An Investigation of Kaczmarz Algorithm Variations and Preconditioning

Rockwell P. Jackson, Liam M. Kelz, Vijval Rajan, Jake Truong-Jones, Rodrigo B. Platte

June-July 2023

Author's Note: From May 29th, 2023 to July 22nd, 2023, we the authors of this report participated in the AM Squared Research Experience for Undergraduates at Arizona State University. In our subsection of the program, we primarily investigated the Kaczmarz algorithm. We eventually decided that we wanted to take what we had learned and make it into a summary of the Kaczmarz method, its many variations, and how they compare. Our goal was to provide a reader-friendly summary of the topics we researched to anyone attempting to learn about them by the end od the REU. To that end, we hope you find this report useful in your endeavors. We thank the National Science Foundation and Arizona State University for the funding and organization of the REU program, and we thank Dr. Rodrigo B. Platte for his guidance and mentorship.

1 Introduction

1.1 Brief History of the Kaczmarz Algorithm

In 1937, the original Kaczmarz algorithm was proposed by Stefan Kaczmarz as a way to approximate the solution to a linear system [4]. The algorithm did not gain much attention however until 2009 when Strohmer and Vershynin [7] proved that the randomized version of the algorithm has an exponential rate of convergence. Since then, the method has become very popular due to its low computational cost compared to other methods which makes it useful in large data sets, and there has been an abundance of recent research into different variations of the algorithm.

1.2 Mathematical Background

This section will go over the classic Kaczmarz method and its variations, Polynomial Interpolation, and the Chebyshev Points. It is assumed that the reader has a background in introductory Linear Algebra.

The Kaczmarz method is used to solve the problem

$$\operatorname*{arg\,min}_{x\in\mathbb{C}^n} \|Ax - b\|, \tag{1}$$

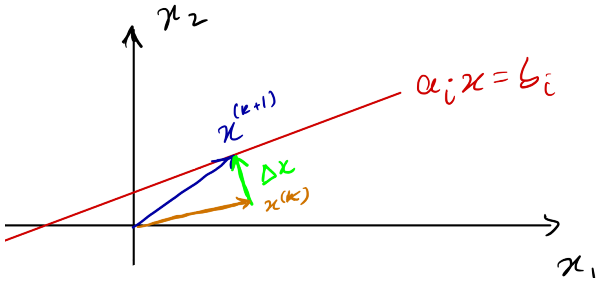


Figure 1: The projection step of $x^{(k)}$ into the solution space of $a_i x = b_i$.

where $A \in \mathbb{C}^{m\times n}$ and $b \in \mathbb{C}^m$. Here $\|\cdot\|$ denotes the Euclidean norm. Of particular interest are large data problems with $m \gg n$ with possibly corrupted information in the rows of A or entries of b. However, in the derivation that follows in this section, we will assume the linear system is consistent, i.e. there exists x such that $Ax = b$. Inconsistent systems and systems with corrupted data will be addresses in Section 3 and Section 6.

1.2.1 The Original Kaczmarz Method

Given a starting guess $x^{(0)} \in \mathbb{C}^n$, the original Kaczmarz method [4] projects the iterate $x^{(k)}$ into the solution space of a single row at each step,

$$x^{(k+1)} = x^{(k)} - \frac{a_i x^{(k)} - b_i}{\|a_i\|^2} a_i^*, \tag{2}$$

where b_i is the i^{th} entry of b, and $i = k \mod m$. Notice that in this notation a_i is a row vector, a_i* is its conjugate transpose, and $x^{(k)}$ is a column vector. The derivation of this update stems from solving the constrained optimization problem

$$x^{(k+1)} = \underset{x \in \mathbb{C}^n, \quad a_i x = b_i}{\arg\min} \|x - x^{(k)}\|. \tag{3}$$

Theorem 1.1. *The solution of problem* (3) *is given by* (2)*.*

Proof. Figure 1 illustrates the proof. Let $\Delta x = x^{(k+1)} - x^{(k)}$ and notice that the vector a_i^* is normal to the hyperplane defined by $a_i x = b_i$. Therefore, Δx must be a scalar multiple of a_i^*. In other words,

$$x^{(k+1)} = x^{(k)} + c a_i^*, \tag{4}$$

where c is the proportionality constant. Because $x^{(k+1)}$ must satisfy the equation $a_i x = b_i$, we have that

$$a_i(x^{(k)} + c a_i^*) = b_i.$$

Solving this equation for c, gives

$$c = \frac{b_i - a_i x^{(k)}}{\|a_i\|^2}.$$

Substituting this into (4) completes the proof. □

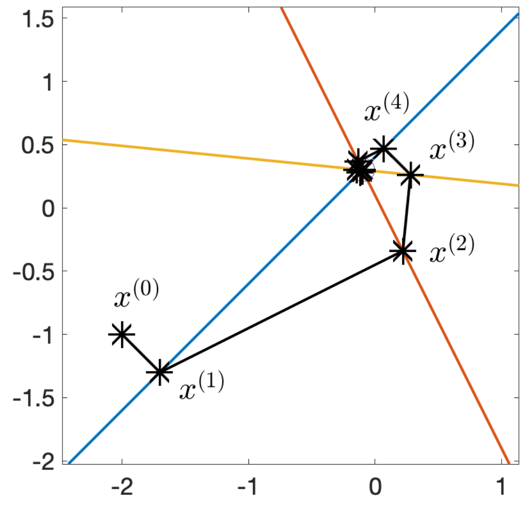


Figure 2: Solution of a 3×2 system of equations by the original Kaczmarz method.

The solution of a 3×2 system using the iteration (2) is presented in Figure 2. The system in this example is

$$\begin{bmatrix} 1 & -1 \\ 1 & 0.5 \\ 0.1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.4 \\ 0.05 \\ 0.29 \end{bmatrix}.$$

Starting from $x^{(0)} = \begin{bmatrix} -2 \\ -1 \end{bmatrix}$, Figure 2 shows the iterates $x^{(k)}$ approaching the solution $\begin{bmatrix} -0.1 \\ 0.3 \end{bmatrix}$.

1.2.2 The Projected Block Kaczmarz Method (PBK)

A natural generalization of (2) is the Projected Block Kaczmarz Method [6]. In this variant, the iterate $x^{(k)}$ is projected into the solution space of the subsystem

$$A_{\tau^{(k)}} x = b_{\tau^{(k)}}, \tag{5}$$

where the subscript $\tau^{(k)}$ denotes a subset of row indices being selected from the system (1) at iteration k (i.e. $A_{\tau^{(k)}} = A(\tau^{(k)}, :)$). This results in the following iterative process:

$$x^{(k+1)} = x^{(k)} - A^{\dagger}_{\tau^{(k)}} (A_{\tau^{(k)}} x^{(k)} - b_{\tau^{(k)}}), \tag{6}$$

where $A^{\dagger}_{\tau^{(k)}}$ is the pseudo-inverse of $A_{\tau^{(k)}}$.

Theorem 1.2. *The solution to the problem*

$$x^{(k+1)} = \underset{x \in \mathbb{C}^n, \quad A_{\tau^{(k)}} x = b_{\tau^{(k)}}}{\arg\min} \|x - x^{(k)}\|$$

is given by (6).

Proof. Let $x^{(k+1)} = x^{(k)} + \Delta x$. Because $x^{(k+1)}$ belongs to the solution space of (5), we have that

$$A_{\tau^{(k)}}(x^{(k)} + \Delta x) = b_{\tau^{(k)}},$$

or

$$A_{\tau^{(k)}}\Delta x = b_{\tau^{(k)}} - A_{\tau^{(k)}}x^{(k)}.$$

Taking the solution of minimal norm of this equation gives

$$\Delta x = A^{\dagger}_{\tau^{(k)}}(b_{\tau^{(k)}} - A_{\tau^{(k)}}x^{(k)}).$$

Hence

$$x^{(k+1)} - x^{(k)} = A^{\dagger}_{\tau^{(k)}}(b_{\tau^{(k)}} - A_{\tau^{(k)}}x^{(k)}).$$

□

1.2.3 The Averaged Block Kaczmarz Method (ABK)

The cost of computing the pseudo-inverse in (6) is $O((\tau^{(k)})^2 n)$ floating point operations (flops) and can be significant when $|\tau^{(k)}|$ (number of selected rows) is large. As an alternative, we can update $x^{(k)}$ using an average of row updates. Given a starting guess $x^{(0)} \in \mathbb{C}^n$, the ABK iteration is given by [6, 2]

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} \sum_{i\in\tau^{(k)}} \frac{w_i^{(k)}}{\|a_i\|^2} a_i^*(a_i x^{(k)} - b_i),$$

where $w_i^{(k)}$ are the average weights (e.g. $w_i^{(k)} = 1/|\tau^{(k)}|$), and $\alpha^{(k)}$ is a relaxation parameter, which is also referred to as the step size.

In matrix form, this is equivalent to

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} A^*_{\tau^{(k)}} W^{(k)} (A_{\tau^{(k)}} x^{(k)} - b_{\tau^{(k)}}), \tag{7}$$

where

$$W^{(k)} = \mathrm{diag}(w_i^{(k)}/\|a_i\|^2).$$

Therefore, we can interpret $\alpha^{(k)} A^*_{\tau^{(k)}} W^{(k)}$ as a cheap approximation to $A^{\dagger}_{\tau^{(k)}}$. Notice that the ABK update only requires $O(\tau^{(k)} n)$ flops.

1.2.4 The Minimal Residual Step Method (MRSM)

Several convergence properties of ABK methods have been presented in [6] for consistent systems and in [2] for systems with corrupted data. Both papers emphasize the importance of the step size $\alpha^{(k)}$ in the convergence rates and propose choices for this parameter that optimize certain error bounds. In this work, we derive an expression for $\alpha^{(k)}$ that minimizes residuals at each iteration.

