Simulation of Approximate Computing Applied to Numerical Methods

New Mexico
Supercomputing Challenge
Final Report
April 1, 2013

Team 48
La Cueva High School

Team Members:

Alexandra Porter

Teacher:

Samuel Smith

Table of Contents

Simulation of Approximate Computing Applied to Numerical Methods 1

1.0 Abstract 1

2.0 Problem Statement 2

3.0 Contribution 2

4.0 Background 3

5.0 Theory 6

6.0 Procedure 9

6.1 Basic Model 9

6.2 Binary Model 10

6.3 Complex Model 11

7.0 Results 14

7.1 Basic Model 14

7.2 Binary Model 15

7.3 Complex Model 16

8.0 Conclusions 42

9.0 Future Work 43

10.0 Appendices 44

10.1 Works Cited 44

10.2 Code 45

10.3 Data for Figures 5 and 6 109

Tables	Page

1. Comparison of Error Locations 14
2. Error when Introduced to Addition Operations 15
3. Standard Deviation of Error when Introduced to Addition Operations 15
4. Average Iterations to Converge: Error Added on 6 of 12 Operations 20
5. Average Iterations to Converge: Error Added on 8 of 12 Operations 20
6. Average Iterations to Converge: Error Added on 10 of 12 Operations 21
7. Average Iterations to Converge: Error Added on 12 of 12 Operations 21
8. Average Iterations to Converge (Random Matrices) 24
9. Standard Deviation of Iterations to Converge (Random Matrices) 24
10. Average Iterations to Converge (Constant Matrix) 27
11. Standard Deviation of Iterations to Converge (Constant Matrix) 28
12. Average Iterations to Converge Varying Matrix Size 31
13. Standard Deviation of Iterations to Converge Varying Matrix Size 32
14. Average Iterations to Converge with Redundancy: 1% Error Probability 35
15. Average Iterations to Converge with Redundancy: 5% Error Probability 35
16. Average Iterations to Converge with Redundancy: 10% Error Probability 36
17. Average Number of Iterations to Converge with Preconditioning 38
18. Effect of Error on Average Number of Iterations to Converge with Preconditioner 39

Figures | **Page**

1. Flowchart for Conjugate Gradient Solver with Error Introduced 9
2. Representation of Single Precision Floating Point Numbers 10
3. Complex Model Architecture 12
4. Complex Model Class Diagram 13
5. L2 Norm of the Residual Vectors from Iterations 0 to 50 17
6. L2 Norm of the Residual Vector from Iterations 0 to 500 18
7. Average Iterations to Converge Varying Places Error is Introduced 22
8. Number of Iterations as Error Increases: Random Matrices 25
9. Number of Iterations as Error Increases: Random Matrices 26
10. Number of Iterations as Error Increases: Constant Matrix 29
11. Number of Iterations as Error Increases: Constant Matrix 30
12. Iterations as Matrix Size and Error Increase 33
13. Effect of Operation Redundancies 37
14. Effect of Preconditioning the Matrix on Iterations to Converge 40
15. Effect of Error on Preconditioned Matrix 41

Equations **Page**

1. Equation Solved by Conjugate Gradient Method 6
2. Vector **x** as a Linear Combination of Vectors 6
3. Substituting **x** into Equation 1 6
4. Equation 3 Multiplied by $\mathbf{p}_i^T$ 6
5. Equation for α_i 6
6. Substitution for **b** in Equation 5 6
7. Equation for $\mathbf{x}_n$ 7
8. Residual vector $\mathbf{r}_0$ 7
9. Iterative α_k 7
10. Iterative $\mathbf{x}_k$ 7
11. Iterative Residual 7
12. Iterative Residual Substituting **b** 7
13. Beta for Calculation of Search Direction 7
14. Iterative Search Direction p_{k+1} 7
15. Preconditioned Equation to be Solved 8
16. Equation Solved by Conjugate Gradient Method 8

1.0 Abstract

The development of increasingly powerful processors is reaching physical limitations when high precision is demanded. However, relaxing the accuracy requirements has the potential to decrease power consumption. Inexact processors have been proposed for image and audio processing, where some error is acceptable because it is imperceptible to humans. However, their potential use has not been as extensively explored for computationally intensive scientific applications. Iterative methods for the solution of systems of equations are promising because the solution can be reached in the presence of some error. The purpose of this project is to simulate and measure the effects of using such processors in combination with traditional processors in the implementation of numerical methods for the solution of a system of equations.

In this project, the conjugate gradient method for the solution of a set of linear equations is implemented in Java. Three models are created to determine potential specifications and the most effective utilization for an inexact processor used in a numerical solution.

The first model simulates the inexact processor by adding between zero and one percent error to results of high-level operations where error is most tolerated. The second model simulates the introduction of error on a binary level. A probability of error is introduced in the least significant bits of selected binary operations. The final model combines the binary simulation with parallelization of simulated inexact processors while an accurate processor controls the inexact processors and completes the more critical operations. This model simulates the

design of a computer that utilizes inexact processors.

2.0 Problem Statement

As more powerful computing power is demanded of devices ranging from exascale supercomputers to mobile devices, computer hardware is reaching limitations in size, speed, and power consumption. However, architectures incorporating approximate hardware hold the potential to overcome these limitations if software can be designed accordingly.

3.0 Contribution

This project investigates the application of approximate hardware to an iterative numerical method, specifically the conjugate gradient method for the solution of a system of equations. The effects of error are tested and architecture for incorporating exact and inexact hardware is simulated.

4.0 Background

As more processing power is demanded of computers, microchips are reaching limits of size and capacity that prevent grater capabilities. One way of ensuring improvements continue is to relax precision requirements and trade accuracy for greater benefits in speed, area, and power consumption. Inaccurate processors have been proposed and tested on applications in image and audio processing. However, they may also have potential in scientific applications where huge numbers of calculations are necessary. Exascale computers, with 10s to 100s of thousands of processors, have great potential for research but have physical limitations of speed and energy consumption [1]. However, the use of inexact processors has the potential to significantly improve the function of these massive computers. On a smaller scale, inexact processing can make mobile devices more cost effective and power efficient. For example, the I-Slate developed at the Rice-NTU Institute for Sustainable and Applied Infodynamics is a tablet which utilizes inexact processing in order to be feasible for use in rural India.

There are a number of ways in which an inaccurate processor can compromise accuracy to achieve greater power efficiency and processing speed. One of these methods, called “pruning”, removes rarely used parts of the circuit to reduce power consumption [2]. At the Rice-NTU Institute for Sustainable and Applied Infodynamics, prototypes were created combining pruned and traditional circuits. In 2001 the pruned circuits were predicted to cut costs by reducing both physical area and power consumption compared to higher fidelity processors [2].

Error was about 8 percent, and the new processors were applied to hearing aids, in which final error is imperceptible to the user and therefore acceptable [2]. In more recent tests, the chips were shown to cut energy demands by a factor of 3.5 while only having about 0.25% error [3].

Another method of introducing error to save energy is confined voltage scaling [3]. Lingamneni et al propose a "cross layer co-design framework" in which voltage scaling is combined with pruning to allow error in the result [4]. By reducing the voltage, power consumption is reduced in exchange for some of the gains obtained through pruning and the processor architecture [4]. Error was varied from 10^{-4}% to 25% and resulted in gains in both area and delay, meaning the chips would require less material and would provide faster processing [4]. As voltage was decreased, these gains were shown to decrease, but by balancing voltage scaling and pruning, all aspects could be optimized [4].

Bates proposes low-precision high dynamic range (LPHDR) processing elements, which are combined in parallel as well as with a traditional exact processor [5]. By using logarithmic representation of numbers, range is preserved with fewer bits representing the mantissa, reducing accuracy by about 0.1% [6].

While there are multiple ways in which an inexact processor can be realized, the applications have so far been limited. Image and audio processing are most tolerant to error because human users are able to compensate for large amounts of error. Lingamneni et al demonstrate the results of their pruning through fast Fourier transforms on images where the final result has no visible differences at 0.54% error and is still recognizable at 7.58% error [4], [5]. However, this project

proposes that the application space for inexact processors may be expanded to solutions of systems of equations in which more accurate solutions are calculated through use of iterative solution methods. In theses methods, iteration may be able to overcome the results of error so that the algorithm would reach a reasonably accurate solution for the system. In this project, one such iterative method, the conjugate gradient, will be explored.

5.0 Theory

The conjugate gradient method for the solution of a system was selected to apply approximate computing because it is an extremely useful method with applications in many fields. Because the conjugate gradient is an iterative method, its use of iterations could theoretically overcome introduction of error.

The goal of the conjugate gradient method is to solve the equation

$$A\mathbf{x} = \mathbf{b} \qquad (1)$$

where A is an n x n matrix and **x** and **b** are vectors of size n. If a line search is first performed in the direction of some vector **u** then the next search direction **v** must be minimized without undoing the minimization of **u** [9]. The vectors **u** and **v** are defined to be conjugate with respect to A, or A-orthogonal, if $\mathbf{u}^T A\mathbf{v} = 0$. The solution **x** can then be defined as a linear combination of vectors,

$$\boldsymbol{x} = \alpha_0\boldsymbol{p_0} + \cdots + \alpha_{n-1}\boldsymbol{p_{n-1}}, \quad (2)$$

where $\mathbf{p_0} \ldots \mathbf{p_{n-1}}$ are n non-zero A-orthogonal vectors, representing the search directions, and $\alpha_0 \ldots \alpha_{n-1}$ are real numbers. Substituting into $A\mathbf{x} - \mathbf{b} = 0$

$$A(\alpha_0\boldsymbol{p_0} + \cdots + \alpha_{n-1}\boldsymbol{p_{n-1}}) - \boldsymbol{b} = 0. \quad (3)$$

Multiplying by $\mathbf{p_i}^T$,

$$\boldsymbol{p_i^T} A(\alpha_0\boldsymbol{p_0} + \cdots + \alpha_{n-1}\boldsymbol{p_{n-1}}) - \boldsymbol{p_i^T}\boldsymbol{b} = 0. \quad (4)$$

Then α_i can be written as

$$\alpha_i = \frac{\boldsymbol{p_i^T b}}{\boldsymbol{p_i^T} A\boldsymbol{p_i}}, \quad (5)$$

and substituting for **b**, $\alpha_i = \frac{\boldsymbol{p_i^T}(\boldsymbol{r_k} + A\boldsymbol{x_k})}{\boldsymbol{p_i^T} A\boldsymbol{p_i}} = \frac{\boldsymbol{p_i^T r_k}}{\boldsymbol{p_i^T} A\boldsymbol{p_i}}.$ (6)

This implies

$$\mathbf{x}_{\boldsymbol{n}} = \sum_{i=0}^{n-1} \frac{\boldsymbol{p}_{\boldsymbol{i}}^{T}\boldsymbol{b}}{\boldsymbol{p}_{\boldsymbol{i}}^{T}A\boldsymbol{p}_{\boldsymbol{i}}}\boldsymbol{p}_{\boldsymbol{i}}. \quad (7)$$

To apply this method iteratively, the solution begins with $\mathbf{x_0}$, which can be an initial guess of the solution or 0. From here, an initial residual vector is calculated,

$$\boldsymbol{r_0} = \boldsymbol{b} - A\boldsymbol{x_0}. \quad (8)$$

The initial search direction, $\mathbf{p_0}$ is then equal to $\mathbf{r_0}$. The scalar α_k is calculated for each iteration k

$$\alpha_k = \frac{\boldsymbol{r_k^T}\boldsymbol{r_k}}{\boldsymbol{p_k^T}A\boldsymbol{p_k}}, \quad (9)$$

so the next optimal vector, $\mathbf{x_{k+1}}$ can be calculated

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + \alpha_k\boldsymbol{p_k}. \quad (10)$$

The residual vector is then updated for the next iteration

$$\boldsymbol{r_{k+1}} = \boldsymbol{r_k} - \alpha_k A\boldsymbol{p_k}, \quad (11)$$

which is the equivalent of

$$\boldsymbol{r_{k+1}} = \boldsymbol{b} - A\boldsymbol{x_{k+1}}. \quad (12)$$

Finally, a new search direction, p_{k+1}, is calculated

$$\beta_k = \frac{\boldsymbol{r_{k+1}^T}\boldsymbol{r_{k+1}}}{\boldsymbol{r_k^T}\boldsymbol{r_k}} \quad (13)$$

$$\boldsymbol{p_{k+1}} = \boldsymbol{r_{k+1}} + \beta_k\boldsymbol{p_k}\,. \quad (14)$$

The scalar β_k is calculated using the Fletcher-Reeves formula, as shown above, and is then applied to determine the next search direction.

For the purposes of this project, singularity is prevented and success of the solver is guaranteed by creating matrix A to be symmetric positive-definite. To do this, a random matrix B of size n x n is created. A is then computed as $A = BB^T$. The

vector **x** is created with random values and multiplied by A to obtain the vector **b**. The conjugate gradient method receives A and **b** as input and solves for **x.**

The Jacobi preconditioner was also used to reduce iterations on diagonally dominant matrices. The goal of a preconditioner is to effectively approximate A^{-1} without the cost of actually calculating it because the system

$$A^{-1}(A\mathbf{x} - b) = 0 \quad (15)$$

will converge faster than

$$A\mathbf{x} - b = 0. \quad (16)$$

The Jacobi preconditioner approximates A^{-1} by taking the inverse of each value on the diagonal of A and assuming all off-diagonal terms are zero. This type of preconditioner is most appropriate for matrices that are diagonally dominant. The type of matrix used here was a blocked diagonal matrix in which smaller matrices are lined up along the diagonal. In order to strengthen the dominance of the diagonal, magnitudes of values on the diagonal were also increased relative to the rest of the matrix.

6.0 Procedure

6.1 Basic Model

To investigate the effects of error on the convergence rate of the conjugate gradient solver, three computational models were created and tested using Java. The simplest model is a conjugate gradient solver in which a random amount of error between 0 and 1% is added to the result of select operations, as shown in Figure 1. Adding error simulates an operation being completed by an inexact processor.

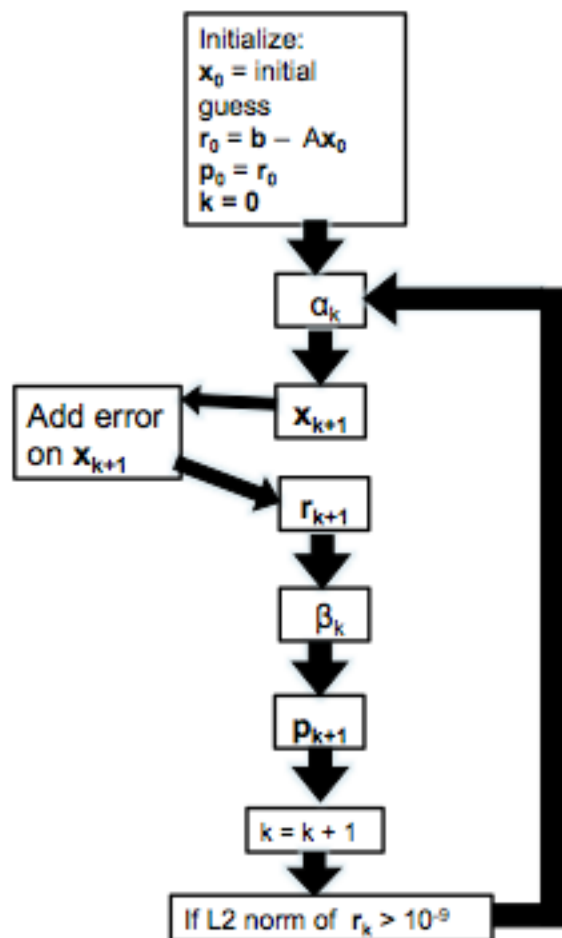

Figure 1 Flowchart for Conjugate Gradient Solver with Error Introduced

6.2 Binary Model

The second model similarly adds error, but does so through a binary simulation. The result of an addition or multiplication operation is converted to single precision floating point and a probability P of m bits being flipped is assumed, where m is a given number of least significant bits. To represent a number in single precision floating point with normalized form, one bit represents the number's sign, S, 23 represent the Mantissa, M, and 8 represent the exponent, E, so that the number is equal to $-1^S*(1+0.M)*2^E$ [8]. For example, if P= 0.1 and m = 10, each of the ten least significant bits has a 10% probability of being flipped, resulting in error. Figure 2 shows two examples of this. Twos-complement is used to represent negative numbers, such as the exponent for the second number. Introducing error in this way is different from utilizing lower precision because it is more accurate. Due to the probabilistic nature of the bit errors, numbers will frequently be recorded with more accuracy than if the bits allowing error were omitted entirely. This model is also unique in combining varying levels of accuracy, rather than having a uniform level of precision.

Sign	Exponent	Mantissa		Normalized Form: $-1^S*(1+0.M)*2^E$		
0	0 0 0 0 0 0 0 1	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	=	$-1^0*(1+0.75)*2^1$	=	3.5
1	0 1 1 1 1 1 1 0	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	=	$-1^1*(1+0.625)*2^{-1}$	=	-0.3125

Figure 2 Representation of Single Precision Floating Point Numbers

6.3 Complex Model

The final model combines the binary simulated error with parallelization to more completely model a computer utilizing inexact processors running in parallel, controlled by an inexact central processor. Calculations for each iteration are divided into two primary sections. Each section is performed simultaneously across a variable number of processors, over which the matrix is divided. For instance, if **x** is of dimension 12 and four processors are used, each processor does the calculations for 3 components of the vectors **x**, **p**, and **r**. On each processor, some calculations are exact while others are inexact, as shown in Figure 3. In the inexact calculations, an input number of least significant bits have an input probability of being flipped. The allocation of exact and inexact operations is varied in order to determine the most effective locations to introduce error. In each model, the L^2 (or Euclidean) norm of the residual is calculated and the method is terminated when this value is below 10^{-9}. While a majority of calculations are done in these blocks of operations, there are also a few that are done with accuracy in each iteration to combine the results from each processor.

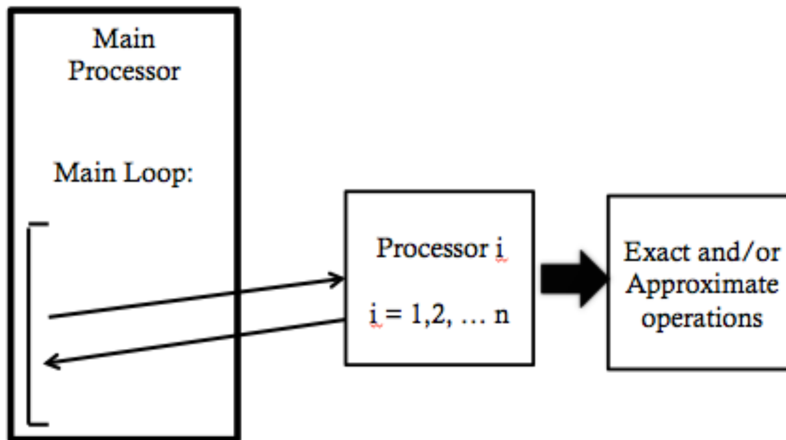

Figure 3   Complex Model Architecture

Figure 4 is the class diagram for the complex model. ParallelSolver is the main processor which uses MatrixMaker to create the system to solve. Block1Data and Block2Data are used in passing information to the processor. ParallelSolver creates instances of Block1Data and Block2Data on each iteration. These are then passed to Processor, which updates the information and returns the data block. BinaryWorker contains methods for completing addition and multiplication with error, with probabilities and allowances for bit flips as input. Relative to actual hardware, ParallelSolver is an accurate central processor, and each instance of the Processor class is an inaccurate processor whose capabilities are simulated by the BinaryWorker.

