Understanding the Bi Conjugate Gradient Stabilized Method (Bi-CGSTAB)

Vinay, Yuvashankar Mojdeh, Sayari Nejad Ash (Chang), Liu

June 25, 2016

Contents

1 Introduction

The Biconjugate Gradient Stabilized Method (abbreviated to Bi-CGSTAB from now on)[18] is an iterative method for solving sparse matrix problems of the form

$$Ax = b \tag{1}$$

The vector x is the unknown, and A and b are known. x and b are vectors of size $n \times 1$, and A is a nonsingular square matrix of size $n \times n$. A is sparse and n is large.

There are a lot of methods for computing the solution for $Ax = b$ that go all the way back to Gauss and Turing. Today one could easily solve a small system of linear equations in MATLAB using the famous `x = A\b` command.

But there are systems of equations where the "\" command is either inefficient, or far too expensive to solve. These tend to be the larger systems of equations.

Bi-CGSTAB is developed for a "Square" Matrix, that is non "symmetric positive definite " (abbreviated to SPD). We shall review these terms in the following section.

This is a very simple guide on understanding the Bi-CGSTAB method. We are going to assume that you have taken a course in numerical methods, or at the very least, know how to multiply a Matrix and a Vector.

2 Review

2.1 Matrix Properties

The general shape of the equation $Ax = b$ is of the form.

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{n1} & \dots & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

For this document, matrices will be denoted with uppercase letters, and vectors will be denote with lowercase letters.

2.2 Square Matrices

A matrix is considered to be Square if it has the same number of columns as rows. This is sometimes referred to as an $n \times n$ matrix in literature.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

2.3 Symmetric Positive Definite

A matrix is said to be Symmetric if $A^T = A$. In other words a matrix is symmetric if the transpose of the matrix is still itself. If you are unsure of how to transpose a matrix, reference[10] has a great tutorial. A matrix is said to be positive definite if

$$\forall x \neq 0, x^T Ax > 0. \tag{2}$$

The basic idea for Positive Definite matrix is that the eigenvalues for the matrix are all greater than zero and that their magnitude is ordered such that the first eigenvalue is the largest and the last eigenvalue is the smallest. Formalized as

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n > 0 \tag{3}$$

A great example of a Positive Definite Matrix is the Identity Matrix.

2.4 Vector Projections and Vector Orthogonalities

In Euclidian geometry, a vector projection is when you project a vector a onto another vector b. Suppose it is midday, and the sun is directly overhead. You decide to place one stick on the ground, and another stick at some angle to the first stick. How long is the shadow?

That is the idea behind vector projections. How long is the shadow (in mathematics it is called magnitude) of vector a onto b when you shine a light perpendicular to b (in mathematics this is called projection).

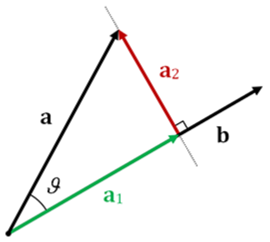

Figure 1: Vector projections in Cartesian space

We formally define a vector projection as

$$p = a \cdot b \tag{4}$$

Two orthogonal vectors are vectors that cast no shadow on each other, this occurs only when the first vector is perpendicular to the second vector. We define this rule as

$$0 = a \cdot b \tag{5}$$

In linear algebra, one can perform projections in the same manner, but instead of $a \cdot b$ we define projections as

$$p = (a, b) \tag{6}$$

This is shorthand for

$$p = a^T b \tag{7}$$

To see this in action, let us confirm that the unit vectors for x and y are orthogonal to each other.

$$\begin{aligned} x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, y &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ p = (x, y) &= x^T y \qquad (8) \\ p = \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} &= 0 \end{aligned}$$

2.5 Eigenvectors and Eigenvalues

An eigenvector v of matrix A is a vector whose direction remains unchanged when multiplied by A. Applying A may change the magnitude of v, but it does not change its direction. The amount of stretching or shrinking is call the corresponding eigenvalue. In other words, there is some scalar constant λ such that $Av = \lambda v$. The value λ is an eigenvalue of A. Note that if we scale an eigenvector, the result is still an eigenvector[12].

This pair is of importance to us because iterative methods often depend on applying one matrix to a vector over and over again. When A is repeatedly applied to an eigenvector v, two things can happen. Either λ is smaller than 1, so $A^i v$ will vanish when i approaches infinity or λ is greater than 1 and $A^i v$ grows at each iterate.

The eigenvalues of a matrix A are called its spectrum and are denoted by $\sigma(A)$. We will refer to spectrum while explaining preconditioning.

Now that we have a basis of the tools we need to understand Bi-CGSTAB, we need to first understand how mathematicians developed the algorithm.

3 Projection Methods and Krylov Subspaces

Bi-CGSTAB is a Krylov Subspace method, which is a projection method used to solve a problem iteratively.

In order to understand what Bi-CGSTAB does, we must first fully understand what all of those words in the previous sentence mean. We will start with the Projection Method,and build ourselves up from there.

3.1 Projection Methods

A projection method is a method that finds a solution to a higher dimensional problem in the lower dimensions. In other words, it finds a solution to a problem within its subspaces. The problem $Ax = b$ lies in the $\mathbb{R}^{n \times n}$. This is because the Matrix A is two dimensional. But if you try to solve the equation differently such as $b - Ax = 0$, the solution to this problem lies in the $\mathbb{R}^n$. It is easy to see that $\mathbb{R}^n$ is a subspace of $\mathbb{R}^{n \times n}$. This makes intuitive sense, as we are narrowing the domain of the problem, and therefore, should be able to converge to a solution faster than when the domain of the problem was of a higher dimension.

There are two types of projection methods, orthogonal and oblique, both of which we will need to understand.

Let us start with the equation

$$Ax = b \tag{9}$$

Where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. We want to compute a solution to this problem such that

$$b - Ax = 0 \tag{10}$$

The difference between the $\mathbf{0}$ vector and the solution will be denoted by r which is defined as the residue.

$$b - Ax = r \tag{11}$$

Let us represent the entire Real number field as $\mathcal{K}$, this is the domain of every possible real number. As an extension of that, our problem lies inside of the Real Number Field in a vector space called $\mathcal{K}_m$where m is the dimensions of the Vector Space. A subspace is denoted by $\mathcal{L}$, and a subspace with m dimensions is denoted by $\mathcal{L}_m$.

Dimensions in this case are not what most people assume dimensions are. It is possible to have more than 3 dimensions in a subspace. In the context of linear algebra, you can have a subspace of 5 or 6 dimensions of vectors, but it would still be considered a subspace to the superspace of the matrix domain.

A subspace is not something that we could travel into, but rather, it is an abstract representation of orthogonal matrices and vectors. To help visualize it in our three dimensional brains, let's pretend that the x direction is in terms of a vector in $\mathbb{R}^n$. The y direction would be another vector in $\mathbb{R}^n$, such that $x^T y = 0$. The z direction would be another vector orthogonal to both the x and y vectors. This is the limit of the Euclidean geometry that we all experience. But, in linear algebra, one could keep adding dimensions to this space as long as they were all orthogonal to the previous vector spaces.

The goal of the projection method is to find an $\hat{x}$ such that

$$b - A\hat{x} \perp \mathcal{L}_m \text{ and } \hat{x} \in \mathcal{K}_m \tag{12}$$

This means that the projection method needs to find a residual vector such that it is orthogonal to the subspace. It also needs to be within the constraints of the $\mathcal{K}_m$ subspace.

When the residual is orthogonal to the subspace, the residual is at a minimum.

For our problem, our $\mathcal{K}_m$ and $\mathcal{L}_m$ would be a vector space of $nx1$ vectors of m dimensions.

If we make an initial guess x_0 then we can rewrite equation (11) as

$$r_0 = b - Ax_0 \tag{13}$$

Making a slight adjustment to our definition of projection being that

$$b - A\hat{x} \perp \mathcal{L}_m \text{ and } \hat{x} \in x_0 + \mathcal{K}_m \tag{14}$$

If we further assume that the final solution $\hat{x}$ can be rewritten as

$$\hat{x} = x_0 + \delta \text{ such that } \delta \in \mathcal{K} \tag{15}$$

Where δ is a vector that will make the initial guess satisfy equation $b - Ax = 0$.

We can say that $b - A(x_0 + \delta) \perp \mathcal{L}$ or rearranged:

$$r_0 - A\delta \perp \mathcal{L} \tag{16}$$

Meaning that the initial residual subtract $A\delta$ should be orthogonal to the Subspace $\mathcal{L}$.

We can finally say that

$$\hat{x} = x_0 + \delta \text{ when } \delta \in \mathcal{K} \tag{17}$$

$$(r_0 - A\delta, w) = 0, \forall w \in \mathcal{L} \tag{18}$$

This means that our final solution to $Ax = b$ is $\hat{x}$ and if $\hat{x}$ is defined as some initial guess x_0 plus some vector δ where δ is within the subspace $\mathcal{K}$. This matrix δ must satisfy the other constraint where the inner product of $r_0 - A\delta$ and any matrix w, must be zero.

In the context of a Krylov Subspace, and two dimensional matrices,the superspace is $\mathbb{R}^{n \times n}$ while the subspace is $\mathbb{R}^n$. δ, $\hat{x}$, x_0 and w are of the subspace $\mathbb{R}^n$

Now the question is how to choose the subspace such that we converge to an answer, and that is where Krylov comes in to save the day.

3.2 Iterative Methods

For solving large linear systems, using a direct method could be expensive in terms of time and storage. Sometimes the matrix is too large that cannot be stored in memory directly. There are methods which compute successive iterations to approximate the solution of $Ax = b$. They do not require storage of the whole matrix, but only the ability to compute matrix vector multiplication. By using iterative methods we can have a significant decrease in computation cost and complexity.

3.3 Power Method

For large matrices we cannot use full transformations and we do not often need all the eigenvalues and eigenvectors. Instead we try to find a proper basis to capture the relevant eigenpairs. That is, the idea behind defining new subspaces. Assume a matrix A of size $n \times n$ and a vector b of size $n \times 1$. We start multiplying b by A iteratively, i.e. the first time we multiply b by A, then multiply result by A, and do this procedure iteratively. At the end, this sequence of vectors will converge to an eigenvector of A which is corresponding to the largest absolute eigenvalue of A.[7]

To demonstrate this claim, let us assume that vector b has some component in the direction of the largest eigenvalue. After each iteration, this component gets larger in comparison to other components orthogonal to the largest eigenvector, and it starts dominating over the other components.

