Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Evaluating Novel Matrix-Vector Multiplication Strategies in the LSTRS and TRUSTμ Methods for Large-Scale Trust-Region Subproblems and Regularization.

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Kien Nguyen Hoang

Delft, the Netherlands
June 2014

MSc THESIS APPLIED MATHEMATICS

"Evaluating Novel Matrix-Vector Multiplication Strategies in the LSTRS and TRUSTμ Methods for Large-Scale Trust-Region Subproblems and Regularization"

Kien Nguyen Hoang

Delft University of Technology

Daily supervisor

Dr.M.D.L.C. Rojas Larrazabal

Responsible professor

Prof.dr.ir. A.W. Heemink

Other thesis committee members

Prof.dr.ir. C. Vuik

Dr.ir. M.B. van Gijzen

June 2014

Delft, the Netherlands

Abstract

In the regularization of ill-posed problems from image restoration, where we want to recover an image from blurred and noisy data, there are many methods that can be useful. Among those, the TRUSTμ method proposed in Rojas and Steihaug, 2002, is an efficient method for solving large-scale regularization problems with additional non-negativity constraints and is based on matrix-vector multiplication. The idea of TRUSTμ method is to solve a sequence of Trust-Region-Subproblems using the method LSTRS from Rojas, Santos and Sorensen, 2008.

The goal of this project was to evaluate the performance and accuracy of LSTRS and TRUSTμ method when the following three different approaches to compute matrix-vector multiplication were used: Fast Monte Carlo Algorithms, GPU computing, and parallel computing. In order to work on the experiments, we developed a MATLAB software implemented the TRUSTμ method and used the LSTRS software from Rojas, Santos and Sorensen, 2008. The TRUSTμ software provided an interface that allows the user to pass the input matrix as an array or as a function for computing matrix-vector products, possibly with parameters. One of the application of this software was in image restoration field to recover images from blurred and noisy data.

keyword: MATLAB, regularization, large-scale, TRUSTμ, LSTRS, matrix-vector product, Fast Monte Carlo algorithm, GPU computation, Parallel computing.

Contents

1 Introduction

From the regularization of ill-posed problems in image restoration concerning with very ill-conditioned matrices, we study TRUSTμ method for solving large-scale non-negative regularization problems . The method is an interior-point iteration that requires the solution of a large-scale trust-region-subproblem (LSTRS) in each iteration. TRUSTμ method is based on matrix-vector multiplication and the input matrix is either in form of an array or a function to compute matrix-vector products, possibly with parameters.

The purpose of this project was to evaluate the accuracy and performance of the TRUSTμ and LSTRS method when using these three approaches to compute the matrix-vector products: Fast Monte Carlo Algorithms, GPU computation, and parallel computing. Moreover in this project, we also developed a MATLAB software implementation TRUSTμ method, in which provided an interface that allows the users to choose how to pass the system matrix: explicitly as an array or implicitly as a function to compute the matrix-vector multiplication with structured parameters.

In this paper, section 2 will include the theoretical background, derive the problem to get to TRUSTμ method, present the Fast Monte Carlo Algorithm with three sampling techniques, describe the GPU computation and parallel computing. Section 3 will present the numerical results of MATLAB implementation of the TRUSTμ method and three approaches to compute matrix-vector products, tested on Regularization and Image restoration problems. Finally, Section 4 will give the conclusions and suggest future developments.

2 Theoretical Background

In Rojas and Steihaug, 2002 [8], the problem is considered as follows:

$$\begin{aligned} &min\frac{1}{2}\|Ax-b\|^2 \\ &s.t.\|x\| \leq \Delta \end{aligned} \tag{1}$$

where $A \in R^{mxn}, m \geq n, b \in R^m \text{and} \Delta > 0$. Throughout the paper, $\|.\|$ denotes the l_2-norm. Let us assume that m and n are large, and that the matrix A might not be explicitly available, but that we know how to compute the action of A and A^T on vectors of appropriate dimensions.

In the problem (1), the matrix A is a discretized version of a blurring operator, b is a vector representation of a degraded image, and Δ is a positive scalar. In image restoration problems, the norm constraint is a so-called regularization term that controls the growth in the size of the least-squares solution observed in most ill-posed problems with noisy data, and the non-negativity constraints reflect the fact that each component of the vector x represents either the color value, or the light-intensity value, of a pixel in the digital representation of the image, and therefore must be non-negative.

In the past, most techniques for image restoration did not take into account the non-negativity constraints. Instead, they solve the regularization problem, for example, the least-squares problem with the norm constraint only, and set to zero the negative components in the solution. This strategy clearly introduces some errors, but produces satisfactory results in certain cases, such as when the images are normal photographs. Nowadays, the techniques have changed from time to time; the non-negativity constraints have proved to be useful with the additional advantages.

Problem (1) always has a solution, which is unique when A has full rank. The optimal conditions were derived to satisfy the solutions of problem (1). Let $\lambda \in R$ and $y \in R^n$, then the Lagrangian function associated with the problem is:

$$L(x, \lambda, y) = \tfrac{1}{2}x^T A^T Ax - (A^T b)^T x + \tfrac{1}{2}b^T b - \tfrac{\lambda}{2}(\|x\|^2 - \Delta^2) + y^T x$$

and the Karush-Kuhn-Tucker (KKT) first order necessary conditions for a feasible point x and Lagrange multipliers λ and y to be a solution of problem (2.1) are:

(i) $A^T Ax - A^T b - \lambda x + y = 0$
$\iff (A^T A - \lambda I)x = A^T b - y$

(ii) $\lambda(\|x\|^2 - \Delta^2) = 0$

(iii) $y^T x = 0$

(iv) $\lambda \leq 0, y \leq 0$

The vector of Lagrange multipliers (or dual variables) y and the duality gap $y^T x$ will play a key role in the TRUSTμ algorithm, which will be presented later in section 2.1.

Since problem (1) is a convex quadratic problem, the KKT conditions are both necessary and sufficient.

Define $H = A^T A$ and $g = -A^T b$.

Now, problem (1) is equivalent to

$$\begin{aligned} & min \frac{1}{2} x^T H x + g^T x \\ & s.t. \|x\| \leq \Delta \\ & x \geq 0 \end{aligned} \tag{2}$$

In order to develop the TRUSTμ method for problem (2), we first eliminate the non-negativity constraints by restricting our attention to $x > 0$ and introducing a modified objective function by the logarithmic barrier function as following:

$$f_\mu(x) = \tfrac{1}{2} x^T H x + g^T x - \mu \textstyle\sum_{i=1}^n log \xi_i$$

where $x = (\xi_1, \xi_2, , \xi_n)^T$ and $\mu > 0$ is the so-called barrier or penalty parameter. The reason for choosing the logarithm function as the barrier is when the value of x is small or close to zero, the logarithm function will be minus infinity, which makes the term $-\mu \sum_{i=1}^n log \xi_i$ very large and any minimazion method will avoid those points. Therefore, the barrier will keep the function away from the small or zero values of x.

The modified function yields a family of problems depending on μ, given by:

$$\begin{aligned} & min f_\mu(x) = \frac{1}{2} x^T H x + g^T x - \mu \sum_{i=1}^n log \xi_i \\ & s.t. \|x\| \leq \Delta \end{aligned} \tag{3}$$

And the Lagrangian function is:

$$L(x, \lambda) = f_\mu(x) - \tfrac{\lambda}{2}(\|x\|^2 - \Delta^2)$$

Where $\lambda \in R$. Let X = diag(x) ,as a function in MATLAB, is a square matrix with values of x on the diagonal. The KKT conditions for a feasible point x and Lagrange multiplier λ to be a solution of problem (3):

(i) $(H - \mu X^{-2} - \lambda I)x = -g$

(ii) $\lambda(\|x\|^2 - \Delta^2) = 0$

(iii) $\lambda \leq 0$

The idea of the method is then to solve a sequence of problems of type (4), while decreasing the parameter towards zero. By using problem (4), we have restricted the solution to have positive components only.

A further simplification is now presented by substituting the non-linear barrier problem (3) by quadratically constrained quadratic problems, or trust-region subproblems, where the objective function will be a quadratic approximation to the logarithmic barrier function, and where the trust-region radius Δ will remain fixed. The subproblems are constructed as follows.

Consider the second-order Taylor expansion of f_μ around a point x,

$$q_\mu(x+h) = f_\mu(x) + \nabla f_\mu(x)^T h + \tfrac{1}{2} h^T \nabla^2 f_\mu(x) h$$

With:

$\nabla f_\mu(x) = Hx + g - \mu X^{-1} e$

$\nabla^2 f_\mu(x) = H + \mu X^{-2}$

On the other hand, we examine the function min $q_\mu(x+h)$ by setting $z = x + h$, notice that x is fixed and h is oscillated, thus the part including x can be dropped in the minimize function.

We obtain a new trust-region subproblem, given by:

$$\begin{aligned} & min f_\mu(x) = \frac{1}{2} z^T (H + \mu X^{-2}) z + (g - 2\mu X^{-1} e)^T z \\ & s.t. \|z\| \leq \Delta \end{aligned} \tag{4}$$

The necessary and sufficient conditions for a feasible point z and Lagrange multiplier $\lambda \in R$ to be a solution of the problem above are:

(i) $(H + \mu X^{-2} - \lambda I) z = 2\mu X^{-1} e - g$

(ii) $(H + \mu X^{-2} - \lambda I)$ is positive semidefinite

(iii) $\lambda(\|x\|^2 - \Delta^2) = 0$

(iv) $\lambda \leq 0$

Note that (ii) always holds in the convex case.

Our method consists of solving a sequence of problems of type (5) for z, for different values of μ and x, while driving the barrier parameter μ towards zero, and preserving positive iterative.

2.1 TRUSTμ method

In this section, the method TRUSTμ will be presented. First, the algorithm will be shown to give an overview of the method. Then we will explain the details of the algorithm, for example the choice of the initial values, range of stopping criteria and presenting the linesearch.

2.1.1 Algorithm TRUSTμ

Input: $H \in R^{nxn}$, symmetric, or routine to compute H times a vector;
$g \in R^n, \Delta > 0, \epsilon_x, \epsilon_y, \epsilon_f, \sigma \in (0, 1)$
Output: x^* satisfying (4) - condition (i), for μ close to zero.

1. Choose $x_0 > 0, \mu_0 > 0$, set k = 0.

2. While (not convergence) do:

2.1 Solve $P(x_k, \mu_k) for z_k$

2.2 Set $h_k = x_k - z_k$

2.3 Compute β_k such that $x_k + \beta_k h_k > 0$

2.4 Set $x_{k+1} = x_k + \beta_k h_k$

2.5 Compute μ_{k+1} such that $\mu_k \to 0$

2.6 Set $k = k + 1$

end while

The input for the algorithm is the symmetric matrix H, which in many cases is large and ill-conditioned. Thus, the algorithm has an alternative option is to provide the routine to compute matrix H times a vector. The vector g and scalar Δ is provided for the constraint optimization. Last but not least, $\epsilon_x, \epsilon_y, \epsilon_f, \sigma$ are used for the stopping criteria of the method.

The first step of the algorithm is to choose initial value x_0, μ_0, these two values are solutions from trust-region-subproblem which can be solved by the LSTRS method and will

be described later in section 2.1.4. Meanwhile in the step 2.1 of the algorithm, $P(x_k, \mu_k)$ denotes the problem (4). The details in solving $P(x_k, \mu_k) for z_k$ will be presented in section 2.1.3.

2.1.2 Update the barrier parameter μ

We can derive a formula for updating the barrier parameter by first computing an approximation to the dual variables y with y satisfies

$y = -(H - \lambda I)x - g$

Notice that a solution z, λ of the trust-region subproblem will follow

$-(H - \lambda I)z - g = \mu X^{-2} z - 2\mu X^{-1} e$

Now define $\tilde{y} = -(H - \lambda I)z - g$

Thus, $\tilde{y}$ is now used as an approximation to y. Also we can compute $\tilde{y}$ as following

$\tilde{y} = \mu(X^{-2} z - 2X^{-1} e)$

When x = z, then we have the approximation to the duality gap

$\tilde{y} = \mu(X^{-2} z - 2X^{-1} e) = \mu(-X^{-1} e)$

$\Rightarrow \tilde{y}^T x = -\mu n$

Which leads to the following formula for μ

$\mu = \frac{1}{n} \tilde{y}^T x$

In practice, the formulation using for updating μ is

$\mu_{k+1} = \frac{\sigma}{n} |\tilde{y}^T x|$

With $\sigma \in (0, 1)$, and $x = x_k$ or $x = x_{k+1}$, with $\tilde{y} = \mu(X^{-2} z - 2X^{-1} e)$ for $\mu = \mu_k, x = x_k$ or $x = x_{k+1}$ and $z = z_k$

2.1.3 Solving $P(x_k, \mu_k)$ for z_k :

$P(x_k, \mu_k)$ denotes the problem from (2.4), which is finding the solution for:

$$\min f_\mu(x) = \frac{1}{2} z^T (H + \mu X^{-2}) z + (g - 2\mu X^{-1} e)^T z$$

$$\text{s.t. } \|z\| \leq \Delta$$

with the input vector x, the matrix X is formed by $X = diag(x)$, and μ tends to go to zero, we have to find z. To solve the problem (4), we need a method which will compute the multiplier when solving the problem, as well as accept the matrix-vector product as input. Possibly, the candidate methods can be Steihaug [1983] and Toint [1981], Sorensen[1997], or Rojas[2000]. As we can see problem (4) is in the form of Trust Region Subproblem [8], and the solution for the problem also needs to deal with large scale issue. Therefore, in this approach, the Lagre-Scale Trust-Region Subproblems (LSTRS) is chosen to be the method for solving problem (4).