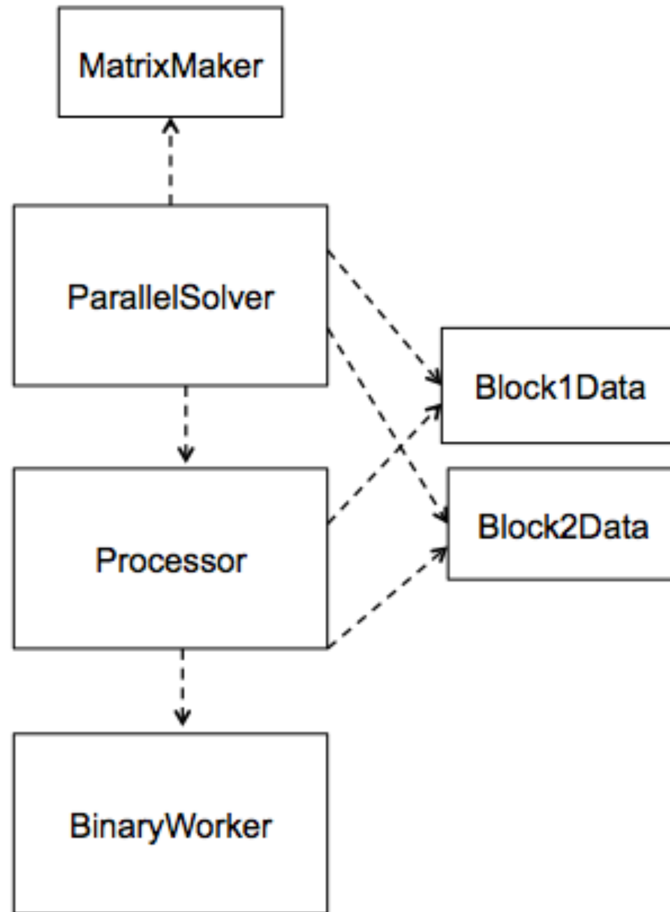

Figure 4 Complex Model Code Diagram

The complex model was also expanded to utilize a preconditioner. The preconditioner, M^{-1}, is calculated initially and then applied to the system by replacing $\mathbf{r}$, the residual vector, with vector $\mathbf{z}$ where $\mathbf{z} = M^{-1}\mathbf{r}$ at each iteration. The simulated inexact processor is utilized in the same way as before in each iteration with the addition of calculating $\mathbf{z}$, but it is also applied in the approximation of A^{-1}, which is M^{-1}, the preconditioning matrix.

7.0 Results

7.1 Basic Model

7.1.1 Comparison of Error Locations

Table 1 shows the average iterations to converge and standard deviation of number of iterations for 100 trials when error is added in the calculation of each of the listed vectors or scalars for a constant matrix. Error is added as a random number between 0 and 0.01 As shown, adding error on any operation increases the average number of iterations, but error in the multiplication result of matrix A and vector **p** increases iterations the most, whereas error in the calculation of the scalar beta has the least effect. This makes sense because the operation in the A**p** multiplication is called the most, but error in beta is incorporated only into a single final scalar so it has a smaller overall effect. Based on these results, the optimal places to add error in the complex model described in Section 6.3 were determined.

	No Error	Transpose Multiply	**p**	**r**	alpha	beta	A***p**
Average # Iterations	11.00	63.14	52.16	47.45	64.60	46.43	190.88
Standard Deviation of Iterations	0.00	6.21	3.64	2.92	7.09	3.63	115.63

Table 1 Comparison of Error Locations

7.2 Binary Model

7.2.1 Amount of Error Corresponding to Binary Error Proportional to Actual Sum

Tables 2 and 3 show the average of error and the standard deviation of error, respectively, when error is introduced on an addition operation. These results show that as the number of bits allowing error and the probability of an error increase, the average amount of error and the standard deviation in error increases. Because error is introduced on the final result of an operation, the results of error introduced on a multiplication operation are the same.

Average Error Proportion				
Bits Allowing Error	5%	10%	15%	20%
5	1.38E-07	2.47E-07	3.44E-07	4.99E-07
10	2.13E-06	7.99E-06	1.09E-05	1.28E-05
15	8.30E-05	2.59E-04	3.53E-04	4.87E-04
20	5.06E-03	7.45E-03	1.49E-02	1.63E-02

Table 2 Error when Introduced to Addition Operations

Standard Deviation of Error Proportion				
Bits Allowing Error	5%	10%	15%	20%
5	3.08609E-07	4.17212E-07	4.76004E-07	5.90272E-07
10	5.71088E-06	1.50998E-05	1.76759E-05	1.63579E-05
15	0.000278272	0.000489602	0.000508882	0.000531472
20	0.012804764	0.013810289	0.018610609	0.018875186

Table 3 Standard Deviation of Error when Introduced to Addition Operations

7.3 Complex Model

7.3.1 Effects of Error on the Norm of the Residual Vector

Figures 5 and 6 show the effects of a localized error on the L2 norm of the residual vector over a number of iterations. In each trial, error is introduced at 100% probability in 10 digits only during the first ten iterations, followed by iterations with no error. Rather than recover and reach a solution when error is no longer being introduced, the method begins to diverge in an oscillating pattern of increasing amplitude. Figure 4 shows the irrecoverable oscillation behavior in the residual L2 norm out to 50 iterations, whereas Figure 5 shows the behavior graphed over 500 iterations.

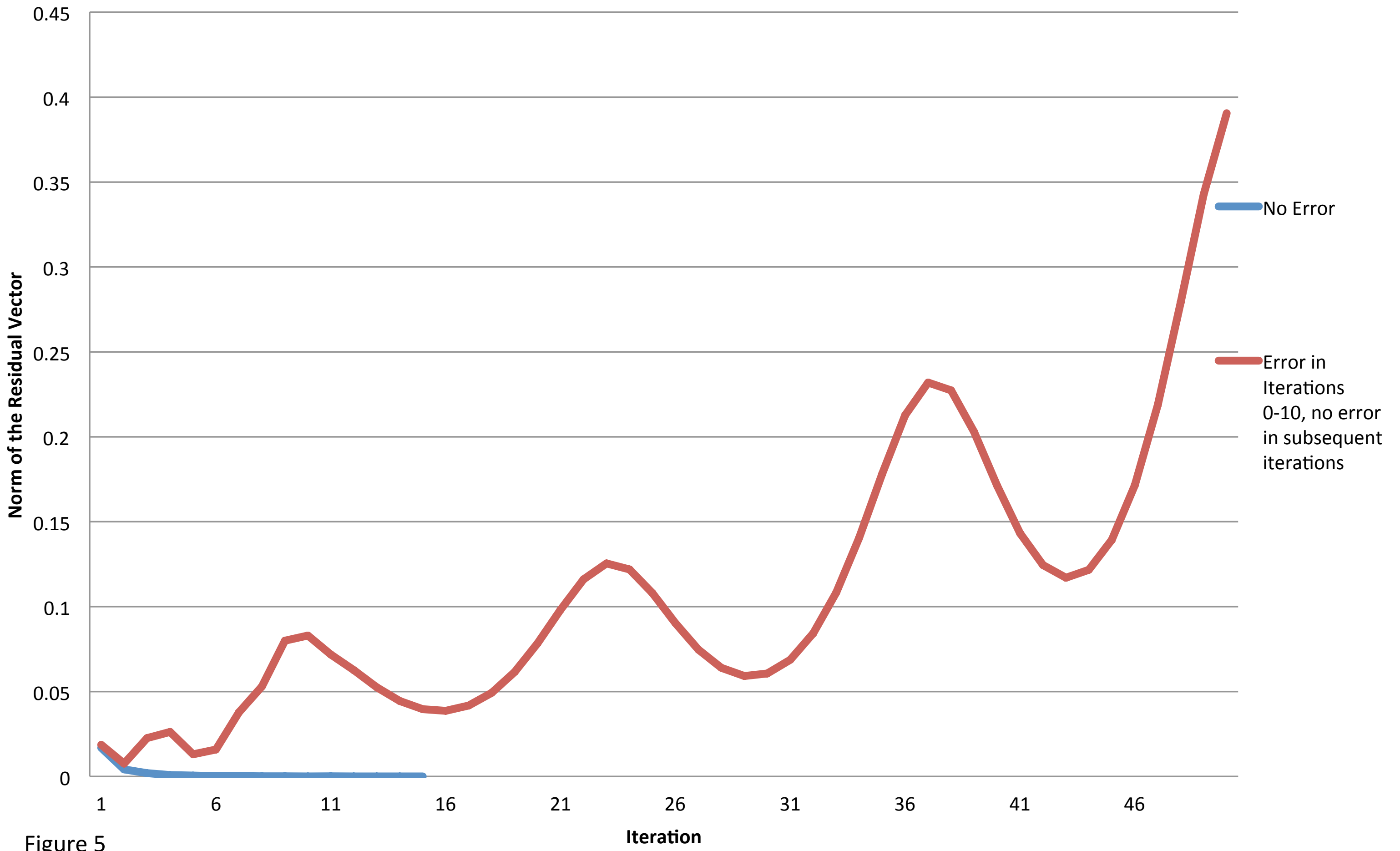

Figure 5

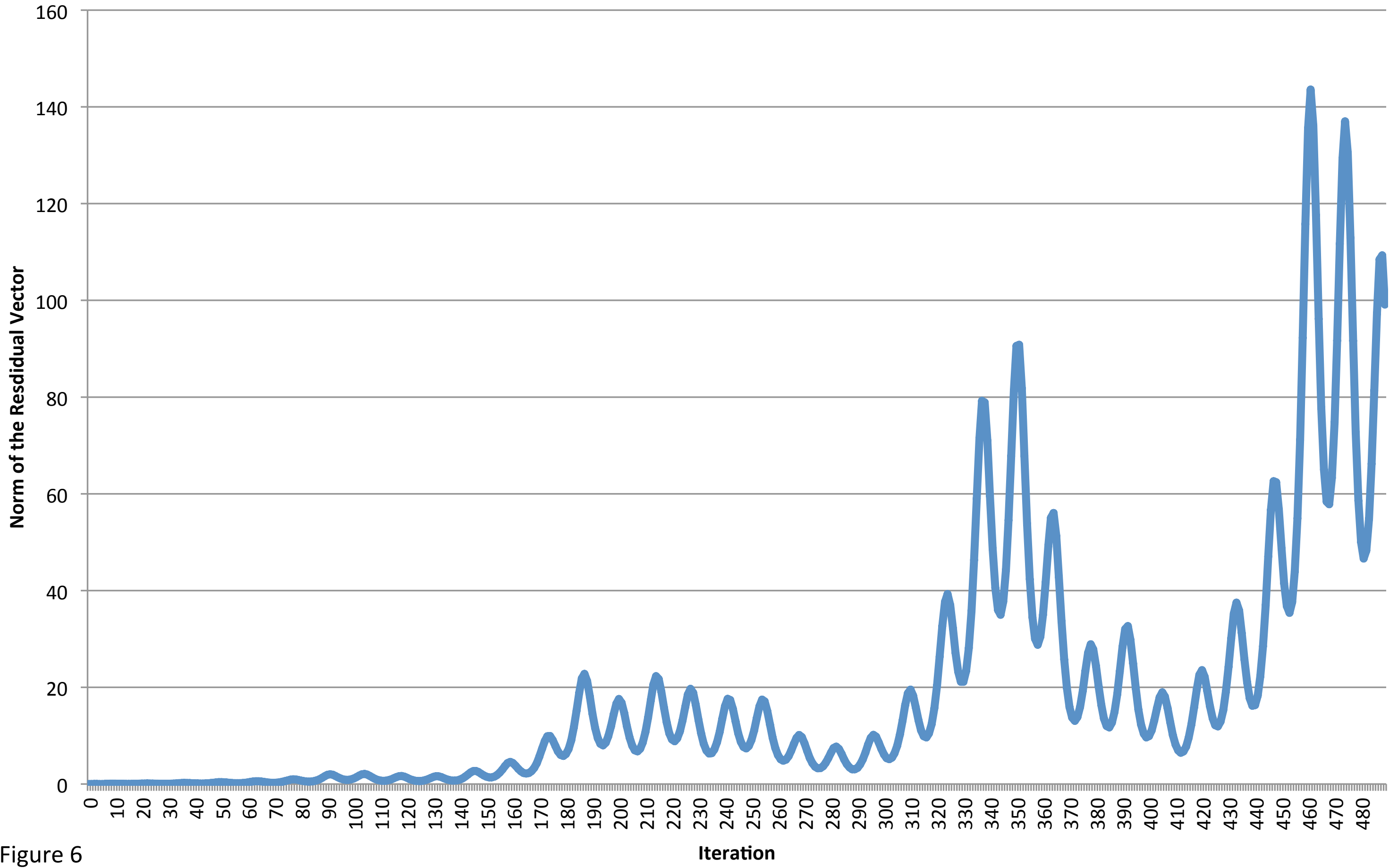

Figure 6

7.3.2 Average Iterations to Converge with Error Varying Number of Locations

Tables 4 through 7 and Figure 7 show average iterations to converge for random matrices as error is increased and the number of places error is added is varied. Each value represents the average iterations for 100 random 12 x 12 matrices. In the simulated inexact processor there are 12 locations, 6 additions and 6 multiplications, which can be calculated either exactly or with simulated error. 10 of the 12 locations are looped so that they the operation is repeated once for each value in the processor's portion of the entire vector **x**. For instance, if **x** is size 12 and there are four processors, then 10 of the 12 operations are repeated 3 times per iteration on each processor. The other two locations, in the A**p** multiplication, are repeated significantly more frequently, in this example 36 times per iteration on each processor. Table 7 shows that while using error on more operations increases gains in efficiency, adding error on all operations results in extreme increases in number of iterations and inconsistencies, as shown in Table 6, effectively preventing more than 10 bits allowing error. Error on 10 of 12 operations seems to be the optimal amount of error because the effects are not unmanageably large. The two operations completed with accuracy are those used in the multiplication A**p**, which, as shown in simple model tests, has the greatest detrimental effect because the addition and multiplication are placed within an inner loop and called more frequently.

Bits Allowing Error	Percent Chance of Error						
	1%	5%	10%	15%	20%	25%	30%
0	28.30	27.79	29.29	28.24	27.85	28.43	28.18
1	27.77	28.96	28.43	28.47	27.71	29.60	28.68
2	28.02	28.58	28.71	28.28	28.73	29.38	29.29
3	27.92	29.76	29.93	29.25	30.16	31.00	31.15
4	28.44	30.57	31.12	31.60	30.88	32.48	32.40
5	29.12	32.07	32.58	33.57	33.46	34.75	34.94
6	31.49	34.84	35.63	35.63	36.57	37.62	36.90
7	33.08	36.33	38.90	40.58	40.95	40.26	40.50
8	35.03	41.11	40.72	42.60	42.30	44.86	45.10
9	38.70	43.33	45.91	46.78	47.97	47.53	48.91
10	41.29	47.58	50.81	52.72	53.27	53.97	56.30
11	44.68	52.59	55.91	56.51	56.94	61.54	63.16
12	50.63	58.14	62.82	62.59	66.87	68.92	71.88
13	55.87	64.76	75.00	75.02	82.69	90.08	83.39
14	62.85	72.53	121.99	100.81	95.74	119.65	114.23
15	75.21	115.49	156.47	118.43	115.74	136.79	149.72

Table 4 Average Iterations to Converge: Error Added on 6 of 12 Operations

Bits Allowing Error	Percent Chance of Error						
	1%	5%	10%	15%	20%	25%	30%
0	31.64	32.14	30.82	31.51	31.27	31.92	31.26
1	31.91	31.38	32.03	31.99	32.37	31.09	32.20
2	31.44	31.33	31.74	31.61	31.65	32.19	32.31
3	31.34	31.63	31.30	32.17	32.57	33.52	32.91
4	31.62	32.62	33.52	34.13	34.88	34.59	35.49
5	32.78	33.83	35.53	37.23	37.27	36.92	38.60
6	32.96	36.06	38.50	39.88	40.12	40.44	40.65
7	34.51	39.79	42.19	43.24	47.29	46.46	43.82
8	38.32	49.03	48.27	47.58	46.84	52.98	49.12
9	43.09	49.30	53.43	51.99	53.55	54.94	53.62
10	46.96	58.42	55.36	60.56	59.17	59.10	61.67
11	50.39	62.16	92.27	76.99	70.94	69.33	71.34
12	55.32	72.28	84.76	86.81	80.99	95.41	133.44
13	65.58	88.20	99.01	127.45	139.32	101.29	105.95
14	71.61	110.83	190.72	126.95	205.02	136.26	163.05
15	91.11	148.88	152.63	178.40	252.15	204.89	199.18

Table 5 Average Iterations to Converge: Error Added on 6 of 12 Operations

	Percent Chance of Error						
Bits Allowing Error	1%	5%	10%	15%	20%	25%	30%
0	29.74	29.41	28.89	29.68	29.69	29.77	29.34
1	28.73	29.68	129.49	29.74	29.92	30.44	29.30
2	30.26	29.96	29.62	29.46	31.19	29.91	30.71
3	30.46	29.52	29.76	31.27	31.11	30.75	31.32
4	30.25	30.67	31.71	31.52	32.15	32.58	33.16
5	30.15	32.52	34.09	34.12	33.70	35.77	35.22
6	31.15	34.18	37.31	37.46	39.38	39.81	138.12
7	32.90	37.39	38.84	44.12	41.45	52.18	45.25
8	36.06	40.64	46.98	46.69	49.22	49.66	49.34
9	37.91	60.32	48.46	51.93	50.79	52.02	155.40
10	41.88	52.98	52.58	53.38	69.73	59.14	58.03
11	46.46	58.18	159.36	68.06	63.59	59.73	74.70
12	54.12	162.56	65.32	156.91	81.33	89.35	124.75
13	57.07	73.92	192.07	283.91	202.33	397.54	303.93
14	161.91	94.42	196.13	232.48	368.97	450.07	777.88
15	86.32	204.52	388.11	382.66	334.08	773.50	567.22

Table 6 Average Iterations to Converge: Error Added on 10 of 12 Operations

	Percent Chance of Error						
Bits Allowing Error	1%	5%	10%	15%	20%	25%	30%
0	36.59	44.02	36.99	36.08	43.27	43.10	39.26
1	37.87	138.25	138.54	39.16	138.06	38.98	138.22
2	237.61	138.64	44.39	40.70	151.36	45.16	55.37
3	140.37	42.03	166.03	143.82	144.36	147.23	224.5
4	40.40	144.42	168.92	143.24	119.71	345.83	164.82
5	42.51	259.47	149.66	151.72	193.00	223.73	90.09
6	46.86	52.37	156.01	152.47	200.43	425.79	158.26
7	48.00	198.48	179.84	597.11	620.14	478.04	247.08
8	350.28	319.81	387.81	548.84	391.84	540.56	695.32
9	262.82	587.39	491.93	737.79	945.92	546.91	765.27
10	152.49	304.95	1078.66	1206.55	1017.49	617.61	1302.83

Table 7 Average Iterations to Converge: Error Added on 12 of 12 Operations

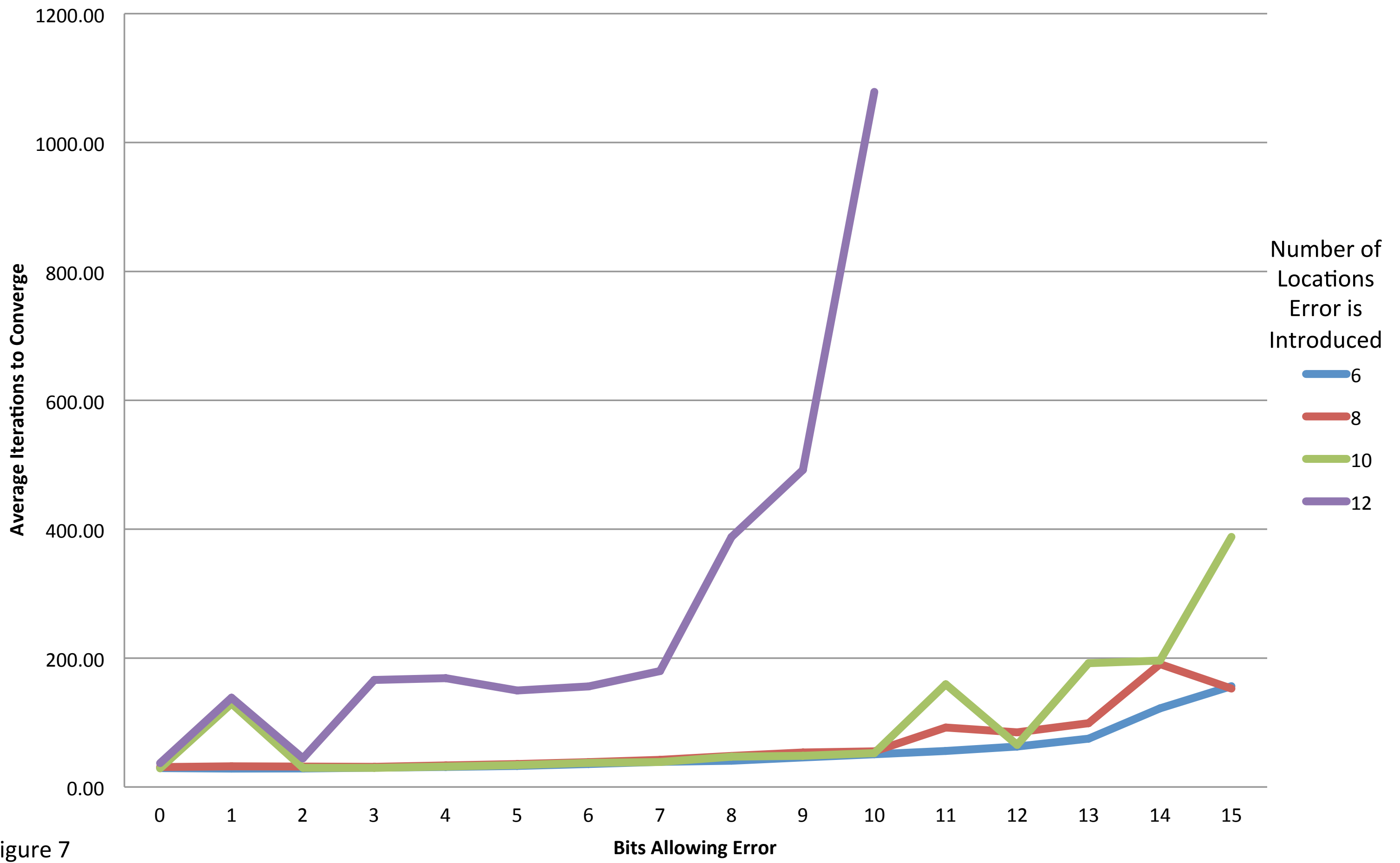

Figure 7

7.3.3 Effect of Error on Iterations to Converge

7.3.3.1 Random Matrices

As shown by Tables 8 and 9, increases in both probability and amount of error result in increases in the number of iterations required to converge and the variability in the number of iterations. Each average represents a mean of 100 trials of random matrices of size 12 x 12 with the corresponding error amounts introduced in 10 out of 12 possible locations in the program on the simulated inexact processor. Figures 8 and 9 show the same results as Table 8 in two different ways. Figure 8 clearly shows an approximately exponential relationship between the number of bits allowing error and the number of iterations required for each probability of error. Figure 9 shows there is some increase in iterations as the probability of error increases, but the correlation is much weaker than when increasing the number of bits. Figure 9 also shows that there is some inconsistency with the number of iterations when the number of bits allowing error is very high; the lines representing 12 and 13 bits are extremely inconsistent. Whereas this is partially due to the use of random matrices for each trial, some of the inconsistency is also due to the nature of probabilities assigned to the errors.

