of 2D Poisson equation in [13], it can be proved that Jacobi method converges in $O(N^2)$ steps. So we chose maximum iteration number to be in that order.

- ~~List of and justification for optimization flags:~~ **List of and justification for optimization flags:**

  We are using "O2" flags because it needed 0.007 seconds on average, whereas other take 0.015 seconds. It massively improves my Jacobi implementation too.

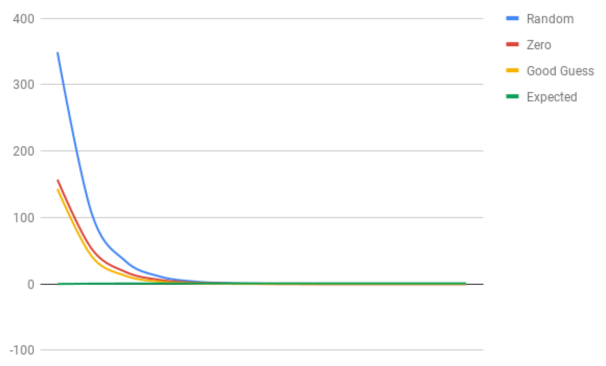

Figure 3: Convergence compared to the expected rate. In the figure x-axis is number of iteration and y-axis is the maximum error in each iteration.

- ~~Timing for test problem for 3 different initialization methods (random, good guess, all zeros or ones):~~ **Timing for test problem for 3 different initialization methods (random, good guess, all zeros or ones):**

  Random: 51.902 ms
  All Zeroes: 54.456 ms
  Good Guess: 46.046 ms

  Here "good guess" is the solution 'x' vector obtained from an earlier solution. If we deal with images with similar patterns, then from the earlier solution database, we can make a good guess.

- ~~Plot showing convergence, comparison of convergence rate with expected rate~~ **Plot showing convergence, comparison of convergence rate with expected rate**

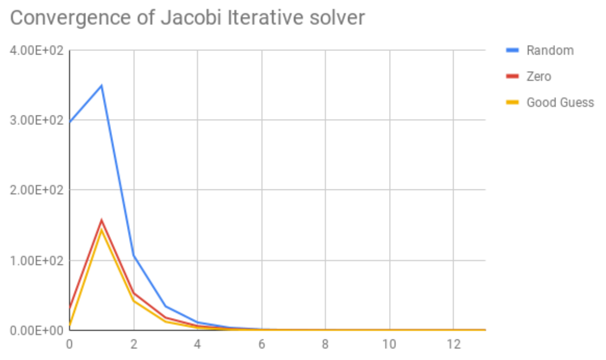

Figure 2: Convergence rate of Jacobi for Different Initialization Techniques

In the figure x-axis is number of iteration and y-axis is the maximum error in each iteration.

The expected rate is cos(pi/(n+1)) [4]. Compared to the expected rate, the convergence of jacobi iteration is shown as follows [Fig. 3].

- ~~Projection of time required for production problem:~~ **Projection of time required for production problem:** Jacobi has running time $O(N^4)$. Therefore we would need 106.8890046 days if an image blending process of gigapixels of images is done in a sequential approach. We calculated this by scaling from the smallest runtime (which is 310 ms for N = 100) to larger runtimes.

- ~~Memory being allocated for test problem:~~ **Memory being allocated for test problem:**

  Maximum resident set size (kbytes): 48176

  Minor (reclaiming a frame) page faults: 12087

- ~~Comparison with memory being used by task:~~ **Comparison with memory being used by task:**

  For a problem size of N = 100, Jacobi method uses 48228 kbytes, whereas for initialization only we require 32492 kbytes. So in total we need 63896 kbytes of memory. So 75% of memory is contributed to Jacobi computation.

  Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000, memory for A is 1000*1000*2 bytes, for B, x and intermediate matrix is 1000*8000*3 bytes. So that sums up to 22400 kbytes. The calculated estimated memory from integer and floating point size is 22400 kbytes, which is comparable to 48228 kbytes obtained using the command `/usr/bin/time -v ./main.out`.

