Novel Monte Carlo Methods for Large-Scale Linear Algebra Operations

Hao Ji
Old Dominion University

NOVEL MONTE CARLO METHODS FOR LARGE-SCALE LINEAR ALGEBRA OPERATIONS

by

Hao Ji
B.S. June 2007, Hefei University of Technology, China
M.S. December 2010, Hefei University of Technology, China

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
August 2016

Approved by:

Yaohang Li (Director)

Stephan Olariu (Member)

Nikos Chrisochoides (Member)

Masha Sosonkina (Member)

Seth H. Weinberg (Member)

ABSTRACT

NOVEL MONTE CARLO METHODS FOR LARGE-SCALE LINEAR ALGEBRA OPERATIONS

Hao Ji
Old Dominion University, 2016
Director: Dr. Yaohang Li

Linear algebra operations play an important role in scientific computing and data analysis. With increasing data volume and complexity in the "Big Data" era, linear algebra operations are important tools to process massive datasets. On one hand, the advent of modern high-performance computing architectures with increasing computing power has greatly enhanced our capability to deal with a large volume of data. One the other hand, many classical, deterministic numerical linear algebra algorithms have difficulty to scale to handle large data sets.

Monte Carlo methods, which are based on statistical sampling, exhibit many attractive properties in dealing with large volume of datasets, including fast approximated results, memory efficiency, reduced data accesses, natural parallelism, and inherent fault tolerance. In this dissertation, we present new Monte Carlo methods to accommodate a set of fundamental and ubiquitous large-scale linear algebra operations, including solving large-scale linear systems, constructing low-rank matrix approximation, and approximating the extreme eigenvalues/ eigenvectors, across modern distributed and parallel computing architectures. First of all, we revisit the classical Ulam-von Neumann Monte Carlo algorithm and derive the necessary and sufficient condition for its convergence. To support a broad family of linear systems, we develop Krylov subspace Monte Carlo solvers that go beyond the use of Neumann series. New algorithms used in the Krylov subspace Monte Carlo solvers include (1) a Breakdown-Free Block Conjugate Gradient algorithm to address the potential rank deficiency problem occurred in block Krylov subspace methods; (2) a Block Conjugate Gradient for Least Squares algorithm to stably approximate the least squares solutions of general linear systems; (3) a BCGLS algorithm with deflation to gain convergence acceleration; and (4) a Monte Carlo Generalized Minimal Residual algorithm based on sampling matrix-vector products to provide fast approximation of solutions. Secondly, we design a rank-revealing randomized Singular Value Decomposition (R^3SVD) algorithm for adaptively constructing low-rank matrix approximations to satisfy application-specific accuracy. Thirdly, we study the block power method on Markov Chain Monte Carlo transition matrices and find that the convergence is actually depending on the number of independent vectors in the block. Correspondingly, we develop a sliding window power method to find stationary distribution, which has demonstrated success in modeling stochastic luminal Calcium release site. Fourthly, we take advantage of hybrid CPU-GPU computing platforms to accelerate the performance of the Breakdown-Free Block Conjugate Gradient algorithm and the randomized Singular Value Decomposition algorithm. Finally, we design a Gaussian variant of Freivalds' algorithm to efficiently verify the correctness of matrix-matrix multiplication while avoiding undetectable fault patterns encountered in deterministic algorithms.

This thesis is dedicated to my family,

for their endless love, support, and encouragement.

ACKNOWLEDGMENTS

First of all, I would like to express my sincerest gratitude to my Ph.D. supervisor Dr. Yaohang Li, for providing me with the opportunity to study in his research group at Old Dominion University. He offered me constant support and encouragement throughout this dissertation work. Without his guidance, this work would not have been possible.

I would also like to thank my committee members: Dr. Stephan Olariu, Dr. Nikos Chrisochoides, Dr. Masha Sosonkina, and Dr. Seth Weinberg, for their helpful comments and invaluable suggestions to improve the contents of this work. I gratefully acknowledge the generous support of Old Dominion University Modeling and Simulation fellowship on this research.

I am grateful for the companionship of my friends. I want to express my special thanks to Dr. Xiaoping Liu from Hefei University of Technology and Dr. Kurt Maly from Old Dominion University, for introducing me to the Ph.D. program in Computer Science at Old Dominion University when I was in China.

Finally, I am especially grateful to my family. Regardless of my successes or failures, they always stand by me, support me, and love me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Statement of the Problem

Numerical linear algebra operations, such as solving systems of linear equations, linear regression, constructing low-rank matrix approximation, approximating extreme eigenvalues/eigenvectors of a matrix, and so on, are behind many real-life applications, ranging from data mining to large-scale simulations and machine learning. The efficiency of these linear algebra applications is crucial for the performance of many big data applications.

With the increasing size and complexity of datasets in the "Big Data" era, many problems involve operations on matrices with millions, billions, or even trillions of elements. The large volume of matrices brings new computational challenges to classical numerical linear algebra algorithms. For example,

(1) Costly matrix pass: When a matrix is too large, it may be unable to fit in the core memory. Very often, in many practical applications, the large matrices are not explicitly stored and the matrix elements will be regenerated when needed. Consequently, the cost of transferring a matrix from slow memory to core memory or regenerating matrix elements easily dominates that of arithmetic calculations. As a result, a pass over the matrix elements becomes a new computational bottleneck in many operations on large matrices. For extremely large matrices, a complete matrix pass is even prohibited.

(2) Scalability to modern parallel and distributed computing architectures: Many traditional, deterministic numerical methods are typically designed to obtain highly accurate solutions with high consumptions of computational power or memory storage, which make them less effective or even infeasible to scale to a large dataset. Modern linear algebra algorithms are expected to fully take advantage of modern parallel and distributed computing paradigms to achieve good performance.

(3) Potential memory errors: When a matrix is large enough, the matrix computations are vulnerable to faults in computer systems. Errors that corrupt the data being processed are no longer negligible, and

fault-tolerant and resilient numerical algorithms are demanded for large-scale linear algebra operations.

Consequently, many traditional algorithms for linear algebra operations, especially those designed to minimize floating-point operations, have difficulty in scaling up to handle increasingly large matrices. Addressing these computational challenges to improve the performance of large-scale linear algebra operations is the key in support of scientific computing and data analysis applications with large volume of data, which will eventually lead to broad scientific and economic impacts. Hence, the objective of this dissertation is to design new computational methods to accommodate large-scale numerical linear algebra operations across modern distributed and parallel computing architectures.

1.2 Our Approaches

The Monte Carlo methods benefit from random sampling and exhibit many attractive advantages when handling extremely large matrices. For instance,

(1) Monte Carlo methods are based on statistical sampling, where most operations are carried out on a small portion of carefully sampled matrix elements, and, thus, the number of passes on all matrix elements can be limited, often by orders of magnitude.

(2) Monte Carlo methods are naturally parallel. Therefore, they are well-suitable to large-scale computing platforms, which are equipped with a large number of multi-core CPUs, many-core coprocessors, and multi-general purpose graphics process units (GPGPU).

(3) Monte Carlo methods are often able to obtain low-accuracy solution approximation quickly, which is particularly suitable for many applications where high-accuracy solutions are not necessary.

Motivated by the attractive features of Monte Carlo methods, in this dissertation, we develop efficient Monte Carlo methods to carry out a set of fundamental and ubiquitous linear algebra operations. The major contributions of this work include,

(1) A necessary and sufficient condition for the convergence of the classical Monte Carlo linear solver using the Ulam-von Neumann algorithm (Chapter III).

(2) Krylov subspace Monte Carlo solvers to handle general large-scale linear systems with pass reduction and convergence acceleration, such as a Breakdown-Free Block Conjugate Gradient algorithm (BFBCG), a Block Conjugate Gradient for Least Squares algorithm (BCGLS), a BCGLS algorithm with Deflation (BCGLSD), and a Monte Carlo Generalized Minimal Residual algorithm (MCGMRES) (Chapter III).

(3) A Rank-Revealing Randomized Singular Value Decomposition (R^3SVD) algorithm to adaptively construct low-rank matrix approximations (Chapter IV).

(4) A Sliding Window Power (SWP) method to rapidly approximate the extreme eigenvalues/eigenvectors of large matrices (Chapter V).

(5) Using GPUs to accelerate matrix computations in BFBCG and Randomized Singular Value Decomposition (RSVD) (Chapter VI).

(6) A Gaussian variant of Freivalds' algorithm (GVFA) to efficiently and reliably validate the correctness of matrix-matrix multiplication (Chapter VII).

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter II presents a review of the relevant literature to Monte Carlo methods, numerical linear algebra operations, and acceleration and validation techniques of matrix operations on modern parallel/distributed platforms. We present our new Monte Carlo methods with rigorous mathematical analysis for solving large-scale linear systems, constructing low-rank matrix approximation, and approximating extreme eigenvalues and eigenvectors in Chapters III, IV, and V, respectively. Chapter VI investigates the accelerated implementations of the Monte Carlo algorithms on hybrid CPU-GPU platforms. Chapter VII proposes a novel approach based on random sampling to verify the correctness of matrix products. Finally, Chapter VIII summarizes the dissertation and discusses our future (post-dissertation) research directions.

CHAPTER II

LITERATURE REVIEW

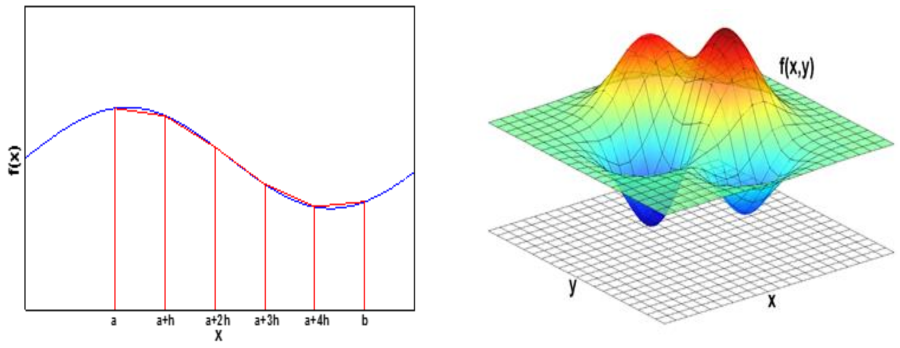
2.1 Monte Carlo Methods

Numerical methods known as Monte Carlo methods can be loosely defined in general terms to be any methods that rely on random sampling to estimate the solutions [1]. Monte Carlo methods are often applied to problems which are either too complicated to be described by a mathematical model or whose parameter space is too large to be explored systematically.

2.1.1 The Basic of Monte Carlo

Monte Carlo methods provide approximate solutions to a variety of mathematical problems by random sampling. To illustrate the principles of Monte Carlo methods, we use the numerical integration as an example.

Suppose we want to calculate a one-dimensional definite numerical integral, $I = \int_a^b f(x)dx$. A common numerical integral method is to divide the one-dimensional interval into N subintervals and then to sum the area corresponding to each subinterval using either rectangular, trapezoidal, or Simpson's rules (Fig. 1(a)) [2]. Similarly, for two-dimensional intervals, the number of 2D subintervals becomes N^2 (Fig. 1(b)). In general, for d-dimensional integration problems, the d-dimensional space needs to be divided into N^d subintervals. For a not very high dimensional problem with $d = 20$ and $N = 100$, the total number of subintervals that need to be evaluated goes up to 10^{40}, which is unapproachable by many numerical integration algorithms.



(a) 1D integral (b) 2D integral

Fig. 1. Numerical integration using deterministic methods

In contrast, Monte Carlo methods estimate the integral by statistical sampling techniques [3]. Let us consider a one-dimensional integral $I_{0-1} = \int_0^1 f(x)dx$, which can be easily extended to a more general integral of $I = \int_a^b f(x)dx$. Suppose that the random variables $x_1, x_2, \dots, x_N$ are drawn independently from the probability density function $p(x)$. A function F may be defined as

$$F = \sum_{i=1}^{N} f(x_i)p(x_i).$$

The expectation value of F becomes

$$E(F) = \int_0^1 f(x)p(x)dx.$$

The crude Monte Carlo integration method assumes that the probability density function $p(x)$ is uniform, i.e., the random samples $f(x_1), f(x_2), \dots, f(x_N)$ are equally important, and then

$$E(F) = \int_0^1 f(x)dx.$$

Correspondingly, the variance of F becomes

$$Var(F) = \frac{1}{N}\int_0^1 (f(x) - E(F))^2 dx = \frac{1}{N}\sigma^2,$$

where σ^2 is the inherent variance of the integrant function $f(x)$. Clearly, we can find that the standard deviation of the estimator θ is $\sigma N^{-\frac{1}{2}}$. This means that as $N \to \infty$, the distribution of F narrows around its mean at the rate of O($N^{-\frac{1}{2}}$).

Now, let us extend the Monte Carlo integration method to a d-dimensional integral $I_d = \int_0^1 \dots \int_0^1 f(x)dx$, the expectation of $F_d = \sum_{i=1}^{N} f(x_i) / N$ on uniformly distributed random variable vectors $x_1, x_2, \dots, x_N$ becomes

$$E(F_d) = \int_0^1 \dots \int_0^1 f(x)dx = I_d.$$

The variance of the estimator F_d is σ_d^2/N, where σ_d^2 is the inherent variance of the integrand function $f(x)$. If $f(x)$ is given, σ_d^2 is a constant and therefore, similar to one-dimensional integral, the convergence rate of Monte Carlo is O($N^{-\frac{1}{2}}$), which is independent of dimensionality.

In summary, compared to the deterministic numerical integration methods, whose convergence rate is O($N^{-\frac{\alpha}{d}}$), where α is the algorithm-related constant and d is the dimension, Monte Carlo integration method yields a convergence rate of O($N^{-\frac{1}{2}}$) [4], which can avoid the "curse of dimensionality." Moreover, computations on each random sample are independent, which can be carried out in an embarrassingly parallel manor to harness the power of large-scale parallel and distributed computing architectures [5, 6].

2.1.2 Importance Sampling

Crude Monte Carlo treats all random samples in an equally important way. In reality, we can often gain additional knowledge from the application domain, which can be taken advantage to come up with better estimators. Variance reduction is a procedure of deriving an alternative estimator to obtain a smaller variance than the crude Monte Carlo estimator and to improve the accuracy of the Monte Carlo estimates given a certain number of samples. In practical applications, a good estimator leading to million times more accurate than a bad one is not rarely seen. Some of the popular variance reduction techniques [1, 3, 4] include stratified sampling, control variates, antithetic variates, and importance sampling. These variance reduction methods, if appropriately used, can significantly improve the efficiency of Monte

Carlo methods in processing and analyzing big data sets. Here, we concentrate on describing the idea of importance sampling. The details of other variance reduction techniques can be found in [1].

The importance sampling technology is often used in statistical resampling, which reduces statistical variance by emphasizing the sampling on regions of interest with higher probability. For example, by introducing a new proposal function $g(x)$, the original integral $I_{0-1} = \int_0^1 f(x)dx$ can be rewritten as

$$I_{0-1} = \int_0^1 \frac{f(x)}{g(x)} g(x)dx = \int_0^1 \frac{f(x)}{g(x)} dG(x),$$

where $G(x)$ is a cumulative density function (CDF). $f(x)/g(x)$ is called the likelihood ratio. With random samples drawn from a proposal distribution whose CDF is $G(x)$ instead of sampling from a uniform distribution, the variance of the importance sampling estimator $F_{importance\ sampling}$ becomes

$$Var\left(F_{importance\ sampling}\right) = \int_0^1 \left(\frac{f(x)}{g(x)} - E(F)\right)^2 dG(x).$$

A good likelihood ratio $\frac{f(x)}{g(x)}$ close to $E(F)$ can result in a significant statistical variance reduction.

In practice, assuming that we know nothing about the target distribution at the very beginning, we may have to start from uniform sampling. However, after initial sampling, we have a better estimation of the target distribution, which results in a more precise proposal function. The resampling process can be guided by the new proposal function and leads to a better approximation of the target distribution.

2.2 Linear Algebra Operations

Linear algebra operations, such as solving linear systems, constructing low-rank matrix approximation, and approximating extreme eigenvalues and eigenvectors are pervasive in various scientific and engineering domains.

2.2.1 Linear System Solvers

2.2.1.1 Classes of Linear Systems Solvers

Considering a linear system in the form of

$$Ax = b,$$

where A is an $n \times n$ non-singular matrix, b is a given constant vector, and x is an unknown vector, existing numerical solvers can be roughly categorized into the following four main groups:

(1) Direct methods [7], such as Gaussian elimination and LU decomposition. These methods transform the original linear system into a form that can be solved in an easier manner. For example, denoting $A = MN$ as the LU factorization of A, the original linear system $Ax = b$ is then transformed into two relatively easy-to-solve linear systems $My = b$ and $Nx = y$. These direct methods are numerically stable and suitable for cases involving small and dense matrices.

(2) Stationary iterative methods [8], such as Jacobi method and Gauss-Seidel method. These methods transform $Ax = b$ into a new linear form $x = Hx + c$. Based on this, starting with a given initial x_0, stationary iterative methods update the solution vector by $x_{k+1} = Hx_k + c$ at each iteration. This iteration process repeats until convergence is reached. The stationary iterative methods are applicable to large and sparse systems, but its convergence condition is theoretically limited, such as requiring the spectral radius of H to be less than 1.

(3) Krylov subspace methods, including Conjugate Gradient (CG) method [9], Biconjugate Gradients (BiCG) method [10], and Generalized Minimal Residual (GMRES) method [11]. In general, Krylov subspace $\{r, Ar, A^2r, \dots, A^sr, \dots\}$ is constructed to search a good approximation to the solution. Compared to stationary iterative methods, Krylov subspace methods often yield broader convergence conditions than stationary iterative methods.

(4) Monte Carlo methods, such as Ulam and von Neumann algorithm [12] and Monte Carlo Almost Optimal (MAO) [13], which apply stochastic sampling to estimate the solution. Consider a linear system $x = Hx + c$, Monte Carlo methods first build up a probability matrix P with an unbiased

estimator $X(\gamma) = \frac{H_{r_0 r_1} H_{r_1 r_2} \dots H_{r_{k-1} r_k}}{P_{r_0 r_1} P_{r_1 r_2} \dots \mathrm{P}_{r_{k-1} r_k}} c_{r_k} \big/ T_{r_k}$, and independent random samples are then spawned to approximate the solution.

2.2.1.2 Conventional Monte Carlo Solvers

Applying Monte Carlo methods to estimate solutions of linear systems is originally proposed by Ulam and von Neumann and later described by Forsythe and Leibler in [12]. Considering a linear system in the form of

$$x \;=\; Hx \;+\; b$$

where H is an $n \times n$ non-singular matrix, b is a given constant vector, and x is the unknown vector, the fundamental idea of the Monte Carlo linear solver using Ulam-von Neumann algorithm is to construct Markov chains by spawning terminating random walks. The transition probabilities of the random walks are defined by a transition matrix P satisfying the following transition conditions:

$$P_{ij} \geq 0;$$

$$\sum_j P_{ij} \leq 1\,;$$

$$H_{ij} \neq 0 \rightarrow P_{ij} \neq 0$$

and the termination probability T_i at row i is defined as

$$T_i = 1 - \textstyle\sum_j P_{ij}.$$

Then, a random walk starting at r_0 and terminating after k steps is defined as

$$\gamma_k \colon r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_k$$

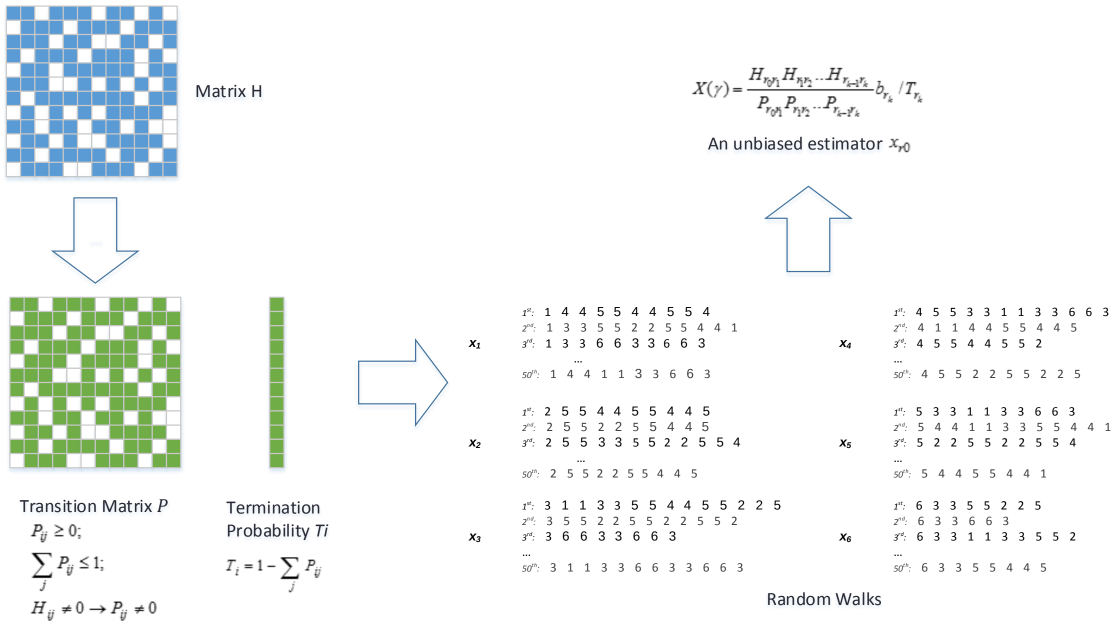
where the integers $r_0, r_1, r_2, \dots, r_k$ are the row indices of matrix H visited during the random walk. A random variable $X(\gamma_k)$ defined as

$$X(\gamma_k) = \frac{H_{r_0 r_1} H_{r_1 r_2} \dots H_{r_{k-1} r_k}}{P_{r_0 r_1} P_{r_1 r_2} \dots P_{r_{k-1} r_k}} b_{r_k} / T_{r_k}$$

is an unbiased estimator of component x_{r_0} in the unknown vector x. The fundamental idea of Ulam-von Neumann algorithm is to statistically sample the underlying Neumann series

$$I + H + H^2 + H^3 + \cdots$$

of the linear system. Denoting $\|\ .\ \|$ to be the L-∞ norm, as specified in the Monte Carlo linear solver literature [3], if $||H|| < 1$, the Neumann series converge to $(I - H)^{-1}$ and, hence, $X(\gamma_k)$ is an unbiased estimator of $(H^k b)_{r_0}$, while $\sum_{k=1}^{\infty} X(\gamma_k) P(\gamma_k)$ equals to the solution x_{r_0}. Fig. 2 shows the procedure of Monte Carlo linear solver using Ulam-von Neumann scheme.


Matrix H
$X(\gamma) = \frac{H_{r_0r_1}H_{r_1r_2}\cdots H_{r_{k-1}r_k}}{P_{r_0r_1}P_{r_1r_2}\cdots P_{r_{k-1}r_k}} b_{r_k} / T_{r_k}$
An unbiased estimator x_{r0}
Transition Matrix *P*
$P_{ij} \geq 0;$
$\sum_j P_{ij} \leq 1;$
$H_{ij} \neq 0 \rightarrow P_{ij} \neq 0$
Termination Probability *Ti*
$T_i = 1 - \sum_j P_{ij}$
x_1
1st: 1 4 4 5 5 4 4 5 5 4
2nd: 1 3 3 5 5 2 2 5 5 4 4 1
3rd: 1 3 3 6 6 3 3 6 6 3
...
50th: 1 4 4 1 1 3 3 6 6 3
x_2
1st: 2 5 5 4 4 5 5 4 4 5
2nd: 2 5 5 2 2 5 5 4 4 5
3rd: 2 5 5 3 3 5 5 2 2 5 5 4
...
50th: 2 5 5 2 2 5 5 4 4 5
x_3
1st: 3 1 1 3 3 5 5 4 4 5 5 2 2 5
2nd: 3 5 5 2 2 5 5 2 2 5 5 2
3rd: 3 6 6 3 3 6 6 3
...
50th: 3 1 1 3 3 6 6 3 3 6 6 3
x_4
1st: 4 5 5 3 3 1 1 3 3 6 6 3
2nd: 4 1 1 4 4 5 5 4 4 5
3rd: 4 5 5 4 4 5 5 2
...
50th: 4 5 5 2 2 5 5 2 2 5
x_5
1st: 5 3 3 1 1 3 3 6 6 3
2nd: 5 4 4 1 1 3 3 5 5 4 4 1
3rd: 5 2 2 5 5 2 2 5 5 4
...
50th: 5 4 4 5 5 4 4 1
x_6
1st: 6 3 3 5 5 2 2 5
2nd: 6 3 3 6 6 3
3rd: 6 3 3 1 1 3 3 5 5 2
...
50th: 6 3 3 5 5 4 4 5
Random Walks
Fig. 2. Monte Carlo linear solver using Ulam-von Neumann scheme

The original Monte Carlo linear solver by Ulam-von Neumann is not efficient and its convergence relies on the properties of H and P. Later algorithms have also been developed to improve the Monte Carlo solver, by selecting a better transition matrix P or applying alternative transformations from $Ax = b$ to $x = Hx + b$ to accelerate convergence. Wasow [14] modified the scheme by Ulam and von Neumann by designing another unbiased estimator, which has been shown to have smaller variance under some special conditions. Halton [15] proposed a sequential Monte Carlo method to accelerate the Monte Carlo process by taking advantage of the rough estimate of the solution to transform the original linear system $x = Hx + b$ to a new system $y = Hy + d$, where $||d|| < ||b||$. Dimov et al. [16, 17] developed an accelerating Monte Carlo scheme to control the convergence of the Monte Carlo algorithm for different unknown elements with different relaxation parameters, which can increase the efficiency of the random walk estimators. Tan [18] studied the antithetic variates techniques for variance reduction in Monte Carlo linear solvers. Srinivasan and Aggarwal [19] used non-diagonal splitting to improve the Monte Carlo linear solvers. Moreover, for applications with large linear systems, Sabelfeld and Mozartova [20] designed a sparsified randomization algorithm by using a sparse, random matrix G, which is an unbiased estimator of H, to replace the original matrix H during the sampling process. Furthermore, Mascagni and Karaivanova [21] investigated the usage of quasirandom numbers in the Monte Carlo solver.

Compared to the deterministic linear solvers, the Monte Carlo linear solvers have several uniquely attractive advantages in handling extremely large coefficient matrices [153,154]. First of all, the Monte Carlo linear solvers are based on sampling, which do not need to access all elements in A at every iteration step. This is particularly suitable for applications such as large-scale sensor networks where every element in A is available for access but getting the complete picture of the matrix A is costly or practically infeasible. This is also helpful for handling incomplete or imperfect data. Secondly, random walks in the Monte Carlo linear solvers can be carried out independently in a distributed manner, which is favorable for the nowadays large-scale parallel and distributed processing platforms. Thirdly, the Monte Carlo linear solvers can obtain a quick approximation to solutions with low resolution. Fourthly, random walks in the Monte Carlo linear solvers have little memory requirements and the random walk algorithm is scalable with the size of the matrices. Finally, for applications interested in only a few elements in the unknown vector, using the Monte Carlo linear solvers based on Ulam-von Neumann algorithm can eliminate unnecessary computations for other elements in the unknown vector.

2.2.2 Constructing Low-rank Matrix Approximations

2.2.2.1 Low-rank Matrix Approximations

Considering an $m \times n$ matrix A with rank r, the optimal k-rank ($k \leq r$) approximation A_k of matrix A yields minimum approximation error among all possible $m \times n$ matrices of rank k [7], i.e.,

$$\|A - A_k\|_F^2 = \min_{rank(X)=k} \|A - X\|_F^2.$$

Within controllable approximation error, a good low-rank approximation of a large matrix can reduce storage requirement and accelerate matrix operations such as matrix-vector or matrix-matrix multiplications. If A is a matrix representing data affinity in a large dataset, low-rank approximation can be used for dimension reduction or noise elimination. As a result, constructing low-rank approximations of large matrices plays a central role in many data analytic applications [1, 22, 23, 24, 25], such as principle component analysis, compressed sensing, data compression, manifold learning, and matrix completion.

The optimal k-rank approximation A_k can be straightforwardly obtained by computing full Singular Value Decomposition (SVD) and truncating it by selecting the dominant singular values and their corresponding singular vectors such that

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T,$$

where $k \leq r$, $\sigma_1, \sigma_2, \ldots, \sigma_k$ are the singular values of A in non-increasing order, and $u_1, \cdots, u_k$ and $v_1, \cdots, v_k$ are the corresponding left and right singular vectors, respectively. Here, by tuning the value of k, the low-rank matrix approximation error measured by Frobenius norm can be controlled by

$$\|A - A_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2.$$

2.2.2.2 Fast Monte Carlo methods for Low-rank approximation

Numerically computing the full SVD of a matrix when both m and n are large is often prohibitively computationally costly as well as memory intensive. As the efficient alternatives, randomized algorithms to approximate SVD have attracted great interest recently and become competitive for rapid low-rank approximations of large matrices [22, 26, 27, 28]. Instead of passing over the large matrix in full SVD, the randomized SVD algorithms focus on efficiently sampling the important matrix elements. Many sampling strategies, including uniform column sampling (with or without replacement)

[29, 30], diagonal sampling or column-norm sampling [31], sampling with k-means clustering [32], and Gaussian sampling [33], have been proposed. As a result, compared to full SVD, randomized SVD methods are memory efficient and can often obtain low-rank approximation in a significantly faster way.

Nevertheless, most of these randomized SVD algorithms require the rank value k to be given as an input parameter in advance. In many practical applications, k is unknown beforehand but is of great importance to the accuracy of the solutions. In general, underestimating k can introduce unacceptable large error in the low-rank approximation while overestimating k can lead to unnecessary computational and memory costs. Without prior knowledge of the distribution of the singular values, in practice, it is not uncommon to re-run fast Monte Carlo methods many times until a good value of k is determined, which is a waste of computational resources.

2.2.3 Approximating Extreme Eigenvalues and Eigenvectors

2.2.3.1 Extreme Eigenvalues and Eigenvectors

Calculating the extreme eigenvalues/eigenvectors of a matrix is often required in many fields of science and engineering. An eigenvector u of an $n \times n$ matrix A is a vector that satisfies

$$Au = \lambda u,$$

where λ is an eigenvalue of matrix A. A few of the largest or smallest eigenvalues and the corresponding eigenvectors are called extreme eigenvalues and eigenvectors. In particular, the dominant eigenvalue of matrix A refers to the eigenvalue with the largest absolute value.

2.2.3.2 Power method for Extreme Eigenvalues/Eigenvectors

Let $\lambda_1, \lambda_2, \dots, \lambda_n$, ($|\lambda_1| \geq |\lambda_2| \geq, \dots, \geq |\lambda_n|$) be eigenvalues of matrix A of order n and v_1, v_2, ..., v_n the corresponding eigenvectors. The direct method of calculating extreme eigenvalues and eigenvectors is to obtain the eigenvalues from the polynomial equation $det(A - \lambda I) = 0$, and then the eigenvectors can be computed by solving each linear system $(A - \lambda_i I)u_i = 0$ accordingly. However, this procedure is not practical for large matrices, due to its high computational cost and memory requirement.

To handle a very large, sparse matrix, one of the most popular methods is power method [34]. Starting from a random vector x_0, the power method is described by the power iteration

$$x_{i+1} = Ax_i,$$

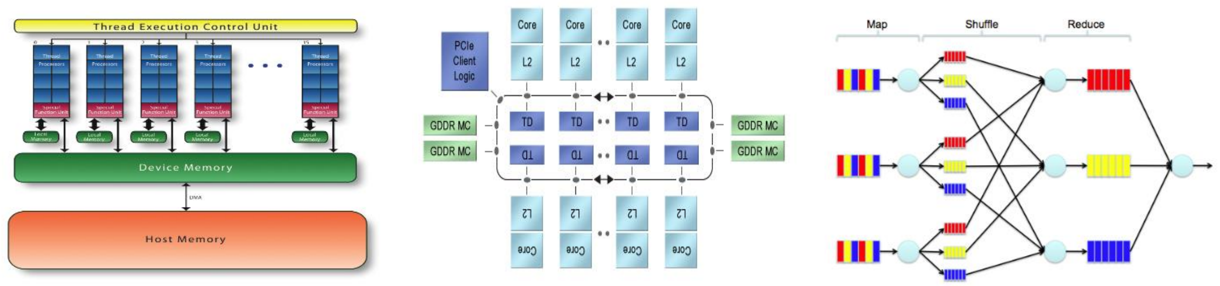
which eventually converges to the dominant eigenvalue and eigenvector [7]. In general, normalizing x_{i+1} to the unit norm is carried out to avoid a vector of large magnitude. The power method has been popularly used in a variety of real-life applications, which is regarded as the only feasible method with least memory requirement when the matrix is very large and sparse. For example, the power method is used in Google's PageRank algorithm to rank webpages in the Internet in their search engine results [35]; Twitter employs the power method to recommend "who to follow" to its users [36]; by exploring the graph constructed via content and link features, the power iteration is also applied to calculate the trust vector as the stationary distribution vector of the graph to fight spams [37].

Similarly, under the assumption of the existence of $(A - \mu I)^{-1}$, the inverse iteration [7] employs $x_{i+1} = (A - \mu I)^{-1} x_i$ to approximate the eigenvector corresponding to eigenvalue closest to μ, where I is the identity matrix. Typically, when μ is set to 0, the resulting vector approximates the eigenvector corresponding to the eigenvalue with smallest magnitude.

One of the main disadvantages in the power method is that its convergence speed is governed by the eigengap between the first two largest eigenvalues λ_1 and λ_2 of A in absolute values. If $|\lambda_2|$ is very close to $|\lambda_1|$, a large number of power iterations are often needed to reach convergence, even for a small matrix. This is particularly unfavorable for large matrices where passing over all elements is costly.

2.3 Parallelism in Matrix Computations

Implementations of large-scale linear algebra, including matrix-vector multiplication, matrix-matrix multiplication, QR decomposition, and Singular Value Decomposition (SVD), require fully taking advantage of modern parallel and distributed computing paradigms to achieve good performance. For instance, General Purpose Graphics Process Units (GPGPU), Intel Xeon Phi, and Cloud Distributed Systems (Fig. 3).



GPGPU Intel Xeon Phi Cloud Distributed Systems

Fig. 3. Modern parallel/distributed computing paradigms/architectures

Many high-performance linear algebra libraries are available to increase application performance on specific hardware architecture. For instance, to support linear algebra computations on GPU, CUBLAS (CUDA Basic Linear Algebra Subroutines) [38] contains the GPU-accelerated functions of basic dense matrix operations. Complementary to CUBLAS, CULA [39] is an extended linear algebra library provides high-level equivalent routines of LAPACK over CUDA runtime, MAGMA library [40] contains advanced matrix decompositions functions, and CUSPARSE [41] is a library for sparse matrix operations. Moreover, on multi-core CPUs and Intel Xeon Phi, MKL (Math Kernel Library) [121] is widely used to accelerate linear algebra routines. Furthermore, on distributed-memory systems, PBLAS (Parallel Basic Linear Algebra Subprograms) [146] and ScaLAPACK (Scalable Linear Algebra PACKage) [147] are popular libraries used in many parallel computing applications.

With the growing size of large matrices in linear algebra operations, efficiently implementing large-scale matrix computations on the emerging big data platforms are of primary interest nowadays. For example, Fig. 4 shows the flowchart of a QR decomposition implementation on a tall-and-skinny matrix using MapReduce [42, 43] in an uneven, distributed fashion, which achieves load balancing and has a clear performance advantage over the classic Householder QR algorithm.

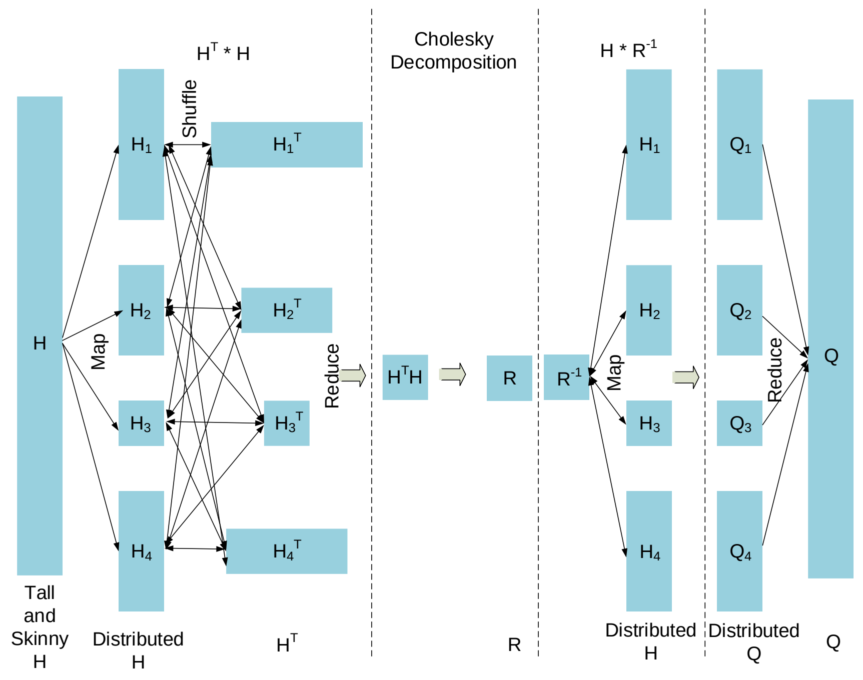


Fig. 4. The computation of a QR decomposition of a “Tall-and-Skinny,” dense matrix *H*

2.4 Numerical Verification Techniques

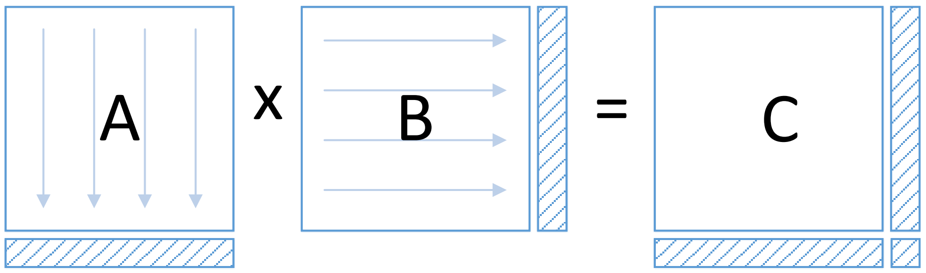
As the demands on modern linear algebra applications created by the latest development of high-performance computing (HPC) architectures continue to grow, so does the likelihood that they are vulnerable to faults. Soft faults in computer systems, defined as intermittent events that corrupt the data being processed, are among the most worrying, particularly when the computation is carried out in a low-voltage computing environment. For example, the 2,048-node ASC Q supercomputer at Los Alamos National Laboratory reports an average of 24.0 board-level cache tag parity errors and 27.7 CPU failures per week [44]; the 131, 072-CPU BlueGene/L supercomputer at Lawrence Livermore National Laboratory experiences one soft error in its L1 cache every 4–6 hours [45]; more recently, a field study on Google’s servers reported an average of 5 single bit errors occur in 8 Gigabytes of RAM per hour using the top-end error rate [46]. The reliability of computations on HPC systems can suffer from soft errors that occur in memory, cache, as well as microprocessor logic [47], and thus produce potentially incorrect results in a wide variety of ways. Therefore, the appropriate approaches to remedy the consequences of soft errors for certain linear algebra applications are needed.