	Percent Chance of Error							
Bits Allowing Error	0%	1%	5%	10%	15%	20%	25%	30%
0	29.49	29.64	29.03	29.38	29.37	28.80	29.83	29.28
1	29.79	29.45	29.39	30.08	29.36	30.46	29.73	29.17
2	30.52	30.18	29.61	30.60	30.07	30.57	30.52	30.89
3	29.38	29.46	29.82	31.05	31.12	31.45	30.89	32.12
4	29.42	29.93	31.10	31.61	33.31	32.76	34.42	33.99
5	29.92	30.27	32.74	33.73	34.99	36.17	36.11	37.79
6	30.60	32.10	35.36	37.04	37.99	39.70	39.84	38.20
7	29.75	34.14	38.35	40.40	43.27	42.88	44.46	45.36
8	29.13	40.39	44.05	43.00	46.13	48.25	51.21	50.71
9	29.69	40.30	47.68	51.79	51.82	50.14	52.82	59.43
10	30.02	43.88	53.46	56.87	56.05	56.79	61.20	60.27
11	29.80	48.63	53.90	61.14	67.35	65.87	80.23	75.67
12	29.69	55.08	68.22	83.49	77.92	85.27	75.03	83.65
13	29.74	57.68	77.49	95.45	97.63	95.45	98.48	109.22

Table 8 Average Iterations to Converge

	Percent Chance of Error							
Bits Allowing Error	0%	1%	5%	10%	15%	20%	25%	30%
0	3.50	4.45	3.53	3.52	3.67	4.09	3.92	4.01
1	3.76	3.47	3.84	3.95	3.57	4.43	3.98	3.95
2	4.04	3.61	3.87	4.56	4.50	4.50	4.51	4.46
3	4.38	4.21	4.05	4.48	5.46	5.47	3.68	3.77
4	4.20	3.99	4.66	4.73	4.66	4.65	9.46	5.06
5	4.19	4.34	4.50	4.16	5.17	9.41	6.10	10.45
6	3.99	5.10	4.37	6.55	6.44	9.75	8.14	5.70
7	3.42	4.47	6.89	8.51	10.64	9.51	12.35	17.50
8	3.57	19.20	11.27	8.36	14.11	12.27	30.09	19.29
9	3.61	7.18	15.88	16.73	28.80	10.87	16.79	57.40
10	3.82	8.90	21.57	23.73	13.76	15.50	30.75	27.38
11	3.75	10.98	12.19	20.85	36.86	39.99	74.84	65.50
12	3.90	13.73	52.79	63.34	47.22	57.38	33.64	60.29
13	4.00	15.39	67.17	83.52	69.43	55.70	73.14	102.70

Table 9 Standard Deviation of Iterations to Converge

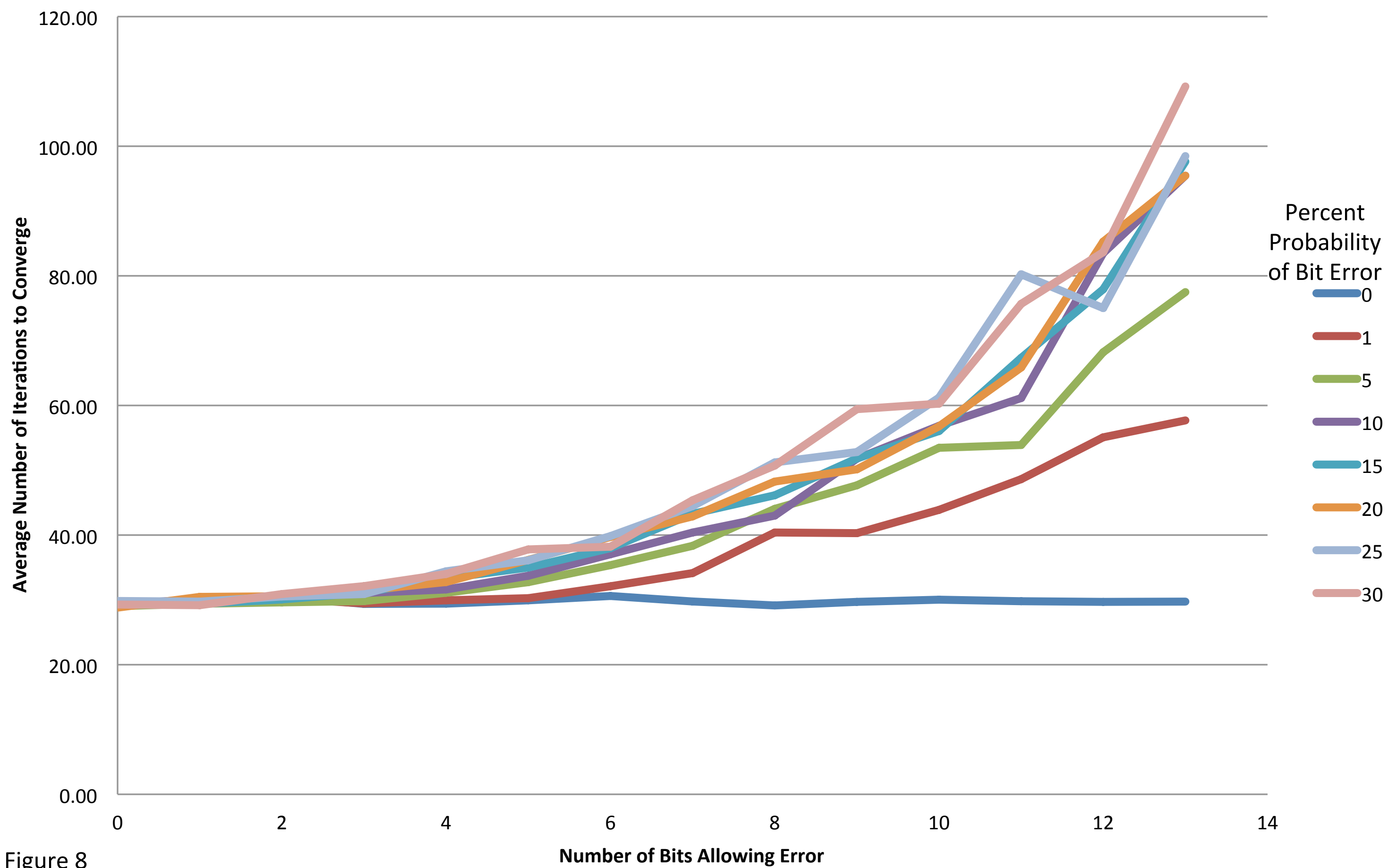

Figure 8

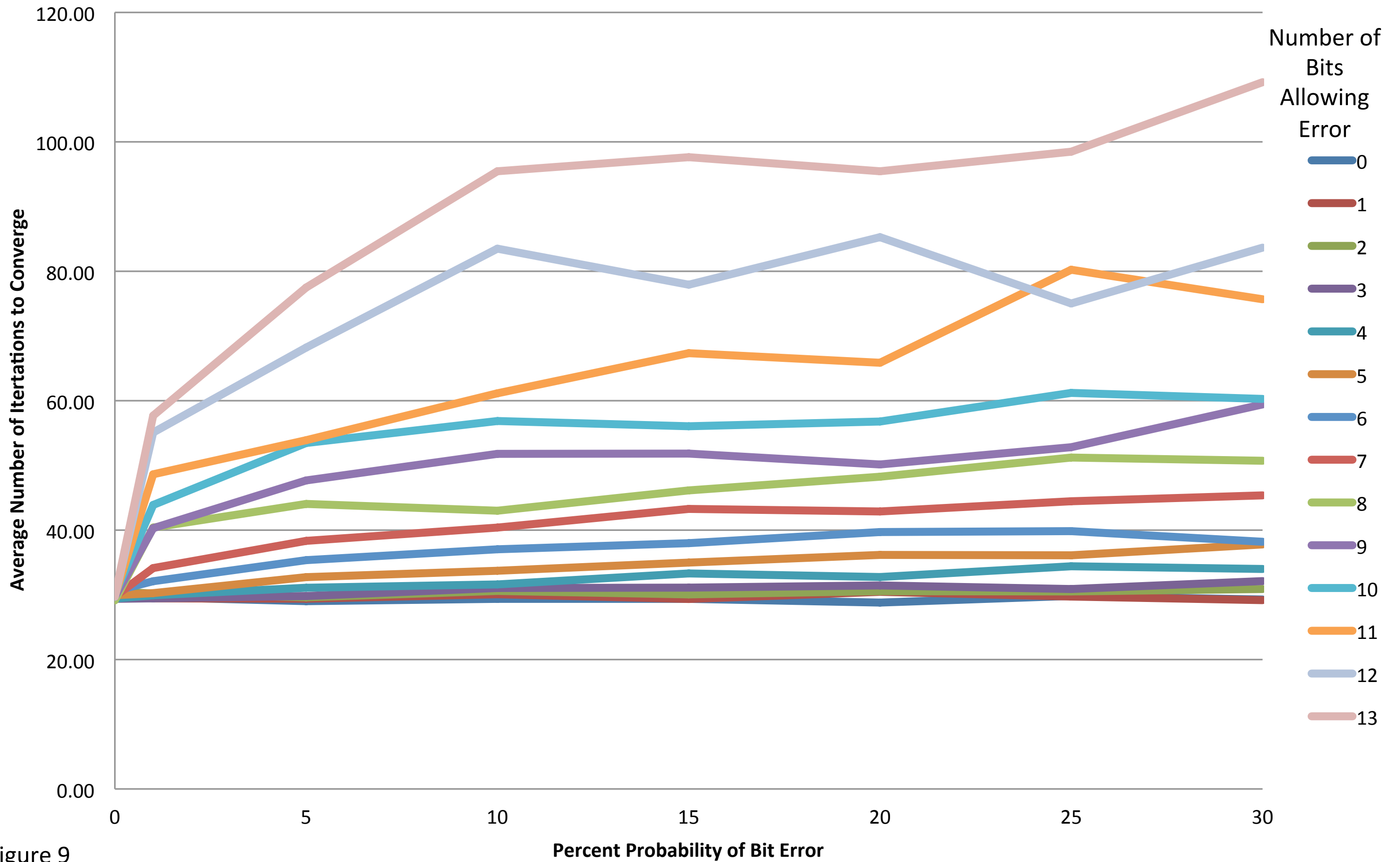

Figure 9

7.3.3.2 Constant Matrix

The results of a single matrix measured with varying probabilities and bits allowing error, as shown in Tables 10 and 11 and Figures 10 and 11, are very similar to the results of the same experiment with random matrices. However, using a constant 12 x 12 matrix reduced standard deviations and made trends in the numbers of iterations more consistent. This indicates than any given matrix will behave somewhat predictably under erroneous conditions; variance in the trials on random matrices is mostly due to differences from matrix to matrix.

	Percent Chance of Error							
Bits Allowing Error	0%	1%	5%	10%	15%	20%	25%	30%
0	30.00	30	30	30	30	30	30.00	30
1	30.00	29.86	30.04	29.78	29.91	30.09	29.93	30.09
2	30.00	29.72	30.09	30.04	30.4	30.58	30.63	30.53
3	30.00	29.95	30.43	30.86	31.08	31.53	31.40	31.67
4	30.00	30.44	31.25	31.87	32.01	32.63	32.80	33.08
5	30.00	31.07	32.56	33.22	34.15	34.55	35.24	35.43
6	30.00	32.16	35.25	36.9	38.12	38.78	39.54	39.77
7	30.00	33.82	38.34	41.46	42.14	42.97	43.95	44.05
8	30.00	37.23	42.75	44.07	45.9	46.4	47.12	47.67
9	30.00	40.16	46.03	48.44	50.05	50.95	51.68	52.47
10	30.00	45.32	50.84	53.21	54.52	56.04	55.67	56.68
11	30.00	48.9	54.86	57.56	57.97	60.06	60.24	60.37
12	30.00	53.89	59.21	61.88	63.2	65.33	67.23	68.87
13	30.00	57.82	64.92	70.38	76.23	79.37	79.73	82.86

Table 10 Average Iterations to Converge

	Percent Chance of Error							
Bits Allowing Error	0%	1%	5%	10%	15%	20%	25%	30%
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	0.00	1.01	0.96	1.12	1.06	1.14	1.20	1.03
2	0.00	1.14	1.06	1.33	1.33	1.16	1.28	1.23
3	0.00	1.15	1.31	1.14	1.67	1.37	1.38	1.18
4	0.00	1.42	1.37	1.32	1.42	1.58	1.60	1.62
5	0.00	1.34	1.45	1.90	1.93	2.25	2.49	2.66
6	0.00	1.76	2.50	3.18	3.67	3.72	3.66	3.89
7	0.00	2.06	3.72	3.52	3.61	3.83	3.44	3.38
8	0.00	3.81	4.01	3.83	3.86	3.91	3.35	3.58
9	0.00	3.83	3.97	3.86	3.75	3.58	3.26	3.59
10	0.00	3.67	3.80	3.81	2.47	2.73	3.12	2.44
11	0.00	4.29	2.96	2.99	2.89	2.37	2.75	2.99
12	0.00	3.07	3.36	3.23	4.18	4.96	5.49	5.68
13	0.00	3.01	4.20	6.00	8.08	8.48	7.88	8.46