In order to derive an expression for $\alpha^{(k)}$, let $\mu^{(k)}$ be a subset of row indices, $\mu^{(k)} \subset \{1, 2, \ldots, m\}$. Computing the residual at this set of rows using $x^{(k+1)}$, the ABK update (7) gives

$$A_{\mu^{(k)}} x^{(k+1)} - b_{\mu^{(k)}} = A_{\mu^{(k)}}(x^{(k)} - \alpha^{(k)} s^{(k)}) - b_{\mu^{(k)}} = A_{\mu^{(k)}} x^{(k)} - b_{\mu^{(k)}} - \alpha^{(k)} A_{\mu^{(k)}} s^{(k)}, \tag{8}$$

where

$$s^{(k)} = A^*_{\tau^{(k)}} W^{(k)} (A_{\tau^{(k)}} x^{(k)} - b_{\tau^{(k)}}).$$

Minimizing the residual in the right hand side of (8),

$$\min_{\alpha^{(k)}} \| A_{\mu^{(k)}} x^{(k)} - b_{\mu^{(k)}} - \alpha^{(k)} A_{\mu^{(k)}} s^{(k)} \|,$$

we obtain

$$\alpha^{(k)} = \frac{(A_{\mu^{(k)}} s^{(k)})^* (A_{\mu^{(k)}} x^{(k)} - b_{\mu^{(k)}})}{\| A_{\mu^{(k)}} s^{(k)} \|^2}. \tag{9}$$

For computational efficiency, our algorithms use $\mu^{(k)} = \tau^{(k)}$ in each iteration. See Algorithm 1 for detials. Another option is to select $\mu^{(k)} = \{1, 2, \ldots, m\}$ (i.e. $A_{\mu^{(k)}} = A$) as suggested in [12].

Algorithm 1 ABK update with MRSM

```
Require: $x^{(0)}_{n\times 1}$, $A_{m\times n}$, $b_{m\times 1}$
  for $k = 1, 2, \ldots$ do
    $\tau^{(k)} \leftarrow$ select block of rows    ▷ computational cost depends on row selection strategy
    $r^{(k)} \leftarrow A_{\tau^{(k)}} x^{(k)} - b_{\tau^{(k)}}$    ▷ $O(|\tau^{(k)}|n)$ flops
    $s^{(k)} \leftarrow A^*_{\tau^{(k)}} r^{(k)}$    ▷ $O(|\tau^{(k)}|n)$ flops
    $c^{(k)} \leftarrow A_{\tau^{(k)}} s^{(k)}$    ▷ $O(|\tau^{(k)}|n)$ flops
    $\alpha^{(k)} \leftarrow {c^{(k)}}^* r^{(k)} / \|c^{(k)}\|^2$    ▷ $O(|\tau^{(k)}|)$ flops
    $x^{(k+1)} \leftarrow x^{(k)} - \alpha^{(k)} s^{(k)}$    ▷ $O(n)$ flops
```

1.2.5 Polynomial Approximation and Chebyshev Points

A common application of least-squares solvers is curve fitting. In the case of interpolation, the matrix is square. Consider a polynomial written using the **monomial** basis $\{1, x, x^2, \ldots\}$ in 1–D :

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots a_n x^n.$$

Given a function f, the interpolation condition $f(x_j) = p_n(x_j)$ leads to the system

$$\begin{aligned}
f_0 &= a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_0^3 + \ldots a_n x_0^n \\
f_1 &= a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 + \ldots a_n x_1^n \\
f_2 &= a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3 + \ldots a_n x_2^n \\
\vdots &\;\;\vdots\;\;\vdots \\
f_n &= a_0 + a_1 x_n + a_2 x_n^2 + a_3 x_n^3 + \ldots a_n x_n^n
\end{aligned}$$

or in matrix form

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

This system is expensive to invert, that is $O(n^3)$, and ill-conditioned.

In order to circumvent the ill-conditioning, one can use Chebyshev polynomials instead. They are defined by

$$T_n(x) = \cos(n \arccos(x)),$$

where, n is the polynomial degree. Writing our polynomial in the Chebyshev basis,

$$p_n(x) = a_0T_0(x) + a_1T_1(x) + a_2T_3(x) + a_3T_3(x) + \dots a_nT_n(x),$$

the interpolation conditions lead to the linear system

$$\begin{bmatrix} 1 & T_1(x_0) & T_2(x_0) & \dots & T_n(x_0) \\ 1 & T_1(x_1) & T_2(x_1) & \dots & T_n(x_1) \\ 1 & T_1(x_2) & T_2(x_2) & \dots & T_n(x_2) \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & T_1(x_n) & T_2(x_n) & \dots & T_n(x_n) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

The choice of interpolation points also important to prevent instabilities and ill-conditioning. In 1–D, it is well known that Chebyshev points should be used [8]. In our examples, we used the Chebyshev points of the second kind, which are given by

$$x_j = \cos(j\pi/n), \quad j = 0, \dots, n.$$

An interpolation matrix formed with Chebyshev polynomials evaluated at Chebyshev points is called a Chebyshev matrix. These matrices will be used in several numerical tests in the following sections.

2 Explanation of and Experiments with Block Kaczmarz Variations

There are multiple variations of the Block Kaczmarz Method, and some are more useful under different circumstances than others. In each step of the algorithm, there is a row selection part and an iteration part. This section will cover three different row selection methods, two different iteration methods, and compare all of the combinations in terms of CPU time and iterations taken against the degree level of the system.

2.1 Row Selection Methods

Row Selection methods dictate which rows of the system the algorithm will use to update the approximation. Such methods covered here are the Random, Targeted, and Random-Targeted methods.

Random Row Selection: This method will select a random block of rows from the system with no preference toward any given row in the system. This method is best used with large data sets in order to avoid calculating the residual entries for each row.

Targeted Row Selection: This method will calculate the residual for all the rows of the system and then select the block of rows with the highest residual entries. This method is best used with small data sets where obtaining the whole residual is not as computationally taxing.

Random-Targeted Row Selection: This method is a hybrid of the previous two options. It will select a random block of rows from the system, then it will calculate the residual entries for that block, and finally it will pick the block of rows of the random subset that have the highest residual entries. For this algorithm to work, it needs to have two block sizes: an initial block size for the random selection, and a runoff block size for picking the highest residuals. The initial block size is equal to some scalar times the runoff block size where the scalar is greater than or equal to 1. Our experiments set this scalar equal to 2 for the sake of simplicity.

Aside: A targeted-random row selection is not useful because it does the taxing residual calculation and then allows the random selection to pick any of those rows regardless of the residual entries. At that point, it would have been better to just pick the rows with the highest residual entries.

2.2 Iteration Methods

Iteration methods dictate what the algorithm does with the selected rows of the system to update the approximation. Such methods covered here are the averaged and projected methods.

Averaged Iteration: The general formula for the Averaged Block Kaczmarz is given by (7). This method will project the current approximation onto each row in the selected block and then take the average of all those projections to update the approximation.

Projected Iteration: The general formula for the Projected Block Kaczmarz is given by (6). This method will project the current approximation onto the subspace created by the rows of the selected block to update the approximation.

2.3 The Effect of Degree Level on CPU Time and Iterations Taken

Each combination of row selection and iteration methods were used to perform polynomial interpolation on the following function: $f(x) = e^x \sin(x^2)$. The results in this section are shown for only one function because the convergence rate for optimal polynomial interpolation is independent of the function used provided it is well behaved, and because this experiment is meant to compare algorithms, not functions. CPU Time and Iterations taken were measured against degree level (the size of the matrix) for a constant block size where the initial block size for RT was twice the runoff block size, the interpolation domain was $[0, 1]$, the Chebyshev matrix was used, the relative tolerance was 10^{-10}, and the maximum iterations allowed was 40000. Each algorithm was provided the actual solution for stopping criteria. The degree level was measured at exponentially discrete points by powers of 2, and the number of runs each data point averages is inversely proportional to the degree level. The results are presented in Figure 3, where we can see that the targeted method performs best in terms of iterations taken regardless of block size or iteration method. This is expected as that row selection method is the greediest of the three. It also initially performs better in terms of CPU time up to a point where calculating the whole residual becomes too expensive to justify the greed. Then, the random or hybrid method becomes the better choice. Larger block sizes seem to delay when this crossover happens, but taking larger block sizes is not always desirable. The random selection method becomes best as degree level becomes large for the averaged iteration method while the hybrid method seems to enjoy some superiority in the projected case. However, the averaged algorithms perform better than their projected counterparts at all stages in terms of CPU time. This is likely due to the pseudo-inverse being expensive to compute.

2.4 The Effect of Block Size on CPU Time and Iterations Taken

The same combinations of methods were tested on a system of 10000 rows and 1000 columns that was made to be consistent and have a good condition number. The independent variable is now block size. The results are presented in Figure 4, where we can see that iterations taken decreases with block size because the algorithm is given more information per iteration. In terms of time, there is a minimum in each method which indicates that larger block sizes are not always better. The optimal points for each algorithm depends on the size of the system, so it is difficult to make a general statement about when each method is best. The reason the projected algorithms have a cliff at a block size of 1000 is because that block size makes the pseudo-inverse square. Because the system is consistent, this makes obtaining the pseudo-inverse equivalent to solving the system in one iteration. Increasing block size beyond that point simply wastes time. The matrix used in this experiment was heavily manufactured, so picking block size to make the pseudo-inverse square will not always work this way, so the cliff that occurs should not be taken as representative.