The following formulas describes the above-mentioned procedure: vector b is composed of n components:

$$b = c_1 v_1 + c_2 v_2 + \ldots + c_n v_n \tag{19}$$

Where c_i is scalar and v_i is of the same size of b. Multiply both sides by A and use the linearity of matrix multiplication:

$$Ab = c_1 A v_1 + c_2 A v_1 + \ldots + c_n A v_n \tag{20}$$

While multiplying b by A, the components in the same direction will multiply, so if A has the eigenvectors e_1, e_2 $\ldots$ e_n we will have:

$$Ab = c_1 e_1 v_1 + c_2 e_2 v_2 + \ldots + c_n e_n v_n \tag{21}$$

Repeating the multiplication by k times, we obtain:

$$A^k b = c_1 e_1^k v_1 + c_2 {e_2}^k v_2 + \ldots + c_n {e_n}^k v_n \tag{22}$$

Now for large k, since e_1 is greater in absolute value than all the other e_is, e_1 will be much greater than absolute value of e_i (all $i = 2 \ldots n$). Hence $c_1 e_1 v_1$ dominates the others and we have:

$$A^k b = c_1 {e_1}^k v_1 \text{ for large values of } k \tag{23}$$

Note that as we mentioned before, any multiple of an eigenvector is still an eigenvector, so we have found an eigenvector corresponding to e_1.

Now we want to see why approximations over this subspace is good. For this purpose, we need to have knowledge about similarity transformations and orthogonal similarity transformations.

Briefly, the similarity transformation for matrix A is a mapping with a transformation matrix $\hat{A}$ such that $\hat{A} = BAB^{-1}$ where B is a nonsingular matrix. An orthogonal similarity transformation for matrix A is of form $Q^T A Q$ where $Q^T Q = I$. Hence, approximation over range Q is:

$$range(Q) : L = Q^T A Q \tag{24}$$

Now let $AQ - QL = R$. If $\frac{R}{A}$ is sufficiently small, then Q is acceptable. The power method leads to a good Q as it converges to the Eigen pair with largest absolute eigenvalue which is badly conditioned, but it is faster and more straightforward.

3.4 Three-Term Recurrence

A three term recurrence relation is a specification on how to compute a new value with the two previous values. There are two different types of three-term recurrence relations, an inhomogeneous three-term recurrence relation and a uniform three-term recurrence relation. A uniform three-term recurrence relation is defined as:

$$y_{n+1} = a_n y_n + b_n y_{n-1} \tag{25}$$

where a and b are arbitrary complex numbers. You can study the other version in [8], we will not need them for this particular method.

3.5 Krylov Subspace

The Power method provides one-vector subspace. Krylov Subspace is an improvement on the Power method that keeps all the vectors. To solve $Ax = b$, if we are only given A and b, the approach would be to try to multiply b by A iteratively, starting from b, to get Ab, A^2b, and so on. In each iteration there is a product of a sparse matrix and a vector which is very quick in the computer. We can use this idea to build a subspace called the Krylov Subspace.

To be more detailed , we compute the residual according to the following equation $r = b - Ax$, where the residual is the difference between real and computed x. In order to compute x_{k+1} one can compute $x_{k+1} = x_k + r_k$ where r_k is the residual at step k. We start with an initial guess. Assume that $x_1 = b$, we have

$$x_2 = b + r_1 \tag{26}$$

$$x_3 = b + r_2 \tag{27}$$

$$\vdots \tag{28}$$

$$x_{k+1} = b + r_k \tag{29}$$

As you can see, x_k is a combination of b, Ab, A^2b, ... Each two terms of the previous sequence are linearly independent, as we cannot define them as a linear combination of each other, so the linear combination of b, Ab, A^2, ..., A^k forms the *jth* Krylov subspace.

Note that this sequence is the same as the sequence generated by the power method, however, in the Krylov subspace we keep all vectors and the subspace is spanned by all of the iterates, rather than using only the last vector in the sequence.

To show the subspace, we can use the term span. Span is defined as follows: let S be a nonempty subset of a vector space $\mathcal{V}$, then the span of $\mathcal{S}$, denoted by $span(S)$, is a set containing of all linear combinations of vectors in S.

We use term $\mathcal{K}_k$ to show the Krylov subspace and $\mathcal{S}_k$ to show the Krylov Span.

$$\mathcal{K}_k(A, b) = \{all\ linear\ combinations\ of\ : b, Ab, A^2b, \ldots, A^kb\} \tag{30}$$

$$\mathcal{S}_k(A, b) = \{b, Ab, A^2b, \ldots, A^kb\} \tag{31}$$

Based on the previous explanations, we also can define the Krylov Span in terms of r_0:

$$\mathcal{S}_k(A, r_0) = \{r_0, Ar_0, A^2r_0, \dots, A^kr_0\} \tag{32}$$

In the literature a Krylov Subspace $\mathcal{K}_m$ is define as

$$\mathcal{K}_k(A, b) = span\{b, Ab, A^2b, \dots, A^{m-1}b\}[1] \tag{33}$$

Where $r_0 = b - Ax_0$. One additional specification is that the vectors can only exist in the Krylov Subspace if its corresponding linear system has a solution. But since we have assumed that the matrix A is nonsingular, we know that it has one unique solution.

The vectors A_kb converge to the direction of the eigenvector corresponding to the largest eigenvalue of A. Thus, this basis tends to be badly conditioned. Therefore an orthogonalization process is applied to the basis vectors in their natural order to obtain numerical stability and to replace these vectors by orthogonal vectors $q_1, q_2, \dots, q_k$ that span the same subspace.

3.6 Orthogonalizing Basis for the Krylov Subspace

The iteration used to construct orthonormal basis

$$\{q_1, q_2, \dots, q_j\}$$

for the Krylov subspace is called Arnoldi's method. This method is like the Gram-Schmidt process[4] with a slight modification.

It is based on the fact that each Krylov subspace can be obtained from the orthonormal basis of the Krylov subspace of one dimension less using the spanning set q_1, q_2, ... ,q_j, Aq_j. In other words, a new direction in the Krylov subspace can be created by multiplying the most recent basis vector q_j by A rather than by multiplying $A_{j-1}b_0$ by A.

3.7 Arnoldi's method

The following algorithm describes Arnoldi's orthogonalization of $b, Ab, \dots, A_{j-1}b$: Remember that the vector b is our initial guess.

Arnoldi's Orthogonalization Algorithm

```
1: $q_1 = b/||b||$;
2: for $j = 1, \dots, n-1$ do
3:     $t = Aq_j$;
4:     for $i = 1, \dots, j$ do
5:         $h_{ij} = q_i^T t$;
6:         $t = t - h_{ij}q_i$;
7:     end for
8:     $h_{j+1,j} = ||t||$;
```

```
9:    $q_{j+1} = t/h_{j+1,j}$;
10: **end for**
```

What this method does is to first normalize vector b and call the normalized vector as q_1. Then in each iteration it multiply q_j by A, we'll call this result

$$t = Aq_j \tag{34}$$

it then orthonormalizes it to every other vector in our subspace. To do so, first it computes q_i^T, where i is the index of every vector that we already have in our subspace, and then multiply it by t. Then it subtracts the projection vector from the original vector to obtain an orthogonal vector to q_i. It then continues orthogonalizing the vector t to every other vector in the subspace. Finally it normalizes the obtained vectors q_j to reach an orthonormal basis.

The orthonormalized basis is $\{q_1, q_2, \ldots, q_j\}$. if the matrix A is symmetric, then this basis is called Lanczos basis.

3.8 Lanczos Bi-Orthogonalization method

Methods like Bi-CGSTAB use two subspaces. One way to simultaneously obtain the bases for these subspaces is to apply the Lanczos Bi-orthogonalization method. It is a process for generating two finite subspaces. It uses the following spans to create these subspaces.

$$S_k(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^kr_0\} \tag{35}$$
$$S_k(A^T, r_0) = span\{r_0, A^Tr_0, (A^T)^2r_0, \ldots, (A^T)^kr_0\} \tag{36}$$

The subspaces according to the above-mentioned spans are defined as:

$$\mathcal{K}_k(A, v_1) = \{v_1, v_2, v_3, \ldots, v_k\} \tag{37}$$
$$\mathcal{K}_k(A^T, w_1) = \{w_1, w_2, w_3, \ldots, w_k\} \tag{38}$$

$\mathcal{K}_k(A, v_1)$ and $\mathcal{K}_k(A^T, w_1)$ are Krylov subspaces of A and A^T, respectively. For v_j and w_i to be orthogonal they should meet the following bi-orthogonality condition:

$$(v_j, w_j) = \delta_{i,j} \tag{39}$$

where $\delta_{i,j}$ is the Kronecker delta [23].

Considering this condition, and considering the fact that all sequences of orthogonal polynomials satisfy a three term recurrence relation, we conclude that for both sequences of vectors v and w, the three term recurrence holds. This is implemented in lines 9 and 10 of the following algorithm. We put v_1 equal to $\frac{r_0}{||r_0||}$, and w is an arbitrary vector usually considered as equal to v_1. Remember that our Matrix A is of size $n \times n$ and vector r_0 is of size $n \times 1$. The following steps represent the Lanczos Bi-orthogonalization procedure:

Lanczos's Bi-Orthogonalization Algorithm

```
1: $v^{(1)} = r^{(0)}/||r^{(0)}||_2$
2: $w^{(1)} = v^{(1)}$
3: $\beta_0 = 0, \gamma_0 = 0$
4: $v^{(0)} := 0, w^{(0)} = 0$
5: for $j = 1 : k$ do
6:     $s = Av^{(j)}$
7:     $z = A^T w^j$
8:     $\alpha_j = (w^{(j)}, s)$
9:     $\tilde{v}^{(j+1)} = s - \alpha_j v^{(j)} - \beta_{j-1} v^{(j-1)}$
10:    $\tilde{w}^{(j+1)} = s - \alpha_j v^{(j)} - \gamma_{j-1} w^{(j-1)}$
11:    $\gamma_j = ||\tilde{v}^{(j+1)}||_2$
12:    $v^{(j+1)} = \tilde{v}^{(j+1)}/\gamma_j$
13:    $\beta_j = (\tilde{w}^{(j+1)}, v^{(j+1)})$
14:    $w^{j+1} = \tilde{w}^{(j+1)}/\beta_j$
15: end for
```

Coefficients β_j and α_j are determined by enforcing the bi-orthogonality condition of w_j and v_j. If w_j is orthogonal to all of the vectors v_i except when $i = j$ then

$$(w_j)^T A v_j = \alpha_j (w_j)^T v_j \tag{40}$$

$$\alpha_j = (w_j)^T A v_j \tag{41}$$

A similar procedure applies to β.The coefficient γ_j is used to scale w_j and v_j.

