Blendenpik: Random Sampling Least-Squares

Malachi, Dong, Aaron, Sharmishtha, Bingsheng

December 2, 2020

The Paper

The paper we are investigating is called "Blendenpik: Supercharging LAPACK's Least-Squares Solver", and as the title implies it mainly consists of solving Least-Squares problems. The authors are Haim Avron, Petar Maymounkov, and Sivan Toledo, and the paper is published in the SIAM journal of scientific computing in 2010 [1].

Least-Squares

The Least-Squares Problem is a well known problem usually taught in Calculus and Stats in its basic form, it is also important in classic linear algebra. We might know its more generalized cousin, regression.

Least-Squares

The Least-Squares Problem is a well known problem usually taught in Calculus and Stats in its basic form, it is also important in classic linear algebra. We might know its more generalized cousin, regression.

Least-Squares problems are in essence asking for a solution to the system $Ax = b$, when there isn't a true solution, that minimizes the residual. That is we look for x_{opt} as defined

$$x_{opt} = \min_{x} ||Ax - b||_2$$

Least-Squares

The Least-Squares Problem is a well known problem usually taught in Calculus and Stats in its basic form, it is also important in classic linear algebra. We might know its more generalized cousin, regression.

Least-Squares problems are in essence asking for a solution to the system $Ax = b$, when there isn't a true solution, that minimizes the residual. That is we look for x_{opt} as defined

$$x_{opt} = \min_{x} ||Ax - b||_2$$

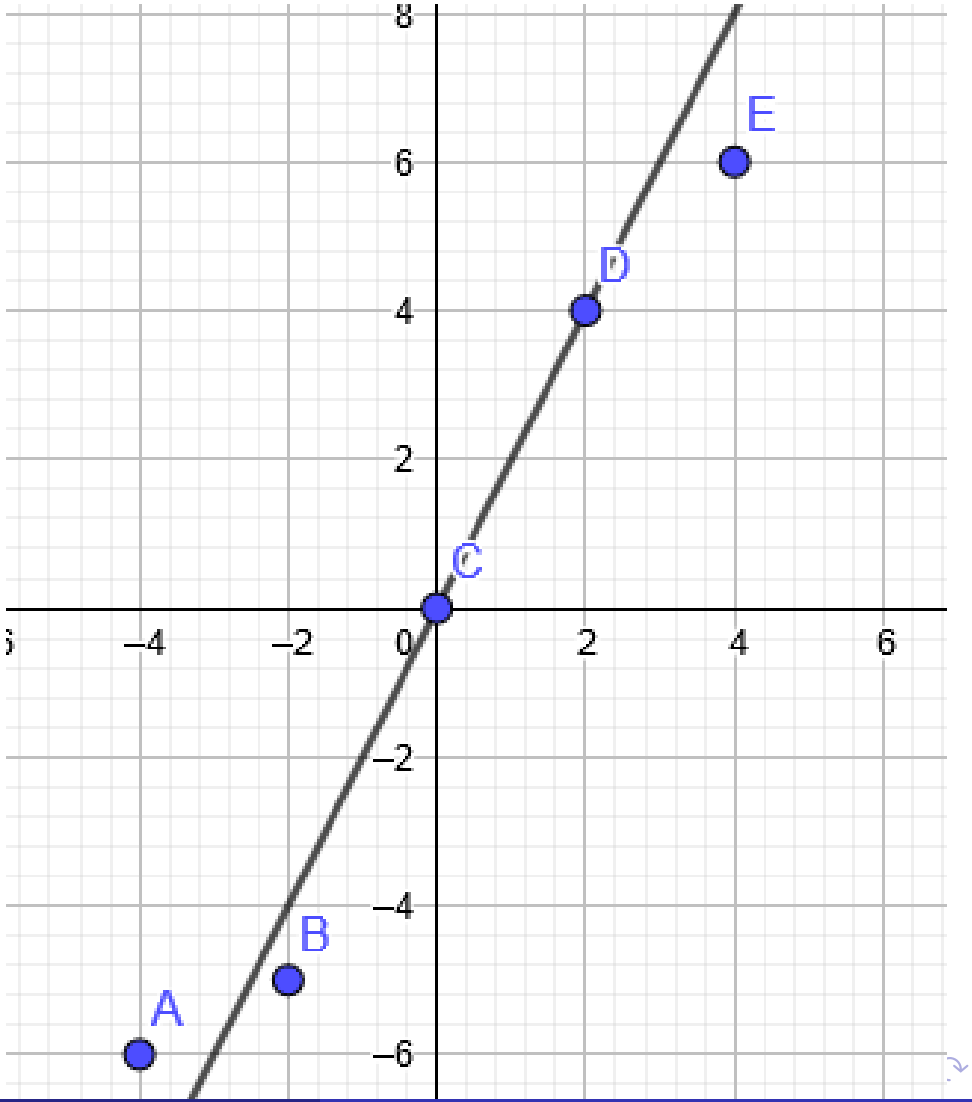
There are many classical ways of solving this problem depending on its size and your area of expertise, but lets talk about what its useful for.

Least-Square Applications

In its most basic form the Least-Squares method is useful for finding a close approximation for semi linear data.

x	-4	-2	0	2	4
y	-6	-5	0	4	6

The grey line is some fit we find, and we need there to be minimal distance from each point to the line.



Applications

There are many examples of Least-Squares problems, as the approach can be generalize quite grandly. It can be used for non linear polynomial fitting, Ridge regression, and others.

Applications

There are many examples of Least-Squares problems, as the approach can be generalize quite grandly. It can be used for non linear polynomial fitting, Ridge regression, and others.

Least-Squares lets us simply create optimal solutions to some complex systems. They can be non differentiable, and the data can be very messy, but Least-Squares doesn't care about those characteristics. It is a simple approach with simple results that are helpful.

Applications

There are many examples of Least-Squares problems, as the approach can be generalize quite grandly. It can be used for non linear polynomial fitting, Ridge regression, and others.

Least-Squares lets us simply create optimal solutions to some complex systems. They can be non differentiable, and the data can be very messy, but Least-Squares doesn't care about those characteristics. It is a simple approach with simple results that are helpful.

The best current solver of these systems are LAPACK, which used QR factorization to iterative solve systems like these.

Applications

There are many examples of Least-Squares problems, as the approach can be generalize quite grandly. It can be used for non linear polynomial fitting, Ridge regression, and others.

