Solving Large Systems of Linear Equations*

Haocheng Dai

1 Quadratic Form

Given A (positive-definite and symmetric) and b, our target is to find a x_* such that

$$Ax_* = b. \tag{1}$$

If we want to find the solution of Eq.(1), we can just try to minimize the convex function $f(x)$:

$$f(x) = \frac{1}{2}x^T Ax - x^T b, \text{ where } \nabla f(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b. \tag{2}$$

As A is symmetric, the gradient reduces to

$$\nabla f(x) = Ax - b$$

Because A is positive-definite, the surface defined by $f(x)$ is shaped like a paraboloid bowl, which is illustrated by Figure 1. Only if $f(x)$ is a convex function, we can find a stationary point.

Gradient is the direction in which the function rises. And here are another two definitions we should memorize through out the note:

- **Error** $e_i = x_i - x_*$ is a vector indicates how far we are from the solution x_*.
- **Residual** $r_i = b - Ax_i = -\nabla f(x_i)$ is a vector indicates how far we are form the correct value b.

And the relationship between error and residual is

$$r_i = -Ae_i. \tag{3}$$

Remember whenever we read **residual**, think **"direction of steepest descent"**; whenever read **gradient** , think **"direction of steepest ascent"**.

*Figure 1,4,3,2,6,5 credit to [1]

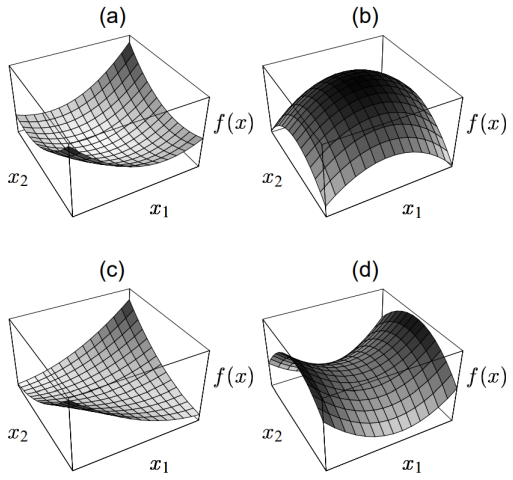

Figure 1: (a) Quadratic form for a positive-definite matrix. (b) For a negative-definite matrix. (c) For a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. (d) For an indefinite matrix. Because the solution is a saddle point, Steepest Descent and CG will not work. In three dimensions or higher, a singular matrix can also have a saddle.

2 Iterative Methods for Optimization

2.1 Steepest Descent Method

In the method of steepest descent, we start at an arbitrary point x_0 and slide down to the bottom of the paraboloid. When we take a step, we choose the direction in which f decreases most quickly, which is the direction $-\nabla f(x_i) = b - Ax_i$.

For example, we will choose a point

$$x_1 = x_0 + \alpha_0 r_0 \tag{4}$$

And the question is how big a step we should take? The line search is a procedure that chooses α to minimize f along the line, as Figure 2(c) shows. On the search line, f is minimized where the gradient is orthogonal to the search line, then we know how to determine α:

$$\begin{aligned}
r_{k+1}^T r_k &= 0 \\
(b - Ax_{k+1})^T r_k &= 0 \\
(b - A(x_k + \alpha_k r_k))^T r_k &= 0 \\
(b - Ax_k)^T r_k - \alpha_k (Ar_k)^T r_k &= 0 \\
\alpha_k (Ar_k)^T r_k &= (b - Ax_k)^T r_k \\
\alpha_k r_k^T A r_k &= r_k^T r_k \\
\alpha_k &= \boxed{\frac{r_k^T r_k}{r_k^T A r_k}}
\end{aligned} \tag{5}$$

In summary, the whole process can be described below:

1. Set $k = 0$, select an initial point $x_k \in \mathbb{R}^n$.
2. Set $r_k = b - Ax_k$. If $r_k = 0$, stop.
3. Update $\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}$.
4. Update $x_{k+1} = x_k + \alpha_k r_k$.
5. Set $k = k + 1$, then go to step 2.