Table 11 Standard Deviation of Iterations to Converge

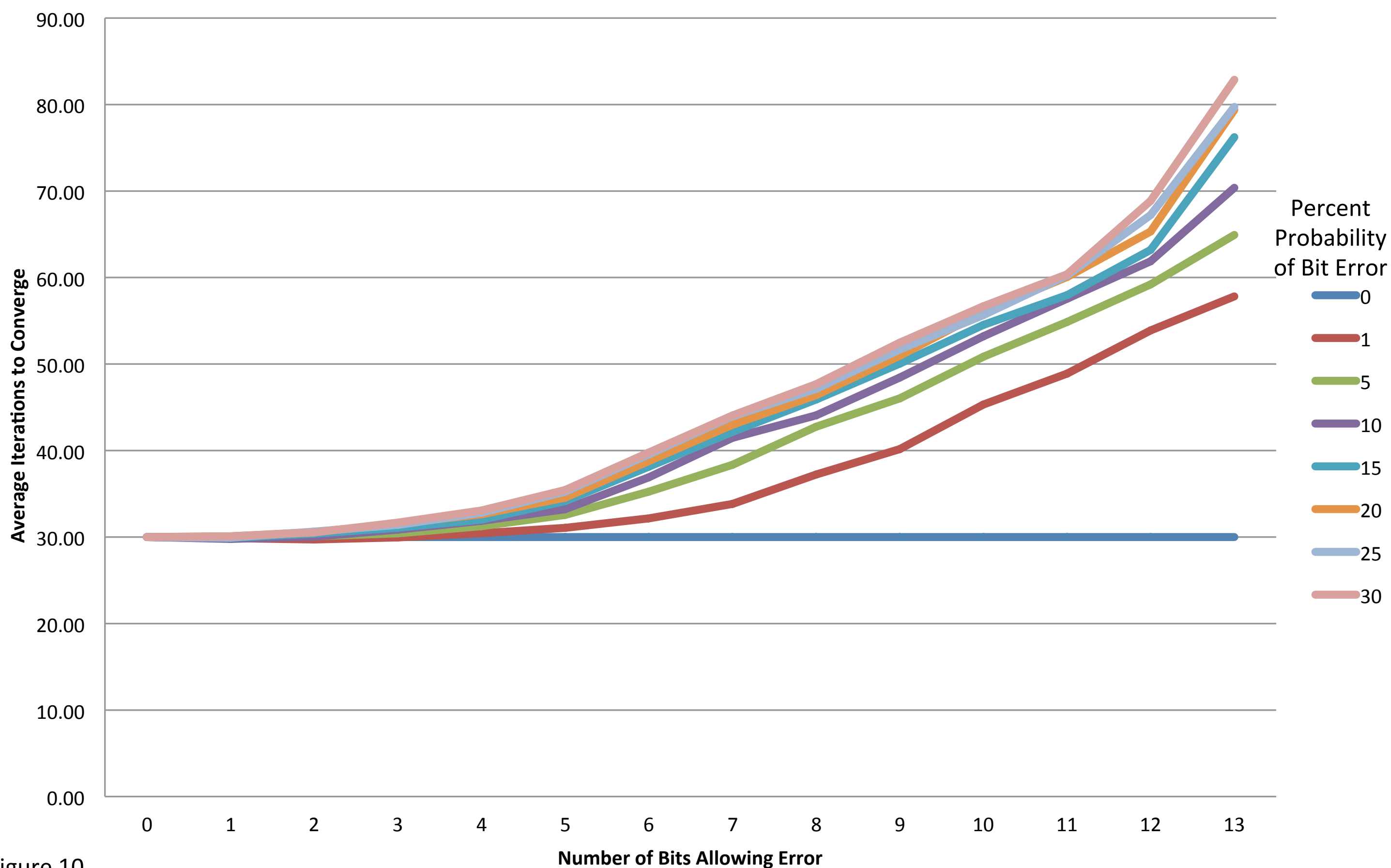

Figure 10

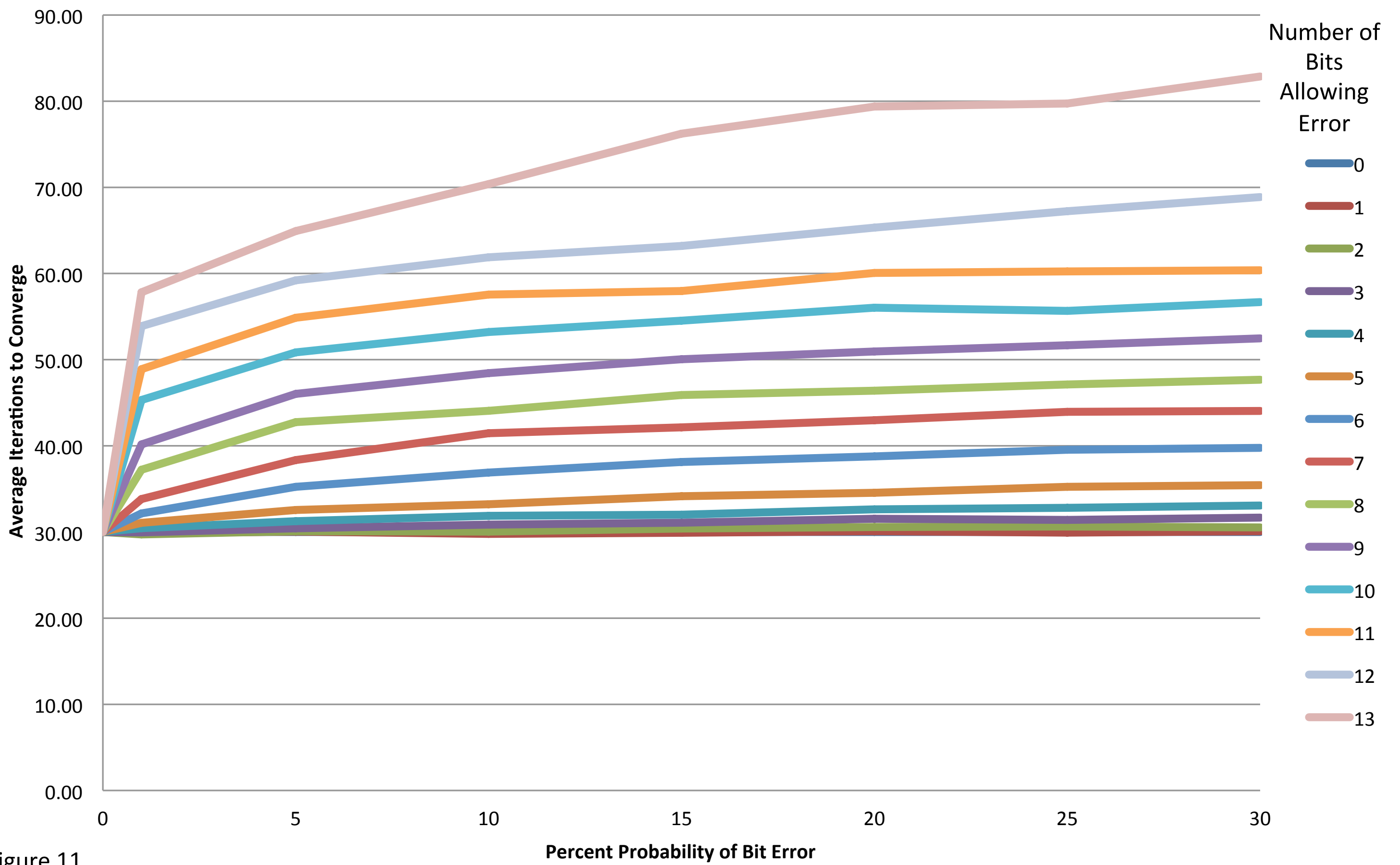

Figure 11

7.3.4 Effect of Matrix Size on Average Iterations to Converge

As shown by Tables 12 and 13 and Figure 12, matrix size increases the number of iterations to converge, again averaged over 100 random matrices, even when no error is introduced. The number of bits was increased because it was shown by Table 6 to have a greater effect than increasing error probability. Figure 12 shows that for a given matrix size, increasing error increases iterations, but larger matrices are more affected.

	Bits Allowing Error (10% Probability)												
Matrix Size	0	1	2	3	4	5	6	7	8	9	10	11	12
6	13.17	13.03	13.02	13.59	13.83	14.81	16.01	17.05	17.67	19.80	21.42	23.09	25.62
10	24.06	24.82	24.33	25.26	26.30	28.27	29.66	32.37	34.97	38.24	41.27	45.76	52.57
20	59.23	58.83	57.55	59.91	62.01	64.50	69.04	72.15	81.40	93.13	103.67	118.91	134.43
30	97.57	97.73	98.50	97.86	99.96	105.56	110.86	119.24	133.32	147.89	163.09	188.89	222.75
40	132.99	139.23	142.89	141.63	140.91	151.23	156.16	164.09	183.59	200.11	232.17	262.99	322.32
50	179.04	173.30	177.71	183.75	190.04	191.09	194.67	212.72	234.26	254.44	285.44	324.54	403.80
75	280.10	276.55	284.66	290.07	293.88	308.08	316.76	336.72	338.91	395.94	448.32	522.44	610.91
100	382.62	389.13	406.78	410.35	400.97	425.97	428.04	452.44	471.70	520.89	604.37	716.97	853.21

Table 12 Average Iterations to Converge Varying Matrix Size

	Bits Allowing Error (10% Probability)												
Matrix Size	0	1	2	3	4	5	6	7	8	9	10	11	12
6	1.22	1.44	1.60	1.56	1.52	1.66	2.38	2.50	2.90	3.23	5.02	5.65	6.65
10	2.86	2.95	2.75	3.18	3.13	3.63	3.79	5.01	5.62	7.10	7.95	12.58	23.69
20	8.24	7.46	9.07	7.71	8.86	8.87	8.96	9.98	13.81	20.34	25.51	28.35	99.52
30	15.20	14.32	12.84	15.40	13.86	17.75	15.44	18.70	19.34	27.02	30.76	35.76	62.98
40	20.77	24.24	22.91	24.50	24.13	24.41	26.31	23.15	30.68	36.64	53.88	60.16	140.51
50	28.50	28.51	32.48	32.48	29.47	32.97	35.12	35.22	39.19	42.32	54.31	73.98	177.81
75	52.97	47.42	51.99	52.68	55.53	62.81	59.60	57.46	56.32	77.49	91.93	115.50	148.95
100	73.74	75.30	76.58	78.68	85.62	75.42	87.51	96.27	101.51	98.04	109.61	152.22	252.46

Table 13 Standard Deviation of Iterations to Converge Varying Matrix Size

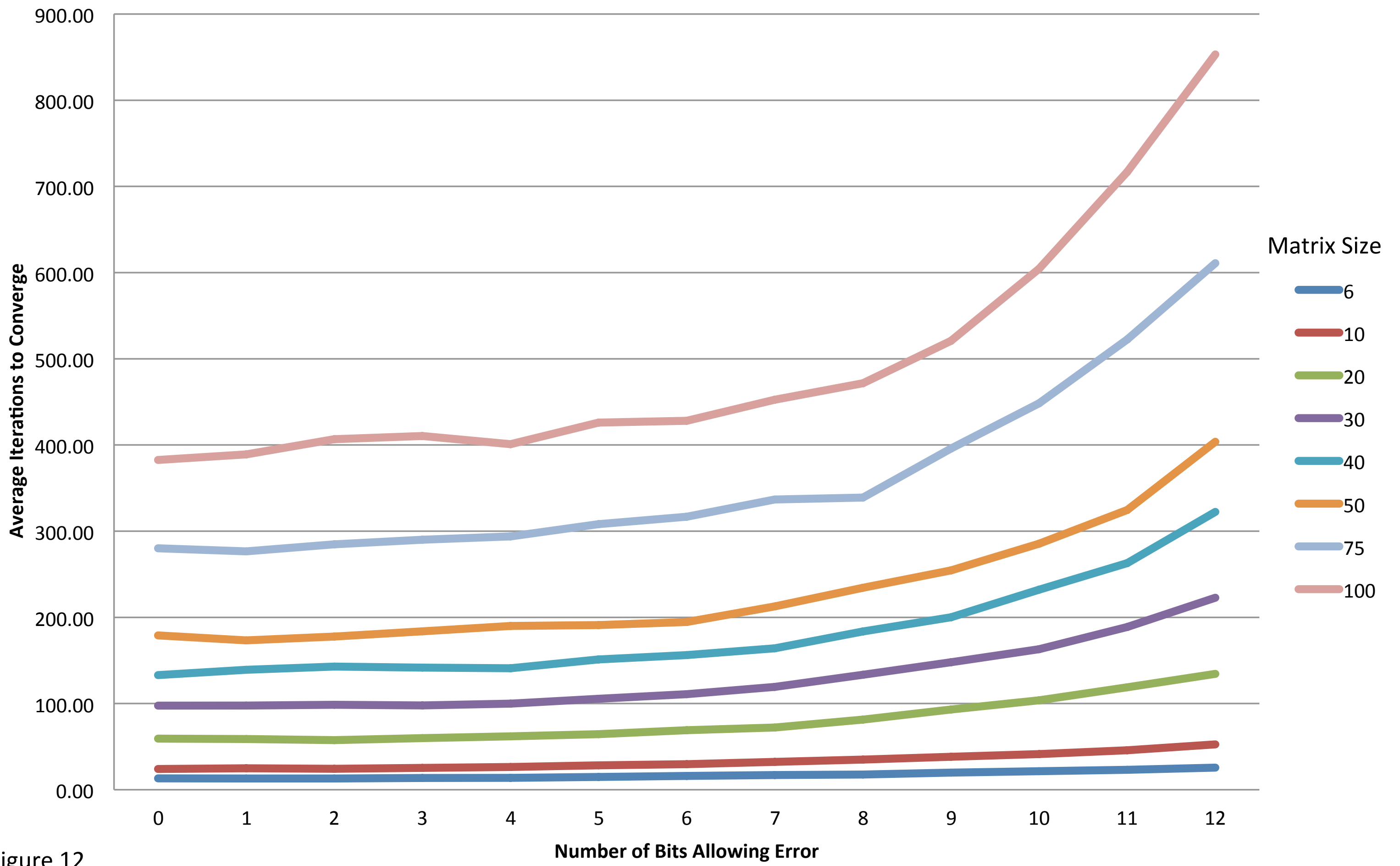

Figure 12

7.3.5 Effect of Operation Redundancies using Random Matrices

In order to reduce the number of iterations required to converge, redundancies were added on operations where error is introduced. If these operations are done erroneously they can be cost-effective in terms of power and processing and therefore improve overall efficiency. In a redundant operation, a value is computed erroneously three times and then the average of the two closest values is used. Tables 14 through 16 show averages for 100 trials of random matrices where error is introduced on 10 of 12 operations (excluding A**p**). As shown by Figure 13, increasing the number of operations on which the redundancy is implemented is only really effective at 1% error. This most likely stems from the fact that with a higher probability of error, all three values for a given operation will probably be significantly incorrect so the redundancy is less effective.

	1% Probability of Error	
Bits Allowing Error	No Redundancy	Redundancy on all 10 locations
0	28.30	29.93
1	27.77	30.23
2	28.02	28.55
3	27.92	30.09
4	28.44	29.21
5	29.12	28.96
6	31.49	29.65
7	33.08	29.78
8	35.03	30.63
9	38.70	30.85
10	41.29	33.30
11	44.68	35.25
12	50.63	37.97
13	82.55	41.46

Table 14 Average Iterations to Converge with Redundancy: 1% Error Probability

	1% Probability of Error	
Bits Allowing Error	No Redundancy	Redundancy on all 10 locations
0	29.03	28.55
1	29.39	29.09
2	29.61	29.40
3	29.82	29.88
4	31.10	30.58
5	32.74	30.58
6	35.36	32.99
7	38.35	34.57
8	44.05	36.88
9	47.68	40.00
10	53.46	45.98
11	53.90	52.79
12	68.22	55.17
13	77.49	61.99

Table 15 Average Iterations to Converge with Redundancy: 5% Error Probability

10% Probability of Error		
Bits Allowing Error	No Redundancy	Redundancy on all 10 locations
0	31.90	31.65
1	31.72	31.51
2	31.51	32.96
3	32.38	32.42
4	32.93	34.09
5	36.24	34.24
6	38.83	37.32
7	42.07	38.68
8	49.05	41.08
9	52.87	46.86
10	58.65	54.12
11	74.75	63.48
12	80.65	69.45
13	92.41	96.16

Table 16 Average Iterations to Converge with Redundancy: 10% Error Probability

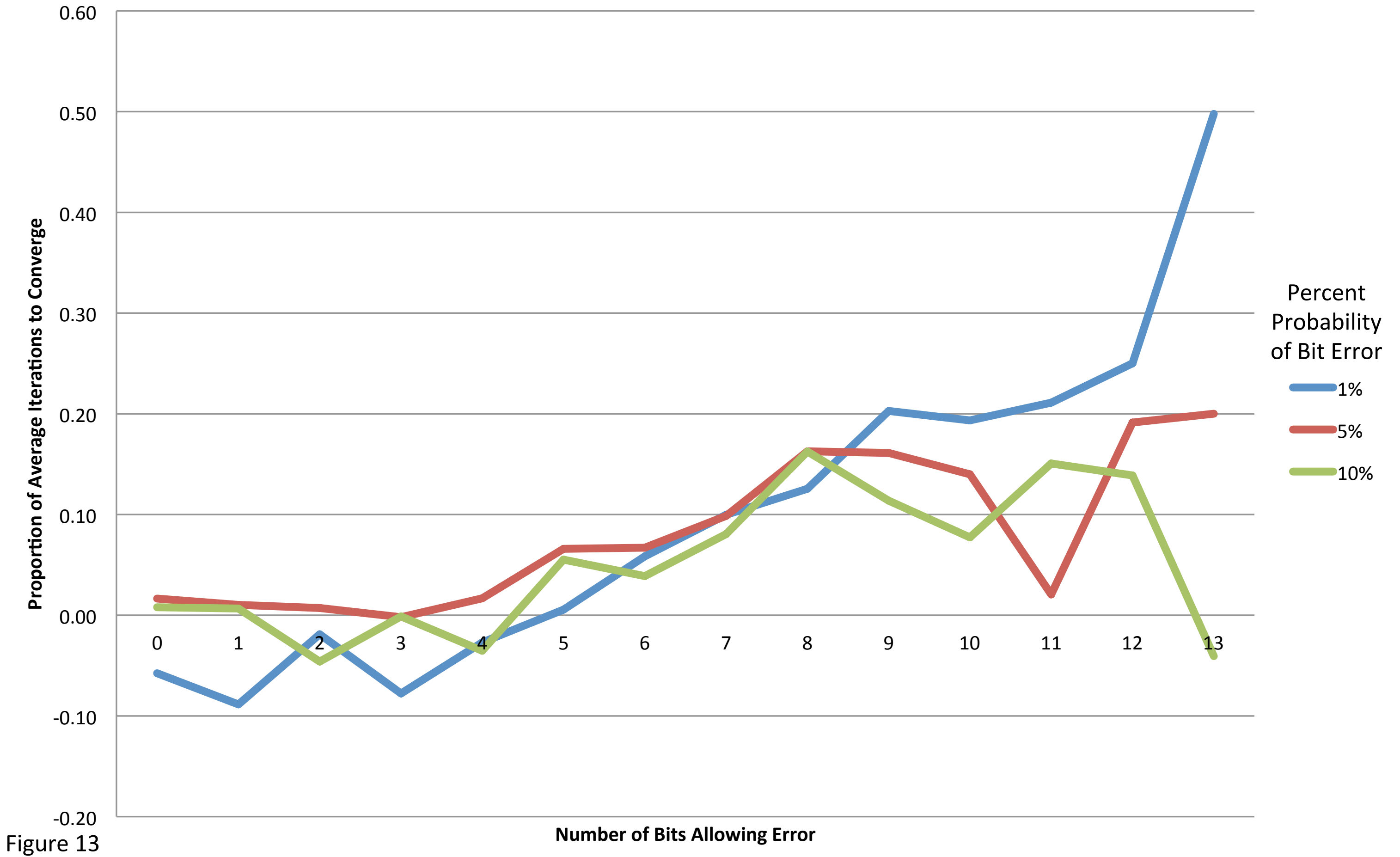

Figure 13

7.3.5 Effect of Preconditioning the Matrix

To further reduce the number of iterations and the overall number of calculations, the Jacobi preconditioner was applied. For diagonally dominant, blocked matrices, it was shown to be extremely effective in reducing the number of iterations, as shown in Table 17 and Figure 14. The convergence of the preconditioned system is highly variable, but Table 18 and Figure 15 show calculations for the preconditioner are tolerant to error overall, probably because the preconditioner itself is a form of approximation.

10% Probability of Error			
Number of Bits Allowing Error	Not Preconditioned	Preconditioned (Without Error)	Preconditioned (With Error)
0	25.83	6.76	6.53
1	25.97	8.06	9.91
2	26.29	8.82	8.56
3	26.87	5.70	9.53
4	27.41	9.10	6.26
5	28.48	13.66	6.65
6	28.64	6.49	8.20
7	31.72	5.38	6.06
8	32.89	6.44	6.39
9	33.96	6.13	12.20
10	34.59	5.98	5.53
11	40.60	6.20	5.33
12	41.97	6.00	6.94
13	46.52	6.35	8.50
14	54.02	5.99	8.71
15	59.90	6.82	8.45

Table 17 Average Number of Iterations to Converge with Preconditioning

	Percent Probability of Error				
Number of Bits Allowing Error	0	1	15	30	50
0	3.00	14.46	2.15	4.94	4.43
1	9.36	36.85	3.95	3.51	13.87
2	6.13	4.82	32.33	5.94	5.10
3	5.22	2.42	5.06	3.41	13.49
4	7.00	12.87	1.64	6.51	2.24
5	26.35	6.75	4.24	11.57	25.71
6	15.04	15.69	34.71	2.22	2.38
7	2.89	3.44	2.08	3.32	14.44
8	1.86	3.70	9.98	4.00	4.78
9	6.83	6.89	2.08	2.48	4.62
10	3.79	27.93	6.42	6.34	4.68
11	3.68	2.55	5.51	39.93	3.72
12	33.31	5.12	1.95	3.29	3.81
13	18.72	1.62	8.21	3.97	5.30
14	9.73	3.09	2.06	16.18	3.19
15	5.35	2.47	35.36	4.44	3.06
16	2.06	1.54	7.33	16.88	13.37
17	6.04	18.88	4.28	3.12	10.21
18	3.70	28.39	3.29	15.83	3.52

Table 18 Effect of Error on Average Number of Iterations to Converge with Preconditioner