Least-Squares lets us simply create optimal solutions to some complex systems. They can be non differentiable, and the data can be very messy, but Least-Squares doesn't care about those characteristics. It is a simple approach with simple results that are helpful.

The best current solver of these systems are LAPACK, which used QR factorization to iterative solve systems like these.

Blendenpik uses preconditoners along with existing iterative solvers, which is not novel, however it adds a random sampling step when building the preconditioning matrix which speeds up the QR factorization greatly and makes it more than competitive with LAPACK.

Blendenpik Motivation: Iterative Solvers

Least-squares solutions can be approximated via iterative methods

Blendenpik Motivation: Iterative Solvers

Least-squares solutions can be approximated via iterative methods

Ex: LSQR uses conjugate gradient technique (analytically equivalent, at least)

Blendenpik Motivation: Iterative Solvers

Least-squares solutions can be approximated via iterative methods

Ex: LSQR uses conjugate gradient technique (analytically equivalent, at least)

Performance of iterative solvers highly depends on **condition number** of system

Motivation: Condition Numbers

Condition Number:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Motivation: Condition Numbers

Condition Number:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Bounds the accuracy of a solution to the system

Motivation: Condition Numbers

Condition Number:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Bounds the accuracy of a solution to the system

Want condition number to be small

Motivation: Condition Numbers

Condition Number:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Bounds the accuracy of a solution to the system

Want condition number to be small

Systems with large condition numbers (ill-conditioned) take significantly longer to converge to a solution within a desired tolerance

Motivation: Preconditioning

Can speed up iterative solvers through **preconditioning**

Motivation: Preconditioning

Can speed up iterative solvers through **preconditioning**

Solutions to $A\mathrm{x} = \mathrm{b}$ will be the same as solutions to $M^{-1}A\mathrm{x} = M^{-1}\mathrm{b}$ for nonsingular M.

Motivation: Preconditioning

Can speed up iterative solvers through **preconditioning**

Solutions to $Ax = b$ will be the same as solutions to $M^{-1}Ax = M^{-1}b$ for nonsingular M.

If A is ill-conditioned, computing a preconditioner and solving preconditioned system can be much faster than computing the solution to the ill-conditioned system

Motivation: Row Sampling for Preconditioning

Blendenpik tries the preconditioning approach, but with **row sampling** to speed up the construction of the preconditioner

Motivation: Row Sampling for Preconditioning

Blendenpik tries the preconditioning approach, but with **row sampling** to speed up the construction of the preconditioner

Idea: build a right-preconditioner $R \in \mathbb{R}^{n \times n}$ from some sample of rows $\hat{m}$ where $m \geq \hat{m} \geq n$

Motivation: Row Sampling for Preconditioning

Blendenpik tries the preconditioning approach, but with **row sampling** to speed up the construction of the preconditioner

Idea: build a right-preconditioner $R \in \mathbb{R}^{n \times n}$ from some sample of rows $\hat{m}$ where $m \geq \hat{m} \geq n$

e.g. if $A \in \mathbb{R}^{30000 \times 750}$, sample 1500 rows of A and compute a preconditioner from $\hat{A} \in \mathbb{R}^{1500 \times 750}$

Motivation: Row Sampling for Preconditioning

Blendenpik tries the preconditioning approach, but with **row sampling** to speed up the construction of the preconditioner

Idea: build a right-preconditioner $R \in \mathbb{R}^{n \times n}$ from some sample of rows $\hat{m}$ where $m \geq \hat{m} \geq n$

e.g. if $A \in \mathbb{R}^{30000 \times 750}$, sample 1500 rows of A and compute a preconditioner from $\hat{A} \in \mathbb{R}^{1500 \times 750}$

Solve $AR^{-1}\mathrm{y} = \mathrm{b}$, then solve $R\mathrm{x} = \mathrm{y}$

Motivation: Matrix Coherence

A possible issue with row sampling comes from the **coherence** of a matrix

Motivation: Matrix Coherence

A possible issue with row sampling comes from the **coherence** of a matrix

Coherence measures how dependent a system is on a small subset of rows

Motivation: Matrix Coherence

A possible issue with row sampling comes from the **coherence** of a matrix

Coherence measures how dependent a system is on a small subset of rows

e.g. assume A is full rank, but removal of one row causes A to be rank deficient

Motivation: Matrix Coherence

A possible issue with row sampling comes from the **coherence** of a matrix

Coherence measures how dependent a system is on a small subset of rows

e.g. assume A is full rank, but removal of one row causes A to be rank deficient

High coherence means more rows must be sampled, slowing down the algorithm

Motivation: Unitary Transforms

Can we reduce coherence to increase effectiveness of row sampling?

Motivation: Unitary Transforms

Can we reduce coherence to increase effectiveness of row sampling?

Yes, using **unitary transformations**

Motivation: Unitary Transforms

Can we reduce coherence to increase effectiveness of row sampling?

Yes, using **unitary transformations**

If U is unitary ($U^{-1} = U^*$), vectors can be multiplied by U, preserving condition number and length, but changing distribution of length across components (this is called **row mixing**)

Motivation: Unitary Transforms

Can we reduce coherence to increase effectiveness of row sampling?

Yes, using **unitary transformations**

If U is unitary ($U^{-1} = U^*$), vectors can be multiplied by U, preserving condition number and length, but changing distribution of length across components (this is called **row mixing**)

Applying a unitary transformation to A reduces risk of row sampling causing poor preconditioning

Motivation: The Algorithm Idea

Apply unitary transformation to A to reduce coherence

Motivation: The Algorithm Idea

Apply unitary transformation to A to reduce coherence

Sample a small subset of rows from A, call it SA

Motivation: The Algorithm Idea

Apply unitary transformation to A to reduce coherence

Sample a small subset of rows from A, call it SA

Create preconditioner R based on SA

Motivation: The Algorithm Idea

Apply unitary transformation to A to reduce coherence

Sample a small subset of rows from A, call it SA

Create preconditioner R based on SA

Use iterative solver to solve preconditioned system $AR^{-1}R\mathrm{x} = \mathrm{b}$

Blendenpik Algorithm

Algorithm 1 Blendenpik algorithm

Input: $\mathbf{A} \in \mathbb{R}^{m \times n}$, $m \gg n$,$\mathbf{b} \in \mathbb{R}^m$ vector ,$\gamma(\geq 1)$ is the Sampling factor.