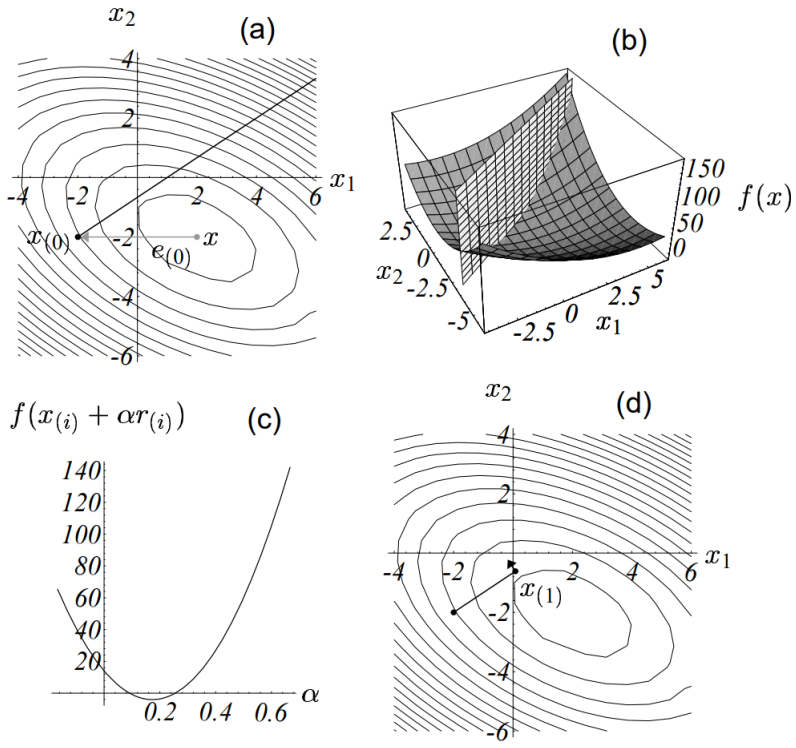

Figure 2: The method of Steepest Descent. (a) Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f. (b) Find the point on the intersection of these two surfaces that minimizes f. (c) This parabola is the intersection of surfaces. The bottom most point is our target. (d) The gradient at the bottom most point is orthogonal to the gradient of the previous step.

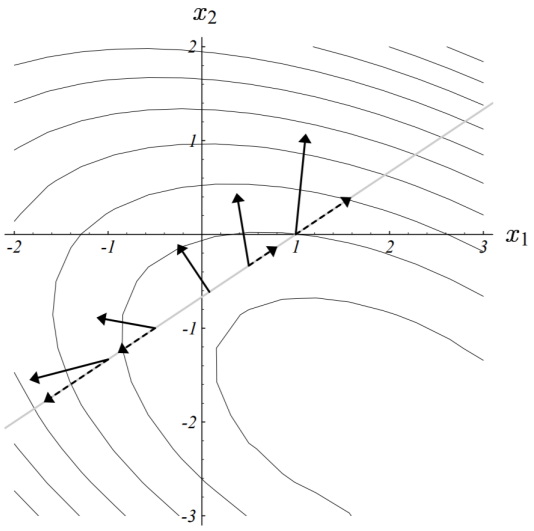

Figure 3: The gradient ∇f is shown at several locations along the search line (solid arrows). Each gradient's projection onto the line is also shown (dotted arrows). The gradient vectors represent the direction of steepest ascent of f, and the projections represent the rate of increase as one traverses the search line. On the search line, f is minimized where the gradient is orthogonal to the search line.

2.2 Conjugate Gradient Method

Steepest descent often finds itself taking steps in the same direction as earlier steps. Wouldn't it be better if, every time we took a step, we got it right the first time? Here's an idea: let's pick a set of orthogonal search directions $\{p_1, p_2, \cdots, p_n\}$. In each search direction, we'll take exactly one step, and that step will be just the right length to line up evenly with x_*. After n steps, we'll be done.