According to this method, vector v_{j+1} is orthonormal to all $w_1, w_2, w_3, \ldots, w_j$ and w_{j+1} is orthonormal to all v_1, v_2,v_3,..., v_k . If you are interested in the proof,[13] and [9] provide clear induction proofs.

4 The Road to Bi-CGSTAB

Now that we have a firm understanding of the background of the algorithm, we will try to understand how the Bi-CGSTAB came to be. The Bi-CGSTAB method is an improvement of the Conjugate Gradient Method. If you are unsure of how the Conjugate Gradient Method came to be, we recommend you read An Introduction to the Conjugate Gradient Method without the Agonizing Pain"[15]. This document is an extension of where that document left off. We'd like to note that a majority of this section is derived from Youssef Saad's book [14], and the textbook "Templates for the solution of linear systems: building blocks for iterative methods"[2].

First we shall break down why it needs to be a Bi-Conjugate Gradient Method, and finally, what makes it stabilized.

Let's check our knowledge before we start. We know that we can solve the problem $Ax = b$ using something called a Projection Method. Which defines that the solution $\hat{x}$ needs to be some linear combination of an initial guess x_0 and some change vector δ. We also need to find this $\hat{x}$ such that the vector inner product of $r_0 - A\delta$ and any vector w from the subspace $\mathcal{L}$ is equal to zero.

We have a subspace called the Krylov Subspace which is an abstract space that has dimensions of b, Ab, A^2b and so on, this is formally written in equation (33). By using Arnoldi's Method, we have also been able to show that every dimensions orthogonal to each other. Which means that the inner product of one dimension with another will result in 0.

How can we find the root of the equation $b - Ax = 0$ using this knowledge? One method that has been used is the method of steepest descent, or Gradient Descent[24].

The best way to think of this method, is by imagining a cone and a ball. Let's say that the cone is our function, and the ball is the root finding method. When using steepest descent, the ball would be bounded by two blinders. The algorithm would analyze the gradient and find the orientation of the steepest descent and would then allow the ball to roll down one step.

This is one way of getting to the answer, but we know that it is not the most efficient, it would be much easier if the ball was allowed to roll naturally into the bottom of the cone without having blinders that limited its movement.

In other words, there must be a way where the ball can take steps towards the root by using more than one dimension. That is where the Conjugate Gradient Method comes in.

4.1 The Conjugate Gradient Method

Let us alter equation (12) into smaller steps so that we can plug it into a computer.

$$\hat{x} = x_0 + \delta \tag{42}$$
$$x_{i+1} = x_i + \alpha_i p_i \tag{43}$$

We are breaking up the δ vector into small pieces, and iterating through them, p_i is a vector, and α is a scalar. As an extension, our residual can be defined as

$$r_{i+1} = r_i - \alpha A p_i \tag{44}$$

Using our knowledge from the Projection Method, we know that if we need to find a solution to the equation $b - Ax = 0$, then the inner product of $r_i - \alpha A p_i$ and r_i need to be zero.

$$(r_i - \alpha A p_i, r_i) = 0$$

or

$$r_{i+1} \perp r_i \tag{45}$$

If this constraint must hold, then

$$r_i^T r_i = 0 \tag{46}$$
$$r_i^T (r_i - \alpha_i A p_i) = 0 \tag{47}$$
$$\alpha = \frac{r_i^T r_i}{r_i^T A p_i} \tag{48}$$

We can further simplify this by using the knowledge that p_i is also a linear combination of the Krylov Subspace, because it's simply a linear combination of the residual vectors.

$$p_{i+1} = r_{i+1} + \beta_i p_i \tag{49}$$

Rearranging for r_{j+1}

$$r_{i+1} = p_{i+1} - \beta p_i \tag{50}$$

$$r_i = p_i - \beta p_{i-1} \tag{51}$$

Let us substitute this formula into the alpha formula

$$\alpha_i = \frac{r_i^T r_i}{(p_i - \beta p_{i-1})^T A p_i} \tag{52}$$

$$\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i - \beta p_{i-1}^T A p_i} \tag{53}$$

Ap_i and p_{i-1} are both linear combinations of the residual vector. Since the residual vectors are all in the Krylov Subspace, these vectors are orthogonal to each other. So finally

$$\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i} \tag{54}$$

If we want p_{i+1} and Ap_i to be orthogonal to each other, we would define it as:

$$\beta_i = -\frac{(r_{i+1}, Ap_i)}{(Ap_i, p_i)} \tag{55}$$

Rearranging equation (54) gets us

$$Ap_i = -\frac{1}{\alpha_i}(r_{i+1} - r_i) \tag{56}$$

Substituting this equation into the previous gives us

$$\beta_i = \frac{1}{\alpha_i} \frac{(r_{i+1}, (r_{i+1} - r_i))}{(p_i, Ap_i)} \tag{57}$$

This amazingly simplifies to

$$\beta = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \tag{58}$$

if you expand it out you will come to realize that $r_{i+1}^T r_i$ is zero due to their orthogonality, and that $\alpha_i p_i^T A p_i$ is essentially (r_i, r_i) from equation [52] So finally, the Conjugate Gradient algorithm is:

4.1.1 The Conjucate Gradient Algorithm

```
1: $r_0 := b - Ax_0$
2: $p_0 := r_0$
3: $k := 0$
4: **loop**
5:     $\alpha_k := \frac{r_k^T r_k}{\rho_k^T A \rho_k}$
6:     $x_{k+1} := x_k + \alpha_k \rho_k$
7:     $r_{k+1} := r_k - \alpha_k A \rho_k$
8:     if $r_{k+1}$ is sufficiently small then exit loop
9:     $\beta_k := \frac{r_k^T r_{k+1}}{r_k^T r_k}$
10:    $\rho_{k+1} := r_{k+1} + \beta_k \rho_k$
11:    $k := k + 1$
12: **end loop**
13: The result is $x_{k+1}$
```

[5] CG is the most popular iterative method for solving large systems of linear equations[15]. CG is effective for systems of the form $Ax = b$ where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, positive-definite matrix.

But one of its limitations is that the matrix A must be Symmetric Positive Definite. Or else the residual vectors cannot become orthogonal when taking small steps[3]. To include non symmetric matrices we turn to the Bi-Conjugate Gradient Method.

4.2 The Bi-Conjugate Gradient Method (Bi-CG)

The idea behind Bi-CG[11] is instead of defining the residual vector as being orthogonal to the subspace $\mathcal{L}$ it is possible to take it a step further, and create two subspaces. This is where the "Bi" comes from.

The reason we need to do that is because if A is not symmetric, then its transpose is not self-adjoint. This means that $A^* \neq A$. A^* is defined as the conjugate transpose, also known as the Hermitian Transpose[22]. The conjugate transpose means that one would take the transpose, and compute the conjugate complex of each entry. A matrix is defined as self adjoint if its conjugate transpose is the same as itself. Let us consider that we have a subspace $\mathcal{K}$ which is defined as:

$$\mathcal{K}_m = span\{v, Av, A^2v, \ldots, A^jv\} \tag{59}$$

Not much is different here, this is the same definition as the Conjugate Gradient, but what changes in Bi-Conjugate Gradient is that there is another subspace $\mathcal{L}$ which is projected orthogonally to the subspace $\mathcal{K}$. This subspace is defined as

$$\mathcal{L}_m = span\{w, A^Tw, (A^T)^2w, \ldots (A^T)^{m-1}w\} \tag{60}$$

This eliminates the problem that Conjugate Gradient had, one does not need a symmetric matrix, because we are projecting onto the conjugated transpose.

The Bi-CG method introduces two star vectors r^* and p^*. These star vectors are used to create the second subspace $\mathcal{L}_m$. They are calculated by

$$r_i^* = r_{(i-1)}^* - \alpha_i A^T p_i^* \tag{61}$$
$$p_i^* = r_{(i-1)}^* - \beta_i p_{(i-1)}^* \tag{62}$$

Then, the Bi-CG algorithm iterates through until it finds a vector such that

$$(r_i^*, r_j) = 0, i \neq j \tag{63}$$
$$(p_i^*, Ap_j) = 0, i \neq j \tag{64}$$

Instead of using Arnoldi's method to compute the orthogonalities, the Bi-CG method uses the Lanczos method, which is a more general type of Arnoldi's method to compute orthogonalities.

The general goal of the Bi-CG method is to orthogonalize the residual vector r, p onto the vector r^*, p^*. The vector r and p is in the $\mathcal{K}$ subspace, and the vector r^*andp^* is in the $\mathcal{L}$ subspace.

Using the constraints of equations (63,64), an algorithm can be developed to find the solution to a non symmetric matrix.

Biconjugate Gradient (BCG)

```
1: Compute $r_0 := b - Ax_0$. Choose $r_0^*$ such that $r_0, r_0^* \neq 0$
2: Set, $p_0 := r_0, p_0^* := r_0^*$
3: for $j = 0, 1, \ldots,$until convergence do
4:     $\alpha_j := \frac{(r_j, r_j^*)}{Ap_j, p_j^*}$
5:     $x_{j+1} := x_j + \alpha_j p_j$
6:     $r_{j+1} := r_j - \alpha_j Ap_j$
7:     $r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
8:     $\beta_j := \frac{(r_{j+1}, r_{j+1}^*)}{r_j, r_j^*}$
9:     $p_{j+1} := r_{j+1} + \beta_j p_j$
10:    $p_{j+1}^* := r_{j+1}^* - \beta_j p_j^*$
11: end for
```

The drawbacks to BiCG is that it is numerically unstable, other than a few papers, not much is known about this algorithm. Here's one textbook trying to explain the convergence of the Bi-CG method:

"*Few theoretical results are known about the convergence of BiCG…* "[2]

Not much has been explored about BiCG because, it is inferior to what GMRES could do and in some cases takes twice as long as GMRES. So before anything useful can be made of this algorithm, one must first prevent it from exploding. This is what Conjugate Gradient Squared attempts to solve.

4.3 Conjugate Gradient Squared

Let us go back to the Conjugate Gradient method, and analyze how the p and r vectors are computed. We are going to try to represent everything in terms of r_0. The initial conditions are:

$$r_0 = b - Ax_0 \tag{65}$$
$$p_0 = r_0 \tag{66}$$

When n = 1:

$$r_1 = r_0 - \alpha_0 A p_0 = r_0 - \alpha_0 A r_0 \tag{67}$$
$$p_1 = r_1 - \beta_0 p_0 = r_0 - \alpha_0 A r_0 - \beta_0 r_0 \tag{68}$$

When n = 2:

$$r_2 = [r_0 - \alpha_0 A r_0] - \alpha_1 A[r_0 - \alpha_0 A r_0 - \beta_0 r_0] \tag{69}$$
$$r_2 = [r_0 - \alpha_0 A r_0] - \alpha_1 A r_0 - \alpha_1 \alpha_0 A^2 r_0 - \alpha_1 A \beta_0 r_0] \tag{70}$$
$$p_2 = \{[r_0 - \alpha_0 A r_0] - \alpha_1 A[r_0 - \alpha_0 A r_0 - \beta_0 r_0]\} - \beta_1[r_0 - \alpha_0 A r_0 - \beta_0 r_0] \tag{71}$$

If we continued this exercise, it would become clear that the r and p vectors are a polynomial combination of A and r_0.