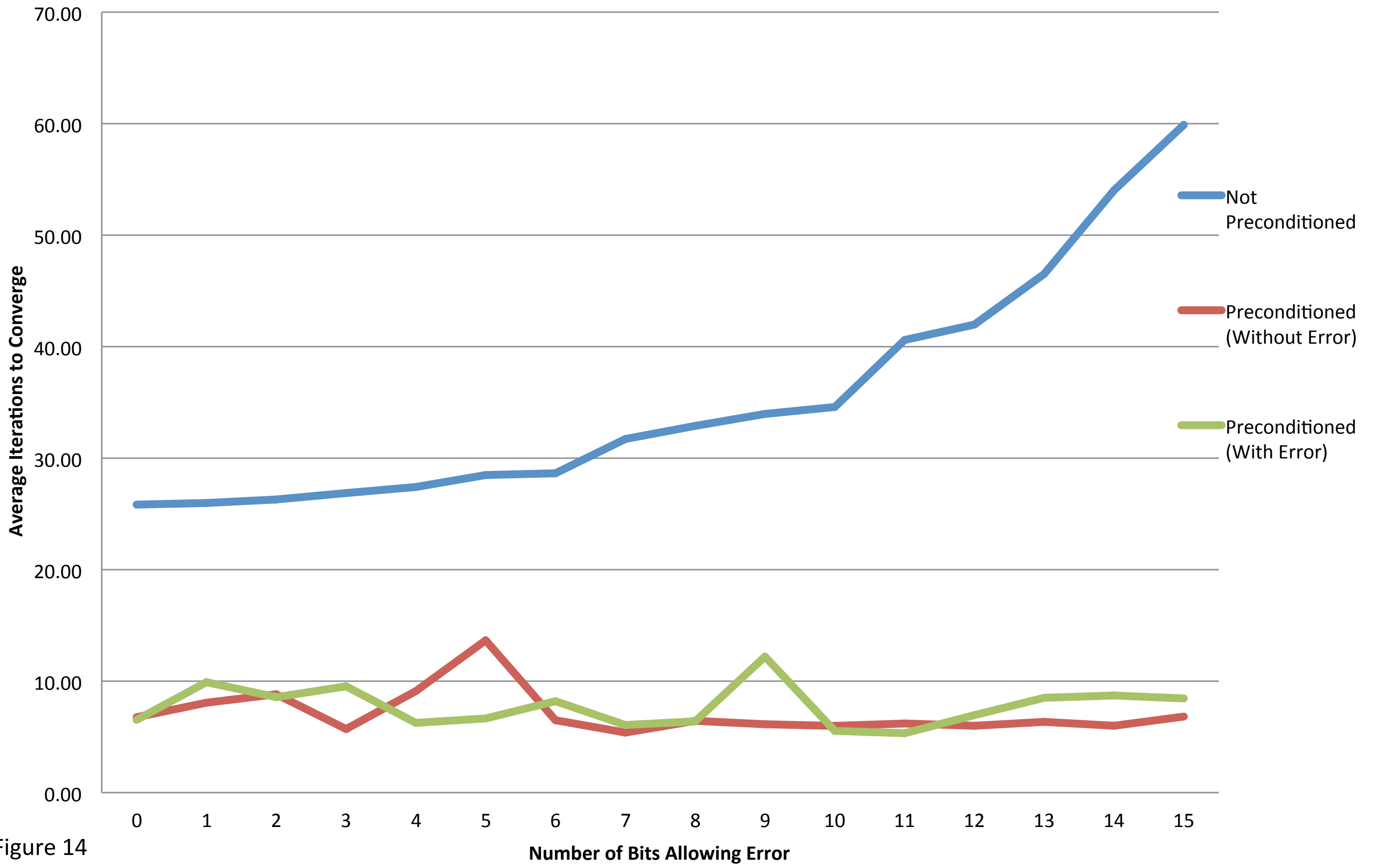

Figure 14

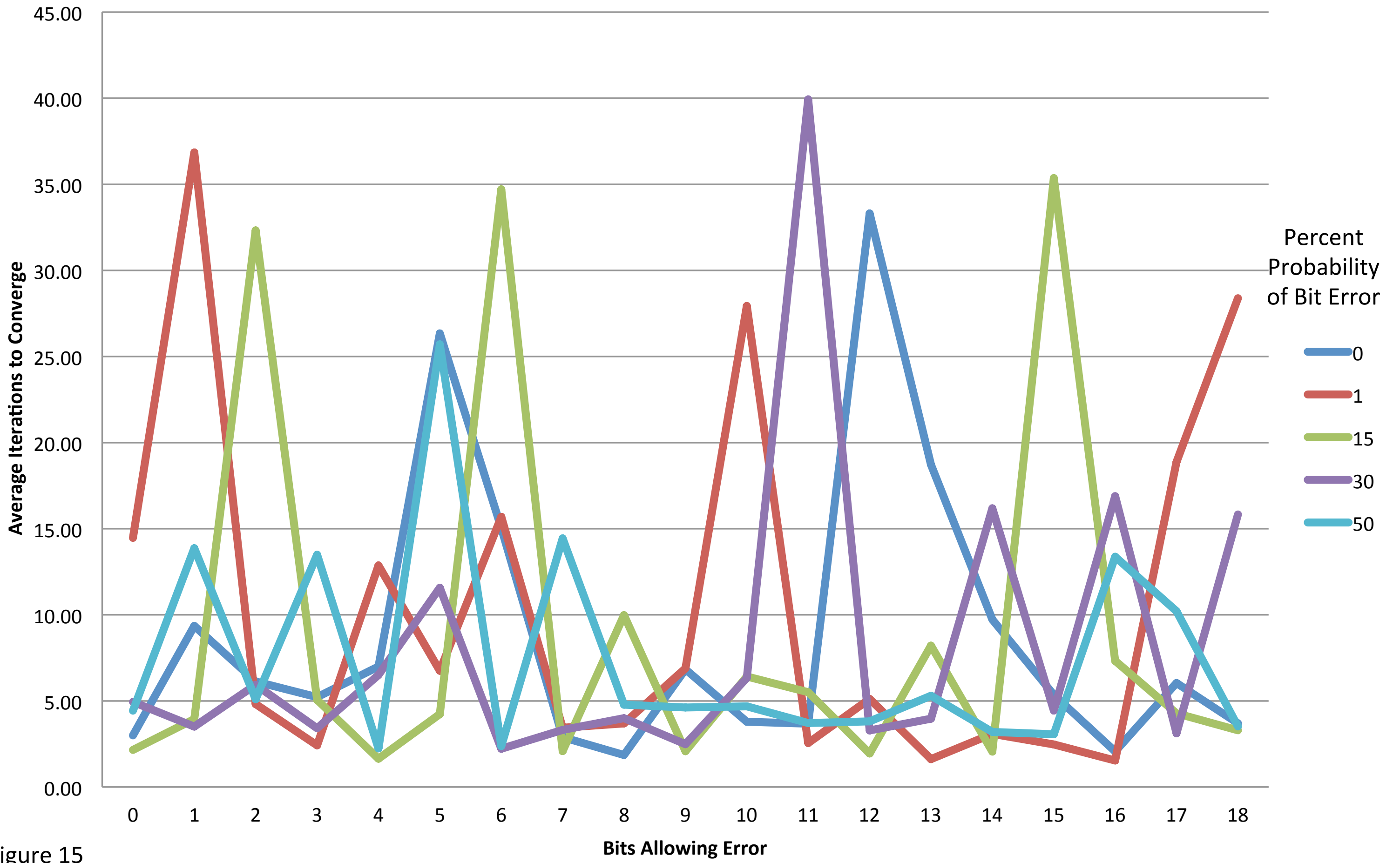

Figure 15

8.0 Conclusions

As shown by this project, the introduction of error increases the number of iterations a conjugate gradient solver requires to converge, but error can be added in ways that do not prevent convergence. This demonstrates that inaccurate processors can be applied to iterative methods such as the conjugate gradient for significant savings. Whereas the number of bits allowing error (out of the 23 representing the mantissa) could not exceed about 13-15 without significant costs, representing more than half of a number's mantissa with inaccurate bits can lead to significant gains, especially because high probabilities of a bit flip are tolerated.

Error was shown to be tolerated on a majority of operations on the inaccurate processor, implying that sufficient inaccurate processing can be tolerated to achieve gains in power consumption, chip area, and speed. The effectiveness of redundancy implies that if the benefits of using the inaccuracy are sufficiently great, they will outweigh costs of performing an operation multiple times, resulting in a faster solution. Likewise, the error tolerance shown by the preconditioned conjugate gradient method indicates that both computation of a preconditioner and the preconditioned method are also viable applications of inaccurate processing.

While error in this project was represented by inaccurate bits prone to random flips, inexact processors can be realized in a number of ways. The results of this experiment imply that regardless of how error is allowed, iterative methods for the solutions of systems of equations may present a feasible expansion of the application space for inexact processors. This could lead to increased processing power on smaller computers, allowing problems previously relegated to

supercomputers to be run on a personal computer or allowing mobile devices to have the power of larger desktop computers. Use of inexact processors could also have a role in the feasibility of future exascale machines.

9.0 Future Work

In order to increase the application space for inexact processors, other linear solvers, preconditioned solvers, and nonlinear solvers could be tested. The design of the solver can also be improved to better take advantage of the inexact processor. If an error large enough to cause divergence could be recognized, the method could reset itself in order to still reach a solution. More redundancies could also be implemented by using parallel inaccurate processors for overall increases in efficiency.

10.0 Appendices

10.1 Works Cited

[1] Pretz, Kathy. "New Rebooting Computing Working Group will Tackle Technological Challenges." IEEE The Institude. 8 March 2013. Web 12 March 2013.

[2] Boyd, Jade. "Computing Experts Unveil Superefficient Inexact©ଋ Chip." *Rice University News and Media*. Rice University, 17 May 2012. Web. 12 Mar. 2013.

[3] Boyd, Jade. "'Pruned' Microchips Are Faster, Smaller, More Energy Efficient." *Rice University News & Media*. Rice University, 17 Mar. 2011. Web. 12 Mar. 2013.

[4] Lingamneni et al. Algorithmic Methodologies for Ultra-efficient Inexact Architectures for Sustaining Technology Scaling. In *ACM Proceedings of the 9th conference on Computing Frontiers*, 2012.

[5] Lingamneni et al. Energy Parsimonious Circuit Design through Probabilistic Pruning. NTU-Rice Institute for Sustainable and Applied Infodynamics, 2011.

[6] Bates, Joseph. "Processing with Compact Arithmetic Processing Element." United States Patent Application Publication, 23 Dec. 2010. Web. 12 Mar. 2013.

[7] Bates, Joseph. "Computing 10,000X More Efficiently." *MIT Media Lab*. Singular Computing LLC, n.d. Web. 6 Dec. 2012.

[8] Edwards, E. "Floating Point Numbers." *Floating Point Numbers*. N.p., n.d. Web. 12 Mar. 2013.

[9] "Conjugate Gradient Method." *Iowa State Computer Science*. Iowa State, 6 Nov. 2007. Web. 6 Dec. 2012.

10.2 Code

10.2.1 Matrix Creation Class

```
/**
 * matrixMaker
 */
import java.util.*;
import java.io.*;
public class matrixMaker
{

    int size;
    double[][] matrix;
    double[] b;
    double randXmag;
    boolean randomMatrices = true;

    public matrixMaker(int sizeIn)
    {

        size = sizeIn;
        assemble();
        makeB();
    }

    public double[][] getA()
    {
        return matrix;
    }

    public double[] getB()
    {

        return b;
    }

    /**
     * Create b by multiplying A and a random x
     */
    private void makeB()
    {

        Random generator = new Random(87355);
        double[] randX = new double[size];
        randXmag=0;
```

```
    for (int i = 0; i<size; i++)
    {
      if (randomMatrices)//random numbers for matrix different each trial
      {
        randX[i]= (int)(Math.abs( Math.random()*100));
      }
      else// same random matrix every trial
      {
        randX[i]= (int)(Math.abs(generator.nextDouble())*100);
      }
      randXmag += randX[i]*randX[i];
    }

    randXmag = Math.sqrt(randXmag);

    b= new double[size];

    for (int i = 0; i<size; i++)
    {
      double ax= 0.0;
      for (int j = 0; j<size; j++)
      {
        ax += matrix[i][j] * randX[j];

      }

      b[i] = ax;

    }
  }

  /**
   * Create matrix A = Btranspose*B
   */
  private void assemble()
  {
    Random generator = new Random(13429);
    matrix = new double[size][size];

    for (int i = 0; i<size; i++)
    {
      for (int j = 0; j<size; j++)
      {

        if (randomMatrices)
        {
```

```
                matrix[i][j] =(int)(Math.abs( Math.random())*10);
            }
            else
            {
                matrix[i][j] =(int)(Math.abs( generator.nextDouble())*10);
            }

        }

    }
    transposeMultiply(matrix);//ensure  solution by making positive definite
}

/**
 * Multiply matrix by its transpose
 */
private void transposeMultiply(double[][] a)
{
    double[][] c = new double[size][size];
    for (int i = 0; i<size; i++)
    {
        for (int j =0; j<size; j++)
        {
            for (int k = 0 ; k<size; k++)
            {
                c[i][j] += a[i][k]*a[j][k];
            }
        }
    }
    matrix = c;
}

public double getXmagnitude()
{
    return randXmag;
}

}
```

10.2.2 Basic Model Iterative Solver

```
/**
 * iterativeSolver
 */
import java.util.*;
import java.io.*;
import Jama.*;
public class iterativeSolver
{

  private int xx=1;
  private int xy;
  private int frequencyFactor = 10;//out of 10000
  private double convergenceThreshold = 0.0000000001;
  private int functionCount;
  private double errorFactor;
  private int finalIterations;

  public static void main(String[] args)
  {
    double tempfact = 1000;

    int[] iterations = new int[100];
    double sum = 0;

    for (int i =0; i<100; i++)
    {
      iterativeSolver mySolve = new iterativeSolver(tempfact);

      iterations[i] = mySolve.getFinali();
      sum+= iterations[i];

    }
    double average = sum/100;

  }

  public static double standardDeviation(int[] x, int size, double average)
  {
    double squareSum = 0;
    for (int i = 0; i< size; i++)
    {
      squareSum += (x[i]-average)*(x[i]-average);
    }
    squareSum/=size-1;//size-1, because its a sample, not a population
```

```
    return Math.sqrt(squareSum);
}

public int getFinali()
{
    return finalIterations;
}

public iterativeSolver(double errorFactor)
{
    this.errorFactor= errorFactor;

    xy =12;
    //create matrix system
    matrixMaker myMatrix = new matrixMaker(xy);
    functionCount = 0;
    double[][] A = myMatrix.getA();

    double[] b = myMatrix.getB();

    double[] answer = solve(A, b);

    double[] check = checkAnswer(answer, A);

}

/**
 * Main solver method
 */
private double[] solve(double[][] A, double[] b)
{
    double bMagnitude = 0;
    for (int i=0; i<xy; i++)
    {
        bMagnitude+= b[i]*b[i];

    }
    bMagnitude = Math.sqrt(bMagnitude);

    double[] x = new double[xy];
    for (int i = 0; i<xy; i++)
    {
        x[i] = 0.1;//initial guess
    }
```

```
    //get r
    double[] r = residualVector(b, A, x);
    double rCheck = Math.sqrt(transposeMultiply(r));

    if (rCheck>0.0000001)
    {

      //get alpha
      double[] p = r;
      double alpha = getAlpha(r, A, p);

      //get current x -> x1
      x = getX(x, alpha, p,0);//x = previous x (x0), used in updating x

      //set up iterative loop
      int i = 0;
      double prevRCheck= rCheck+10;

      while(rCheck>convergenceThreshold)
      {
        double[] r0 = r;
        r = updateR( r0,  alpha, A, p);
        prevRCheck = rCheck;
        rCheck = Math.sqrt(transposeMultiply(r))/bMagnitude;
        double beta = getBeta(r, r0);

        p = updateP(r, beta, p);
        alpha = getAlpha(r, A, p);
        double[] x0 = x;
        x = getX(x0, alpha, p, i);

        i++;

      }

      finalIterations= i;

    }

    return x;

  }

  /**
   * Residual vector r = b- Ax
   */
```

```
private double[] residualVector(double[] b, double[][] A, double[] x)
{
   double[] r = new double[xy];//same dimensions as x
   double[] ax =  multiplyAx(A, x);

   //r = b - Ax
   for (int i = 0; i<xy; i++)//through b
   {
      r[i] = b[i] - ax[i];
   }
   return r;

}

/**
 * Calculate scalar alpha
 */
private double getAlpha(double[] r, double[][] A, double[] p)
{

   double numerator;
   double denominator = 0;

   numerator = transposeMultiply(r);
   //denomiator * A
   double[] dTemp = new double[xy];
   dTemp = multiplyAx(A, p);

   for (int i = 0 ; i< xy; i++)
   {
      denominator += dTemp[i] * p[i] ;
      denominator +=  getRand(denominator);
   }

   return numerator/denominator;
}

/**
 * multiply vector by its transpose
 */
private double transposeMultiply(double[] v)
{

   double x = 0;

   for (int  i = 0; i< xy; i++)
```

```
    {
        x += v[i]*v[i];
        //  x+=getRand(x);
    }

    return x;
}

/**
 * update x = x+ alpha*p
 */
private double[] getX(double[] x0, double alpha, double[] p, int iterations)
{
    double[] x = new double[xy];

    for (int i = 0; i< xy; i++)
    {
        x[i] = x0[i] + alpha * p[i] ;
        //  x[i]+= getRand(x[i]);
    }
    return x;
}

/**
 *  update r = r - alpha*A*p
 */
private double[] updateR(double[] r0, double alpha, double[][] A, double[] p0)
{

    double[] r = new double[xy];
    //r = r0 - alpha0 * A * p0

    double[] Ap0 = multiplyAx(A, p0);

    for (int i = 0; i<xy; i++)
    {
        r[i] = r0[i] - alpha*Ap0[i];
        //   r[i] +=getRand(r[i]);
    }

    return r;
}

/**
 * Update p = r+pprev*beta
 */
```

```
private double[] updateP(double[] r, double beta, double[] p0)
{
   functionCount++;
   double[] p = new double[xy];
   for (int i = 0; i<xy; i++)
   {
      p[i] = r[i] + p0[i]*beta;
      //  p[i] +=  getRand(p[i]);
   }

   return p;
}
/**
 * beta- ratios of current and previous residuals
 */
private double getBeta(double[] r, double[] r0)
{
   double beta = 0.;
   beta = transposeMultiply(r)/transposeMultiply(r0);
   // beta += getRand(beta);
   return beta;
}

private double[] checkAnswer(double[] answer, double[][] A)
{
   double[] check = multiplyAx(A, answer);

   return check;

}
/**
 * Multiply a matrix by a vector
 */
private double[] multiplyAx(double[][] A, double[] x)
{
   double[] Ax = new double[xy];
   //check = A * b
   for (int i = 0; i<xy; i++)
   {
      double ax= 0.0;
      for (int j = 0; j<xy; j++)
      {
         ax += A[j][i] * x[j];
         //  ax +=  getRand(ax);

      }
```

```
            Ax[i] = ax;
        }
        return Ax;
    }

    private void writeToFile(String fileName, ArrayList<Double> data)
    {
        FileWriter fWriter = null;
        BufferedWriter writer = null;
        try
        {
            fWriter = new FileWriter(fileName+".txt");
            writer = new BufferedWriter(fWriter);

            for (int i = 0; i< data.size(); i++)
            {
                writer.write(data.get(i)+"");
                writer.newLine();
            }

            writer.close();
        } catch (Exception e) {
            System.out.println("write fail");
        }
    }

    /**
     * returns a portion of the inputed number be added as error
     */
    private double getRand(double numIn)
    {

        double sign = Math.random();
        double randPercent = Math.random()/errorFactor;//between 0, 1

        if (sign <0.5)
            return -1.0* (randPercent) * numIn;
        return (randPercent) * numIn;

    }

}
```