And here comes the conjugate gradient method. Remember that for any starting point x_0, the conjugate gradient method converges to the unique minimum x_* of f in only n steps. In our case, n is the total number of pixels in the image. A is a $n \times n$ symmetric and positive-definite matrix. Let $\{p_1, p_2, \cdots, p_n\}$ be A-conjugate vectors. (We cannot have more than n linearly independent vectors in $\mathbb{R}^n$, hence we cannot have more than n vectors that are A-conjugate. Plus, if $A = I$, then $\{p_1, p_2, \cdots, p_n\}$ are orthogonal vectors.) Since these vectors are independent, we can denote solution x_* with

$$x_* = x_0 + \sum_{i=1}^{n} \alpha_i p_i, \tag{6}$$

where α_i is the step size along direction p_i.

Eq.(7) is the update rule:

$$\begin{aligned} x_{k+1} &= x_k + \alpha_k p_k \\ x_1 &= x_0 + \alpha_0 p_0 \\ x_2 &= x_0 + \alpha_0 p_0 + \alpha_1 p_1 \\ x_k &= x_0 + \alpha_0 p_0 + \cdots + \alpha_{k-1} p_{k-1} \\ x_* &= x_n = x_0 + \alpha_0 p_0 + \cdots + \alpha_{n-1} p_{n-1} \end{aligned} \tag{7}$$

From Eq.(7), we can get

$$x_k - x_0 = \alpha_0 p_0 + \cdots + \alpha_{k-1} p_{k-1} \tag{8}$$

$$x_* - x_0 = \alpha_0 p_0 + \cdots + \alpha_{n-1} p_{n-1} \tag{9}$$

Since p_k is conjugate to each other, for every $k \neq i$, $p_k^T A p_i = 0$. Times $p_k^T A$ on both sides of Eq.(9), we'll get

$$p_k^T A(x_k - x_0) = p_k^T A(\alpha_0 p_0 + \cdots + \alpha_{k-1} p_{k-1}) = 0 \tag{10}$$

$$p_k^T A(x_* - x_0) = p_k^T A(\alpha_0 p_0 + \cdots + \alpha_{n-1} p_{n-1}) = \alpha_k p_k^T A p_k \tag{11}$$

From Eq.(11), we'll get

$$\alpha_k = \frac{p_k^T A(x_* - x_0)}{p_k^T A p_k} = \frac{p_k^T A(x_* - x_k + x_k - x_0)}{p_k^T A p_k} \tag{12}$$

Take Eq.(10) into Eq.(12), we'll get

$$\begin{aligned} \alpha_k &= \frac{p_k^T A(x_* - x_k)}{p_k^T A p_k} \\ &= \frac{p_k^T (A x_* - A x_k)}{p_k^T A p_k} \\ &= \frac{p_k^T (b - A x_k)}{p_k^T A p_k} \\ &= \boxed{\frac{p_k^T r_k}{p_k^T A p_k}} \end{aligned} \tag{13}$$

r_k is the negative gradient of f at x_k, along which the steepest descent method would require to move. It can be updated by it's definition:

$$r_k = \boxed{b - A x_k} \tag{14}$$

Or, it can just be updated by taking the relationship between e and r, which is usually how we implemented in algorithm:

$$\begin{aligned} x_{k+1} &= x_k + \alpha_k p_k \\ x_{k+1} - x_* &= x_k - x_* + \alpha_k p_k \\ e_{k+1} &= e_k + \alpha_k p_k \\ -A e_{k+1} &= -A e_k - \alpha_k A p_k \\ r_{k+1} &= \boxed{r_k - \alpha_k A p_k} \end{aligned} \tag{15}$$

According to experiment, there's not much numerical difference between the above two update methods. However, we are not doing steepest descent here, which may still have slight move on previous search directions p and cause "zig-zag" path in descent. **The α_k above already tells us how long a step should take in the corresponding p_k such that p_k is conjugate to all the previous p,** and that is guaranteed by Eq.(10).