The way mathematicians express this relationship is:

$$r_j = \phi_j(A) r_0 \tag{72}$$
$$p_j = \pi_j(A) r_0 \tag{73}$$

Where ϕ_j is a polynomial with degree j. The basecase constraint of $\phi_0(A) = 1$ and $\pi_0(A) = 0$.

We could conduct the same exercise on the Bi-CG method, and two new facts would become clear:

$$r_j^* = \phi_j(A^T) r_0^* \tag{74}$$
$$p_j^* = \pi_j(A^T) r_0^* \tag{75}$$

Now if we plug these equations into how Bi-CG computes its α for instance we find that

$$\alpha_j = \frac{(\phi_j(A) r_0, \phi_j(A^T) r_0^*)}{(A\pi_j(A) r_0, \pi_j(A^T) r_0^*)} \tag{76}$$

We can simplify the numerator by:

$$(\phi_j(A) r_0, \phi_j(A^T) r_0^*) = (\phi_j(A) r_0)^T \phi_j(A^T) r_0^* \tag{77}$$
$$(\phi_j(A) r_0, \phi_j(A^T) r_0^*) = r_0^T \phi_j(A)^T \phi_j(A^T) r_0^* \tag{78}$$
$$(\phi_j(A) r_0, \phi_j(A^T) r_0^*) = r_0^T \phi_j^2(A^T) r_0^* \tag{79}$$
$$(\phi_j(A) r_0, \phi_j(A^T) r_0^*) = (\phi_j^2(A) r_0, r_0^*) \tag{80}$$

This same process can be done to the denominator, so α_j is transformed into

$$\alpha_j = \frac{(\phi_j^2(A)r_0, r_0^*)}{(A\pi_i^2(A)r_0, r_0^*)} \tag{81}$$

Now that we know that the residuals are polynomial functions of A and r_0 we can perform some interesting calculations. Let's express $\phi(A)$ and $\pi(A)$ as

$$\phi_{j+1}(A) = \phi_j(A) - \alpha_i A\pi_j(A) \tag{82}$$
$$\pi_{j+1}(A) = \phi_{j+1}(A) - \beta_j\pi_j(A) \tag{83}$$

This is basically representing p and r in terms of ϕ, π. Now let's square these terms. We get

$$\phi_{j+1}^2(A) = \phi_j^2(A) - 2\phi_j(A)\alpha_j A\pi_j(A) + \alpha_i^2 A^2\pi_j(A)^2 \tag{84}$$
$$\pi_{j+1}^2(A) = \phi_{j+1}^2(A) - 2\phi_{j+1}(A)\beta_j\pi_j(A) + \beta_j^2\pi_j^2(A) \tag{85}$$

The trouble is that there is a $\phi_{j+1}(A)\beta_i\pi_i(A)$. This makes it difficult to convert into an iterative method. What can instead be done is to try and define what this term is.

$$\phi_j(A)\pi_j(A) = \phi_j(A)(\phi_j(A) + \beta_{j-1}\pi_{j-1}(A)) \tag{86}$$
$$\phi_j(A)\pi_j(A) = \phi_j^2(A) + \beta_{j-1}\pi_{j-1}(A) \tag{87}$$

Now that all of these terms are defined in terms of each other, we can conclude that

$$r_j = \phi_j^2(A)r_0 \tag{88}$$
$$p_j = \pi_j^2(A)r_0 \tag{89}$$

and a new term

$$q_j = \phi_{j+1}(A)\pi_j(A)r_0 \tag{90}$$

A few more iterations of refinement which we will not go into here, will finally result in the algorithm.

Conjugate Gradient Squared

```
1: Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary.
2: Set, $p_0 := u_0 := r_0$
3: **for** $j = 0, 1, \ldots,$until convergence **do**
4:     $\alpha_j := \frac{(r_j, r_0^*)}{Ap_j, r_0^*}$
5:     $q_j := u_j - \alpha_j Ap_j$
6:     $x_{j+1} := x_j + \alpha_j(u_j + q_j)$
7:     $r_{j+1}^* := r_j - \alpha_j A(u_j + q_j)$
```

```
8:     $\beta_j := \frac{(r_{j+1}, r_0^*)}{r_j, r_0^*}$
9:     $u_{j+1} := r_{j+1} + \beta_j q_j$
10:    $p_{j+1} := u_{j+1} + \beta_j(q_j + \beta_j p_j)$
11: end for
```

One of the advantages of the Conjugate Gradient Squared Method[16] is that one does not need to transpose A at all. This is beneficial when dealing with very large matrices, or matrices that are not definite. It is known that CGS converges to a solution in half the time of Bi-CG, but it has drawbacks as well. CGS does not completely improve upon the stability of Bi-CG, since it is a squared function, the method often suffers from a build up of rounding errors and overflow errors.

5 The Bi-CGSTAB Method

In order to understand and fully appreciate the Bi-CGSTAB method, it was necessary to understand the mountain of knowledge that came before. Bi-CGSTAB was published around 8 years after CGS in 1992, and it is still regarded as a very good tool for solving very large sparse matrices.

The main logical leap in Bi-CGSTAB is defining the residual vectors as

$$r_i = \psi_i(A)\phi_i(A)r_0 \tag{91}$$

$\psi_i(A)$ is a recursive function which is defined as

$$\psi_{j+1}(A) = (I - \omega_j A)(\psi_j(A)) \tag{92}$$

The basecase for this example would be $\psi_0(A) = 1$ This is defined as a smoothing function, and attempts to reduce the wild jumps produced by the CGS method. The same smoothing function could be added to the residual vector

$$p_i = \psi_i(A)\pi_i(A)r_0 \tag{93}$$

Now we can define our r and p vectors as:

$$r_{i+1} = \psi_i(A)(r_i - \alpha_i A p_i) = (I - \omega_i A)(r_i - \alpha_i A p_i) \tag{94}$$

$$p_{i+1} = r_{i+1} + \beta_i(I - \omega_i A)p_j \tag{95}$$

The only thing remaining is to attempt to compute the greek letters ω, α and β.

5.1 Computing the Coefficients

The first problem that is clear is that it is not possible to compute the needed coefficients because Bi-CGSTAB is does not store much information. For example, since BI-CGSTAB never takes the transpose of the matrix A, it cannot compute β. Getting around this takes a little bit of ingenuity but it is possible.

5.1.1 Computing β

In Bi-CG the β constant was defined as

$$\beta_i = \frac{(r_{i+1}, r^*_{i+1})}{(r_i, r^*_i)} \tag{96}$$

Let us simplify this term by introducing a new constant to represent the numerator and denominator. We'll call it ρ.

$$\beta_i = \frac{\rho_{i+1}}{\rho_i} \tag{97}$$

To compute β we need to compute ρ. Let's create a prototype function $\tilde{\rho}$:

$$\tilde{\rho}_i = (\phi_i(A)r_0, \psi_i(A^T)r^*_0) \tag{98}$$

The reasoning behind this is that we are still trying to orthogonalize the A and the A^T subspaces, this doesn't tell much because it's still very abstract. Let's simplify further:

$$\tilde{\rho}_i = (\phi_i(A)r_0, \psi_i(A^T)r^*_0) \tag{99}$$
$$\tilde{\rho}_i = (\phi_i(A)r_0)^T \psi_i(A^T)r^*_0 \tag{100}$$
$$\tilde{\rho}_i = r_0^T \phi_i(A)^T \psi_i(A^T)r^*_0 \tag{101}$$
$$\tilde{\rho}_i = (\psi_i(A)\phi_i(A)r_0, r^*_0) \tag{102}$$

Substituting equation (91) we get:

$$\tilde{\rho} = (r_i, r^*_0) \tag{103}$$

This can be taken one step further, let's expand the definition of $\psi_2(A^T)r_0*$

$$\tilde{\rho}_2 = (\phi_2(A)r_0, \psi_2(A^T)r^*_0) \tag{104}$$
$$\tilde{\rho}_2 = (\phi_2(A)r_0, \omega_1\omega_0(A^T)^2r^*_0 - \omega_1A^Tr^*_0 - \omega_0A^Tr^*_0 - r^*_0) \tag{105}$$

Let's clean this up and represent all of the ωs as a constant η

$$\tilde{\rho}_2 = (\phi_2(A)r_0, \eta_1(A^T)^2 - \eta_1A^Tr^*_0 - \eta_0A^Tr^*_0 - r^*_0) \tag{106}$$

We know that $\phi_j(A)$ has its own coefficients, which we'll call γ_i

$$\tilde{\rho}_2 = (\gamma_2\phi_2(A)r_0, \eta_1(A^T)^2 - \eta_1A^Tr^*_0 - \eta_0A^Tr^*_0 - r^*_0) \tag{107}$$

Thinking in terms of Krylov spaces, there is a method to simplify this term. We know that every term other than the leading term is orthogonal to each other. This is because we made them orthogonal using Lanczos's Method. So all of the terms except the leading ones cancel out. This leaves us with:

$$\tilde{\rho}_2 = (\phi_2(A)r_0, \frac{\eta_{1,2}}{\gamma_{1,2}}(A^T)^2 r_0^*) \tag{108}$$

This means that $\tilde{\rho}_2$ is the orthogonalization of $\phi_2(A)r_0$, and A^T and $\frac{\eta_1}{\gamma_1}$ of $\phi_2(A)$. So in general:

$$\tilde{\rho}_i = (\phi_i(A)r_0, \frac{\eta_{1,i}}{\gamma_{1,i}}\phi_i(A^T)r_0^*) \tag{109}$$

Now that the prototype $\tilde{\rho}$ is defined in such a way, we can observe that back in Bi-CG the ρ function was defined as:

$$\rho = (\phi_i(A)r_0, \phi_i(A^T)r_0^*) \tag{110}$$

Or in other words,

$$\tilde{\rho}_i = \frac{\eta_{1,i}}{\gamma_{1,i}}\rho_i \tag{111}$$

The next step would be to simplify the η and γ. Since both of those constants are all multiples of the previous η and γ

$$\eta_j = \omega_j\omega_{j-1}\ldots\omega_1\omega_0 \tag{112}$$
$$\gamma_j = \alpha_j\alpha_{j-1}\ldots\alpha_1\alpha_0 \tag{113}$$

we can write these constants as

$$\eta_{1,(i+1)} = -\omega_i\eta_{1,i}, \text{ and } \gamma_{1,(i+1)} = -\gamma_j\eta_{1,i} \tag{114}$$