Matrix-matrix multiplication is one of the most fundamental numerical operations in linear algebra. Many important linear algebraic algorithms, including linear solvers, least squares solvers, matrix decompositions, factorizations, subspace projections, and eigenvalue/singular values computations, rely on the casting the algorithm as a series of matrix-matrix multiplications. This is partly because matrix-matrix multiplication is one of the level-3 Basic Linear Algebra Subprograms (BLAS) [48, 49, 50]. Efficient implementation of the BLAS remains an important area for research, and often computer vendors spend significant resources to provide highly optimized versions of the BLAS for their machines. Therefore, if a matrix-matrix multiplication can be carried out free of faults, the linear algebraic algorithms that spend most of their time in matrix-matrix multiplication can themselves be made substantially fault-tolerant [51]. Two relevant algorithms from the literature for error detection in matrix-matrix multiplication are described below.

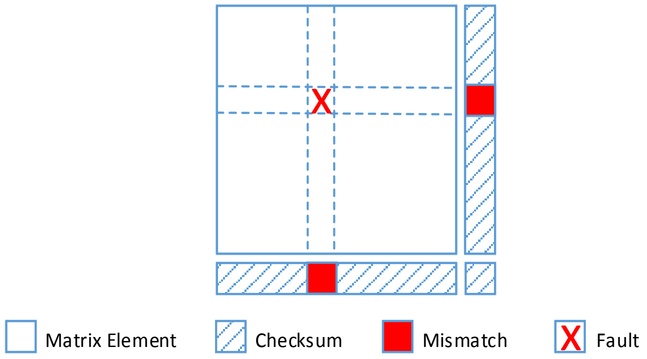
2.4.1 The Huang-Abraham Scheme

The Huang and Abraham scheme [52] is an algorithm-based fault tolerance (ABFT) method that simplifies detecting and correcting errors when carrying out matrix-matrix multiplication operations. This is slightly different from the matrix product verification problem. The fundamental idea of the Huang-Abraham scheme is to address the fault detection and correction problem at the algorithmic level by calculating matrix checksums, encoding them as redundant data, and then redesigning the algorithm to operate on these data to produce encoded output. Compared to the traditional fault tolerant techniques, such as checkpointing [53], the overhead of storing additional checksum data in the Huang-Abraham scheme is small, particularly when the matrices are large. Moreover, no global communication is necessary in the Huang-Abraham scheme [11]. The Huang and Abraham scheme forms the basis of many subsequent checking schemes, and has been extended for use in various HPC architectures [128,129,130].

Fig. 5 illustrates the Huang and Abraham scheme [52] for detecting faults in matrix-matrix multiplication. First of all, column sums for A and row sums for B are generated and are added to an augmented representation of A and B. These are treated as particular checksums in the subsequent multiplication. Then, multiplication of the extended matrices produces the augmented matrix for C (Fig. 5 (a)) where the checksums can be readily compared. Mismatches in the row and column checksums indicate an element fault in matrix product, C (Fig. 5 (b)).



(a): Generation of column checksum for A and row checksum for B and multiplication of the extended matrices to produce the checksum matrix for C



(b): Mismatches in the row and column checksums indicate an element fault in matrix product
Fig. 5. The Huang-Abraham scheme for detecting faults in matrix-matrix multiplication

However, there are certain patterns of faults undetectable by the Huang-Abraham scheme. Here is a simple 2×2 example to illustrate such an undetectable pattern.

Consider the matrices

$$A = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & -6 \\ 1 & 6 \end{bmatrix}, C = \begin{bmatrix} 5 & 6 \\ 7 & 6 \end{bmatrix}.$$

Clearly $A \times B = C$ holds in this example. Then we use the Huang-Abraham scheme to calculate the column checksum for A and row checksum for B and we can get

$$A_F = \begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 5 & 7 \end{bmatrix} \text{ and } B_F = \begin{bmatrix} 1 & -6 & -5 \\ 1 & 6 & 7 \end{bmatrix}.$$

Then

$$A_F \times B_F = \begin{bmatrix} 5 & 6 & 11 \\ 7 & 6 & 13 \\ 12 & 12 & 24 \end{bmatrix} = C_F.$$

However, if there is a fault during the computation of C which causes the swap between the first and second column, an erroneous result matrix $C' = \begin{bmatrix} 6 & 5 \\ 6 & 7 \end{bmatrix}$ is generated by swapping columns of C. Column or row swapping, usually caused by address decoding faults [131], is a commonly observed memory fault pattern [132]. The problem is that the checksum matrix of C' becomes $C'_F = \begin{bmatrix} 6 & 5 & 11 \\ 6 & 7 & 13 \\ 12 & 12 & 24 \end{bmatrix}$, where both the row and column checksums match those of the true product of $A \times B$. Consequently, the Huang-Abraham scheme fails to detect this fault.

The Huang-Abraham scheme can be viewed as a linear constraint satisfaction problem (CSP), where the variables are the n^2 entries in the resulting matrix, the constraints are the $2n$ row and column checksums, and the $2n \times n^2$ coefficient matrix in the underdetermined linear CSP system equation specifies the selection of row or column elements, as shown in Fig. 6. Clearly, a result matrix, C, that does not satisfy the CSP equations indicates errors in C detectable by the Huang-Abraham scheme. The unique, correct result matrix, C, satisfies the CSP equations. Nevertheless, other possible result matrices satisfying the CSP equations are the fault patterns undetectable by the Huang-Abraham scheme. Only when at least n^2 constraints with different element selection are incorporated so that the rank of the coefficient matrix in the CSP equation is n^2, can the undetectable fault patterns be eliminated. However, in this case, it is equivalent to simply checking every element in C.

$$\mathbf{2n}\left\{ \overbrace{\begin{bmatrix} \mathbf{1} & \mathbf{1} & \cdots & \mathbf{1} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{1} & \mathbf{1} & \cdots & \mathbf{1} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ & & & & & & \vdots & & & & & & \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{1} & \mathbf{1} & \cdots & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{1} & \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{1} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{1} & \cdots & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{1} & \cdots & \mathbf{0} \\ & & & & & & \vdots & & & & & & \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1} & \cdots & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{1} \end{bmatrix}}^{\boldsymbol{n^2}} \right. \begin{bmatrix} \boldsymbol{C_{1,1}} \\ \boldsymbol{C_{1,2}} \\ \vdots \\ \boldsymbol{C_{1,n}} \\ \boldsymbol{C_{2,1}} \\ \boldsymbol{C_{2,2}} \\ \vdots \\ \boldsymbol{C_{2,n}} \\ \vdots \\ \boldsymbol{C_{n,1}} \\ \boldsymbol{C_{n,2}} \\ \vdots \\ \boldsymbol{C_{n,n}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{(C_F)_{1,n+1}} \\ \boldsymbol{(C_F)_{2,n+1}} \\ \vdots \\ \boldsymbol{(C_F)_{n,n+1}} \\ \boldsymbol{(C_F)_{n+1,1}} \\ \boldsymbol{(C_F)_{n+1,2}} \\ \vdots \\ \boldsymbol{(C_F)_{n+1,n}} \end{bmatrix}$$

Fig. 6. Underdetermined CSP system in the Huang-Abraham Scheme

It is important to notice that there are an infinite number of existing fault patterns that satisfy the checksum constraints and thus are undetectable by the Huang-Abraham scheme, even in the above simple 2×2 example (the rank of the CSP coefficient matrix is 3). Moreover, as dimension, n, increases, the number of checksum constraints increases only linearly but the number of elements in a matrix has quadratic growth. Therefore, the undetectable patterns in Huang-Abraham scheme increase dramatically with n. As a result, for multiplications in large matrices, fault detection methods based on the Huang-Abraham scheme can generate false positive results for a large number of circumstances.

2.4.2 The Freivalds' Algorithm

The fault detection methods based on the Huang-Abraham scheme are deterministic algorithms. With the tradeoff of random uncertainty, Freivalds [54] showed that a probabilistic machine can verify the correctness of matrix product faster than direct recalculation. The procedure of the corresponding method, later named Freivalds' algorithm (Algorithm 2.1), is described as follows.

```
Algorithm 2.1: Freivalds' Algorithm
Step 1. Randomly sample a vector $\omega \in \{0, 1\}^n$ with probability ½ of 0 or 1.
Step 2. Calculate the projection of C onto $\omega$: $C\omega = C \times \omega$.
Step 3. Calculate the projection of the product $A \times B$ onto $\omega$: $AB\omega = A \times (B \times \omega)$.
```

Obviously, if $A \times B = C$, $C\omega = AB\omega$ always holds. Freivalds proved that when $A \times B \neq C$, the probability of $C\omega = AB\omega$ is less than or equal to 1/2. The runtime of the above procedure is $O(n^2)$ with an implied multiplier of 3, as it is comprised of three matrix-vector multiplications. This is an upper bound as one can perhaps optimize the evaluation of $B\omega$ and $C\omega$. By iterating the Freivalds' algorithm k times, the runtime becomes $O(kn^2)$ and the probability of a false positive becomes less than or equal to 2^{-k}, according to the one-sided error. More generalized forms of Freivalds' algorithm have also been developed, mainly based on using different sampling spaces [133,134,135,136]. Given at most p erroneous entries in the resulted matrix product, Gasieniec, Levcopoulos, and Lingas extended Freivalds' algorithm to one with correcting capability running in $O(\sqrt{p}n^2 log(n) log(p))$ time [137].

CHAPTER III

MONTE CARLO METHODS FOR LINEAR SYSTEMS SOLVERS

3.1 Convergence Analysis of Ulam-von Neumann Algorithm

The fundamental idea behind conventional Monte Carlo solvers as introduced in Chapter II, is to construct Markov chains based on random walks to estimate the underlying Neumann series

$$I + H + H^2 + H^3 + \ldots$$

to evaluate solutions of the linear systems.

As pointed out in [3], if $\|H\| > 1$, the Monte Carlo method breaks down. Nevertheless, it is well known that the necessary and sufficient condition for the Neumann series to converge is $\rho(H) < 1$, where $\rho(H)$ is the spectral radius of H. Proposition 3.1 shows that $\|H\| < 1$ is a stricter condition than $\rho(H) < 1$. Therefore, there exists a family of matrices whose corresponding Neumann series converge but that the Monte Carlo linear solver cannot converge.

Proposition 3.1. For an $N \times N$, nonsingular matrix H, $\rho(H) \leq \|H\|$.

Proof. Let λ be an eigenvalue of H and y the corresponding eigenvector. Thus $\lambda y = Hy$, and $\|\lambda y\| = \|\lambda\|\|y\| = \|Hy\| \leq \|H\|\|y\|$. Finally, $\|\lambda\| \leq \|H\|$ for all eigenvalues of H and $\rho(H) \leq \|H\|$, since $\rho(H)$ is the largest absolute value of the eigenvalues of H.

3.1.1 Suggestive Examples

To investigate the condition for convergence of conventional Monte Carlo linear solvers, we start considering a set of suggestive examples with 2×2 matrices (Table 1) to study the behavior of the Monte Carlo linear solver using Ulam-von Neumann algorithm [153]. We find that although the Monte Carlo solver is based on sampling the Neumann series, the convergence of Neumann series is not a sufficient condition for the convergence of the Monte Carlo solver. Actually, properties of H are not the only factors determining the convergence of the Monte Carlo solver; the underlying transition probability matrix P plays an important role.

TABLE 1
Behavior of the Monte Carlo Linear Solver using Ulam-von Neumann Algorithm in 6 Cases of 2×2 Matrices under Different Conditions and Transition Matrices

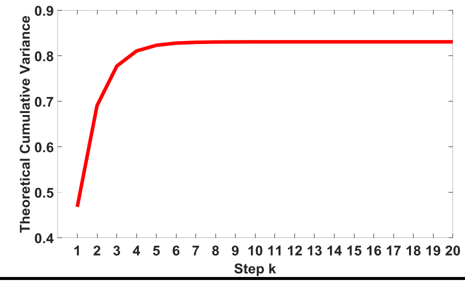
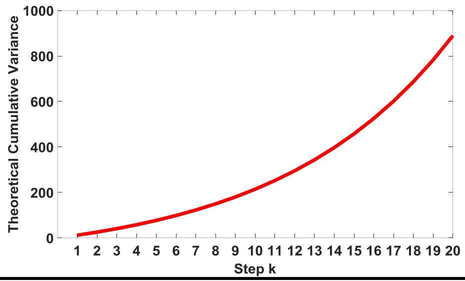
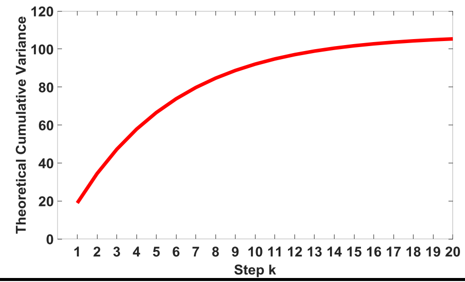
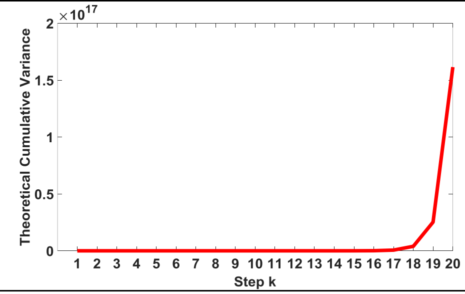
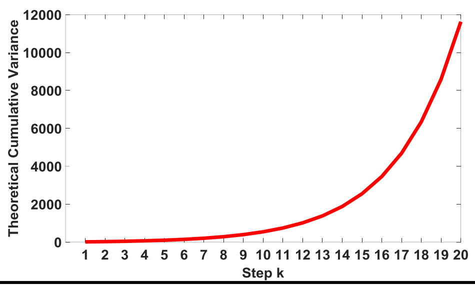
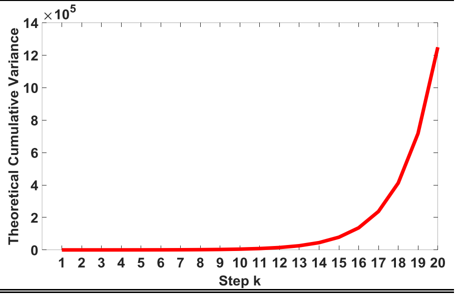
Case	H and P	Conditions	Converged?	$Var\left(\sum_k X(\gamma_k)\right)$
1	$H = \begin{bmatrix} 0.1 & 0.3 \\ 0.3 & -0.05 \end{bmatrix}$ $P = \begin{bmatrix} 0.1 & 0.3 \\ 0.3 & 0.05 \end{bmatrix}$	$\|H\| < 1$ $\rho(H) < 1$ $\rho(H^+) < 1$ $\rho(H^*) < 1$	Yes	
2	$H = \begin{bmatrix} 0.1 & 0.3 \\ 0.3 & -0.05 \end{bmatrix}$ $P = \begin{bmatrix} 0.009 & 0.891 \\ 0.8 & 0.1 \end{bmatrix}$	$\|H\| < 1$ $\rho(H) < 1$ $\rho(H^+) < 1$ $\rho(H^*) > 1$	No	
3	$H = \begin{bmatrix} 0.8 & 0.35 \\ 0.1 & -0.01 \end{bmatrix}$ $P = \begin{bmatrix} 0.8 & 0.1 \\ 0.7 & 0.2 \end{bmatrix}$	$\|H\| > 1$ $\sum_{j=1}^{N} \|H_{ij}\| > 1$ for some but not all i $\rho(H) < 1$ $\rho(H^+) < 1$ $\rho(H^*) < 1$	Yes	
4	$H = \begin{bmatrix} 0.8 & 0.35 \\ 0.1 & -0.01 \end{bmatrix}$ $P = \begin{bmatrix} 0.1 & 0.8 \\ 0.7 & 0.2 \end{bmatrix}$	$\|H\| > 1$ $\sum_{j=1}^{N} \|H_{ij}\| > 1$ for some but not all i $\rho(H) < 1$ $\rho(H^+) < 1$ $\rho(H^*) > 1$	No	

TABLE 1 Continued

Case	H and P	Conditions	Converged?	$Var\left(\sum_k X(\gamma_k)\right)$
5	$H = \begin{bmatrix} 0.4012 & 0.5305 \\ 0.5305 & -0.7023 \end{bmatrix}$ $P = \begin{bmatrix} 0.3306 & 0.5694 \\ 0.3303 & 0.5697 \end{bmatrix}$	$\|H\| > 1$ $\sum_{j=1}^{N} \lvert H_{ij} \rvert > 1$ for some but not all i $\rho(H) < 1$ $\rho(H^{+}) > 1$ $\rho(H^{*}) > 1$	No	
6	$H = \begin{bmatrix} 0.3968 & -0.7162 \\ -0.7162 & -0.6226 \end{bmatrix}$ $P = \begin{bmatrix} 0.2565 & 0.6435 \\ 0.5350 & 0.3650 \end{bmatrix}$	$\|H\| > 1$ $\sum_{j=1}^{N} \lvert H_{ij} \rvert > 1$ for all i $\rho(H) < 1$ $\rho(H^{+}) > 1$ $\rho(H^{*}) > 1$	No	

One can find that in all of these six suggestive cases in Table 1, the H matrices satisfy the spectral radius condition where $\rho(H) < 1$; however, the Monte Carlo linear solver does not converge in all of these cases. Hence, it is clear that the convergence of the underlying Neumann series is not a sufficient condition for the Monte Carlo linear solver to converge. More interestingly, cases 1 and 2 use the same H matrix where $||H|| < 1$ but different transition matrices P. The Monte Carlo linear solver converges in case 1 but diverges in case 2, indicating that the selection of transition matrix P is important. If P is selected improperly, the Monte Carlo linear solver may diverge even if $||H|| < 1$ holds. Furthermore, the H matrix in case 3 does not satisfy condition $||H|| < 1$, but the Monte Carlo linear solver does not break down, which disagrees with the analysis in [3] that "if $||H|| > 1$, the Monte Carlo method breaks down." The phenomenon in case 3 suggests that there are some situations when $||H|| > 1$ but $\rho(H) < 1$ that the Monte Carlo linear solver can still converge, i.e., $||H|| < 1$ is not a necessary condition for convergence in the Monte Carlo linear solver. Similar to the situation in cases 1 and 2, case 4 has the same H matrix as case 3 but different transition matrix P, which results in divergence. Cases 5 and 6 show the behavior of the Monte Carlo linear solver under $\rho(H^+) > 1$ when $\sum_{j=1}^{N}\left|H_{ij}\right| > 1$ for some but not all i and $\sum_{j=1}^{N}\left|H_{ij}\right| > 1$ for all i, respectively.

3.1.2 A Necessary and Sufficient Condition

We consider a Monte Carlo linear solver as converging if the variance of the estimator $\sum_k X(\gamma_k)$,

$$Var\left(\sum_k X(\gamma_k)\right) = \sum_k Var\big(X(\gamma_k)\big)$$

is bounded as $k \to \infty$, provided that every random walk γ_k is independent. We first investigate the impact of selecting a transition matrix P on the convergence of the Monte Carlo linear solver. For convenience, we state what mathematical results are needed as lemmas. Also note that $Var(X(\gamma_k))$ diverging as $k \to \infty$, implies the same of $Var(\sum_k X(\gamma_k))$. Hence, when we study the convergence/divergence behavior of the Monte Carlo linear solver in the theorems, we only consider $Var(X(\gamma_k))$ instead of $Var(\sum_k X(\gamma_k))$. Without loss of generality and for simplicity, we also assume that the Markov chains in the Monte Carlo linear solver are ergodic and that every element in the constant vector b in the linear system satisfies $b_i \neq 0$, for all i.

By taking both H and P into consideration, we derive a necessary and sufficient condition for convergence of the Ulam–von Neumann Monte Carlo method, as shown in Theorem 3.3. Lemma 3.2 is used in the proof of Theorem 3.3.

Lemma 3.2. Let H be an $N \times N$ nonsingular matrix and b be a nonzero vector. If $\rho(H) < 1$, $\sum_{k=0}^{\infty}\left(H^k b\right)_{r_0}^2$ is bounded.

Proof: For any $\varepsilon > 0$, a matrix R is generated such that

$$R = \frac{H}{\rho(H) + \varepsilon}.$$

Due to that $0 < \rho(H) < 1$, it is easy to show that $\rho(R) = \frac{\rho(H)}{\rho(H)+\varepsilon} < 1$. Then,

$$\lim_{k\to\infty} R^k = 0.$$

Or, equivalently, this indicates that a natural number K exists such that $\forall k > K, \left\|R^k\right\| < 1$.

Accordingly,

$$\forall k > K, \left\|R^k\right\| = \left\|\left(\frac{H}{\rho(H)+\varepsilon}\right)^k\right\| = \frac{\left\|H^k\right\|}{(\rho(H)+\varepsilon)^k} < 1$$

That is,

$$\forall k > K, \left\|H^k\right\| < (\rho(H)+\varepsilon)^k .$$

Therefore, $\forall k > K,$

$$\left|\left(H^k b\right)_{r_0}\right| \le \left\|H^k b\right\| \le \left\|H^k\right\| \|b\| < (\rho(H)+\varepsilon)^k \|b\|$$

and

$$\left(H^k b\right)_{r_0}^2 \le \left\|H^k b\right\|^2 < (\rho(H)+\varepsilon)^{2k} \|b\|^2.$$

In particular, since ε can be any positive number, we can set $\varepsilon = c^{\frac{1}{2}} - \rho(H) > 0$, where c is a positive number such that $\rho(H)^2 < c < 1$. Then

$$\left(H^k b\right)_{r_0}^2 < c^k \|b\|^2, \qquad \forall k > K$$

Hence,

$$\begin{aligned}\sum_{k=0}^{\infty}\left(H^k b\right)_{r_0}^2 &= \sum_{k=0}^{K}\left(H^k b\right)_{r_0}^2 + \sum_{k=K+1}^{\infty}\left(H^k b\right)_{r_0}^2 \\ &\le \sum_{k=0}^{K}\left(H^k b\right)_{r_0}^2 + \sum_{k=K+1}^{\infty} c^k \|b\|^2 \\ &= \sum_{k=0}^{K}\left(H^k b\right)_{r_0}^2 + \|b\|^2 \sum_{k=K+1}^{\infty} c^k \\ &= \sum_{k=0}^{K}\left(H^k b\right)_{r_0}^2 + \frac{\|b\|^2 c^{K+1}}{1-c}\end{aligned}$$

Since $\sum_{k=0}^{K}\left(H^k b\right)_{r_0}^2$ has finite number of terms, and $\frac{\|b\|^2 c^{K+1}}{1-c}$ is a constant, $\sum_{k=0}^{\infty}\left(H^k b\right)_{r_0}^2$ is bounded. $\square$

Theorem 3.3. Given an $N \times N$ nonsingular matrix H such that $\rho(H) < 1$, a nonzero vector b, and a transition matrix P, the necessary and sufficient condition for convergence of the Monte Carlo linear solver using the Ulam-von Neumann algorithm is $\rho(H^*) < 1$, where H^* is an $N \times N$ matrix such that $H^*_{ij} = \frac{H^2_{ij}}{\mathrm{P}_{ij}}$.

Proof. Since

$$\begin{aligned} Var(X(\gamma_k)) &= E((X(\gamma_k))^2) - \big(E(X(\gamma_k))\big)^2 \\ &= \sum_{r_1=1}^{N}\sum_{r_2=1}^{N}\cdots\sum_{r_k=1}^{N} P_{r_0r_1}P_{r_1r_2}\dots P_{r_{k-1}r_k}T_{r_k}\left(\frac{H_{r_0r_1}H_{r_1r_2}\dots H_{r_{k-1}r_k}b_{r_k}}{P_{r_0r_1}P_{r_1r_2}\dots P_{r_{k-1}r_k}T_{r_k}}\right)^2 - \left(H^k b\right)^2_{r_0} \\ &= \sum_{r_1=1}^{N}\sum_{r_2=1}^{N}\cdots\sum_{r_k=1}^{N} \frac{H^2_{r_0r_1}H^2_{r_1r_2}\dots H^2_{r_{k-1}r_k}b^2_{r_k}}{P_{r_0r_1}\mathrm{P}_{r_1r_2}\dots P_{r_{k-1}r_k}T_{r_k}} - \left(H^k b\right)^2_{r_0} \\ &= \left({H^*}^k b^*\right)_{r_0} - \left(H^k b\right)^2_{r_0} \end{aligned}$$

where b^* is a nonzero vector such that $b^*_i = \frac{b_i^2}{T_i}$, and T_i is the termination probability at row i, in the Ulam-von Neumann algorithm. If the k random walks are independent, it follows that

$$\begin{aligned} Var\left(\sum_{k=0}^{\infty} X(\gamma_k)\right) &= \sum_{k=0}^{\infty} Var\big(X(\gamma_k)\big) \\ &= \sum_{k=0}^{\infty}\left(\left({H^*}^k b^*\right)_{r_0} - \left(H^k b\right)^2_{r_0}\right) \\ &= \sum_{k=0}^{\infty}\left({H^*}^k b^*\right)_{r_0} - \sum_{k=0}^{\infty}\left(H^k b\right)^2_{r_0} \end{aligned}$$

Since $\rho(H) < 1$, Lemma 3.2 implies the second term $\sum_{k=0}^{\infty}\left(H^k b\right)^2_{r_0}$ is bounded. Therefore, whether $Var(\sum_{k=0}^{\infty} X(\gamma_k))$ is bounded depends solely on the first term, $\sum_{k=0}^{\infty}\left({H^*}^k b^*\right)_{r_0}$, which is bounded if and only if $\rho(H^*) < 1$. In conclusion, $\rho(H^*) < 1$ is the necessary and sufficient condition for convergence of the Monte Carlo linear solver using Ulam-von Neumann algorithm. ☐

The derived necessary and sufficient condition clarifies the confusions on the convergence of the Ulam–von Neumann Monte Carlo linear solver [153]. Fig. 7 summarizes the relationship between matrix H and the convergence of the Monte Carlo linear solver using Ulam-von Neumann algorithm below,

(1) The convergence of Neumann series is not a sufficient condition for the convergence of Monte Carlo.

(2) The transition matrix P plays an important role. An improper selection of transition matrix may result in divergence even though the condition $||H|| < 1$ holds.

(3) If $||H|| < 1$ is satisfied, there always exist certain transition matrices that guarantee convergence of the Monte Carlo linear solver. These transition matrices are trivial to find.

(4) The Monte Carlo linear solver may or may not converge if $||H|| < 1$ and $\rho(H) < 1$. If $\sum_{j=1}^{N}|H_{ij}| > 1$ for every row i in H or, more generally, $\rho(H^+) > 1$ where H^+ is a nonnegative matrix that $H_{ij}^+ = |H_{ij}|$, the Monte Carlo linear solver cannot converge, regardless how transition matrix P is selected.

(5) The sufficient and necessary condition for the Monte Carlo linear solver to converge is $\rho(H^*) < 1$, where $H_{ij}^* = {H_{ij}}^2/P_{ij}$ given H and a transition matrix P.

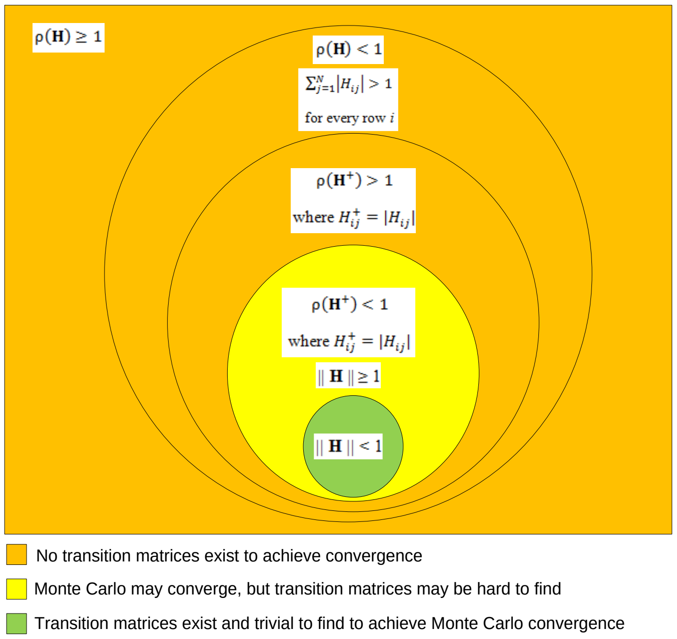

$\rho(\mathbf{H}) \geq 1$
$\rho(\mathbf{H}) < 1$
$\sum_{j=1}^{N}|H_{ij}| > 1$ for every row i
$\rho(\mathbf{H}^+) > 1$ where $H_{ij}^+ = |H_{ij}|$
$\rho(\mathbf{H}^+) < 1$ where $H_{ij}^+ = |H_{ij}|$
$||\mathbf{H}|| \geq 1$
$||\mathbf{H}|| < 1$
No transition matrices exist to achieve convergence
Monte Carlo may converge, but transition matrices may be hard to find
Transition matrices exist and trivial to find to achieve Monte Carlo convergence

Fig. 7. Summary of relationship between matrix H and convergence in Monte Carlo linear solver using Ulam-von Neumann algorithm

3.1.3 Limitations of Conventional Monte Carlo Solvers

The fundamental mechanism of conventional Monte Carlo solvers is constructing Markov chains based on random walks to estimate the underlying Neumann series to evaluate solutions of the linear systems. Therefore, provided that the random walks are based on Markov chains and the estimation is for the Neumann series, our convergence analysis in this section is applicable to the other conventional Monte Carlo solvers.

In practice, the general form of a linear system $Ax = b$ is often considered, instead of the form $x = Hx + c$. When applying conventional Monte Carlo solvers to the linear system $Ax = b$, it may face the following difficulties,

(1) Unless A is diagonally dominant, not all general $Ax = b$ can be easily recast into $x = Hx + c$ with $||H|| < 1$ to guarantee that the Monte Carlo solvers converge.

(2) In the case of $||H|| \geq 1$, finding a transition matrix P becomes a constraints satisfaction problem defined as follows:

Variables: $\{P_{ij} |\ i = 1 \ldots N, j = 1 \ldots N\}$;

Domain: [0,1];

Constraints: $P_{ij} \geq 0; \sum_j P_{ij} \leq 1\,; H_{ij} \neq 0 \rightarrow P_{ij} \neq 0;\ \rho(H^*) < 1$.

Unfortunately, solving this constraint satisfaction problem can be at least as hard as solving the original problem of $x = Hx + c$. More seriously, for in the case of $\rho(H^+) > 1$ where H^+ is a nonnegative matrix that $H_{ij}^+ = |H_{ij}|$, there exists no transition matrix P to make the Monte Carlo linear solvers converge.

(3) The convergence rate of the conventional Monte Carlo is dominated by $||\ H\ ||$. In the case that $||\ H\ ||$ is close to 1.0, the convergence of the underlying Neumann series is quite slow.

Therefore, due to the restricted convergence conditions, the applicability of conventional Monte Carlo solvers using Neumann series to general large-scale systems of linear equations is severely limited.

If the convergence condition of Monte Carlo linear solvers can be loosened, a much wider collection of matrices can be solved by Monte Carlo linear solvers.

3.2 Breakdown-Free Block Conjugate Gradient (BFBCG) Algorithm

Our analysis on the classical Monte Carlo linear solver using Ulam-von Neumann algorithm indicates its limitation in convergence condition as well as convergence speed in solving general linear systems. Here, rather than sampling the Neumann series, we focus on developing new Monte Carlo method to sample Krylov subspace to approximate the solution to the linear system.

3.2.1 Sampling Krylov Subspace

To sample the underlying Krylov subspace of a linear system $Ax = b$ in an efficient way, we convert the linear system into a block form by appending the right-hand side vector b and a Gaussian matrix Ω, such as

$$AX = B$$

where $B = [b, \Omega]$ is a block matrix containing s ($s \geq 1$) multiple right-hand sides. Fig. 8 illustrates the procedure of converting the original system to a block form.

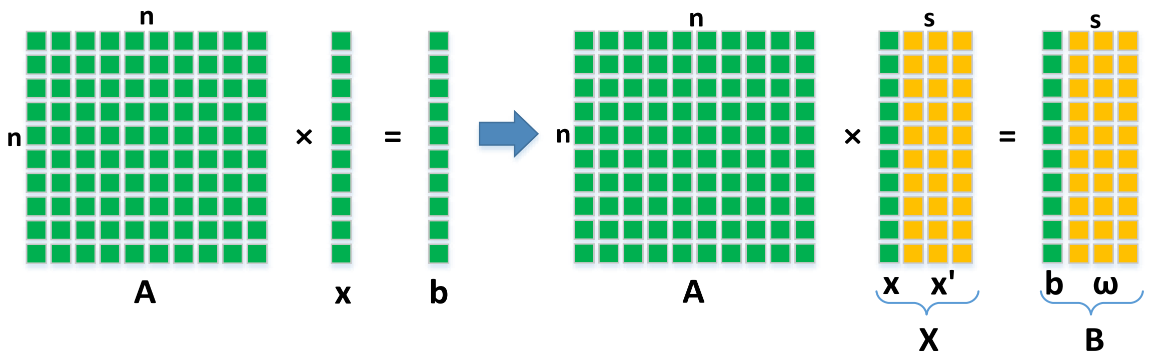


Fig. 8. Expanding a single right-hand side to multiple right-hand sides by supplying Gaussian random vectors

The columns of matrix B are expected to be statistically independent vectors, which can explore the Krylov subspace in a block manner, such that

1st random direction: $r_0^{(0)}, Ar_0^{(0)}, A^2 r_0^{(0)}, A^3 r_0^{(0)}, \ldots$

2nd random direction: $r_0^{(1)}, Ar_0^{(1)}, A^2 r_0^{(1)}, A^3 r_0^{(1)}, \ldots$

...

sth random direction: $r_0^{(s-1)}, Ar_0^{(s-1)}, A^2 r_0^{(s-1)}, A^3 r_0^{(s-1)}, \ldots$

where $r_0^{(i)}$ denotes the ith initial residual direction. In fact, using block Krylov subspace has many attractive features,

(1) A block formulation can potentially accelerate convergence and reduce the total number of passes over A, which is particularly favorable in handling large-scale matrices in which a pass over all elements in A is costly.

(2) Block matrix computations can lead to computational efficiency [59, 60, 61] for linear systems involving very large coefficient matrices. If $s \ll n$, the block methods involve a lot of multiplication operations on "tall-and-skinny" matrices, which can be easily parallelized with Level 3 BLAS subroutines [62, 63, 64].

(3) Solutions corresponding to multiple right-hand sides can be evaluated simultaneously. This is particularly useful for applications such as multi-objective optimization [65] being interested in finding solutions with respect to different right-hand side vectors.

(4) When the right-hand sides are augmented with Gaussian random vectors, extreme eigenvalues/eigenvectors of the coefficient matrix can be rapidly approximated via Monte Carlo sampling. These approximate eigenvectors can later be used in the deflation process to further accelerate convergence speed of the solvers.

3.2.2 BCG and Rank Deficiency

Despite the attractive features, a well-known practical issue of the blocking scheme is the rank deficiency problem that can lead to block methods breakdown. More specifically, in constructing block Krylov subspace, inverting block matrices is often needed to evaluate multiple right-hand sides simultaneously. During the iterations, some of these block matrices may lose rank. Consequently, inverting a block matrix with rank defect is one of the roots of the breakdown problem in block-type Krylov subspace methods. As a result, breakdown becomes a major cause of numerical instability in almost every block Krylov subspace method [66, 67, 68, 69, 70]. Although certain work in the literature [71] indicates that breakdown usually happens with a very small probability in practice, breakdown, if it actually occurs, may seriously hurt the computational performance. For mission-critical applications, this is particularly unfavorable.

We use the original Block Conjugate Gradient (BCG) algorithm by O'Leary [69] as an example to illustrate the rank deficiency problem.

Algorithm 3.1: Original Block Conjugate Gradient (BCG) Algorithm

Input: matrix $A \in \mathbb{R}^{n\times n}$, matrix $B \in \mathbb{R}^{n\times s}$, initial guess $X_0 \in \mathbb{R}^{n\times s}$, preconditioner $M \in \mathbb{R}^{n\times n}$, tolerance $tol \in \mathbb{R}$, and maximum number of iterations $maxit \in \mathbb{R}$

Output: an approximate solution $X_{sol} \in \mathbb{R}^{n\times s}$

```
$R_0 = B - AX_0$
$Z_0 = MR_0$
$P_0 = Z_0\gamma_0$
for $i = 0, \dots, maxit$
        $\alpha_i = \left(P_i^T AP_i\right)^{-1} \gamma_i^T \left(Z_i^T R_i\right)$
        $X_{i+1} = X_i + P_i\alpha_i$
        $R_{i+1} = R_i - AP_i\alpha_i$
        if converged, then stop.
        $Z_{i+1} = MR_{i+1}$
        $\beta_i = \gamma_i^{-1}\left(Z_i^T R_i\right)^{-1}\left(Z_{i+1}^T R_{i+1}\right)$
        $P_{i+1} = (Z_{i+1} + P_i\beta_i)\gamma_{i+1}$
end
$X_{sol} = X_{i+1}$
```

As shown in Algorithm 3.1, X_0 is the initial solution guess and M is a symmetric and positive definite (SPD) preconditioner. P_i denotes the search directions. α_i and β_i are $s \times s$ parameter matrices to ensure orthogonality of R_{i+1} and P_i as well as conjugacy (A-orthogonality) of $P_0, \dots, P_{i+1}$, respectively. γ_i is an arbitrary non-singular $s \times s$ matrix, which in practice is selected, for example, to orthogonalize P_i to decrease round-off errors and to enhance numerical stability [69].