Then the problem comes to how to determine p_k, and we give the following expression (imagine β is a negative number, so it's just like subtracting conjugation from residual)

$$\boxed{p_k = r_k + \sum_{i<k} \beta_i p_i = r_k + \beta_{k-1} p_{k-1},} \text{ where } p_0 = r_0. \tag{16}$$

It's analogous to the usual Gram-Schmidt process for obtaining an orthogonal basis. β_i **indicates the conjugate projection component on p_i.** The update rule for p_k is that, in each iteration, the new conjugate direction is current negative gradient eliminated all the previous conjugate directions, as Figure 4 illustrates.

Since it's an iterative method, we only have to consider eliminating last search direction p_{k-1} in term of conjugation. **Residual is orthogonal (error is conjugate) to all previous search directions**, which can be explained by derivation below:

$$\begin{aligned}
e_k = x_k - x_* &= -\sum_{i=k}^{n} \alpha_i p_i \\
p_j^T A e_k &= -\sum_{i=k}^{n} \alpha_i p_j^T A p_i && j < k \\
-p_j^T r_k &= -\sum_{i=k}^{n} \alpha_i p_j^T A p_i \\
p_j^T r_k &= 0 && j \neq i, \text{ then } p_i \text{ is conjugate to } p_j
\end{aligned} \tag{17}$$

Also **residual is orthogonal (not conjugate) to all previous residuals**, which can be explained by derivation below:

$$\begin{aligned}
p_k &= r_k + \sum_{i<k} \beta_i p_i \\
r_j^T p_k &= r_j^T r_k + \sum_{i<k} \beta_i r_j^T p_i && k < j \\
r_j^T r_k &= 0 && \text{according to Eq.(17)}
\end{aligned} \tag{18}$$

As each new search direction is constructed from the residual to be A-orthogonal to all the previous residual and search directions, the linear span of previous search directions and residual are identical. So comes from the above two conclusions.

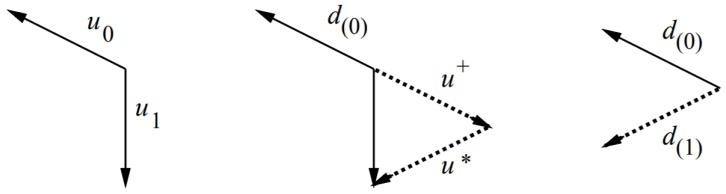

Figure 4: Gram-Schmidt conjugation of two vectors. Begin with two linearly independent vectors u_0 and u_1. Set $d_0 = u_0$. The vector u_1 is composed of two components: u^*, which is conjugate to d_0 and u^+, which is parallel to d_0. After conjugation, only the A-orthogonal portion remains, and $d_1 = u^*$. **If we stretch the space until the ellipses appeared circular, we'll find that d_0 and d_1 are strictly orthogonal.**

Similar to the solution of α, we use the conjugate property of p_i to calculate β_k. Times $p_k^T A$ on both sides of Eq.(12), it goes to

$$p_k^T A p_{k+1} = p_k^T A(r_{k+1} + \sum_{i<k+1} \beta_i p_i) \tag{19}$$

$$0 = p_k^T A r_{k+1} + \sum_{i<k+1} \beta_i p_k^T A p_i \tag{20}$$

$$0 = p_k^T A r_{k+1} + \beta_k p_k^T A p_k \tag{21}$$

From Eq.(19), we get the expression of β:

$$\beta_k = \boxed{-\frac{p_k^T A r_{k+1}}{p_k^T A p_k}} \tag{22}$$

Let us simplify this expression by taking the inner product of r_k and Eq.(14):