As a result

$$\beta_i = \frac{\tilde{\rho}_{i+1}}{\tilde{\rho}_i} = \frac{\omega_i\rho_{i+1}}{\alpha_i\rho i} = \frac{\alpha_i\rho_{i+1}}{\omega_i\rho_i} \tag{115}$$

5.1.2 Computing α

To compute α we perform similar calculations, and use similar ideas. From Bi-CG we know that α_i was

$$\alpha_i = \frac{(\phi_i(A)r_0, \phi_i(A^T)r_0^*)}{(A\pi_i(A)r_0, \pi_i(A^T)r_0^*)} \tag{116}$$

Transforming this to Bi-CGSTAB gives us

$$\alpha_i = \frac{(\phi_i(A)r_0, \psi_i(A^T)r_0^*)}{(A\pi_i(A)r_0, \psi_i(A^T)r_0^*)} \tag{117}$$

Note that we were able to replace both $\psi(A^T)$ and $\pi(A^T)$ with $\psi(A^T)$ because the leading polynomials are both the same for those variables. Finally simplifying these results gives us

$$\alpha_i = \frac{(\psi_i(A)\phi_i(A)r_0, r_0^*)}{(\psi_i(A)A\pi_i(A)r_0, r_0^*)} \tag{118}$$

Substituting for $\tilde{\rho}$ and p_j gives us

$$\alpha_i = \frac{\tilde{\rho}_i}{Ap_i, r_0^*} \tag{119}$$

$$\alpha_i = \frac{\tilde{\rho}_i}{(Ap_i, r_0^*)} \tag{120}$$

5.1.3 Calculating ω

The power from Bi-CGSTAB comes from the ω, as this is the smoothing function. The function of the ω is to reduce the 2-norm of the residual vector to increase convergence. Let's try to find an ω that minimizes the residual error.

$$r_{i+1} = (I - \omega_i A)r_i \tag{121}$$

We can introduce another variable that helps compute the ω, let's call it s_i. s_i is equivalent to r_i.

$$r_{i+1} = (I - \omega_i A)s_i \tag{122}$$

The method of minimizing the 2-norm of the residual is by ensuring that the inner product As_i and $(I - \omega_i A)s_i)$ is equal to zero. Formalized as

$$((I - As_i)s_i, As_i) = 0 \tag{123}$$

Once again, we are doing this in order to orthogonalize the $(I - As_i)s_i$ and the As_i subspaces. This is computed to

$$(As_i)^T(s_i - \omega_j As_i) = 0 \tag{124}$$

$$(As_i)^T s_i - (As)^T \omega_j As_i = 0 \tag{125}$$

$$\omega_i = \frac{(As_i, s_i)}{(As_i, As_i)} \tag{126}$$

We must finally rewrite r_{j+1} in terms of α and ω so that it can update the approximate solution to the problem.

$$r_{i+1} = s_i - \omega_j As_i \tag{127}$$

$$r_{i+1} = r_i - \alpha_i Ap_i - \omega_i As_i \tag{128}$$

We use this residual to compute

$$x_{i+1} = x_i + \alpha_i p_i + \omega_i s_i \tag{129}$$

Here is the final algorithm for Bi-CGSTAB[18].

BICGSTAB Transpose-Free Variants

```
1: Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary.
2: Set, $p_0 := u_0 := r_0$
3: for $j = 0, 1, \ldots$,until convergence do
4:     $\alpha_j := \frac{(r_j, r_0^*)}{Ap_j, r_0^*}$
5:     $s_j := r_j - \alpha_j A p_j$
6:     $\omega_{j+1} := \frac{As_j, s_j}{As_j, As_j}$
7:     $x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
8:     $r_{j+1} := s_j = \omega_j A s_j$
9:     $\beta_j := \frac{r_{j+1}, r_0^*}{r_j, r_0^*} \times \frac{\alpha_j}{\omega_j}$
10:    $p_{j+1} := r_{j+1} + \beta_j (p_j - \omega_j A p_j)$
11: end for
```

6 Preconditioning

Preconditioning is a key factor in solving iterative methods. The underlying idea of using preconditioners is to make solvers converge faster using less iterations. The convergence of the iterative method depends on the spectrum of the coefficient matrix and can be significantly improved using preconditioning. The preconditioning modifies the spectrum of the coefficient matrix in order to reduce the number of iterative steps required for convergence.

- The preconditioned system should be easy to solve.
- The preconditioner should be cheap to construct and apply.[17]

In general, a good preconditioner should meet the following requirements:

The first property means that the preconditioned iteration should converge rapidly, while the second ensures that each iteration is not too expensive. Notice that these two requirements are in competition with each other. It is necessary to keep a balance between the two requirements. In other words, to get to a preconditioned system one needs to use a suitable preconditioning technique, considering the trade-off between the two properties.

Considering the system $Ax = b$, the preconditioned system using a nonsingular matrix M will be:

$$M^{-1}Ax = M^{-1}b \tag{130}$$

or in other words

$$AM^{-1}y = b \tag{131}$$

$$x = M^{-1}y \tag{132}$$

Where the first system is left-preconditioned and the second one is right-preconditioned and matrix M approximates the coefficient matrix A. The preconditioned system has the same solution, but is easier to solve. In addition, we can apply split preconditioning as well:

$$(M_1)^{-1}A(M_2)^{-1}y = (M_1)^{-1}b \tag{133}$$

$$x = M^{-1}y \tag{134}$$

Where preconditioner M is splitted to M_1 and M_2, $M = M_1M_2$

The rate of convergence for a linear system $Ax = b$ solved in the Krylov subspace depends on the condition number of matrix A. The general idea is to compute a matrix M in such a way that it is invertible and close to A so $M^{-1}A$ is closer to the identity matrix than it is to A, thus it will have a smaller condition number and the preconditioned system will be solved faster.

Which preconditioner is more suitable depends on the problem characteristics, i.e. the characteristic of matrix A, and the solver that we use. For example, some preconditioners are beneficial only while being applied to Symmetric Positive Definite matrices such as preconditioned conjugate gradient and Cholesky factorization[19]. The matrix that we consider is non SPD, large, and sparse. Notice that even if we have a sparse matrix A, and even if M is sparse as well, there is no reason for M^{-1} to be a sparse matrix due to the fill-in steps that may occur during preconditioning and also since the inverse of a sparse matrix is not necessarily sparse. Hence, not all the techniques can be applied to solve the preconditioned system. For example, if we have a large sparse matrix, it may be advantageous to split A to two other matrices M and N such that $A = M - N$ for a given vector v and then compute $\omega = M^{-1}Av$ as $\omega = (I - M^{-1}N)v$ by the following procedure:

$$r = Nv \tag{135}$$

$$\omega = M^{-1}r \tag{136}$$

$$\omega = v - \omega \tag{137}$$

The matrix N may be sparser than A and r may take less time to compute in comparison to doing Av product.

6.1 LU Factorization

Every square matrix A can be decomposed into a product of a lower triangular matrix L and a upper triangular matrix U. For a system $Ax = b$, this technique splits matrix A to two matrices L and U such that

$$Ly = b \tag{138}$$

$$Ux = y \tag{139}$$

Notice that these computation can be done quickly as the matrices are triangular. To have a better understanding, let us look at the following example.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{11}l_{21} & u_{12}l_{21} + u_{22} & u_{13}l_{21} + u_{23} \\ u_{11}l_{31} & u_{12}l_{31} + u_{22}l_{32} & u_{13}l_{31} + u_{23}l_{32} + u_{33} \end{bmatrix} = LU$$

usually the diagonal elements of L are set to 1, $L_{11} = L_{22} = L_{33} = 1$. Solving for the other l and u, we get the following equations:

$$u_{11} = a_{11} \tag{140}$$

$$u_{12} = a_{12} \tag{141}$$

$$u_{13} = a_{13} \tag{142}$$

$$u_{22} = a_{22}u_{12}l_{21} \tag{143}$$

$$u_{23} = a_{23}u_{13}l_{21} \tag{144}$$

$$u_{33} = a_{33}(u_{13}l_{31} + u_{23}l_{32}) \tag{145}$$

And for l we have:

$$L_{21} = \frac{a_{21}}{u_{11}} \tag{146}$$

$$L_{31} = \frac{a_{31}}{u_{11}} \tag{147}$$

$$L_{32} = \frac{(a_{32} - u_{12}l_{31})}{u_{22}} \tag{148}$$

As can you see, there is a calculation pattern for calculating u_{ij} and l_{ij} which can be expressed in the following formulas. For U

$$u_{ij} = a_{ij} - \sum_{k=1}^{(j-1)} u_{kj}l_{ik} \tag{149}$$

and for L

$$l_{ij} = \frac{1}{u_j j}(a_{ij} - \sum_{k=1}^{(j-1)} u_{kj}l_{ik}) \tag{150}$$

One problem is that $u_{jj} = 0$. To avoid this problem, there is a method called pivoting, which rearranges the rows of A prior to LU decomposition in a way that the largest element of each column gets onto the diagonal of A. [21] has a good explanation for pivoting if you are interested.

6.2 Incomplete LU Factorization

When a sparse matrix is factored by LU or Gaussian elimination[20], fill-in usually takes place. This means that the factors L and U of the coefficient matrix A are considerably less sparse than A. Even though sparsity-preserving pivoting techniques are used to reduce fill-in, sparse direct methods are not considered suitable for solving very large linear systems due to time and space constraints. However, by discarding part of the fill-in in the course of the factorization process, simple but powerful preconditioners can be obtained using ILU factorization. An incomplete LU factorization of a matrix is a sparse approximation of the LU factorization often used as a preconditioner. It instead tries to find L and U matrices such that $A \approx LU$. It is important to know how to choose patterns of zero and nonzero elements for both L and U and how to compute the values of nonzero elements.

In general, ILU works better for sparse matrices than LU. The reason is that the LU factors can be much less sparse than the original matrix, which would result in a higher computation cost. For this reason, in our first attempt we used ILU as a preconditioner.

There exist some improved versions of ILU factorizations of which we give an explanation for ILU no-fill-in (abbreviated to ILU(0)) and ILU crout(abbreviated to ILUC). The ILU(0) factorization is the simplest form of the ILU preconditioners. ILU(0) has the same zero pattern as the original matrix A. 0 means that this method has 0 levels of fill-in. In ILU(0) the computations are conducted as in the traditional LU factorization, but any new nonzero element $u_{i,j}$ and $l_{i,j}$ created in the process is dropped if it appears in the place of a zero element in the original matrix . Hence, the factors together have the same number of nonzeros as the original matrix.