- ~~Projection of memory required for production problem:~~ **Projection of memory required for production problem:** Considering integer size to be 2 bytes and floating point size to be 8 bytes. For N = 1000000, memory for A is 1000000*1000000*2 bytes, for B, x and intermediate matrix is 1000000*8000000*3 bytes. So that sums up to 22400 MBytes. We did not use the sparse representation in our implementation in progress report 1, but doing so saves 2/3rd of space contributed by A.

## 3 Solver Comparison

- ~~Which solver does better?~~ **Which solver does better?** In my implementation Jacobi iterative solver does better both in compute time and memory.

- ~~Which solver will do better for a production scale problem?~~ **Which solver will do better for a production scale problem?**

  Since our A matrix is sparse it is better to use Jacobi method. Sparse matrices do not generally have sparse LU decomposition, so it gets harder to fit the matrices in memory as the problem size goes larger. For Jacobi method, we can just store the A matrix in sparse representation.

- ~~Discuss how the production problem projections will be constrained and what will need to be taken into account for parallelization:~~ **Discuss how the production problem projections will be constrained and what will need to be taken into account for parallelization:**

  The Jacobi iterative solver can be implemented in parallel in the following way. Let $p$ be the number of processors, $f$ be the time for a floating point operation, $\alpha$ be the startup time, and $\beta$ be the time per word in a message. So the time required is going to be Time = number of steps * cost per step = $O(N) * ((N/p) * f + \alpha + (n/p) * \beta) = O(N^2/p) * f + O(N) * \alpha + O(N^{3/2}/p) * \beta$.
  So, O(N/p) floating point operations are needed to do startup and O(n/p) boundary values are passed to other nodes [4].

## 4 Discussion and Conclusions

1. ~~Basic overview of what we have done:~~ **Basic overview of what we have done:** We implemented two solver for the Ax=B problem: Jacobi and LU Decomposition. For accurate solution, direct method is more appropriate. It does not need any estimation, and it is insensitive to all of the randomness from the model. Comparing LU decomposition with Jacobi iterative method we realized that LU decomposition is more resource hungry and takes more clock ticks than Jacobi iteration.

2. ~~Justification for solver choices:~~ **Justification for solver choices:** Since A matrix is sparse and positive definite, Jacobi iterative method is guaranteed to converge. LU decomposition works well as a straightforward direct method. Its computational requirements are high, so for higher image size, it becomes infeasible to solve with LU decomposition.

3. ~~Discussion of constraints of production problem with requirements for parallelization:~~ **Discussion of constraints of production problem with requirements for parallelization:** The problem has some sequential parts, which cannot be parallelized. In order to parallelize the problem, some preprocessing is required, which would be sequential.

# Appendices

## A Acknowledgements

I took help from the template code provided by Professor Lavely and Professor Blanton for Jacobi iterative method implementation, and the script to cycle through optimization flags. All of the computations are performed on my ACI account, which sponsored by PSU CSE department. I would also like to thank my classmates for helping me with ACI commands.

## B Code

- ~~Where to find the code:~~ **Where to find the code:**
  `https://github.com/amatur/cse597_solvers`
- ~~File names and descriptions: Can be found at~~ **File names and descriptions:**
  - `cycleThrough.sh`: The shell script to iterate through all of the optimization flags to find out the best one.
  - `gpl.txt, license.txt`: Licensing information
  - ~~Instructions for compiling and reproducing your results: Can~~ `main.cpp`: source file containing Jacobi and LU decomposition solver
  - `Makefile`: makefile for generating executable main.out |:
- **Instructions for compiling and reproducing your results:**
  Instructions can also be found at `https://github.com/amatur/cse597_solvers`
- ~~Listing of which nodes types we ran on: I compiled and ran~~Log in to ACI, using ssh: `ssh USERID@aci-b.aci.ics.psu.e`
  After logging into ACI, you need to request a compute node with desired number of processors and nodes. For example, to run the code on ~~standard open compute nodein ACI using the following command: `qsub -I -A open -l walltime=2:00:00 -l nodes=1:ppn=1.`~~maximum 1 node:

  ```
  qsub -I -A open -l walltime=2:00:00 -l nodes=1:ppn=1
  ```