$$\begin{aligned}
r_{k+1}^T r_{k+1} &= r_{k+1}^T r_k - \alpha_k r_{k+1}^T A p_k \\
\alpha_k r_{k+1}^T A p_k &= r_{k+1}^T r_k - r_{k+1}^T r_{k+1} \\
\alpha_k r_{k+1}^T A p_k &= -r_{k+1}^T r_{k+1} \text{ (Each new residual is orthogonal to all the previous residuals.)} \\
r_{k+1}^T A p_k &= -\frac{1}{\alpha_k} r_{k+1}^T r_{k+1} \\
p_k^T A r_{k+1} &= -\frac{1}{\alpha_k} r_{k+1}^T r_{k+1} \text{ (A is symmetric.)}
\end{aligned} \tag{23}$$

Therefore by substituting Eq.(12) and Eq.(20) into Eq.(19), β can be simplified as

$$\begin{aligned}
\beta_k &= \frac{1}{\alpha_k} \frac{r_{k+1}^T r_{k+1}}{p_k^T A p_k} \\
&= \frac{p_k^T A p_k}{p_k^T r_k} \frac{r_{k+1}^T r_{k+1}}{p_k^T A p_k} \\
&= \boxed{\frac{r_{k+1}^T r_{k+1}}{p_k^T r_k}}
\end{aligned} \tag{24}$$

With x_k, α_k and p_k, the update rule is as clear as

$$x_{k+1} = x_k + \alpha_k p_k \tag{25}$$

In summary, the whole process can be described below:

1. Set $k = 0$, select an initial point $x_0 \in \mathbb{R}^n$.
2. Set $r_0 = -\nabla f(x_0), p_0 = r_0$. If $r_0 = 0$, stop.
3. Solve the step size along p_k: $\alpha_k = \frac{p_k^T r_k}{p_k^T A p_k}$.
4. Update the estimation with α_k, p_k: $x_{k+1} = x_k + \alpha_k p_k$.
5. Set $r_{k+1} = -\nabla f(x_{k+1})$ or $r_{k+1} = r_k - \alpha_k A p_k$. If $r_{k+1} = 0$, stop.
6. Solve the conjugation components among r_k: $\beta_k = \frac{r_{k+1}^T r_{k+1}}{p_k^T r_k}$.
7. Doing conjugation on r_{k+1}: $p_{k+1} = r_{k+1} + \beta_k p_k$.
8. Set $k = k + 1$, then go to step 3.

2.3 Iterative Method

The key to making progress is to note that in general, the matrix A is extremely sparse, since the linear relationships usually only relate nearby grid points together. We therefore seek methods which don't require ever explicitly specifying all the elements of A, but exploit its special structure directly. Many of these methods are iterative - we start with a guess x_k, and apply a process that yields a closer solution x_{k+1}.

Let's recap the idea of Eigen first: $Ax = \lambda x$. If $|\lambda| < 1$, then $A^i v = \lambda^i v$ will vanish as i approaches infinity. If $|\lambda| > 1$, then $A^i v$ will grow into infinity as i approaches infinity.

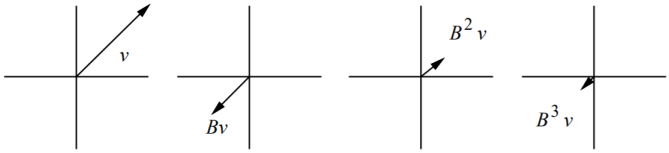

Figure 5: As i increases, $A^i v$ converges to zero with $|\lambda| < 1$.

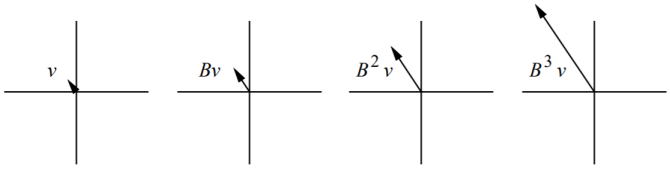

Figure 6: As i increases, $A^i v$ diverges to infinity with $|\lambda| > 1$.