Output: $\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2$

while not returned **do**

  $\mathbf{M} \longleftarrow F_{\tilde{m}}(\mathbf{DM})$ ▷ $\mathbf{D}$ is a diagonal matrix and $F_{\tilde{m}}$ is random unitary transform

  Let $\mathbf{S} \in \mathbb{R}^{\tilde{m} \times \tilde{m}}$ be a random diagonal matrix:

$$\mathbf{S}_{ii} = \begin{cases} 1 & , \text{ with probability } \gamma n/\tilde{m} \\ 0 & , \text{ with probability } 1 - \gamma n/\tilde{m} \end{cases}$$

  $\mathbf{M}_s = \mathbf{SM}$ ▷ Row Sampling

  $\mathbf{M}_s = \mathbf{QR}$ ▷ QR preconditioning

  $\tilde{\kappa} \longleftarrow \kappa_{\text{estimate}}\ (\mathbf{R})$

  if $\tilde{\kappa}^{-1} > 5\epsilon_{\text{machine}}$ **then**

    $\mathbf{y} = \arg\min_{\mathbf{z}} \left\|\mathbf{AR}^{-1}\mathbf{z} - \mathbf{b}\right\|_2$ ▷ Preconditioned iterative solve

    Solve for $\mathbf{R}\hat{\mathbf{x}} = y$ using LSQR/LSMR

    Return $\hat{\mathbf{x}}$

  else

    if # iterations > 3 then **then**

      Failure, solve using Baseline Least squares(LAPACK) and return

Zero padding

$$\tilde{m} \longleftarrow \begin{cases} 2^{\lceil \log_2 m \rceil} & \text{, WHT} \\ \lceil m/1000 \rceil \times 1000 & \text{, DCT or DHT} \end{cases}$$

$$M \longleftarrow \begin{bmatrix} A \\ 0 \end{bmatrix} \in \mathbb{R}^{\tilde{m} \times n}$$

$\widetilde{m}$ is the padding size, as determined by the *Transform Type* and its needs for speed. We are padding the original matrix because in later steps when we need to apply uniform transformations, each of them requires specific shape of the input matrix.

Unitary Transformations

$$\mathbf{M} \longleftarrow F_{\tilde{m}}(\mathbf{DM})$$

Unitary transformation is applied on padded matrix M, which is $\widetilde{m} \times n$. It performs row mixing that preserves the condition number. They do, however, change the coherence which is important for the sampling step.

$F_{\tilde{m}}$ is the transformation constructed as given from the 3 choices. Notice we cannot directly applied these transformations because Any fixed unitary transformation $F_{\tilde{m}}$ leads to a high $\mu(F_{\tilde{m}}(\mathrm{A}))$ on some A's, so we use a random unitary transformation. We construct $\mathcal{F}$ from a product of a fixed $\mathcal{F}$ and a random diagonal matrix D with ± 1 diagonal entries.

Unitary Transformations

Theorem 5. *Let A be an $m \times n$ full rank matrix where $m \geq n$. Let F be an $m \times m$ unitary matrix, let D be a diagonal matrix whose diagonal entries are i.i.d Rademacher random variables ($\Pr(D_{ii} = \pm 1) = 1/2$), and let $\mathcal{F} = FD$. With probability of at least 0.95 we have*

$$\mu(\mathcal{F}A) \leq Cn\eta \log m \,,$$

where $\eta = \max |F_{ij}|^2$ and some constant C.

Theorem 5 shows that if we take uniform transformation on an arbitrary matrix, then the coherence of transformed matrix is upper-bounded with high probability. And that upper bound only determined by the the shape of the input matrix and the largest entry of the transformation matrix applied.

Unitary Transformations

5 algorithms for Unitary Transformations $\mathcal{F}$ are suggested in paper [1]:

- DFT - Discrete Fourier Transform
- DCT - Discrete Cosine Transform
- DHT - Discrete Hartley Transform
- WHT - Discrete Walsh-Hadamard Transform
- Kac - Kac's random walk on n-dimensional rotation matrices

Although the paper suggest these 5, they do not implement DFT and Kac and neither do we. Because DFT of a real vector is a complex vector which entails operation-count and memory penalties and it is unlikely that DFT would outperform DCT or DHT. Kac's random walk suffers from poor cache locality due to random index selection. They almost exclusively use DHT as it works the most consistently and involves less computational effort.

Sampling

Let $\mathbf{S} \in \mathbb{R}^{\tilde{m} \times \tilde{m}}$ be a random diagonal matrix:

$$\mathbf{S}_{ii} = \begin{cases} 1 & , \text{ with probability } \gamma n/\tilde{m} \\ 0 & , \text{ with probability } 1 - \gamma n/\tilde{m} \end{cases}$$

$\mathbf{M}_s = \mathbf{SM}$ ▷ Row Sampling

$\mathbf{M}_s = \mathbf{QR}$ ▷ QR preconditioning

After the row mixing step we sample the rows.

Let γn be the number of sampled rows, where γ is some parameter greater than 1. Then we sample row i with probability $\frac{\gamma n}{\tilde{m}}$, and we **DON'T** sample that row with probability $1 - \frac{\gamma n}{\tilde{m}}$.

Sampling

Theorem 2. *Let A be an $m \times n$ full-rank matrix, and let $\mathcal{S}$ be a random sampling operator that samples $r \geq n$ rows from A uniformly. Let $\tau = C\sqrt{m\mu(A)\log(r)/r}$ where C is some constant defined in the proof. Assume that $\delta^{-1}\tau < 1$. With probability of at least $1-\delta$ the sampled matrix $\mathcal{S}A$ is full rank, and if $\mathcal{S}A = QR$ is a reduced QR factorization of $\mathcal{S}A$, we have*

$$\kappa(AR^{-1}) \leq \frac{1+\delta^{-1}\tau}{1-\delta^{-1}\tau}.$$

Theorem 2 shows that if we pick γ properly, the condition number of that preconditioner being created is upper bounded. In the paper they mentioned in theory, $\Theta(n\log(m)\log(n\log(m)))$ rows are sufficient to obtain $\kappa = \mathcal{O}(1)$. The preconditioner is the R matrix calculated from the reduced QR factorization of matrix SM.