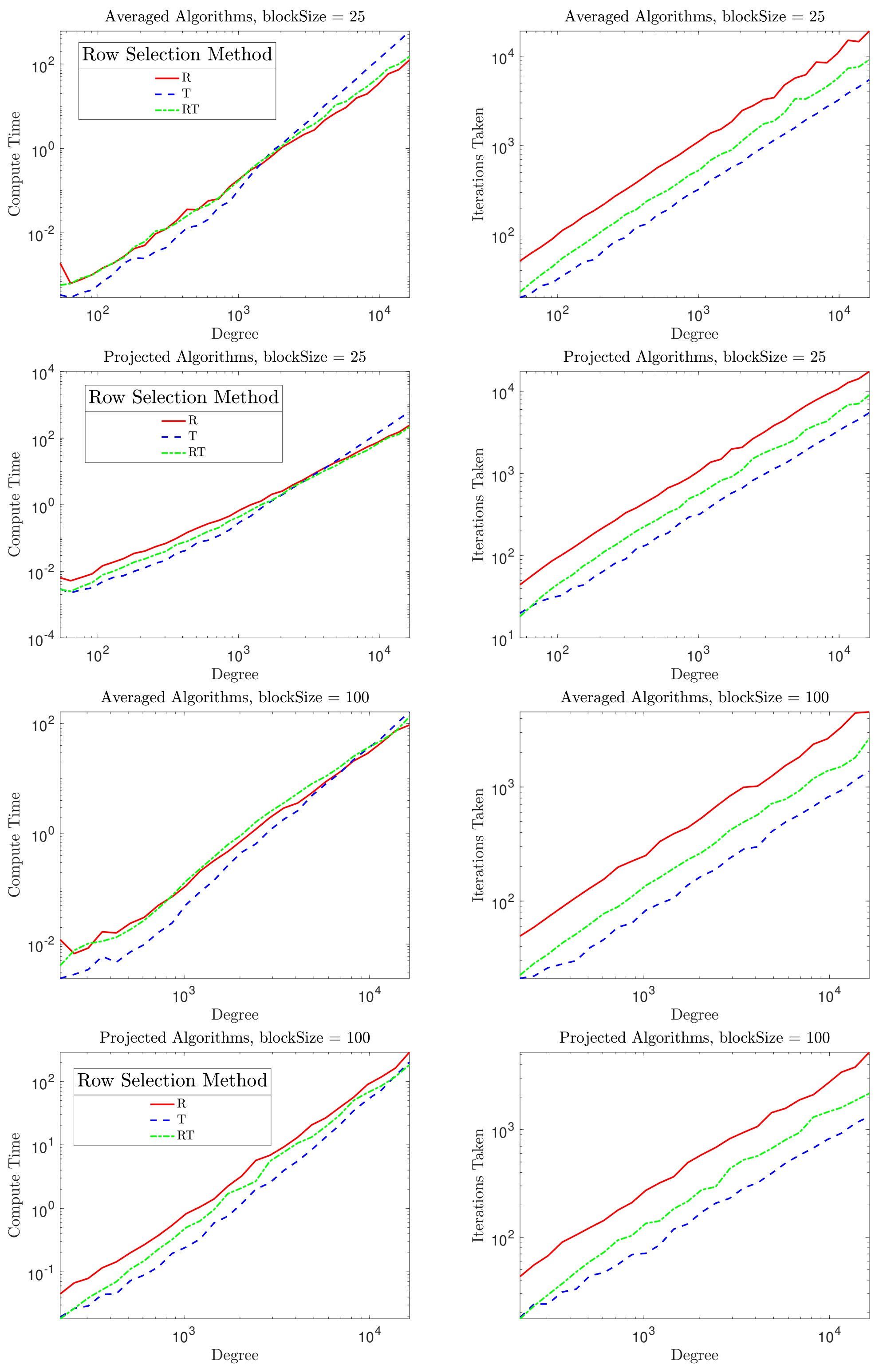


Figure 3: CPU time in seconds (left) and Iterations (right) against degree level for block sizes 25 and 100

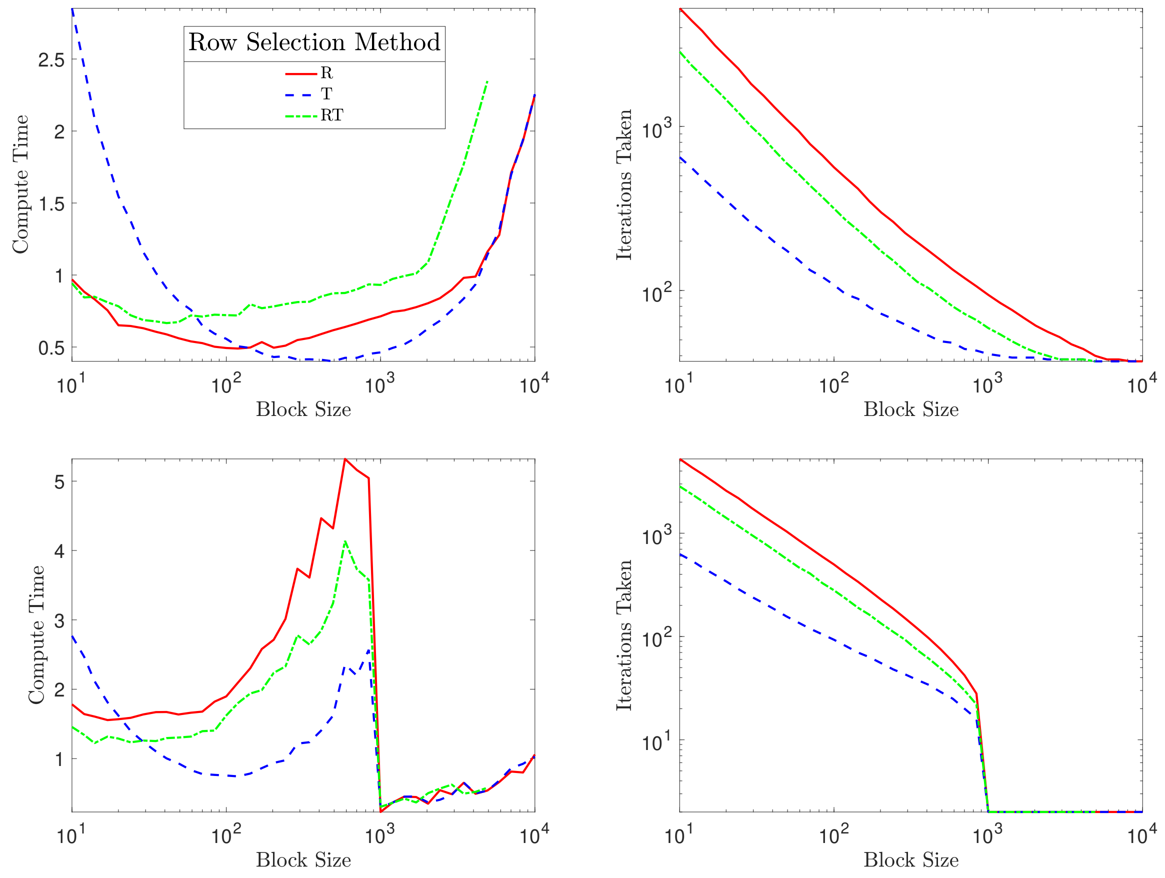


Figure 4: CPU time in seconds (left) and Iterations (right) as a function of the block size for averaged (top) and projected (bottom) algorithms.

3 Inconsistent Systems

All of the versions of Kaczmarz mentioned so far are designed for consistent systems, where there is an solution that will exactly satisfy the whole system. However, this leaves out the possibility of an inconsistent system. Such systems don't have an exact solution, but a least squares solution is still desired. For full-rank inconsistent systems, the solution will be the unique least squares solution. For rank-deficient systems, the solution will be the unique minimum norm least squares solution. The methods we discussed before will begin to converge towards the least squares solution, but when the approximation reaches the error horizon, it will fail to converge. One solution to this problem could be to progressively relax the step-size α, which shrinks the error horizon, but the smaller step-sizes means convergence will take longer. This penalty increase dramatically with desired precision. Therefore, a better solution is desirable which is presented in this section.

3.1 Extended Kaczmarz Method

This method introduces the new variable z to solve the system $A'z = 0$ with a z initial equal to b. The update step for x and residual calculation are changed as to minimize $Ax - b + z$. The z system functions as a translation to the hyper-planes by calculating an adjustment to b. This allows $Col(A) = 0$ for the least squares solution, allowing the new system to be solved with our current methods. This effectively splits the inconsistent system into two consistent systems.

3.2 Separated Extended Block Kaczmarz

The Extended Block Kaczmarz method updates both the z variable and the x variable every iteration. While the x variable does rely on the z in order to converge, the z variable doesn't rely at all on the x. In addition, the convergence times for z and x are separate, with the convergence of z being correlated with the number of columns. We determined that there is significant decrease in computation time if the z system is solved separately, especially for tall rectangular matrices. These results are demonstrated in Figure 5, where we can see that the advantage increases with the ratio of rows to columns.

4 Adaptive Alpha

Many variations of Kaczmarz incorporate an α parameter, also known as the "relaxation parameter" or "step size". This is a scalar for the vector that updates the approximation. As previously discussed, this parameter has the ability to cause systems to converge that normally would not (by progressive relaxation), or by increasing the rate of convergence by choosing better values of alpha. In this section, we introduce multiple adaptors for alpha that can assist in achieving the latter.