Proposition 3.4 states that the preconditioned residual matrix Z_i and the search matrix P_i have the same matrix rank as the residual block R_i. Therefore, loss of full rank in R_i will lead to rank deficiency of Z_i and P_i during BCG iterations. Consequently, ${Z_i}^T R_i$ and ${P_i}^T A P_i$ become singular and thus it is improbable to obtain $\left({Z_i}^T R_i\right)^{-1}$and $\left({P_i}^T A P_i\right)^{-1}$ to evaluate α_i and β_i. As a result, BCG breakdown occurs.

Proposition 3.4. Suppose R_i is an $n \times s$ residual matrix of rank r_i ($r_i \leq s$) at the ith iteration, then

$$rank(P_i) = rank(Z_i) = rank(R_i) = r_i,$$

where $rank(\cdot)$ denotes the rank of a matrix.

Proof. First, we show that $rank(Z_i) = rank(R_i) = r_i$. From Algorithm 3.1, matrix Z_i is defined as $Z_i = MR_i$. Since M is assumed to be SPD, then $rank(Z_i) = rank(R_i) = r_i$.

Next we show that $rank(P_i) = rank(R_i)$. The search matrix P_i is given by

$$P_i = (Z_i + P_{i-1}\beta_{i-1})\gamma_i. \tag{1}$$

Left multiplying (1) by $P_i^T A$ on both sides, we get

$$P_i^T A P_i = P_i^T A Z_i \gamma_i + P_i^T A P_{i-1} \beta_{i-1} \gamma_i.$$

Notice that columns in P_i are A-orthogonal to P_{i-1}, i.e., $P_i^T A P_{i-1} = 0$, then

$$P_i^T A P_i = P_{\mathrm{i}}^T A Z_i \gamma_i.$$

Using the basic properties of matrix rank, we can get

$$\begin{aligned} rank(P_i) &= rank\left(P_i^T A P_i\right) \\ &= rank\left(P_i^T A Z_i \gamma_i\right) \\ &\leq rank(Z_i) \\ &= rank(R_i). \end{aligned} \tag{2}$$

On the other hand, since columns in R_i are orthogonal to P_{i-1}, i.e., $R_i^T P_{i-1} = 0$, left multiplying both sides of (1) by R_i^T and eliminating the zero terms, we obtain

$$R_i^T P_i = R_i^T Z_i \gamma_i = R_i^T M R_i \gamma_i.$$

According to the basic properties of matrix rank again, we have

$$\begin{aligned} rank(P_i) &\geq rank\left(R_i^T P_i\right) \\ &= rank\left(R_i^T M R_i \gamma_i\right) \\ &= rank(R_i). \end{aligned} \qquad (3)$$

Based on (2) and (3), $rank(P_i) = rank(R_i) = r_i$ is concluded. □

In practice, rank deficiency may be caused by many different reasons, for instances, inappropriate guess of initial vectors, unbalanced convergence speeds of solutions with respect to multiple right-hand sides, and accumulation of round-off errors. The possible situations of rank deficiency in BCG are summarized as follows.

(1) Two or more vector components in the initial block residue R_0 are linearly dependent. For example, if the multiple right-hand sides in matrix B contain linearly dependent vectors and X_0 simply takes zero vectors as the initial guess, then the initial block residue R_0 will include linearly dependent vectors. In practice, this breakdown situation can be eliminated by ensuring the linear independence of column vectors in R_0, such as carefully selecting initial guess X_0. An alternative approach is orthogonalizing R_0 [72] to eliminate the dependent vectors in R_0.

(2) Convergence of one or more vector components in the block residue R_i . During BCG iterations, solutions with respect to some right-hand sides may converge faster than the others, which results in near zero vectors in R_i. This typically happens when the norms of the component vectors in R_0 are significantly different in magnitude. An obvious approach is to normalize the right-hand sides in B so as to keep the norms of the component vectors of R_0 at a similar scale [71] to hopefully balance the number of convergence steps for the multiple right-hand sides. Since convergence has already been achieved in some solutions, removing these solutions and their corresponding residual vectors [69] not only avoids BCG breakdown, but also eliminates unnecessary numerical computations.

(3) Two or more vector components in the block residue R_i at the ith iteration become linearly dependent. If one is only interested in a single solution with respect to a specific right-hand side, for

example, the multiple right-hand sides block expanded from a single right-hand side, the variable BCG algorithm [68] by constructing an A-orthogonal projector to reduce the block size can sufficiently address the breakdown problem caused by this factor. Nevertheless, if solutions to all right-hand sides are of interest, assuming that the right-hand sides of the corresponding linearly dependent vectors have not converged yet and thus none of the vector components in R_i are zero, reducing the block sizes will result in loss of solutions.

3.2.3 The BFBCG Algorithm

We present a simple solution to address the rank deficiency problem in BCG, which results in a Breakdown Free Block Conjugate Gradient (BFBCG) algorithm (Algorithm 3.2). The fundamental idea of BFBCG is, in case of the rank of the block search direction vectors being reduced, the parameter matrices are calculated in the reduced Krylov subspace to minimize the block nonnegative quadratic function of

$$F(X) = trace\left((X - X^*)^T A(X - X^*)\right),$$

where $trace(\cdot)$ is the trace of a matrix and $X^* = A^{-1}B$ is the desired block solution. As a result, BFBCG avoids estimation of the inverse of a potentially non-full rank matrix and thus addresses the rank deficiency problem.

Algorithm 3.2: Breakdown-Free BCG (BFBCG) Algorithm

Input: matrix $A \in \mathbb{R}^{n\times n}$, right hand side matrix $B \in \mathbb{R}^{n\times s}$, initial guess $X_0 \in \mathbb{R}^{n\times s}$, preconditioner $M \in \mathbb{R}^{n\times n}$, tolerance $tol \in \mathbb{R}$, and maximum number of iterations $maxit \in \mathbb{R}$
Output: an approximate solution $X_{sol} \in \mathbb{R}^{n\times s}$

```
$R_0 = B - AX_0$
$Z_0 = MR_0$
$\tilde{P}_0 = orth(Z_0)$
for $i = 0, \dots, maxit$
    $Q_i = A\tilde{P}_i$
    $\tilde{\alpha}_i = \left(\tilde{P}_i^T Q_i\right)^{-1}\left(\tilde{P}_i^T R_i\right)$
    $X_{i+1} = X_i + \tilde{P}_i\tilde{\alpha}_i$
    $R_{i+1} = R_i - Q_i\tilde{\alpha}_i$
    if converged, then stop.
    $Z_{i+1} = MR_{i+1}$
```

```
        $\tilde{\beta}_i = -\left(\tilde{P}_i^{\,T} Q_i\right)^{-1} \left(Q_i^{\,T} Z_{i+1}\right)$
        $\tilde{P}_{i+1} = orth\left(Z_{i+1} + \tilde{P}_i \tilde{\beta}_i\right)$
end
$X_{sol} = X_{i+1}$
```

To illustrate the differences in comparison with the BCG algorithm described in Algorithm 3.1, the matrix symbols with a "~" notation are used to indicate that the dimension of these matrices may reduce in case of rank deficiency in BFBCG. New forms of calculating parameter matrices $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ are derived based on potentially reduced search subspace. In case of lost rank in search directions or residual vectors, $\tilde{\alpha}_i$ is designed to ensure that the next residual vectors R_{i+1} are orthogonal to search space $\mathcal{P}_i$. A new form of $\tilde{\beta}_i$ is derived so that the new search space $\mathcal{P}_{i+1}$ is conjugate to all previous search spaces $\mathcal{P}_j$ $(j < i + 1)$.

Compared to the original BCG algorithm [69], our BFBCG algorithm has the following major differences:

(1) Matrix operation $orth(\cdot)$ is employed for extracting an orthogonal basis $\tilde{P}_i \in \mathbb{R}^{n\times r_i}$ from the search space $\mathcal{P}_i$. $orth(\cdot)$ can be efficiently implemented using QR decomposition with column pivoting. In case of rank deficiency, the dimension of the search space $\mathcal{P}_i$ will be reduced, which avoids the situations of revisiting the subspace already visited in the BCGAdQ algorithm described in [73].

(2) If rank deficiency occurs at the ith iteration, $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ turn into rectangular matrices of size $r_i \times s$, where r_i is the dimension of search space $\mathcal{P}_i$ at the ith iteration, while they are restricted as square matrices in BCG.

(3) Matrices γ_i are no longer necessary in the BFBCG algorithm.

In addition to breakdown avoidance, the BFBCG algorithm maintains several favorable features in practice. For example, at each iteration, matrix A is visited only once. Meanwhile, $\left(\tilde{P}_i^{\,T} Q_i\right)^{-1}$ calculated in $\tilde{\alpha}_i$ can be reused for computing $\tilde{\beta}_i$.

We use Theorems 3.5 and 3.10 to justify the derivation of $\tilde{\alpha}_i$ and $\tilde{\beta}_i$, respectively. Theorem 3.5 shows, in case of rank deficiency at the ith iteration in BFBCG, the rectangular parameter matrix $\tilde{\alpha}_i$

ensures that R_{i+1} is orthogonal to the search space $\mathcal{P}_i$.

Theorem 3.5. Suppose R_i loses full rank at the ith iteration. Let $\mathcal{P}_i$ denote the corresponding search space with dimension r_i ($r_i < s$). Given matrix $\tilde{\alpha}_i \in \mathbb{R}^{r_i \times s}$ so that

$$\tilde{\alpha}_i = \left(\tilde{P}_i{}^T Q_i\right)^{-1} \left(\tilde{P}_i{}^T R_i\right),$$

where $\tilde{P}_i \in \mathbb{R}^{n \times r_i}$ consists of orthonormal basis of $\mathcal{P}_i$ and $Q_i \in \mathbb{R}^{n \times r_i}$ denotes the matrix product $A\tilde{P}_i$, the next residual matrix R_{i+1} derived from $\tilde{\alpha}_i$ is orthogonal to the search space $\mathcal{P}_i$.

Proof. As $\tilde{P}_i \in \mathbb{R}^{n \times r_i}$ is the orthonormal basis of the search space $\mathcal{P}_i$ and $Q_i = A\tilde{P}_i$, $\tilde{P}_i{}^T Q_i \in \mathbb{R}^{r_i \times r_i}$ is nonsingular. Therefore, there exists a matrix $\tilde{\alpha}_i \in \mathbb{R}^{r_i \times s}$ such that

$$\tilde{\alpha}_i = \left(\tilde{P}_i{}^T Q_i\right)^{-1} \left(\tilde{P}_i{}^T R_i\right). \tag{4}$$

Since R_{i+1} is constructed from

$$R_{i+1} = R_i - Q_i \tilde{\alpha}_i, \tag{5}$$

in BFBCG, left multiplying (5) by $\tilde{P}_i{}^T$, and then by definition of $\tilde{\alpha}_i$ in (4), we can get

$$\begin{aligned}
\tilde{P}_i{}^T R_{i+1} &= \tilde{P}_i{}^T R_i - \tilde{P}_i{}^T Q_i \tilde{\alpha}_i \\
&= \tilde{P}_i{}^T R_i - \tilde{P}_i{}^T Q_i \left(\tilde{P}_i{}^T Q_i\right)^{-1} \tilde{P}_i{}^T R_i \\
&= 0,
\end{aligned}$$

which indicates that the derived R_{i+1} is orthogonal to the search space $\mathcal{P}_i$. □

Based on Theorem 3.5, other orthogonality properties of BFBCG can be obtained easily, which are summarized as the following two corollaries. Corollary 3.6 extends Theorem 3.5 and shows that R_{i+1} is not only orthogonal to search space $\mathcal{P}_i$ at the ith iteration, but to all previous search spaces $\mathcal{P}_j$ ($j < i + 1$). Moreover, observing that search spaces $\mathcal{P}_j$ are derived from subspaces spanned by residual matrices R_j ($j < i + 1$), Corollary 3.7 states that R_{i+1} is M-orthogonal to all previous residual matrices under preconditioning matrix M (assuming that M is symmetric positive definite).

Corollary 3.6. ${R_{i+1}}^T\tilde{P}_j = 0$ for all $j < i + 1$.

Corollary 3.7. ${R_{i+1}}^T M R_j = {Z_{i+1}}^T R_j = 0$ for all $j < i + 1$.

At the ith iteration, BFBCG explores the block Krylov subspace [67, 69] defined as

$$D_i(A, M, R_0) = block\_span\{MR_0, MAMR_0, \dots, (MA)^i MR_0\}$$

$$= \left\{ \sum_{j=0}^{i} (MA)^j MR_0 \Psi_j \,;\, \Psi_j \in \mathbb{R}^{s\times s} \right\}$$

which is the union of the previous subspaces spanned by the matrices MR_j $(j < i + 1)$. By Corollary 3.7, $R_{i+1}$is orthogonal to the Krylov subspace explored before as well, which implies that X_{i+1} from BFBCG is the minimizer of the block nonnegative quadratic function of

$$F(X) = trace\big((X - X^*)^T A(X - X^*)\big)$$

over the Krylov subspace $X_0 + span\{MR_0, MAMR_0, \dots, (MA)^i MR_0\}$ at the ith iteration, where $X^* = A^{-1}B$ is the desired block solution.

The other parameter matrix $\tilde{\beta}_i$ in BFBCG is chosen to ensure that the next search space $\mathcal{P}_{i+1}$ is conjugate to the previous search space $\mathcal{P}_j$ $(j < i + 1)$ in case of rank deficiency, which is shown in Theorem 3.10. The following Lemmas 3.8 and 3.9 will be used for the proof of Theorem 3.10. The proofs for Lemmas 3.8 and 3.9 are included in Appendix A.

Lemma 3.8. Suppose R_i is an $n \times s$ residual matrix of rank r_i $(r_i \leq s)$ at the ith iteration, then $rank\big(\tilde{P}_i^T R_i\big) = r_i$.

Lemma 3.9. Z_{i+1} is conjugate to search spaces $\mathcal{P}_j$ where $j < i$.

Lemma 3.8 indicates that the matrix rank of $\tilde{P}_i^T R_i$ is always equal to that of R_i. We can also learn from Lemma 3.8 that the parameter matrix $\tilde{\alpha}_i$ has rank $r_i$which is consistent with the rank of R_i at every iteration step i. In other words, $\tilde{\alpha}_i$ will not be a zero matrix unless R_i is a zero matrix. This fundamentally prevents BFBCG from suffering the potential stagnation problem occurred in Krylov subspace methods [74, 75], where the solution matrices in two (and further) consecutive iterations will not be updated due to zero parameter matrix while convergence has not been reached yet.

Lemma 3.9 indicates that Z_{i+1} from BFBCG is conjugate to all previous search spaces $\mathcal{P}_j$ $(j < i)$ except $\mathcal{P}_i$. This inspires us to derive a parameter matrix $\tilde{\beta}_i$ to construct a new search space $\mathcal{P}_{i+1}$ from Z_{i+1} by removing the conjugation part of $\mathcal{P}_i$. Theorem 3.10 shows that, in case of rank deficiency occurring at the ith iteration, the rectangular parameter matrix $\tilde{\beta}_i$ ensures that the new search space $\mathcal{P}_{i+1}$ is conjugate to all previous search spaces $\mathcal{P}_j$ $(j < i + 1)$.

Theorem 3.10. Suppose R_i loses full rank at the ith iteration. Let $\mathcal{P}_i$ denote the corresponding search space with dimension r_i $(r_i < s)$. Given matrix $\tilde{\beta}_i \in \mathbb{R}^{r_i \times s}$ so that

$$\tilde{\beta}_i = -\left(\tilde{P}_i^{\,T} Q_i\right)^{-1} Q_i^{\,T} Z_{i+1},$$

where $\tilde{P}_i \in \mathbb{R}^{n \times r_i}$ consists of orthonormal basis for $\mathcal{P}_i$ and $Q_i \in \mathbb{R}^{n \times r_i}$ denotes the matrices product $A\tilde{P}_i$. Then, the new search space $\mathcal{P}_{i+1}$ obtained from $\tilde{\beta}_i$ is conjugate to all previous search spaces $\mathcal{P}_j$ where $j < i + 1$.

Proof. Based on Gram-Schmidt conjugation process, the new search directions P_{i+1} at ith iteration can be generated by

$$P_{i+1} = Z_{i+1} + \sum_{j=0}^{i} \tilde{P}_j \beta_{i+1,j},$$

where $\tilde{P}_j$ is the orthonormal basis of $\mathcal{P}_j$ and $\beta_{i+1,j}$ is the associated weight matrix of $\tilde{P}_j$. As $\tilde{P}_i^{\,T} Q_i \in \mathbb{R}^{r_i \times r_i}$ is nonsingular, by selecting $\beta_{i+1,j} = 0$ for all $j < i$ and $\tilde{\beta}_i = \beta_{i+1,i} = -\left(\tilde{P}_i^{\,T} Q_i\right)^{-1} Q_i^{\,T} Z_{i+1}$, it is easy to show that

(1) for any $\tilde{P}_j$ where $j < i$, according to Lemma 3.9,

$$\tilde{P}_j^{\,T} A P_{i+1} = \tilde{P}_j^{\,T} A Z_{i+1} + \tilde{P}_j^{\,T} A \sum_{k=0}^{j} \tilde{P}_k \beta_{i+1,k}$$

$$= \tilde{P}_j^{\,T} A Z_{i+1} = 0.$$

(2) for $\tilde{P}_j$ where $j = i$,

$$\tilde{P}_i^{\,T} A P_{i+1} = \tilde{P}_i^{\,T} A Z_{i+1} + \tilde{P}_i^{\,T} A \sum_{k=0}^{i} \tilde{P}_k \beta_{i+1,k}$$

$$= \tilde{P}_i^{\,T} A Z_{i+1} + \tilde{P}_i^{\,T} A \tilde{P}_i \beta_{i+1,i}$$

$$= \tilde{P}_i^{\,T} A Z_{i+1} - \tilde{P}_i^{\,T} A \tilde{P}_i \left(\tilde{P}_i^{\,T} A \tilde{P}_i\right)^{-1} \tilde{P}_i^{\,T} A Z_{i+1}$$

$$= \tilde{P}_i^{\,T} A Z_{i+1} - \tilde{P}_i^{\,T} A Z_{i+1} = 0.$$

Let the range of P_{i+1} be the new search space $\mathcal{P}_{i+1}$ and then the new search space $\mathcal{P}_{i+1}$ is conjugate to all previous search spaces $\mathcal{P}_j$ $(j < i + 1)$. □

In fact, $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ defined in BFBCG are generalized forms of the parameter matrices α_i and β_i in BCG algorithms to avoid breakdown during BFBCG iterations. When R_i's have full column rank, the BFBCG algorithm is equivalent to the original BCG. In particular, $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ are square matrices with full rank that are coincide with α_i and β_i from the original BCG where γ_i in Algorithm 3.1 is replaced by the inverse of upper triangular part of QR decomposition, while a simplified form $\tilde{\beta}_i$ is chosen in BFBCG to avoid augmented condition number of ${Z_i}^T R_i$. On the other hand, if R_i loses full rank during BFBCG iterations, the rectangular parameter matrices $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ are employed to maintain orthogonality properties in block Krylov subspace and avoid breakdown due to the rank deficiency problem.

3.2.4 Convergence Analysis

We investigate the theoretical number of iterations of BFBCG. Then, the convergence rate of BCGLS is further estimated.

3.2.4.1 Number of Iterations

To solve a linear system with s multiple right hand sides using BCG, the block Krylov subspace

$$D_i(A, M, R_0) = block\_span\{MR_0, MAMR_0, \dots, (MA)^i MR_0\}$$
$$= \left\{ \sum_{j=0}^{i} (MA)^j MR_0 \Psi_j \, ; \, \Psi_j \in \mathbb{R}^{s \times s} \right\}$$

is constructed to find an approximate X_{i+1} at next iteration, where M is an SPD preconditioner. As pointed out by [76], if the effect of roundoff errors can be ignored, the BCG algorithm is able to find the exact solutions after at most $\lceil n/s \rceil$ iterations, where s is the number of right-hand sides.

As a generalized form of BCG, BFBCG shares the same convergence property only if the residual matrix remains full rank s during all iterations. When rank deficiency occurs, BFBCG continues to explore the Krylov subspaces from the reduced search spaces. Proposition 3.11 shows that once a residual matrix loses full rank, rank deficiency will be inherited in the subsequent residual matrices.

Proposition 3.11. If residual matrix R_i loses full column rank at the ith iteration, the subsequent residual matrices R_j $(j > i)$ are also rank deficient.

Proof. Since

$$R_{i+1} = R_i - A\tilde{P}_i \tilde{\alpha}_i$$
$$= R_i - A\tilde{P}_i \left(\tilde{P}_i^T A \tilde{P}_i\right)^{-1} \tilde{P}_i^T R_i$$
$$= \left(I - A\tilde{P}_i \left(\tilde{P}_i^T A \tilde{P}_i\right)^{-1} \tilde{P}_i^T\right) R_i,$$

Then, $rank(R_i) \geq rank(R_{i+1})$ can be obtained based on the properties of matrix rank. For $j > i$, $rank(R_i) \geq rank(R_j)$ can be derived in a similar way. □

In the case that rank deficiency occurs, the Krylov subspace can no longer be expanded by s dimensions in future iterations. Instead, the dimension of the corresponding Krylov subspace increases by the rank of the residual matrix, which is less than the number of right hand side s, at each subsequent iteration step. Consequently, in general, more than $\lceil n/s \rceil$ iterations are needed in BFBCG to find the solutions in case of rank deficiency.

3.2.4.2 Convergence Rate

Defining the error matrix E_{i+1} as

$$E_{i+1} = \left[e_{i+1}^{(0)}, e_{i+1}^{(1)}, \dots, e_{i+1}^{(s-1)}\right] = X_{i+1} - X^*$$

at the ith iteration, where $e_{i+1}^{(k)}$ be the kth column of E_{i+1} and $X^* = A^{-1}B$ is the desired block solution, the block nonnegative quadratic function can be represented as

$$trace\big((X_{i+1} - X^*)^T A (X_{i+1} - X^*)\big) = \sum_{k=0}^{s-1} \left\| e_{i+1}^{(k)} \right\|_A^2.$$

To determine the convergence rate of BCG, the initial residual matrix $R_0 = B - AX_0$ plays an important role in bounding the errors at each iteration step. Under the assumption that R_0 has full column rank, O'Leary [69] showed that the minimum error square norm $\left\| e_{i+1}^{(k)} \right\|_A^2$, $(0 \le k \le s-1)$ is bounded as

$$\left\| e_{i+1}^{(k)} \right\|_A^2 \le c^{(k)} \left(\frac{1 - \sqrt{\kappa^{-1}}}{1 + \sqrt{\kappa^{-1}}} \right)^{2(i+1)}$$

at each iteration. Here $\kappa = \lambda_n / \lambda_s$ where $\lambda_n \ge \lambda_{n-1} \ge \cdots \ge \lambda_1$ are the eigenvalues of MA, respectively, and $c^{(k)}$ is a constant only related to $e_0^{(k)}$. Nevertheless, if R_0 does not have full rank, the above error bound does not hold. Assuming that R_0 has rank r_0, Theorem 3.12 shows that the convergence rate of BFBCG method is bounded by $\left(\frac{1 - \sqrt{\kappa'^{-1}}}{1 + \sqrt{\kappa'^{-1}}} \right)^2$ where $\kappa' = \lambda_n / \lambda_{r_0}$.

Theorem 3.12. Suppose R_0 is rank deficient with rank r_0 $(r_0 < s)$, the minimum error square norm $\left\| e_{i+1}^{(k)} \right\|_A^2$ is bounded as

$$\left\|e_{i+1}^{(k)}\right\|_A^2 \le c\left(\frac{1-\sqrt{\kappa'^{-1}}}{1+\sqrt{\kappa'^{-1}}}\right)^{2(i+1)},$$

where c is a constant related only with E_0 and $\kappa' = \lambda_n/\lambda_{r_0}$.

Proof. Assuming that the $n \times s$ residual matrix R_0 has rank r_0, which is potentially rank deficient, then there exists a nonsingular $s \times s$ matrix δ such that

$$R_0 = ({R_0}', 0)\delta$$

where R_0 is an $n \times r_0$ matrix with full rank. Since $E_0 = A^{-1}R_0$ and $E_{i+1} = \phi_i(MA)E_0$, where $\phi_i(MA)$ is a polynomial of degree i, we have that $E_{i+1} = ({E_{i+1}}', 0)\delta$ and each column in E_{i+1} can be expressed as

$$e_{i+1}^{(k)} = \sum_{j=0}^{r_0-1} \delta_{jk}\, {e'}_{i+1}^{(j)}$$

where ${e'}_{i+1}^{(j)}$ is the jth column of ${E_{i+1}}'$. Hence, the error bound of the square norm $\left\|e_{i+1}^{(k)}\right\|_A^2$ becomes

$$\left\|e_{i+1}^{(k)}\right\|_A^2 = \left\|\sum_{j=0}^{r_0-1} \delta_{jk}{e'}_{i+1}^{(j)}\right\|_A^2$$

$$\le \sum_{j=0}^{r_0-1} \delta_{jk}^2 \left\|{e'}_{i+1}^{(j)}\right\|_A^2$$

$$\le \sum_{j=0}^{r_0-1} \delta_{jk}^2\, c^{(j)} \left(\frac{1-\sqrt{\kappa'^{-1}}}{1+\sqrt{\kappa'^{-1}}}\right)^{2(i+1)}$$

$$\le c\left(\frac{1-\sqrt{\kappa'^{-1}}}{1+\sqrt{\kappa'^{-1}}}\right)^{2(i+1)}$$

where $c = \sum_{j=0}^{r_0-1} \delta_{jk}^2\, c^{(j)}$, and $\kappa' = \lambda_n/\lambda_{r_0}$. □

In the case of rank deficiency, if R_i loses full rank to r_i, BCG has to restart with the reduced block size. Restart is unfavorable in parallel computing, where reinitiating processes and redistributing workload are necessary. More importantly, the restarting BCG uses the range of R_i as the initial search space and abandons all search spaces explored before. As a result, the restarted BCG has a lower convergence rate of $\left(\frac{1-\sqrt{\kappa''^{-1}}}{1+\sqrt{\kappa''^{-1}}}\right)^2$, where $\kappa'' = \lambda_n/\lambda_{r_i}$. In contrast, without restarting, BFBCG yields faster convergence than restarting BCG, because BFBCG still takes advantage of search space information previously constructed. Hence, the overall convergence rate of BFBCG lies between $\left(\frac{1-\sqrt{\kappa''^{-1}}}{1+\sqrt{\kappa''^{-1}}}\right)^2$ and $\left(\frac{1-\sqrt{\kappa^{-1}}}{1+\sqrt{\kappa^{-1}}}\right)^2$, where $\kappa'' = \lambda_n/\lambda_{r_i}$ and $\kappa = \lambda_n/\lambda_s$, respectively.

3.2.5 Numerical Results

3.2.5.1 Handling Rank Deficiency

We use a matrix “Kuu” from the UFL sparse matrix collection [77] as the coefficient matrix of a block linear system with 200 right-hand sides to demonstrate the effectiveness of BFBCG in addressing the breakdown problem with combined rank deficiency situations. “Kuu” is a $7{,}102 \times 7{,}102$ SPD matrix with $340{,}200$ nonzero elements arising from a structural problem whose sparse pattern is shown in Fig. 9. To construct linearly dependent vector components in the initial block residue R_0, we intentionally set the elements in the first 198 columns of the right-hand side matrix B as randomly generated numbers while the last two columns are created as linear combinations of the first 198 columns. The initial guess X_0 is

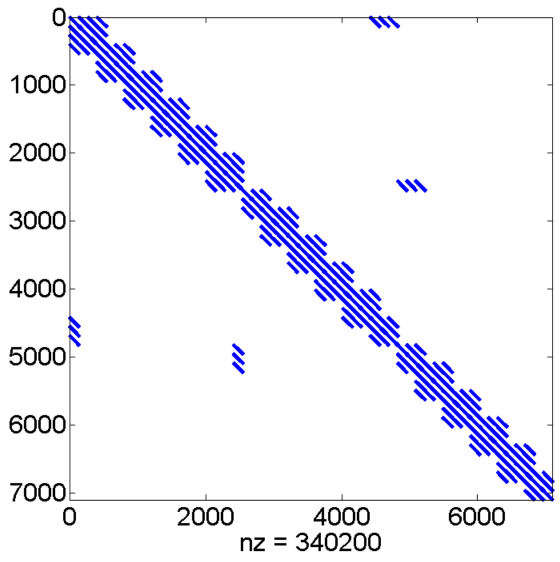


Fig. 9. Sparse pattern of matrix “Kuu”

set to be the same as B and a preconditioner M is constructed using the Crout version of ILU factorization [78] with the element drop tolerance "0.01". The desired error tolerance of all solutions is 10^{-7}.

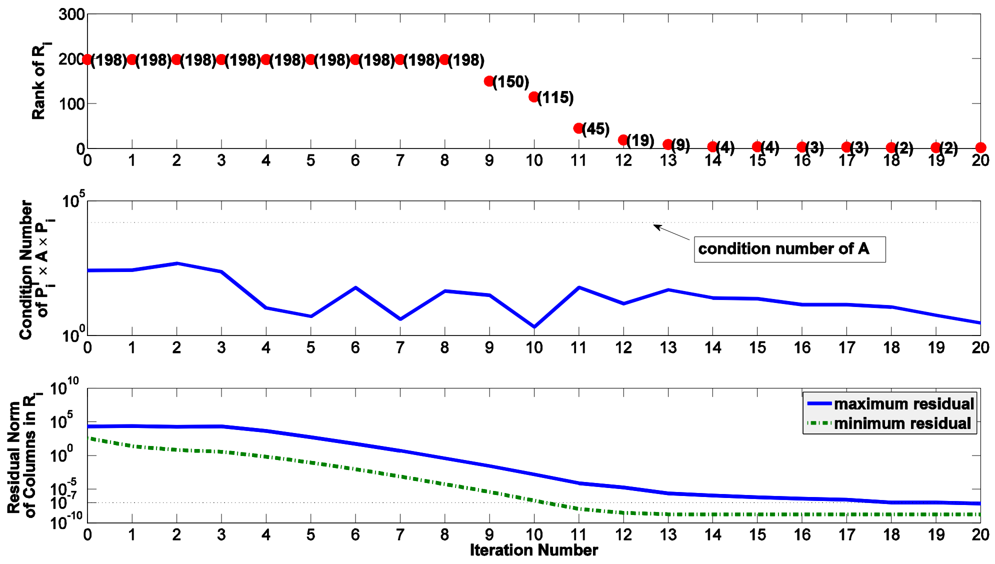


Fig. 10. Matrix rank of residue R_i, condition number of $P_i^T AP_i$, and the corresponding maximum and minimum residual norm for a block linear system with 200 right-hand sides using the Matrix "Kuu" as coefficient matrix along BFBCG iterations

Fig. 10 illustrates the change of matrix rank of residual matrix R_i (upper), condition number of $P_i^T AP_i$ (middle), as well as the maximum and minimum residual norms of columns in R_i (lower) along the BFBCG iterations. The condition number of $P_i^T AP_i$ is bounded by the condition number of A during the iterations. One can find that rank deficiency happens at the very beginning because of the linearly dependent vectors in B that we set intentionally. The rank of the residual matrix R_i starts to drop down to 150 when $i = 9$ because linear dependence occurs during the BFBCG process; however, none of the systems converge to the desired resolution yet. When $i = 11$, the residual norms in some columns in R_i are smaller than the given error tolerance indicating that some but not all systems have reached convergence. Correspondingly, the matrix rank of R_i decreases further to 45. After all, BFBCG is able to deal with the combination of various rank deficiency situations and continues to improve the solution accuracy based on the reduced subspace. Eventually, all systems reach convergence at the 20th iteration.

3.2.5.2 Handling the Near-breakdown Problem

Recent studies [79, 80] have shown that the almost linearly dependent vectors in the residual block matrices may cause loss of orthogonality during linear system solving iterations and thus slow down or even prevent convergence of the block Krylov subspace methods. This is referred to as the near-breakdown problem. We hereby investigate the impacts of the near-breakdown problem on BFBCG in comparison with the original BCG algorithm. To simulate the near-breakdown situations, we use a linear system of a 10×10 random coefficient matrix with a small condition number to eliminate the impact from the matrix itself and initialize a block residual matrix R_0 with two nearly linearly dependent vectors, where the second column is generated by multiplying the first one by 10 while adding small random perturbations. The coefficient matrix and the right-hand side block matrix are specified in Appendix B.

TABLE 2
Comparison between BCG and BFBCG in the Case of Near-Breakdown

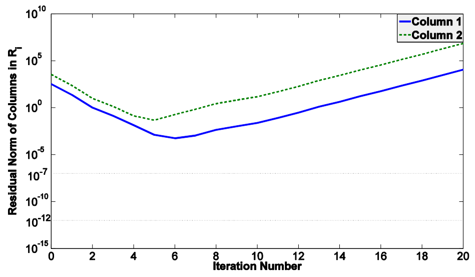
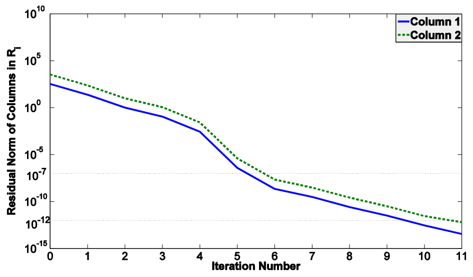
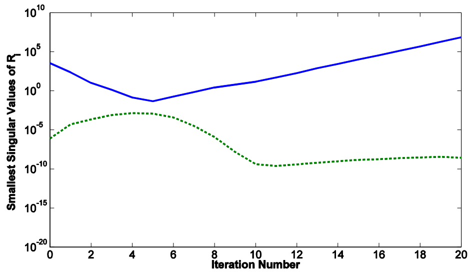
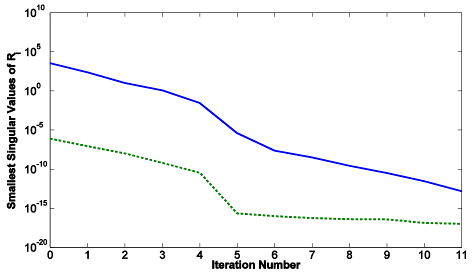
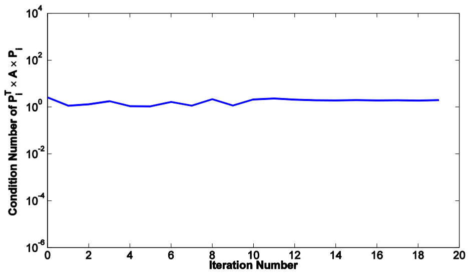
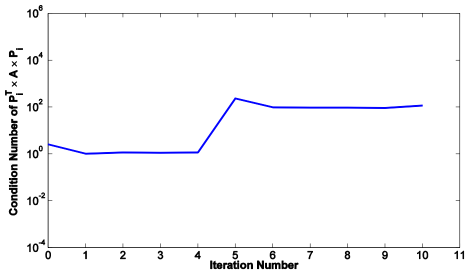
	BCG	BFBCG
Convergence		
Smallest singular values of R_i		
Condition number of $P_i^T A P_i$		

TABLE 2 Continued

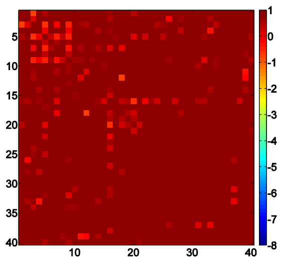
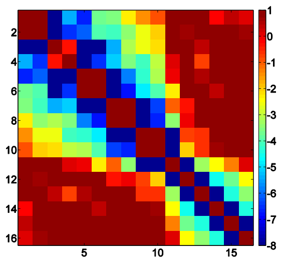
	BCG	BFBCG
Colormap of A-orthogonality between Search Matrices		

Table 2 compares BCG and BFBCG in the case when near-breakdown occurs. We monitor the smallest singular value of R_i and a parameter τ is designated as a tolerance threshold of linear dependence among the block residual vectors in BFBCG. Here, τ is set to 10^{-12}. One can find that the nearly linearly dependent vectors in R_i result in a certain loss of A-orthogonality among search matrices during the iterations in both BCG and BFBCG, which is consistent with the analysis in [79, 81]. This is due to the fact that constructing the new search matrices is sensitive to the round-off errors when the residual matrices are nearly rank-deficient. Nevertheless, the computation of the parameter matrix β_i in BCG requires an evaluation of $\gamma_i^{-1}\left(Z_i^T R_i\right)^{-1}$, where nearly linear dependence in R_i can lead to large round-off errors. As shown in Table 2, BCG suffers from complete loss of A-orthogonality and fails to converge. In contrast, the computation of $\tilde{\beta}_i$ in BFBCG relies only on calculating $\left(P_i^T A P_i\right)^{-1}$ and thus maintains relatively better A-orthogonality. Moreover, BFBCG is designed to enforce A-orthogonality of every two consecutive search matrices. As a result, BFBCG is able to evolve with nearly linear dependence in R_i. When some singular values of R_i fall under threshold τ, search matrices of reduced size are generated in such a way that A-orthogonality with the previous search directions is maintained. Consequently, the relatively better A-orthogonality allows BFBCG to reach solutions with desired precision.