Solving

$\tilde{\kappa} \longleftarrow \kappa_{\text{estimate}}(\mathbf{R})$
if $\tilde{\kappa}^{-1} > 5\epsilon_{\text{machine}}$ **then**
  $\mathbf{y} = \arg\min_{\mathbf{z}} \left\| \mathbf{A}\mathbf{R}^{-1}\mathbf{z} - \mathbf{b} \right\|_2$ ▷ Preconditioned iterative solve
  Solve for $\mathbf{R}\hat{\mathbf{x}} = y$ using LSQR/LSMR
  Return $\hat{\mathbf{x}}$
else
  if # iterations > 3 then **then**
    Failure, solve using Baseline Least squares(LAPACK) and return

These preconditioners can fail to generate a full rank matrix when they are sampled, and thus they suggest allowing for 3 attempts at preconditioning before they assume something is wrong with the original matrix and fall back on LAPACK (a standard software library for numerical linear algebra).

Otherwise, use a random sample of rows and use LSQR/LSMR to solve the system.

Time Complexity

Runtime of the row-mixing phase is $\mathcal{O}(mn \log m)$

Runtime of QR phase is $\mathcal{O}(n^3)$

Runtime of each iteration of LSQR is $\mathcal{O}(mn)$

The worst case scenario for run time is when the algorithm runs its max iterations of preconditioners and still fails to condition the matrix to within the desired bound, which is,

$$\mathcal{O}(mn \log m + n^3) + \mathcal{O}(\text{solve Least Square use LAPACK})$$

System Set up

The experiments published in the paper use matrices generated at random

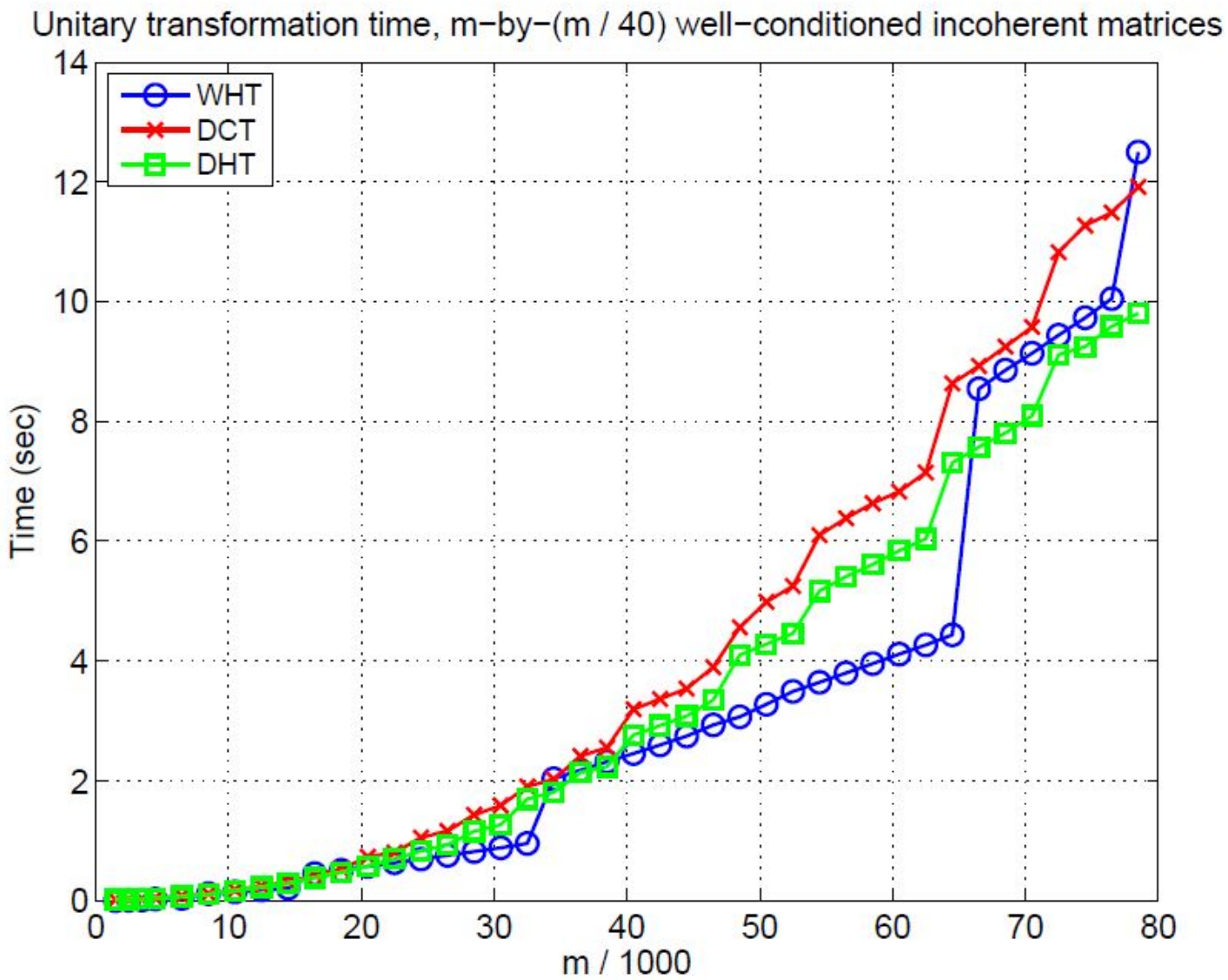
$A \in \mathbb{R}^{m \times n}$ is assumed to be non-singular