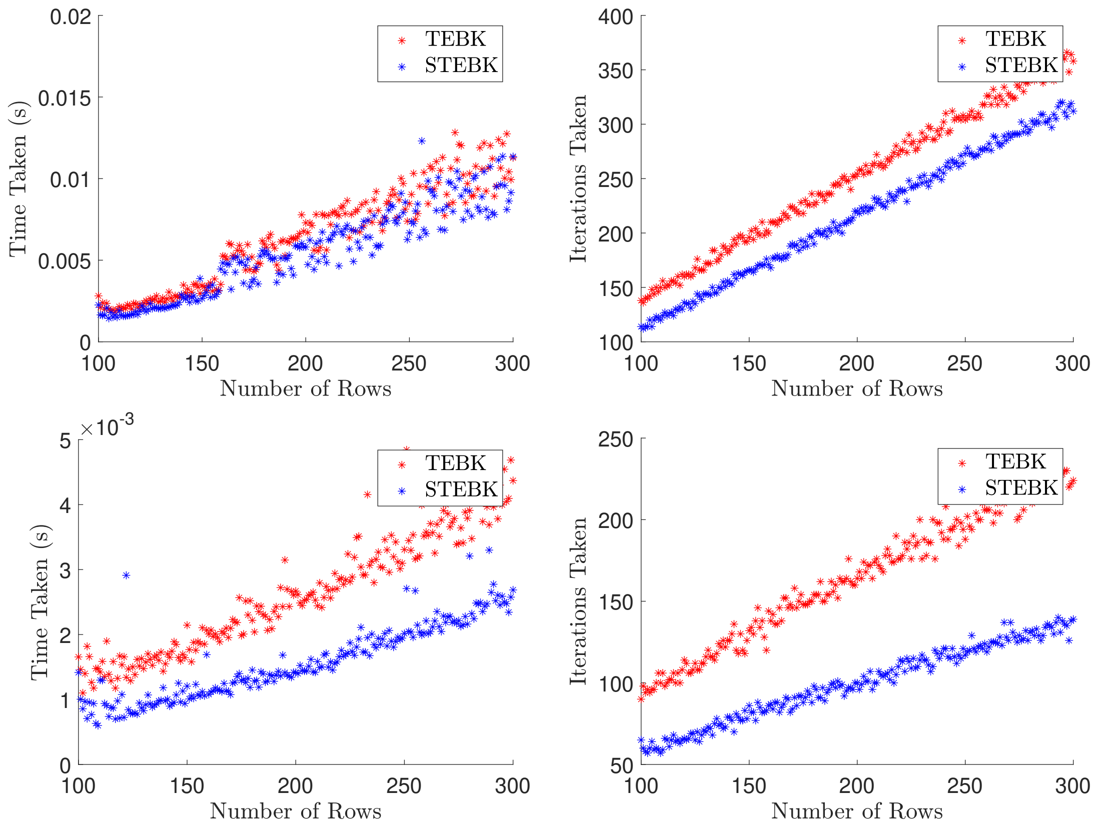


Figure 5: CPU time in seconds (left) and Iterations (right) as a function of the number of rows for a number of columns equal to 50% of the rows (top) and 10% of the rows (bottom).

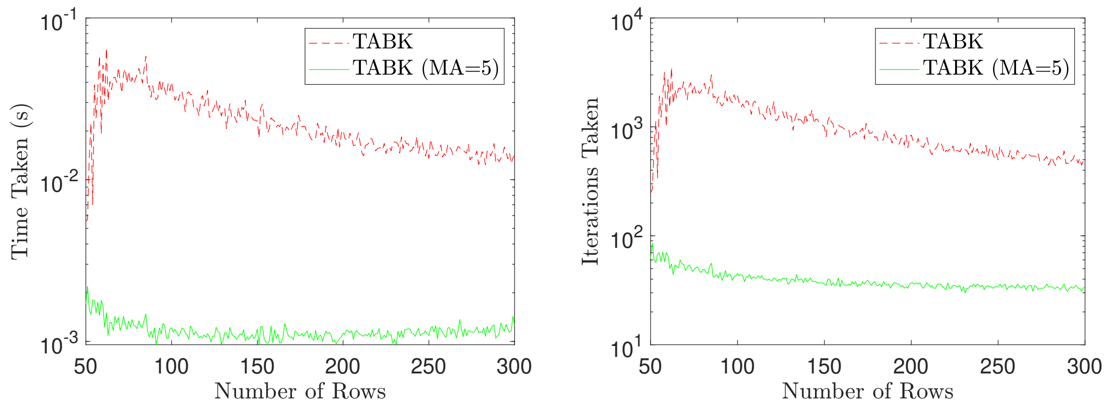


Figure 6: CPU time in seconds (left) and Iterations (right) as a function of the number of rows for a random matrix with 20 columns and a block-size of 10.

4.1 Global Adaptive Alpha

During our research into step-size and convergence rate, we explored a method to progressively increase alpha. This method attempts to optimize alpha over the course of convergence. While more work needs to be done to stabilize and formalize this method, it successfully demonstrated the proof of concept around methods that can iteratively increase convergence rate. It uses the change in residual as a heuristic for convergence rate, and iteratively approaches the maximum alpha that still converges. This alpha is often not efficient, but is close to the optimal alpha.

4.2 Multi-Alpha

Multi-Alpha is an implementation of the Greedy Block Adaptive Alpha that utilizes information from previous iterations in order to speed up convergence. This method uses a rolling window history (the length of which we refer to as nAlpha) of previous blocks and update calculations in the calculations of the Greedy Block Adaptive Alpha. Since these histories are a function of both block-size and the length of the histories, this method has the potential to dramatically increase the number of calculations required for each iteration. However, the majority of increases to the rate of convergence happen with the first few values of nAlpha. While the most significant increases to convergence occur with when increasing from an nAlpha of 1 to 2, an nAlpha of 2 does not guarantee convergence due to possibility of local minima. These observations are demonstrated in Figure 6 which compares TABK using MRSM and TABK using MultiAlpha, and in Figure 7 which compares different levels of MultiAlpha.

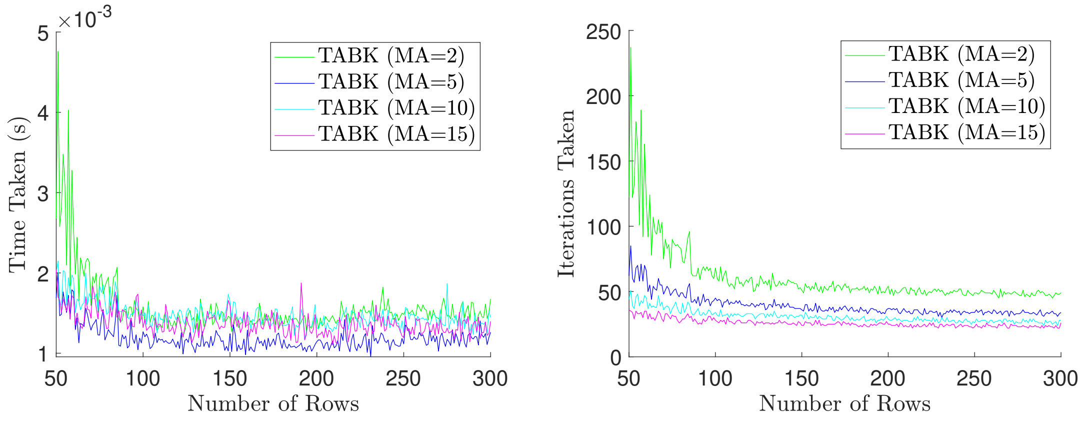


Figure 7: CPU time in seconds (left) and Iterations (right) as a function of the number of rows to compare different multi-alpha levels for a random matrix with 20 columns and a block-size of 10.

5 Preconditioning Methods

5.1 Overview

Condition number is the product of the norm of the inverse of a matrix and the norm of the original matrix. Condition number measures the sensitivity of the solution to small perturbations in the original matrix. The solution to a matrix system using a matrix with a small condition number will be changed minimally with small perturbations to the original matrix. Conversely, the solution to a matrix system with a large condition number will be greatly affected to small perturbations in the original matrix. Condition number $\kappa(A)$ of a matrix A is given by:

$$\kappa(A) = \left\|A^{-1}\right\| \|A\| .$$

The condition numbers greatly affect the time taken for iterative solvers to converge. This is illustrated in Figure 8 where we show how a system with a lower condition number is more quickly solved compared to a system with a high condition number.

Preconditioning a system before applying an iterative solver can reduce the time taken to convergence. One form of preconditioning is the process of lowering condition number through applying transformations to the original matrix. Transforming the original matrix results in easier to solve matrix systems. A system

$$Ax = b$$

can be transformed into a system by applying a left preconditioner C^{-1}

$$C^{-1}Ax = C^{-1}b.$$

which is faster to solve given the correct C^{-1} [9]. In this paper, preconditioning was used to reduce the condition number of the original matrix.

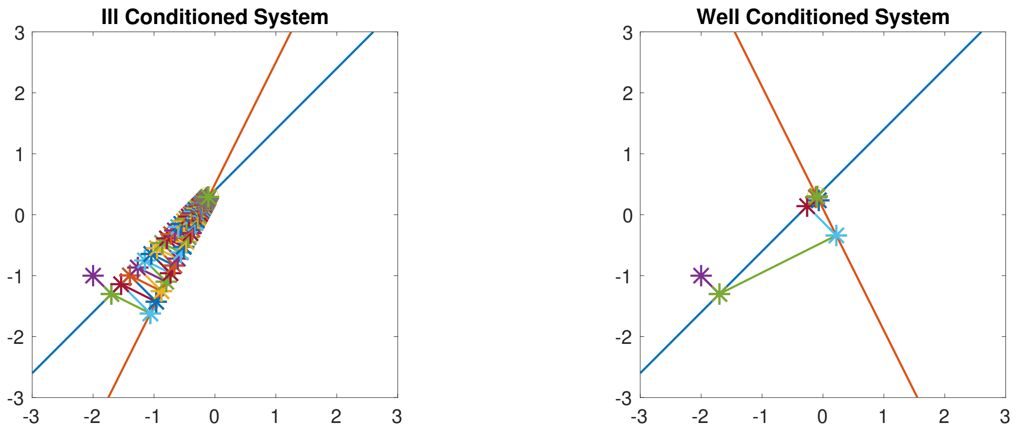


Figure 8: Comparison of convergence of Kaczmarz on an Ill Conditioned System vs. Well Conditioned System

5.2 Algebraic Multigrid

Algebraic multgrid (AMG) is an optimally fast method which runs in $O(n)$ time [3]. There has been a lot of research on parallelizing algebraic multigrid so it can run in constant time [3]. AMG involves smoothing and coarse-grid correction. Smoothing applies an iterative algorithm to reduce high frequency error [11]. Coarse-grid correction reduces low-frequency error [3]. AMG can be used as a preconditioner to reduce condition number.