3.2.5.3 Comparison with Restarting Scheme

When a breakdown actually occurs, the original BCG algorithm has to restart with a reduced block size. Table 3 compares the performance of BCG with restarting and BFBCG on a set of SPD matrices from structural engineering applications in the Harwell-Boeing sparse matrix collection [82].We use a right-hand side matrix B consisting of 10 random column vectors. Particularly, we scale the elements in the first 8 columns of matrix B by the matrix norm of A to amplify the magnitude difference among column vectors so that rank deficiency can easily occur. The Crout version of ILU preconditioners is applied. The computational experiments are carried out on the XSEDE TACC Stampede System [83]. When breakdown happens, causing the loss of all search spaces that have been explored before restarting, BCG typically takes more iteration steps to reach convergence than BFBCG. In contrast, BFBCG is able to continue to update the solution blocks from the reduced search spaces without being interrupted. Moreover, restarting requires additional operations to reinitiate the computational process, which results in significantly more computational time in BCG than that in BFBCG.

TABLE 3
Performance Comparison between BFBCG and Restarting BCG on SPD Matrices from Static Analyses in Structural Engineering Application

Name	Rows	Columns	Nonzeros	BCG with Restarting			BFBCG	
				# of Iterations	# of Restarts	Computational Time (s)	# of Iterations	Computational Time (s)
BCSSTK14	1,806	1,806	32,630	9	5	1.54	8	0.68
BCSSTK15	3,948	3,948	60,882	19	11	6.28	14	3.18
BCSSTK16	4,884	4,884	147,631	8	4	3.8	8	2.44
BCSSTK17	10,974	10,974	219,812	19	16	40.69	15	17.39
BCSSTK18	11,948	11,948	80,519	14	8	28.49	14	18.44

3.3 Block Conjugate Gradient for Least Square (BCGLS) Algorithm

The applicability of BFBCG is limited to linear systems with symmetric positive definite (SPD) coefficient matrices. In this section, we extend the breakdown-free techniques in BFBCG to more general linear systems, where a Block Conjugate Gradient for Least Square (BCGLS) algorithm [151] is developed to handle the least squares problem and general linear systems at a large scale.

3.3.1 The BCGLS Algorithm

Considering the least squares solutions to a linear system of equations with multiple right-hand sides $AX = B$, where A is an $m \times n$ $(m \geq n)$ sparse, rectangular or square matrix with rank n, X is an $n \times s$ unknown matrix, B is an $m \times s$ right-hand side matrix, and s $(s \geq 1)$ is the number of right-hand sides. The block matrices operations in BCGLS are developed to approximate the least squares solutions by ensuring orthogonality properties while minimizing the residual (error) function

$$Trace\big((B - AX)^T(B - AX)\big),$$

over the underlying Krylov subspace, where $Trace(\cdot)$ refers to as the trace of a matrix.

Algorithm 3.3 presents BCGLS to address the potential breakdown problem caused by rank deficiency. Similar to BFBCG, we perform a rank revealing operation $orth(\cdot)$ on $S_{i+1} + \tilde{P}_i\tilde{\beta}_i$ to remove linearly dependent or zero vectors. When rank deficiency occurs, the dimension of space $\mathcal{P}_i$ reduces from s to r_i $(r_i < s)$ and correspondingly the search block $\tilde{P}_i$ shrinks to be an $n \times r_i$ matrix. Then, parameter matrices $\tilde{\alpha}_i$ and $\tilde{\beta}_i$ are designed to be $r_i \times s$ rectangular matrices and $\tilde{Q}_i$ appears to be $m \times r_i$, with respect to the change in search direction block $\tilde{P}_i$.

Algorithm 3.3: Block Conjugate Gradient for Least Square (BCGLS) Algorithm

Input: matrix $A \in \mathbb{R}^{m\times n}$, matrix $B \in \mathbb{R}^{m\times s}$, initial guess $X_0 \in \mathbb{R}^{n\times s}$, tolerance $tol \in \mathbb{R}$, and maximum number of iterations $maxit \in \mathbb{R}$
Output: an approximate solution $X_{sol} \in \mathbb{R}^{n\times s}$

```
$R_0 = B - AX_0$
$S_0 = A^T R_0$
$\tilde{P}_0 = orth(S_0)$
for $i = 0, \dots, maxit$
    $\tilde{Q}_i = A\tilde{P}_i$
    $\tilde{\alpha}_i = \left(\tilde{Q}_i{}^T \tilde{Q}_i\right)^{-1} \tilde{Q}_i{}^T R_i$
    $X_{i+1} = X_i + \tilde{P}_i\tilde{\alpha}_i$
    $R_{i+1} = R_i - \tilde{Q}_i\tilde{\alpha}_i$
    if converged within $tol$, then stop.
        $S_{i+1} = A^T R_{i+1}$
    if no rank deficiency occurs, then
        $\tilde{\beta}_i = \left(S_i{}^T S_i\right)^{-1} S_{i+1}{}^T S_{i+1}$
    else
        $\tilde{\beta}_i = -\left(\tilde{Q}_i{}^T \tilde{Q}_i\right)^{-1} \tilde{Q}_i A S_{i+1}$
    endif
    $\tilde{P}_{i+1} = orth\big(S_{i+1} + \tilde{P}_i\tilde{\beta}_i\big)$
```

```
end
$X_{sol} = X_{i+1}$
```

At the ith iteration step, an optimal point minimizing the underlying residual (error) function is chosen from $X_0 + span\{A^T R_0, (A^T A)A^T R_0, \dots, (A^T A)^i A^T R_0\}$ to be the least squares approximation, which can be expressed as a polynomial $\phi_i(A^T A)$ of degree i. Therefore, as an analogue to the standard Block Conjugate Gradient (BCG) methods [84], BCGLS yields a faster convergence rate of $\left(\frac{1-\sqrt{\kappa\prime^{-1}}}{1+\sqrt{\kappa\prime^{-1}}}\right)^2$, compared to Conjugate Gradient Least Squares (CGLS) [144,145] with a single right-hand side, where $\kappa' = \lambda_n/\lambda_s$, and λ_n and λ_s are the nth and sth eigenvalues of the product matrix $A^T A$, respectively.

In practice, it is very rare that the residual block has an exact linear dependency in BCGLS; however, much more often, vectors in the residual block will become nearly linearly dependent. In fact, linear dependency in the residual block S_i is monitored during BCGLS. If the smallest eigenvalue of S_i is lower than a designated threshold parameter τ, the search space will be reduced accordingly. Studies [79, 80] have shown that the nearly linear dependency in the block matrices may cause near-breakdown and have a serious impact to the convergence of block Krylov subspace methods. We use numerical examples to show that the linear dependency threshold parameter has an impact on solution precision as well as convergence speed and needs to be carefully selected.

3.3.2 Numerical Results

3.3.2.1 Handling Rank Deficiency

We compute the least squares solutions of a linear system with coefficient matrix "illc1850" to demonstrate the robustness of BCGLS in the case of rank deficiency. "illc1850" is a $1{,}850 \times 712$ rectangular matrix with $8{,}636$ nonzero elements arisen from the least squares problem in surveying [77]. A right-hand side block matrix B containing 100 column vectors with full column rank are generated randomly. A system is considered converged if the relative residual error of each solution with respect to its corresponding right-hand side is within the tolerance of 10^{-7}.

We start BCGLS with a zero initial solution block. Fig. 11 shows the number of columns in search matrix P_i after the rank-revealing operations (upper), the condition number of $Q_i^T A Q_i$ (middle), and the maximum and minimum relative residual errors among all solution columns in X_i (lower) along BFBCGLS iterations. One can find that rank deficiency (from 100 down to 88) starts to occur at the 6th iteration. After all, BCGLS is able to continue to explore the Krylov subspaces with reduced search space without suffering a breakdown, which leads to further residual error reduction in all systems as shown in Fig. 11 (lower).

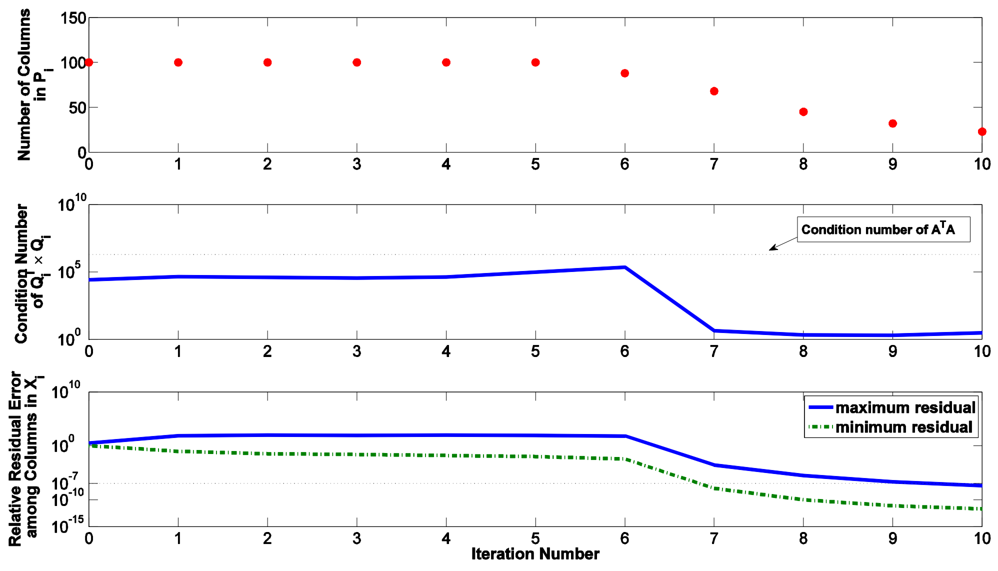


Fig. 11. Number of Columns in P_i (upper), condition number of $Q_i^T A Q_i$ (middle), and maximum and minimum relative residual norms of columns in X_i (lower) along BFBCGLS iterations

Fig. 12 compares the solution precision measured by the maximum residual norm among columns in X_i with respect to different linear dependency threshold parameter τ values. It is interesting to note that, when a large τ value is used, only solutions in low precision are obtained in BCGLS. This is due to the fact that, when a large τ value is reached, some solutions or linear combinations of solutions are considered converged and the search space is reduced without further improving these solutions. More importantly, a large τ value slows down convergence because of early reduction of search space. On the other hand, a τ value close to float-point number representation precision (10^{-16}.) does not necessarily lead to more precise solutions due to low-quality search spaces where the Galerkin conditions are not fully satisfied anymore. Our results indicate that the appropriate τ value should be in the range of $10^{-12} \sim 10^{-14}$ for BCGLS using double precision floating point operations.

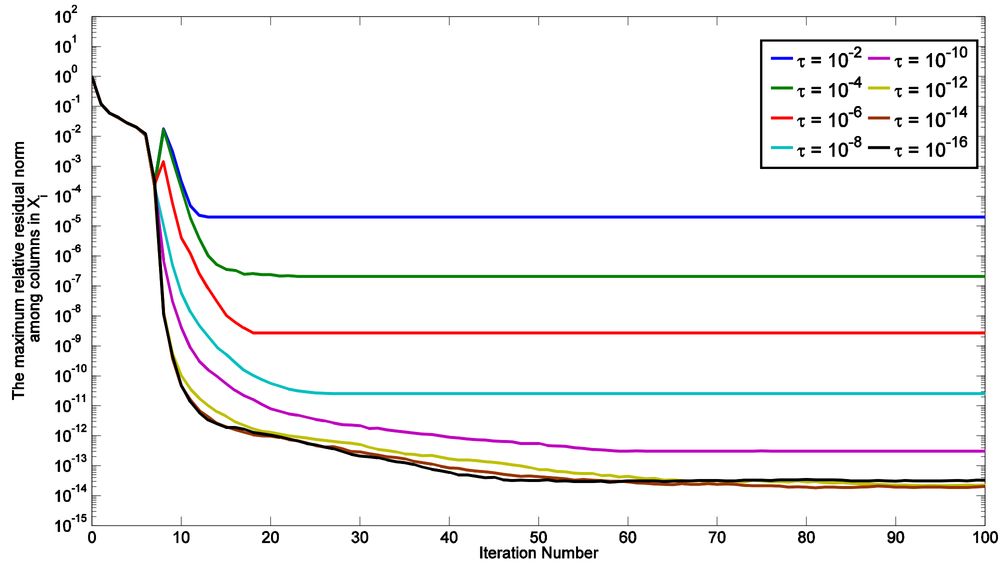


Fig. 12. Solution precisions obtained using different linear dependency threshold parameter τ values

3.3.2.2 Reducing Number of Passes

We compare CGLS and BCGLS to find the least squares solution of linear systems in terms of matrix passes. The least squares matrices chosen from the UFL sparse matrix collection [77] are used as coefficient matrices. In BCGLS, the right-hand side matrix B is set to be with 10 columns, where the first column coincides with the right-hand vector in CGLS while other columns are Gaussian random vectors. A satisfactory solution is considered achieved if the relative residual error of the first solution vector is within the tolerance of 10^{-7}.

Fig. 13 shows the number matrix passes performed using CGLS and BCGLS. One can find that the number of passes on all of the coefficient matrices is significantly reduced by using BCGLS, because of the improved convergence rate achieved in the block form of BCGLS. For matrices like “photogrammetry”, the total number of matrix passes is reduced by about 100 times by using BCGLS.

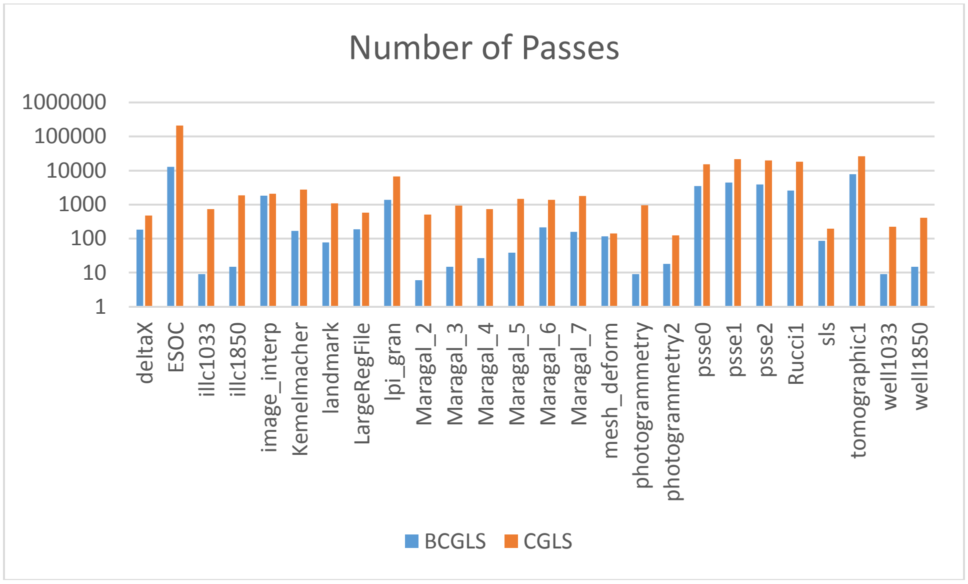


Fig. 13. The number of passes over matrix *A*

3.4 BCGLS Algorithm with Deflation (BCGLSD)

In this section, we propose a BCGLS algorithm with Deflation (BCGLSD) to accelerate block linear system convergence with deflation matrices. To obtain a high-quality of deflation matrix, Monte Carlo importance sampling is carried out to estimate and continuously refine the approximate smallest eigenvalues and eigenvectors of the large coefficient matrix during the course of iterations. These approximated eigenvectors are used to build up the deflation matrices. Numerical examples are provided to demonstrate the effectiveness of BCGLSD.

3.4.1 The BCGLSD Algorithm

Deflation is one of the popular techniques used in Krylov subspace methods to accelerate convergence via pre-adding the Krylov subspace with a space spanned by a deflation matrix, which contains approximations to the extreme eigenvectors [85,86,87]. Deflation has been widely used to handle positive definite systems [88, 89] and unsymmetric systems [90,91,92,93]. Recently, when multiple right-hand sides of a linear system are considered, the deflated algorithms are applied to BCG [71] and BGMRES [80]. More comprehensive analysis of deflated Krylov subspace methods can be found in [86, 85, 70, 94].

Deflation can be applied to BCGLS to improve convergence speed in finding solutions for least squares problem. Given a deflation matrix W, an augmented block Krylov subspace

$$span\{W, A^T R_0, (A^T A)^1 A^T R_0, \dots, (A^T A)^i A^T R_0, \dots\},$$

is constructed. In BCGLSD, as shown in Algorithm 3.4, an initial guess X_0 is formed as

$$X_0 = X_{-1} + W(L^T L)^{-1} L^T R_{-1}$$

where X_{-1} is an arbitrary block matrix and $L = AW$. Meanwhile, matrix orthogonalization related to W is carried out to generate the subsequent search matrices.

Algorithm 3.4: BCGLS Algorithm with Deflation (BCGLSD)

Input: matrix $A \in \mathbb{R}^{m\times n}$, matrix $B \in \mathbb{R}^{m\times s}$, matrix $X_{-1} \in \mathbb{R}^{n\times s}$, tolerance $tol \in \mathbb{R}$, maximum number of iterations $maxit \in \mathbb{R}$, and deflation matrix $W \in \mathbb{R}^{n\times t}$
Output: $X_{sol} \in \mathbb{R}^{n\times s}$

```
$L = AW$
$R_{-1} = B - AX_{-1}$
$X_0 = X_{-1} + W(L^T L)^{-1} L^T R_{-1}$
$R_0 = B - AX_0$
$S_0 = A^T R_0$
$\tilde{P}_0 = orth(S_0 - W(L^T L)^{-1} L^T A S_0)$
for $i = 0, \dots, maxit$
    $\tilde{Q}_i = A\tilde{P}_i$
    $\tilde{\alpha}_i = \left(\tilde{Q}_i^T \tilde{Q}_i\right)^{-1} \tilde{Q}_i^T R_i$
    $X_{i+1} = X_i + \tilde{P}_i \tilde{\alpha}_i$
    $R_{i+1} = R_i - \tilde{Q}_i \tilde{\alpha}_i$
    if converged within $tol$, then stop.
    $S_{i+1} = A^T R_{i+1}$
    $\tilde{\beta}_i = -\left(\tilde{Q}_i^T \tilde{Q}_i\right)^{-1} \tilde{Q}_i A S_{i+1}$
    $\tilde{P}_{i+1} = orth\left(S_{i+1} + \tilde{P}_i \tilde{\beta}_i - W(L^T L)^{-1} L^T A S_{i+1}\right)$
end
$X_{sol} = X_{i+1}$
```

Theorem 3.13 shows that the residual matrices R_i and the search matrices $\tilde{P}_i$ are constructed A-orthogonal and $A^T A$-orthogonal to deflation matrix W in BCGLSD, respectively.

Theorem 3.13. When deflated by a deflation matrix W, the following two orthogonality relations hold in DBCGLS,

(1) $W^T A^T A\tilde{P}_i = 0$;

(2) $W^T A^T R_i = 0$.

Proof. (1) Since $\tilde{P}_{i+1}$ is an orthogonal basis of the space spanned by the columns of $S_{i+1} + \tilde{P}_i\tilde{\beta}_i - W(L^T L)^{-1}L^T AS_{i+1}$, there exists an $n \times r_i$matrix δ such that

$$\tilde{P}_{i+1} = \left(S_{i+1} + \tilde{P}_i\tilde{\beta}_i - W(L^T L)^{-1}L^T AS_{i+1}\right)\delta.$$

Then, we have

$$\begin{aligned} W^T A^T A\tilde{P}_{i+1} &= W^T A^T A\left(S_{i+1} + \tilde{P}_i\tilde{\beta}_i - W(L^T L)^{-1}L^T AS_{i+1}\right)\delta \\ &= W^T A^T A\tilde{P}_i\tilde{\beta}_i\delta \end{aligned}$$

Clearly, because $W^T A^T A\tilde{P}_0 = 0$, subsequently, $W^T A^T A\tilde{P}_i = 0$ for all i.

(2) Since $R_{i+1} = R_i - \tilde{Q}_i\tilde{\alpha}_i$ and (1), we have

$$\begin{aligned} W^T A^T R_{i+1} &= W^T A^T R_i - W^T A^T \tilde{Q}_i\tilde{\alpha}_i \\ &= W^T A^T R_i \end{aligned}$$

As $X_0 = X_{-1} + W(L^T L)^{-1}L^T R_{-1}$ and $R_0 = (I - AW(L^T L)^{-1}L^T)R_{-1}$, it follows that

$$W^T A^T R_0 = W^T A^T(I - AW(L^T L)^{-1}L^T)R_{-1} = 0.$$

As a deduction, we can get $W^T A^T R_i = 0$ for all $i \geq 0$. $\square$

According to Theorem 3.13, in subsequent iterations in DBCGLS algorithm, the block Krylov subspace

$$span\left\{A^T R_0, (A^T A)^1 A^T R_0, \dots, (A^T A)^i A^T R_0, \dots\right\}$$

is constructed to be orthogonal to the subspace spanned by W. Let $H = \mathrm{I} - W(L^T L)^{-1}L^T A$ be a matrix projection onto the orthogonally complement subspace of W, DBCGLS is, in fact, equivalent to BCGLS starting with $A^T R_0$ on a system with the transformed coefficient matrix $H^T A^T AH$.

The effectiveness of BCGLSD depends strongly on the quality of the deflation matrix. The ideal deflation matrix W is composed of the exact extreme eigenvectors of $A^T A$. Suppose the columns in W contain t eigenvectors of matrix $A^T A$ corresponding to the t smallest eigenvalues, the impacts from these eigenvectors of matrix A^TA can be removed from matrix H^TA^TAH at the beginning, and thus DBCGLS algorithm has potentially faster convergence in a deflated system with a smaller condition number $\kappa' = \lambda_n/\lambda_{s+t}$, where λ_n and λ_{s+t} are the nth and the $(s+t)$th eigenvalues of A^TA, respectively.

3.4.2 Importance Sampling

When expanding a linear system to a block linear system, instead of adding arbitrary non-linearly dependent vectors to the multiple right-hand side B, we insert Gaussian distributed random vectors, such that $B = [b, \Omega]$, where Ω is an $n \times (s-1)$ Gaussian matrix. As the iterations move on, the approximate solution vectors gradually approach the space spanned by the smallest s eigenvectors of A, due to the effect of Monte Carlo sampling $A^{-1}\Omega$ which will be discussed in Chapter IV.

In BCGLSD, importance sampling is performed to improve the quality of the approximate smallest eigenvector vectors. Fig. 14 illustrates the importance sampling procedure of generating and refining deflation matrices. The basic idea is that after k steps of iterations, we restart BCGLSD by supplying a set of new vectors – a basis of the space spanned by the current solution vectors, to the right-hand side. In this way, the inverse power effect $(A^{-1})^p\Omega$ is expected to be performed on the solution matrix, where p denotes the number of restarts. As a result, this allows enhancing the accuracy of the approximate smallest s eigenvectors of A, which can be used to build a better quality deflation matrix.

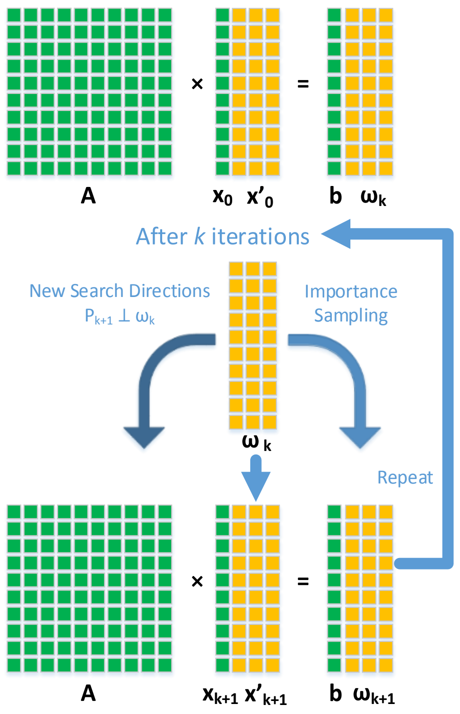


Fig. 14. Generate and refine deflation matrices via importance sampling

3.4.3 Numerical Results

3.4.3.1 Convergence Accelerations Using Deflation

We compare the convergence of CGLS, BCGLS, and BCGLSD on the least squares problem with coefficient matrix “wang4" from semiconductor device problem [77]. “wang4" is a $26{,}068 \times 26{,}068$ unsymmetric matrix with 177,196 nonzero elements. Assuming that we are only interested in the solution to one single right-hand side. To accommodate with the block form in BCGLS and BCGLSD, we expand the single right-hand side to a block form with 100 right-hand sides by supplying 99 Gaussian random vectors to the right-hand side. In BCGLSD, the importance sampling is carried out every 512 steps to refine the deflation matrix W.

Fig. 15 displays the numerical results of CGLS, BCGLS, and BCGLSD. One can clearly observe that by expanding the linear system from a single right-hand side to a block form with multiple right-hand sides, BCGLS (1,834 steps) takes fewer iteration steps to converge to 10^{-7} relative residual error than CGLS (59,765 steps). Even though overall BCGLS involves more computational operations measured by the total number of matrix-vector multiplications than those of CGLS, it is important to note that BCGLS is a communication-efficient algorithm that can significantly reduce the number of passes over matrix A, the main computational bottleneck if passing over all elements in A is extremely costly. More importantly, when an approximate deflation matrix is applied, convergence can be significantly accelerated, where the number of iterations to reach convergence is further reduced down to 935 steps.

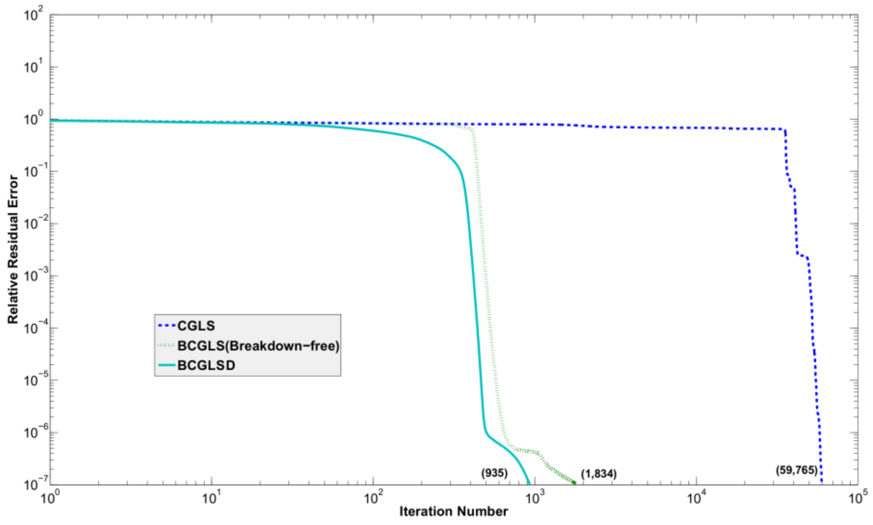


Fig. 15. Comparison of convergence in CGLS, BCGLS, and BCGLSD on a least squares problem using “wang4" as the coefficient matrix

3.4.3.2 Handling ill-conditioned Coefficient matrices using Deflation

We use a linear system with “gre_1107", a $1{,}107 \times 1{,}107$ unsymmetric matrix with $5{,}664$ nonzero elements, as the coefficient matrix to study the behavior of BCGLS in ill-conditioned least squares problems [77]. Fig. 16 shows the eigenvalue distribution of $A^T A$. One can find that the 40 extremely small eigenvalues lead to a large condition number in $A^T A$. The condition number of $Q_i^T Q_i$ is bounded by that of $A^T A$.

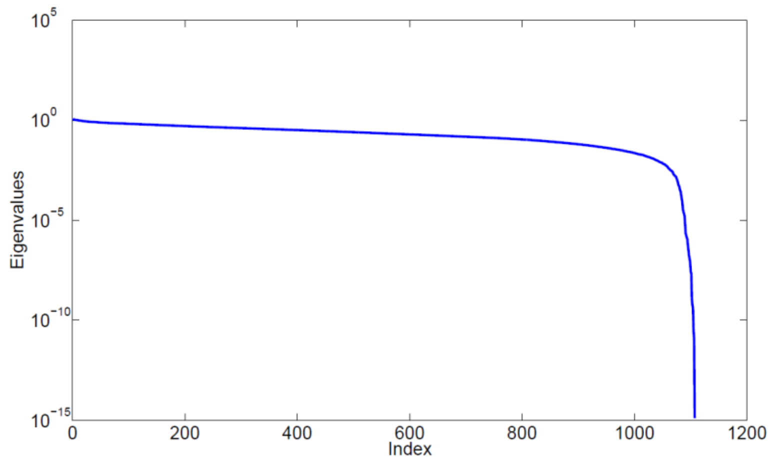


Fig. 16. Distribution of the eigenvalues in A^TA (“gre_1107") the condition

As shown in Fig.s 17, when the condition number of $Q_i^TQ_i$ is small during BCGLS iterations before step 11, orthogonality is well preserved. However, at iteration step 11, the large condition number of $Q_i^TQ_i$ causes subsequent loss of orthogonality, as shown in the colormap of A^TA-orthogonality among the first 31 search matrices. Consequently, BCGLS converges slowly and does not reach the desired precision of 10^{-7} even after $10{,}000$ iterations. An appropriate deflation matrix can address this issue and accelerate the convergence of BCGLS. Here we use a deflation matrix consisting of 40 approximated eigenvectors corresponding to the 40 extreme eigenvalues obtained from importance sampling. When the deflation matrix is applied, the condition number of $Q_i^TQ_i$remains relatively small and orthogonality is mostly preserved during BCGLSD iterations, as shown in Fig. 18. As a result, BCGLSD converges at iteration step 11.

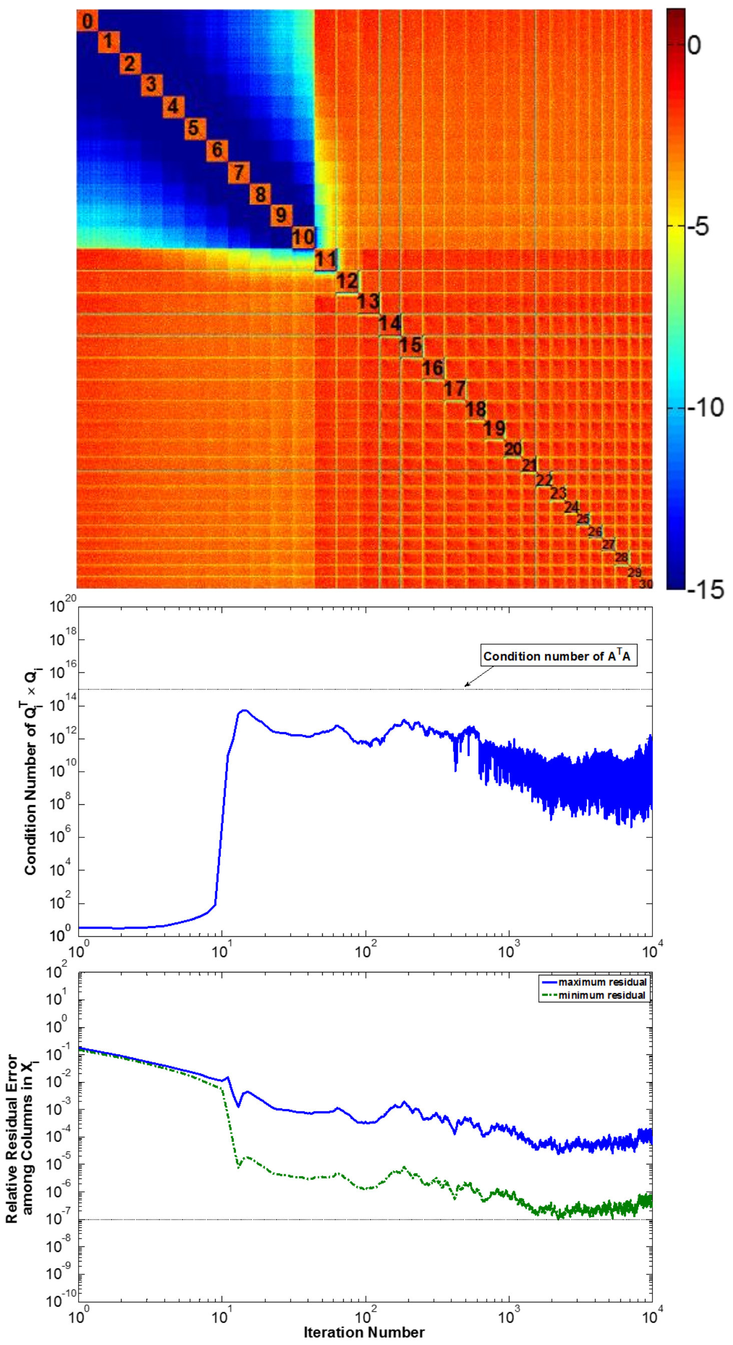


Fig. 17. Colormap of A^TA -orthogonality between Search Matrices in the first 31 iterations (upper), condition number of $Q_i^T Q_i$ (middle), and maximum and minimum relative residual norms of columns in X_i (lower) for a block linear system with 100 right hand sides using “gre_1107" as the coefficient matrix along BCGLS iterations

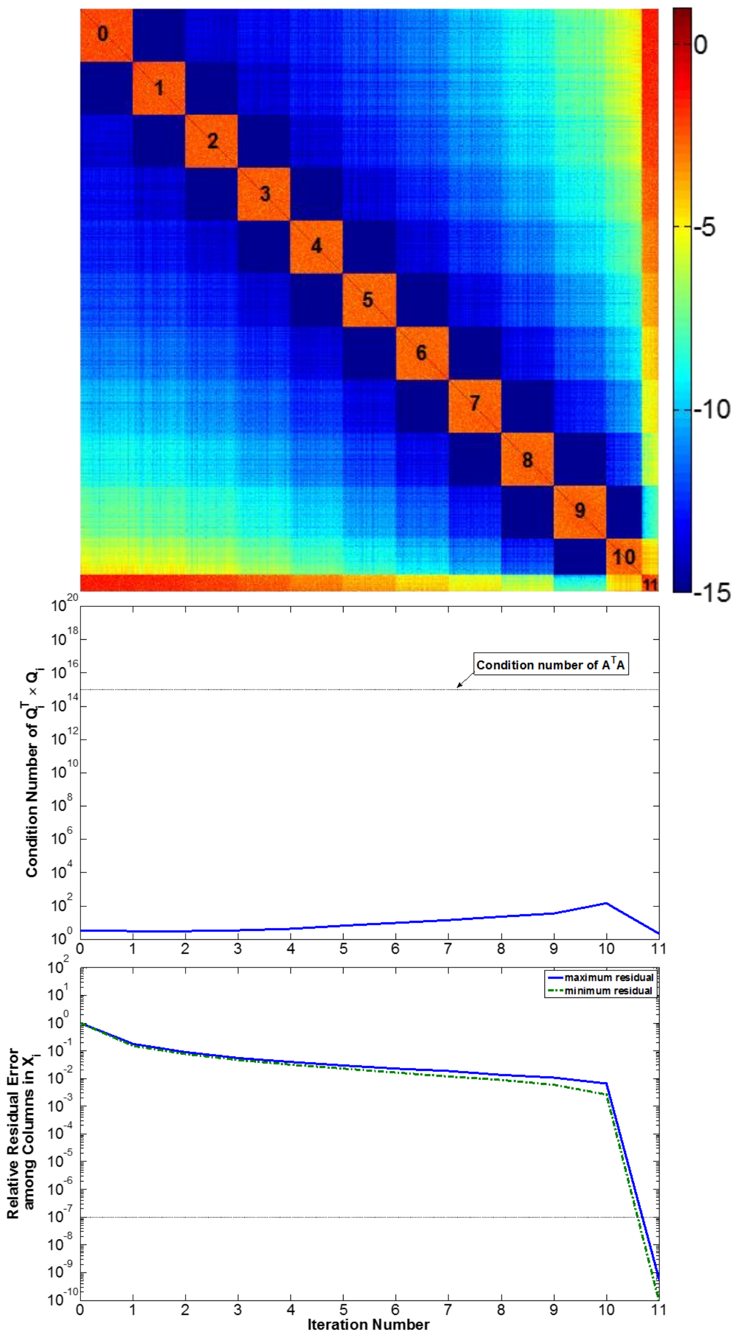


Fig. 18. Colormap of A^TA -orthogonality between Search Matrices in the first 12 iterations (upper), condition number of $Q_i^TQ_i$ (middle), and maximum and minimum relative residual norms of columns in X_i (lower) for a block linear system with 100 right hand sides using “gre_1107" as the coefficient matrix along BCGLSD iterations, where the deflation matrix consists of 40 approximated extreme eigenvectors

3.5 Monte Carlo GMRES (MCGMRES) Algorithm

In Krylov-subspace based solvers, performing precise matrix-vector multiplications at each iteration can be expensive when coefficient matrices are very large or matrix elements need to be regenerated when accessed. In practice, an inexact scheme is suggested as a remedy to the costly matrix-vector multiplication [95, 96, 97, 98], which uses a quick approximation. However, the inexact scheme relies on the underlying approximation error -- if the approximation error is high, the approximate matrix-vector products may lead to divergence of the Krylov-subspace solvers [99, 100].

Recent advances in Monte Carlo sampling algorithms enable approximating matrix-vector products with relatively low computation cost while yielding high confidence [22, 26, 27, 29]. Here, a Monte Carlo GMRES (MCGMRES) algorithm is developed where Monte Carlo sampling is used to approximate matrix-vector multiplications.