$m \geq n$ is assumed

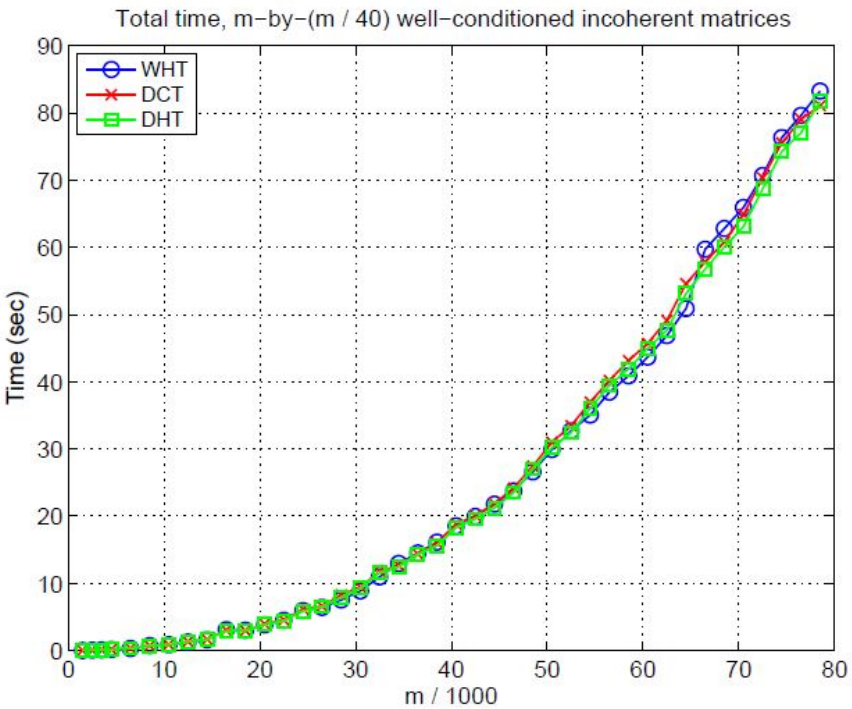
Random matrices were always generated using MATLAB's rand function (random, independent, uniform numbers)

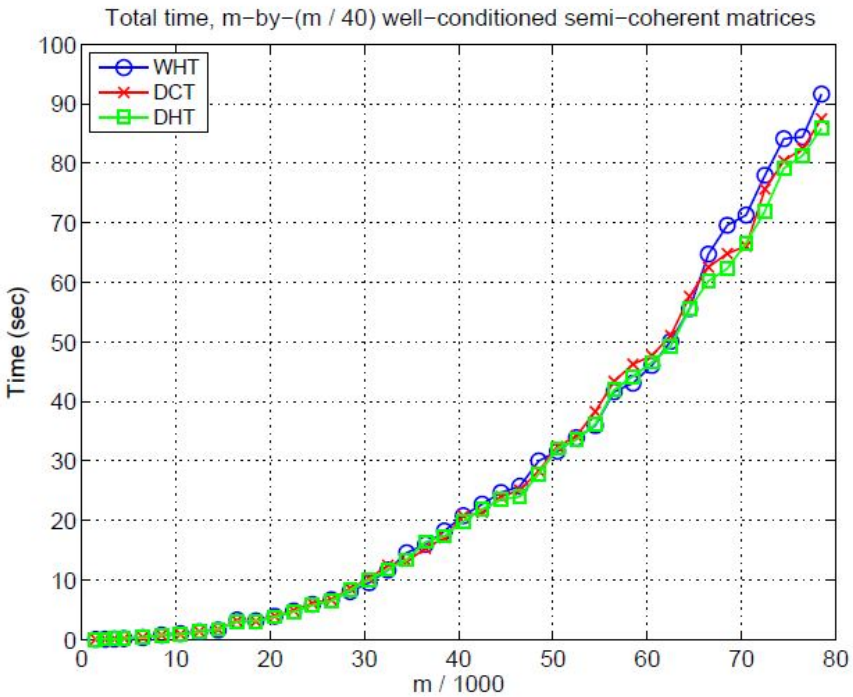
Two matrix sizes were used: 30000×750 and 40000×2000