2.1.4 Choice of initial value x_0, μ_0

To compute initial values for x and μ , we first solve the trust-region subproblem without the non-negativity constraints:

$\min \frac{1}{2}x^T Hx + g^T x$

s.t. $\|x\| \geq \Delta$

Denote the solution to this problem and the corresponding Lagrange multiplier by x_{TRS} and λ_{TRS} respectively. x_{TRS} and λ_{TRS} will satisfy the options:

$(H - \lambda_{TRS} I)x_{TRS} = -g$
with $H - \lambda_{TRS} I$ positive semidefinite, $\lambda_{TRS} < 0$ and $\lambda_{TRS}(\Delta - x_{TRS}) = 0$.
We use x_{TRS} and λ_{TRS} to compute the initial value for μ as following:
We first compute an approximate initial value for the dual variables y as:

$\tilde{y}_0 = -g - (H - \lambda_{TRS} I)x_{TRS}$
then compute μ_0 as:

$\mu_0 = \frac{\sigma}{n}|\tilde{y}_0^T x|$
We then choose x_0 as either $x_0 = \|x_{TRS}\|$ with zero components replaced by a small positive value, or $x_0 = x_{TRS}$ with negative and zero components replaced by a small positive value, so $x_0 > 0$.
We use x_0 to test for convergence as later described in section 2.1.6.

2.1.5 Line search

A linesearch is necessary to ensure that the iterates x_k remain positive, since there is no guarantee that z_k computed in step 2.1 of the algorithm will have only positive components. The $(k+1)^{th}$ iterate is computed as, $h_k x_{k+1} = x_k + \beta_k h_k = z_k - x_k$ and

$\beta_k < min_{i,s.t.1 \leq i \leq n, \zeta_i \leq 0} \frac{\xi_i}{|\eta_i|}$
where $x_k = (\xi_1, \xi_2, ..., \xi_n)^T, z_k = (\zeta_1, \zeta_2, ..., \zeta_n)^T, h_k = (\eta_1, \eta_2, ..., \eta_n)^T$.
In practice we use the following safeguarded formula to update the iterates:

$x_{k+1} = x_k + \min \{ 1, 0.9995 \beta_k \} h_k$

2.1.6 Stopping criteria

The stopping criteria rely on the change in value of the objective function, the proximity of the iterates and the size of the duality gap. For the latter, we compute $\tilde{y}_k$ by

$\mu(X^{-2}z - 2X^{-1}e)$, with $\mu = \mu_{k-1}$ or $\mu = \mu_k$, $x = x_{k-1}$ or $x = x_k$ and $z = z_{k-1}$ computed in step 2.1 of algorithm 2.1.1.

Let $f(x) = \frac{1}{2}x^T Hx + g^T x$, and $\epsilon_x, \epsilon_y, \epsilon_f \in (0, 1)$, then for $f \leq 1$, algorithm 2.1.1 proceeds until

$|f(x_k) - f(x_{k-1})| \leq \epsilon_f |f(x_k)|$

Or $\|x_k - x_{k-1}\| \leq \epsilon_x \|x_k\|$

Or $|\tilde{y}_k^T x_k| \leq \epsilon_y \|x_k\|$

The purpose of the stopping criteria is to test the relative error as well as the duality gap. These values define the correctness of the solution. For $k = 0$ we only check the last condition for the initial values of x and $\tilde{y}$m for example $|\tilde{y}_0^T x_0| \leq \epsilon_y \|x_0\|$ with $\tilde{y}_0$ and x_0 as in section 2.1.3.

2.2 Matrix-vector Multiplication Approaches

In TRUSTμ method, one of the inputs of the algorithm is the matrix H. In the software, providing the matrix H has two options: the matrix H is given in array form (as for sparse matrix, the data is given in arrays of data values and indices), or the other option is to provide the function for matrix-vector multiplication. These options allow the software to handle the data flexibly and to get more efficient use of computational resources.

In this project, we would like to compute the matrix-vector multiplication on three approaches: Fast Monte Carlo Algorithm, GPU computation and Parallel Computing.

2.2.1 Fast Monte Carlo Algorithm for Matrices

Nowadays, large-size input data requires enormous computational ability to process. In modern computers, the growth of external memory capacity is enormous, while RAM and computing speeds increased at lower pace. In this application, as the data is formulated as large-scale matrix, we consider the algorithms which make more efficient use of computational resources, such as the computation time, memory, and the number of passes over the data. Monte Carlo algorithm appeared as one of the most simple, yet still powerful above other methods. Fast Monte Carlo algorithms aim to approximate the computations on large-scale matrices. In this section, we describe the algorithm for the approximation of the product between two matrices; this algorithm is called the BasicMatrixMultiplication (BMM) algorithm [1]. The BMM algorithm selects a subset of columns and corresponding

rows from two matrices and computes the estimated product.

As in [1], we use $A^{(j)}$ and $A_{(i)}$ to denote respectively the j^{th} column and the i^{th} row of a given matrix A, $\|.\|$ denotes the 2-norm, $\|.\|_F$ denotes the Frobenius norm and Pr will denote the probability.

Algorithm BASICMATRIXMULTIPLICATION :

Input: $A \in R^{mxn}, B \in R^{nxp}, c \in Z^+$ such that $1 \leq c \leq n$ and $\{p_i\}_{i=1}^n$ such that $p_i \geq 0$ and $\sum_{i=1}^n p_i = 1$

Output: $C \in R^{mxc}$ and $R \in R^{cxp}$ such that $AB \approx CR$

1. For t = 1 to c,

(a) Pick $i_t \in \{1, ..., n\}$ with $Pr[i_t = k] = p_k, k = 1, 2.., n$, independently and with replacement.

(b) Set $C^{(t)} = A^{(i_t)}/\sqrt{cp_{i_t}}$ and $R_{(t)} = B_{(i_t)}/\sqrt{cp_{i_t}}$

2. Return C, R.

3. $AB \approx CR$

The input of the algorithm is $A \in R^{mxn}, B \in R^{nxp}$, in general A and B are matrices, for the special case of matrix-vector product, B is a vector or can be consider as matrix nx1.

The product AB can be written as the sum of n rank-one matrices:

$$AB = \sum_{t=1}^n A^{(i_t)} B_{(i_t)}$$

As the matrix multiplication is formed in this style, the algorithm follows as: with a positive natural number c such that $1 \leq c \leq n$ and a probability distribution $\{p_i\}_{i=1}^n$, we pick a subset of columns from matrix A, and multiply with the corresponding set of rows in B, with an appropriate scaled term. The result from the computation approximated the multiplication between two matrices, A and B, shown as:

$$AB \approx CR = \sum_{t=1}^c C^{(t)} R_{(t)} = \sum_{t=1}^c \frac{1}{cp_{i_t}} A^{(i_t)} B_{(i_t)}$$

It is clearly that sampling column and row pairs give the approximated result with the scaling factor cp_{i_t}, if the t^{th} is sampled.

An important issue is choosing the scaling factor and the probabilities $\{p_i\}_{i=1}^n$. In the BASICMATRIXMULTIPLICATION algorithm, the scaling factor is chosen as $\frac{1}{\sqrt{cp_{i_t}}}$. There are many different approaches for the sampling probabilities, in this project, we offer three sampling techniques as follows:

(*) Uniform sampling: each column in A has an equal chance to be picked in the approximation. It mean $p_i = \frac{1}{n}$ with $i = 1, .., n$.

(*) Optimal sampling: proposed by Drineas, Kannan, and Mahoney, 2006, the optimal probabilities were defined as follows:

$$p_k = \frac{\|A^{(k)}\|\|B_{(k)}\|}{\sum_{k'=1}^{n} \|A^{(k')}\|\|B_{(k')}\|}$$

Optimal sampling minimizes the term $E[\|AB - CR\|_F^2]$, which implies the minimized upper bound of $\|AB - CR\|_F^2$ and $\|AB - CR\|_2^2$.

(*) Piecewise uniform sapling: proposed by Madrid,Guerra,and Rojas, 2012 [5]. This technique divides the interval $[1, n]$ into c subintervals, $c < n$, then uniformly select one column in each subinterval. Piecewise uniform sampling prefers small sample size. In the experiments done by Madrid,Guerra,and Rojas, 2012, it indicated that for affinity* matrices, piecewise uniform sampling showed approximation with higher accuracy for small sample size up to 25% of the input size.

Affinity: a matrix whose entries represent a measure of pairwise affinity or similarity of pixel in an image.

The computational cost to implement the BASICMATRIXMULTIPLICATION method depends on the sampling criterion. In the case of uniform sampling, only a single pass over the data is sufficient, and the requirement time and storage to sample A, B and construct C, R is $O(c(m + p))$. In other cases with two passes over the data, for both uniform and nonuniform sampling, the requirement is $O(c(m + n + p))$ [6].

2.2.2 GPU implementation

One main purpose of the project is to develop a MATLAB implementation the TRUSTμ method. By using MATLAB, the software may use the computers graphics processing unit

(GPU) for matrix operations. In this case, the execution in the GPU is faster than in the CPU, therefore using GPU would increase and improve the performance.

Send data to the GPU:

In MATLAB, gpuArray class represents data that is stored on the GPU. To transfer the data between MATLABs Workspace and the GPU, we use the function `gpuArray`. For example:

```
A = rand(100,100);
B = gpuArray(A);
```

B is the MATLAB `gpuArray` object that represents the data of the matrix size 100×100, generated by `rand()` function, stored on the GPU. The input data must be non-sparse, and it is either numerical or logical.

Retrieve Data from the GPU:

We use the `gather()` function to retrieve data from the GPU to the MATLAB workspace. This takes data that is on the GPU represented by a `gpuArray` object, and makes it available in the MATLAB workspace as a regular MATLAB variable. For example: `C = gather(B);`

Executing MATLAB Code on GPU:

There are two options to get MATLAB code to work on the GPU. The first one is to put the MATLAB code into a function file and run it directly on the GPU. The second option is to use the `gpuArray` objects to create, transfer, and handle data on GPU, with the built-in functions from the list in [3]. In this project, we use the second option as we will transfer the matrix data to the GPU, compute the matrix-vector multiplication, and then get the results back.

2.2.3 Parallel Computing

Besides the GPU, MATLAB provides another option to speed up operation by means of parallel computing. The idea of parallelization is to divide work into independent tasks and run them simultaneously. The Parallel Computing Toolbox consists of several high-level routines that take into account multicore processors.

To improve the performance of a program, we can create tasks in parallel by using `parfor` a parallel for-loop that can form independent tasks to multiple MATLAB sessions. A job manager is the part of the computer that coordinates the execution of jobs and evaluates

tasks. The job manager distributes tasks to the individual MATLAB sessions, also known as workers, for evaluation. The basic advantage of a parfor-loop in MATLAB is that: part of the parfor-loop is executed on the MATLAB client (where the `parfor` is issued) and the other part is executed in parallel on MATLAB workers. Because several MATLAB workers can be computing concurrently on the same loop, a parfor-loop can provide significantly better performance than a for-loop.

For example, the simple for loop in MATLAB:

```
for i = 1:10
y(i) = rand();
end
```

can be easily replaced by a *parfor* - loop as following:

```
parfor i = 1:10
y(i) = rand();
end
```

The difference is that: the for loop executes a series of statements (the loop body) over a range of values, while in parfor-loop, the loop body is executed in parallel, which will be much faster in execution. The necessary data on which `parfor` operates is sent from the client to workers, where most of the computation happens, and the results are sent back to the client and pieced together.

In order to tell MATLAB that the program runs in multiple processors, beside using `parfor`, we need to give the instruction to MATLAB by using `matlabpool`.
At the beginning of the code, we need to indicate:

```
matlabpool('open',nrOfCore);
```

where *nrOfCore* is the number of cores you want to use (max is number of cores in your PC). This instruction opens the parallel computing toolbox.
At the end of the code, type:

```
matlabpool('close');
```

This will close the parallel computing, however, the license will still be in use until you close your MATLAB session.