As a result, the most important problem of the factorization of sparse matrices, the fill-in, is eliminated. This method is not as accurate as ILU, but it uses less memory that makes this trade-off a good one. To do so, we need to modify the LU factorization method in such a way that only nonzero elements of L and U are computed and other elements are ignored. The algorithm for ILU(0) is:

Incomplete LU Factorization KIJ-variants

```
1: for $k = 1, 2, \ldots, n-1$ do
2:   for $i = k+1 \ldots n$ and $(i,k) \in NZ(A)$ do
3:     $a_{ik} = a_{ik}/a_{kk}$
4:     for $j = k+1 \ldots n$ and $(i,j) \in NZ(A)$ do
5:       $a_{ij} = a_{ij} - a_{ik}a_{kj}$
6:     end for
7:   end for
8: end for
```

This is the same as the LU algorithm except we only choose elements of U and L from $NZ(A)$ which is the set of nonzero elements of the original matrix. A drawback of the ILU(0) strategy is that it ignores the actual size of the entries and the exact factors L and U.

ILU crout attempts to reduce the number of nonzero elements in the preconditioner and the computation time during the iterations. To explain how ILUC works first we need to understand LU crout (abbreviated to LUC) factorization. The procedure is as follows:

Crout LU Factorization

```
1: for $k = 1, 2, \dots, n$ do
2:     for $i = 1 \dots k-1$ and if $a_{ki} \neq 0$ do
3:         $a_{k,k:n} = a_{k,k:n} - a_{ki} * a_{i,k:n}$
4:     end for
5:     for $i = 1 \dots k-1$ and if $a_{ki} \neq 0$ do
6:         $a_{k+1:n,k} = a_{k+1:n,k} - a_{ik} * a_{k+1:n,i}$
7:     end for
8:     $a_{ik} = a_{ik}/a_{kk}$ for $i = k+1, \dots, n$
9: end for
```

At step k the entries $a_{k+1:n,k}$ in the unit lower triangular factor L and $a_{k,k:n}$ in the upper triangular factor U is computed. Then all the updates of the previous steps are applied to the entries $a_{k+1:n,k}$ and $ak, k : n$. What it means is that we are storing L by columns and U by rows, which means A is being stored such that its lower triangular part is stored by columns and its upper triangular part is stored by rows. Now, in order to explain the the procedure for ILU(C), we first need to have some knowledge about dual dropping strategy. We use this strategy because we want our preconditioner matrix to be as sparse as possible for we are going to do experiments on large sparse matrices. This strategy is important for the ILU factorization to generate an efficient and reliable preconditioned matrix. The dual dropping strategy makes it possible to determine the sparsity of incomplete factorization preconditioners by two fill-in control parameters: t : dropping tolerance p : the number of p largest nonzero elements in the magnitude In other words: 1. Any elements of L or U whose magnitude is less than tolerance t is dropped. 2. In the k-th column of L, the number of the p largest nonzero elements in the magnitude are kept. Similarly, the number of the p largest nonzero elements in the k-th row of U, which includes the diagonal elements, are kept. This controls the total memory storage that can be used by the preconditioner. Read [25] If you are interested to see how we can determine p. The purpose of t is to keep only the large entries of L and U. Dropping small entries allows one to save computational time. The purpose of the parameter p is to keep the memory requirement under control. Finally, extending the algorithm for LUC and using the dropping strategy, the ILUC procedure is as follows:

Crout Version of the ILU Factorization

```
1: for $k = 1, 2, \dots, n$ do
2:     Initialize row $z$ : $z_{1:k-1} = 0, z_{k,k:n} = a_{k,k:n}$
3:     for $\{i | 1 \leq i \leq k-1$ and $l_{ki} \neq 0\}$ do
4:         $z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
5:     end for
```

```
 6:     Initialize column $\omega : \omega_{1:k} = 0, \omega_{k+1:n} = a_{k+1:n,k}$
 7:     for $\{i | 1 \leq i \leq k-1$ and $l_{ki} \neq 0\}$ do
 8:         $\omega_{k+1:n} = z_{k+1:n} - u_{ik} * l_{k+1:n,i}$
 9:     end for
10:     Apply a dropping rule to row $z$
11:     Apply a dropping rule to column $\omega$
12:     $u_{k,:} = z$
13:     $l_{:,k} = \omega / u_{kk}, l_{kk} = 1$
14: end for
```

The ILU(C) algorithm has the same steps as LUC with the addition of steps 10 and 11 where it uses the dropping strategy.

For matrices which are diagonally dominant, a very effective way to obtain a proper preconditioner is to proceed with an LU-decomposition, but to preserve the sparsity in the factors L and U by ignoring some or all elements causing ll-in additional to that of A[16]. Hence, we chose ILU as our preconditioner. Also some common preconditioners like jacobian preconditioner are not good choices for large sparse matrices as they need the same storage as the original matrix. In order to use preconditioners in Bi-CGSTAB, we must modify the Bi-CGSTAB Algorithm to accommodate for M_0 and M_1, remember that $M = M_1 M_2$

Preconditioned Bi-CGSTAB Algorithm

```
 1: $r_0 = b - Ax_0$
 2: Choose an arbitrary vector $\hat{r}_0$ such that $(\hat{r}_0, r_0) \neq 0$, e.g.,$\hat{r}_0 = r_0$
 3: $\rho_0 = a = \omega_0 = 1$
 4: $v_0 = p_0 = 0$
 5: for $i = 1, 2, 3, \ldots$ do
 6:     $\rho_i = (\hat{r}_0, r_{i-1})$
 7:     $\beta = (\rho_i / \rho_{i-1})(a / \omega_{i-1})$
 8:     $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
 9:     $y = M^{-1} p_i$
10:     $v_i = Ay$
11:     $a = \rho_i / (\hat{r}_0, v_i)$
12:     $s = r_{i-1} - a v_i$
13:     If $||s||$ sufficiently small, then set $x_i = x_{i-1} + a p_i$ and quit
14:     $z = M^{-1} s$
15:     $t = Az$
16:     $\omega_i = (M_1^{-1} t, M_1^{-1} s) / (M_1^{-1} t, M_1^{-1} t, )$
17:     $x_i = x_{i-1} + ay + \omega_i z$
18:     If $x_i$ is accurate enough then quit
19:     $r_i = s - \omega_i t$
20: end for
```

7 Numerical Experiments

" ...if it disagrees with experiment, then it's wrong. In that simple statement is the key to science. "

- Richard Feynman (1918 - 1988)[6]

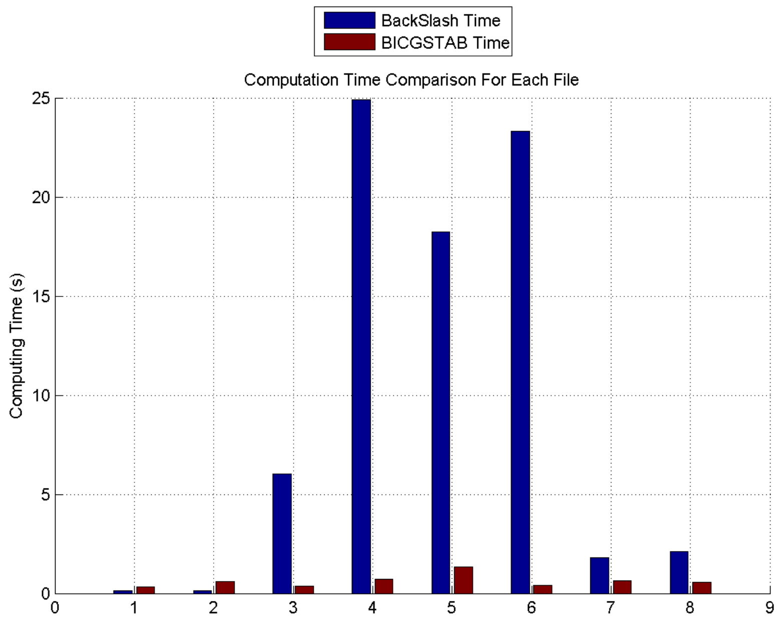

Figure 2: Computing Time for Each File

This is a timing graph comparing the backslash operation and BICGSTAB. The blue bar on the left is the backslash time and the red bar on the right is the BICGSTAB time which includes the time to compute the ILU preconditioners. As we can see in (Figure 2), the M1 and M2 backslash time is approximately 1/2 of BICGSTAB's with the ILU preconditioner. These are the matrices provided by Dr. Ned Nedialkov. It was impossible to find a precounditioner that could outrun the Backslash operator. We discovered this when we gave the BI-CGSTAB an LU Matrix, which is one of the best preconditioners. It's so good, that BI-CGSTAB solved the problem in one iteration. Even though we used one of the best preconditioners available, one iteration of Bi-CGSTAB took longer than the entirety of the backslash operator. This is just the time that Bi-CGSTAB takes to converge to a solution, the LU Decomposition time was not included. Therefore there can be no preconditioner to speed up the convergence rate such that the computation time is faster than the backslash operator for those two matrix. This shows that Bi-CGSTAB is very inefficient for the matrices given by Dr. Nedialkov. However we can see from the graph, that

Table 1: Running Time Comparison For Each Matrix

	ID	BackSlash_Time	BICGSTAB_Time
M1.mat	1	1.4063E-01	3.2813E-01
M2.mat	2	1.4063E-01	5.9375E-01
ASIC_320ks.mat	3	6.0313E+00	3.5938E-01
ecl32.mat	4	2.4922E+01	7.0313E-01
ibm_matrix_2.mat	5	1.8219E+01	1.3438E+00
offshore.mat	6	2.3313E+01	4.0625E-01
venkat01.mat	7	1.7969E+00	6.2500E-01
wang3.mat	8	2.0938E+00	5.7813E-01

Table 2: Characteristic of Matrices Being Tested

Matrix	ID	M	N	NoneZeros	Sparse_Density
M1.mat	1	1.9880E+03	1.9880E+03	1.1515E+05	0.029136
M2.mat	2	4.3390E+03	4.3390E+03	8.0402E+04	0.004271
ASIC_320ks.mat	3	3.2167E+05	3.2167E+05	1.3161E+06	0.000013
ecl32.mat	4	5.1993E+04	5.1993E+04	3.8042E+05	0.000141
ibm_matrix_2.mat	5	5.1448E+04	5.1448E+04	5.3704E+05	0.000203
offshore.mat	6	2.5979E+05	2.5979E+05	4.2427E+06	0.000063
venkat01.mat	7	6.2424E+04	6.2424E+04	1.7178E+06	0.000441
wang3.mat	8	2.6064E+04	2.6064E+04	1.7717E+05	0.000261

the backslash operator skyrockets on the following matrices: `ASIC_320ks.mat, ecl32.mat, ibm_matrix_2.mat, offshore.mat, venkat01.mat, wang3.mat`.