Paper's Result: Unitary Transformation time

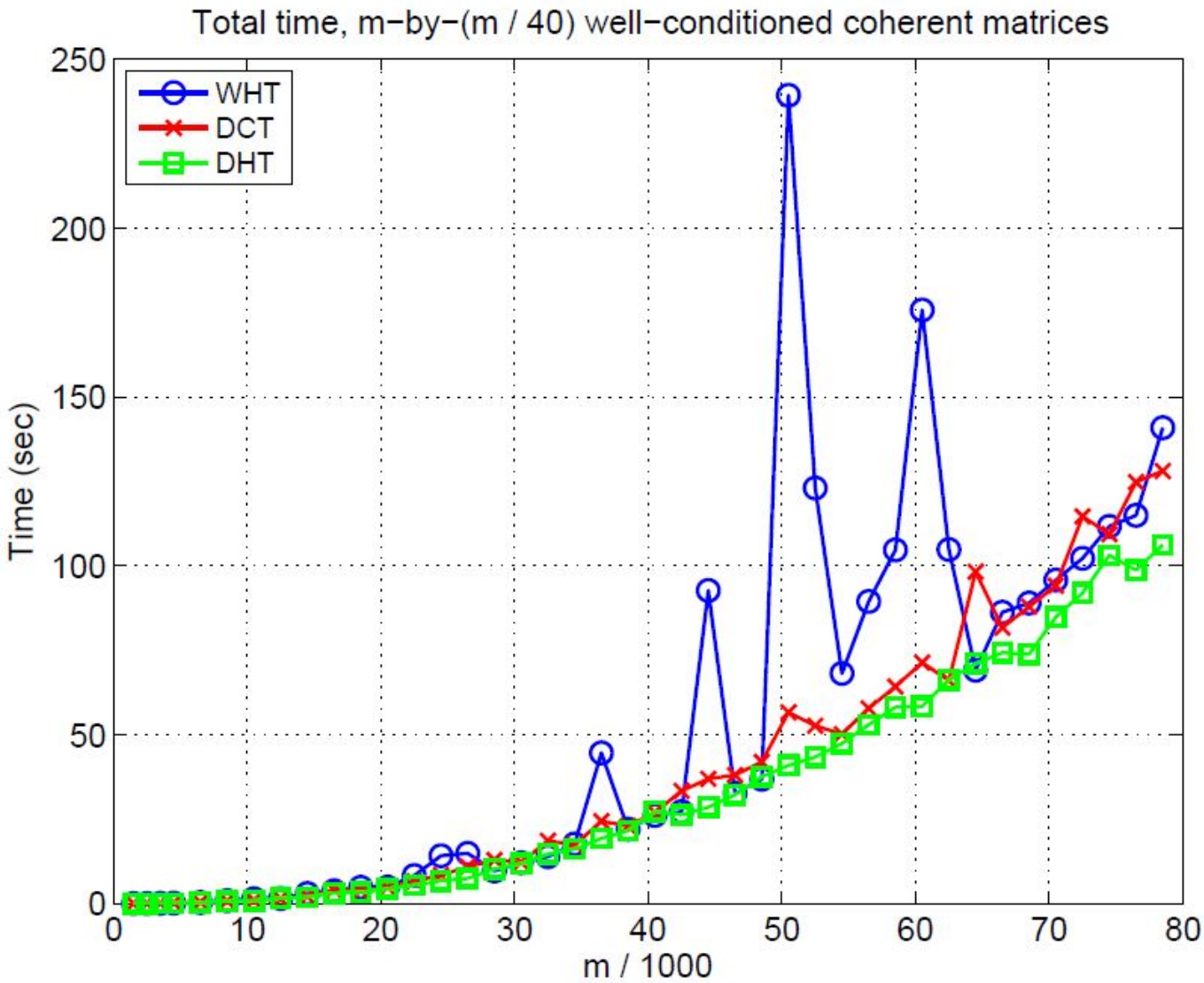


Paper's Result: Total Running Time of Algorithm

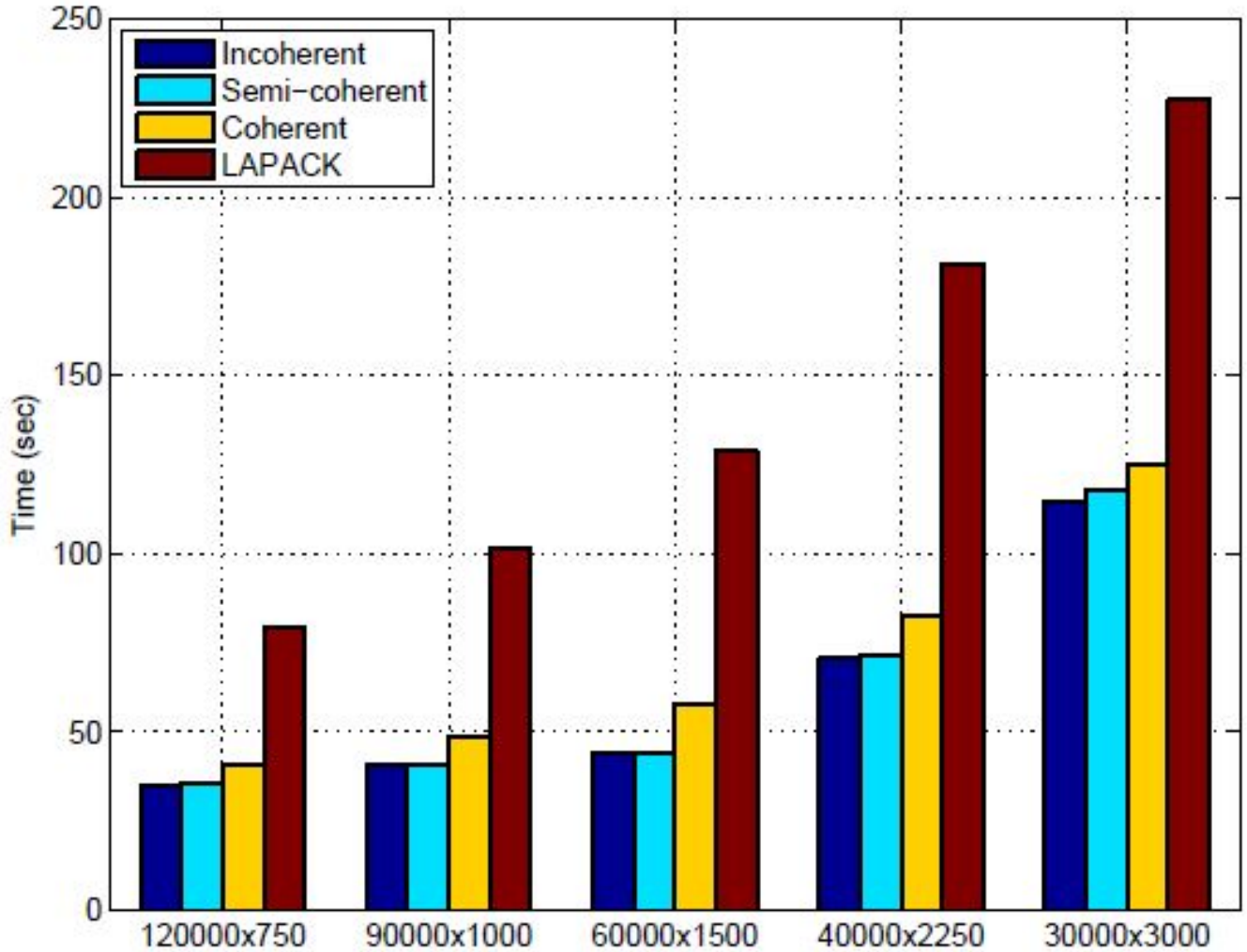




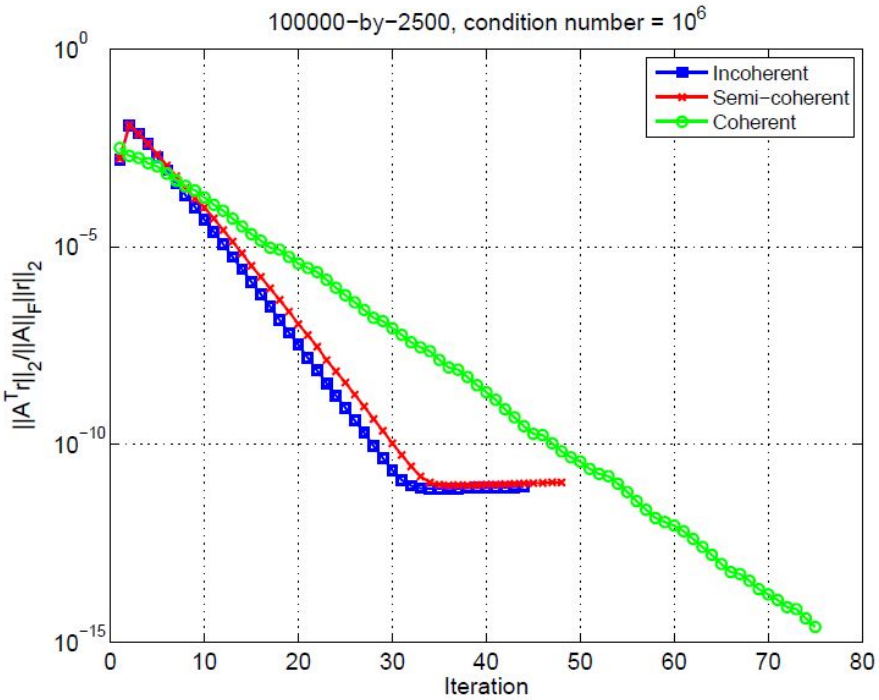
Paper's Result: Total Running Time of Algorithm (Cont.)