10.2.3 Binary Worker

```
/**
 * Binary Worker
 */
public class BinaryWorker
{

   public BinaryWorker()
   {

   }

   private static int mBits = 23;
   private static int eBits = 8;
   /**
    * static addition of A and B with given error values
    */
   public static double add(double A, double B, double errorChance, int errorDigits)
   {
      double sum = A + B;

      int sumSign = 0;
      double mantissa = sum;
      double exponent = 0;


         if (sum<0)
            sumSign = -1;
         else
            sumSign = 1;

         while (Math.abs(mantissa)-1>0)
         {
            mantissa /=2;
            exponent ++;

         }
         while (Math.abs(mantissa)-1<0 && Math.abs(sum)!=0)
         {
            mantissa *= 2;
            exponent --;

         }
```

```
    int[] mSum = DoubletoBinary(Math.abs(mantissa)-1, mBits);//fractional part

    mSum = addError(mSum, errorChance, errorDigits, mBits);
    mantissa = sumSign * ( 1+ BinarytoDecimal(mSum, mBits));//add one back in
    while (exponent>0)
    {
      mantissa *= 2;
      exponent --;
    }
    while (exponent<0)
    {
      mantissa /= 2;
      exponent ++;
    }
    return mantissa;
  }


  /**
   * static multiply A and B with given error values
   */
  public static double multiply(double A, double B, double errorChance, int
errorDigits)
  {

    double mult = A * B;

    int multSign = 0;
    double mantissa = mult;
    double exponent = 0;

      if (mult<0)
        multSign = -1;
      else
        multSign = 1;

      while (Math.abs(mantissa)-1>0)
      {
        mantissa /=2;
        exponent ++;

      }
      while (Math.abs(mantissa)-1<0 && Math.abs(mult)!=0)
      {
        mantissa *= 2;
```

```
            exponent --;

          }

        int[] mMult = DoubletoBinary(Math.abs(mantissa)-1, mBits);//fractional part

        mMult = addError(mMult, errorChance, errorDigits, mBits);

        mantissa = multSign * ( 1+ BinarytoDecimal(mMult, mBits));//add one back in

        while (exponent>0)
        {
          mantissa *= 2;
          exponent --;
        }
        while (exponent<0)
        {
          mantissa /= 2;
          exponent ++;
        }
        return mantissa;

      }

      /**
       * Duplicate operation to prevent errors
       */
      public static double tripleAdd(double A, double B, double errorChance, int
    errorDigits)
      {
        double a1 = add( A,  B,  errorChance,  errorDigits);
        double a2 =add( A,  B,  errorChance,  errorDigits);
        double a3 =add( A,  B,  errorChance,  errorDigits);

        double a12 = Math.abs (a1-a2);
        double a23 = Math.abs(a2-a3);
        double a13 = Math.abs (a1-a3);

        if (a12<a13 && a12<a23)
        {
          return (a1+a2)/2;
        }
        if (a13<a12 && a13<a23)
        {
          return (a1+a3)/2;
        }
```

```
    if (a23<a13 && a23<a12)
    {
      return (a2+a3)/2;
    }

    return a1;
  }

  public static double tripleMultiply(double A, double B, double errorChance, int
errorDigits)
  {
    double a1 = multiply( A,  B,  errorChance,  errorDigits);
    double a2 =multiply( A,  B,  errorChance,  errorDigits);
    double a3 =multiply( A,  B,  errorChance,  errorDigits);

    double a12 = Math.abs (a1-a2);
    double a23 = Math.abs(a2-a3);
    double a13 = Math.abs (a1-a3);

    if (a12<a13 && a12<a23)
    {
      return (a1+a2)/2;
    }
    if (a13<a12 && a13<a23)
    {
      return (a1+a3)/2;
    }
    if (a23<a13 && a23<a12)
    {
      return (a2+a3)/2;
    }

    return a1;
  }

  /**
   * used to create targeted extremely large errors
   */
  public static double bigErrorMult(double A, double B, double errorChance, int
errorDigits)
  {

    double mult = A * B;

    int multSign;
    if (mult<0)
```

```
        multSign = -1;
    else
        multSign = 1;
    double mantissa = mult;
    double exponent = 0;

    while (Math.abs(mantissa)-1>0)
    {
        mantissa /=2;
        exponent ++;

    }
    while (Math.abs(mantissa)-1<0 && Math.abs(mult)!=0)
    {
        mantissa *= 2;
        exponent --;

    }
    int[] mMult = DoubletoBinary(Math.abs(mantissa)-1, mBits);//fractional part

    mMult = addError(mMult, 100, errorDigits, mBits);

    for (int i =0; i<5; i++)
    {
        mMult[i] = Math.abs(mMult[i] -1);
    }
    mantissa = multSign * ( 1+ BinarytoDecimal(mMult, mBits));//add one back in
    while (exponent>0)
    {
        mantissa *= 2;
        exponent --;
    }
    while (exponent<0)
    {
        mantissa /= 2;
        exponent ++;
    }
    return mantissa;

}

/**
 * convert a double to an array of binary digits
 * @return xb2 = x base 2
 */
private static int[] DoubletoBinary(double x, int bits)
```

```
{
   int[] xb2 = new int[bits];//x base 2

   int i = 0;
   double numTemp = Math.abs(x);

   while (i<bits)
   {
      numTemp =numTemp * 2;
      xb2[i] = (int)numTemp;
      numTemp = numTemp - (int)numTemp;
      i++;

   }
   if (x <0)//2s complement for negativity
   {

      for (int j =0; j<bits; j++)//flip mBits
      {
         if (xb2[j] ==1) xb2[j] = 0;
         else xb2[j] = 1;
      }
      //add 1
      xb2[bits-1]++;
      int temp = xb2[bits-1];
      int k = bits-1;
      while (temp ==2)
      {
         if (k>0)
         {
            xb2[k-1] ++;
            xb2[k] = 0;
            temp = xb2[k-1];

         }
         else
         {
            temp = 0;
         }
         k--;
      }

   }
   return xb2;
}
```

```
/**
 * convert binary double to decimal form
 */
private static double BinarytoDecimal(int[] x2, int bits)
{
   double x=0;
   for (int i = 0; i<bits; i++)
   {
      if (x2[i] ==1)
      {
         x += 1./Math.pow(2, i+1);
      }
   }
   return x;
}

/**
 * convert binary integer to decimal form
 */
private static int InttoDecimal(int[] x2, int bits)
{
   int x=0;
   for (int i = 0; i<bits; i++)
   {
      if (x2[i] ==1)
      {
         x += Math.pow(2, i);
      }
   }
   return x;
}

/**
 * called on binary number to generate error
 */
private static int[] addError(int[] x, double errorChance, int digits, int bits)
{

   int[] result = new int[bits];
   for (int i = bits-digits; i<bits; i++)
   {
      double rand =( Math.random() *100);

      if (rand < errorChance)
      {
         result[i] = Math.abs(x[i] -1);
```

```
        }
        else
        {
            result[i] = x[i];

        }
    }
    for (int i = 0; i<bits-digits; i++)
    {
        result[i] = x[i];
    }
    return result;
}

public static void additionTester()
{
    double num1 = 8.0483;
    double num2 = 30065.808;
    int chance = 1;
    int digits = 15;
    for (int j =0; j<20; j++)
    {
        for (int i = 0; i<10; i++)
        {
            double sum = add(num1, num2,chance, digits);
            double percentError =Math.abs( (sum -
(num1+num2))/(num1+num2))*100;
            System.out.println(percentError);
        }

        chance++;
    }

}

public static void multiplicationTester()
{
    double num1 = 8.0483;
    double num2 = 30065.808;
    int chance = 1;
    int digits = 15;
    for (int j =0; j<20; j++)
    {
        for (int i = 0; i<10; i++)
        {
```

```
            double sum = multiply(num1, num2,chance, digits);
            double percentError =Math.abs( (sum -
(num1*num2))/(num1*num2))*100;
            System.out.println(percentError);
         }

         chance++;
      }

   }
}
```

10.2.4 Parallel Solver

```
/**
 * ParallelSolver
 */
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.geom.*;
import Jama.*;
public class ParallelSolver
{

  private byte[] values;
  private int count;
  private long sum;
  private double logSum;
  private Worker[] workers;
  private int size;
  private double errFact;
  private int workerMode;//1 or 2= blocks 1 or 2

  private int xx=1;
  private int xy;

  private static double normThreshold =0.0000000001;//measure to determine if
converged
  private double iterationMax = 100000; //max out to warn if solution not
converging
  private double errorFactor;
  private static int numProcessors = 4;
  private static final int NUM_THREADS = numProcessors;
  Block1Data[] b1d = new Block1Data[numProcessors];
  Block2Data[] b2d = new Block2Data[numProcessors];
  private double xMag;
  private Processor[] pros = new Processor[numProcessors];
  private int runFinished;
  private int iFinal;
  private double errorFinal;

  private boolean errorOn =true;
  public static void main(String[] args)
  {

    double[] iterationsAverages = new double[22];//average for each data point
```

```
        int numTrials = 10;
        double[] iterations = new double[numTrials]; //array of iteration counts,
rewritten for each data set
        int numIncluded = 0;// unconverging runs discluded but rarely

        for (int j=0; j<10; j++)//j = number of bits allowing error
        {
            numIncluded = 0;
            for (int i = 0; i<numTrials; i++)
            {
                ParallelSolver s = new ParallelSolver(30,12,13);//%, xy ,bits
                System.out.println(s.getiFinal());
                if (s.getiFinal()>1000)
                    System.out.println("maxout");
                else
                {
                    iterationsAverages[j]+= s.getiFinal();
                    iterations[i] = s.getiFinal();
                    numIncluded++;
                }

            }
            iterationsAverages[j]/=numIncluded;
            //System.out.println(iterationsAverages[j]+" "+standardDeviation(iterations,
numIncluded, iterationsAverages[j])+" "+numIncluded);
        }
    }

    public  ParallelSolver(int errorChance, int xy, int bits)
    {

        this.size = numProcessors;
        this.xy = xy;
        long start = System.currentTimeMillis();
        matrixMaker myMatrix = new matrixMaker(xy);// generate import matrix
problem
        // StructuredMatrixMaker myMatrix = new StructuredMatrixMaker(xy);
        double[][] A = myMatrix.getA();
        double[] b = myMatrix.getB();
        xMag = myMatrix.getXmagnitude(); //for checking - x is not known

        for (int i = 0; i<numProcessors; i++)
        {
            pros[i] = new Processor(A, b,xy,errorOn, errorChance, bits);//initialize
processors
        }
```

```
        double[] answer = solve(A, b);//call conjugate gradient method

        double[] check = checkAnswer(answer, A);//multiply back to check
        double xMagCheck = 0;

        xMagCheck = Math.sqrt(xMagCheck);

    }

    private void initializeThreads() {
        long start = System.currentTimeMillis();

        int sliceLength = size / NUM_THREADS;

        workers= new Worker[NUM_THREADS];
        if ((size % NUM_THREADS )== 0)
        {
            for (int i = 0; i < workers.length; i++)
            {
                workers[i] = new Worker(sliceLength * i, sliceLength);
            }
        }
        else
        {
            for (int i = 0; i < workers.length-1; i++)
            {
                workers[i] = new Worker(sliceLength * i, sliceLength);
            }
            workers[workers.length-1] = new  Worker(sliceLength*(workers.length-1),
sliceLength+ (size % NUM_THREADS ));//lsast worker gets remainder
        }

    }

    /**
     * Calls threads to start
     */
    private void processWithThreads()
    {

        for (int i = 0; i < workers.length; i++) {
            workers[i].start();
        }
        for (int i = 0; i < workers.length; i++) {
            try {
                workers[i].join();
```

```
        }
        catch (InterruptedException e) {}
    }

}

/**
 * Calls processors to run blocks 1 and 2 of code
 * Parallelizes processors by assigning separate threads
 */
public synchronized void update(byte value)
{
    if (workerMode ==1)
    {
        int k= (int) value;
        b1d[k] = pros[k].block1(b1d[k]);

    }
    else
    {
        int k= (int) value;
        b2d[k] = pros[k].block2(b2d[k]);

    }

}

/**
 * Main iterative method- controls threads of processors
 */
private double[] solve(double[][]A, double[] b)
{

    double bMagnitude = 0;
    for (int i=0; i<xy; i++)
    {
        bMagnitude+= b[i]*b[i];
    }
    bMagnitude = Math.sqrt(bMagnitude);//b magnitude for normalizing rCheck

    double[] x = new double[xy];
    for (int i = 0; i<xy; i++)
    {
        x[i] = 0;//initial guess
    }
```

```
    //get r
    double[] r = residualVector(b, A, x);

    double rCheck = Math.sqrt(transposeMultiply(r));
    rCheck/=bMagnitude;//L2 norm of residual used as a check

    if (rCheck> 0.0000000001)//if guess isn't correct
    {

      double[] p = r;//initial p is r
      double alpha = getAlpha(r, A, p);

      int i = 0;

      while( rCheck>normThreshold && i< iterationMax)//while not close enough
and above threshold
      {

        //---------BLOCK 1--------------

        double[] r0 = r;//previous r is current r
        r =new double[xy];

        double transR = 0;
        double transR0 = 0;
        int startPoint = 0;
        int endPoint = startPoint + xy/numProcessors;
        for (int  k = 0; k<numProcessors; k++)//loop processors
        {
          //data to pass
          double[] tempX = new double[xy/numProcessors];
          double[] tempR = new double[xy/numProcessors];
          for (int m = startPoint; m< endPoint; m++)
          {
            tempX[m-startPoint] = x[m];
            tempR[m-startPoint] = r0[m];
          }
          //put data in block 1 to pass to processors
          b1d[k] = new Block1Data();
          b1d[k].x = tempX;
          b1d[k].alpha = alpha;

          b1d[k].r0 = tempR;
          b1d[k].p = p;
          b1d[k].startPoint = startPoint;
          b1d[k].endPoint = endPoint;
```

```
    b1d[k].iteration = i;
    startPoint = endPoint;
    endPoint += xy/numProcessors;
  }

  //call processor
  runFinished = 0;
  workerMode = 1;
  initializeThreads();
  processWithThreads();

  //return data
  startPoint = 0;
  endPoint = startPoint + xy/numProcessors;
  for (int  k = 0; k<numProcessors; k++)
  {
    transR += b1d[k].rtrans;

    transR0 += b1d[k].r0trans;
    for (int m = b1d[k].startPoint; m<b1d[k].endPoint; m++)
    {
      x[m] = b1d[k].x[m-b1d[k].startPoint];
      r[m] = b1d[k].r[m-b1d[k].startPoint];
    }

    startPoint = endPoint;
    endPoint += xy/numProcessors;
  }
  double beta = transR/ transR0;
  rCheck = Math.sqrt(transR);

  //---------BLOCK 2--------------

  startPoint = 0;
  endPoint = startPoint + xy/numProcessors;
  for (int  k = 0; k<numProcessors; k++)//pass data...
  {
    //send data, setup
    double[] tempR = new double[xy/numProcessors];
    for (int m = startPoint; m< endPoint; m++)
    {
      tempR[m-startPoint] = r[m];
    }
    //data to block 2 to go to processor
    b2d[k] = new Block2Data();
    b2d[k].beta = beta;
```

```
          b2d[k].r = tempR;
          b2d[k].p = p;
          b2d[k].startPoint = startPoint;
          b2d[k].endPoint = endPoint;
          startPoint = endPoint;
          endPoint += xy/numProcessors;
        }
        //call processor
        runFinished = 0;
        workerMode = 2;
        initializeThreads();
        processWithThreads();

        for (int  k = 0; k<numProcessors; k++)
        {
          //return data
          for (int m = b2d[k].startPoint; m<b2d[k].endPoint; m++)
          {
            p[m] = b2d[k].pseg[m-b2d[k].startPoint];
          }

        }

        //alpha = final block, requires full updated p
        double alphaDenom=0;
        startPoint = 0;
        endPoint = startPoint + xy/numProcessors;
        for (int k = 0; k<numProcessors; k++)
        {
          alphaDenom +=  pros[k].alphaDenominator(p, startPoint, endPoint);

          startPoint = endPoint;
          endPoint += xy/numProcessors;

        }
        alpha = transR/alphaDenom;
        rCheck/=bMagnitude;

        i++;
      }

      iFinal = i;
    }

    return x;
}
```

```
/**
 * Calculate residual = Ax-b
 */
private double[] residualVector(double[] b, double[][] A, double[] x)
{
   double[] r = new double[xy];//same dimensions as x

   int startPoint = 0;
   int endPoint = startPoint + xy/numProcessors;
   for (int k = 0; k<numProcessors; k++)
   {
      double[] tempR = pros[k].residualVector(x, startPoint, endPoint);
      for (int i = startPoint; i<endPoint; i++)
      {
         r[i] = tempR[i-startPoint];

      }
      startPoint = endPoint;
      endPoint += xy/numProcessors;
   }
   return r;
}

/**
 * Multply Ax to compare to b
 */
private double[] checkAnswer(double[] answer, double[][] A)
{
   double[] check = multiplyAx(A, answer);

   return check;

}

/**
 * Calculate scalar alpha
 * (used for initial value before loop)
 */
private double getAlpha(double[] r, double[][] A, double[] p)
{

   double numerator;
   double denominator = 0;

   numerator = transposeMultiply(r);
```

```
    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
        denominator +=  pros[k].alphaDenominator(p, startPoint, endPoint);

        startPoint = endPoint;
        endPoint += xy/numProcessors;
    }
    return numerator/denominator;
}

/**
 * Multiply vector by its transpose
 */
private double transposeMultiply(double[] v)
{

    double x = 0;

    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
        x +=  pros[k].transposeMultiply(v, startPoint, endPoint);

        startPoint = endPoint;
        endPoint += xy/numProcessors;
    }

    return x;
}

/**
 * Multiply a matrix by a vector
 */
private double[] multiplyAx(double[][] A, double[] x)
{
    double[] Ax = new double[xy];
    //check = A * b
    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
        double[] temp =  pros[k].multiplyAx(x, startPoint, endPoint);
```

```
        for (int i = startPoint; i<endPoint; i++)//place in actual result
        {
            Ax[i] = temp[i-startPoint];
        }
        startPoint = endPoint;
        endPoint += xy/numProcessors;
    }
    return Ax;
}

/**
 * Calculate the standard deviation of a list
 */
public static double standardDeviation(double[] x, int size, double average)
{
    double squareSum = 0;
    for (int i = 0; i< size; i++)
    {
        squareSum += (x[i]-average)*(x[i]-average);
    }
    squareSum/=size-1;//size-1, because its a sample, not a population

    return Math.sqrt(squareSum);
}

public double getErrorOut()
{
    return errorFinal;
}

public int getiFinal()
{
    return iFinal;
}
/**
 * Class for worker for multithreading
 */
private class Worker extends Thread
{

    private int rangeStart;
    private int rangeLength;

    public Worker(int rangeStart, int rangeLength) {
        this.rangeStart = rangeStart;
        this.rangeLength = rangeLength;
```

```
        }

        public void run() {
            for (int i = rangeStart; i < rangeStart + rangeLength; i++) {
                update((byte)i);
            }
            runFinished++;

        }

    }

}
```