3 Numerical Experiments and Discussion

In this section, we will present a numerical study to investigate the performance and accuracy when we use different approaches for matrix-vector multiplication, such as the three Fast Monte Carlo Algorithms (FMCA), GPU computation, and Parallel computing. We performed the TRUSTμ experiments using a MATLAB prototype that we developed, and the LSTRS experiments using the public version in [7]. We organized the presentation into two parts: we will present the TRUSTμ prototype in Section 3.1, and the investigation of matrix-vector multiplication approaches in Section 3.2.
In our project, we mainly did the experiments on two test problems: the reguralization problem proposed in [6] and an image restoration problem. The regularization solutions were computed by solving the quadratically constrained least squares problem and the Trust-Region-Subproblems (TRS):

$$min\frac{1}{2}||Ax-b||^2 \quad \text{s.t. } ||x|| \leq \Delta \tag{5}$$

Matrix A is a discretized operator from an ill-posed problem, typically matrix A is very ill-conditioned. Problem (5) is equivalent to a trust-region problem with $H = A^TA$ and $g = -A^Tb$. We used the MATLAB routine **phillips** and **blur** from [2], which implemented the problem in [6] and blurring operation, respectively. For the image restoration problem, we used a black and white picture of an art gallery in Paris.

Problem phillips:
The test problem phillips from Regularization Tools is a discretization of the well-known Fredholm integral equation deviced by D. L. Phillips [6]. The dimension of the problem is $n \times n$ and the order of n must be multiple of 4 as indicated in [6]. The true solution to the inverse problem was computed by the routine **phillips** and was stored in a vector for later comparison. In this paper, we use two values for n: 300 and 1000.

Problem image restoration:
In image restoration problems, we would like to recover an image from blurred and noisy data. In this problem, the matrix A represents the blurring operator and was generated with the routine **blur** from Regularization Tools [2]. The routine **blur** added the atmospheric turbulence blur to the problem. The image deblurring problem arises in connection with the degradation of images by atmospheric turbulence blur, modelled by a Gaussian point-spread function:

$$h(x,y) = \frac{1}{2\pi\sigma^2}exp(-\frac{x^2+y^2}{2\sigma^2})$$

The matrix A is a symmetric matrix with dimension of n^2 x n^2 , stored in sparse format. The routine **blur** requires three input arguments: `n, band` and `sigma`. Only elements within a distance `band - 1` from the diagonal are stored. If `band` is not specified, then

`band` = 3 is used. The parameter `sigma` controls the amount of smoothing (the larger the `sigma`, the less ill-posed the problem). If `sigma` is not specified, then `sigma` = 0.7 is used.

In our experiment for image restoration problem, the 2D degraded image was resized to one dimension vector using the MATLAB routine `imresize`, the image was resized columnwise and was stored in vector b. The true solution vector contained a black and white picture of an art gallery in Paris. There is noise added to the picture, the noise vector s was generated as a random Gaussian vector using MATLAB routine `randn`. The noise level is defined as $\frac{||s||}{||b||}$. In our experiments, we use noise level of 10^{-2}. The image's dimension is 256x256, thus the size of A is 65536. Note that in the GPU experiments, we had to resize the image to 96x96 due to memory limitation.
In the remainder of the paper, MVP will denote the number of matrix-vector products that was used to compute the solutions.

3.1 TRUSTμ Prototype

We tested the TRUSTμ prototype on the two test problems: **phillips** of dimensions 300 and 1000 and image restoration problem of dimension 65536.
The experiments were performed in MATLAB R2011b on a Dell Desktop with a DuoCore, 3.16GHz processor, with 4GB of memory, running Debian GNU/Linux 7. The floating-point arithmetic was IEEE standard double precision with machine precision $2^{-52} \approx 2.2204 \times 10^{-16}$.

3.1.1 TRUSTμ performance and accuracy. Problem phillips, small and medium scale

In this experiment, we would like to investigate the performance and accuracy of TRUSTμ method on problem **phillips** using the TRUSTμ prototype that we developed. We used LSTRS as the Trust-Region Subproblem (TRS) solver for TRUSTμ. The LSTRS solution denoted in this section was the solution of LSTRS on problem **phillips**. It was also used as the initial value for TRUSTμ, with zero and negative elements were replaced by 10^{-5}.

Figure 1 illustrates the solutions computed with TRUSTμ and LSTRS. In the lower right corner of Figure 1, we can easily see that while the LSTRS solution has negative elements, TRUSTμ solution is non-negative. Both the TRUSTμ and LSTRS solutions were close to the true solution.

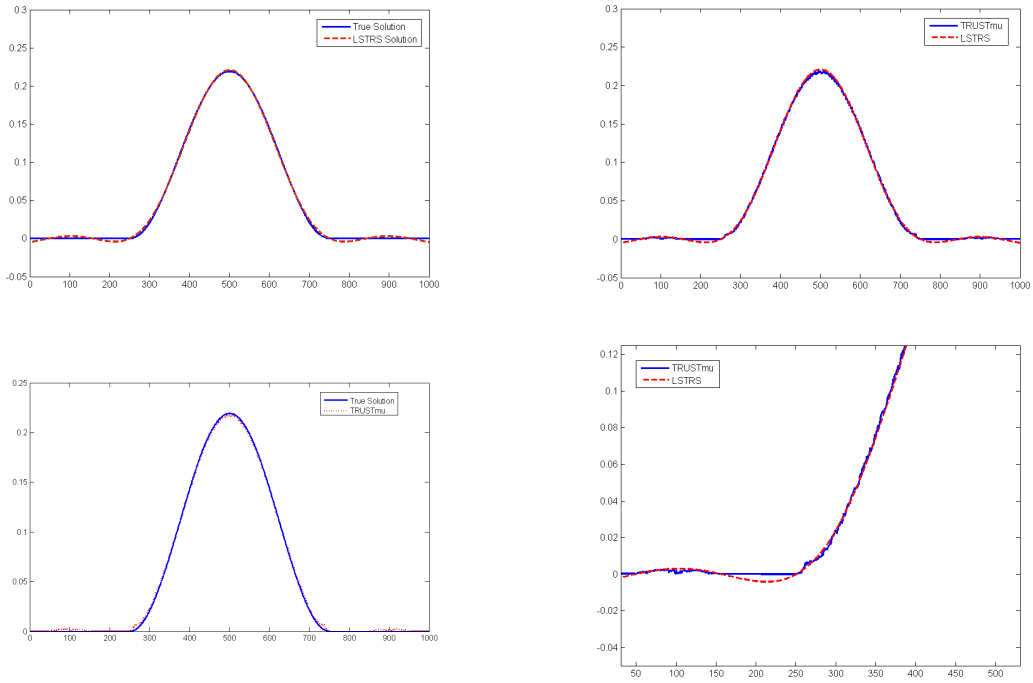


Figure 1: Problem **phillips** of dimension 1000, the solutions computed by TRUSTμ and LSTRS (dashed) and true solution (solid).

	Phillips dimension 300	Phillips dimension 1000
LSTRS mvp	156	342
LSTRS rel.error	3.83e-002	4.32e-002
TRUSTμ mvp	678	783
TRUSTμ rel.error	9.73e-003	1.77e-002

Table 1. Problem phillips: solutions of LSTRS and TRUSTμ in terms of number of matrix-vector multiplication(mvp) and relative error.

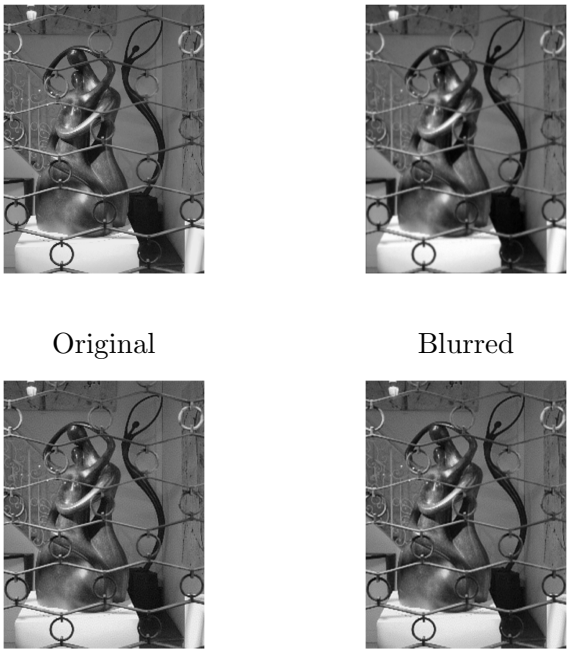
From Table 1, notice that TRUSTμ's relative error was less than LSTRS, in case of the dimension 300, the magnitude was in term of 10^{-3} in TRUSTμ and 10^{-2} in LSTRS. Meanwhile, it took more computation for TRUSTμ to compute to the solution than LSTRS. The reason is the initial value used in TRUSTμ was the of LSTRS solution with non-negative elements, it meant the total number of mvp in TRUSTμ included the mvp of LSTRS.

In conclusion, TRUSTμ, with non-negative solution, showed a lower relative error but took more computation than LSTRS, as the size of the problem increased, the more number of mvp was required to come to the solution.

3.1.2 TRUSTμ performance and accuracy. Problem image restoration, large scale

In this experiment, we would like to investigate the performance and accuracy of TRUSTμ method on image restoration problem using the TRUSTμ prototype that we developed. The image dimension was 256x256, thus the problem size was in large scale of 65536. As we knew that the data of an image must be non-negative, therefore, if a solution appeared to have negative elements, MATLAB automatically truncated these elements to zero, which caused the loss of image data. Hence, TRUSTμ with non-negative solution was expected to get more accuracy and lossless data in this image restoration problem.

We used LSTRS as the TRS solver for TRUSTμ. The LSTRS solution denoted in this section was the solution of LSTRS on the image restoration problem with same settings as TRUSTμ. It was also used as the initial value for TRUSTμ, with zero and negative elements was replaced by 10^{-5}.



Original Blurred

LSTRS TRUSTμ

Figure 2: Problem Paris of dimension 256x256, including the original picture, the blurred image and the two solution of LSTRS and TRUSTμ. The images were resized for visual purpose.

Figure 2 illustrated the solution of TRUSTμ and LSTRS. Visually, the two images from TRUSTμ and LSTRS were clearer than the blurred image and look similar to each other. Applying the command `min` and `find` on the two solutions to check the minimum and non-negative elements, with TRUSTμ solution, the minimum value was 1.21e-2, while LSTRS solution showed minimum equal zero and there were 17 elements that had zero value, which meant MATLAB truncated the corresponding negative elements to those zero values.

	LSTRS	TRUSTμ
mvp	201	995
rel.error	3.90e-2	4.19e-2
iteration	3	3

Table 2. Comparison problem Paris solutions of LSTRS and TRUSTμ in terms of number of matrix-vector multiplication(mvp), relative error and iteration.
In this experiment, since the problem size increased to 65536, TRUSTμ took greater of number of mvp to compute the solution than LSTRS did. TRUSTμ, to came up with the solution, took the LSTRS solution with negative and zero elements changed to small values as the initial value, then it took TRUSTμ three iterations to get to the final non-negative solution, in each iteration there were 201, 392 and 201 mvp respectively. Thus, in total, TRUSTμ took a number of mvp about four times larger than LSTRS.

3.2 Three Approaches To Compute Matrix-Vector Multiplication

We did our experiments on two test problems: phillips of dimension 1000 and image restoration problem of dimension 65536. For testing the implementation of Fast Monte Carlo Algorithms, various number of sample sizes were chosen, ranging from 10% to 50% of the problem size.

In the beginning, we would like to run the problems on one machine and make the comparison when running on CPU, on GPU and in Parallel. However, due to memory limitation, GPU computation only worked with problem that had small or medium scale, while in Parallel computing, in order to observe the speedup, the problem should be in large scale. Therefore, we decided to make two separated comparison: one comparison CPU with GPU, and one comparison CPU with Parallel computing.

3.2.1 Classic Matrix-Vector Multiplication and Monte Carlo - problem phillips with LSTRS and FMCA, medium scale with sample sizes from 10% to 50% of problem size

In this experiment, we would like to investigate the performance of LSTRS method on regularization problem phillips when the matrix-vector multiplication is implemented in different approaches: MATLAB built-in matrix-vector multiplication function and three Monte Carlo sampling techniques (uniform, optimal and piecewise uniform sampling - PWS). In this section, the MATLAB built-in matrix-vector multiplication function is denoted as classic or classical matvec, and the Fast Monte Carlo Algorithms is abbreviated as FMCA.

The experiments were performed on MATLAB R2011b, on a Dell Desktop with a Duo-Core, 3.16GHz processor, 4 GB of memory, running on Debian GNU/Linux 7 for GPU Computation, with GPU NVIDIA GT200 - Tesla C1060. The floating-point arithmetic was IEEE standard double precision with machine precision $2^{-52} \approx 2.2204 \times 10^{-16}$.