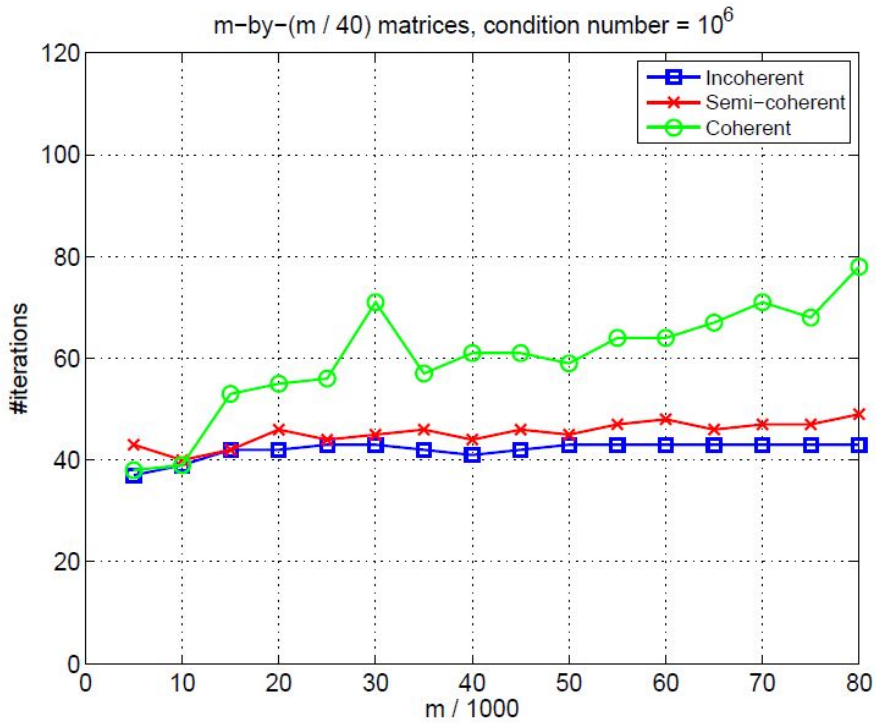


Paper's Result: Running time of Blendenpik vs LAPACK



Paper's Result: Convergence rate experiments





Paper's Result:

The paper also covered:

A breakdown of the running time of the algorithm: The row mixing pre-processing phase is not a bottleneck of the algorithm. Most of the time is spent on factoring the preconditioner and on LSQR iterations

Experiments examining strategies with no row mixing vs. the regular strategy: Results support the theory that no mixing runs faster on an incoherent input matrix.

Paper's Result: Critic

Positive:

Randomized algorithms can be effective in numerical linear algebra: Blendenpik beats LAPACK's direct dense least-squares solver by a large margin on essentially any dense tall matrix.

The algorithm is robust and predictable

The algorithm works very well on increasingly ill-conditioned matrices showing almost no increase in running time

Negative:

Real Data: All the experiments of the paper were done on randomly generated data that are uniformly distributed. At least one experiment in a real dataset would be preferable.

The experiments do not address how Blendednpik performs in sparse matrices and tiny matrices (nearly square ones). They only mention that in these cases LAPACK performs better.

Our Experiments: Speed vs. Aspect Ratio

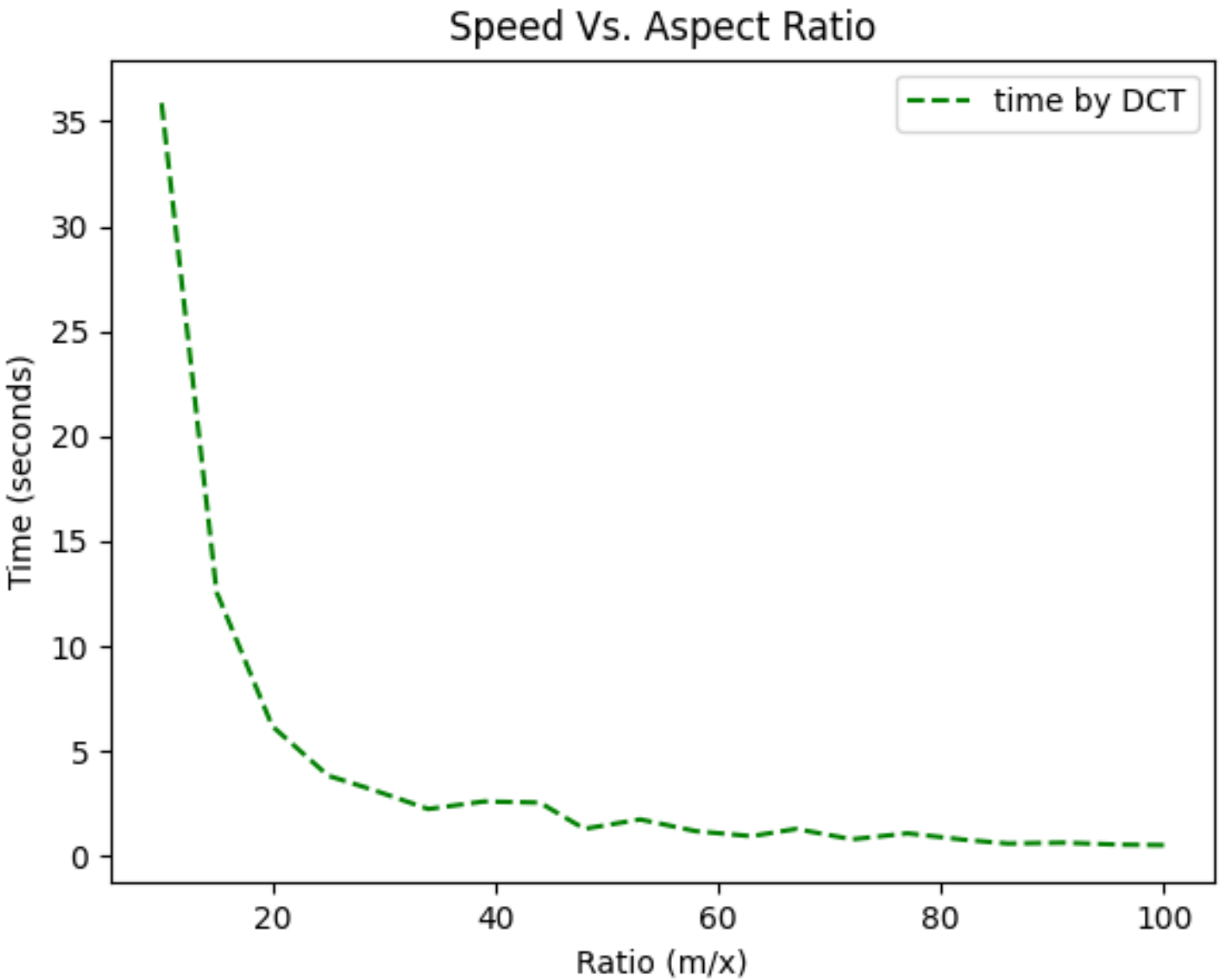
We test how the aspect ratio of the system (n/m) affects the speed of Blendenpik.