3.5.1 Inexact Matrix Product using Sampling

In the literature, a number of Monte Carlo sampling approaches have been used to approximate matrix products, such as random walk-based sampling [101] and row/column sampling [102]. Here, we use column sampling [102] as an example to carry out an inexact matrix-vector product. Let A be an $m \times n$ matrix and b an $n \times 1$ vector, where both m and n are large. Then, the product $c = Ab$ can be approximated based on s sampled columns of matrix A, as shown in Algorithm 3.5.

```
Algorithm 3.5: Inexact Matrix-vector Product with Column Sampling ( IMv )
Step 1: Generate a random integer $k$ between 1 and $n$ with probabilities $p_j, j = 1, \dots, n$;
Step 2: $Q^{(i)} = A^{(k)}/\sqrt{sp_k}$ , $t_{(i)} = b_{(k)}/\sqrt{sp_k}$ , and $i = i + 1$;
Step 3: Repeat steps 1-2 until $i = s$;
Step 4: Compute matrix product $Qt$.
```

Let $A^{(k)}$ be the kth column of matrix A and $b_{(k)}$ be the kth element of b. Since the product c can be expressed as $c = \sum_{k=1}^{n} A^{(k)} b_{(k)}$, if we assign x as a discrete random variable with probability $\left(x_i = \frac{A^{(i)} b_{(i)}}{\mathrm{p}_i}\right) = p_i$, $i = 1, \dots, n$, where probability p_i is assigned to ith column $A^{(i)}$ or the corresponding ith element of $b_{(i)}$, one can find that the product c is the expectation of x, such that

$$c = \sum_{k=1}^{n} \frac{A^{(k)} b_{(k)}}{p_k} p_k = E(x)$$

By constructing matrices Q and t with contain s samples from matrix A and vector b, respectively, the product c is approximated as

$$Qt = \frac{1}{s} \sum_{k=1}^{s} \frac{A^{(i_k)} b_{(i_k)}}{p_{i_k}} \approx E(x) = c$$

where i_k's are integers between 1 and n. Theoretically, the approximation error in column sampling is bounded as

$$\|Ab - Qt\|_2 = O\left(\frac{\|A\|_F \|b\|_F}{\sqrt{s}}\right)$$

with high probability [102].

3.5.2 The MCGMRES Algorithm

The MCGMRES algorithm is built by integrating the GMRES algorithm with column sampling scheme. Algorithm 3.6 shows the procedure of MCGMRES to solve a system of linear equations

$$Ax = b,$$

where matrix $A \in \mathbb{R}^{m\times n}$ and vector $b \in \mathbb{R}^m$.

Algorithm 3.6: Monte Carlo GMRES (MCGMRES) Algorithm

Input: matrix $A \in \mathbb{R}^{m\times n}$, vector $b \in \mathbb{R}^m$, sampling size $s \in \mathbb{N}$ per iteration, initial solution guess $x_0 \in \mathbb{R}^n$, tolerance $tol \in \mathbb{R}$, and maximum number of iterations $maxit \in \mathbb{R}$

Output: $x_{sol} \in \mathbb{R}^n$

```
$r_0 = b - IMv(A, x_0, s)$
$h_{10} = \|r_0\|_2$
for $i = 0, \dots, maxit$
    $q_{i+1} = {}^{r_i}/_{h_{i+1,i}}$
    $r_i = IMv(A, q_i, s)$
    for $j = 1, \dots, i$
        $h_{j,i} = q_j^T r_i$
        $r_i = r_i - h_{j,i} q_j$
    end
    $h_{i+1,i} = \|r_i\|_2$
    Solve the least-squares problem $min\left\|h_{10} e_1 - \widetilde{H}_i y_i\right\|_2$
    $x_{i+1} = x_0 + Q_i y_i$
    If converged, then stop.
```

```
end
$x_{sol} = x_{i+1}$
```

Vector x_0 is the initial solution guess and $\widetilde{H}_i$ is an upper Hessenberg matrix. At the ith iteration, x_{i+1} has the form $x_0 + Q_i y_i$, where y_i is the least squares solution to $\left\| h_{10} e_1 - \widetilde{H}_i y_i \right\|_2$ which locates x_{i+1} to minimize $\|b - Ax\|_2$ over space

$$x_0 + span\{r_0, r_1, \dots, r_i\}.$$

Here r_i is the ith approximate residual vector. $IMv(A, q_i, s)$ performs column sampling to generate a vector approximating Aq_i, based on the s sampled columns of matrix A. Thus, the space $x_0 + span\{r_0, r_1, \dots, r_i\}$ is an approximation to the actual Krylov subspace $x_0 + \{r_0, Ar_0, A^2 r_0, A^3 r_0, \dots\}$.

By writing inexact matrix-vector operation in the form

$$x' = (A + \Delta A)x,$$

where ΔA is the perturbation on the matrix A, the study in [99] shows that inexact matrix-vector products would not significantly affect the convergence of GMRES, if the perturbation error $\|\Delta A\|_2$ can satisfy the condition of

$$\frac{\|\Delta A\|_2}{\|A\|_2} \in [tol, 1],$$

where tol donotes the specified tolerance threshold. Suppose that column sampling without replacement is carried out in MCGMRES. Then, it is clearly seen that $\|\Delta A\|_2 < \|A\|_2$ holds, since column sampling uses only a subset of the columns in A to produce inexact matrix-vector product. Therefore, MCGMRES based on column sampling not only can decrease the computational cost at each iteration, but also is able to greatly maintain the convergence properties of GMRES.

3.5.3 Numerical Results

We use a $10{,}000 \times 10{,}000$ random matrix to show the capability of MCGMRES in trading off speed and accuracy. The column sampling is carried out without replacement.

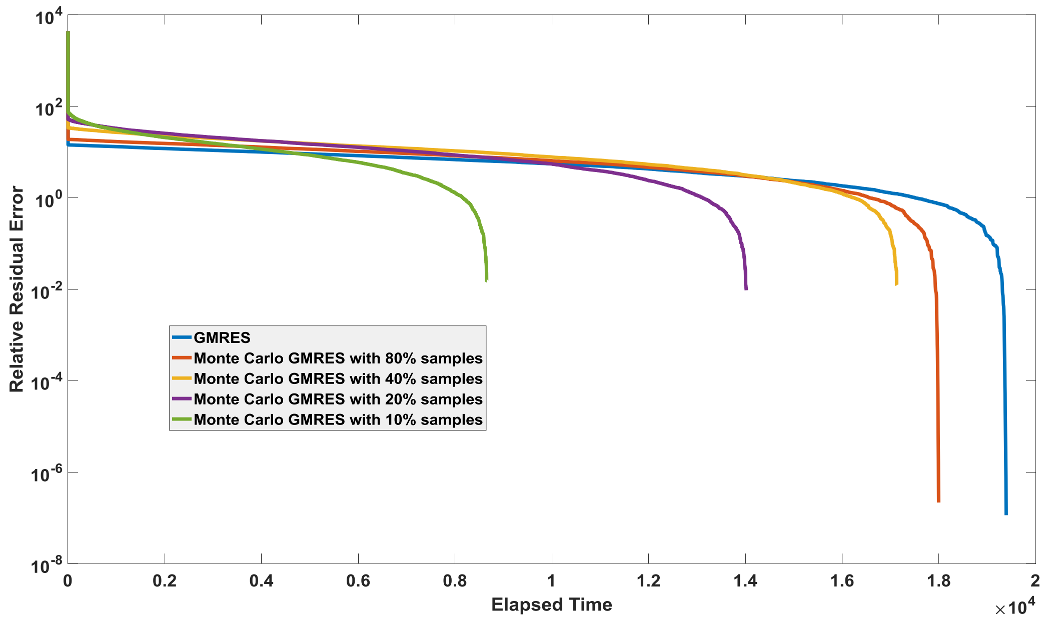


Fig. 19. Comparison of MCGMRES with different percentage of samples

Fig. 19 shows the numerical results of MCGMRES with different percentage of samples. One can find that by using 10% randomly selected columns, MCGMRES takes only about 44% computational time of the original GMRES to obtain a solution with accuracy of 10^{-2}. When more samples are used in MCGMRES, the computational costs gradually increases, but it allows higher accuracy solutions to be achieved. It is important to note that MCGMRES would be a good choice for some large-scale problems where a high-accuracy solution is not necessary but fast approximation of the solution is important.

CHAPTER IV

MONTE CARLO METHODS FOR LOW-RANK MATRIX APPROXIMATIONS

Constructing a low-rank matrix approximation with a suitable rank is critical to many data analytic applications. In this chapter, we present a Rank-Revealing Randomized Singular Value Decomposition (R^{3}SVD) algorithm to incrementally construct a low-rank approximation while estimating the appropriate rank (Section 4.1).

The main contribution of this work is the design of an importance sampling method - a new form of Gaussian sampling based on orthogonal projection to obtain the leftover dominant subspace and add up to existing low-rank approximation. Several application examples are provided to demonstrate that R^{3}SVD is more efficient in terms of computation time and memory while providing a better rank estimation, compared to the other existing approaches. Moreover, R^{3}SVD is a memory-aware algorithm that the computation can be tailored to tasks to fit in the constant amount of memory.

4.1 Rank-Revealing Randomized Singular Value Decomposition (R^{3}SVD) algorithm

Our R^{3}SVD algorithm [150] is based on the randomized SVD algorithm with Gaussian sampling (RSVD) proposed by Halko et al. [33, 103], although it can be straightforwardly extended to other randomized SVD strategies. In this section, we first overview the RSVD algorithm and existing strategies used to estimate rank value k. Then, we describe our R^{3}SVD algorithm to adaptively estimate a low-rank approximation. Finally, numerical examples are presented.

4.1.1 RSVD and Rank Estimation

The basic idea of RSVD is to use Gaussian vectors to construct a small condensed subspace from the range of A, whose dominant actions could be quickly estimated from this small subspace with relatively low computation cost while yielding high confidence. The procedure of RSVD is described in

Algorithm 4.1.

```
Algorithm 4.1: Randomized SVD Algorithm with Gaussian Sampling (RSVD)
Input: $A \in \mathbb{R}^{m\times n}$, a target matrix rank $k \in \mathbb{N}$, and an oversampling parameter $p \in \mathbb{N}$ satisfying $k + p \leq min(m, n)$.
Output: Low rank approximation $U_L \in \mathbb{R}^{m\times k}$, $\Sigma_L \in \mathbb{R}^{k\times k}$, and $V_L \in \mathbb{R}^{n\times k}$

Construct an $n \times (k + p)$ Gaussian random matrix $\Omega$
$Y = A\Omega$
Compute an orthogonal basis $Q = qr(Y)$
$B = Q^T A$
$[U_B, \Sigma_B, V_B] = svd(B)$
Update $U_B = QU_B$
$U_L = U_B(:,1:k)$, $\Sigma_L = \Sigma_B(1:k, 1:k)$ , and $V_L = V_B(:,1:k)$
```

Given a desired rank k and an oversampling parameter p (typically a small constant), RSVD constructs an $n \times (k + p)$ Gaussian random matrix block Ω, whose elements are normally distributed. Ω condenses a large matrix A into a "tall-and-skinny," dense block matrix Y by $Y = A\Omega$. Y captures the most important actions of A and a basis Q is derived by decomposing Y. Q is designed to approximate the left singular vectors of A by minimizing $||QQ^T A - A||_F^2$. Then, Q is applied back to A to obtain a "short-and-wide" block matrix $B = Q^T A$. Calculation of SVD on B yields an approximated Singular Value Decomposition of A. The result $U_L \Sigma_L {V_L}^T$ forms a k-rank matrix approximation to A.

The major operations in RSVD include matrix-block matrix multiplications as well as QR and SVD decompositions on the block matrices. Specifically, matrix-block matrix multiplications take $O(2(k + p)T_{mult})$ floating-point operations, where T_{mult} denotes the computational cost of a matrix-vector multiplication. For a large matrix A where $m, n \gg k + p$, the computational cost of matrix-block matrix multiplications dominates those of the decomposition operations, which requires $O((k + p)^2(m + n))$ floating operations. RSVD needs to store the intermediate matrices, such as Ω, Y, Q, and B, and thus its space complexity is $O(2(m + n)(k + p))$. As a result, with a tradeoff of accuracy, RSVD is usually more efficient than the full SVD algorithms in terms of computational and memory cost.

The desired rank k is a required input parameter in the randomized SVD algorithms. However, in many practical applications, the value of k is unknown beforehand and needs to be appropriately estimated. In the literature, two popular strategies are often used to estimate k. One is to evaluate a suitable basis Q and then determine the appropriate k before carrying out RSVD. For instance, Voronin and Martinsson [104] proposed two algorithms, Autorank I and Autorank II, to evaluate a basis Q for a range space that captures the most actions of matrix A. Autorank I is based on overestimation by using a very large value k at the beginning and then selecting dominant information from the resulting pool of singular values/vectors. Although Autorank I is often able to obtain good low-rank approximations, largely overestimated k will result in significant computation cost increase, because the computational cost of decompositions on the tall-and-skinny or short-and-wide matrices in RSVD grows rapidly with $O(k^2)$ and is no longer negligible. At the same time, the memory requirement of Autorank I increases in the order of k. Instead of overestimating k, Autorank II gradually samples the range of A to calculate error $||QQ^TA - A||_F^2$ in order to obtain an estimation of k. Similar to Autorank II, the Adaptive Randomized Range Finder algorithm [33] employs the incremental sampling approach with a probabilistic error estimator based on the relation between the rank k with respect to the theoretical error bound to predict a reasonable basis Q with a reasonable value of k. However, this theoretical error bound is loose and consequently k is often largely overestimated, which will be shown in section 4. More recently, the Randomized Blocked algorithm [105], a block version of Randomized Range Finder algorithm, is developed to improve computational efficiency. Instead of using the probabilistic error estimator, the Randomized Blocked algorithm explicitly updates matrix A by removing the portion projected on Q and terminates at a situation when matrix A becomes small enough. An alternative strategy is to adaptively estimate a suitable rank k during RSVD. A simple approach is restarting RSVD, which starts with a small guessed rank and then repeats RSVD computation until the low-rank approximation with the desired accuracy is reached. This restarting approach can often result in a good low-rank approximation; however, the previous RSVD trials are only used to estimate k and do not contribute to final low-rank approximation.

4.1.2 The R^{3}SVD Algorithm

Algorithm 4.2 describes the proposed R^{3}SVD algorithm. The rationale of R^{3}SVD is to build a low-rank approximation incrementally based on orthogonal Gaussian projection. Initially, a t-rank approximation is obtained, where t is an initial guess of k which can be justified according to the memory available. The energy percentage is estimated accordingly. If the energy percentage obtained so far does not satisfy the application requirement, a new t-rank approximation is calculated in the subspace orthogonal to the space of the previous low-rank approximation. Then, the new t-rank approximation will be added to the previous one to form a $2t$-rank approximation and its corresponding energy percentage is estimated. The above process is repeated until the incrementally built low-rank approximation has secured a satisfactory percentage of energy from A.

Compared to RSVD in Algorithm 4.1, R^{3}SVD incorporates three major changes including importance sampling, orthogonalization process, and stopping criteria based on energy estimation.

Algorithm 4.2: Rank Revealing Randomized SVD (R^{3}SVD) Algorithm

Input: $A \in \mathbb{R}^{m\times n}$, sampling size $t \in \mathbb{N}$ per iteration, oversampling number $p \in \mathbb{N}$, maximum number of iterations $maxit \in \mathbb{N}$, and energy threshold $\tau \in \mathbb{R}$.
Output: Low rank approximation $U_L \in \mathbb{R}^{m\times k'}$, $\Sigma_L \in \mathbb{R}^{k'\times k'}$, $V_L \in \mathbb{R}^{n\times k'}$, and estimated rank k'

```
// initialization
Construct an $n \times (t+p)$ standard Gaussian matrix $\Omega$
$G_0 = \Omega$ and $V_L = \emptyset$, $U_L = \emptyset$, $\Sigma_L = \emptyset$
$k' = 0$
for $i = 0: maxit$
    $Y_i = AG_i$
    $Q_i = qr(Y_i, 0)$
    $B_i = {Q_i}^T A$
    $[U_{B_i}, \Sigma_{B_i}, V_{B_i}] = svd(B_i, 0)$
    $U_i = Q_i U_{B_i}$
    $\Sigma_i = \Sigma_{B_i}$
    $V_i = qr(V_{B_i} - V_L({V_L}^T V_{B_i}), 0)$          // orthogonalization process
    $U_L \leftarrow [U_L, U_i(:,1:t)]$, $\Sigma_L \leftarrow \begin{bmatrix} \Sigma_L & 0 \\ 0 & \Sigma_i(1:t,1:t) \end{bmatrix}$, $V_L \leftarrow [V_L, V_i(:,1:t)]$
    for $j = 1: t$
        $k' = i \times t + j$
        $\tilde{\varphi}_{k'} = \frac{\sum_{i=1}^{k'} {\sigma'_i}^2}{\|A\|_F^2}$          // estimate energy percentage
        if $\tilde{\varphi}_{k'} \geq \tau$, then stop;
    end
    $G_{i+1} = G_i - V_i(V_i^T G_i)$        // update Gaussian matrix
end
[$\Sigma_L$, Idx] = sort($\Sigma_L$,'descend');        // sort the approximate singular values
$V_L$ = $V_L$ (:, Idx);
$U_L$ = $U_L$ (:, Idx);
```

4.1.2.1 Importance Sampling

Suppose that V_L is an $n \times t$ matrix composed of t right singular vectors of a low-rank approximation $U_L \Sigma_L V_L$, which is supposed to capture most of the energy in A. Then, the range space $ran(A^T)$ can be divided into two orthogonal spaces: space $ran(V_L)$ spanned by the columns in V_L and its orthogonal complement $ran(V_L)^{\perp}$. Obviously, if V_L consists of only partial dominant actions of A, the rest dominant information is left over in the space $ran(V_L)^{\perp}$.

R^{3}SVD is designed to incrementally add up a low-rank approximation. To this end, R^{3}SVD needs to sample the space $ran(V_L)^{\perp}$ orthogonal to V_L to extract the left-over dominant information of A. Here, we employ importance sampling by constructing a new sampling matrix G, such as

$$G = (I - P_V)\Omega$$

where P_V is an orthogonal projection onto the space $ran(V_L)$, Ω is a standard Gaussian matrix, and I is the identity matrix. Theorem 4.1 shows that G is a Gaussian matrix orthogonal to $ran(V_L)$.

Theorem 4.1. Assuming that V_L is an $n \times t$ non-empty matrix with orthonormal columns, then

1) G is orthogonal to V_L; and

2) elements in G are normally distributed.

Proof. 1) Since V_L is an $n \times t$ matrix with orthonormal columns, P_V can be derived as $P_V = V_L V_L^{\mathrm{T}}$. Obviously, $V_L^T(I - P_V) = V_L^T - V_L^T V_L V_L^T = 0$ holds.

2) As $I - P_V$ is the orthogonal projection onto $ran(V_L)^{\perp}$, which is the orthogonal complement of space $ran(V_L)$, we can denote an $n \times (n-t)$ matrix $\tilde{V} = (\tilde{v}_{ij})$ as a basis of the space $ran(V_L)^{\perp}$ and then $I - P_V = \tilde{V}\tilde{V}^T$. Then, each element g_{ij} in G can be expressed as

$$g_{ij} = \sum_{s=1}^{n} \left(\sum_{h=1}^{n-t} \tilde{v}_{ih} \tilde{v}_{sh} \right) \omega_{sj}$$

where ω_{sj} denotes an element of Ω in row s of column j. Since element ω_{sj}'s are independent standard normal distributed variables, the characteristic function $\Phi_{g_{ij}}(x)$ can be obtained as

$$\Phi_{g_{ij}}(x) = \Phi_{\sum_{s=1}^{n}(\sum_{h=1}^{n-t} \tilde{v}_{ih}\tilde{v}_{sh})\omega_{sj}}(x)$$

$$= \prod_{s=1}^{n} \prod_{h=1}^{n-t} e^{-\frac{1}{2}(\tilde{v}_{ih}\tilde{v}_{sh}x)^2}$$

$$= \prod_{h=1}^{n-t} \prod_{s=1}^{n} e^{-\frac{1}{2}(\tilde{v}_{ih}\tilde{v}_{sh}x)^2}$$

$$= \prod_{h=1}^{n-t} e^{-\frac{1}{2}\tilde{v}_{ih}{}^2(\sum_{s=1}^{n} \tilde{v}_{sh}{}^2)x^2}.$$

As the columns of $\tilde{V}$ are orthonormal such that $\left(\sum_{s=1}^{n} \tilde{v}_{sh}{}^2\right) = 1$, we have

$$\Phi_{g_{ij}}(x) = e^{-\frac{1}{2}\sum_{h=1}^{n-t} \tilde{v}_{ih}{}^2 x^2}.$$

Since the characteristic function uniquely determines the probability distribution of a random variable [106], it suffices to show that g_{ij} is normally distributed with expected value zero and variance $\tilde{\sigma}_{ij}{}^2 = \sum_{h=1}^{n-t} \tilde{v}_{ih}{}^2$, i.e., $g_{ij} \sim N\left(0, \tilde{\sigma}_{ij}{}^2\right)$. □

To avoid resampling of the space $ran(V_L)$, the product of matrix AG in R^{3}SVD focuses on revealing the dominant information from the space $ran(V_L)^{\perp}$ orthogonal to $ran(V_L)$. Since the number of dominant singular values is unknown in advance, R^{3}SVD generates a series of Gaussian matrices $G_1, G_2, \ldots$ to iteratively explore the orthogonal subspace of the obtained low-rank approximation until a satisfactory low-rank approximation is obtained.

4.1.2.2 Orthogonalization Process

Let $V_L = [V_1, V_2, \ldots V_i\,]$ denote a matrix containing the approximate right singular vectors obtained in R^{3}SVD at the ith iteration step. The singular vectors in the V_{i+1} must be orthogonal to V_L.

However, the inherent numerical errors may cause loss of orthogonality between V_{i+1} and V_L.

To ensure the orthogonality property, we generate V_{i+1} by employing an orthogonalization process to remove the components of V_{B_i} that are not orthogonal to the previous right singular vectors in V_L such that

$$V_{i+1} = qr(V_{B_{i+1}} - V_L(V_L{}^T V_{B_{i+1}}), 0).$$

Proposition 4.2 indicates that the resulting matrix V_{i+1} generated at the $(i+1)$th iteration step in R^{3}SVD is orthogonal to V_L.

Proposition 4.2. Using the transformed matrix V_{i+1}, $V_L{}^T V_{i+1} = 0$ holds.

Proof. Denoting $Z_{i+1} = V_{B_{i+1}} - V_L(V_L{}^T V_{B_{i+1}})$, we can get $V_L{}^T Z_{i+1} = V_L{}^T \left(V_{B_{i+1}} - V_L(V_L{}^T V_{B_{i+1}})\right) = 0$. Since V_{i+1} is a basis of $ran(Z_{i+1})$, $V_L{}^T V_{i+1} = 0$ holds. □

Based on Proposition 4.2, the orthogonality property of the resulting left singular vectors U_L can be proved, which is shown in Proposition 4.3.

Proposition 4.3. Using the transformed matrix V_{i+1}, $U_L{}^T U_{i+1} = 0$ holds.

Proof. Denoting the QR decomposition of Y_{i+1} by $Y_{i+1} = Q_{i+1}R_{i+1}$. We can have

$$\begin{aligned} Q_j{}^T Q_{i+1} &= Q_j{}^T Y_{i+1} R_{i+1}^{-1} \\ &= Q_j{}^T A G_{i+1} R_{i+1}^{-1} \\ &= Q_j{}^T A(I - P_{V_L})\Omega R_{i+1}^{-1} \\ &= B_j(I - P_{V_L})\Omega R_{i+1}^{-1} \\ &= U_{B_j}\Sigma_{B_j} V_{B_j}{}^T (I - P_{V_L})\Omega R_{i+1}^{-1} \end{aligned}$$

where $V_L = [V_1, V_2, \dots V_i\ ]$.

Denote $V^- = [V_1, V_2, \dots V_{j-1}\ ]$ and $V^+ = [V_{j+1}, \dots, V_i\ ]$ for $j \le i$. According to Proposition 4.2, the columns in V_L are orthogonal to each other. Then, $(I - P_{V_L})$ can be expressed as

$$\left(I - P_{V_L}\right) = (I - P_{V^-})\left(I - P_{V_j}\right)(I - P_{V^+}).$$

Since $V_j = qr\left(V_{B_j} - V^-\left(V^{-T}V_{B_j}\right), 0\right)$, it follows that

$$(I - P_{V^-})V_{B_j} = V_jR_j.$$

Therefore,

$$\begin{aligned} {V_{B_j}}^T\left(I - P_{V_L}\right) &= {V_{B_j}}^T(I - P_{V^-})\left(I - P_{V_j}\right)(I - P_{V^+}) \\ &= {R_j}^T{V_j}^T\left(I - P_{V_j}\right)(I - P_{V^+}) \\ &= 0, \end{aligned}$$

and thus ${Q_j}^TQ_{i+1} = 0$, for $j \leq i$. Hence,

$${U_L}^TU_{i+1} = [U_1, U_2, \ldots U_i\,]^TU_{i+1} = \begin{bmatrix} U_{B_1}^TQ_1^TQ_{i+1}U_{B_{i+1}} \\ U_{B_2}^TQ_2^TQ_{i+1}U_{B_{i+1}} \\ \vdots \\ U_{B_{i+1}}^TQ_i^TQ_{i+1}U_{B_{i+1}} \end{bmatrix} = 0.$$

□

The orthogonalization process requires $O(\,(2ti + 1)(t + p)n\,)$ operations to ensure the orthogonality properties of singular vectors obtained in the previous iterations. Moreover, by taking advantage of the orthogonality between V_{i+1} and V_L, the next Gaussian matrices G_{i+1} can be fast generated using the following short recursive formula,

$$G_{i+1} = \left(I - \sum_{j=1}^{i} P_{V_j}\right)\Omega = G_i - P_{V_i}G_i,$$

where P_{V_j} is the orthogonal projection onto the space spanned by V_j, such that $P_{V_j} = V_j{V_j}^T$, and Ω is a standard Gaussian matrix. Since G_{i+1} is generated directly from G_i, the orthogonal Gaussian sampling takes only $O\left((2t + 1)(t + p)n\,\right)$ operations.

4.1.2.3 Energy Estimation and Stopping Criteria

The incremental low-rank approximation buildup process in R^{3}SVD will be terminated when sufficient percentage of energy of A is secured. The percentage threshold τ is typically specified by the applications, which often ranges from 80% to 99%.

Let $U_L = [U_1, U_2, \ldots U_i\ ]$ denote a matrix of the approximate left singular vectors. The actual energy percentage of the low-rank approximation obtained at the ith iteration step can be evaluated based on

$$\varphi = \frac{\left\|U_L {U_L}^T A\right\|_F^2}{\|A\|_F^2}.$$

However, calculating $\left\|U_L {U_L}^T A\right\|_F^2$ at each iteration is rather costly. In R^{3}SVD, the following measure $\tilde{\varphi}_{k\prime}$ is carried out to quickly estimate the energy percentage of the obtained low-rank approximation, such that

$$\tilde{\varphi}_{k\prime} = \frac{\sum_{i=1}^{k\prime} {\sigma_i'}^2}{\|A\|_F^2}.$$

where σ_i' denotes the ith approximate singular value in R^{3}SVD. It is important to note the approximate singular values σ_i''s are available during calculation of SVD on B_i, where $B_i = Q_i^T A$. Therefore, the energy percentage can be evaluated at (almost) no cost.

Proposition 4.4 shows the estimated energy $\tilde{\varphi}_{k\prime}$ equals to the accurate energy φ, and it guarantees that the low-rank approximation obtained by R^{3}SVD satisfies the accuracy requirement of the applications.

Proposition 4.4. $\tilde{\varphi}_{k\prime} = \varphi'$.

Proof. Since the columns in U_L are orthogonal, we have

$$\left\|U_L {U_L}^T A\right\|_F^2 = \sum_{j=1}^{i} \left\|U_j {U_j}^T A\right\|_F^2 = \sum_{j=1}^{i} \left\|{U_j}^T A\right\|_F^2$$

$$= \sum_{j=1}^{i} \left\| {U_{B_j}}^T {Q_j}^T A \right\|_F^2$$

$$= \sum_{j=1}^{i} \left\| {U_{B_j}}^T U_{B_j} \Sigma_{B_j} {V_{B_j}}^T \right\|_F^2$$

$$= \sum_{j=1}^{i} \left\| \Sigma_{B_j} \right\|_F^2 = \sum_{i=1}^{k\prime} {\sigma_i'}^2.$$

Hence,

$$\tilde{\varphi}_{k\prime} = \frac{\sum_{i=1}^{k\prime} {\sigma_i'}^2}{\|A\|_F^2} = \varphi' = \frac{\left\| U_L {U_L}^T A \right\|_{\mathrm{F}}^2}{\|A\|_F^2}.$$

□

4.1.2.4 Complexity Analysis

As discussed above, at each iteration, R^{3}SVD carries out orthogonal Gaussian sampling to compute a new t-rank approximation of the leftover subspace orthogonal to the low-rank approximation obtained so far. Suppose that R^{3}SVD uses s iterations to achieve a satisfactory low rank approximation with $k' \approx ts$ as the result rank and assume that the computational cost of matrix-block matrix multiplications dominates those of QR and SVD decompositions on the block matrices. The computational cost of R^{3}SVD is

$$O(2(k' + sp)T_{mult}).$$

In the case that matrix A is sparse and both m and n are large, we can obtain the time complexity with simpler terms. In particular, as $T_{mult} \approx cm$, where c is sparsity ratio, the time complexity of R^{3}SVD can be expressed as $O\left({k'}^2 min(m, n)\right)$.

In additional to the storage of matrix A, the major computations of R^{3}SVD are carried out on a series of block matrices with $(t + p)$ columns or rows. Therefore, since $t < k$, R^{3}SVD takes a constant space complexity of $O\left(2(m + n)(t + p)\right)$, which is lower than that of RSVD, $O\left(2(m + n)(k + p)\right)$.

4.1.3 Numerical Results

In this section, we use several numerical examples to demonstrate the effectiveness of R^{3}SVD for low-rank approximation in image compression and matrix completion.

4.1.3.1 Comparisons with RSVD

We compare the performance of R^{3}SVD, full SVD, Autorank II, restarting RSVD, Adaptive Randomized Range Finder algorithm, and Randomized Blocked algorithm in constructing low rank approximations to compress a 7671×7680 NASA synthesis image chosen from the Mars Exploration Rover mission [107]. A compressed image is considered satisfactory if the low-rank approximation captures 99% energy of the original image matrix.

Both R^{3}SVD and RSVD start with an initial guess $t = 15$ of the target rank and $p = 5$ extra oversampling vectors. In restarting RSVD, as the approximate singular values are available during RSVD, the energy estimation introduced in Section 4.1.2.3 is used. If the guessed rank turns out to be insufficient to obtain a low rank approximation with satisfactory accuracy, the restarting approach repeats the RSVD computation with gradually increasing rank Δt=15. Table 4 compares the computational performance of full SVD, Autorank II algorithm, Adaptive Randomized Range Finder algorithm, Randomized Blocked Algorithm, R^{3}SVD, and restarting RSVD in terms of rank, computational time, maximum memory usage, and energy percentage of the obtained low rank approximation. The optimal low-rank approximation (rank 46) to obtain 99% energy of the original matrix can be obtained by carrying out full SVD, which takes over 760 seconds on a Dell Precision-M6500 laptop (Intel CoreTM i5CPU, 2.67GHz, 4GBRAM). Restarting RSVD reduces the computational time to 13.77 seconds with a low-rank approximation of rank 79. Compared to restarting RSVD, R^{3}SVD further reduces both the computational time to 4.54 (32.97%) and rank to 62 (78.48%). This is because R^{3}SVD carries out important sampling based on the approximate right singular vectors in V_L, which is computed by multiplying A twice per iteration. The power iteration allows more precise estimation of the dominant actions than a single iteration of A multiplication in restarting RSVD. It is also important to notice that the algorithms based on the strategy of estimating k before RSVD, including Autorank II, Adaptive Randomized Range Finder, and

Randomized Blocked algorithm require more computational time as well as the memory than restarting RSVD and R^{3}SVD. The Adaptive Randomized Range Finder uses a probabilistic error estimator, which leads to a highly overestimated rank (641).

TABLE 4
R^{3}SVD, Full SVD, Autorank II, Restarting RSVD, Adaptive Randomized Range Finder Algorithm, and Randomized Blocked Algorithm

	Rank	Computational Time (second)	Maximum Memory Usage (bytes)	Energy Percentage Achieved
Full SVD	46	760.55	1.41×10^9	99.024%
Autorank II Algorithm [104]	105	32.66	2.03×10^7	99.184%
Adaptive Randomized Range Finder Algorithm [33]	641	55.65	1.19×10^8	99.999%
Randomized Blocked Algorithm [105]	105	20.90	4.80×10^8	99.184%
Restarting RSVD	79	13.77	1.60×10^7	99.000%
R^{3}SVD	62	4.54	4.91×10^6	99.006%

Another important advantage of R^{3}SVD is that R^{3}SVD maintains constant memory usage in the computational process. Fig. 20 shows the memory usages in R^{3}SVD and restarting RSVD as the guessed rank gradually increases. One can find that for a larger guessed rank, restating RSVD requires more memory because of decomposing block matrices with more columns or rows. In contrast, the decomposition operations in R^{3}SVD are carried out on block matrices with fixed $(t + p)$ number of columns or rows. As a result, the memory usage does not increase as the guessed rank increases. As shown in Table 4, the memory usage in R^{3}SVD is significantly less than those of the other algorithms.

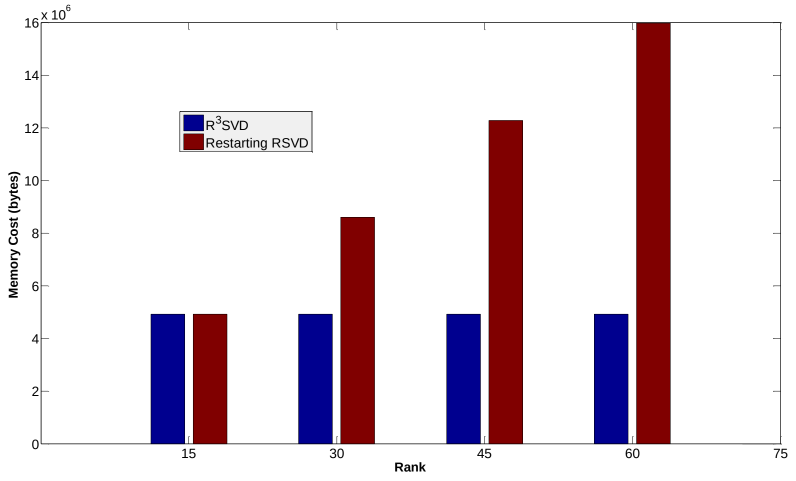
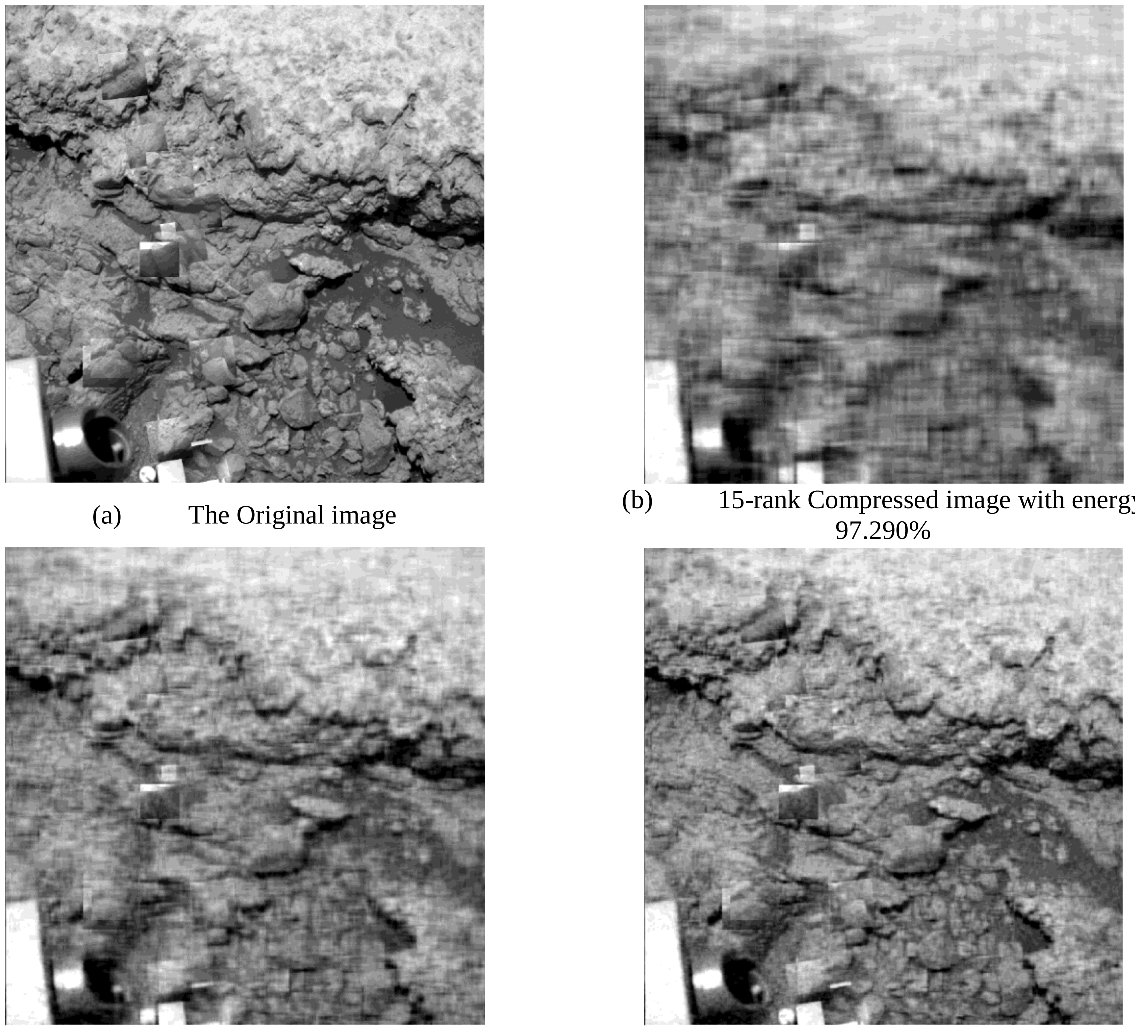


Fig. 20. Memory usage in R^{3}SVD and restarting RSVD

Fig. 21 presents the compressed images in R^{3}SVD, where Fig. 21(a) is the original image and Figs 21(b) to 21(d) illustrate the adaptive compressed images with increasing ranks. With the resulting 62-rank low-rank approximation, a compressed image with 99.006% energy of the original image is obtained.