While AMG is a fast method, it requires significant setup. There are a number of parameters which need to be fine tuned for AMG to work correctly. This paper used `PyAMG` to test out different parameters for AMG on various random and sparse matrices.

5.3 PyAMG

`PyAMG` is a Python library for creating tools using algebraic multigrid [1]. `PyAMG` was developed to create readable and simple-to-use AMG methods [1]. Preconditioners were created by choosing a solver (ruge-stuben, smoothed aggregation solver, or rootnode solvers), and applying them to a system. The goal of these experiments was to to create a algebraic multigrid preconditioner which lowered the condition number of a random matrix utilizing `PyAMG`. `PyAMG` works with square matrices, so all matrices used in the experiments were square.

5.4 Experiments with PyAMG

Different types of square, random, sparse, and random sparse matrices were generated to test the effectiveness of using AMG as a preconditioner to reduce condition number. `PyAMG` was used to create the AMG methods. `NumPy` was used to generate random numbers and to hold the matrices used in the experiments. `SciPy` was used to work specifically with sparse matrices. The experiment consisted of 4 steps. First, the system was generated

$$Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad x, b \in \mathbb{R}^n.$$

The x solution was generated randomly. Each system was consistent with a random x to ensure that the Kaczmarz algorithm could converge. Then, the AMG preconditioner was generated using the original system and applied on the left

$$MAx = Mb.$$

If the condition number of the preconditioned matrix was lower than the original system, the time taken for Kaczmarz to converge on each system was recorded and compared.

5.5 Sparse random matrices

Random matrices were first tested on the `PyAMG` preconditioner. A pseudo random sparse matrix was generated using `NumPy` and `SciPy`. `NumPy` generated the pseudo random numbers which filled the `SciPy` matrix. The CSR format was used to hold the sparse matrix. The AMG preconditioner was created using the smoothed aggregation solver. The AMG preconditioner increased the condition number of the sparse random matrices. Matrices with better structure were explored to avoid the matrix structure problems with sparse random matrices.

5.6 Tridiagonal matrices

A well structured matrix was first generated to test the `PyAMG` preconditioner. A sparse tridiagonal symmetric matrix was generated in the form

$$T_n = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

This matrix is also known as the second difference matrix. For testing, we used a matrix size of 100×100.

The AMG preconditioner was created using the smoothed aggregation solver. The AMG preconditioner successfully reduced the condition number from very high to 1. Matrices with diagonal structure were explored to recreate the effectiveness of AMG on tridiagonal matrices.

5.7 Gaussian random sparse matrices

Gaussian random sparse matrices were used as a base matrix to perform further experiments. Gaussian random sparse matrices are better structured than random sparse matrices.

Gaussian random matrices were generated in the form:

$$A = \begin{bmatrix} \mathrm{A}_{1,1} & \mathrm{A}_{1,2} & \dots & \mathrm{A}_{1,n} \\ \mathrm{A}_{2,1} & \mathrm{A}_{2,2} & \dots & \mathrm{A}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{A}_{n,1} & \mathrm{A}_{n,2} & \dots & \mathrm{A}_{n,n} \end{bmatrix}$$

where

$$\mathrm{A}_{i,j} = \exp(-c^2(x_i - x_j)^2) \quad \text{for } i, j = 1, \dots, n.$$

where n is the size of the matrix, and $c \geq 0$.

These matrices were augmented into sparse matrices by setting a tolerance level for the values:

$$\mathrm{A}_{i,j} = \begin{cases} \exp(-c^2(x_i - x_j)^2) & \text{if } \exp(-c^2(x_i - x_j)^2) \geq 1e - 6 \\ 0 & \text{otherwise} \end{cases}$$

The resulting matrix A was the Gaussian sparse random matrix, also known as the Gaussian kernel.

The AMG preconditioner was created using the smoothed aggregation solver and the Gaussian sparse random matrices. The AMG preconditioner reduced the condition number of the original matrix, but not to 1. Still, the condition number was greatly reduced. Kaczmarz was able to converge faster on the preconditioned system versus the original system.

5.8 Matrix diagonalization

Matrix diagonalization methods were explored to make the matrix more diagonal before preconditioning. This was performed to try and exploit the sensitivity of AMG to matrix structure. Reverse Cuthill-McKee and Row Permutation methods were explored to make the original matrix more diagonal.

5.8.1 Reverse Cuthill McKee

Reverse Cuthill-McKee was used as a row permutation algorithm to make a matrix more diagonal. It was explored because of its use of row permutation, which ensures it does not affect the solution space and also to maintain low computational complexity. Figure 9 shows how matrix diagonalization is increased after the application of reverse cuthill-mckee.

After applying reverse cuthill-mckee, the AMG preconditioner reduced the condition number of the matrix. The condition number was not reduced to 1 but was reduced. Kaczmarz did not converge with either matrix.

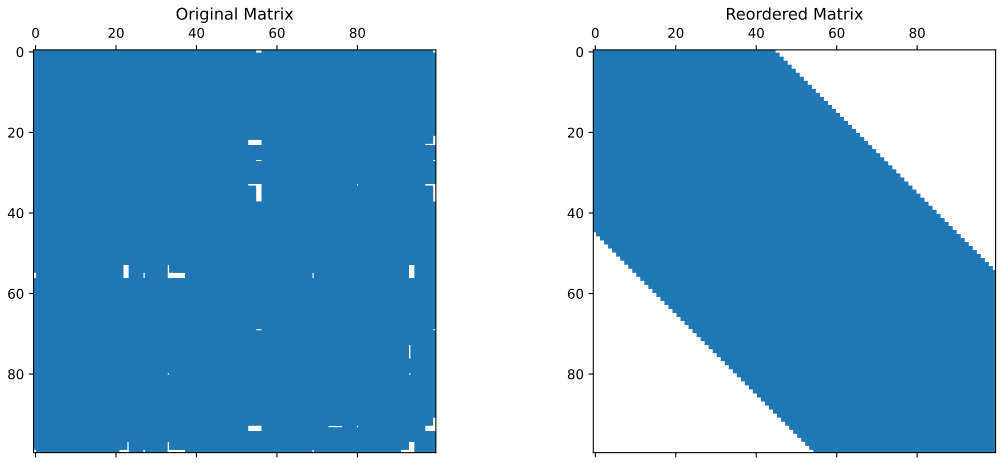


Figure 9: Comparison diagonalization of original matrix and reordered Cuthill-McKee matrix

5.8.2 Row Permutation

A row permutation method was applied to make the original matrix more diagonal. Applying the row permutation matrix did not reduce the condition number after applying the AMG preconditioner. The AMG preconditioner greatly increased the condition number of the preconditioned system compared to the original system.

5.9 Solvers

Three different solvers were explored while forming the AMG preconditioner, the ruge-stuben solver, the smoothed aggregation solver, and the rootnode solver.

All three methods reduced the condition number of the original Gaussian random matrix. The smoothed aggregation solver reduced the condition number by the greatest factor, and was used for all of the AMG preconditioners.

5.10 Sketching Experiments in Python

Utilizing a method of preconditioning Kaczmarz (RKM) by sketching proposed by Katrutsa and Oseledets [5], we first sought to replicate their results and explore further. The sketching method attempts to compress matrix A into smaller more manageable matrices to be preconditioned. The cost of preconditioning the entire matrix A is computationally expensive. It consists of the QR decomposition and the inversion of the upper triangular matrix R which results in a total complexity of $O(mn^2) + O(n^3)$ This becomes problematic when m is significantly larger than n.

Through the sketching process, the original matrix A is replaced by a smaller sketched matrix

$$\hat{A} = SA \in \mathbb{R}^{r \times n}$$

$S \in \mathbb{R}^{r \times m}$ is a given matrix and $r < m$ [10]. After calculating QR decomposition of the sketched matrix

$$\hat{A} = \hat{Q}\hat{R}$$

Where $\hat{Q}$ is orthogonal and $\hat{R}$ is the upper triangular matrix, we approximate P_R^* using $\hat{P}_R = \hat{R}^{-1}$ as the preconditioner

The sketching method introduces the variable γ where the selection of sketched rows is

$$r = \gamma n$$

where the given value $\gamma \geq 1$. Thus, the complexity of QR decomposition of a sketched matrix can be written as $O(rn^2)$ or $O(n^3)$ and no longer depends on the size of m in matrix A

Once the replicated experiment demonstrated similar results to Katrutsa and Oseledets, [5] we sought to explore three additional areas of interest:

- Apply the preconditioning method to sparse matrices
- Investigate an optimal gamma by expanding the size of the matrix and increasing gamma range
- Apply the preconditioning method to alternative versions of the Kaczmarz method, specifically randomized average block Kaczmarz.

5.10.1 Replicated Results

The replicated experiment implements the same matrix conditions as the original experiment, utilizing a random matrix of size $m = 1000$, $n = 100$ with the only difference being a change in the seed used to randomly generate the matrix. CPU time (plus time to create preconditioner) is plotted against Relative Error which is defined as:

$$\frac{\|A\hat{P}_R x_k - b\|_2}{\|b\|_2}$$

In Figure 10 we show similar results to the original experiment, demonstrating improved efficiency of convergence with preconditioning when $\gamma > 1$.