There are obvious consequences of increasing this ratio. All things being equal, increasing the ratio will increase the time it takes to run the solver.

We look for certain ratios around which Blendenpik works notably better or worse.

We also test the aspect ratio across different preconditioners

Results


Speed Vs. Aspect Ratio
time by DCT
Time (seconds)
0
5
10
15
20
25
30
35
20
40
60
80
100
Ratio (m/x)

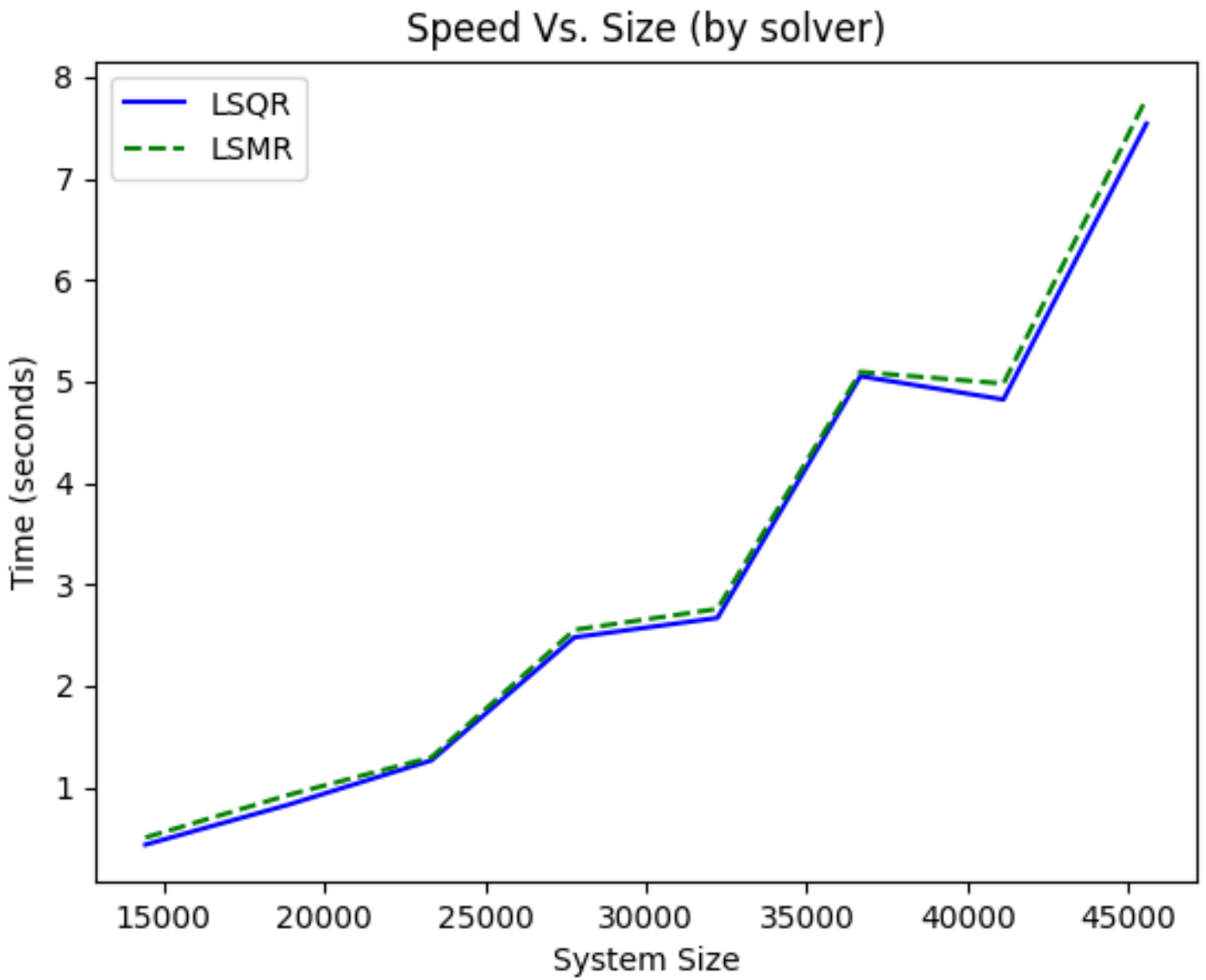
LSQR vs. Other Solvers

we test how different least square solvers influence the performance of Blendenpik in terms of time.

More specifically, we test LSQR and LSMR on various size of systems with a fixed aspect ratio.

LSMR, a relatively new algorithm based on Fong & Saunders algorithm in 2010, should perform faster than LSQR, which uses the Paige & Saunders algorithm for more than 30 years.

Results


Speed Vs. Size (by solver)
LSQR
LSMR
Time (seconds)
System Size
1
2
3
4
5
6
7
8
15000
20000
25000
30000
35000
40000
45000

Real Data

We intend to examine the performance of this algorithm on CIFAR-10 & CIFAR-100 image classification datasets

Both datasets contain 50000 training images with size 32*32, while CIFAR-10 labels images with 10 classes and CIFAR-100 labels with 100 classes

we will keep track of the time used by different solvers

we will provide statistical results like confusion matrix

We will analyze if this algorithm improves upon the current accepted implementations.

Time for solving with PureLSQR: 113.5

Time for solving with LSQR: 1169.9

Time for solving with LSMR: 435.6