(a) The Original image

(b) 15-rank Compressed image with energy 97.290%

(c) 30-rank Compressed image with energy 98.410%

(d) 62-rank Compressed image with energy 99.006%

Fig. 21. The original image and the compressed images with increasing ranks in R^{3}SVD

One advantage of R^{3}SVD is that the computational process can be tailored into a series of sampling tasks that can fit into the available memory in a computer via adjusting the sampling size parameter t. Fig. 22 compares the energy percentage of the obtained low rank approximations (upper) and the memory usage (lower) in R^{3}SVD with $t = 20$, 15, 10, and 5. One can find that a smaller sample size in R^{3}SVD yields proportionally less consumption of memory but without significantly affecting the rank in the obtained low-rank approximation. The resulting ranks are 63, 62, 61, and 60, respectively. Therefore, calculating sampling size parameter t according to the available memory in a computer can lead to the best performance of R^{3}SVD.

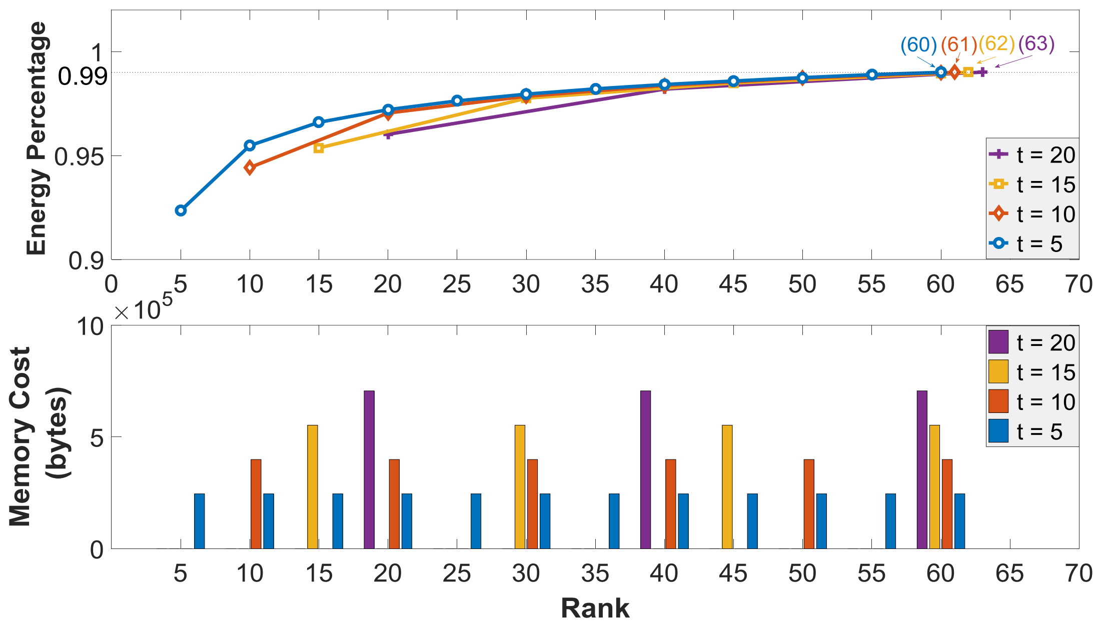


Fig. 22. The energy percentage of the obtained low rank approximations (upper) and the required memory space (lower) in R^{3}SVD with t = 20, 15, 10, and 5 and the oversampling parameter $p = 5$.

4.1.3.2 Application in Matrix Completion

R^{3}SVD can be effectively applied to applications of matrix completion, whose goal is to recover the missing (unknown) entries of an incomplete matrix [25, 108, 109, 110,156]. Matrix completion algorithms have been widely used in many applications, including machine learning [111, 112], computer vision [113], and image/video processing [114]. In the literature, matrix completion algorithms can be classified into two groups. One approach is based on semi-definite programming solvers to find the optimal matrix completion solution [109, 149]. While effective for completing matrices with missing entries, such methods need to solve large-scale systems of linear equations and are not suitable for large problem size. In fact, as pointed out in [25], the methods have difficulty in handling matrices with size

100×100, due to their high computational costs. In contrast, an alternative approach of using Singular Value Thresholding operation offers a rapid way to generate approximations to the exact matrix completion solution [25], which aims to efficiently address large-scale problems. In this class of matrix completion algorithms, low rank matrix approximation is a core component and the computational efficiency of constructing high-quality low rank approximation is essential to their performance.

We modify the Singular Value Thresholding (SVT) algorithm in [25] by replacing the underlying Lanczos algorithm with our R^{3}SVD algorithm to compute dominant singular values and vectors at each SVT iteration. Fig. 23 shows a 1024×1024 aerial image chosen from the USC-SIPI Image Database [115] as well as 10% of the pixels $\mathcal{P}_\Omega(A)$ uniformly sampled from the image (the background is set to grey to highlight these samples), where Ω represents the set of indices of samples and $\mathcal{P}_\Omega(\cdot)$ denotes the operator that sets the entries outside Ω to be zero. As shown in Table 5, the modified SVT algorithm obtains the completed image with similar recovery errors and rank as that of the original SVT. However, R^{3}SVD significantly reduces the computational time in SVT. This is due to the fact that the Lanczos bidiagonalization algorithm with partial reorthogonalization used in original SVT has computational complexity of $O(min(m,n)^2k)$ [28,116] while R^{3}SVD offers a faster way with computational complexity of $O(min(m,n)k^2)$ in contrast. As a result, the modified SVT method using R^{3}SVD achieves about 1.69 times speedup over the original SVT method using Lanczos algorithm.

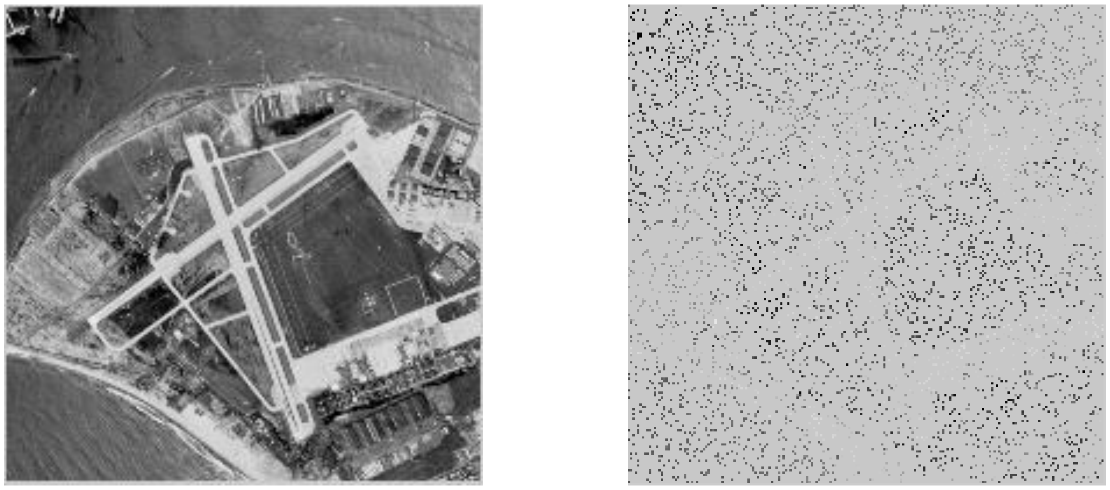
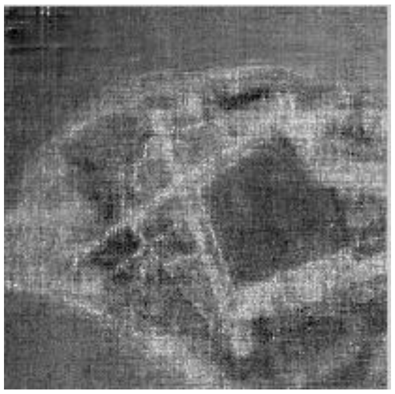
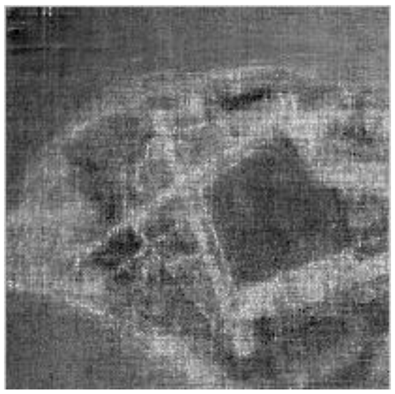

(a) The Original Image A (b) 10% uniform samples $\mathcal{P}_\Omega(A)$

Fig. 23. The original image and the sample image

TABLE 5
The Completed Images using the Original SVT Algorithm and the Modified SVT Algorithm using R^{3}SVD

	Completed Image X	# of Iterations	Elapsed Time (s)	Rank	$\frac{\lVert\mathcal{P}_\Omega(A-X)\rVert_F^2}{\lVert\mathcal{P}_\Omega(A)\rVert_F^2}$	$\frac{\lVert A-X\rVert_F^2}{\lVert A\rVert_F^2}$
Original SVT		822	3457.122	190	9.981×10^{-3}	6.772×10^{-2}
Modified SVT using R^{3}SVD		823	2045.295	189	9.979×10^{-3}	6.772×10^{-2}

CHAPTER V

MONTE CARLO METHODS FOR EXTREME EIGENVALUES/EIGENVECTORS

Finding the dominant eigenvector of a matrix is of great interest in many practical applications. In this chapter, we revisit the generalized block power method for approximating the eigenvector associated with the dominant eigenvalue $\lambda_1 = 1$ of the transition matrix P associated with Markov chain (Section 5.1). The convergence analysis of the block power method shows that when s linearly independent random vectors are used, the block power method converges to the dominant eigenvector at a rate related to the $(s+1)$th dominant eigenvalue $|\lambda_{s+1}|$ of P, rather than the well-known second dominant eigenvalue $|\lambda_2|$ in Markov Chain Monte Carlo theory, which makes the block power method particularly powerful for Markov chains where $|\lambda_{s+1}|$ and $\lambda_1 = 1$ are well separated but $|\lambda_2|$ and $\lambda_1 = 1$ are not.

The block power method requires costly matrix-block multiplications at each iteration. To reduce the computational costs, we design a Sliding Window Power (SWP) algorithm to take advantage of the vectors generated in the previous iterations to build up the block matrices (Section 5.2). The numerical results on a Markov chain application in modeling stochastic luminal Calcium release site are provided to demonstrate the effectiveness of the SWP method.

5.1 Block Power Method

5.1.1 Block Power Iteration

Let P denote an $n \times n$ probability transition matrix of a finite state Markov chain K. Based on the fundamental theorem of Markov chains [7], if K is irreducible, aperiodic, and positive-recurrent, there is an unique stationary distribution characterized by a probability vector π such that

$$\pi^T P = \pi^T.$$

Here π corresponds to the left eigenvector associated with the dominant eigenvalue λ_1= 1 of P. The power method [34] is a simple numerical algorithm that can be applied to approximate the stationary distribution vector of a discrete Markov chain. Starting from a random probability distribution vector x_0, the power method is described by the power iteration

$$x_{i+1} = P^T x_i.$$

The convergence speed of the power method is governed by the second dominant eigenvalue $|\lambda_2|$ of P.

The block power method is a block generalization of the power method. Each iteration step consists of an iteration operation and a decomposition operation [155]. In the iteration operation, subspace iteration (a.k.a. orthogonal iteration or simultaneous iteration) [7, 117] is employed to compute the multi-dimensional invariant subspace. Let X_0 be an $n \times s$ matrix with orthonormal columns and the following subspace iteration process generates a series of matrices X_i.

```
for i = 1, 2, 3, …
    $Z_i = P^T X_{i-1}$
    $X_i R_i = Z_i$        // QR factorization of $Z_i$
end
```

As a result, X_i tends towards the invariant subspace of P with respect to the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_s$. Generally, the subspace iteration method is used to estimate the largest s eigenvalues and eigenvectors of a matrix, which converges at a rate proportional to $|\lambda_{s+1}|/|\lambda_s|$ [7].

The subspace iteration can be tailored for Markov chain applications. Since the dominant eigenvalue λ_1 in the transition matrix P is 1, the normalization step in the subspace iteration process is no longer necessary and the block power method can be simplified as

```
for $i = 1, \ldots, k$
    $X_i = P^T X_{i-1}$
end
```

Due to the fact that the left eigenvector associated with the dominant eigenvalue is only of interest in Markov chain applications, the approximate eigenvector can be extracted from the space spanned by the block matrix X_k by the following Schur-Rayleigh-Ritz step [117] in the decomposition step.

```
$Q_k R_k = X_k$                // QR Decomposition
$B_k = {Q_k}^T P^T Q_k$        // Projection
$U_k T_k {U_k}^T = B_k$        // Schur Decomposition
$Y_k = Q_k U_k$
```

Provided that, in the Schur decomposition of B_k, U_k is chosen so that the diagonal elements of T_k are appeared in non-increasing order of absolute value, ${Y_k}^{(1)}$, the first column vector in Y_k, is an approximation to the dominate left eigenvector v_1 of P.

5.1.2 Convergence Analysis

Theorem 5.2 shows that the block power method converges to the dominant left eigenvector v_1 of P at a rate of $O(|\lambda_{s+1}|^k)$. We first state a special case of a theorem (Theorem 3.2 described in [117]) as Lemma 5.1, which will be used in the proof of Theorem 5.2. When the Markov transition matrix is considered and only the dominant eigenvector is of interest, Lemma 5.1 shows that the Schur-Rayleigh-Ritz approximation to the high powers of an upper triangular matrix converges to the first column of identity matrix.

Lemma 5.1. Suppose that T is an upper triangular matrix where $\lambda_1 = 1,\ \lambda_2,\ \lambda_3, \dots, \lambda_n$ are diagonal elements appearing in non-increasing order of magnitude. Let Y'_k denote the Schur-Rayleigh-Ritz approximation corresponding to $T^k X_0$, where X_0 is an $n \times s$ initial block matrix and k is the number of subspace iterations. Then,

$$dist\left(span\left\{{Y'_k}^{(1)}\right\}, span\{I^{(1)}\}\right) = O(|\lambda_{s+1}|^k).$$

where ${Y'_k}^{(1)}$ and $I^{(1)}$ denote the first column of matrix Y'_k and identity matrix, respectively.

Theorem 5.2. Let Y_k be the Schur-Rayleigh-Ritz approximation corresponding to ${P^T}^k X_0$, where X_0 is an $n \times s$ initial block matrix. Assuming that the Schur decomposition of B_k results in an upper triangular matrix T_k with all diagonal elements sorted in non-increasing order of magnitude, then vector ${Y_k}^{(1)}$ converges to the dominate left eigenvector v_1 of matrix P at a rate that is $\mathrm{O}\left(|\lambda_{s+1}|^k\right)$, such that

$$dist\left(span\{{Y_k}^{(1)}\}, span\{v_1\}\right) = O\left(|\lambda_{s+1}|^k\right).$$

Proof. Suppose that $WTW^{-1} = P^T$ is a Schur decomposition of P^T, where W is an unitary matrix consisting of the Schur vectors and the Schur form T is an upper triangular matrix with its diagonal elements as $\lambda_1 = 1,\ |\lambda_2|,\ |\lambda_3|, \dots, |\lambda_n|$ placed in non-increasing order of magnitude. Then the projection of ${P^T}^k$ on X_0 becomes

$$\begin{aligned} X_k &= {P^T}^k X_0 = (WTW^{-1})^k X_0 \\ &= WT^kW^{-1}X_0 \\ &= T^kW^{-1}X_0 \end{aligned}$$

where $\tilde{X}_0$ denotes $W^{-1}X_0$.

Let Q_kR_k be a QR decomposition of X_k and then we can get $W^{-1}Q_kR_k = T^k\tilde{X}_0$. Because both W^{-1} and Q_S are orthogonal matrices, denoting $Q'_k = W^{-1}Q_k$, we can find that Q'_k is a basis for the range of $T^k\tilde{X}_0$. Afterward, considering projecting P^T onto Q_k to construct an s $\times$ s matrix B_k, we have

$$B_k = {Q_k}^T P^T Q_k = {Q'_k}^T W^{-1}WTW^{-1}WQ'_k = {Q'_k}^T TQ'_k.$$

Clearly, the Schur vectors U_k of B_k are also the Schur vectors of ${Q'_k}^T TQ'_k$. Let Y'_k denote the Schur-Rayleigh-Ritz approximation corresponding to $T^k\tilde{X}_0$. Then, the Schur-Rayleigh-Ritz approximation Y_k to X_k can be expressed as

$$Y_k = Q_kU_k = WQ'_kU_k = WY'_k\,.$$

Based on the assumption that all diagonal elements in T_k are sorted in non-increasing order of magnitude and according to Lemma 1, we have

$$dist\left(span\left\{{Y'_k}^{(1)}\right\}, span\{I^{(1)}\}\right) = O\left(|\lambda_{s+1}|^k\right),$$

and thus,

$$dist\left(span\{{Y_k}^{(1)}\}, span\{W^{(1)}\}\right) = O\left(|\lambda_{s+1}|^k\right).$$

Since $\lambda_1 = 1,\ \lambda_2,\ \lambda_3, \ldots, \lambda_n$ are placed in non-increasing order of magnitude in T, $W^{(1)}$ is exactly the principle left eigenvector v_1 of matrix P. Hence, we can conclude that ${Y_k}^{(1)}$ converges to the principle left eigenvector of matrix P at a rate that is $O(|\lambda_{s+1}|^k)$. □

Theorem 5.2 indicates that when s linearly independent probability vectors are evolved simultaneously under Markov transition, correlating these vectors can lead to a faster approximation of the dominant eigenvector in the block power method. The convergence rate, instead of the well-known one related to λ_2 in the fundamental Markov chain theory, now depends on the $(s+1)$th dominant eigenvalue λ_{s+1} of P. An intuitive explanation of the block power method is that the subspace iteration is able to fast remove the influence from the eigenvalues whose magnitudes are less than $|\lambda_s|$ and the following Schur-Rayleigh-Ritz step is used as a direct method on the resulted block matrix leads to fast approximate the dominant eigenvector. Therefore, the block power method is particularly powerful for Markov chains where $|\lambda_{s+1}|$ and 1 are well separated but $|\lambda_2|$ and 1 are not.

Theorem 5.2 assumes that the upper triangular matrix T_k generated in Schur decomposition of B_k have all diagonal elements sorted in non-increasing order of magnitude. However, this is not always guaranteed in Schur decomposition in practice. Therefore, in the literature, eigendecomposition is applied on B_k instead to generate the Ritz pairs to approximate the eigenvalue/eigenvector pairs.

$$V_k \Lambda_k {V_k}^{-1} = B_k \qquad \text{// Eigendecomposition}$$

$$Y_k = Q_k V_k$$

The Ritz vector corresponding to the largest Ritz value is then outputted as the approximate dominant eigenvector of P. Moreover, as the transition matrix in the Markov chain applications typically has the dominant eigenvalue $\lambda_1 = 1$, the norm $\|P{Y_k}^{(1)} - {Y_k}^{(1)}\|$ is often used as an estimate for the error $dist(span\{{Y_k}^{(1)}\}, span\{v_1\})$ to indicate how well the computed vector approximates the actual dominate eigenvector along power iterations.

5.1.3 Numerical Results

We firstly use a simple Markov chain with five states as an example to illustrate how the block power method can gain the convergence acceleration. Then, a more "realistic" transition matrix from a Markov chain application [148] is examined to demonstrate the applicability of the block power method. In both examples, Gaussian random vectors are generated as the initial vectors, and the tolerance of convergence is set to 10^{-7}.

5.1.3.1 A Simple Example

In this work, consider a Markov chain with 5 states $\{1,2,3,4,5\}$, as shown in Fig. 24.

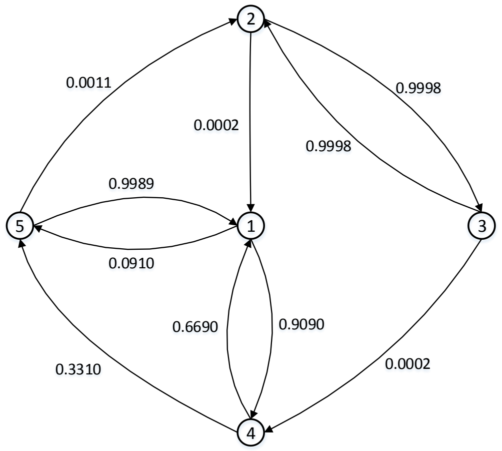


Fig. 24. A Markov chain with five states

where the corresponding transition matrix P is formed as

$$P = \begin{bmatrix} 0 & 0 & 0 & 0.9090 & 0.0910 \\ 0.0002 & 0 & 0.9998 & 0 & 0 \\ 0 & 0.9998 & 0 & 0.0002 & 0 \\ 0.6690 & 0 & 0 & 0 & 0.3310 \\ 0.9989 & 0.0011 & 0 & 0 & 0 \end{bmatrix}.$$

Fig. 25 displays the numerical results of using the power method as well as the block power method with block size 2 and 3 to compute the distribution over states. One can notice that once three linearly dependent vectors are used, the convergence of block power method to the stationary distribution can be significantly accelerated. The block power method with block size 3 requires only 28 iteration steps to converge, which is much less than those of the power method (78,813) and the block power method with block size 2 (18,826). This is consistent with the convergence rate analyzed in Theorem 5.2.

The distribution of the eigenvalues in the transition matrix P is $|\lambda_1| = 1$, $|\lambda_2| = 0.9998$, $|\lambda_3| = 0.9996$, $|\lambda_4| = 0.5483$, and $|\lambda_5| = 0.5483$. Due to the fact that $|\lambda_1|$, $|\lambda_2|$, $|\lambda_3|$ are clustered but $|\lambda_1|$and $|\lambda_4|$ are well separated, the block power method block size 3 converges much faster than the power method and the block power method with smaller block size .

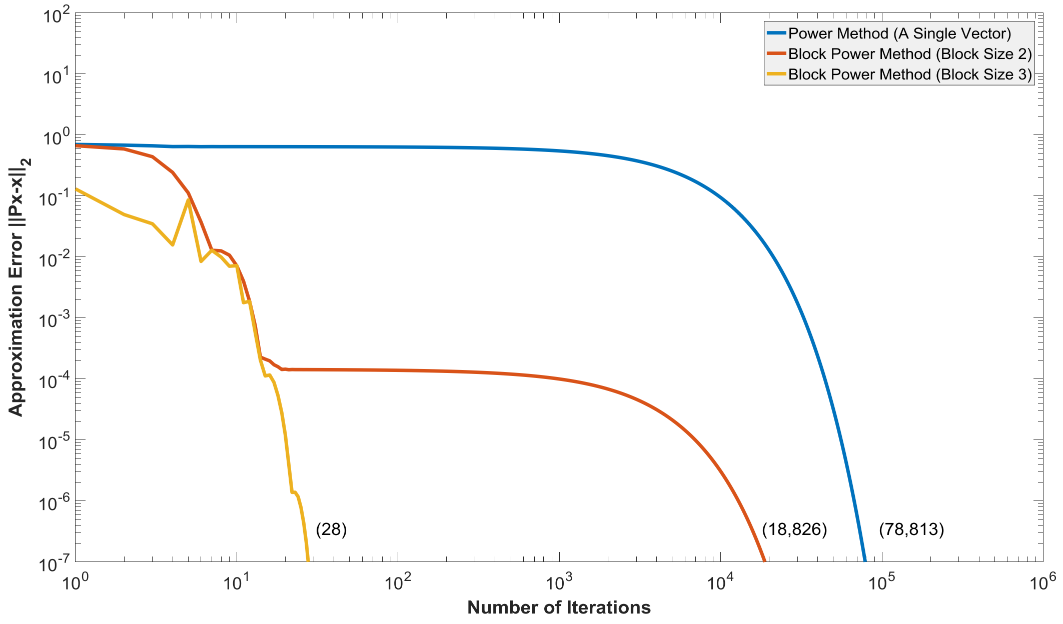


Fig. 25. Convergence comparison of the power method and the block power method (block size 2 and 3) in terms of number of iterations

5.1.3.2 Example with a Larger Matrix

We apply the block power method to a $16{,}968 \times 16{,}968$ transition matrix arisen from a Markov chain application in modeling stochastic luminal Calcium release site [118]. Fig. 26 compares the computational results of the simple power method and the block power method with block size $k = 5$ and 10. One can find that using the strategy of correlating multiple linearly independent vectors in block power method has the potential to reduce the number of iteration steps to reach an approximate dominant vector with the desired accuracy. For example, the block power method with block size 5 (63,444 steps) requires fewer iteration steps to converge to 10^{-7} than the power method (164,454 steps). Further reduction is gained when larger block sizes are used. As shown in Fig. 26, when a relatively large block size 10 is employed, the number of iterations needed is further reduced down to 34,340 steps.

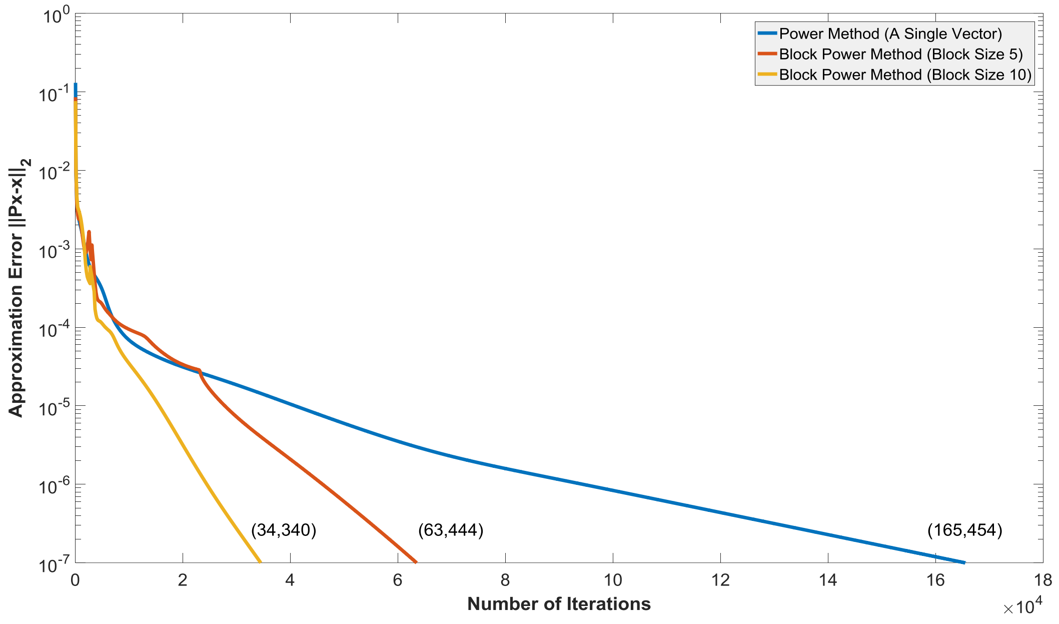


Fig. 26. Convergence of the power method and the block power method (k = 5 and 10) on a transition matrix of size

5.2 Sliding Window Power (SWP) Method

Despite the faster convergence rate, one of the main computational concerns in the block power method is that each block power iteration requires s matrix-vector multiplications, where s is the block size. For very large matrices that the element blocks are stored across distributed devices, the computation and memory requirements of the block power method are much higher than those of the simple power method. In this section, we describe a Sliding Window Power (SWP) method that we design to take advantage of the subsequent vectors within last s iterations to build up the block matrix, while avoiding the costly matrix-block multiplications in the block power method.

5.2.1 The SWP Algorithm

The fundamental idea of the SWP method is to take advantage of the intermediate subsequent vectors in simple power iterations to form the multi-dimensional invariant subspace as follows,

for $i = 1, \ldots, k$

```
        $x_i = P^T x_{i-1}$
    end
    $W_k = [x_{k-s+1}, x_{k-s+2}, \dots, x_k]$
```

where s denotes the window size of W_k. In fact, the sliding window matrix represents a truncated Krylov subspace based on x_0, i.e.,

$$W_k = [x_{k-s+1}, x_{k-s+2}, \dots, x_k] = \left[{P^T}^{k-s+1}x_0, {P^T}^{k-s+2}x_0, \dots, {P^T}^{k}x_0\right].$$

Then, the eigendecomposition step can be carried out on ${W_k}^T P^T W_k$ to obtain the Ritz pairs and the approximated dominate eigenvector is extracted accordingly.

```
    $Q_k R_k = W_k$                    // QR Decomposition
    $B_k = {Q_k}^T P^T Q_k$            // Projection
    $V_k \Lambda_k {V_k}^{-1} = B_k$   // Eigendecomposition
    $Y_k = Q_k V_k$
```

SWP expects to yield fast convergence as block power methods. However, due to the fact that these intermediate vectors in the truncated Krylov subspace are highly correlated, the convergence rate of SWP depends on the actual rank of W_k. In fact, the convergence speed of SWP with window size s lies between $|\lambda_{s+1}|^k$ and $|\lambda_2|^k$. In the best case, SWP will have the similar convergence rate related to λ_{s+1} as the block power method. If the rank of W_k is 1, the performance of SWP is downgraded to the simple power method. Nevertheless, if this actually occurs, the power iteration should have already converged. While SWP has approximately equivalent computational cost compared to simple power method, in practice, SWP is usually more efficient than the block power method in terms of the computational cost.

5.2.2 Numerical Results

We apply SWP to compute the stationary distribution of the $16{,}968 \times 16{,}968$ transition matrix described in Section 5.1.3.2. Fig. 27 compares the performance of power method, block power method with block size 10, and SWP with window size 10 in terms of the number of iterations. As illustrated in

Fig. 27, one can find that the convergence trajectory of the SWP method lies in-between the block power method and the power method. This is because the convergence rate of the sliding window power method is theoretically bounded by that of the block power method. By reusing the previously generated vectors in the power iterations to form the block matrix, SWP gains convergence acceleration and reduces iteration steps to 96,232, which is fewer than that of the simple power method (165,454).

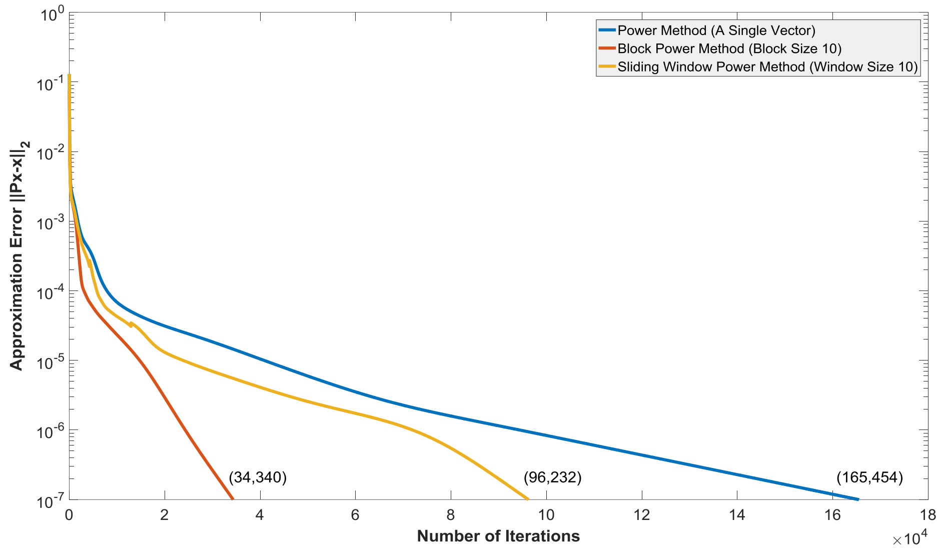


Fig. 27. Convergence comparison of Power method, Block Power method, and Sliding Window Power method in terms of number of iterations

One advantage of the SWP method is that it is possible to reduce the overall number of matrix-vector multiplications needed to obtain the stationary distribution vector with satisfactory accuracy, which is particularly favorable for Markov chain applications with very large transition matrices, where the matrix-vector multiplication operations dominates the computational cost. Fig. 28 shows their comparison in terms of the number of matrix-vector multiplications. The SWP method, benefited from the accelerated convergence rate of the block form and matrix-vector multiplication per iteration, requires fewer matrix-vector multiplications (96,232) than the others (165,454 and 343,400, respectively.)

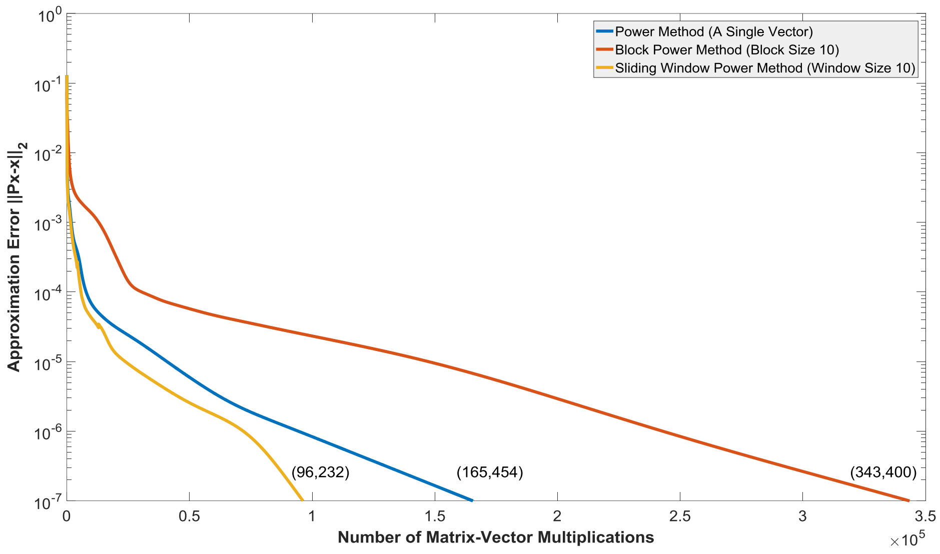


Fig. 28. Convergence comparison of Power method, Block Power method, and SWP in terms of number of matrix-vector multiplications

Increasing the number of vectors in the block has the potential to improve the convergence rate of SWP and further reduce the number of Matrix-vector multiplications needed to reach convergence. However, with a larger number of vectors in the block, more memory storage would be required in SWP, as shown in Fig. 29. Thus, in practice, the appropriate window size of W_k should be selected according to the memory available.

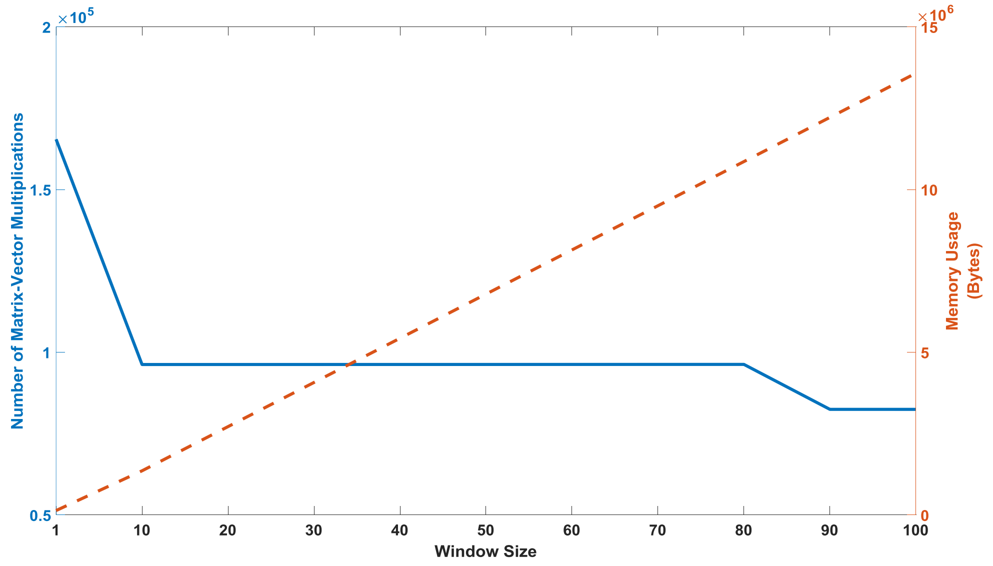


Fig. 29. The number of matrix vector multiplications and the memory usage to convergence in SWP using different window size values

CHAPTER VI

HYBRID CPU-GPU ACCELERATION OF MONTE CARLO ALGORITHMS

Modern many-core devices, such as Graphics Processing Units (GPU) and Intel Xeon Phi processors, are capable of delivering higher computing power than multi-core CPUs. This has led to increasing interest in using GPU or Intel Xeon Phi as coprocessors (accelerators) to enable additional accelerations to scientific computations carried out on a host system. For instance, once a many-core device is attached to the host system, intensive computational operations can be offloaded to the many-core hardware during execution, which is referred to as the “offload mode” [119,120]. In this Chapter, we take advantage of the GPU accelerators to improve the performance of two Monte Carlo algorithms, BFBCG (Section 6.1) and RSVD (Section 6.2).

6.1 Accelerating BFBCG

We first analyze the performance of various matrix operations of BFBCG in a GPU-only implementation to identify the main performance bottleneck. Then, to handle large linear systems whose coefficient matrices cannot fit in the GPU memory, a hybrid (offload) computing scheme is presented to offload compute-intensive matrix operations to GPU processors and to hide the CPU-GPU memory transaction overhead. Finally, we compare the performance of our BFBCG implementation on CPU-GPU processors with the one on CPU with Intel Xeon Phi as coprocessor using the automatic offload mode.

The computational experiments described in this work are carried out on the XSEDE TACC Stampede System [83], where the compute node has dual Intel Xeon E5-2680 CPUs sharing 32 GB memory, one Intel Xeon Phi SE10P Coprocessor with 8GB memory, and one NVIDIA K20 GPU with 5GB memory. The BCG program is compiled using the Intel icc compiler with “-O3” optimization flag on CPU and Intel Xeon Phi processors while using NVIDIA nvcc compiler with "-O3" flag on GPU.