From Table 2, it is not hard to spot M1 and M2's sparse density is much higher than the other matrices, and this is the reason why the backslash operator is better than than BICGSTAB. BICGSTAB is designed for very large sparse matrices. All 6 matrices we chose are nonsymmetric and real diagonal matrices (please refer to Figure 7 to Figure 12), These are ideal for ILU decomposition. ILU was chosen because it provided the highest speed up compared to the compute time. We can see the backslash operation is unstable yet BICGSTAB still can finish the task in a relatively consistent speed. However the backslash operation has a much better accuracy comparing to BICGSTAB as shown in Figure 3 and Table 3.

The reason behind this is that BICGSTAB exits when the error is below the given threshold of 1×10^{-8}. Matlab's backslash operator has a different threshold that it adheres to. Overall BICGSTAB is still a much better algorithm to use on those specific type of matrices.

Figures 5 through 12 were generated using the `spy` command on MATLAB. The `spy` command shows the density of nonzero across a matrix. Figure 5 and Figure 6 show the matrices provided by Dr. Nedialkov, they are the smallest and most dense of the matrices.

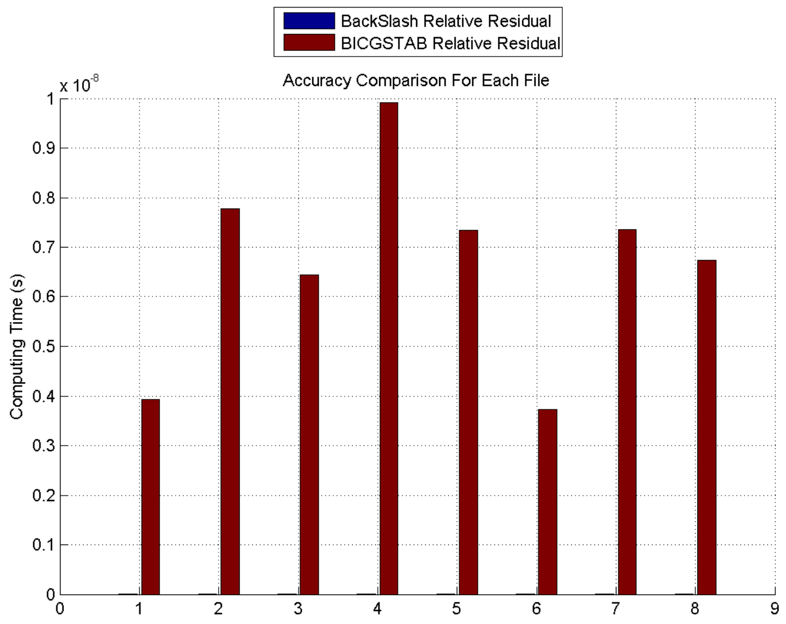

Figure 3: Computing Accuracy for Each File

Table 3: Relative Residual Comparison For Each Matrix

Matrix	ID	BackSlash_Relative_Residual	BICGSTAB_Realtive_Residual
M1.mat	1	1.4915E-14	3.9336E-09
M2.mat	2	8.4273E-12	7.7747E-09
ASIC_320ks.mat	3	4.2495E-12	6.4470E-09
ecl32.mat	4	6.6974E-14	9.9204E-09
ibm_matrix_2.mat	5	3.4360E-14	7.3403E-09
offshore.mat	6	8.4390E-14	3.7312E-09
venkat01.mat	7	5.6605E-16	7.3571E-09
wang3.mat	8	7.4037E-16	6.7403E-09

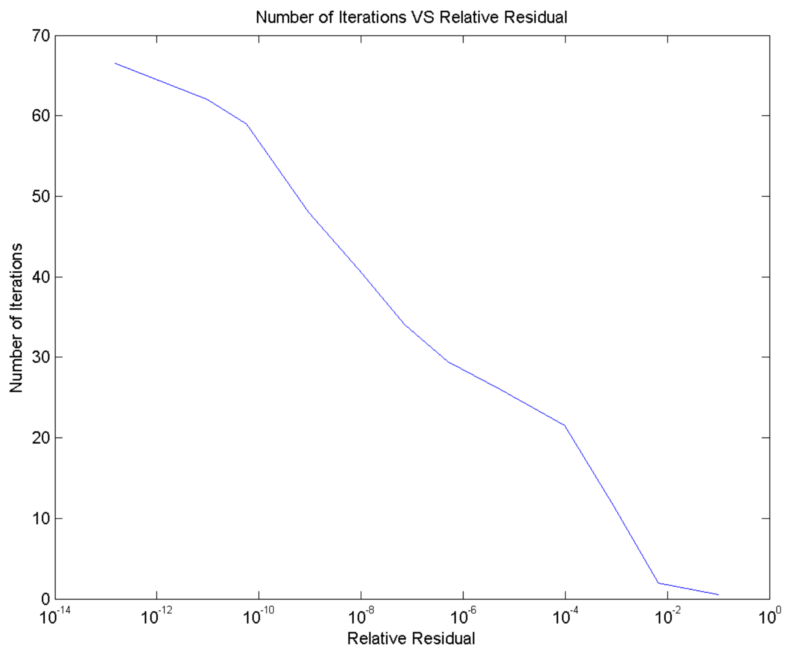

Figure 4: Number of Iterations VS Relative Residual
Figure 4 shows that the $log($`relative residual`$) \propto$ `Number of Iterations` linearly.

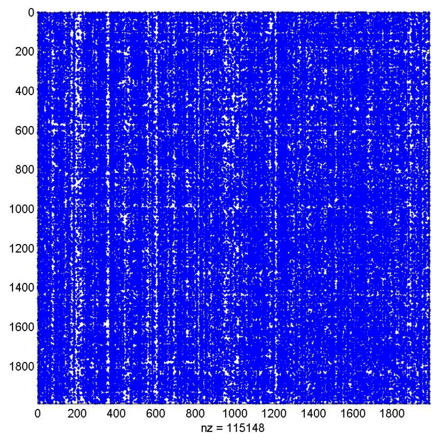

Figure 5: `M1` Matrix Probe Result

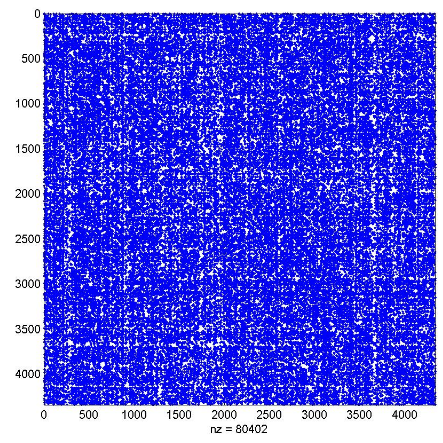

Figure 6: `M2` Matrix Probe Result

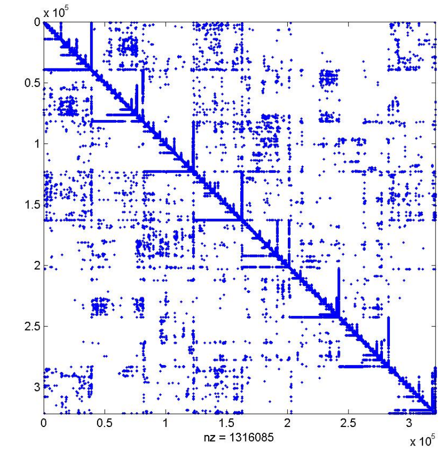

Figure 7: `ASIC_320ks` Matrix Probe Result

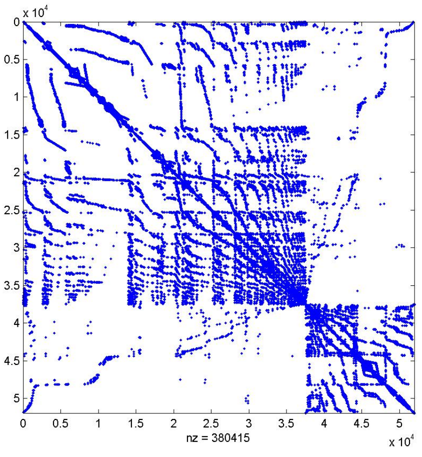

Figure 8: `ecl32` Matrix Probe Result

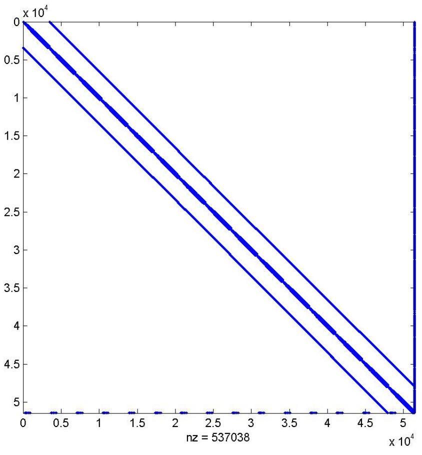

Figure 9: `ibm_matrix_2` Matrix Probe Result

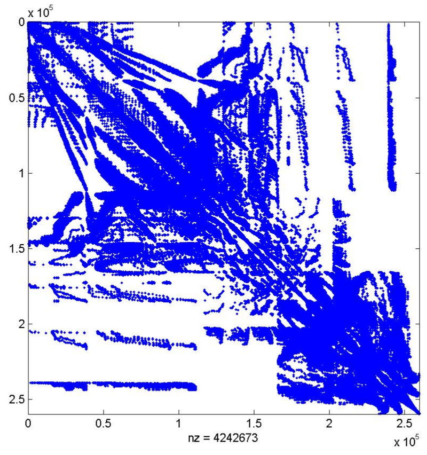

Figure 10: `offshore` Matrix Probe Result

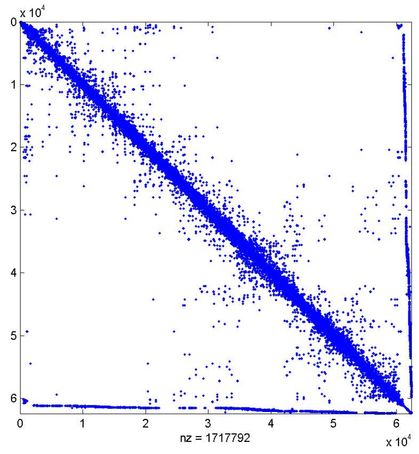

Figure 11: `venkat01` Matrix Probe Result

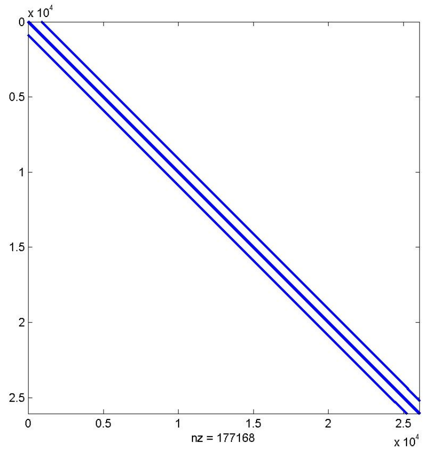

Figure 12: `wang3` Matrix Probe Result

8 Conclusion

In this document we've shown how the BI-CGSTAB algorithm came to be, and proved its effectiveness. We hope that you found this resource helpful. The Bi-CGSTAB is but one of the many sparse solvers out there. Sparse solving is sometimes more an art than a science. Depending on the matrix, one method may be better than another. One pre conditioner may be better than another. It all depends on the type of matrix. Knowledge of how these methods work will provide you with a better idea of which system to use. If you are interested in learning more about sparse solving, we recommend you research GMRES, and TFQRM.