10.2.5 Processor

```
/**
 * Processor
 * simulated processor used in ParallelSolver
 */
public class Processor
{

   double[][] A;//stored in local memory upon initialization (constant values)
   double[] b;
   int xy;//dimension of the matrices, vectors
   private boolean errorOn;
   int errorPercent;
   private int digits;

   public Processor( double[][]A, double[]b, int xy, boolean errorOn, int
errorPercent, int bits)
   {
      this.A = A;
      this.b = b;
      this. xy = xy;
      this.errorOn = errorOn;
      this.errorPercent = errorPercent;
      this.digits = bits;
   }

   /**
    * Multiply a segment of a matrix by vector segment (for parallelization)
    * startPoint and endPoint correspond to range on this thread
    */
   public double[] multiplyAx(double[] x, int startPoint, int endPoint)
   {
      int segmentLength = endPoint -startPoint;
      double[] Ax = new double[segmentLength];

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         double ax= 0.0;
         for (int j = 0; j<xy; j++)
         {
            ax += A[j][i] * x[j];

         }
```

```
        Ax[i-startPoint] = ax;//fill in all of this Ax
      }

      return Ax;
   }

   /**
    * Calculate residual vector portion
    */
   public double[] residualVector(double[] x, int startPoint, int endPoint)
   {

      int segmentLength = endPoint -startPoint;
      double[] Ax = new double[segmentLength];
      double[] r = new double[segmentLength];

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         double ax= 0.0;
         for (int j = 0; j<xy; j++)
         {
            ax += A[j][i] * x[j];

         }

         Ax[i-startPoint] = ax;//fill in all of this Ax
         r[i-startPoint] =b[i]- Ax[i-startPoint];
      }

      return r;
   }

   /**
    * Part of alpha denominator
    */
   public double alphaDenominator(double[] p, int startPoint, int endPoint)
   {

      double denominator=0;
      double[] dTemp = new double[xy];
      int segmentLength = endPoint -startPoint;

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
```

```
    {
      double ax= 0.0;
      for (int j = 0; j<xy; j++)
      {
        ax += A[j][i]*p[j];

      }

      dTemp[i-startPoint] = ax;
      double tempMult =dTemp[i-startPoint] * p[i];
      denominator +=tempMult;

    }

    return denominator;

  }

  /**
   * part of vector by its transpose
   */
  public double transposeMultiply(double[] v, int startPoint, int endPoint)
  {
    double sum=0;
    for (int i =startPoint; i<endPoint; i++)
    {
      sum += v[i]*v[i];
    }
    return sum;
  }

  /**
   * return segment of x
   */
  public double[] getX(double[] x0, double alpha, double[] p, int startPoint, int
endPoint)
  {
    int segmentLength = endPoint -startPoint;
    double[] x = new double[segmentLength];
    for (int i = startPoint; i<endPoint; i++)
    {
      x[i-startPoint] = x0[i] + alpha * p[i];
    }
    return x;

  }
```

```
   /**
    * update segment of r
    */
   public double[] updateR(double[] r0, double alpha, double[] p0, int startPoint, int
endPoint)
   {
      int segmentLength = endPoint -startPoint;
      double[] Ap0 = new double[segmentLength];
      double[] r = new double[segmentLength];

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         double ap= 0.0;
         for (int j = 0; j<xy; j++)
         {
            ap += A[j][i] * p0[j];

         }

         Ap0[i-startPoint] = ap;
         r[i-startPoint] =r0[i]- alpha*Ap0[i-startPoint];
      }

      return r;
   }

   /**
    * update segment of p
    */
   public double[] updateP(double[] r, double beta, double[] p0, int startPoint, int
endPoint)
   {
      int segmentLength = endPoint -startPoint;

      double[] p = new double[segmentLength];
      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         p[i-startPoint] = r[i] + p0[i]*beta;
      }

      return p;
   }
```

```
    /**
     * Block 1 of code- groups operations for updating r and alpha
     */
    public Block1Data block1(Block1Data b1d)
    {

        int startPoint = b1d.startPoint;
        int endPoint = b1d.endPoint;
        int segmentLength = endPoint -startPoint;
        double[] xreturn = new double[segmentLength];
        double[] rreturn= new double[segmentLength];
        double r0trans = 0;
        double rtrans =0;

        double[] Ap0 = new double[segmentLength];
        for (int i = startPoint; i<endPoint; i++)
        {
            //update x
            if (Math.abs(b1d.alpha)>0)
            {
                if (errorOn)
                {
                    double tempMult1 = BinaryWorker.multiply(b1d.alpha, b1d.p[i],
errorPercent, digits);
                    //  double tempMult1 =b1d.alpha * b1d.p[i];
                    // xreturn[i-startPoint] =b1d.x[i-startPoint]+tempMult1;
                    xreturn[i-startPoint] = BinaryWorker.add(b1d.x[i-startPoint],
tempMult1,errorPercent,digits);
                }
                else
                {
                    xreturn[i-startPoint] = b1d.x[i-startPoint] + b1d.alpha * b1d.p[i];
                }
            }
            else
            {
                xreturn[i-startPoint] = b1d.x[i-startPoint];
                System.out.println("Stop"+b1d.alpha);
            }

            //update r
            double ap= 0.0;
            for (int j = 0; j<xy; j++)
            {
                if (errorOn)
                {
```

```
          // double tempMult3 =
BinaryWorker.multiply(A[j][i],b1d.p[j],errorPercent, digits);
          // ap = BinaryWorker.add(ap, tempMult3, errorPercent, digits);
          ap+=A[j][i]*b1d.p[j];
          //  ap = ap+ tempMult3;
        }
        else
        {
          ap+=A[j][i]*b1d.p[j];
        }
      }

      Ap0[i-startPoint] = ap;//fill in all of this Ax
      if (errorOn)
      {

        //%, digits
        double tempMult =0;

        tempMult= -1*BinaryWorker.multiply(b1d.alpha,Ap0[i-
startPoint],errorPercent,digits);
        //  tempMult= -1*b1d.alpha*Ap0[i-startPoint];
        rreturn[i-startPoint] =BinaryWorker.add(b1d.r0[i-
startPoint],tempMult,errorPercent,digits);
        // rreturn[i-startPoint] =b1d.r0[i-startPoint]+tempMult;

        tempMult = BinaryWorker.multiply(rreturn[i-startPoint], rreturn[i-
startPoint],errorPercent,digits);
        // tempMult = BinaryWorker.bigErrorMult(rreturn[i-startPoint], rreturn[i-
startPoint],0,0);
        // tempMult = rreturn[i-startPoint]* rreturn[i-startPoint];
        rtrans = BinaryWorker.add(rtrans, tempMult, errorPercent,digits);
        //   rtrans = rtrans + tempMult;

        tempMult =BinaryWorker.multiply(b1d.r0[i-startPoint], b1d.r0[i-
startPoint],errorPercent,digits);
        // tempMult =b1d.r0[i-startPoint]*b1d.r0[i-startPoint];
        r0trans = BinaryWorker.add(r0trans,tempMult, errorPercent,digits);
        //  r0trans = r0trans+tempMult;
      }

      else
      {
        double tempMult= -1*b1d.alpha*Ap0[i-startPoint];
        rreturn[i-startPoint] =b1d.r0[i-startPoint]+tempMult;
        r0trans += b1d.r0[i-startPoint] * b1d.r0[i-startPoint];
```

```
            rtrans += rreturn[i-startPoint] * rreturn[i-startPoint];
        }

      }

      b1d.r = rreturn;
      b1d.x = xreturn;
      b1d.r0trans = r0trans;
      b1d.rtrans = rtrans;

      return b1d;
   }

   /**
    * Block 2 of code updates p
    */
   public Block2Data block2(Block2Data b2d)
   {

      int startPoint = b2d.startPoint;
      int endPoint = b2d.endPoint;
      int segmentLength = endPoint -startPoint;

      double rTrans = 0;
      double[] pTemp = new double[segmentLength];

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         if (errorOn)
         {
            double tempMult =BinaryWorker.multiply(b2d.p[i],b2d.beta, errorPercent,
digits);
            pTemp[i-startPoint] = BinaryWorker.add(b2d.r[i-startPoint],tempMult,
errorPercent,digits);
         }
         else
         {

            pTemp[i-startPoint] = b2d.r[i-startPoint]+b2d.p[i]*b2d.beta;
         }
      }
      b2d.pseg = pTemp;

      return b2d;
```

```
    }

    public void errorOff()
    {errorOn=false;
    }

    public void errorOn()
    {errorOn=true;}

}
```

10.2.6 Preconditioned Parallel Solver

```
/**
 * PreconditionedSolver
 * almost exactly like ParallelSolver, but with preconditioning
 */
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.geom.*;
import Jama.*;
public class PreconditionedSolver
{

   private byte[] values;
   private int count;
   private long sum;
   private double logSum;
   private Worker[] workers;
   private int size;
   private double errFact;
   private int workerMode;//1 or 2= blocks 1 or 2

   private int xx=1;
   private int xy =20;

   private static double normThreshold =0.0000000001;//measure to determine if
converged
   private double iterationMax = 100000;
   private double errorFactor;
   private static int numProcessors = 1;
   private static final int NUM_THREADS = numProcessors;
   Block1Data[] b1d = new Block1Data[numProcessors];
   Block2Data[] b2d = new Block2Data[numProcessors];
   private double xMag;
   private PreconProcessor[] pros = new PreconProcessor[numProcessors];
   private int runFinished;
   private int iFinal;
   private double errorFinal;

   private boolean errorOn =true;
   public static void main(String[] args)
   {
      double[] iterationsAverages = new double[22];

      int numTrials = 1;
```

```
      double[] iterations = new double[numTrials];
      int numIncluded = 0;
      for (int j=0; j<1; j++)
      {
        numIncluded = 0;
        for (int i = 0; i<numTrials; i++)
        {
          PreconditionedSolver s = new PreconditionedSolver(0,12,0);//%, xy ,bits

          if (s.getiFinal()>1000)
            System.out.println("maxout");
          else
          {
            iterationsAverages[j]+= s.getiFinal();
            iterations[i] = s.getiFinal();
            numIncluded++;
          }

        }
        iterationsAverages[j]/=numIncluded;
        System.out.println(iterationsAverages[j]+" "+standardDeviation(iterations,
numIncluded, iterationsAverages[j])+" "+numIncluded);
      }

    }

    public  PreconditionedSolver(int errorChance, int xy, int bits)
    {

      this.size = numProcessors;
      this.xy = xy;
      long start = System.currentTimeMillis();
      StructuredMatrixMaker myMatrix = new StructuredMatrixMaker(xy,true);
      double[][] A = myMatrix.getA();
      double[] b = myMatrix.getB();
      // double[][] pc = JacobiPreconditioner(A,xy, errorChance, bits);//pc = pre
conditioner
      double[][] pc = CholeskyPreconditioner(A,xy, errorChance, bits);//pc = pre
conditioner
      // CholeskyPreconditioner(A,xy, errorChance, bits);
      xMag = myMatrix.getXmagnitude();

      for (int i = 0; i<numProcessors; i++)
      {
        pros[i] = new PreconProcessor(A, b,pc,xy, errorOn, errorChance, bits);
      }
```

```
        double[] answer = solve(A, b, pc);

        double[] check = checkAnswer(answer, A);
        double xMagCheck = 0;

        xMagCheck = Math.sqrt(xMagCheck);

    }

    /**
     * create Jacobi preconditioner matrix
     * technically only need array- matrix fits template for cholesky
     */
    private double[][] JacobiPreconditioner(double[][] A, int xy, int errorChance, int
bits)
    {
        double[][] preconditioner = new double[xy][xy];//two dims is not necessary,
but it makes same code useable for cholesky
        for (int i =0; i<xy;i++)
        {
            preconditioner[i][i] =1;//BinaryWorker.multiply(1.,(1./A[i][i]),
errorChance,bits);///A[i][i]; //inverse of diagonal
            //preconditioner[i][i] =(1./A[i][i]);///A[i][i]; //inverse of diagonal

        }
        return preconditioner;
    }

    /**
     * create Cholesky preconditioner matrix
     */
    private double[][] CholeskyPreconditioner(double[][] A,int xy, int errorChance, int
bits)
    {

        double[][] l = new double[xy][xy];

        for(int i = 0; i< xy;i++)
        {
            for(int k = 0; k < (i+1); k++)//lower diagonal
            {
                if (A[i][k] == 0)//setting 0s makes this a partial
                {
                    l[i][k] = 0;
                }
```

```
        else
        {
          double sum = 0;
          for(int j = 0; j < k; j++)
          {
            sum += l[i][j] * l[k][j];
          }

          if (i==k)//on the diagonal
          {
            l[i][i] = Math.sqrt(Math.abs(A[i][i]- sum));

          }
          else //below diagonal
          {

            l[i][k] = (1./l[k][k]) * (A[i][k] -sum );

          }
        }
      }

    }

    System.out.println("L:");
    for (int i =0; i<xy; i++)
    {
      for (int j=0; j<xy; j++)
      {
        System.out.print(l[i][j]+" ");
      }
      System.out.println();
    }
    //invert L
    Matrix lMatrix = new Matrix(l);
    Matrix linverse = lMatrix.inverse();
    double[][] lInv = new double[xy][xy];
    double[][] ltransInv = new double[xy][xy];
    for (int i =0; i<xy; i++)
    {
      for (int j=0; j<xy; j++)
      {
        lInv[i][j] = linverse.get(i,j);
        ltransInv[j][i] = lInv[i][j];
      }
```

```
    }

    double[][] precon = multiplyMA(lInv, ltransInv);

    System.out.println("precon:");
    for (int i =0; i<xy; i++)
    {
       for (int j=0; j<xy; j++)
       {
          System.out.print(precon[i][j]+" ");
       }
       System.out.println();
    }
    return precon;

 }
 /**
  * create Cholesky decomposition of A
  */
 private double[][] CholeskyDecomposition(double[][] A,int xy)
 {

    double[][] l = new double[xy][xy];

    for(int i = 0; i< xy;i++)
    {
       for(int k = 0; k < (i+1); k++)//lower diagonal
       {

          double sum = 0;
          for(int j = 0; j < k; j++)
          {
             sum += l[i][j] * l[k][j];
          }

          if (i==k)//on the diagonal
          {
             l[i][i] = Math.sqrt(A[i][i]- sum);
          }
          else //below diagonal
          {
             l[i][k] = (1./l[k][k]) * (A[i][k] -sum );
          }

       }
```

```
    }
    return l;

  }

  private void initializeThreads() {
    long start = System.currentTimeMillis();

    int sliceLength = size / NUM_THREADS;

    workers= new Worker[NUM_THREADS];
    if ((size % NUM_THREADS )== 0)
    {
      for (int i = 0; i < workers.length; i++)
      {
        workers[i] = new Worker(sliceLength * i, sliceLength);
      }
    }
    else
    {
      for (int i = 0; i < workers.length-1; i++)
      {
        workers[i] = new Worker(sliceLength * i, sliceLength);
      }
      workers[workers.length-1] = new  Worker(sliceLength*(workers.length-1),
sliceLength+ (size % NUM_THREADS ));//lsast worker gets remainder
    }

  }

  private void processWithThreads()
  {

    for (int i = 0; i < workers.length; i++) {
      workers[i].start();
    }
    for (int i = 0; i < workers.length; i++) {
      try {
        workers[i].join();
      }
      catch (InterruptedException e) {}
    }

  }

  public synchronized void update(byte value)
```

```
{
   if (workerMode ==1)
   {
      int k= (int) value;
      b1d[k] = pros[k].block1(b1d[k]);

   }
   else
   {
      int k= (int) value;
      b2d[k] = pros[k].block2(b2d[k]);
   }

}

private double[] solve(double[][]A, double[] b, double[][] pc)
{

   double bMagnitude = 0;
   for (int i=0; i<xy; i++)
   {
      bMagnitude+= b[i]*b[i];

   }
   bMagnitude = Math.sqrt(bMagnitude);

   double[] x = new double[xy];
   for (int i = 0; i<xy; i++)
   {
      x[i] = Math.random();//b[i];//initial guess
   }

   //get r
   double[] r = residualVector(b, A, x);
   double[] z =new double[xy]; //z acts a preconditioned intermediate step
   z= multiplyAx(pc, r);
   double[] p = new double[xy];
   double rCheck = Math.sqrt(transposeMultiply(r));
   rCheck/=bMagnitude;

   double[] rReset = new double[xy];
   double[] xReset = new double[xy];
   for (int i = 0; i<xy; i++)
   {
      rReset[i] = r[i];
      xReset[i] = x[i];
```

```
    p[i] = z[i];
}

if (rCheck> 0.0000000001)
{

    double alpha = getAlpha(r, A, p, z);

    int i = 0;

    while( rCheck>normThreshold && i< iterationMax)
    {

        //---------BLOCK 1--------------

        double[] r0 = r;
        r =new double[xy];

        double transR = 0;
        double transR0 = 0;
        int startPoint = 0;
        int endPoint = startPoint + xy/numProcessors;
        for (int  k = 0; k<numProcessors; k++)//loop processors
        {
            //data to pass
            double[] tempX = new double[xy/numProcessors];
            double[] tempR = new double[xy/numProcessors];
            double[] tempZ = new double[xy/numProcessors];
            for (int m = startPoint; m< endPoint; m++)
            {
                tempX[m-startPoint] = x[m];
                tempR[m-startPoint] = r0[m];
                tempZ[m-startPoint] = z[m];
            }
            b1d[k] = new Block1Data();
            b1d[k].x = tempX;
            b1d[k].alpha = alpha;
            b1d[k].z = tempZ;
            b1d[k].r0 = tempR;
            b1d[k].p = p;
            b1d[k].startPoint = startPoint;
            b1d[k].endPoint = endPoint;
            b1d[k].iteration = i;
            startPoint = endPoint;
            endPoint += xy/numProcessors;
        }
```

```
//call processor
runFinished = 0;
workerMode = 1;
initializeThreads();
processWithThreads();

//return data
startPoint = 0;
endPoint = startPoint + xy/numProcessors;
for (int  k = 0; k<numProcessors; k++)
{
  transR += b1d[k].rtrans;

  transR0 += b1d[k].r0trans;
  for (int m = b1d[k].startPoint; m<b1d[k].endPoint; m++)
  {
    x[m] = b1d[k].x[m-b1d[k].startPoint];
    r[m] = b1d[k].r[m-b1d[k].startPoint];
    z[m] = b1d[k].zreturn[m-b1d[k].startPoint];

  }

  startPoint = endPoint;
  endPoint += xy/numProcessors;
}

double beta = transR/ transR0;

rCheck = Math.sqrt(transR);

//---------BLOCK 2--------------

startPoint = 0;
endPoint = startPoint + xy/numProcessors;
for (int  k = 0; k<numProcessors; k++)//pass data...
{
  //send data, setup
  double[] tempR = new double[xy/numProcessors];
  double[] tempZ = new double[xy/numProcessors];
  for (int m = startPoint; m< endPoint; m++)
  {
    tempR[m-startPoint] = r[m];
    tempZ[m-startPoint] = z[m];

  }
```

```
        b2d[k] = new Block2Data();
        b2d[k].beta = beta;
        b2d[k].r = tempR;
        b2d[k].p = p;
        b2d[k].z = tempZ;
        b2d[k].startPoint = startPoint;
        b2d[k].endPoint = endPoint;
        startPoint = endPoint;
        endPoint += xy/numProcessors;
    }
    //call processor
    runFinished = 0;
    workerMode = 2;

    initializeThreads();
    processWithThreads();

    for (int  k = 0; k<numProcessors; k++)//pass data...
    {
        //return data

        for (int m = b2d[k].startPoint; m<b2d[k].endPoint; m++)
        {
            p[m] = b2d[k].pseg[m-b2d[k].startPoint];

        }

    }

    //alpha = final block, requires full updated p
    double alphaDenom=0;
    startPoint = 0;
    endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
        alphaDenom +=  pros[k].alphaDenominator(p, startPoint, endPoint);

        startPoint = endPoint;
        endPoint += xy/numProcessors;

    }
    alpha = transR/alphaDenom;

    rCheck/=bMagnitude;
    System.out.println(rCheck);
    i++;
```

```
        }

        iFinal = i;
    }

    return x;
}

private double[] residualVector(double[] b, double[][] A, double[] x)
{
    double[] r = new double[xy];//same dimensions as x

    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
        double[] tempR = pros[k].residualVector(x, startPoint, endPoint);
        for (int i = startPoint; i<endPoint; i++)
        {
            r[i] = tempR[i-startPoint];

        }
        startPoint = endPoint;
        endPoint += xy/numProcessors;
    }
    return r;
}

private double[] checkAnswer(double[] answer, double[][] A)
{
    double[] check = multiplyAx(A, answer);

    return check;

}

private double getAlpha(double[] r, double[][] A, double[] p, double[] z)
{

    double numerator =0;
    double denominator = 0;

    //numerator = rtranspose * z
    for (int i =0; i<xy; i++)
    {
        numerator += r[i]*z[i];
```

```
    }

    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
      denominator +=  pros[k].alphaDenominator(p, startPoint, endPoint);

      startPoint = endPoint;
      endPoint += xy/numProcessors;
    }
    return numerator/denominator;
  }

  private double transposeMultiply(double[] v)
  {

    double x = 0;

    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
      x +=  pros[k].transposeMultiply(v, startPoint, endPoint);

      startPoint = endPoint;
      endPoint += xy/numProcessors;
    }

    return x;
  }

  private double[] multiplyAx(double[][] A, double[] x)
  {
    double[] Ax = new double[xy];
    //check = A * b
    int startPoint = 0;
    int endPoint = startPoint + xy/numProcessors;
    for (int k = 0; k<numProcessors; k++)
    {
      double[] temp =  pros[k].multiplyAx(A,x, startPoint, endPoint);
      for (int i = startPoint; i<endPoint; i++)//place in actual result
      {
        Ax[i] = temp[i-startPoint];
      }
      startPoint = endPoint;
```

```
        endPoint += xy/numProcessors;
    }

    return Ax;
}

private double[][] multiplyMA(double[][] pc, double[][] A)
{
    double[][] pA = new double[xy][xy];//preconditioned A
    //check = A * b
    double pAtemp = 0;
    for (int k= 0; k<xy;k++)
    {
        for (int j = 0; j<xy; j++)
        {
            pAtemp = 0;
            for (int i = 0; i<xy; i++)//place in actual result
            {
                pAtemp += pc[k][i]*A[i][j];
            }
            pA[k][j] = pAtemp;
        }
    }

    return pA;
}

public static double standardDeviation(double[] x, int size, double average)
{
    double squareSum = 0;
    for (int i = 0; i< size; i++)
    {
        squareSum += (x[i]-average)*(x[i]-average);
    }
    squareSum/=size-1;//size-1, because its a sample, not a population

    return Math.sqrt(squareSum);
}

public double getErrorOut()
{
    return errorFinal;
}

public int getiFinal()
{
```

```
        return iFinal;
    }
    private class Worker extends Thread
    {

        private int rangeStart;
        private int rangeLength;

        public Worker(int rangeStart, int rangeLength) {
            this.rangeStart = rangeStart;
            this.rangeLength = rangeLength;
        }

        public void run() {
            for (int i = rangeStart; i < rangeStart + rangeLength; i++) {
                update((byte)i);
            }
            runFinished++;

        }

    }

}
```