6.1.1 BFBCG on GPU

We investigate the native implementation of BFBCG (Algorithm 3.2 in Section 3.2.3) on GPU processors, where all numerical operations are carried out on GPU and the coefficient matrix also resides in the GPU memory. This implementation uses the matrix functions in the CUDA Basic Linear Algebra Subroutines (CUBLAS) library [38] for dense matrix operations, advanced matrix decompositions functions in the MAGMA library [40] for Cholesky factorizations, and sparse matrix routines in the CUSPARSE library [13] for sparse matrix operations. For comparison purposes, a CPU implementation of BFBCG is built using the multithreaded Intel Math Kernel Library (MKL) [121].

Fig. 30 compares the average elapsed computational time per iteration for different matrix operations in BFBCG on CPU and GPU processors. The coefficient matrix is "nd12k" from the UFL sparse matrix collection [77], which is a $36{,}000 \times 36{,}000$ sparse, SPD matrix with $14{,}220{,}946$ nonzero entries. The number of right hand sides is set to $2{,}048$. The elements in the right-hand side matrix are random numbers generated uniformly from interval [0, 1). The reported execution times are obtained from an average over 10 runs.

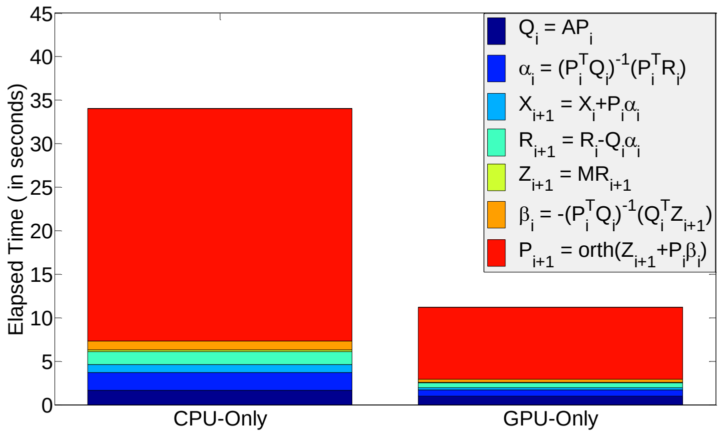


Fig. 30. The average elapsed computational time for different steps in BFBCG on CPU and GPU processors

One can notice that the computational times of all matrix operations per iteration in BFBCG on GPU are less than those on CPU, where the improvements of tall-and-skinny matrix operations are of most significance. Nevertheless, the dominating operation in both CPU and GPU implementations is constructing the new search direction matrix P_{i+1}, i.e., $P_{i+1} = orth(Z_{i+1} + P_i\beta_i)$.

To reduce the computational cost in the constructing new search direction matrix P_{i+1}, we modify the BFBCG algorithm by using eigendecomposition on $\mathrm{Z}^{\mathrm{T}}Z$, where $Z = Z_{i+1} + P_i\beta_i$, instead. In this case, $\mathrm{Z}^{\mathrm{T}}Z$ is a small $s \times s$ symmetric matrix. Therefore, although calculation of $\mathrm{Z}^{\mathrm{T}}Z$ leads to additional overhead of matrix-matrix multiplications, computing the eigendecomposition on $\mathrm{Z}^{\mathrm{T}}Z$ is still significantly less costly than directly applying DGEQP3 to the $n \times s$ tall-and-skinny matrix Z for QR decomposition. As shown in Fig. 31, the eigenvectors V of $\mathrm{Z}^{\mathrm{T}}Z$ can be computed by using the DSYEVD routine. Once the eigenvectors V is available, the search matrix P_{i+1}, as an orthogonal basis of the space spanned by Z, can be very efficiently derived by normalizing each column of matrix product ZV using DNRM2 routine.

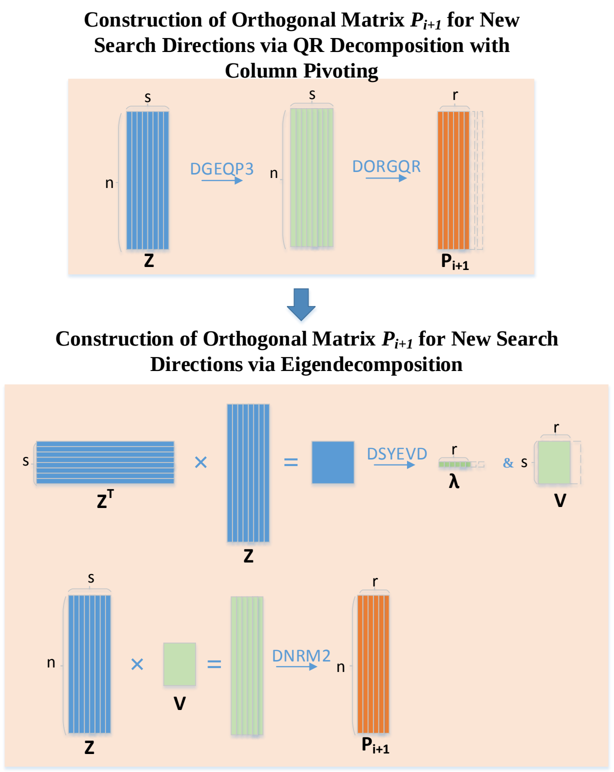

Construction of Orthogonal Matrix P_{i+1} for New Search Directions via QR Decomposition with Column Pivoting
s
n
Z
DGEQP3
s
n
DORGQR
r
P_{i+1}
Construction of Orthogonal Matrix P_{i+1} for New Search Directions via Eigendecomposition
s
Z^T
×
Z
=
DSYEVD
r
λ
& s
r
V
s
n
Z
×
V
=
DNRM2
n
r
P_{i+1}

Fig. 31. Eigendecomposition on $\mathrm{Z}^{\mathrm{T}}Z$ to replace QR decomposition on Z to obtain orthogonal new search direction matrix P_{i+1}

Fig. 32 shows the performance of the improved BFBCG implementation using eigendecomposition on $\mathrm{Z}^{\mathrm{T}}Z$ to obtain new search directions. In comparison with Fig. 30, one can find that the time spent on constructing new search direction matrix P_{i+1} is significantly reduced by 60.7% and 73.5% on CPU and GPU implementations, respectively. The overall speedup of the GPU-only implementation over the CPU implementation reaches 2.63.

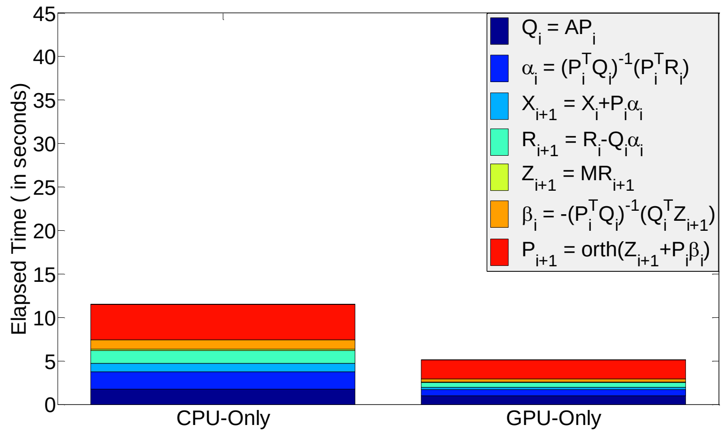


Fig. 32. Comparison of the average elapsed computational time per iteration for different steps in BFBCG on CPU and GPU processors when eigendecomposition on Z^TZ is used to replace QR decomposition on Z to obtain search direction matrix P_{i+1}

6.1.2 BFBCG on Hybrid CPU-GPU

In the case that the coefficient matrix is too big or the number of right hand sides is too many, consequently, the GPU memory is not big enough to fit all the matrices in BFBCG iterations. In this section, a BFBCG implementation on hybrid CPU-GPU processors is presented. In our implementation, CPU only coordinates data transfer and computation offload to GPU and does not directly participate in BFBCG computation. We use the routines in the CUBLAS-XT library [38] to support overlapping data transfers and execution for dense matrix operations. Page-locked memory is employed to increase the bandwidth between host memory and GPU memory.

Based on the sparse matrix routines in CUSPARSE, we implement the tiled multiplication between a sparse matrix and a tall-and-skinny matrix. Similar to the tiling strategy used in the CUBLAS-XT library, rows of sparse matrix is partitioned into tiles that can fit in the GPU memory while the tall-and-skinny matrix is split into tiles by columns. The tile size is selected so that the tiles can fit in the GPU memory. The procedure of tiling is illustrated in Fig. 33.

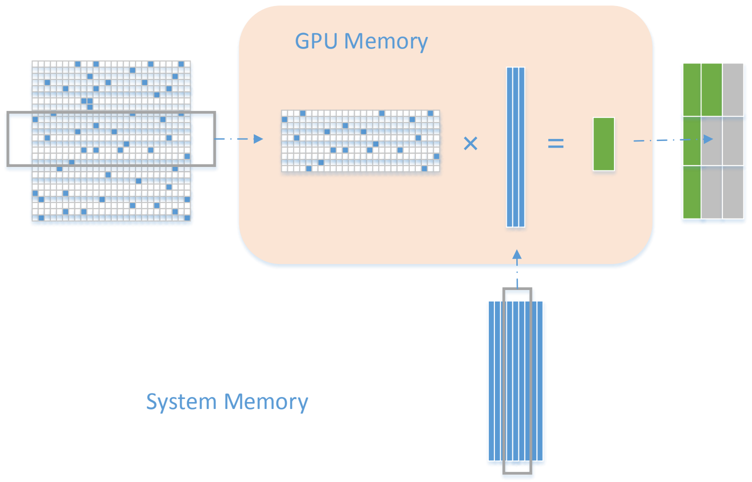
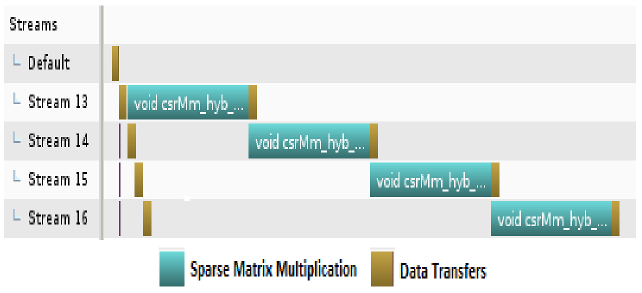
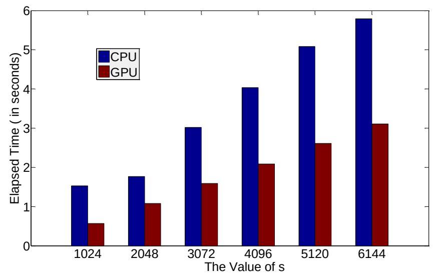


Fig. 33. Tiled multiplication between a sparse matrix and a tall-and-skinny matrix

An important feature of the hybrid CPU-GPU BFBCG implementation is that data transfers and kernel computation for each tile can be performed concurrently so that memory transaction time can be hidden. We assign each tile with a GPU stream, and asynchronous operations are placed into each stream. Fig. 34 shows timeline of sparse matrix multiplication and data transfer in an instance of calculating the product of the sparse coefficient matrix and the tall-and-skinny solution matrix, and the elapsed computational time at different block sizes. One can find that except for initialization, more than half of data transfer operations occur concurrently with matrix multiplications, which can be hidden efficiently.





(1) Overlap of computing and data transferring (2) The elapsed computational time

Fig. 34. Data transfers and kernel computation for each tile are performed concurrently to hide the memory transaction time between CPU and GPU

Fig. 35 shows the elapsed computational time per iteration in hybrid CPU-GPU BFBCG implementation in comparison with the GPU-only computational time and data transfer time without overlapping. In hybrid CPU-GPU scheme, 50.1% of the data transfer time is hidden due to concurrent execution with matrix operations.

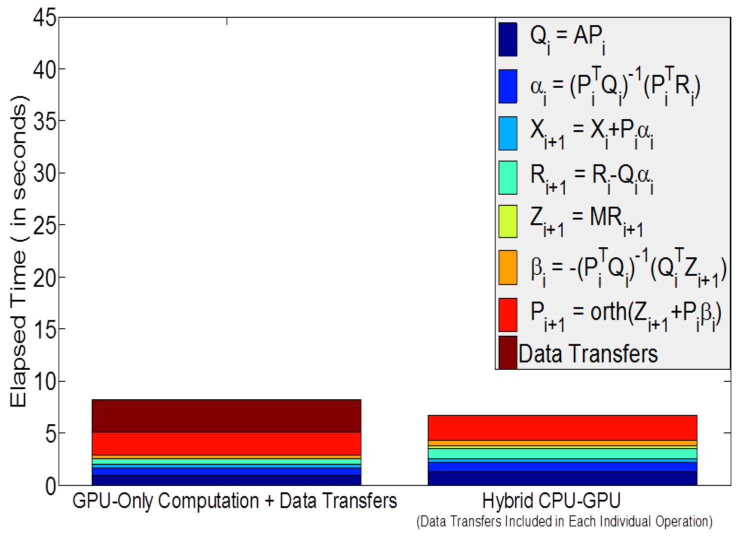


Fig. 35. Comparison of the elapsed computational time per iteration in hybrid CPU-GPU BFBCG implementation with the GPU-only computational time and data transfer time. 50.1% of the data transfer time is hidden in the hybrid CPU-GPU scheme.

6.1.3 Computational Results

In this section, we compare the performance of the hybrid CPU-GPU implementation of BFBCG with the BFBCG implementation on CPU-Xeon Phi Processor using automatic offload mode against their theoretical performance peak, where MKL library provides the optimal computational work division for matrix operations of BFBCG over CPU- Xeon Phi Processor. The theoretical peak performance is widely used as upper bound in comparing computational power among parallel computing systems [122]. For a certain parallel computing system, the corresponding theoretical peak double precision performance P can be calculated as

$$P = ncores \times clockspeed \times flops/cycle$$

where $ncores$ represents the number of cores in a processor, $clockspeed$ is the corresponding clock rate, and $flops/cycle$ denotes the number of double-precision floating point operations per cycle [123, 124, 125].

Each Dual Xeon E5 processor has 8 cores clocked at 2.7GHz. Because the Dual Xeon E5 processor supports the Fused Multiply-Add (FMA) operations, in which one multiply and one add can be completed in a single cycle, each core of Dual Xeon E5 can perform up to 8 double-precision floating point operations per clock cycle. As a result, the theoretical peak double precision performance P_{cpu} of CPU can reach

$$P_{cpu} = (8 \times 2) \times 2.7 \times 8 = 345.6\ GFLOPS$$

The NVIDIA K20 GPU [126,127] has 13 Streaming Multiprocessors (SMs) clocked at 0.706GHz while 64 double-precision floating point units on each SM. The theoretical peak double precision performance P_{gpu} of GPU is calculated as

$$P_{gpu} = (64 \times 13) \times 0.706 \times 2 = 1{,}174.784\ GFLOPS$$

For the 61-core coprocessor Xeon Phi SE10P, each core clocked at 1.1GHz has 16 floating-point operations in double precision per clock cycle. As 60 cores are commonly used for computing, the theoretical peak performance P_{mic} of Xeon Phi coprocessor is

$$P_{mic} = 60 \times 1.1 \times 16 = 1{,}056\ GFLOPS$$

Ideally, if the linear algebra routines for those matrix operations in BFBCG can fully take advantage of the peak performance on hardware while the memory transaction overheads are hidden, executing BFBCG implementation directly on GPU or Intel Xeon Phi can roughly outperform CPU-only version by three times, according to the theoretical peak performance analysis on these hardware devices.

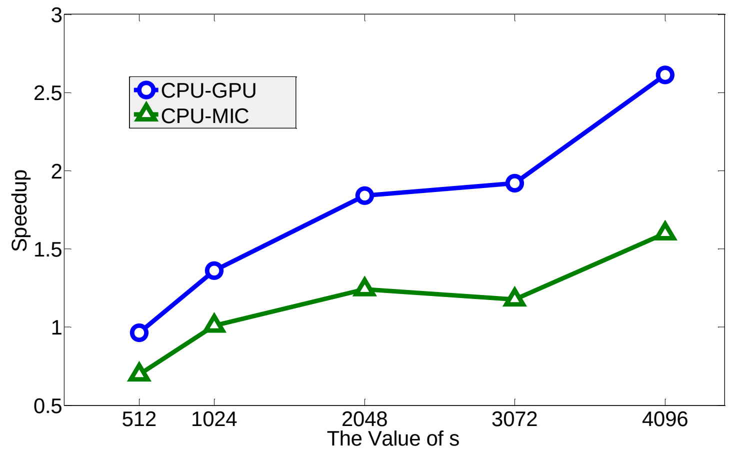


Fig. 36. The overall speedup of CPU-GPU and CPU-Xeon Phi of BFBCG implementations with different number of right hand sides

We use a large linear system with “thermomech_TC” from the UFL sparse matrix collection [77] as the coefficient matrix to test the CPU-GPU and CPU-Xeon Phi implementations of BFBCG.

“Thermomech_TC” is a $102{,}158 \times 102{,}158$ sparse, SPD matrix with $711{,}558$ nonzero entries. Fig. 36 compares the overall speedup factors for the CPU-GPU implementation and the CPU-MIC implementation of BCG algorithm over the CPU-Only version with different number of right hand sides s. The overall speedup of CPU-GPU can reach up to 2.61 when 4,096 right hand sides are used, which is significantly higher than that of CPU-Xeon Phi (1.61) in automatic offload mode.

6.2 Accelerating RSVD

We use the randomized SVD algorithm with Gaussian Sampling (RSVD) as an example to illustrate our hybrid CPU-GPU accelerated implementation. First of all, we present a GPU-accelerated implementation to quickly obtain the approximate of dominant singular components of a given large matrix. Noticing that the main bottleneck in the GPU implementation is the deterministic SVD on GPU with "short-and-wide" matrix, we apply SVD decomposition on a derived square matrix to reduce the overall computational time. Then, in the case of matrices with a small dominant rank k value, a hybrid GPU-CPU scheme is carried out to further improve the efficiency of our implementation.

6.2.1 RSVD on GPU

Fig. 37 shows the procedure of the RSVD algorithm (Algorithm 4.1 in Section 4.1.1). The overall performance of RSVD depends on the efficiency of matrix-matrix multiplication, QR factorization, and SVD on small matrices. Fortunately, after random matrix sampling by Ω, the large matrix A is condensed into either "tall-and-skinny" or "short-and-wide" matrix, such as Y and Q are $m \times (k+p)$ "Tall-and-skinny" matrices, B is an $(k+p) \times n$ "short-and-wide" matrix where $k+p$ is much smaller than $min(m,n)$. These small and dense matrices are particularly suitable fit in GPU memory to take advantage of high-performance computation provided. We implemented RSVD on GPU using CUBLAS [38] and CULA [39], and its corresponding CPU version using the Intel multi-thread MKL (Math Kernel Library) for the sake of performance illustration.

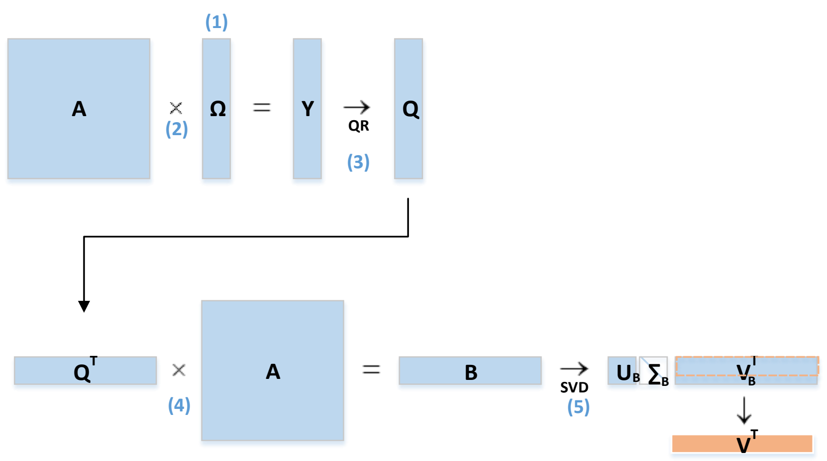


Fig. 37. Procedure of RSVD to approximate right-singular vectors

The elapsed time spent on each primary computational component in randomized SVD is shown in Fig. 38for a 4,096 × 4,096 random matrix where k is 128 and p is 3. Multiplication between A and a "tall-and-skinny" or "short-and-wide" matrix can be efficiently carried out on the GPU's SIMT architecture and hence the computational time in generating matrix Ω and performing matrix-matrix multiplications shrinks to nearly negligible. Nevertheless, deterministic SVD, particularly when the target matrix is small, has difficulty in fully taking advantage of GPU architecture, due to the fact that a series of sequential Householder transformations need to be applied. As a result, deterministic SVD becomes the main bottleneck and thus this GPU implementation has only 1.65 over that of the CPU.

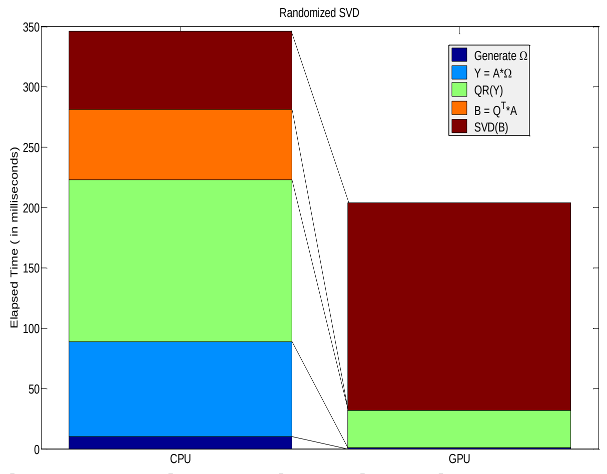


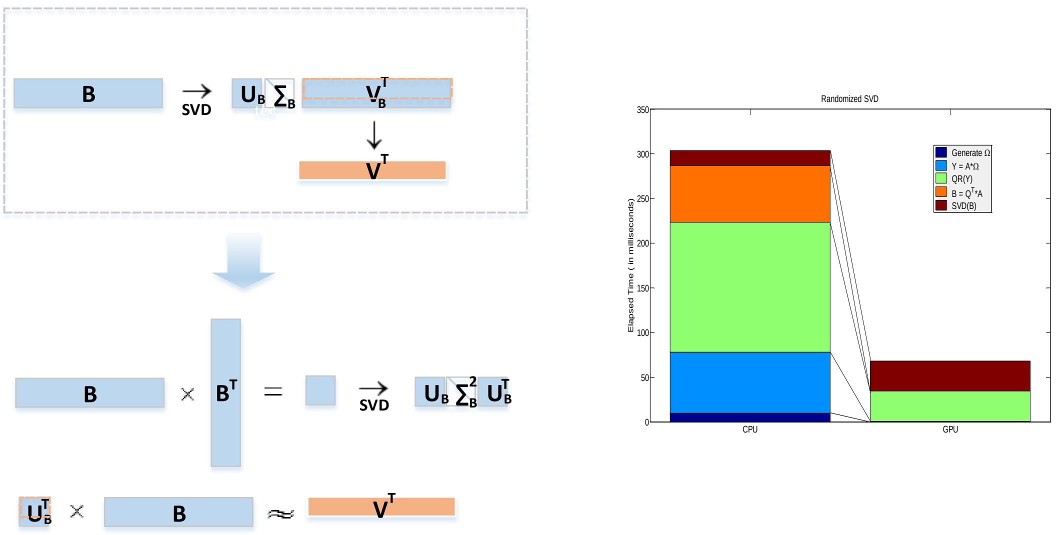
Fig. 38. The elapsed computational time used in randomized SVD on CPU-only and GPU-only

6.2.2 RSVD on Hybrid GPU-CPU

To reduce the computational cost of deterministic SVD in GPU randomized SVD implementation, we alternatively calculate the top-k singular vectors of BB^T instead of directly carrying out deterministic SVD on the "short-and-wide" matrix B. Fig. 39 (1) depicts the procedure of obtaining approximate SVD decomposition of B. Note that SVD decomposition of B is defined as $B = U_B \Sigma_B V_B^T$. Since BB^T is a small square matrix whose size is independent of the size of the original matrix A, and has SVD format as,

$$BB^T = U_B \Sigma_B U_B^T,$$

U_B could be very efficiently derived from BB^T rather than from B.



(1) Procedure of using BB^T (2) Elapsed time on CPU-only and GPU-only

Fig. 39. Obtaining approximate SVD decomposition of B

Once the left singular vectors U_B become available, under the assumption that $U_B^T U_B \approx I$, where I is an identity matrix, the top k singular components could be approximated effectively through a single matrix-matrix operation

$$U_B^T B \approx \Sigma_B V_B^T.$$

Fig. 39 (2) shows the elapsed time of the improved implementation by using BB^T on the same 4,096 × 4,096 random matrix used in Fig. 38. One can find that the portion of SVD computation time is significantly reduced on both CPU and GPU implementations. Consequently, the achieved speedup of GPU implementation grows up to 4.6.

As shown in Fig. 39, even though the alternative approach of approximating top-k singular values/singular vectors on BB^T is used, the computational time of deterministic SVD on GPU is still more than that of the CPU version due to hidden setup on GPU. To further understand the performance of deterministic SVD on GPU, we compute deterministic SVD to a set of square matrices varying in size. Fig. 40 compares the computational time of deterministic SVD on CPU and GPU. One can find that the CPU implementation outperforms the GPU one on small matrices less than $2{,}500 \times 2{,}500$. Therefore, using GPU to run SVD operations on small matrices is not appropriate, particularly for applications where the singular values decay very quickly and k is typically set with very small value. A simple hybrid GPU-CPU scheme is employed in our implementation that when the $k \times k$ square matrix is small, deterministic SVD decomposition will be transferred to the CPU to carry out instead.

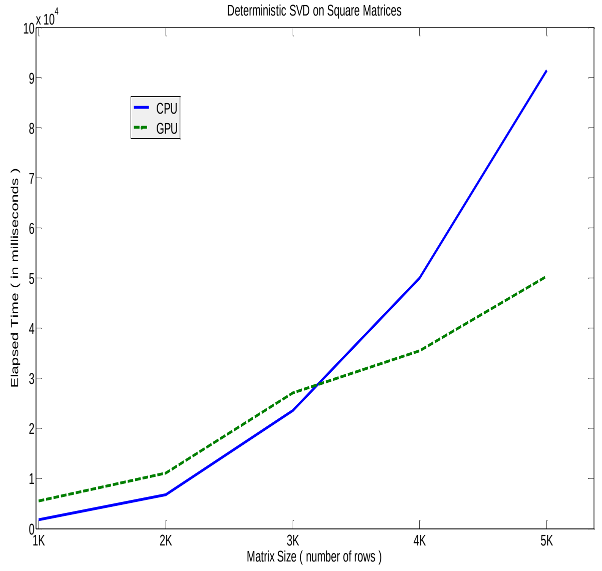


Fig. 40. Comparison of running time for performing deterministic SVD on GPU and CPU

6.2.3 Computational Results

We present the numerical results obtained with GPU-accelerated implementation on large random matrices and a Mars image. The experiments are carried out on a Linux computer with an Intel Core i5-2500K CPU 3.30GHz, 8GB of RAM, and an NVIDIA GK110GL GPU.

6.2.3.1 Random Matrices

We generate a series of large random dense matrices of varying sizes to benchmark the performance achieved by using our GPU-accelerated randomized SVD algorithm. Fig. 41 compares the computational time in logarithmic scale of performing complete SVD and randomized SVD on CPU as well as GPU-accelerated randomized SVD algorithm. The same k and p ($k = 256$ and $p = 3$) values are used. Compared to doing the complete SVD calculation on the matrix, randomized SVD has a clear computational advantage when only the top-k approximated singular components are needed. Similar to many other GPU-based algorithms, our GPU randomized SVD implementation favors larger matrices. For a $20{,}000 \times 20{,}000$ matrix, the speedup can reach up to 6~7.

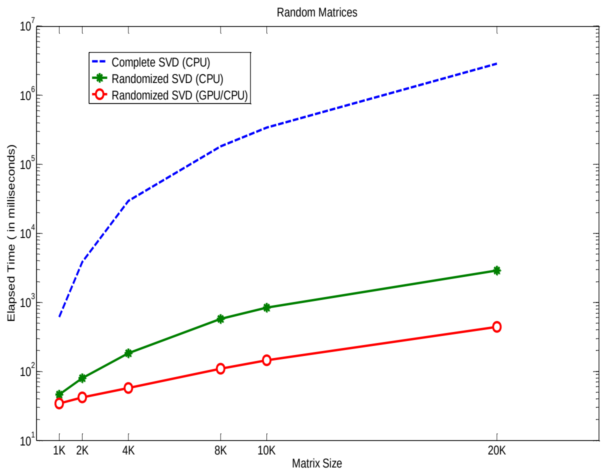


Fig. 41. Comparison of elapsed time (logarithmic scale) of deterministic SVD, CPU versions of RSVD and GPU-accelerated RSVD

6.2.3.2 Image Compression

We apply the randomized SVD algorithm for lossy data compression to a NASA synthesis image from the Mars Exploration Rover mission [107] shown in Fig. 42. The image is an RGB $7671 \times 7680 \times 3$ matrix, which requires 176.74 million bytes for memory storage.

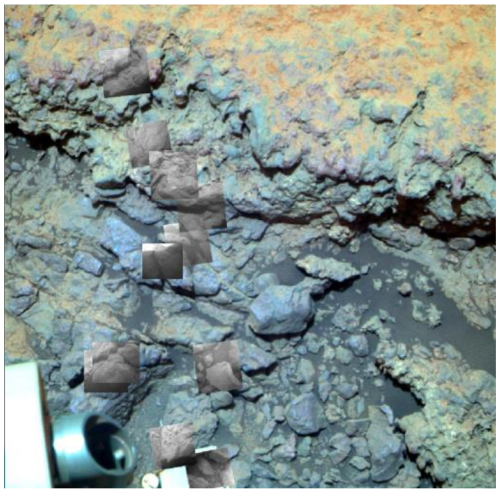


Fig. 42. The original image

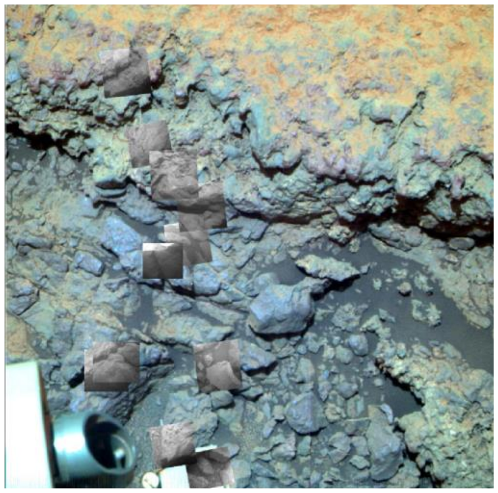


Fig. 43. The reconstructed image

In order to compress the image, we use our GPU-accelerated implementation to obtain its low rank approximation A_k with rank 470,

$$A_k = M \times N$$

where M is a 7671×470 matrix and N is a 470×7680 matrix on each color channel (R,G,B). Fig. 43 shows the reconstructed image, where M is computed by combining the 470 left singular vectors with the corresponding singular values while N is stored as the 470 right singular vectors as columns. To outline the effectiveness of our implementation of randomized SVD, Table 6 lists the elapsed computational time and error used in compression with the Mars Image. As one can find, compared to deterministic SVD which consumes more than one thousand seconds to obtain the top 470 approximation, the GPU-accelerated randomized SVD only takes slightly more than one second. The overall storage of the decomposed image requires less than 1/8 of that of the original matrix with an acceptable 1.63% error.

TABLE 6
Elapsed Computational Time and Error in Compression with the Mars Image

	Elapsed Time (in seconds)	Error in Compression
Deterministic SVD	1144.71	1%
Randomized SVD	1.29	1.63%

CHAPTER VII

MATRIX PRODUCT VERIFICATION

When matrices are very large, potential memory errors can no longer be neglected in large-scale linear algebra operations on high-performance computing (HPC) architectures. In this Chapter, we propose a Gaussian variant of Freivalds' algorithm (GVFA) to verify the correctness of matrix-matrix multiplication (Section 7.1). Our theoretical analysis shows that when $A \times B \neq C$, the chance of GVFA produces $C\omega_G = AB\omega_G$ occurs has measure zero in exact arithmetic. We also analyze false positive probabilities in the GVFA, by taking floating point round-off error into account. In Section 7.2, we provide further discussions about potential advantages of GVFA in enhancing the resilience of linear algebraic computations.

7.1 Gaussian Variant of Freivalds' Algorithm (GVFA)

7.1.1 The GVFA Algorithm

The original Freivalds' algorithm (see Section 2.4.2), and most of its extensions are based on integer matrices or matrices over a ring and sampling from discrete spaces. In this work, we extend Freivalds' algorithm by using Gaussian random vectors for the projection [152]. We use the fact that the multivariate normal distribution has several nice properties [138], which have been used for detecting statistical errors in distributed Monte Carlo computations [139]. The extended algorithm is described in Algorithm 7.1, which requires three matrix-vector multiplications, and only one vector comparison for fault detection.

```
Algorithm 7.1: Gaussian variant of Freivalds' algorithm (GVFA)
Step 1. Generate a Gaussian random vector, $\omega_G$, where $\omega_G$ is an n-vector of independent (but not necessarily identically) distributed normal random variables with finite mean and variance.
Step 2. Calculate the projection of $C$ on $\omega_G$: $C\omega_G = C \times \omega_G$.
Step 3. Calculate the projection of product $A \times B$ on $\omega_G$: $AB\omega_G = A \times (B \times \omega_G)$.
```

7.1.2 Theoretical Justification

Similar to Freivalds' algorithm, in GVFA if $A \times B = C$, $C\omega_G = AB\omega_G$ always holds within a certain floating point round-off threshold. When $A \times B \neq C$, the chance that $C\omega_G = AB\omega_G$ is a false positive event occurs with measure zero in exact arithmetic, as shown in Theorem 7.2.

We first state a result of Lukacs and King [140], shown as Proposition 7.1, which will be used in the proof of Theorem 7.2. The main assumption of Proposition 7.1 is the existence of the nth moment of each random variable, which many distributions, particularly the normal distribution, have. One important exception of the normal is that it is the limiting distribution for properly normalized sums of random variables with two finite moments. This is Lindeberg's version of the Central Limit Theorem [141].

Proposition 7.1. Let $X_1, X_2, \ldots, X_n$ be n independently (but not necessarily identically) distributed random variables with variances σ_i^2, and assume that the nth moment of each X_i $(i = 1,2,\ldots,n)$ exists and is finite. The necessary and sufficient conditions for the existence of two statistically independent linear forms $Y_1 = \sum_{i=1}^{n} a_i X_i$ and $Y_2 = \sum_{i=1}^{n} b_i X_i$ are

(1) Each random variable which has a nonzero coefficient in both forms in normally distributed.

(2) $\sum_{i=1}^{n} a_i b_i \sigma_i^2 = 0$.

Theorem 7.2. If $A \times B \neq C$, the set of Gaussian vectors where $C\omega_G = AB\omega_G$ holds in Algorithm 7.1 has measure zero.

Proof: Let the matrix $\Delta \in \mathbb{R}^{n\times n}$ denote $AB - C$. Since $A \times B \neq C$, $rank(\Delta) = r > 0$, and $dim\big(null(\Delta)\big) = n - rank(\Delta) = n - r < n$. Here $dim(\cdot)$ denotes dimension and $null(\cdot)$ denotes the null space, i.e. $null(\Delta) = \{x \in \mathbb{R}^n : \Delta \times \mathrm{x} = 0\}$.

We can now find $n - r$ of orthonormal vectors, $v_1, v_2, \ldots, v_{n-r}$, to form a basis for $null(\Delta)$, such that $null(\Delta) = span\{v_1, v_2, \ldots, v_{n-r}\}$, and r more orthonormal vectors, $v_{n-r+1}, v_{n-r+2}, \ldots, v_n$, such that

$$\mathcal{R}^n = span\{v_1, v_2, \ldots, v_{n-r}, v_{n-r+1}, v_{n-r+2}, \ldots, v_n\}.$$

Any vector, and in particular the Gaussian vector, ω_G can be written in this basis as $\omega_G = \sum_{i=1}^{n} \delta_i v_i$, where δ_i 's are the weights in this particular orthonormal coordinate system. If we denote $V = [v_1, v_2, \dots, v_{n-r}, v_{n-r+1}, v_{n-r+2}, \dots, v_n]$, we have

$$V\omega_G = [\delta_1, \delta_2, \dots, \delta_{n-r}, \delta_{n-r+1}, \delta_{n-r+2}, \dots, \delta_n].$$

$C\omega_G = AB\omega_G$ holds in Algorithm 7.1 only if $A(B\omega_G) - C\omega_G = (AB - C)\omega_G = \Delta\omega_G = 0$. This means $\omega_G \in null(\Delta)$, i.e., $\delta_{n-r+1} = 0, \delta_{n-r+2} = 0, \dots, \delta_n = 0$. Due to the fact that ω_G is a Gaussian random vector and V is an orthogonal matrix, Proposition 7.1 tells us that the elements, δ_i, in the resulting vector $V\omega_G$ are normally distributed and statistically independent. With a continuous probability distribution, the discrete event where $\delta_i = 0$ for all $i > n - r$ occurs on a set of measure zero and we will say here that it has probability zero. Hence, GVFA using a Gaussian random projection will have unmatched $C\omega_G$ and $AB\omega_G$ when $A \times B \neq C$ on all but a set of measure zero of Gaussian vectors, which we will say is probability one. □

This argument in Theorem 7.2 is rather direct, but we must point out that the arguments are true when the computations are exact. In next subsection, we will analyze GVFA when float-point errors are present.

7.1.3 Practical Use in Floating-Point Matrix Product Verification

In computer implementations of arithmetic with real numbers, one commonly uses floating-point numbers and floating-point arithmetic. Floating-point numbers are represented as finite numbers in the sense that they have a fixed mantissa and exponent size in number of bits. Therefore, there will be a small probability, p, that $C\omega_G = AB\omega_G$ still holds due to unfortunate floating-point operations in a system with a known machine epsilon, ϵ, when $A \times B \neq C$. The value of p depends on the magnitude of the error between $A \times B$ and C as well as ϵ, whose upper bound is justified in Theorem 7.3.