A very important skill in understanding linear algebra is to think of a vector as a sum of other vectors whose behavior is understood. Say a vector x is illustrated as a sum of two eigenvectors v_1, v_2. Applying A to x is equivalent to applying A to the eigenvectors, and summing the result.

On repeated application, we have $A^i x = A^i v_1 + A^i v_2 = \lambda^i v_1 + \lambda^i v_2$. If the magnitudes of all the eigenvalues are smaller than 1, $A^i x$ will converge to 0, because the eigenvectors that compose x converge to 0 when A is repeatedly applied. If one of the eigenvalues has magnitude greater than 1, x will diverge to infinity. This is why numerical analysts attach importance to the **spectral radius** of a matrix:

$$\rho(R) = max|\lambda_j|, \text{ where the } \lambda_j \text{ are the eigenvalues of } R \tag{26}$$

An iterative scheme converges if and only if $\rho(R) < 1$. The size of the spectral radius determines the convergence rate, and ideally we would like to find splittings which result in as small a $\rho(R)$ as possible.

Jacobi Method. The Jacobi method is one of the simplest iterations to implement. We split A into $D+R$, where D is the diagonal component and R is the remainder. The convergence properties are then set by the matrix $D^{-1}R$.

$$\begin{aligned} Ax &= b \\ (D+R)x &= b \\ Dx &= b - Rx \\ x^{(m+1)} &= D^{-1}(b - Rx^{(m)}) \end{aligned} \tag{27}$$

We start with an initial guess x_0, and then successively improve it according to Eq.(22). And entry-wise iteration is shown as below:

Algorithm 1 Jacobi Method

for $j = 1$ to N^2 **do**

$x_j^{(m+1)} = \frac{1}{a_{jj}}(b_j - \sum_{k \neq j} a_{jk} x_k^{(m)})$

end for

where $\frac{1}{a_{jj}}, b_j, a_{jk}$ correspond to D^{-1}, b, R respectively.

Gauss-Seidel Method. The Gauss–Seidel method improves on the Jacobi algorithm, by noting that if we are up-dating a particular point $u_j^{(m+1)}$, we might as well reference the already updated values $x_1^{(m+1)}, x_2^{(m+1)}, \ldots, x_{j-1}^{(m+1)}$ in the calculation, rather than using the original values $x_1^{(m)}, x_2^{(m)}, \ldots, x_{j-1}^{(m)}$. According to the property of Gauss-Seidel, we split A into $L_* + U$, where L_* is a lower triangular component and U is a strictly upper triangular component.

$$\begin{aligned} Ax &= b \\ (L_* + U)x &= b \\ L_* x &= b - Ux \\ x^{(m+1)} &= L_*^{-1}(b - Ux^{(m)}) \end{aligned} \tag{28}$$

Algorithm 2 Gauss-Seidel Method

for $j = 1$ to N^2 **do**

$x_j^{(m+1)} = \frac{1}{a_{jj}}(b_j - \sum_{k=1}^{j-1} a_{jk} x_k^{(m+1)} - \sum_{k=j+1}^{N^2} a_{jk} x_k^{(m)})$

end for

The entry-wise iteration can be written as:

Multigrid Method. This method is to make the progress converge faster on a final grid. The idea of multigrid is to take advantage of the fact that Jacobi and Gauss-Seidel method converge very fast on a coarse grid. And the effect of the iterations is essentially smoothing out the solution error. Initially, the solution error is the true solution itself. But over the iterations, the error gets way more smoother. The important steps are:

- **Smoothing** – reducing high frequency errors, using a few iterations of the Jacobi or Gauss–Seidel method.
- **Residual Computation** – computing residual error after the smoothing operation(s).
- **Restriction** – downsampling the residual error to a coarser grid.
- **Interpolation** – interpolating a correction computed on a coarser grid into a finer grid.
- **Correction** – Adding prolongated coarser grid solution onto the finer grid.