In our experiments, we would like to compute the Quasi-Optimal and Boundary solutions of the regularization problem phillips. Thus, the setting `lopts.interior` was set to *'n'*, indicated that no Interior solution should be computed. Since the FMCA with the randomize routine might lead to a *bad* approximation in matrix-vector product, which would cause the program to halt when solving the Interior solutions , therefore this setting is an important aspect to prevent the program running into these *bad* approximation.

Sample size	100			200		
Ave.results	rel.error	mvp	exe.time	rel.error	mvp	exe.time
Uniform	1.88e-1	284.3	1.55	1.81e-1	305.5	1.70
Optimal	1.73e-1	207.4	1.08	1.57e-1	217.3	1.11
PWS	3.09e-2	106.9	0.88	2.66e-2	97.3	0.93
Classic	4.32e-2	342.0	1.75	4.32e-2	342.0	1.75

Sample size	300			400		
Ave.results	rel.error	mvp	exe.time	rel.error	mvp	exe.time
Uniform	1.73e-1	314.2	1.93	1.67e-1	371.6	2.01
Optimal	1.48e-1	233.6	1.16	1.37e-1	185.8	1.07
PWS	1.29e-1	102.4	0.91	1.13e-1	70.4	0.46
Classic	4.32e-2	342.0	1.75	4.32e-2	342.0	1.75

Sample size	500		
Ave.results	rel.error	mvp	exe.time
Uniform	1.34e-1	176.2	1.07
Optimal	8.70e-2	159.6	1.02
PWS	4.71e-2	91.2	0.87
Classic	4.32e-2	342.0	1.75

Table 3. phillips problem dimension 1000, LSTRS solutions with Monte Carlo sampling techniques, sample size 100 to 500. Each sampling technique was ran for 100 times and average results were calculated.

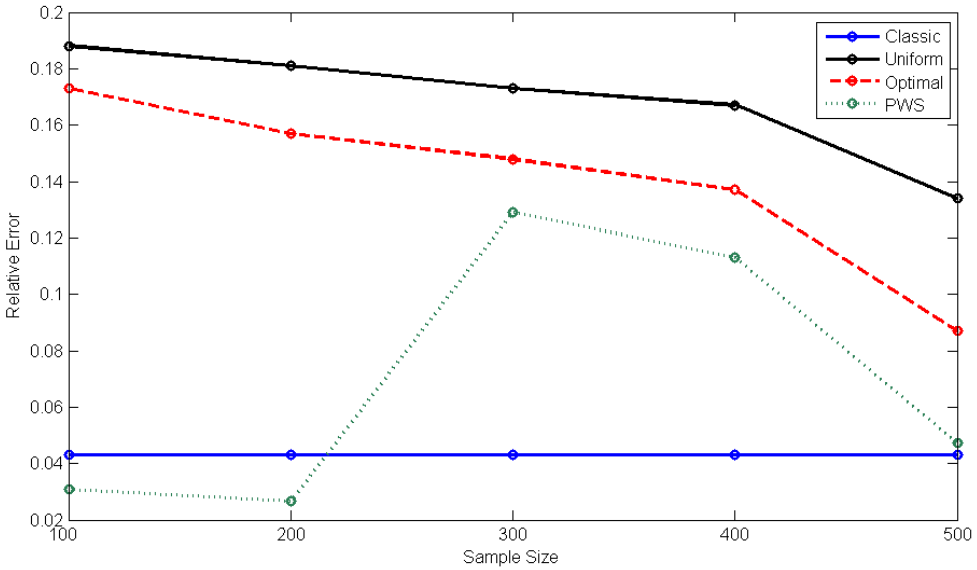
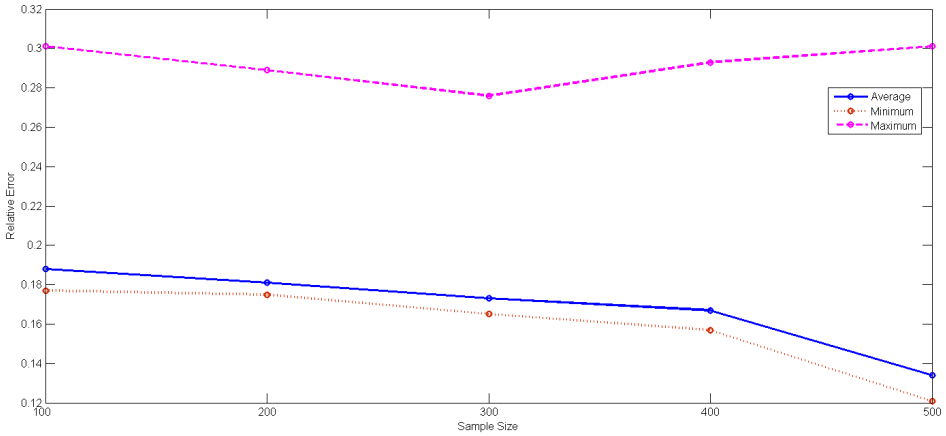
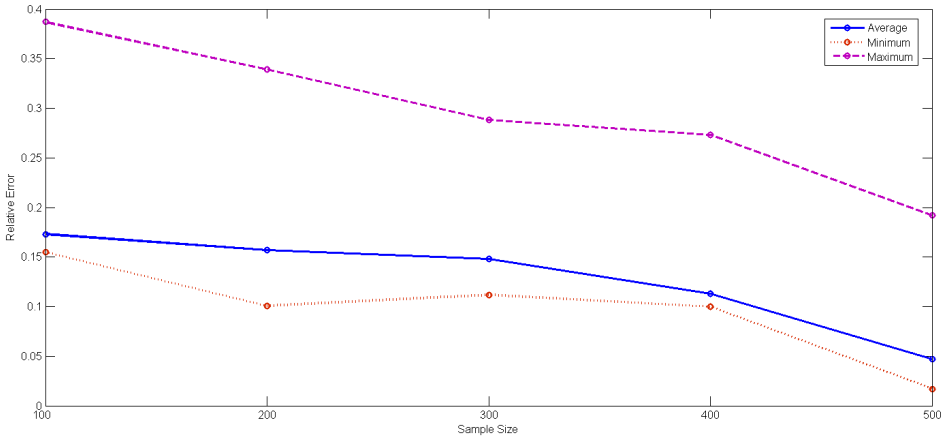


Figure 3: The average relative error of LSTRS solutions with FMCA on phillips problem dimension 1000. The sample size ranged from 100 to 500.

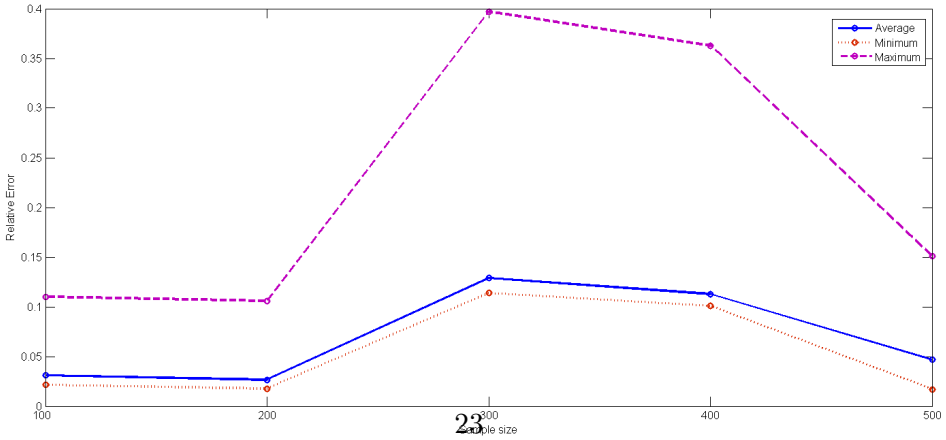
Figure 3 illustrated the average relative errors of LSTRS solution, in comparison of classical matvec with the three FMCA sampling techniques. Among the three FMCA, PWS had the lowest relative error. As the sample size increased, the relative errors of uniform and optimal got closer to classical matvec's. At sample size 100 and 200, we can see that the relative error of PWS was smaller than classic, which indicated that PWS had a better solution. This was consistent with the assumption in [5] that PWS performs better with small sample size from 10% to 20% of the problem size.



Uniform Sampling



Optimal Sampling



Piecewise Uniform Sampling

Figure 4: phillips problem, dimension 1000, the solutions computed by LSTRS and Monte Carlo sampling techniques, in comparison of maximum, minimum and average relative errors.

Figure 4 showed the relative errors of each FMCA sampling techniques, with sample size from 100 to 500, in terms of maximum, minimum and average. In all three results, we can see that the minimum solutions were closer to the average than the maximum. This indicated that the *bad* solutions - maximum relative errors were rare, and therefore our results were reliable.

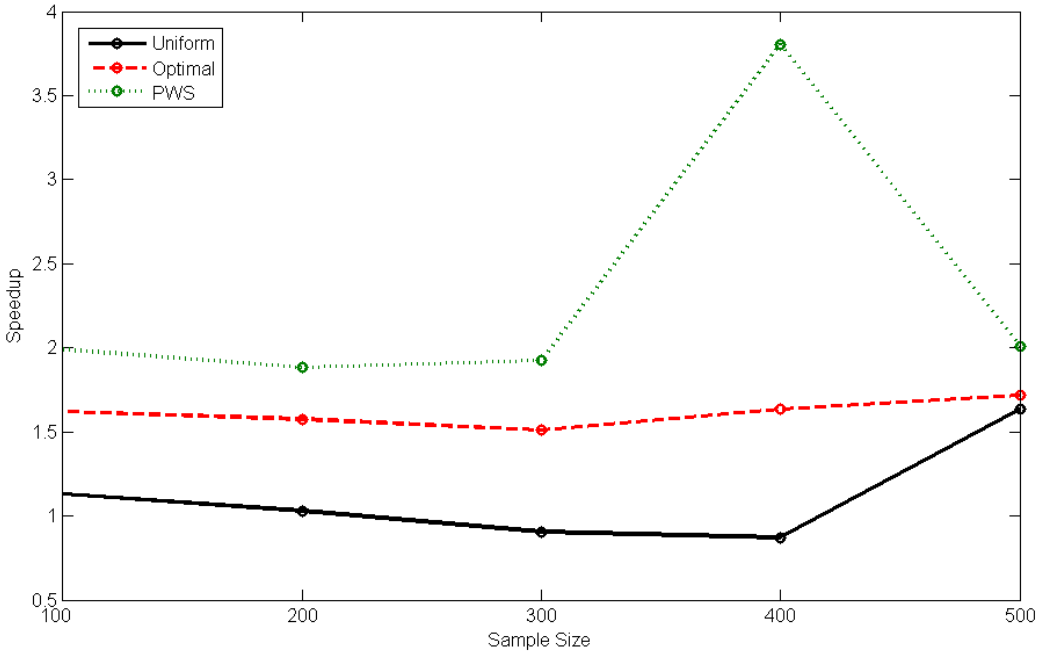


Figure 5: The speedup in execution times of LSTRS solutions with FMCA on phillips problem dimension 1000. The speedup is calculated as the different from execution times of classical matvec and FMCA. The sample size ranged from 100 to 500.

Figure 5 showed the speedup in execution time of LSTRS solutions with FMCA, for sample size from 100 to 500. Among the three techniques, PWS had the least execution time hence the greatest speedup, then came optimal. According to Table 3, it took less computation (number of mvp) for PWS and optimal to compute the solution than classic and uniform. At sample size 100 and 200, compared to classic, PWS took about one third of computation and about one half of execution time, yet yielded better solution.

In conclusion, consistent with the prediction from [5] that as matrix A was affinity, for small sample size from 10% to 20% of the problem size, PWS performed better than the other two techniques as well as better than classical matvec, with less computation, smaller

execution time and better solutions. Since the sample size increased, the relative errors of uniform, optimal and PWS tended to get closer to classical matvec's. Meanwhile, PWS still showed least computation and execution time, then came optimal, uniform and classic had the most number of mvp and execution time. Also, a conclusion from [5] indicated that with FMCA, if one of the matrix was ill-conditioned then optimal performed better than uniform, which we can clearly observe in this experiment.

3.2.2 Parallel Computing - image restoration problem with LSTRS and FMCA, large scale with sample sizes from 10% to 50% of problem size

Image restoration problem Paris with image dimension 256x256, problem size 65536. The original 'paris.jpg' image was blurred with **blur** function from Regularization Tools, Band 3, Sigma 1.5.
The experiment was performed on MATLAB R2011b, on a Dell Desktop with a QuadCore, 2.66Ghz processor, with 4GB of memory, running on Debian GNU/Linux 7. The floating-point arithmetic was IEEE standard double precision with machine precision $2^{52} \approx 2.2204 \times 10^{-16}$.
In this experiment, we would like to test the performance and accuracy of LSTRS on image restoration, when the matrix-vector multiplication was implemented with classic and FMCA. The test was run on single core, denoted as CPU or '1 Core', and on four cores, denoted as Parallel 4 Cores. The command `MATLABpool` was called to control the number of cores running with MATLAB, as in one experiment we compared the performance when running the problem on different number of cores: four, three and two cores.