References

[1] U. M. Ascher and C. Greif, *A First Course on Numerical Methods*, vol. 7, Siam, 2011.

[2] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*, vol. 43, Siam, 1994.

[3] N. Black and S. Moore, *Biconjugate gradient method.* `http://mathworld.wolfram.com/BiconjugateGradientMethod.html`.

[4] H. M. Colleg, *The gram-schmidt algorithm.* `https://www.math.hmc.edu/calculus/tutorials/gramschmidt/gramschmidt.pdf`.

[5] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical analysis, Springer, 1976, pp. 73–89.

[6] R. Gower, *The key to science.* `https://www.youtube.com/watch?v=tD_XAX--Ono`.

[7] M. Heath, *Computing: An Introductory Survey*, McGraw-Hill, 1998.

[8] P. F. Heiter, *Stable implementation of three-term recurrence relations*, Master's thesis, Universitat Ulm Fakultat fur Mathematik und Wirtschaftswissenschaften, 2010.

[9] V. John, *Other krylov subspace methods for non-symmetric for non-symetric systems.* `https://www.wias-berlin.de/people/john/LEHRE/NUMERIK_II/linsys_8.pdf`.

[10] Katja, *Transpose and inverse.* `http://katjaas.nl/transpose/transpose.html`.

[11] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, (1950).

[12] C. Lee, *Eigenvalues and eigenvectors.* `http://www.ms.uky.edu/~lee/amspekulin/eigenvectors.pdf`.

[13] D. P. OLeary, *Bi-orthogonalization methods.* `https://www.cs.umd.edu/users/oleary/a600/600sparseitbiorthhand.pdf`.

[14] Y. Saad, *Iterative methods for sparse linear systems*, Siam, 2003.

[15] J. R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, (1994).

[16] P. Sonneveld, *Cgs, a fast lanczos-type solver for nonsymmetric linear systems*, SIAM journal on scientific and statistical computing, 10 (1989), pp. 36 – 52.

[17] A. van der Ploeg, *Preconditioning for sparse matrices with applications*, PhD thesis, Rijksuniversiteit Groningen, feb 1994.

[18] H. A. Van der Vorst, *Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems*, SIAM Journal on scientific and Statistical Computing, 13 (1992), pp. 631–644.

[19] L. Vandenberghe, *Cholesky factorization.* `http://www.seas.ucla.edu/~vandenbe/103/lectures/chol.pdf`.

[20] ——, *Lu factorization.* `https://www.seas.ucla.edu/~vandenbe/103/lectures/lu.pdf`.

[21] Vismor, *Pivoting to preserve sparsity.* `https://vismor.com/documents/network_analysis/matrix_algorithms/S8.SS3.php`.

[22] E. W. Weisstein, *Conjugate transpose.* `http://mathworld.wolfram.com/ConjugateTranspose.html`.

[23] ——, *Kronecker delta.* `http://mathworld.wolfram.com/KroneckerDelta.html`.

[24] Wikipedia, *Gradient descent.* `https://en.wikipedia.org/wiki/Gradient_descent`.

[25] Y. Zhang, T.-Z. Huang, Y.-F. Jing, and L. Li, *Flexible incomplete cholesky factorization with multi-parameters to control the number of nonzero elements in preconditioners*, Numerical Linear Algebra with Applications, 19 (2012), pp. 555–569.

A Matlab Code

matlab/project/main.m

```
%------------------------------------------------------------------------
%                          Information here
%------------------------------------------------------------------------
close all;
if(1) %debug
clear all;
%Number of tries
warning('off','all')
%Tolerance
tol = 1e-8
%Filenames
filename = {'M1.mat';
     'M2.mat';
     'ASIC_320ks.mat';
     'ecl32.mat';
     'ibm_matrix_2.mat';
     'offshore.mat';
     'venkat01.mat';
     'wang3.mat'};

[row,col] = size(filename);

disp('------------------------------begin
   ------------------------------');

tbs = zeros(row,1);
tilu = zeros(row,1);
rrbs = zeros(row,1);
rrilu = zeros(row,1);

%Generating timing diagram
for i = 1:row
clear M1 M2 xbicgsta flag relres iter;
%Read files
load(char(filename(i)));
disp(sprintf('%s', char(filename(i))));
A = Problem.A;
[m(i),n(i)] = size(A);
disp(sprintf('Size: %d x %d',m(i),n(i)));
```

```
nz(i) = max(size(nonzeros(A)));
disp(sprintf('None zeros: %d', nz(i)));

%Generate x and b
x = rand(m(i),1);
b = A*x;

%backslash operation
tstart = cputime;
xb = A\b;
tbs(i) = cputime - tstart;

%calculate relative residual
rrbs(i) = norm(x - xb) / norm(x);
disp(sprintf('Backslash: Time takes to calculate: %g seconds', tbs
   (i)));

% ILU nofill
setup.type = 'nofill';
setup.milu = 'off';
setup.droptol = 0.005;
tstart = cputime;
[M1,M2] = ilu(A,setup);
[xbicgstab,flag,relres,iter] = bicgstab(A,b,tol,9999,M1,M2);
tilu(i) = cputime - tstart;
rrilu(i) = relres;
disp(sprintf('ILU preconditioned BICGSTAB: Time takes to calculate
   : %g seconds', tilu(i)));

% Just output something to ensure it fits
switch (flag)
    case 0
        disp ('bicgstab converged to the desired tolerance tol 
           within maxit iterations.');
    case 1
        disp ('bicgstab iterated maxit times but did not converge.
           ');
    case 2
        disp ('Preconditioner M was ill-conditioned.');
    case 3
        disp ('bicgstab stagnated. (Two consecutive iterates were 
           the same.)');
    case 4
        disp ('One␣of␣the␣scalar␣quantities␣calculated␣during␣
            bicgstab␣became␣too␣small␣or␣too␣large␣to␣continue␣
            computing.');
end

disp('␣');
end

%Generate a table
T = table(transpose([1:row]),transpose(m),transpose(n),transpose(
    nz),tbs,tilu,rrbs,rrilu,...
      'RowNames',filename,...
      'VariableNames',{'ID' 'M' 'N' 'NoneZeros' 'BackSlash_Time' '
         BICGSTAB_Time' 'BackSlash_Relative_Residual' '
         BICGSTAB_Realtive_Residual'})

%Generate graphs
fig1 = figure;
title('Computation␣Time␣Comparison␣For␣Each␣File');
grid on
hold on
b = bar([tbs,tilu]);
legend(b,{'BackSlash␣Time'; 'BICGSTAB␣Time'},'Location','
   northoutside');
ylabel('Computing␣Time␣(s)');
hold off
saveas(fig1,'graph1.png');


fig2 = figure;
title('Accuracy␣Comparison␣For␣Each␣File');
grid on
hold on
b = bar([rrbs,rrilu]);
legend(b,{'BackSlash␣Relative␣Residual'; 'BICGSTAB␣Relative␣
   Residual'},'Location','northoutside');
ylabel('Computing␣Time␣(s)');
hold off
saveas(fig2,'graph2.png');
end
```

```
%
    ________________________________________________________________
% Relative Residual VS number of iteration
%
    ________________________________________________________________


clear M1 M2 xbicgsta flag relres iter;
load('wang3.mat');
disp(sprintf('Relative␣Residual␣VS␣Iteration␣Comparison␣with␣file␣
   wang3.mat'));

A = Problem.A;
[m,n] = size(A);
disp(sprintf('Size:␣%d␣x␣%d',m,n));
nz = max(size(nonzeros(A)));
disp(sprintf('None␣zeros:␣%d', nz));
x = rand(m,1);
b = A*x;

% looping here

tol = [1e-1 1e-2 1e-3 1e-4 1e-5 1e-6 1e-7 1e-8 1e-9 1e-10 1e-11 1e
   -12];
it = zeros(max(size(tol)),1);
rr = zeros(max(size(tol)),1);
for i = 1:max(size(tol))
% ILU nofill
setup.type = 'nofill';
setup.milu = 'off';
setup.droptol = 0.005;

[M1,M2] = ilu(A,setup);
[xbicgstab,flag,relres,iter] = bicgstab(A,b,tol(i),9999,M1,M2);
it(i) = iter;
rr(i) = relres;

disp(sprintf('Relative␣Residual:%g␣␣␣␣Number␣of␣Iteration:%g',
   relres,iter));

switch (flag)
```

```
    case 0
        disp ('bicgstab␣converged␣to␣the␣desired␣tolerance␣tol␣
           within␣maxit␣iterations.');
    case 1
        disp ('bicgstab␣iterated␣maxit␣times␣but␣did␣not␣converge.
           ');
    case 2
        disp ('Preconditioner␣M␣was␣ill-conditioned.');
    case 3
        disp ('bicgstab␣stagnated.␣(Two␣consecutive␣iterates␣were␣
           the␣same.)');
    case 4
        disp ('One␣of␣the␣scalar␣quantities␣calculated␣during␣
           bicgstab␣became␣too␣small␣or␣too␣large␣to␣continue␣
           computing.');
end
disp('␣');
end

%Generate graph
fig3 = figure
semilogx(rr,it)
xlabel('Relative␣Residual');
ylabel('Number␣of␣Iterations');
title('Number␣of␣Iterations␣VS␣Relative␣Residual')
saveas(fig3,'graph3.png');
```