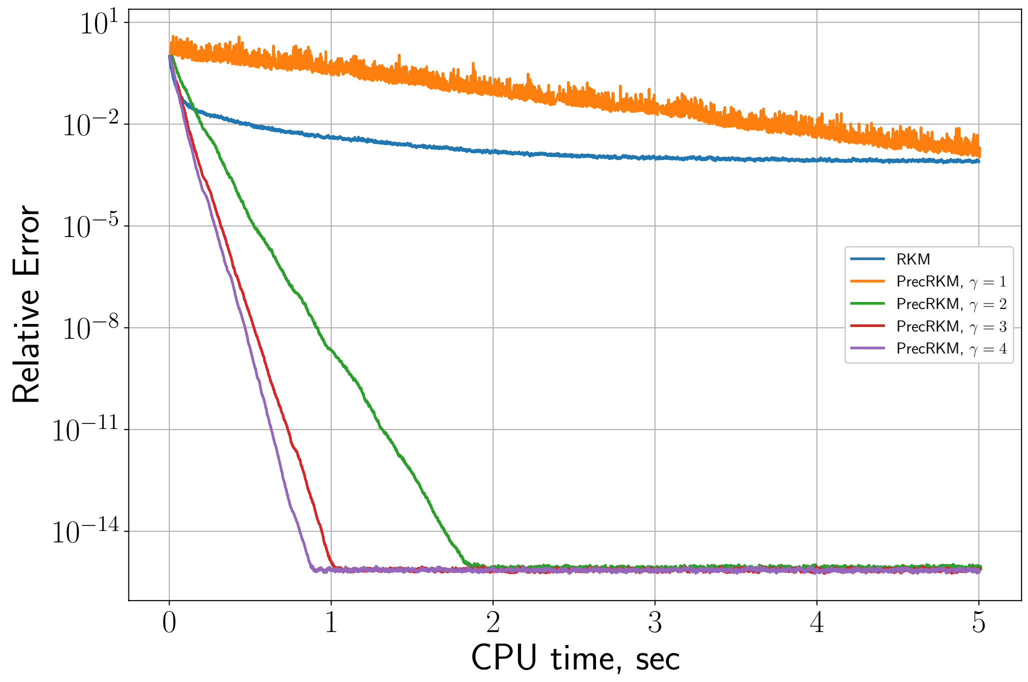
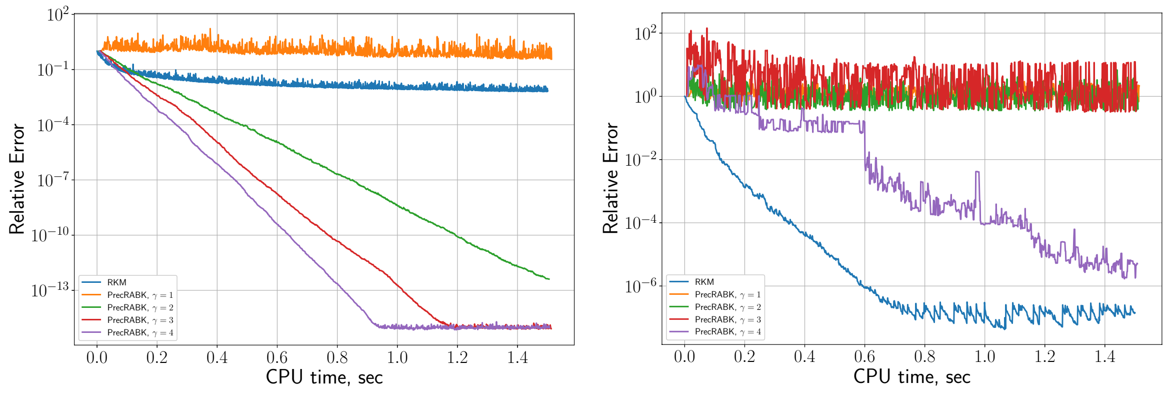


Figure 10: Comparison of convergence speed for preconditioned randomized Kaczmarz method



(a) Sparse matrix with a density of 10 percent (b) Sparse matrix with a density of 1 percent

Figure 11: Comparison of preconditioned randomized Kaczmarz method on varying density of sparse matrices

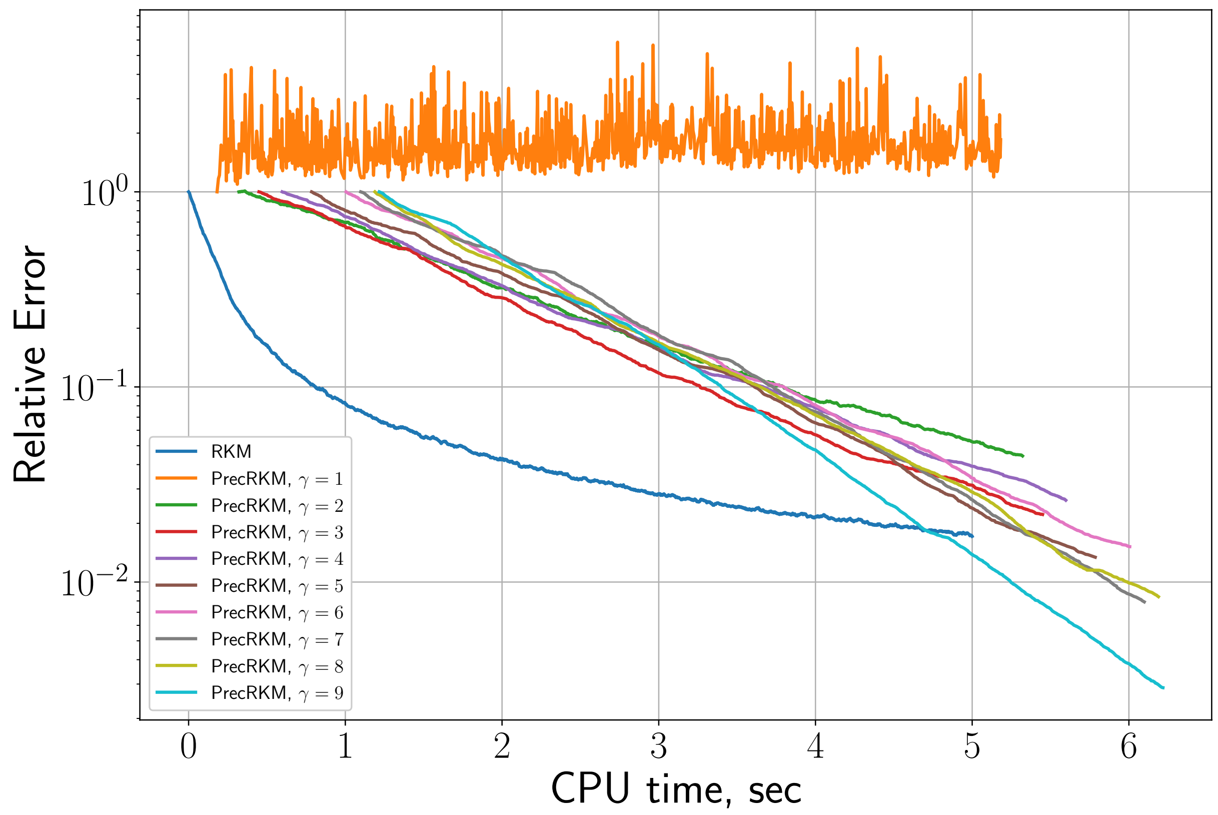


Figure 12: Expanded matrix size and γ range increased

5.10.2 Sparse Matrices

Next, we maintain the same matrix dimension and generate sparse matrices with varying levels of density to view the effectiveness of the preconditioning method on different types of matrices. In Figure 11 we show a continued effectiveness of the preconditioning method on a sparse matrix with a density of 10%. However, we observe a breakdown of the preconditioning method when the matrix has a low density of 1% (high sparsity).

5.10.3 Expanded Matrix and Gamma Range

Next, we adjust the matrix size and range of γ to explore the effectiveness of this preconditioning method on larger data sets as well as begin to investigate the optimal range of γ. We expand to a matrix of size $m = 100000$, $n = 1000$ and increase the γ range to 9. In Figure 12, we begin to see how preconditioning can become computationally expensive as the size of the matrix increases.

5.10.4 Preconditioned Randomized Average Block Kaczmarz

To overcome the computational expense of the increased matrix size, we apply the preconditioning method to a more efficient form of Kaczmarz, the randomized average block Kaczmarz (RABK). Figure 13 shows a significant improvement in the preconditioned RABK