The tables below represented the results of LSTRS and FMCA - both running on Parallel 4 cores and on CPU 1 core, the sample sizes were chosen equal to 15%, 25% and 50% of the problem size.

Sample size: 15% = 9830

Sampling	Uniform		Optimal		PWS		Classical	
	4 cores	1 core	4 cores	1 core	4 cores	1 core	4 cores	1 core
Relative Error	3.18e-001		2.83e-001		2.61e-001		1.06e-001	
Number of mvp	233.6		178.0		154.1		201.0	
Execution time	8.2	8.7	13.7	14.5	27.1	30.1	6.1	7.0
Speedup	1.06		1.06		1.11		1.15	
Iteration	3.7		2.7		3.3		3.0	
Nr of run	83		91		93		10	

Sample size: 25% = 16384

Sampling	Uniform		Optimal		PWS		Classical	
	4 cores	1 core	4 cores	1 core	4 cores	1 core	4 cores	1 core
Relative Error	2.57e-001		2.37e-001		1.81e-001		1.06e-001	
Number of mvp	173.2		321.0		109.8		201.0	
Execution time	7.3	8.0	15.7	17.1	16.7	18.2	6.1	7.0
Speedup	1.10		1.09		1.09		1.15	
Iteration	3.3		4.1		3.1		3.0	
Nr of run	93		91		100		10	

Sample size: 50% = 32768

Sampling	Uniform		Optimal		PWS		Classical	
	4 cores	1 core	4 cores	1 core	4 cores	1 core	4 cores	1 core
Relative Error	2.14e-001		1.94e-001		1.52e-001		1.06e-001	
Number of mvp	167.5		230.0		168.2		201.0	
Execution time	8.3	9.4	14.8	16.3	60.9	66.7	6.1	7.0
Speedup	1.13		1.10		1.10		1.15	
Iteration	3.3		4.1		3.1		3.0	
Nr of run	93		91		100		10	

Table 4. Image restoration problem Paris, solved with LSTRS and three approaches to compute matrix-vector multiplication: Monte Carlo sampling techniques (Uniform, Optimal and Piecewise Uniform Sampling - PWS) and built-in MATLAB matrix-vector multiplication operation (classical) - sample sizes were $15\%, 25\%$ and 50% of problem size.

Figure 6 illustrated the behavior of the relative error of three FMCA sampling techniques when increasing the sample sizes from $15\%, 25\%$ to 50%. All three techniques tended to get closer to classical matrix-vector multiplication's relative error when the sample sizes were increased. In comparison among the classical matrix-vector multiplication and the three Monte Carlo sampling techniques, classical matrix-vector multiplication had the lowest relative error, then came PWS, optimal and uniform was the worst. However, because this experiment was an image restoration problem and the relative error was of order $O(10^{-1})$, the visual pictures was one of the most important in consideration.

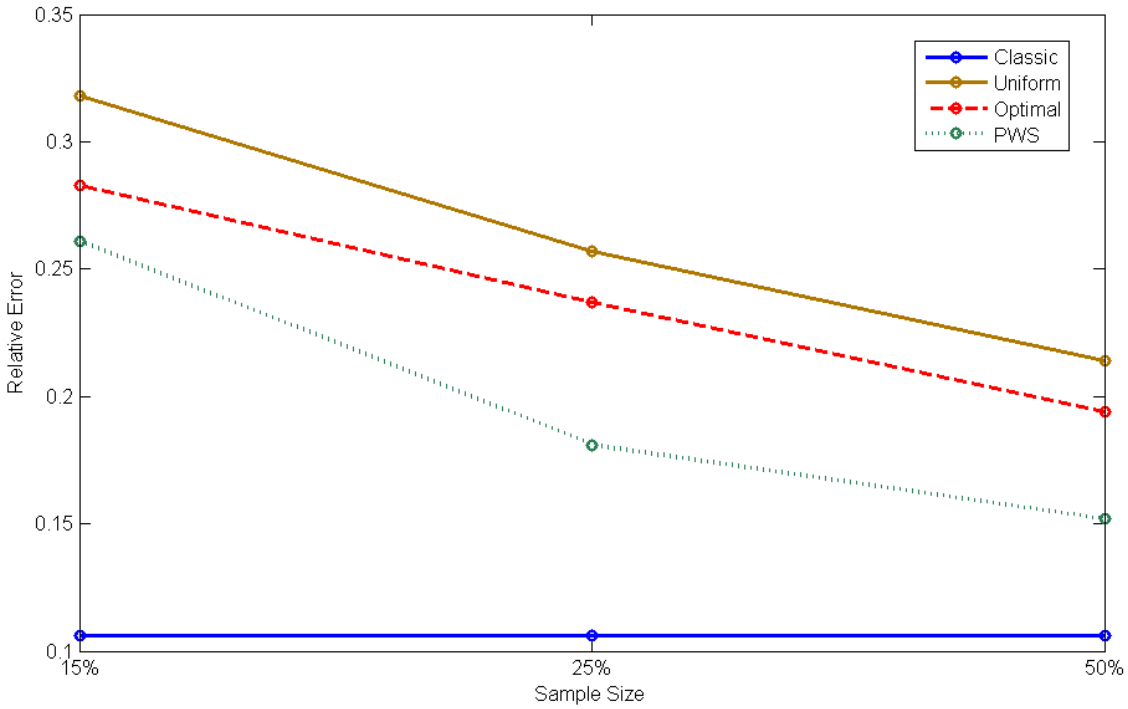
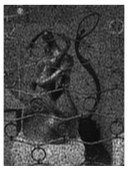


Figure 6: The relative error of three FMCA sampling techniques and classic. sample sizes were $15\%, 25\%$ and 50% of problem size.

Below are the worst and best pictures of the solutions.

Classic.
e=1.06e-1



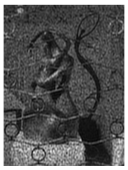
Uniform.
e=2.21e-1

Optimal.
e=1.99e-1

PWS.e=2.46e-
1

Figure 7: Sample size 50%. Maximum relative error.

Classic.
e=1.06e-1



Uniform.
e=2.10e-1

Optimal.
e=1.91e-1

PWS.e=1.44e-
1

Figure 8: Sample size 50%. Minimum relative error.

From Figure 7 and 8, we can observe the visual solutions of Paris problem, with classical matrix-vector multiplication and three Monte Carlo sampling techniques, including the best and worst solution for the sample size 50%. In all cases, classical matrix-vector multiplication showed the best visual picture among the techniques, while uniform and optimal showed pictures with lots of noise, and PWS showed clearer pictures but with blur. This observation was similar to an experiment in [5] indicated that PWS yielded an valid approximation that contained the data for performing an acceptable segmentation of the image, while the approximations obtained by uniform and optimal were not useful for segmentation purposes. Therefore, in our image restoration problem, uniform and optimal solutions contained noisy data, and PWS solutions got better result but still were a blurred image.

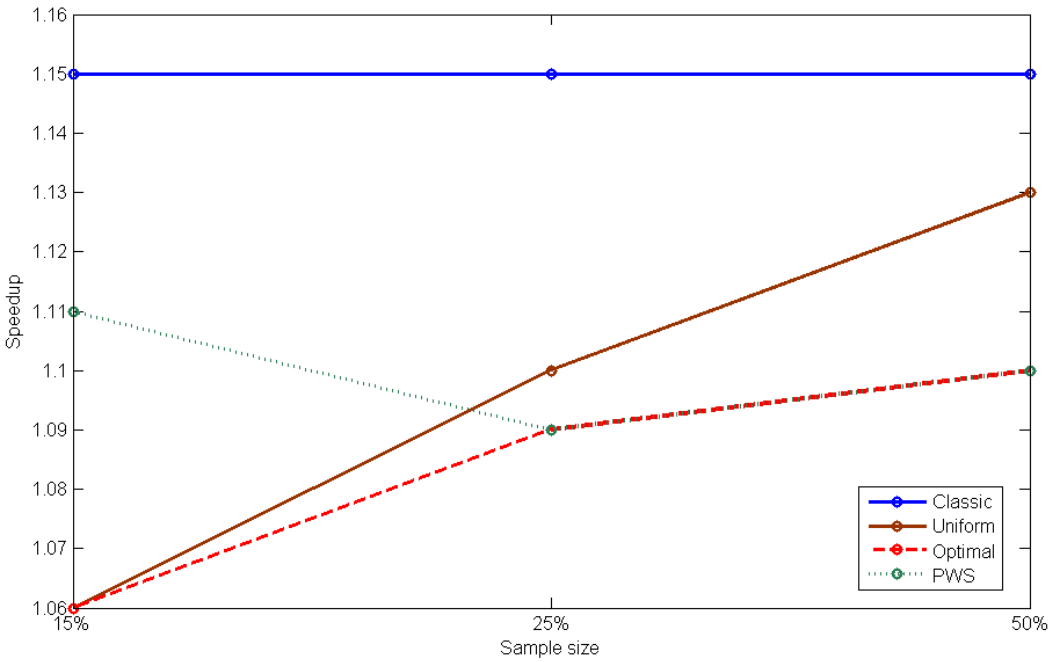


Figure 9: The speedup of three FMCA sampling techniques and classic. Sample size of $15\%, 25\%$ and 50%.

From Figure 9, we can observe the difference in speedup when running the problem in parallel 4 cores. In all cases, classical matrix-vector multiplication performed the best speedup. Among the three Monte Carlo techniques, we can see at sample size 15% of input size, PWS showed better result than uniform and optimal. This again illustrated the assumption in [5] that with affinity matrix input, PWS performed better for small sample

sizes. When the sample sizes increased to 25% and 50% of input size, PWS's speedup had decreased, while uniform and optimal showed better results.

From Table 4, we can spot the difference in execution time when running the problem in parallel 4 cores and on CPU 1 core. Among the three Monte Carlo sampling techniques, uniform had the smallest execution time, and then came optimal and PWS performed slowest. A time profile of the three techniques showed that most of the time was spent on generating the indices for the sampled matrix. In uniform, generating the indices required one call to `randi()`; in optimal, generating the indices required two steps: first compute the probabilities, and then one call to `randsample()`; and in PWS, generating the indices required a *for* loop, inside which there was one call to `randi()`. For example, to generate a sample of size c, uniform and optimal required only one call to the random function, while PWS required c calls to the random function. Between uniform and optimal, optimal sampling took an extra step to compute the probabilities, thus uniform performed faster.

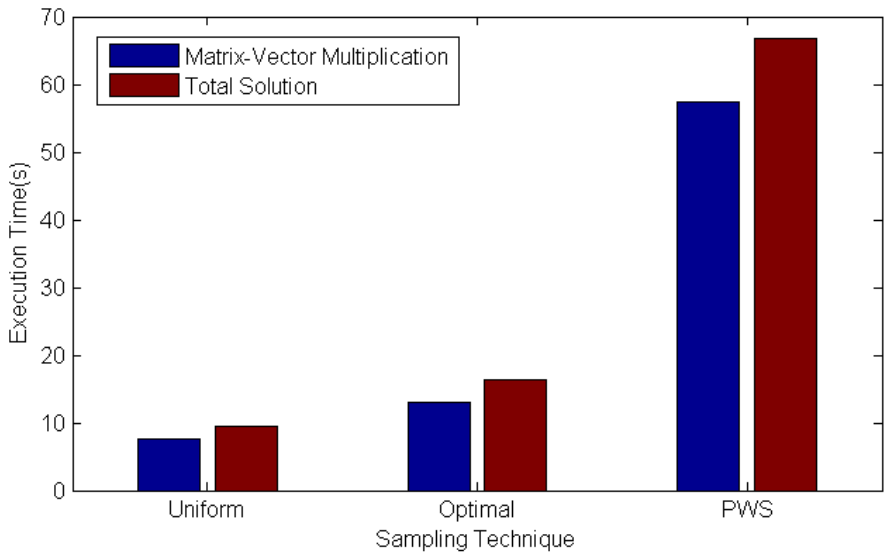


Figure 10: The time profile of the three techniques in matrix-vector multiplication. Sample size 50% of input size.

Figure 10 showed the time profile for the three sampling techniques, comparing the execution between the matrix-vector multiplication and the total time of the method to get the final solution. As we can see, the time to compute matrix-vector multiplication were about 80% to 85% of the total time.