  Now, after you are in the compute node which has a name like `comp-sc-0161`, load the following modules by typing:

  ```
  \DIFadd{module load gcc/7.3.1
  ```

  Download the code:

  ```
  git clone git@github.com:amatur/cse597_solvers.git
  ```

  Change directory into `cse597_solvers` and Make the code after cleaning up:

  ```
  \DIFadd{make clean
  make
  ```

  Run the code:

  ```
  \DIFadd{./main.out 4000
  ```

  You should see output with runtime for both direct and iterative solvers.

```
\DIFadd{Total time(seconds) taken by CPU for LU Decomposition Only: 18.7073
Total time(seconds) taken by CPU for Forward Substitution: 0.01055
Total time(seconds) taken by CPU for Backward Substitution: 0.0106
Total time(seconds) taken by CPU for LU Decomposition: 18.7366
Total time(seconds) taken by CPU for Jacobi: 0.365309
```

- **Listing of which nodes types we ran on:**
  - Standard compute (Ivy Bridge): comp-sc-0106
  - High memory: comp-hc-0015
  - Basic node: comp-bc-0352

## C Licensing and Publishing

- ~~Discussion of license choice:~~ **Discussion of license choice:**

  The GNU General Public License (GNU GPL or GPL) is a free, copyleft license. That means we can distribute it under the same license terms. Since we want our codes to be distributed, and tested among people, we chose GNU as our desired license. We did not need any warranty with this as the code was simple.

- ~~Discussion of location of publication:~~ **Discussion of location of publication:**
  The code is published in github.

  `https://github.com/amatur/cse597_solvers`. Since we want our results to be reproducible and tested by anyone interested, we published it in github with public access.

## References

[1] Lu decomposition takes more computational time than gaussian elimination! Retrieved from `https://autarkaw.org/2008/06/04/lu-decomposition-takes-more-computational-time-than-gaussian-elimination-what-gives/`, accessed 2018-09-23.

[2] Poisson blending. Retrieved from `http://eric-yuan.me/poisson-blending/`, accessed 2018-09-23.

[3] Poisson image editing. Retrieved from `http://www.ctralie.com/Teaching/PoissonImageEditing/`, accessed 2018-09-23.

[4] Solving the discrete poisson equation using jacobi, sor, conjugate gradients, and the fft. Retrieved from `https://people.eecs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html`, accessed 2018-09-23.

[5] Berisha, S., and Nagy, J. G. Iterative methods for image restoration. In *Academic Press Library in Signal Processing*, vol. 4. Elsevier, 2014, pp. 193–247.

[6] Bhat, P., Curless, B., Cohen, M., and Zitnick, C. L. Fourier analysis of the 2d screened poisson equation for gradient domain problems. In *European Conference on Computer Vision* (2008), Springer, pp. 114–128.

[7] Daks, A. Object detection data augmentation via poisson blending. Retrieved from `http://alondaks.com/2017/12/01/object-detection-data-augmentation-via-poisson-blending/`, accessed 2018-09-23.

[8] Hockney, R. W. A fast direct solution of poisson's equation using fourier analysis. Tech. rep., STANFORD UNIV CA STANFORD ELECTRONICS LABS, 1964.

[9] Paudel, K. Stitching of x-ray images, 2012.

[10] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. *ACM Transactions on graphics (TOG) 22*, 3 (2003), 313–318.

[11] Pérez, P., Gangnet, M., and Blake, A. Poisson image editing. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 313–318.

[12] Philip, S., Summa, B., Bremer, P.-T., and Pascucci, V. Parallel gradient domain processing of massive images. In *EGPGV* (2011), pp. 11–19.

[13] Saad, Y. *Iterative methods for sparse linear systems*, vol. 82. siam, 2003.