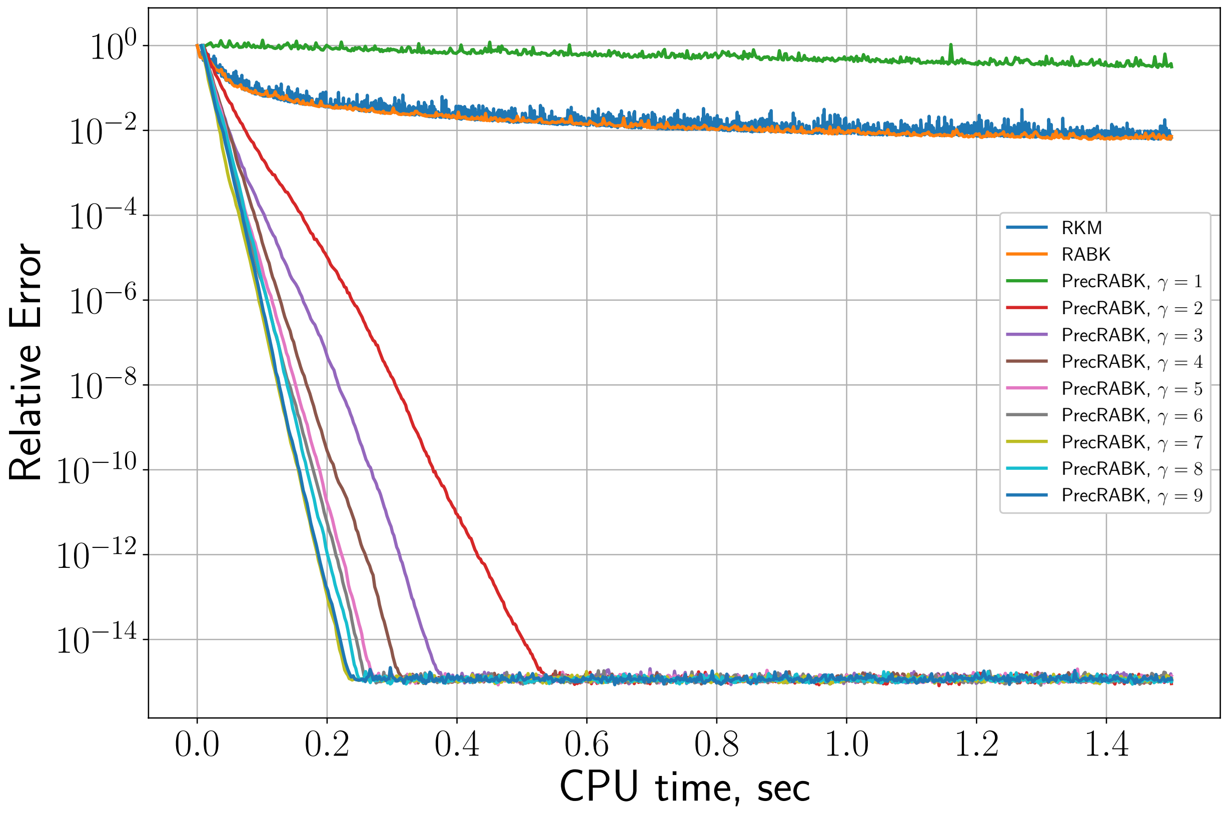


Figure 13: Preconditioned randomized Kaczmarz

compared to RABK or RKM. Additionally, the results suggest diminishing returns in preconditioning convergence speed when $\gamma \geq 6$.

5.10.5 Iterative Method Comparison

Finally, we compare RKM, RABK, preconditioned RKM, preconditioned RABK and a least squares solver imported from the `SciPy` python library on a single graph. The `SciPy` LSQR function is an iterative method commonly used for solving sparse matrices and offers a contextual baseline for the practical efficacy of the preconditioner. In Fig. 14 and Fig.15, we demonstrate that the preconditioned RABK converges quickest in both random and sparse matrices.

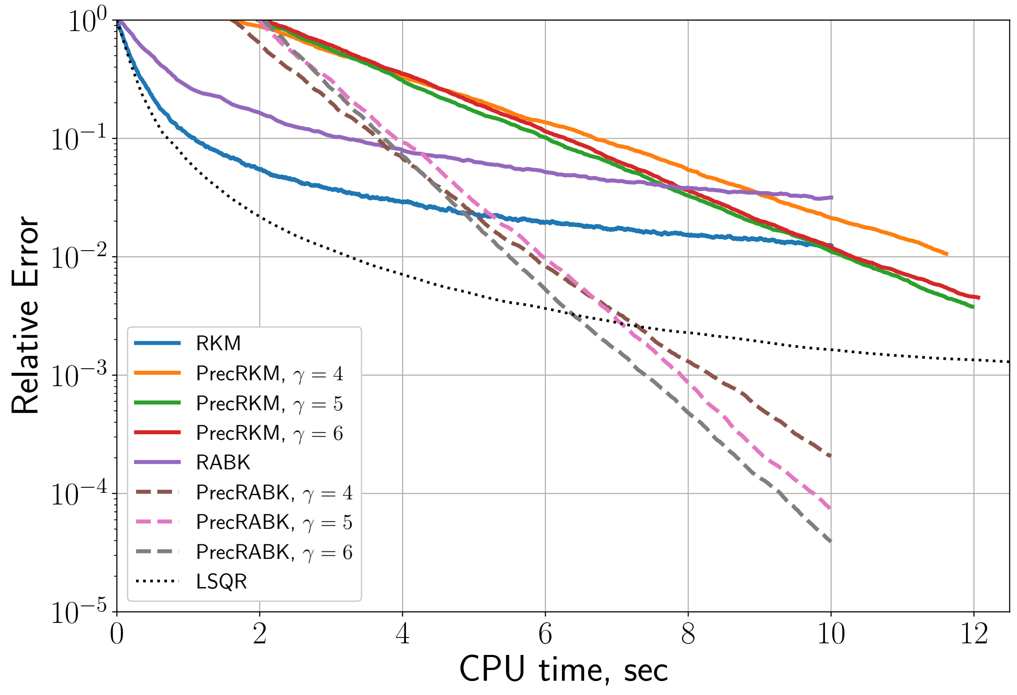


Figure 14: Iterative method comparison - Random Matrix

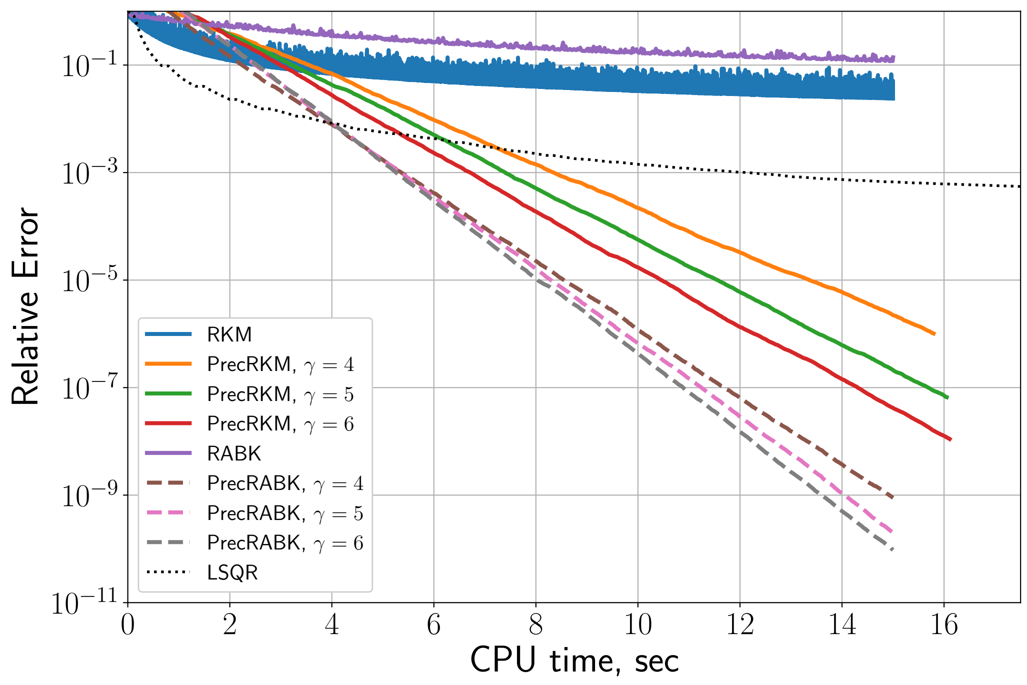


Figure 15: Iterative method comparison - Sparse Matrix

6 Corrupted Data Problems

In applications involving real-world data, one problem that may arise is the presence of corrupted data. That is, given the linear system $Ax = b$, some percentage of the entries of b are corrupted in some way; it could be that the sensor fails and inputs default values of zero, or it just assigns some random value to those entries. Whatever the type of corruption may be, it is a much larger disturbance than just a small amount of noise, and that renders the previously described methods unusable. Fortunately, there is a way to work around this problem.

6.1 Corrupted Data and the Quantile Row Selection Method

As long as the corruption is not affecting a high enough percentage of the system, the solution can still be obtained if the corrupted rows are excluded from the row selection. One way to identify a corrupted row would be to check the residual entries. If the solution is being approached and there is a corrupted row, that row will have a much larger residual entry than the others which would indicate to the algorithm that it should not be used. Using this logic, if one had a guess $\beta \in [0, 1]$ indicating what percentage of the rows have corrupted RHS entries, they could have the algorithm calcualte the residual and only include the $q \in [0, 1]$ of the entries with the lowest residuals. This method is called the Quantile Averaged Block Kaczmarz algorithm which is used in [2]. We implement it in this section.

Quantile Row Selection: This method will calculate the residual for all the rows of the system, then it will take the q-quantile of the absolute residual and exclude any row with a residual entry of higher magnitude than the computed quantile. Finally, the method will select the block of remaining rows with the highest residual entries. This method is best used when corrupted data is present.

While earlier the computational advantages of random row selection were displayed in Figure 3, in [2] the convergence proof requires that the full residual is calculated in order to achieve convergence in theory. However, the authors show empirical evidence that they may take a sample of the rows and calculate the residual of that sample instead to save computational time. We implement this variation as well.

Sampled Quantile Row Selection: This method will take a randomly selected sample of the system (Note: [2] varies the sample size in their experiments, but we have chosen to make the sample a fiex percentage of the size of the system for the sake of simplicity.) Then it functions the same as quantile row selection for the sample except that every row that passes the quantile test is iterated upon. This choice was made to simplify comparisons between the two quantile versions so far. This method is best used when obtaining the full residual is not computationally feasible. Like [2], we assume in our experiments that corruption happens at random (this may not be true in practice) so the sampling should be random as well. This creates one more problem that needs to be fixed: while the percentage of corrupted rows in a sample of the system should be equal to that of the whole system on average, if too many specific iterations sample rows such that the corruption level is greater than in the whole system, it can cause corrupted rows to pass the quantile test and prevent convergence. To allow this version to consistently converge, we patched the code so that the algorithm will skip iterations where the norm of the sampled residual is significantly higher than the norm of the previous sampled residual (and therefore indicates that corrupted rows passed the quantile test). We acknowledge that this may have affected our results, but time constraints disallowed us from solving this problem more robustly as of the end of the REU.