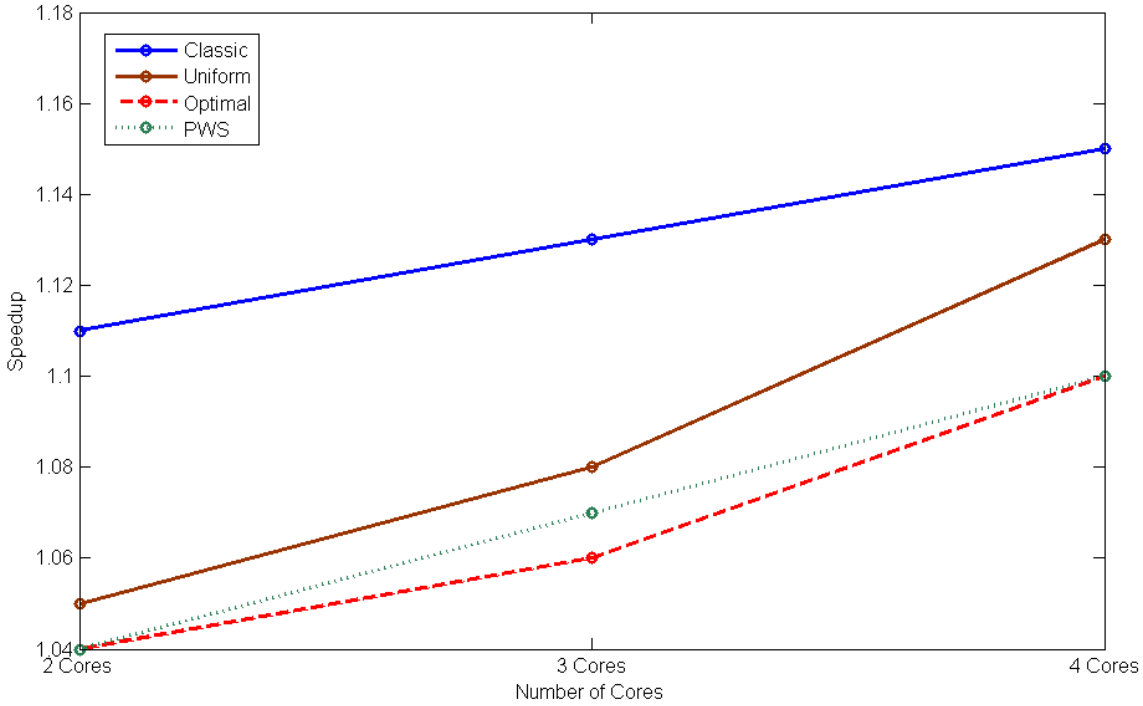


Figure 11: The speedup of three FMCA sampling techniques and classic, running on different number of cores. Sample size of 50%.

Figure 11 illustrated the speedup up of three FMCA sampling techniques and classic matrix-vector multiplication function when running on two,three and four cores. The speedup was ideally linear when doubling the number of processors would double the speed, however in this experiment, we can see that the speed increased slightly when changing from two cores to three or four cores.

Since we were working on the image restoration problem which the blurring operator stored as a sparse matrix, this might speedup the implementation matrix-vector multiplication in MATLAB. As a matter of fact, since an update in version R2007a, MATLAB implicitly supported multithreaded computation for sparse matrix [4]. We took a step forward to take a test on Paris problem, running on full matrix to see the speedup when running in parallel 4 cores. We had to decrease the image dimension from 256x256 to 96x96, because this is the maximum memory capacity of the computer when dealing with full matrix. We used a sample size 25% of problem size.

Image restoration problem Paris with image dimension 96x96 problem size 9216:

The original 'paris.jpg' image is blurred with **blur** function from Regularization Tools, Band 3, Sigma 1.5.

Sample size: 25% = 2304 , matrix A is full

Sampling	Uniform		Optimal		PWS		Classical	
	4 cores	1 core	4 cores	1 core	4 cores	1 core	4 cores	1 core
Relative Error	2.81e-001		2.53e-001		2.21e-001		1.14e-001	
Number of mvp	190.3		305.6		198.1		201.0	
Execution time	75.6	89.3	83.5	99.7	89.4	107.7	74.0	90.7
Speedup	1.18		1.19		1.21		1.26	
Iteration	3.5		4.1		3.6		3.0	
Nr of run	93		91		100		10	

Sample size: 25% = 2304 , matrix A is sparse

Sampling	Uniform		Optimal		PWS		Classical	
	4 cores	1 core	4 cores	1 core	4 cores	1 core	4 cores	1 core
Relative Error	2.81e-001		2.53e-001		2.21e-001		1.14e-001	
Number of mvp	190.3		305.6		198.1		201.0	
Execution time	8.4e-1	8.5e-1	9.1e-1	9.3e-1	4.1	4.3	6.8e-1	7.2e-1
Speedup	1.01		1.02		1.05		1.06	
Iteration	3.5		4.1		3.6		3.0	
Nr of run	93		91		100		10	

Table 5. Paris problem with blurring operator matrix is sparse and full, solved with LSTRS with three approaches to compute matrix-vector multiplication: Monte Carlo sampling techniques (Uniform, Optimal and Piecewise Uniform Sampling - PWS) and built-in MATLAB matrix-vector multiplication operation (classical) - with sample size of 25%.

From Table 5, we noticed the difference between solving the problem with input matrix A in full and sparse format was the execution time and the speedup. As expected, the execution time when solving the problem with a full matrix was much larger than with a sparse matrix. From the speedup comparison, the speedups for the sparse matrix ranged from 1.01 to 1.06, while for the full matrix, we obtained speedups from 1.18 to 1.26. This meant that by default MATLAB already supports multithreaded computation for sparse matrix so that the gain in sparse matrix case is less than in full matrix case.

3.2.3 GPU Computation - problem phillip with LSTRS and FMCA, medium scale

In this experiment, we would like to investigate the performance of LSTRS method on regularization problem phillips dimension 1000 when the matrix-vector multiplication is implemented in different approaches: MATLAB built-in matrix-vector multiplication function and three Monte Carlo sampling techniques (uniform, optimal and piecewise uniform sampling - PWS), the performance were tested on CPU and GPU. FMCA was set to run for up to 100 times, and the average results are computed.

The experiments were performed on MATLAB R2011b on a Dell Desktop with a Duo-Core, 3.16GHz processor, 4 GB of memory, running on Debian GNU/Linux 7 for GPU Computation, with GPU NVIDIA GT200 - Tesla C1060. The floating-point arithmetic was IEEE standard double precision with machine precision $2^{-52} \approx 2.2204 \times 10^{-16}$.

In our experiments with FMCA, we would like to compute the Quasi-Optimal and Boundary solutions of the regularization problem phillips. Thus, the setting `lopts.interior` was set to *'n'*, indicated that no Interior solution should be computed. Since the FMCA with the randomize routine might lead to a *bad* approximation in matrix-vector product, which would cause the program to halt when solving the Interior solutions , therefore this setting is an important aspect to prevent the program running into these *bad* approximation.

Figure 12 illustrated the relative errors of LSTRS on problem phillips when running on CPU and GPU, with FMCA implementation. It was clearly shown that the relative error of LSTRS with classical matvec on CPU and GPU were equal. The same results occurred for FMCA, thus in the figure, we only included one data for each FMCA sampling techniques.
Among classical matvec on CPU, GPU and FMCA, the relative errors on CPU (and GPU) were the lowest. Again we could see the same behavior for FMCA: at small sample sizes of 10% to 20% of input size, PWS showed the best results as matrix A was affinity, and because matrix A was also ill-conditioned then optimal performed more accurate than uniform.
However, one of the most important aspect in our experiments when running the problem on CPU and GPU was to compare the performance between these two platforms in execution time. It was expected that running on GPU would be faster than on CPU.

Figure 13 represented the speedup of LSTRS running on GPU compared to CPU, with classical matvec and three FMCA techniques. As we can see, classic had the most speedup. Among the three FMCA techniques, optimal showed best performance in speedup, then

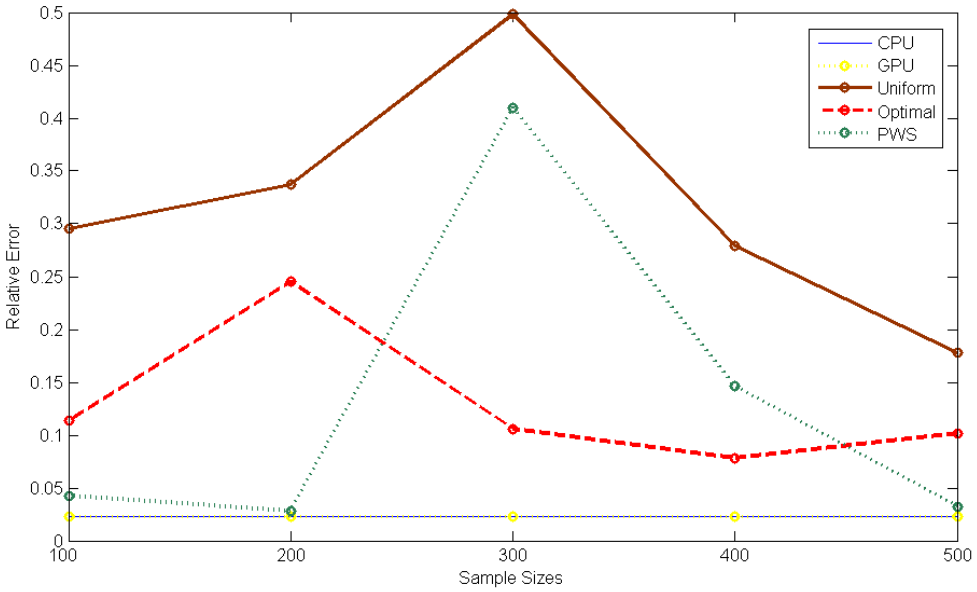


Figure 12: The relative errors of LSTRS on problem phillips dimension 1000, with three FMCA sampling techniques and classic, running on CPU and GPU. Sample size from 10% to 50% of input size.

came PWS and uniform was the lowest. To explain this behavior in speedup of FMCA, we would reminded the number of function calls and implementation in each sampling technique to generate a sampled matrix of size c: uniform took one call to `randi` function, PWS took c call to `rand` function, while optimal took one to `randsample` function but had an extra step to compute the probabilities. Therefore when running the operation on GPU, uniform had the lowest speedup because it had the least computation. On the other hand, optimal with the most computation showed the highest gain in speedup.
Moreover, even though the computation on GPU was faster than on CPU, we had to consider the time consume for data transfer between GPU and CPU, for example we had to transfer the data from CPU to GPU using `gpuArray`, did the computation on GPU then collected the data using `gather`, and it took times for this process to be completed, hence the speedup on GPU was reduced. Therefore, the more data in transfer would increase the execution time and in order to get the best speedup, we had to transfer as much information as possible in one operation. As it was clearly indicated in Figure 13, for three FMCA techniques, the speedup was maximum at sample size 10% of input size, and as the sample sizes increased, the speedup decreased, till sample size 50% of input size, we had no gain in execution time as the speedup equals 1.

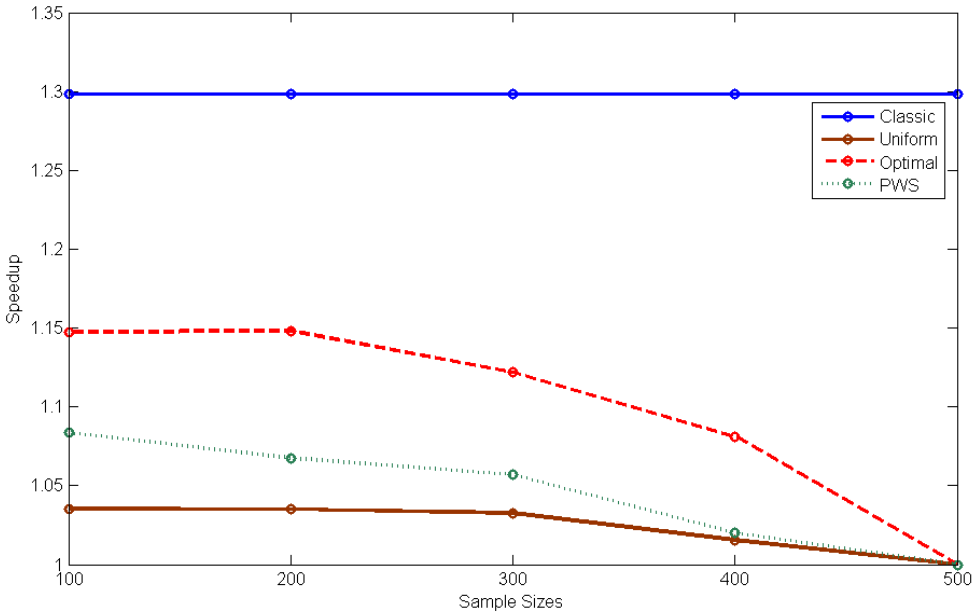
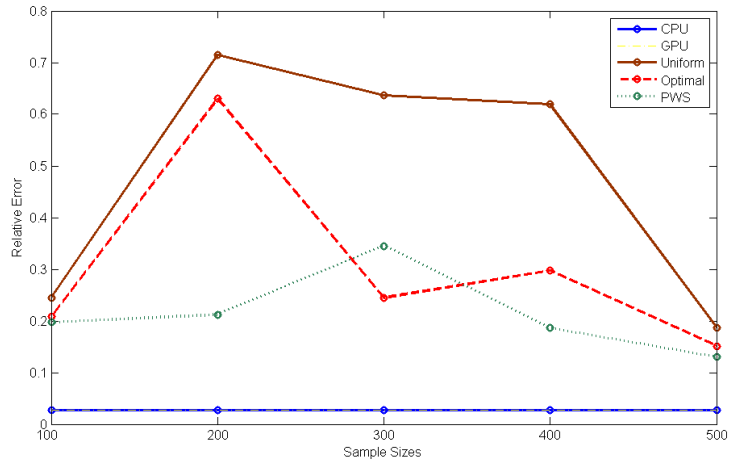