Figure 1 and 2 shows the convergence of the conjugate gradient methods with and without preconditioner, respectively. On small size Lena, the CG method without preconditioner takes 54 iterations for residual's norm to reach below 10, while the one with preconditioning takes 22 interations. On large size Lena, the CG method without preconditioner takes 112 iterations for residual's norm to reach below 50, while the one with preconditioning takes only 87 interations.

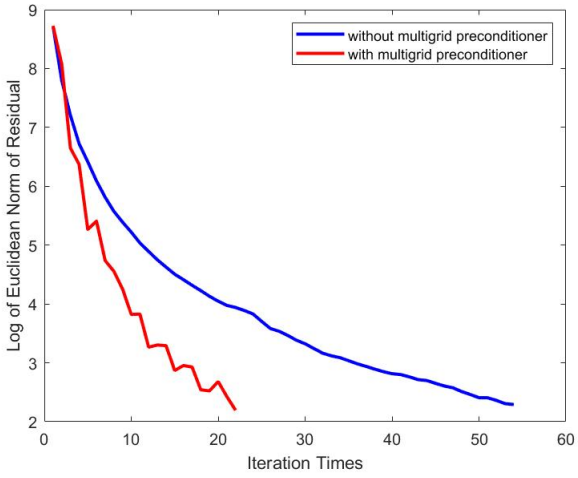

Figure 7: The convergence on small size Lena.

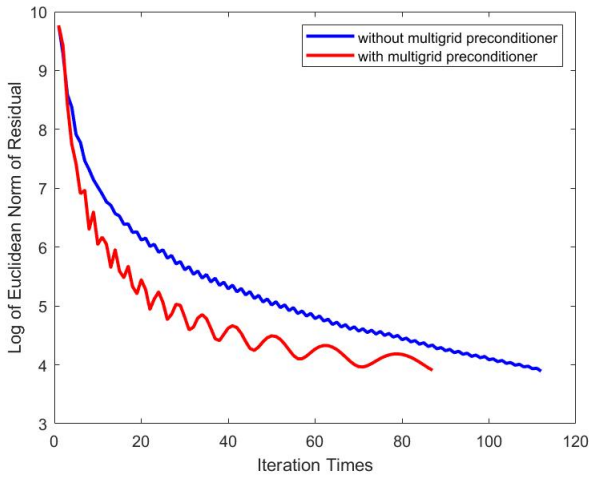

Figure 8: The convergence on large size Lena.

3 Implementation

3.1 Conjugate Gradiant Method

The b, A, x_0 here correspond to above F, E, U, respectively. And I initialize the F, g to Laplacian filtered *lena*, all-one-matrix, respectively. The condition for stopping the iteration is $r_{k+1} < 10$.

Figure 3 shows the result where $g(x, y) = x^2 + y^2$. The progress takes 59 iterations in total.

Figure 9: From left to right: G, U, F.

3.2 Iterative Method

Below is my implementation of multigrid method. As a preconditioner, the multigrid works just like $z_1 = M^{-1}r_1$, so the multigrid function should be called like $z_1 = multigrid(0, r_1)$.

Algorithm 3 Multigrid

function MULTIGRID(x_0, b)
  for $i = [1 : 10]$ **do**
    $x_0 = jacobi(x_0, b)$ ▷ Smoothing
  end for
  $residual = Ax - b$ ▷ Residual Computation
  $residual\_coarse = sample(residual)$ ▷ Restriction
  if $size(b, 1) > 1$ **then**
    $error\_coarse = multigrid(error\_coarse, residual\_coarse)$
  end if
  $error = interpolate(error\_coarse)$ ▷ Interpolation
  $x = x_0 - error$ ▷ Correction
  return x
end function

Figure 4 shows the result of multigrid precondtioned congjugate gradient methond.

Figure 10: From left to right: G, U, F.

References

[1] J. R. Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.