10.2.7 Structured Matrix Maker

```
/**
 * matrixMaker
 */
import java.util.*;
import java.io.*;
public class StructuredMatrixMaker
{

   int size;
   double[][] matrix;
   double[] b;
   double randXmag;
   boolean randomMatrices = true;

   /**
    * constructor for making a normally distributed matrix
    */
   public StructuredMatrixMaker(int sizeIn, boolean sparse)
   {

      size = sizeIn;
      assembleSparse();
      makeB();
   }

   /**
    * constructor for a block diagonal matrix
    */
   public StructuredMatrixMaker(int sizeIn)
   {
      size = sizeIn;
      assembleBlock();
      makeB();
   }

   public double[][] getA()
   {
      return matrix;
   }

   public double[] getB()
   {
```

```
    return b;
  }

  /**
   * Create b by multiplying A and x
   */
  private void makeB()
  {

    Random generator = new Random(87355);
    double[] randX = new double[size];
    randXmag=0;

    for (int i = 0; i<size; i++)
    {
      if (randomMatrices)
      {
        randX[i]= (int)(Math.abs( Math.random()*100));
      }
      else
      {
        randX[i]= (int)(Math.abs(generator.nextDouble())*100);
      }
      randXmag += randX[i]*randX[i];
    }

    randXmag = Math.sqrt(randXmag);
    b= new double[size];

    for (int i = 0; i<size; i++)
    {
      double ax= 0.0;
      for (int j = 0; j<size; j++)
      {
        ax += matrix[i][j] * randX[j];

      }

      b[i] = ax;

    }

  }

  /**
   *
```

```
 */
private void assembleSparse()
{
   Random generator = new Random(13429);
   matrix = new double[size][size];
   int valueProb = 30;
   int valueRand;

   for (int i = 0; i<size; i++)
   {
      for (int j = 0; j<size; j++)
      {
         valueRand = (int)(Math.random()*100.);
         if (valueRand< valueProb || i==j)
         {
            if (randomMatrices)
            {
               matrix[i][j] =(int)(Math.abs( Math.random())*10);
            }
            else
            {
               matrix[i][j] =(int)(Math.abs( generator.nextDouble())*10);
            }
         }
         else
         {
            if (i!=j)
            {
               matrix[i][j] = 0;
               matrix[j][i] = 0;
            }

         }

      }

   }

   transposeMultiply(matrix);
   for (int i =0; i<size; i++)
   {
      for (int j=0; j<size; j++)
      {
         System.out.print(matrix[i][j]+" ");
      }
      System.out.println();
```

```
    }
  }

  /**
   * block Diagonal - blocks of matrices allong the diagonal
   */
  private void assembleBlock()
  {
    Random generator = new Random(13429);
    matrix = new double[size][size];
    int counter = 0;
    while( counter<( size))
    {
      int currentSize =  1+(int)(Math.random() * ((int)size/2));

      if ((counter + currentSize)> size)//last block just finished out the matrix
        currentSize = size - counter;

      for (int i = counter; i<counter + currentSize ; i++)//through mini block
defined by size along diagonal
      {
        for (int j =counter; j< counter + currentSize; j++)
        {
          if (randomMatrices)
          {
            if (i==j)
              matrix[i][j] = Math.random() * 100;
            else
              matrix[i][j] = Math.random() * 0.1;
          }
          else
          {
            matrix[i][j] =(int)(Math.abs( generator.nextDouble())*10);
          }
        }
      }
      counter += currentSize;
    }


    transposeMultiply(matrix);
  }

  private void transposeMultiply(double[][] a)
  {
    double[][] c = new double[size][size];
```

```
        for (int i = 0; i<size; i++)
        {
            for (int j =0; j<size; j++)
            {
                for (int k = 0 ; k<size; k++)
                {
                    c[i][j] += a[i][k]*a[j][k];
                }
            }
        }
        matrix = c;
    }

    public double getXmagnitude()
    {
        return randXmag;
    }

}
```

10.2.8 Preconditioned Processor

```
/**
 * Processor
 * simulated processor used in PreconditionedSolver
 * basically Processor, but with precondtioning; ie z
 */
public class PreconProcessor
{

   double[][] A;//stored in local memory upon initialization (constant values)
   double[] b;
   double[][] pc;
   int xy;
   private boolean errorOn;
   int errorPercent;
   private int digits;
   public PreconProcessor( double[][]A, double[]b,double[][] pc, int xy, boolean
errorOn, int errorPercent, int bits)
   {
      // this.mainSolve = mainSolve;
      this.A = A;
      this.b = b;
      this. xy = xy;
      this.pc = pc;
      this.errorOn = errorOn;
      this.errorPercent = errorPercent;
      this.digits = bits;
   }

   public double[] multiplyAx(double[][] Ain, double[] x, int startPoint, int endPoint)
   {
      int segmentLength = endPoint -startPoint;
      double[] Ax = new double[segmentLength];

      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
         double ax= 0.0;
         for (int j = 0; j<xy; j++)
         {
            ax += Ain[j][i] * x[j];

         }
```

```
        Ax[i-startPoint] = ax;//fill in all of this Ax
      }

      return Ax;
   }

   public double[] residualVector(double[] x, int startPoint, int endPoint)
   {

      int segmentLength = endPoint -startPoint;
      double[] Ax = new double[segmentLength];
      double[] r = new double[segmentLength];
      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
        double ax= 0.0;
        for (int j = 0; j<xy; j++)
        {
          ax += A[j][i] * x[j];

        }

        Ax[i-startPoint] = ax;//fill in all of this Ax
        r[i-startPoint] =b[i]- Ax[i-startPoint];
      }

      return r;
   }

   public double alphaDenominator(double[] p, int startPoint, int endPoint)
   {

      double denominator=0;
      double[] dTemp = new double[xy];
      int segmentLength = endPoint -startPoint;
      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
        double ax= 0.0;
        for (int j = 0; j<xy; j++)
        {
          ax += A[j][i]*p[j];

        }

        dTemp[i-startPoint] = ax;
```

```
      double tempMult =dTemp[i-startPoint] * p[i];
      denominator +=tempMult;
    }

    return denominator;

  }

  public double transposeMultiply(double[] v, int startPoint, int endPoint)
  {
    double sum=0;
    for (int i =startPoint; i<endPoint; i++)
    {
      sum += v[i]*v[i];
    }
    return sum;
  }

  public double[] getX(double[] x0, double alpha, double[] p, int startPoint, int
endPoint)
  {
    int segmentLength = endPoint -startPoint;
    double[] x = new double[segmentLength];
    for (int i = startPoint; i<endPoint; i++)
    {
      x[i-startPoint] = x0[i] + alpha * p[i];
    }
    return x;

  }

  public double[] updateR(double[] r0, double alpha, double[] p0, int startPoint, int
endPoint)
  {
    int segmentLength = endPoint -startPoint;
    double[] Ap0 = new double[segmentLength];
    double[] r = new double[segmentLength];
    for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
    {
      double ap= 0.0;
      for (int j = 0; j<xy; j++)
      {
        ap += A[j][i] * p0[j];

      }
```

```
        Ap0[i-startPoint] = ap;//fill in all of this Ax
        r[i-startPoint] =r0[i]- alpha*Ap0[i-startPoint];
      }

      return r;
   }

   public double[] updateP(double[] r, double beta, double[] p0, int startPoint, int
endPoint)
   {
      int segmentLength = endPoint -startPoint;

      double[] p = new double[segmentLength];
      for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
      {
        p[i-startPoint] = r[i] + p0[i]*beta;
      }

      return p;
   }

   public Block1Data block1(Block1Data b1d)
   {

      int startPoint = b1d.startPoint;
      int endPoint = b1d.endPoint;
      int segmentLength = endPoint -startPoint;
      double[] xreturn = new double[segmentLength];
      double[] rreturn= new double[segmentLength];
      double[] zreturn = new double[segmentLength];
      double r0trans = 0;
      double rtrans =0;

      double[] Ap0 = new double[segmentLength];
      for (int i = startPoint; i<endPoint; i++)
      {

        //update x
        if (Math.abs(b1d.alpha)>0)
        {
          if (errorOn)
          {
             double tempMult1 = BinaryWorker.multiply(b1d.alpha, b1d.p[i],
errorPercent, digits);
```

```
            //  double tempMult1 =b1d.alpha * b1d.p[i];
            // xreturn[i-startPoint] =b1d.x[i-startPoint]+tempMult1;
            xreturn[i-startPoint] = BinaryWorker.add(b1d.x[i-startPoint],
tempMult1,errorPercent,digits);
         }
         else
         {
            xreturn[i-startPoint] = b1d.x[i-startPoint] + b1d.alpha * b1d.p[i];
         }
      }
      else
      {
         xreturn[i-startPoint] = b1d.x[i-startPoint];
         System.out.println("Stop"+b1d.alpha);
      }

      //update r
      double ap= 0.0;
      for (int j = 0; j<xy; j++)
      {
         if (errorOn)
         {
            // double tempMult3 =
BinaryWorker.multiply(A[j][i],b1d.p[j],errorPercent, digits);
            // ap = BinaryWorker.add(ap, tempMult3, errorPercent, digits);
            ap+=A[j][i]*b1d.p[j];
            //  ap = ap+ tempMult3;
         }
         else
         {
            ap+=A[j][i]*b1d.p[j];
         }
      }

      Ap0[i-startPoint] = ap;//fill in all of this Ax
      if (errorOn)
      {

         //%, digits
         double tempMult =0;

         tempMult= -1*BinaryWorker.multiply(b1d.alpha,Ap0[i-
startPoint],errorPercent,digits);
         //   tempMult= -1*b1d.alpha*Ap0[i-startPoint];
         rreturn[i-startPoint] =BinaryWorker.add(b1d.r0[i-
startPoint],tempMult,errorPercent,digits);
```

```
            //  rreturn[i-startPoint] =b1d.r0[i-startPoint]+tempMult;

            double pcr= 0.0;
            for (int j = 0; j<xy; j++)
            {
               pcr+=pc[j][i-startPoint]*rreturn[i-startPoint];

            }
            zreturn[i - startPoint] = pcr;

            tempMult = BinaryWorker.multiply(rreturn[i-startPoint], zreturn[i-
startPoint],errorPercent,digits);
            //   tempMult = BinaryWorker.bigErrorMult(rreturn[i-startPoint],
zreturn[i-startPoint],0,0);
            //   tempMult = rreturn[i-startPoint]* zreturn[i-startPoint];
            rtrans = BinaryWorker.add(rtrans, tempMult, errorPercent,digits);
            // rtrans = rtrans + tempMult;

            tempMult =BinaryWorker.multiply(b1d.r0[i-startPoint], b1d.r0[i-
startPoint],errorPercent,digits);
            //tempMult =b1d.r0[i-startPoint]*b1d.z[i-startPoint];
            r0trans = BinaryWorker.add(r0trans,tempMult, errorPercent,digits);
            //r0trans = r0trans+tempMult;
         }

         else
         {
            double tempMult= -1*b1d.alpha*Ap0[i-startPoint];
            rreturn[i-startPoint] =b1d.r0[i-startPoint]+tempMult;
            double pcr= 0.0;
            for (int j = 0; j<xy; j++)
            {

               pcr+=pc[j][i]*b1d.r0[i-startPoint];
            }
            zreturn[i - startPoint] = pcr;
            r0trans += b1d.r0[i-startPoint] * b1d.r0[i-startPoint];
            rtrans += rreturn[i-startPoint] * rreturn[i-startPoint];
         }

      }

      b1d.r = rreturn;
      b1d.x = xreturn;
      b1d.zreturn = zreturn;
      b1d.r0trans = r0trans;
```

```
        b1d.rtrans = rtrans;
        return b1d;

    }
    public Block2Data block2(Block2Data b2d)
    {

        int startPoint = b2d.startPoint;
        int endPoint = b2d.endPoint;
        int segmentLength = endPoint -startPoint;

        double rTrans = 0;
        double[] pTemp = new double[segmentLength];
        for (int i =startPoint; i<endPoint; i++)//rows that are the portion for this
procesor
        {
            if (errorOn)
            {
                double tempMult =BinaryWorker.multiply(b2d.p[i],b2d.beta, errorPercent,
digits);

                pTemp[i-startPoint] = BinaryWorker.add(b2d.z[i-startPoint],tempMult,
errorPercent,digits);

            }
            else
            {

                pTemp[i-startPoint] = b2d.z[i]+b2d.p[i]*b2d.beta;
            }

        }
        b2d.pseg = pTemp;

        return b2d;

    }

    public void errorOff()
    {errorOn=false;
    }

    public void errorOn()
    {errorOn=true;}

}
```

10.2.9 Block 1 Data

```
public class Block1Data
{
   //in
   public double[] r0;
   public  double alpha;
   public double[] p;
   public int startPoint;
   public int endPoint;
   public double[] z;
   //both...
   public double[] x;
   //out
   public double[] r;
   public double rtrans;
   public double r0trans;
   public int iteration;
   public double[] zreturn;
   public Block1Data()
   {
   }
}
```

10.2.10 Block 2 Data

```
public class Block2Data
{
   //in
   public double[] r;//segment
   public double beta;
   public double[] p;//all of p in
   public double[] z;
    public int startPoint;
   public int endPoint;
   //out
   public double[] pseg;
   public double rtrans;

   public Block2Data()
   {
   }
}
```

10.3 Data for Figures 5 and 6

L2 Norm of Residual Vector over Iterations

No Error	Error in Iterations 0-10, no error in subsequent iterations		
		0.231903862	0.973140394
		0.227340747	0.94989954
		0.203029101	0.846905802
		0.171370073	0.716066659
		0.143307367	0.602132658
0.016872523	0.018700557	0.12451438	0.528192568
0.004193074	0.007552921	0.117027939	0.50287009
0.001941269	0.022573382	0.121656115	0.530301712
8.33E-04	0.026238675	0.139357607	0.615895313
5.07E-04	0.013067864	0.171478943	0.767575019
1.97E-04	0.015835132	0.218827013	0.99250854
2.60E-04	0.037739804	0.279126672	1.287000485
6.61E-05	0.052982376	0.342774741	1.616812601
8.81E-05	0.0799631	0.390514454	1.898139997
5.18E-05	0.082963182	0.401713821	2.019263336
9.12E-05	0.071781692	0.372063548	1.925448583
1.47E-05	0.062640587	0.318155034	1.675047965
6.59E-10	0.052505788	0.261921764	1.381319649
8.48E-10	0.044380122	0.217479231	1.129745284
2.93E-12	0.039610562	0.190459489	0.95831358
	0.038650585	0.182302437	0.87652618
	0.041748969	0.193650745	0.884776648
	0.049250807	0.225814124	0.983606707
	0.061514863	0.280504576	1.173186225
	0.078389295	0.357651731	1.442900879
	0.098110808	0.450565546	1.750867594
	0.116051159	0.539593059	2.005915406
	0.125426486	0.592937231	2.093033839
	0.121929376	0.586097158	1.960699874
	0.107975103	0.525229449	1.672700787
	0.090295927	0.440997162	1.344223165
	0.074659726	0.36279293	1.058657058
	0.063961154	0.306395229	0.850011124
	0.05916382	0.277218496	0.723191283
	0.060597171	0.276697585	0.673645883
	0.068715946	0.306307336	0.697406879
	0.08431625	0.36887312	0.793733249
	0.108256982	0.467449485	0.961603285
	0.140492382	0.601110014	1.189103364
	0.178060189	0.75640118	1.435906093
	0.212696799	0.897743257	1.622680957

1.662270438	3.800699623	9.622404956
1.531908785	3.160938788	7.925358996
1.296463092	2.628057593	6.96856017
1.046215361	2.288406437	6.750577198
0.839769448	2.169596313	7.277834053
0.699204264	2.28485191	8.596624996
0.627505927	2.656909767	10.76819158
0.62272128	3.323965095	13.77292494
0.684478759	4.329414467	17.31238427
0.814274105	5.686675472	20.57117904
1.009657847	7.301957182	22.31135647
1.251079447	8.870480204	21.69633806
1.484771788	9.882941875	19.11279623
1.624424104	9.930589316	15.77429565
1.602429045	9.09167194	12.73287858
1.433270538	7.859297245	10.50032875
1.198917127	6.73279577	9.212607349
0.97932459	6.003912114	8.862042248
0.81767445	5.799983267	9.425606549
0.728474542	6.188632622	10.88840549
0.714908746	7.244530278	13.17008312
0.780093138	9.063773684	15.95190503
0.931373031	11.72150428	18.4811892
1.178786352	15.14307155	19.6785464
1.526536405	18.86415717	18.85425502
1.953296062	21.83756834	16.3742217
2.382905841	22.78974279	13.28280329
2.677526711	21.26066796	10.46112227
2.710598186	18.08624723	8.320158891
2.482294037	14.56906144	6.949213822
2.120573715	11.59042696	6.31603867
1.767946038	9.489555255	6.381854081
1.509295904	8.308719888	7.144167684
1.37800065	8.00174439	8.626175397
1.385224517	8.53007613	10.80848125
1.540213228	9.869160016	13.48372194
1.85688014	11.93245679	16.06685831
2.347128767	14.41191606	17.61133876
2.997339853	16.61423522	17.37438903
3.724031799	17.59209109	15.51723023
4.334627549	16.79757385	12.94022773
4.582946808	14.61409607	10.51475546
4.358778102	11.96996976	8.70349516

7.65087872	3.350399938	35.74462447
7.369650645	4.024218283	46.28813854
7.854565536	5.083928175	59.00293888
9.110889094	6.506100826	71.39769966
11.10260216	8.122989585	79.21435823
13.61248247	9.529885656	78.91021208
16.05178681	10.18225745	71.00600092
17.46530798	9.79718168	59.56784329
17.08615257	8.638450119	48.67704685
15.05294518	7.258666772	40.56160081
12.26144234	6.09592018	35.96157599
9.57849152	5.362817375	34.99722824
7.445379019	5.134345449	37.71404256
5.969580374	5.447556304	44.23350697
5.115179707	6.354593938	54.53258084
4.819471524	7.92893256	67.7932344
5.038725023	10.22283515	81.39698934
5.746964851	13.15168373	90.58913914
6.890820817	16.29315971	90.82414532
8.296787138	18.7548191	81.82227605
9.570813173	19.5215271	67.85110372
10.15759914	18.30183276	53.77054322
9.709930691	15.83559072	42.35810482
8.422228703	13.19460381	34.46711128
6.830009275	11.10836883	30.03812033
5.381273298	9.89255677	28.79060323
4.283511416	9.651532469	30.49984625
3.580482734	10.4524708	34.9640045
3.253763053	12.40735484	41.67930594
3.280143876	15.67285496	49.26037265
3.650858212	20.3608209	55.04957389
4.360915103	26.30533793	56.0479515
5.365954178	32.66244067	51.29625562
6.505035521	37.65475158	42.82337954
7.438076633	39.27624552	33.64884832
7.748813012	36.96878841	25.75659319
7.265443386	32.21350559	19.84536236
6.23342378	27.14705703	15.90889852
5.074010168	23.23814871	13.72187634
4.090087451	21.14752032	13.07317965
3.409295887	21.12783057	13.84001918
3.058742	23.36633087	15.96002589
3.03677456	28.13368814	19.29510587

23.36928247	22.56607338	136.1237534
27.08771073	23.53761635	117.6103753
28.90098244	22.1316399	96.17066638
27.83196977	19.21741592	77.78527889
24.41205636	16.07903962	65.00985492
20.14286567	13.60128706	58.38936716
16.30817258	12.17215064	57.83520504
13.53458064	11.92016143	63.28959643
11.99707586	12.91826638	74.7491319
11.70589437	15.26803204	91.66493311
12.67176187	19.06434754	111.6796272
14.94856303	24.21928283	129.3121686
18.5615732	30.11010195	137.0204114
23.29411878	35.22455075	130.4950223
28.32920706	37.50531963	112.9003542
32.04083486	35.85902998	91.650943
32.67175404	31.2593865	72.66502023
29.83497506	25.77561604	58.59714348
24.91524623	21.02660058	49.9900097
19.72973388	17.7572704	46.6270292
15.39797113	16.17414563	48.26547724
12.29474032	16.32042253	54.83651349
10.41509222	18.28394871	66.1921475
9.648428361	22.25736961	81.34507189
9.899104648	28.47107242	97.28044267
11.09967899	36.95505573	108.4520254
13.13222514	47.03325215	109.3111595
15.65829904	56.67587921	99.11008336
17.93934277	62.6212061	82.65748908
18.95621117	62.40166156	65.8086113
18.08652944	56.70064793	52.06791879
15.69784585	48.70011252	42.60469319
12.77239447	41.48519959	37.44774793
10.13128272	36.80179604	36.36116824
8.159841329	35.38382627	39.23911998
6.948195294	37.61270062	46.15058132
6.478593008	43.92923825	57.05977668
6.73546624	54.93030486	71.1434485
7.749067641	71.13634482	85.7687067
9.59022946	92.25705594	
12.31265026	115.7631617	
15.81184819	135.5415314	
19.58435455	143.5714898	