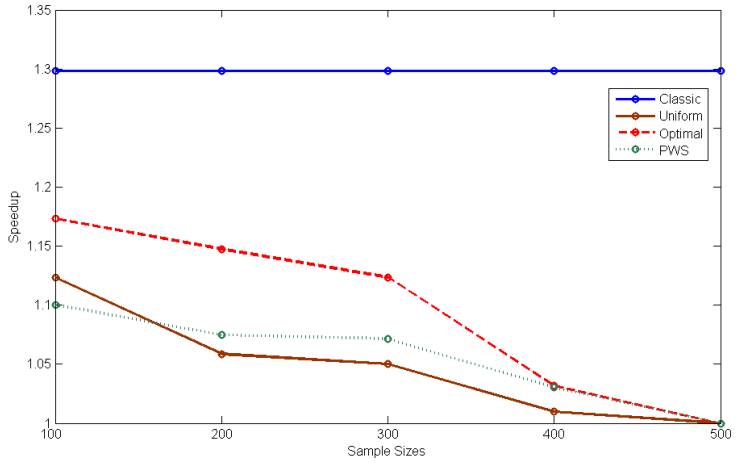
Figure 13: The speedup of LSTRS on problem phillips dimension 1000, with three FMCA sampling techniques and classic, running on CPU and GPU. Sample sizes from 10% to 50% of input size. The speedup was computed by dividing the execution time running on GPU to time on CPU.

In addition to this experiment, we ran the test not only on LSTRS but also on TRUSTμ with the same settings, with the purpose to verify the behavior that we had observed. The initial value for TRUSTμ was chosen as the LSTRS solution with zero or negative elements were replace by 10^{-5}.

According to Figure 14, we observed the similar behavior of TRUSTμ on problem phillips, same as LSTRS did in our previous experiment. In relative errors, CPU and GPU showed the best results, among the three FMCA, PWS performed better than the other two techniques with sample sizes from 10% to 20% of input size. In speedup, we could see that classical matvec had the most speedup. The three FMCA performed as optimal,PWS and uniform in speedup descending order. Since the sample sizes increased from 10% to 50% of input size, the speedup of FMCA decreased and dropped close to 1 at sample size 50% of input size.



Relative Error



Speedup

Figure 14: The relative errors and speedup of TRUSTμ on problem phillips dimension 1000, with three FMCA sampling techniques and classic, running on CPU and GPU. Sample size from 10% to 50% of input size.

4 Conclusion

In this paper, we studied the TRUSTμ method for solving large-scale non-negativity regularization. The method based on matrix-vector multiplication. We also represented the fast Monte Carlo algorithm with three sampling techniques and described the GPU computation and parallel computing. In this project, we developed a MATLAB prototype implemented the TRUSTμ method and performed the TRUSTμ experiments in [8], as well as evaluated the accuracy and performance of the method when using the three approaches to compute the matrix-vector products.

In conclusion, in the experiments to compute regularized non-negative solutions to the inverse problem and image restoration problem, compare to LSTRS method, TRUSTμ computed positive restorations with better accuracy but took higher computational cost. For the evaluation of the three approaches to compute the matrix-vector multiplication, we divided into separated comparison. Among the three fast Monte Carlo Algorithms (FMCA) and classical matvec, consistent with the assumption from [5] that when matrix A was affinity, for small sample size from 10% to 20% of the problem size, piecewise uniform sampling(PWS) showed the best performance. Since the sample size increased, the accuracy of uniform, optimal and PWS got closer to classical matvec's. In parallelization, we investigated the parallel performance on up to 4 processors and found that the speed-ups were modest. Our experiments also confirm the fact that by default MATLAB supports multithreaded computation for sparse matrix. Finally, in GPU computation experiments, we could see that among three FMCA techniques and classical matvec, running on GPU and on CPU, classical matvec showed the best speedup, then came optimal, PWS and uniform had the lowest speedup. The reason to the behavior in speedup of FMCA was the implementation in each sampling technique to generate a sampled matrix: optimal with the most computation showed the highest gain in speedup. Moreover, another aspect in GPU computation was the time consume for transferring data between CPU-GPU. For the FMCA, the speedup was maximum at sample size 10% of problem size, and as the sample size increased, the speedup decreased. Our observation shows that the speed-up on the GPU is rather disappointing: we only gain by percentages, not factors.

References

[1] P. Drineas, R. Kannan, and M. W. Mahoney. Fast Monte Carlo Algorithms for Matrices I: Approximating matrix multiplication. *SIAM J. Comput.*, 36(1):132–157, 2006.

[2] P. C. Hansen. Regularization Tools: A MATLAB package for analysis and solution of discrete ill-posed problems. *Numerical Algorithms*, 6:1–35, 1994.

[3] The Mathworks Inc. Run built-in functions on a GPU, June 2013.

[4] The Mathworks Inc. Which MATLAB functions benefit from multithreaded computation?, June 2013.

[5] H. Madrid, V. Guerra, and M. Rojas. Sampling techniques for Monte Carlo matrix multiplication with applications to image processing. *Lect. Notes Comput. Sc.*, 7329:45–54, 2012.

[6] D. L. Phillips. A technique for the numerical solution of certain integral equations of the first kind. *J. ACM*, 9:84–97, 1962.

[7] M. Rojas, S. A. Santos, and D. C. Sorensen. Algorithm 873: LSTRS: MATLAB software for large-scale trust-region subproblems and regularization. *ACM Trans. Math. Software*, 34(2):11, 2008.

[8] M. Rojas and T. Steihaug. An interior-point trust-region-based method for large-scale non-negative regularization. *Inverse Problems*, 18(5):1291–1307, 2002.

APPENDIX

Settings for the experiment:

TRUSTμ performance and accuracy. Problem phillips, small and medium scale:
For TRUSTμ : $\mu = 0.1; \sigma = 0.01; \epsilon_x = \epsilon_f = 10^{-5}, \epsilon_y = 10^{-12}$; initial value of TRUST$\mu$ equal solution of TRS solver, with negative or zero values replaced by 10^{-5}.
For both LSTRS and TRUSTμ : $epsilon.Delta = 10^{-2}; epsilon.HC = 10^{-4}$;
$epsilon.alpha = 10^{-8}; lopts.heuristics = 1; lopts.alpha =' deltaU'$;
$eigensolverpar.tol = 0.5, eigensolverpar.p = 7$.

TRUSTμ performance and accuracy. Problem image restoration, large scale:
For TRUSTμ: $\mu = 0.1; \sigma = 0.01; \epsilon_x = \epsilon_f = 10^{-5}, \epsilon_y = 10^{-12}$; initial value of TRUST$\mu$ equal solution of TRS solver, with negative or zero values replaced by 10^{-5}.
For both $LSTRS$ and TRUSTμ : $eigensolver =' tcheigs\_lstrs\_gateway'$, with normalized vector $eigensolverpar.v0 = ones(65537, 1)/sqrt(65537); epsilon.Delta = 10^{-2}; epsilon.HC = 10^{-4}; epsilon.alpha = 10^{-8}; eigensolverpar.tol = 0.5, eigensolverpar.p = 7$.

Classic Matrix-Vector Multiplication and Monte Carlo - problem phillips with LSTRS and FMCA, medium scale with various sample sizes.

$epsilon.Delta = 10^{-2}; epsilon.HC = 10^{-4}; epsilon.alpha = 10^{-8}$;
$lopts.heuristics = 1; lopts.alpha =' deltaU'; lopts.interior =' n'$
$eigensolverpar.tol = 0.5, eigensolverpar.p = 7$.

Parallel Computing - image restoration problem with LSTRS and FMCA, large scale with various sample sizes.
$eigensolver =' tcheigs_lstrs_gateway', epsilon.Delta = 1e-2; epsilon.HC = 1e-4$; with normalized vector $eigensolverpar.v0 = ones(65537, 1)/sqrt(65537)$;
$eigensolverpar.tol = 0.7; eigensolverpar.p = 7$;
The original 'paris.jpg' image is blurred with 'blur' function from Regularization Tools, Band 3, Sigma 1.5.

Image restoration problem Paris with image dimension 96x96 problem size 9216:
$eigensolver =' tcheigs_lstrs_gateway', epsilon.Delta = 1e-2; epsilon.HC = 1e-4$; with normalized vector $eigensolverpar.v0 = ones(9217, 1)/sqrt(9217)$;
$eigensolverpar.tol = 0.7; eigensolverpar.p = 7; lopts.interior =' n'$.
The original 'paris.jpg' image is blurred with 'blur' function from Regularization Tools, Band 3, Sigma 1.5.

GPU Computation - problem phillip with LSTRS and FMCA, medium scale.
$epsilon.Delta = 10^{-2}; epsilon.HC = 10^{-4}; epsilon.alpha = 10^{-8};$
$lopts.heuristics = 1; lopts.alpha =' deltaU'; lopts.interior =' n'$
$eigensolverpar.tol = 0.3, eigensolverpar.p = 5.$

GPU Computation - problem phillip with TRUSTμ and FMCA, medium scale.
$epsilon.Delta = 10^{-2}; epsilon.HC = 10^{-4}; epsilon.alpha = 10^{-8};$
$lopts.heuristics = 1; lopts.alpha =' deltaU'; lopts.interior =' n'$
$eigensolverpar.tol = 0.3, eigensolverpar.p = 5.$
$\mu = 0.1; \sigma = 0.01; \epsilon_x = \epsilon_f = 10^{-5}, \epsilon_y = 10^{-12};$
initial value of TRUSTμ equals solution of TRS solver, with negative or zero values replaced by 10^{-5}.

TRUSTμ: Matlab software for large-scale non-negative regularization.

SOFTWARE MANUAL

Author:
NGUYEN HOANG KIEN and MARIELBA ROJAS
Delft University of Technology.

In this document, we would like to present the software manual of a MATLAB implementation of TRUSTμ method with three approaches to compute the matrix-vector multiplication: Fast Monte Carlo algorithm, GPU, and Parallel computing. TRUSTμ method was described in Rojas and Steihaug,[3]. The method is based on a non-negatively constrained quadratic problem. The method is an interior-point iteration that solves a sequence of large-scale and possibly ill conditioned trust-region subproblem. TRUSTμ method relies on matrix-vector products only. In the MATLAB implementation, the Hessian matrix of the quadratic function in the trust-region subproblem can be provided either explicitly, or implicitly as a matrix-vector multiplication routine. A description of the MATLAB software, version 1.1, is presented. A guide for using the software and examples are provided.

Authors's Addresses:
K.H. Nguyen, Delft University of Technology (kiendhth@gmail.com)
M. Rojas, Delft Institute of Applied Mathematics, Delft University of Technology, 2600GA Delft ,The Netherlands (marielba.rojas@tudelft.nl)

1. INTRODUCTION:

This document contains the software manual for version 1.1 of a MATLAB implementation of the TRUSTμ method [3] for large-scale non-negative quadratic problem:

$$\begin{gathered} min \tfrac{1}{2} x^T H x + g^T x - \mu \sum_{i=1}^{n} log \xi_i \\ \text{s.t.} \|x\| \leq \Delta \\ x \geq 0 \end{gathered}$$

in which, H is an n x n, real, symmetric matrix, g is an n-dimensional real vector, Δ is a positive scalar, $\|.\|$ denotes the Euclidean norm, $x = (\xi_1, \xi_2, ...\xi_n)^T$ and $\mu > 0$ is the barrier or penalty parameter. The idea of TRUSTμ method is to solve a sequence of trust-region subproblem using the linesearch with barrier μ converges to 0.

The MATLAB implementation of TRUSTμ described in this manual allows the user to provide the matrix H explicitly as an array in MATLAB form, or as a function of matrix-vector multiplication, which preserves the matrix-free nature of the method. Moreover, the software offers four options to compute the matrix-vector product: built-in MATLAB function *mtimes*, Fast Monte Carlo algorithm, GPU computation, or Parallel computing.

This document is organized as follows: in Section 2, we describe the main features of the software: interface, data structure, options for using matrix-vector product. In Section 3, we provide instructions for installing and running the software, also we provide some example using the software.