Aside: In [2], the authors explain and show that a projected iteration method would not be suitable for corrupted data problems. This is because, in any given iteration, the corrupted rows may not get caught by the quantile test. So, if a corrupted row passes the test and the iteration type is projected, the current iteration will be projected onto a corrupted hyperplane which can prevent convergence. For that reason, we do not experiment with projected methods in this section.

6.2 The Effect of Degree Level on CPU Time and Iterations Taken

Both the quantile and sampled quantile row selection methods were used to solve randomly generated, overdetermined, consistent systems that were corrupted by setting random entries of the RHS of the system to 0. CPU Time and Iterations taken were measured against the number of rows in the matrix where the quantile was $q = 0.85$ the block size of the quantile version is taken to be the block size of the sampled quantile version, the number of columns in every system was 100, the relative tolerance was 10^{-10}, and the maximum iterations allowed was 40000 (Note: because the absolute size of samples, quantiles, and numbers of corrupted rows all scale as degree level increases, block size is never constant). Each algorithm was provided the actual solution for stopping criteria. The degree level was measured at exponentially discrete points by powers of 2, and the number of runs each data point averages is inversely proportional to the degree level. The results are presented in Figure 16 where we can see that the number of iterations trends downward as the number of rows increases. This is likely due to the nature of the system: because it is always consistent, the increasing block size means that the blocks are including more non-corrupted rows in their calculations as the number of rows increases, and, thus, each iteration is leveraging more information which allows for less total iterations. CPU Time has a general upward trend which is expected given that larger amounts of information take more time to process. However, the initial decrease is the exception to this trend. This is likely because the earlier systems are closer to being underdetermined which the algorithms are not designed to solve. We also see that the sampled version of the quantile method can improve the speed of convergence, but the proportionality of the block size may have ensured that it always performed better than the normal version, so it is empirically unclear when each method would be preferable.

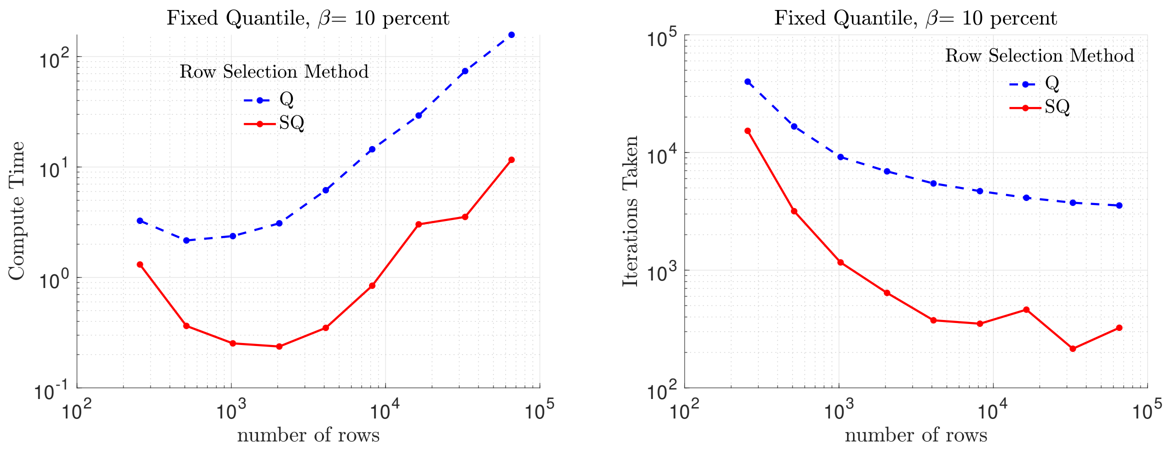


Figure 16: CPU time in seconds (left) and Iterations (right) against number of rows

6.3 Adaptive Quantile and Demo

Up to this point we have assumed that our initial guess for the level of corruption was accurate. This may not be the case: in cases where the corruption level is underestimated, corrupted rows will pass the quantile test and prevent convergence, and in cases of overestimation, the convergence rate is needlessly slowed. This next method seeks to alleviate these problems.

Adaptive Quantile: This method functions the same as QABK with two additions: if the norm of the current residual is signifigantly larger than that of the previous one, it is taken as a sign that corrupted data passed the quantile test, makes the quantile threshold lower and skips the iteration. Conversely, if the norm is signifigantly smaller than its predecessor, it makes the quantile threshold higher to speed up convergence. This method may be useful if a precise estimate for the corruption level is not available.

Aside: The thresholds for the adaptor have been arbitrarily decided in these experiments, mainly due to lack of time. We acknowledge that this version is not as robust as it could be otherwise could be, and we leave improvements upon the adaptor for future work.

The adaptive and regular versions of Quantile ABK were tested on a random consistent system of 100 rows and 10 columns. Random entries of the RHS were corrupted by setting them to 0. Relative error, $\|x^{(k)} - x\|/\|x\|$, where x is the exact solution, was measured against CPU time and Iterations taken where the starting quantile was $q = 0.9$, block size was 10, and the number of iterations was 10000. The results are presented in Figure 17 where we can see that the standard quantile method fails to converge in 10000 iterations. This is expected as the starting quantile underestimated the level of corruption, so it included too many corrupted rows in its iterations to possibly converge upon the correct answer. Meanwhile, the adaptive version demonstrates its ability to detect abnormally large increases in the

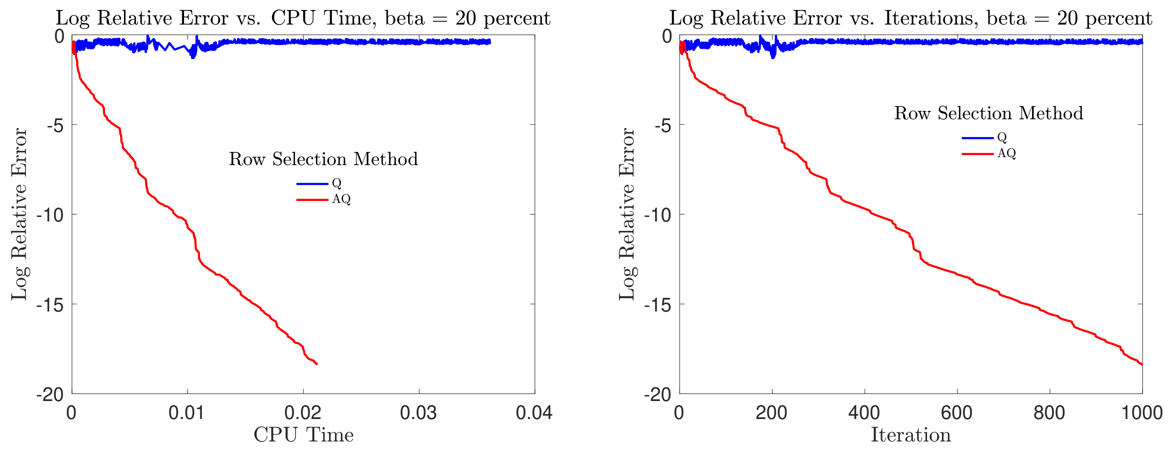


Figure 17: Log Relative Error against CPU Time in seconds (left) and iteration (right)

residual's norm compared to the previous iteration, and then lowers its quantile to avoid including corrupted data. Thus, it converges in the allowed iterations. This shows that an adaptive version of the quantile method is sometimes desirable.

7 Conclusion

In this paper, we provided the mathematical basis for the Kaczmarz method and its variants, explained and experimented with multiple different row selection and iteration methods, and extended our scope to include variations that deal with inconsistent systems, improvements upon alpha, ill-conditioned matrices, and corrupted data. Whichever method is best will depend on what specific scenario the user encounters in their endeavors. Nonetheless, the Kaczmarz method proves an effective tool for approximating solutions to linear systems, and, with improvements mentioned in this paper or beyond, it also proves versatile.

References

[1] N. Bell, L. N. Olson, and J. Schroder. PyAMG: Algebraic multigrid solvers in Python. *Journal of Open Source Software*, 7(72):4142, 2022.

[2] L. Cheng, B. Jarman, D. Needell, and E. Rebrova. On block accelerations of quantile randomized Kaczmarz for corrupted systems of linear equations. *Inverse Problems*, 39(2):024002, 2022.

[3] R. Falgout. An introduction to algebraic multigrid. *Computing in Science and Engineering*, 2006.

[4] S. Kaczmarz. Angenüherte auflösung von systemen linearer gleichungen. *Bull. Int. Acad. Polon. Sci.*, 35:355–357, 1937.

[5] A. Katrutsa and I. Oseledets. Preconditioning Kaczmarz method by sketching. *arXiv preprint arXiv:1903.01806*, 2019.

[6] I. Necoara. Faster randomized block Kaczmarz algorithms. *SIAM Journal on Matrix Analysis and Applications*, 40(4):1425–1452, 2019.

[7] T. Strohmer and R. Vershynin. A randomized Kaczmarz algorithm with exponential convergence. *Journal of Fourier Analysis and Applications*, 15(2):262, 2009.

[8] L. N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition.* SIAM, 2019.

[9] A. Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015.

[10] D. P. Woodruff. Computational advertising: Techniques for targeting relevant ads. *Foundations and Trends® in Theoretical Computer Science*, 10(1-2):1–157, 2014.

[11] J. Xu and L. Zikatanov. Algebraic multigrid methods. *Acta Numerica*, pages 591–721, 2017.

[12] Y. Zeng, D. Han, Y. Su, and J. Xie. Randomized Kaczmarz method with adaptive stepsizes for inconsistent linear systems. *Numerical Algorithms*, pages 1–18, 2023.