2. MATLAB SOFTWARE IMPLEMENTATION:

In this section, we will describe the MATLAB implementation of TRUSTμ method [3].

2.1 The interface of TRUSTμ:

The routine to call for the method is named **TRUSTmu**, the general form for the calling is as follows:

$[x, info] = TRUSTmu(H, g, Delta, sigma, Epsilons, trs\_solver, Opts, Hpar)$

2.1.1. Inputs of the software:

The input is described in the form as follows: **Input name** (*type of input, default value if available*): input description.

Compulsory (3) :

- **H** ($string, function\_handle, or double$): real, n x n, symmetric matrix, or string or function-handle specifying a matrix-vector multiplication routine.
- **g** (*double*) : real, n x 1 array.
- **Delta** (*double*) : positive scalar (trust-region radius).

Optional (5) :

- **sigma** ($double, 10^{-2}$) : parameter to update μ.
- **Epsilons** (*struct*): contains the tolerances for the stopping criteria.
  - * **Epsx** ($double, 10^{-5}$): tolerance for x.
  - * **Epsy** ($double, 10^{-12}$): tolerance for y.
  - * **Epsf** ($double, 10^{-5}$): tolerance for f.
- **trs_solver** ($string,' lstrs'$) : solver for Trust-Region Subproblem (TRS).
- **Opts** (*struct*): options for TRS solver and initial values for TRUSTμ method.
  - * **x0** ($double, [\,]$) : initial value for x.
  - * μ**0** ($double, [\,]$) : initial value for barrier parameter μ.
  - * **xinitopt** ($string,' pos'$): decide the value to pass to TRUSTμ solver: either it is the initial value with small elements ($\leq 10^{-5}$) replaced by a larger number (10^{-2}), or the absolute of initial value with small elements ($\leq 10^{-5}$) replaced by a larger number (10^{-2}) . Options are *'pos'* and *'abspos'*.
  - * **maxit** ($double, 50$): maximum number of iteration for TRUSTμ solver.
  - * **mvopt** ($string,' mv'$): decide the approach to compute matrix-vector product. The options are *'mv, mcmv_uni, , mcmv_opt, mcmv_pws, mv_par,*

mv_gpu'. ; in which *'mv'* is MATLAB built-in *'mtimes'* function, *mcmv_uni, , mcmv_opt, mcmv_pws* are Monte Carlo algorithm with uniform sampling, optimal sampling and piecewise uniform sampling respectively, *mv_par* is parallel computing and *mv_gpu* is GPU computation.

* **trs_opt** (*struct*): Option parameters for TRS solver, provided by user.

- **Hpar**: structure containing parameters for H.

  * **A** ($double, string, or function\_handle$): matrix storage for the matrix-vector product routine.

  * **mu** (*double*): barrier parameter storage for the matrix-vector product routine, updated inside TRUSTμ method.

  * **x2** ($double, [\,]$): vector storage for the matrix-vector product routine.

  * **mv** ($string or function\_handle, [\,]$): store the function name of the matrix-vector product routine.

  * **c** ($double, [\,]$): the sample size for Monte Carlo algorithm, default value is [ ] which is set to 20% of the problem size.

  * **colnorm** ($double, [\,]$): the column norm to calculate the probabilities in optimal sampling.

2.1.2. Outputs:

- **x** : solution of the trust-region problem.

- **info**:

  * **info.mvp** : number of matrix-vector multiplication used in total.

  * **info.iter** : number of iteration of TRUSTμ solver.

  * **info.x0** : solution of trust-region subproblem without negativity constraint.

2.2. Method for calling the matrix-vector multiplication :

- Fast Monte Carlo Multiplication: in order to use the FMCA, change the value of mvopt in Opts.

  Option to choose:

  $'mcmv\_uni'$: FMCA with uniform sampling

  $'mcmv\_opt'$: FMCA with optimal sampling

  $'mcmv\_pws'$: FMCA with piecewise uniform sampling

- GPU computation for matrix-vector multiplication: change the value of mvopt in Opts to $'mv\_gpu'$

- Parallel computing for matrix-vector multiplication: change the value of mvopt in Opts to $'mv\_par'$

2.3. The trust-region subproblem (TRS) solver interface:

Choosing the TRS solver comes from the option **trs_solver**. If the TRS solver is user-defined, it needs to takes inputs and outputs in a corresponding form as follows:

$[x, infor] = trs\_solver(H, g, Delta, trs\_opt);$

The function should take three compulsory inputs **H, g, Delta** described in 2.1.1. Also, further options for the TRS solver (if needed) should be put in the structure option trs_opt.

The function returns:

- **x** : solution for TRS solver.

- **infor** (*struct*): include the information for TRS solver.

  mvp: number of matrix-vector multiplication used in TRS solver.

  iter: number of iteration processed in TRS solver.

3. SOFTWARE INSTALLATION AND EXAMPLES:

In this section, we will present the TRUSTμ software package files, installation manual and some examples on executing the software. For the TRS solver, we offer an existing solver called LSTRS, [2]. The download link for LSTRS software can be found at:

$$http://ta.twi.tudelft.nl/wagm/users/rojas/lstrs.html$$

The MATLAB M-files containing the components of the TRUSTμ software are presented as follows:

- **TRUSTmu.m**: interface routine for TRUSTμ method.
- **TRUSTmu_method.m**: main iteration for TRUSTμ method.
- **lstrs_gateway.m**: gateway routine for LSTRS solver.
- **user_trs_gateway.m**: gateway routine for user-defined TRS solver.
- **mv.m**: routine for matrix-vector multiplication.
- **mv_lstrs.m**: routine for matrix-vector multiplication in LSTRS solver.
- **mv_trustmu.m**: routine for matrix-vector multiplication in TRUSTμ method.
- **mcmv_uni_matrix.m**: routine for FMCA matrix-vector multiplication - uniform sampling with matrix input.
- **mcmv_uni_routine.m**: routine for FMCA matrix-vector multiplication - uniform sampling with routine input.
- **mcmv_opt.m**: routine for FMCA matrix-vector multiplication - optimal sampling.
- **mcmv_pws_matrix.m**: routine for FMCA matrix-vector multiplication - piecewise uniform sampling with matrix input.
- **mcmv_pws_routine.m**: routine for FMCA matrix-vector multiplication - piecewise uniform sampling with routine input.
- **mv_trustmu_par.m**: routine for matrix-vector multiplication with parallel computing.
- **mv_trustmu_gpu.m**: routine for matrix-vector multiplication with GPU computation.

- **pre_prob.m**: compute the column norm of the input matrix.
- **prob.m**: compute the probabilities in FMCA optimal sampling.
- **set_mvpar.m**: routine to set parameters for variable **Hpar**.

The file **Phillips_test.m** is included in the software, the file contains the example for using TRUSTμ method to solve Phillips problem, C. Hansen 2007.

SOFTWARE INSTALLATION:

The TRUSTμ MATLAB software is distributed as an archive in zip or rar format in the TRUSTmu.zip and TRUSTmu.rar, respectively. The Unix/Linux command *tar xvf lstrs.rar* will create a directory LSTRS in the current directory where all the M-files listed above will be stored. For the zip format we recommend that the user creates a directory TRUSTμ-directory and store the TRUSTμ files in that directory.

In either case, the TRUSTμ directory should be included in MATLAB's search path. This can be accomplished with one of the following commands: path(path,'TRUSTμ-directory') or addpath 'TRUSTμ-directory'. Moreover, if you download and use LSTRS solver, also include the search path to LSTRS-directory using the same commands.

EXAMPLES:
Phillips Problem:
Phillips test problem from Regularization Tools is a discretization of the well-known Fredholm integral equation, deviced by D. L. Phillips, [1].

```
1
2
3     % File TRUST_Phillips.m
4     % Solving Problem Phillips using TRUSTmu method
5     % Problem size 300
6
7     % Generate problem
8     % matrix A: blurring operator; vector b: degraded image.
9     % vector xexact : true solution.
10
11    [A,b,xexact] =  phillips(300);
12
13
```

Figure 1. Generate Phillips problem of size 300

Settings for the experiment:

For $TRUST\mu$: $\mu = 0.1; \sigma = 0.01; \epsilon_x = \epsilon_f = 10^{-5}, \epsilon_y = 10^{-12}$; initial value of $TRUST\mu$ equal solution of TRS solver, with small or zero values replaced by 10^{-5}.
For both $LSTRS$ and $TRUST\mu$:
$epsilon.alpha = 10^{-8}; epsilon.Delta = 10^{-2};$
$eigensolverpar.p = 7, eigensolverpar.tol = 0.5;$
$epsilon.HC = 10^{-4}; lopts.heuristics = 1; lopts.alpha =' deltaU'.$

```
% File TRUST_Phillips.m
% Solving Problem Phillips using TRUSTmu method
% Problem size 300

% Set mu
mu = 0.1;
% Stopping Criteria
Epsilons.epsx = 1e-5;
Epsilons.epsy = 1e-12;
Epsilons.epsf = 1e-5;
% Options for initial settings
Opts.x0 = [];
Opts.mu0 = mu;
Opts.maxit = 50;
Opts.xinitopt = 'pos';
% Options for TRS solver
Opts.eigensolverpar.tol = 0.7;
Opts.eigensolverpar.p = 7;
Opts.epsilon.Delta = 1e-2;
Opts.epsilon.HC = 1e-4;
Opts.epsilon.alpha = 1e-8;
Opts.lopts.heuristics = 1;
Opts.lopts.name = 'Phillips problem';
Opts.lopts.alpha = 'deltaU';
% Set option for matrix vector multiplication
% default value: 'mv' - MATLAB built in mtimes function
Opts.mvopt = 'mv';
% Opts.mvopt = 'mcmv_uni'; % Uniform Sampling of FMCA
% Opts.mvopt = 'mcmv_opt'; % Optimal Sampling of FMCA
% Opts.mvopt = 'mcmv_pws'; % Piecewise Uniform Sampling of FMCA
% Opts.mvopt = 'mv_par'; % Parallel Computing
% Opts.mvopt = 'mv_gpu'; % GPU Computation
% Settings for Hpar structure
Hpar = set_mvpar(A,mu,zeros(size(g)),@mv_lstrs,200,[],pre_prob(A));
```

Figure 2. Setting for $TRUST\mu$ function.

```
1
2
3       % File TRUST_Phillips.m
4       % Solving Problem Phillips using TRUSTmu method
5       % Problem size 300
6
7       % Call TRUSTmu method with input as a routine
8       [x ,infor] = TRUSTmu(@mv,g,Delta,sigma,Epsilons,@lstrs,Opts,Hpar);
9       % Call TRUSTmu method with input as a matrix
10      [x ,infor] = TRUSTmu(A,g,Delta,sigma,Epsilons,@lstrs,Opts,Hpar);
```

Figure 3. Call to $TRUST\mu$ function with 2 options to choose input as a matrix or a matrix-vector multiplication routine.

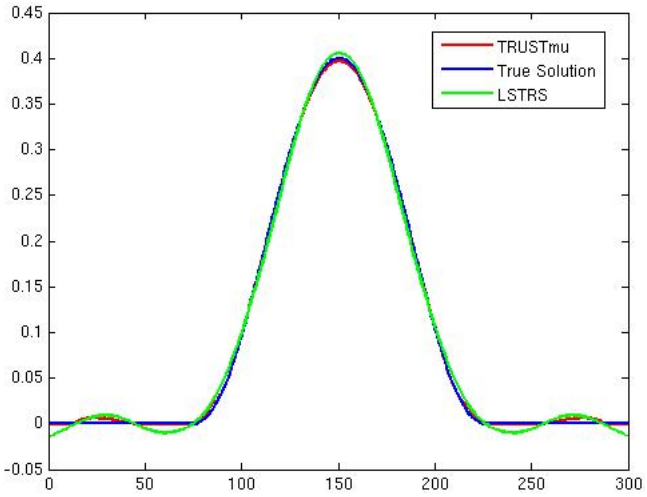


Figure 4. $TRUST\mu$ solution plot for regularization problem Phillips of size 300. The LSTRS solution is the result from TRS solver for initial value of $TRUST\mu$. The blue curve is the exact solution which has been added to the plot for comparison (this curve is not generated by $TRUST\mu$).

References

[1] P. C. Hansen. Regularization tools: A matlab package for analysis and solution of discrete ill-posed problems. *Numerical Algorithms*, 6:1–35, 1994.

[2] Marielba Rojas, Sandra A. Santos, and Danny C. Sorensen. Algorithm 873: Lstrs: Matlab software for large-scale trust-region subproblems and regularization. *ACM Trans. Math. Software*, 34(2):11, 2008.

[3] Marielba Rojas and Trond Steihaug. An interior-point trust-region-based method for large-scale non-negative regularization. *Inverse Problems*, 18(5):1291–1307, 2002.