Theorem 7.3. Assume that ω_G is a standard Gaussian random vector, whose elements are i.i.d. normal variables with mean 0 and variance 1, i.e. the standard normal. Let $\Delta = A \times B - C$, then the probability, p, that $C\omega_G = AB\omega_G$ holds in Algorithm 7.1 using a standard Gaussian random vector ω_G under floating-point uncertainty of size ϵ is

$$p \leq 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}}\right|\right) - 1,$$

where $\Phi(\cdot)$ is the cumulative density function of the standard normal, and $\tilde{\sigma}$ is a constant only related to Δ.

Proof: $A \times B \neq C, \;\; \Delta = A \times B - C \neq 0$. Consider the ith element, g_i, of the product vector $g = \Delta \times \omega_G$, we have

$$g_i = (\Delta \times \omega_G)_i = \sum_{j=1}^{n} \Delta_{ij}(\omega_G)_j.$$

Given ϵ, only if $|g_i| \leq \epsilon$ for all $i = 1, \dots, n$, $C\omega_G = AB\omega_G$ can hold. Since ω_G is a standard normal random vector, g_i for all $i = 1, \dots, n$, are normally distributed as well. This is because they are linear combinations of normals themselves. The key is to compute what the mean and variance is of the g_i. The components of ω_G are i.i.d. standard normals. Thus we have that $E\left[(\omega_G)_j\right] = 0$ and $E\left[{(\omega_G)_j}^2\right] = 1$, for all $j = 1, \dots, n$. Also, we have that $E\left[(\omega_G)_i(\omega_G)_j\right] = 0$ when $i \neq j$. This allows us to compute the mean:

$$E(g_i) = E\left[\sum_{j=1}^{n} \Delta_{ij}(\omega_G)_j\right] = \sum_{j=1}^{n} \Delta_{ij} E\left[(\omega_G)_\mathrm{j}\right] = 0,$$

and the second moment about the mean, i.e. the variance:

$$E\left[g_i^2 - E(g_i)^2\right] = E\left[g_i^2\right] = E\left[\sum_{j=1}^{n} \Delta_{ij}(\omega_G)_j\right]^2 = E\left[\sum_{j=1}^{n} \Delta_{ij}^2 \times 1\right] = \sum_{j=1}^{n} \Delta_{ij}^2.$$

So we have that g_i's are normally distributed with mean zero and variance ${\tilde{\sigma}_i}^2 = \sum_{j=1}^{n} \Delta_{ij}^2$, i.e. $g_i \sim N\left(0, {\tilde{\sigma}_i}^2\right)$.

Then, the probability that $|g_i| \leq \epsilon$ can be computed as follows. Since $g_i \sim N\left(0, {\tilde{\sigma}_i}^2\right)$, we know that $\frac{g_i}{{\tilde{\sigma}_i}^2} \sim N(0,1)$, and so we define the new variables $\tilde{g_i} = \frac{g_i}{{\tilde{\sigma}_i}^2}$ and $\tilde{\epsilon} = \frac{\epsilon}{{\tilde{\sigma}_i}^2}$, and so we have

$$\begin{aligned} p(|g_i| \leq \epsilon) &= p(-\epsilon \leq g_i \leq \epsilon) \\ &= p(-\tilde{\epsilon} \leq \tilde{g_\iota} \leq \tilde{\epsilon}) \\ &= \int_{-\tilde{\epsilon}}^{\tilde{\epsilon}} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} dt \\ &= \Phi(\tilde{\epsilon}) - \Phi(-\tilde{\epsilon}). \end{aligned}$$

Since the probability density function of a standard normal is an even function, we have $\Phi(\tilde{\epsilon}) + \Phi(-\tilde{\epsilon}) = 1$, and so we can use $-\Phi(-\tilde{\epsilon}) = \Phi(\tilde{\epsilon}) - 1$ to get:

$$p(-\epsilon \leq g_i \leq \epsilon) = 2\Phi(\tilde{\epsilon}) - 1 = 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}_i}\right|\right) - 1.$$

Now let us consider computing an upper bound on $p(|g_i| \leq \epsilon, i = 1, \ldots, n)$. We have proved that g_i's are normal random variables, but they are not necessarily independent. And so for this we use some simple ideas from conditional probability. By example, consider

$$p(|g_1| \leq \epsilon \; and \; |g_2| \leq \epsilon\,) = p(|g_2| \leq \epsilon \mid given \; |g_1| \leq \epsilon\,)p(|g_1| \leq \epsilon\,) \leq p(|g_1| \leq \epsilon\,).$$

The inequality holds due to the fact that the probabilities are numbers less than one. Now consider our goal of bounding

$$p(|g_i| \leq \epsilon\,, i = 1, \ldots, n) \leq p(|g_1| \leq \epsilon\,) = 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}_1}\right|\right) - 1,$$

by iterating the conditional probability argument n times. By reordering we could have chosen the bound utilizing any of g_i's. However, let us define $\tilde{\sigma} = \max_i \sqrt{\sum_{j=1}^{n} \Delta_{ij}^2}$, i.e., the maximal standard deviation over all the g_i's, which is only related to the matrix Δ. We can use that value instead to get

$$p = p(|g_i| \leq \epsilon\,, i = 1, \ldots, n) \leq 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}}\right|\right) - 1.$$

□

As an interesting corollary, we can get a better bound in the case that g_i's are independent. In that case

$$p(|g_i| \leq \epsilon\,, i = 1, \ldots, n) = \prod_{i=1}^{n} p(|g_i| \leq \epsilon\,) = \prod_{i=1}^{n} 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}_i}\right|\right) - 1.$$

Let $\tilde{\sigma} = \max_i \sqrt{\sum_{j=1}^{n} \Delta_{ij}^2}$, i.e., i.e., the maximal standard deviation over all the g_i's, which is only related to the matrix Δ. Hence for all $i = 1, \dots, n$, we have that

$$2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}_i}\right|\right) - 1 \leq 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}}\right|\right) - 1.$$

And so, finally we get that

$$\begin{aligned} p &= p(|g_i| \leq \epsilon\,, i = 1, \dots, n) \\ &= \prod_{i=1}^{n} 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}_i}\right|\right) - 1 \\ &\leq \left[2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}}\right|\right) - 1\right]^n \\ &\leq 2\Phi\left(\left|\frac{\epsilon}{\tilde{\sigma}}\right|\right) - 1. \end{aligned}$$

The last inequality is true since the number raised to the nth power is less than one.

Note, that independence gives probability of a false positive that is n times smaller than in the general, dependent case. The conclusion of this seems to be that the bound in the dependent case is overly pessimistic, and we suspect that in cases where the matrix Δ is very sparse, due to a very small number of errors, that we are in the independent g_i's case or have very little dependence, and these more optimistic bounds reflect what happens, computationally.

Theorem 7.3 reveals two interesting facts about GVFA in term of practical floating-point matrix product verification:

(1) The bigger the error caused by the fault, the higher the probability that it can be captured. p is usually very small because the floating point bound, ϵ, is very small.

(2) Similar to the original Freivalds' algorithm, higher confidence can be obtained by iterating the algorithm multiple times. In fact, if we iterate k times using independent Gaussian random vectors, the probability of false positive decreases exponentially as p^k. Actually, due to the fact that p is usually very small, one or a very small number of iterations will produce verification with sufficiently high confidence.

One comment that should be made is that if we consider $\int_{-\tilde{\epsilon}}^{\tilde{\epsilon}} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} dt$ when $\tilde{\epsilon}$ is small, we can easily approximate this. Since the integrand is at its maximum at zero, and is a very smooth function, analytic actually, this integral is approximately the value of the integrand at zero times the length of the integration interval, i.e. $\int_{-\tilde{\epsilon}}^{\tilde{\epsilon}} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} dt \leq\ 2\tilde{\epsilon}\ \frac{1}{\sqrt{2\pi}} = \tilde{\epsilon}\sqrt{\frac{2}{\pi}}$. This is justified as $\tilde{\epsilon}$ is a number on the order of the machine epsilon, which is 2^{-23}in single precious or 2^{-52} in double precision floating point, divided by $\tilde{\sigma_i}^2 = \sum_{j=1}^{n} \Delta_{ij}^2$.

Compared to the deterministic methods such as Huang-Abraham scheme (see Section 2.4.1), GVFA has the following advantages:

(1) Certain fault patterns, as shown in Section 2.4, are undetectable in deterministic methods such as the Huang-Abraham scheme. Deterministic methods absolutely cannot detect faults with certain patterns, i.e., certain patterns are detected with probability zero. In contrast, there are no fault patterns that are undetectable by GVFA with 100% probability. Moreover, iterating the algorithm multiple times can increase the probability of detecting any fault pattern any value less than one by iteration.

(2) From the computational point-of-view, normal random vectors are generated independently of A, B, and C, which avoids the costly computation of checksums.

7.2 Extensions of GVFA

7.2.1 Huang-Abraham-like GVFA

GVFA can also be implemented in a way similar to that of Huang-Abraham scheme by providing row and column verifications. The Huang-Abraham-like GVFA is described in Algorithm 7.2. Similar to the Huang-Abraham scheme, a mismatch element of the row vectors of $\omega_R C$ and $\omega_R AB$ as well as that of the column vectors of $C\omega_c$ and $AB\omega_c$ uniquely identify a faulty element in C. By considering floating-point errors, the false positive probability of identifying this fault becomes p^2, according to the analysis in Section 7.1.3. However, the computational cost doubles with six matrix-vector multiplications and two vector comparisons. This is essentially the same work as doing two independent iterations of GVFA, and obtains the same bound.

```
Algorithm 7.2: Huang-Abraham-like GVFA
Step 1. Generate a row Gaussian random vector, $\omega_R$ and a column Gaussian random vector $\omega_C$ where
        $\omega_R$ and $\omega_c$ are n-vectors of independent (but not necessarily identically) distributed normal
        random variables with finite mean and variance.
Step 2. Calculate the projection of $C$ on $\omega_R$ and $\omega_c$: $\omega_R C = \omega_R \times C$ and $C\omega_c = C \times \omega_c$.
Step 3. Calculate the projection of product $A \times B$ on $\omega_R$ and $\omega_c$: $\omega_R AB = (\omega_R \times A) \times B$ and $AB\omega_c =$
        $A \times (B \times \omega_c)$.
```

7.2.2 Implementation using Fused Multiply-Add Hardware

The Fused Multiply-Add (FMA) machine instruction performs one multiply operation and one add operation with a single rounding step [142]. This was implemented to enable potentially faster performance in calculating the floating-point accumulation of products, $a := a + b \times c$. Recall that GVFA employs three matrix-vector multiplications to project $A \times B$ and C onto a normal random vector, which requires a sequence of product accumulations that cost $3n(2n-1)$ floating-point operations. Therefore, the performance of GVFA can be potentially boosted on modern computing architectures that support the FMA. More importantly, due to a single rounding step used in the FMA instruction instead of two roundings within separate instructions, less loss of accuracy occurs when using the FMA instruction in calculating the accumulation of products [143]. This should further reduce the floating-point rounding errors that cause false positives.

7.2.3 Applicability

GVFA can be easily extended to a more general matrix multiplication operation where A is $m \times p$, B is $p \times n$, and C is $m \times n$. The overall computational time then becomes $O(mp + np)$. The algorithm can be further extended to verify the product of N matrices, which requires overall $N + 1$ matrix-vector multiplications. GVFA can also be applied to verifying a wide variety of matrix decomposition operations such as LU, QR, Cholesky, as well as eigenvalue computations, and singular value decompositions. In this case, faults are not in the product matrix but occur in the decomposed ones instead. Anyway, GVFA can be directly applied with no modifications necessary.

GVFA is a new tool to detect faults in numerical linear algebra, and since it is based on random Gaussian projection, it is related to the many new randomized algorithms being used directly in numerical linear algebra [33,102]. The fundamental idea of these randomized algorithms is to apply efficient sampling on the potentially very large matrices to extract their important characteristics so as to fast approximate numerical linear algebra operations. We believe that GVFA will be a very useful tool in the development of fault-tolerant and otherwise resilient algorithms for solving large numerical linear algebra problems. In fact, it seems that GVFA's similarity to other, new, stochastic techniques in numerical linear algebra affords the possibility of creating stochastic linear solvers that are by their very nature resilient and fault-tolerant. This is highly relevant for new machines being developed in HPC to have maximal floating-point operations per second (FLOPS) while existing within restrictive energy budgets. These HPC systems will be operating at voltages lower than most current systems, and so they are expected to be particularly susceptible to soft errors. However, even if one is not anticipating the use of these high-end machines, the trend in processor design is to lower power, and is being driven by the explosion of mobile computing. Thus, the ability to reliably perform complicated numerical linear algebraic computations on systems more apt to experience soft faults is a very general concern. GVFA will make it much easier to perform such computations with high fidelity in HPC, cloud computing, mobile applications, as well in big-data settings.

CHAPTER VIII

SUMMARY AND POSTDISSERTATION REASEARCH

The efficiency of large-scale linear algebra operations is essential for the performance of scientific computing and big data analysis applications. The large volume of matrices in these applications brings grand computational challenges to classical numerical linear algebra algorithms, including costly matrix pass, limited scalability to modern parallel and distributed computing architectures, as well as potential memory errors.

The dissertation describes our past five years' research work on designing new Monte Carlo algorithms to carry out efficient and reliable large-scale linear algebra operations while taking advantage of modern parallel computing architectures. In particular, Monte Carlo algorithms for addressing the problems of solving systems of linear equations, constructing low-rank approximations, finding extreme eigenvalues/eigenvectors, and verifying the correctness of matrix-matrix multiplications are developed with mathematical rigor and are supported with numerical results on real-life applications.

The fundamental research on my dissertation provides me with a solid base of knowledge in numerical linear algebra for parallel high-performance computing systems. There are several interesting avenues for future work which we would like to explore. For example, enhancing sampling efficiency in matrix-vector products along MCGMRES iterations, implementing our R^3SVD algorithm on big data analysis platforms, and applying the sliding window power method to fast estimate multiple extreme eigenvalues/eigenvectors for *ab initio* nuclear physics applications.

REFERENCES

[1] H. Ji, and Y. Li, “Monte Carlo Methods and their Applications in Big Data Analysis,” *Mathematical Problems in Data Science - Theoretical and Practical Methods*, Springer, ISBN: 978-3-319-25127-1, 2015.

[2] R. L. Burden, and J. D. Faires, *Numerical Analysis*, Brooks/Cole, Cengage Learning, 2011.

[3] J. M. Hammersley, and D. C. Handscomb, *Monte Carlo Methods*, Chapman and Hall, Methuen & Co., London, and John Wiley & Sons, New York, 1964.

[4] J. S. Liu, *Monte Carlo strategies in scientific computing*, Springer Science & Business Media, 2008.

[5] Y. Li, and M. Mascagni, “Grid-based Monte Carlo Application,” *Grid Computing Third International Conference*, pp. 13–24, 2002.

[6] Y. Li, and M. Mascagn, “Analysis of large-scale grid-based Monte Carlo applications,” *Int. J. High Perform. Comput. Appl.*, vol. 17, pp. 369–382, 2003.

[7] G. H. Golub, and C. F. Van Loan, *Matrix computations*, Johns Hopkins University Press, 2012.

[8] Y. Saad, *Iterative methods for sparse linear systems*, SIAM, 2003.

[9] J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method without the Agonizing Pain, Tech. Report*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[10] R. Fletcher, “Conjugate gradient methods for indefinite systems,” *Numerical analysis*, Springer Berlin Heidelberg, pp. 73-89, 1976.

[11] Y. Saad, and M. H. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM J. Sci. and Stat. Comput.*, vol. 7. no. .3, pp. 856-869, 1986.

[12] G. E. Forsythe, and R. A. Leibler, “Matrix inversion by a Monte Carlo method,” *Math. Tables Other Aids Comput.*, vol. 4, pp. 127–129, 1950.

[13] I. Dimov, “Minimization of the probable error for some Monte Carlo methods,” *in Proc. of the Summer School on Mathematical Modelling and Scientific Computations*, Bulgarian Academy of Sciences, Sofia, pp. 159–170, 1991.

[14] W. Wasow, “A note on the inversion of matrices by random walks,” *Math. Tables Other Aids Comput.*, vol. 6, pp. 78–81, 1952.

[15] J. H. Halton, “Sequential Monte Carlo techniques for the solution of linear systems,” *J. Sci. Comput.*, vol. 9, pp. 213–257, 1994.

[16] I. T. Dimov, T. T. Dimov, and T. V. Gurov, “A new iterative Monte Carlo Approach for Inverse Matrix Problem,” *J. Comput. Appl. Math.*, vol. 92, pp. 15–35, 1998.

[17] I. Dimov, *Monte Carlo Methods for Applied Scientists*, World Scientific Publishing, Singapore, 2008.

[18] C. J. K. Tan, “Antithetic Monte Carlo Linear Solver,” *in Proc. of ICCS 2002*, pp. 383–392, 2002.

[19] A. Srinivasan, and V, Aggarwal, “Improved Monte Carlo linear solvers through non-diagonal splitting,” *in Proc. of ICCSA*, pp. 168–177, 2003.

[20] K. Sabelfeld, and N. Mozartova, “Sparsified randomization algorithms for large systems of linear equations and a new version of the random walk on boundary method,” *Monte Carlo Methods Appl.*, vol. 15, pp. 257–284, 2009.

[21] M. Mascagni, and A. Karaivanova, “A Parallel Quasi-Monte Carlo Method for Solving Systems of Linear Equations,” *in Proc. of ICCS 2002*, pp. 598–608, 2002.

[22] H. Ji, and Y. Li, “GPU accelerated randomized singular value decomposition and its application in image compression,” *in Proc. of MSVESCC*, pp. 39-45, 2014.

[23] I. Jolliffe, *Principal Component Analysis*, 2nd ed. New York, NY, USA: Springer-Verlag, 2002.

[24] D. Donoho, “Compressed sensing,” *IEEE Trans. Inf. Theory*, vol. 52, no. 4, pp. 5406–5425, 2006.

[25] J. F. Cai, E. J. Cande`s, and Z. Shen, “A Singular Value Thresholding Algorithm for Matrix Completion,” *SIAM J. Optimiz.*, vol. 20, pp. 1956-1982, 2010.

[26] M. W. Mahoney, “Randomized algorithms for matrices and data,” *Found. Trends Mach. Learning*, vol. 3, no. 2, pp. 123–224, 2011.

[27] E. Liberty, F. Woolfe, P. G. Martinsson, V. Rokhlin, and M. Tygert, “Randomized algorithms for the low-rank approximation of matrices,” *Proc. Natl. Acad. Sci.,*vol. 104, no. 51, pp. 20167–20172, 2007.

[28] P. Drineas, E. Drinea, and P. S. Huggins, “An experimental evaluation of a Monte-Carlo algorithm for singular value decomposition,” *in Proc. of 8th Panhellenic Conf. Informat*., Nicosia, Cyprus, pp. 279–296, 2003.

[29] P. Drineas, R. Kannan, and M. W. Mahoney, “Fast Monte-Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix,” *SIAM J. Comput.*, vol. 36, no. 1, pp. 158–183, 2006.

[30] S. Eriksson-Bique, M. Solbrig, M. Stefanelli, S. Warkentin, R. Abbey, and I. Ipsen, “Importance sampling for a Monte Carlo matrix multiplication algorithm, with application to information retrieval,” *SIAM J. Sci. Comput.*, vol. 33, no. 4, pp. 1689–1706, 2011.

[31] P. Drineas, and M. W. Mahoney, “On the Nyström method for approximating a Gram matrix for improved kernel-based learning,” *J. Mach. Learn. Res.*, vol. 6, pp. 2153–2175, 2005.

[32] K. Zhang, I. W. Tsang, and J. T. Kwok, “Improved Nyström lowrank approximation and error analysis,” *in Proc. 25th Int. Conf. Mach. Learning*, pp. 1232–1239, 2008.

[33] N. Halko, P. G. Martinsson, and J. A. Tropp, “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions,” *SIAM Rev*., vol. 53, no. 2, pp. 217–288, 2011.

[34] R. Motwani, and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[35] L. Page, “*Pagerank: Bringing order to the web*,” Technical report, Stanford Digital Library Project, 1997.

[36] Y. Kim, and K. Shim, “TWILITE: A recommendation system for Twitter using a probabilistic model based on latent Dirichlet allocation,” *Information Systems*, vol. 42, pp. 59-77, 2014.

[37] Y. Yang, S. Yang, and B. Hu, “Fighting WebSpam: detecting Spam on the Graph via content and link features,” *in Proceedings of PAKDD*, 2008.

[38] *CUBLAS library*, NVIDIA Corporation, Santa Clara, 2008.

[39] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, “CULA: hybrid gpu accelerated linear algebra routines,” *SPIE Defense and Security Symposium (DSS*), vol. 7705, 2010.

[40] S. Tomov, et al. , *MAGMA Library*. Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA, 2014.

[41] *CUSPARSE library*, NVIDIA Corporation, Santa Clara, California, 2014.

[42] S. V. Kuznetsov, “An Approach of the QR Factorization for Tall-and-Skinny Matrices on Multicore Platforms,” *Appl. Parallel Sci. Comput.*, pp. 235-249, 2013.

[43] A. R.Benson, D. F. Gleich, and J. Demmel, “Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures,” *in Proc. of IEEE BigData*, 2013.

[44] E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, “Predicting the number of fatal soft errors in los alamos national laboratory’s asc q supercomputer,” *IEEE Trans. Device Mater*. Rel., vol. 5, pp. 329–335, 2005.

[45] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, “ Extending stability beyond cpu millennium: a micron-scale atomistic simulation of kelvin-helmholtz instability,” *in Proc. of SC 2007*, pp. 1–11, 2007.

[46] B. Schroeder, E. Pinheiro, and W. D. Weber, “DRAM errors in the wild: a large-scale field study,” *Commun. ACM*, vol. 54, pp. 100–107, 2011.

[47] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” *in Proc. of DSN 2002*, pp. 389–398, 2002.

[48] J. J. Dongarra, J. D. Cruz, S. Hammerling, and I. S. Duff, “Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs,” *ACM Trans. Math. Softw.*, vol. 16, pp. 18–28, 1990.

[49] J. W. Demmel, and N. J. Higham, “Stability of block algorithms with fast level-3 BLAS,” *ACM Trans. Math. Softw.*, vol. 18, pp. 274–291, 1992.

[50] K. Gallivan, W. Jalby, and U. Meier, “The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory,” *SIAM J. Sci. Stat. Comp.*,vol. 8, pp. 1079–1084, 1987.

[51] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. A. Van de Gejin, “Fault-tolerant high-performance matrix multiplication: Theory and practice,” *in Proc. of DSN 2001*, pp. 47–56, 2001.

[52] K. H. Huang, and J. A. Abraham, “Algorithm-based fault tolerance for matrix operations,” *IEEE Trans. Comput.,* vol. 100, pp. 518–528, 1984.

[53] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, “Algorithm-based fault tolerance applied to high performance computing,” *J. Parallel Distrib. Comput.*, vol. 69, pp. 410–416, 2009.

[54] R. Freivalds, “Probabilistic machines can use less running time,” *in Proc. of IFIP Congress 77*, pp. 839–842, 1977.

[55] D. D. Chinn, and R. K. Sinha, “Bounds on sample space size for matrix product verification,” *Inform. Process. Lett.*, vol. 48, pp. 87–91, 1993.

[56] N. Alon, O. Goldreich, J. Hastad, and R. Peralta, “Simple construction of almost kwise independent random variables,” *in Proc. of FOCS 1990*, pp. 544–553, 1990.

[57] J. Naor, and M. Naor, “Small-bias probability spaces: Efficient constructions and applications,” *SIAM J. Comput.,* vol. 22, pp. 838–856, 1993.

[58] C. Lisboa, M. Erigson, and L. Carro, “A low cost checker for matrix multiplication,” *in Proc. of IEEE Latin-American Test Workshop*, 2007.

[59] D. P. O'Leary, “Parallel implementation of the block conjugate gradient algorithm,” *Parallel Comput*. vol. 5, no. 1, pp. 127-139, 1987.

[60] H. Ji, M. Sosonkina, and Y. Li, “An implementation of block conjugate gradient algorithm on CPU-GPU processors,” *in Proc. of Co-HPC 2014*, pp. 72-77, 2014.

[61] G.W. Stewart, “Block Gram-Schmidt orthogonalization,” *SIAM J. Sci. Comput.*, vol. 31, no. 1, pp. 761-775, 2008.

[62] J. J. Dongarra, J. D. Cruz, S. Hammerling, I. S. Duff, “Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs,” *ACM Trans. Math. Softw.* vol. 16, no. 1, pp. 18-28, 1990.

[63] K. Gallivan, W. Jalby, U. Meier, “The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory,” *SIAM J. Sci. and Stat. Comput*., vol. 8, no. 6, pp.1079-1084, 1987.

[64] J. W. Demmel, and N. J. Higham, “Stability of block algorithms with fast level-3 BLAS,” *ACM Trans. Math. Softw.*, vol. 18, no. 3, pp. 274-291,1992.

[65] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, John Wiley & Sons, Inc., New York, NY, USA, 2001.

[66] C. G. Broyden, “A breakdown of the block CG method,” *Optim. Methods Softw.*, vol. 7, pp. 41–55, 1996.

[67] M. H. Gutknecht, “Block Krylov space methods for linear systems with multiple right-hand sides: An introduction,” *in Modern Mathematical Models, Methods and Algorithms for Real World Systems*, Anamaya Publishers, New Delhi, India, pp. 420–447, 2006.

[68] A. A. Nikishin, and A. Y. Yeremin, “Variable block CG algorithms for solving large sparse symmetric positive definite linear systems on parallel computers, I: general iterative scheme,” *SIAM J. Matrix Anal. Appl.*, vol. 16, pp. 1135–1153, 1995.

[69] D. P. O’Leary, “The block Conjugate Gradient algorithm and related methods,” *Linear Algebra Appl.*, vol. 29, pp. 293–322, 1980.

[70] M. Robb, and S. Miloud, “Exact and inexact breakdowns in the block GMRES method,” *Linear Algebra Appl.*, vol. 419, pp. 265–285, 2006.

[71] J. Chen, “A deflated version of the block Conjugate Gradient algorithm with an application to Gaussian process maximum likelihood estimation,” *Preprint ANL/MCS-P1927-0811*, Argonne National Laboratory, Argonne, IL, 2011.

[72] Y. T. Feng, D. R. J. Owen, and P. Peric, “A block Conjugate Gradient method applied to linear systems with multiple right-hand sides,” *Comput. Methods Appl. Mech. Eng.,* vol. 127, pp. 203–215, 1995.

[73] A. A. Dubrulle, “Retooling the method of block Conjugate Gradients,” *Electron. Trans. Numer. Anal.*, vol. 12, pp. 216–233, 2001.

[74] Z. Ilya, D. P. O’Leary, and H. Elman, “Complete stagnation of GMRES,” *Linear Algebra Appl.*, vol. 367, pp.165–183, 2003.

[75] Z. Leyk, “Breakdowns and stagnation in iterative methods,” *BIT*, vol. 37, pp. 377-403, 1997.

[76] R. Barrett, et al., *Templates for the solution of linear systems: building blocks for iterative methods,* SIAM, 1994.

[77] T. A. Davis, *University of Florida sparse matrix collection*, http://www.cise.u.edu/research /sparse/matrices/.

[78] N. Li, Y. Saad, and E. Chow, “Crout version of ILU for general sparse matrices,” *SIAM J Sci. Comput.*, vol. 25, pp. 716–728, 2003.

[79] T. Schmelzer, *Block Krylov methods for Hermitian linear systems*, Diploma thesis, Department of Mathematics, University of Kaiserslautern, Germany, 2004.

[80] M. Robb, and S. Miloud, “Exact and inexact breakdowns in the block GMRES method,” *Linear Algebra Appl.,* vol. 419, pp. 265–285, 2006.

[81] M. H. Gutknecht, *Block Krylov space solvers: A survey,* available online: http://www.sam.math.ethz.ch/ mhg/talks/bkss.pdf, 2005.

[82] I. Duff, R. Grimes, and J. Lewis, *Users guide for the Harwell-Boeing sparse matrix collection*, Research and Technology Division, Boeing Computer Services, Seattle, Washington, USA, 1992.

[83] J. Towns, et al., “XSEDE: Accelerating scientific discovery,” *Comput. Sci. Eng.*, vol. 16, no. 5, pp.62–74, 2014.

[84] H. Ji, and Y. Li, “A breakdown-free block conjugate gradient method,” *BIT*, submitted, 2016.

[85] A. Gaul, M. H. Gutknecht, J. Liesen, and R. Nabben, “A framework for deflated and augmented krylov subspace methods,” *SIAM J. Matrix Anal. Appl.*, vol. 34, no. 2, pp.495-518, 2013.

[86] J. Erhel, and F. Guyomarc'h, “An augmented conjugate gradient method for solving consecutive symmetric positive definite linear systems,” *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp.1279-1299, 2000.

[87] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h, “A deflated version of the conjugate gradient algorithm,” *SIAM J. Sci. Comput.*, vol. 21, no. 5, pp.1909-1926, 2000.

[88] R. A. Nicolaides, “Deflation of conjugate gradients with applications to boundary value problems,” *SIAM J. Numer. Anal.* vol. 24, no. 2, pp. 355-365, 1987.

[89] Z. Dostal, “Conjugate gradient method with preconditioning by projector,” *Internat J. Comput. Math.*, vol. 23, no. 34, pp. 315-323, 1988.

[90] J. Erhel, K. Burrage, and B. Pohl, “Restarted GMRES preconditioned by deflation,” *J. Comput. Appl. Math.,* vol. 69, no. 2, pp.303-318,1996.

[91] R. B. Morgan, “A restarted GMRES method augmented with eigenvectors,” *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 4 , pp.1154-1171, 1995.

[92] E. De Sturler, “Nested krylov methods based on GCR,” *J. Comput. Appl. Math.*, vol. 67, no. 1, pp.15-41, 1996.

[93] S. A. Kharchenko, and A. Y. Yeremin, “Eigenvalue translation based preconditioners for the GMRES (k) method,” *Numer. Linear Algebra Appl.*, vol. 2, no. 1, pp.51-77, 1995.

[94] M. H. Gutknecht, “Deated and augmented krylov subspace methods: A framework for deated BiCG and related solvers,” *SIAM J. Matrix Anal.Appl.*, vol. 35, no. 4, pp.1444-1466, 2014.

[95] R.B. Sidje, and N. Winkles, “Evaluation of the performance of inexact GMRES,” *J. Comput. Appl. Math.*, vol. 235, no. 8, pp.1956-1975, 2011.

[96] V. Simoncini. "Variable Accuracy of Matrix-Vector Products in Projection Methods for Eigencomputation," *SIAM J. Num. Anal.*, vol.43, no. 3, pp. 1155-1174, 2005.

[97] X. Du, and D.B. Szyld, "Inexact GMRES for singular linear systems," *BIT,* vol. 48, no. 3, pp.511-531, 2008.

[98] R.B. Sidje, "Inexact uniformization and GMRES methods for large Markov chains," *Numer. Linear Algebr.*, vol. 18, no.6, pp.947-960, 2011.

[99] A. Bouras, and V., Frayssé. "Inexact matrix-vector products in Krylov methods for solving linear systems: A relaxation strategy," *SIAM J. Matrix Anal. A.*, vol. 26, no. 3, pp.660-678, 2005.

[100] L. Giraud, S. Gratton, and J. Langou, "Convergence in Backward Error of Relaxed GMRES," *SIAM J. Sci. Comput.,* vol. 29, no. 2, pp.710-728, 2007.

[101] B. W. David, "Generating random spanning trees more quickly than the cover time," *in Proc. of STOC 1996*, ACM, New York, NY, USA, pp. 296-303, 1996.

[102] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication," *SIAM J. Comput.*, vol. 36, no. 1, pp. 158–183, 2006.

[103] N. Halko, "*Randomized methods for computing low-rank approximations of matrices*," Ph.D. dissertation, University of Colorado, 2012.

[104] S. Voronin, and P.G. Martinsson, "RSVDPACK: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and GPU architectures," *arXiv preprint arXiv:1502.05366*, 2015. Available at: http://arxiv.org/abs/1502.05366

[105] S. Voronin, and P.G. Martinsson, "A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices," *arXiv preprint arXiv:1503.07157*. 2015. Available at: http://arxiv.org/abs/1503.07157

[106] A. Mathai, and G. Pederzoli, *Characterizations of the Normal Probability Law*. New Delhi, India: Wiley Eastern Ltd., 1977.

[107] *NASA's Planetary Photojournal*. [Online] available at : http://photojournal.jpl.nasa.gov.

[108] H. Ji, E. O'Saben, A. Boudion, and Y. Li, "March Madness Prediction: A Matrix Completion Approach," *in Proc. of MSVESCC 2015*, pp. 41-48, 2015.

[109] E.J. Cande`s, and B. Recht, "Exact Matrix Completion via Convex Optimization," *Foundations on Computational Math*., vol 9, pp. 717- 772, 2009.

[110] B. Recht, M. Fazel, and P.A. Parrilo, "Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization," *SIAM Rev.*, vol. 52, no. 3, pp.471-501, 2010.

[111] B. Recht, "A Simpler Approach to Matrix Completion," *J. Machine Learning Research*, vol. 12, pp. 413-3430, 2011.

[112] J. D. M. Rennie, and N. Srebro, "Fast maximum margin matrix factorization for collaborative prediction," *in Proc. ICML*, 2005.

[113] D. Zhang, Y. Hu, J. Ye, X. Li, and X. He, "Matrix completion by truncated nuclear norm regularization," *in Proc. of CVPR*, pp. 2192–2199, 2012.

[114] H. Ji, C. Liu, Z. Shen, and Y. Xu, "Robust video denoising using low rank matrix completion," *in Proc. of IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1791–1798, 2010.

[115] A. Weber, *SIPI image database,* The USC-SIPI Image Database, Signal & Image Processing Institute, Department of Electrical Engineering, Viterbi School of Engineering, Univ. of Southern California (2012), available at : http://sipi.usc.edu/database

[116] R. M. Larsen, *Lanczos bidiagonalization with partial reorthogonalization*, Department of Computer Science, Aarhus University, Technical report, DAIMI PB-357, 1998.

[117] S. F. McCormick, and T. Noe, "Simultaneous iteration for the matrix eigenvalue problem," *Linear Algebra Appl.,* vol. 16, no.1, pp. 43-56, 1977.

[118] H. Ji, Y. Li, and S. Weinberg, "Calcium Ion Fluctuations Alter Channel Gating in a Stochastic Luminal Calcium Release Site Model," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, in press, 2015.

[119] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of Intel Xeon Phi, GPU, and CPU," *arXiv preprint arXiv:1311.0378*, 2013.

[120] A. Yaseen, H. Ji, and Y. Li, "A Load-Balancing Workload Distribution Scheme for Three-Body Interaction Computation on Graphics Processing Units (GPU) ," *J. Parallel. Distrib. Comput.*, vol. 87, pp. 91-101, 2016.

[121] Intel, *MKL. Intel Math Kernel Library*, 2013.

[122] G. Hager, and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[123] S. S. Konstantin, *Memory Bandwidth for Intel Xeon Phi (And Friends)* , 2013. Retrieved from http://clusterdesign.org/2013/02/memory-bandwidth-for-intel-xeon-phi-and-friends/

[124] FLOPS. *In Wikipedia*. Retrieved from http://en.wikipedi a.org/wiki/FLOPS.

[125] F. Masci, *Benchmarking the Intel Xeon Phi Coprocessor*, 2014.

[126] NVIDIA, *NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM K110*, Retrieved from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[127] *NVIDIA, Tesla K20 GPU Active Accelerator*, Retrieved from http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v02.pdf.

[128] E. N. Elnozahy, L. Alvisi, Y.M. Wang, D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375-408, 2002.

[129] P. Banerjee, J.A. Abraham, "Bounds on algorithm-based fault tolerance in multiple processor systems," *IEEE Trans. Comput.*, vol. 100, no. 4, pp. 296-306, 1986.

[130] F.T. Luk, and H. Park, "An analysis of algorithm-based fault tolerance techniques," *J. Parallel Distrib. Comput.*, vol. 5, no. 2, pp.172-184, 1988.

[131] A. J. V. de Goor, *Testing semiconductor memories: theory and practice*. John Wiley & Sons, New York, 1991.

[132] K. L. Cheng, C.W. Wang, and J.N. Lee, "FAME: a fault-pattern based memory failure analysis framework," *in Proc. of ICCAD 2003*, pp. 595-598, 2003.

[133] D. D. Chinn, and R.K. Sinha, "Bounds on sample space size for matrix product verification," *Inform. Process. Lett.*, vol. 48, no. 2, pp. 87-91, 1993.

[134] N. Alon, O. Goldreich, J. Hastad, and R. Peralta, "Simple construction of almost k-wise independent random variables," *in Proc. of the 31st Annual Symposium on Foundations of Computer Science*, pp. 544-553, 1990.

[135] J. Naor, and M. Naor, "Small-bias probability spaces: Efficient constructions and applications," *SIAM J. Comput.*, vol. 22, no.4, pp. 838-856 , 1993.

[136] C. Lisboa, M. Erigson, and L. Carro, "A low cost checker for matrix multiplication," *In: IEEE Latin-American Test Workshop*, 2007.

[137] L. Gasieniec, C. Levcopoulos, and A. Lingas, "Efficiently correcting matrix products," *In: Algorithms and Computation*, pp. 53-64. Springer , 2014.

[138] Y. Li, and M. Mascagni, "Analysis of large-scale grid-based Monte Carlo applications," *Int. J. High Perform. Comput. Appl.*, vol. 17, no. 4, pp. 369-382, 2003.

[139] R. J. Muirhead, *Aspects of multivariate statistical theory*, Wiley, New York, 1982.

[140] E. Lukacs, and E.P. King, "A property of the normal distribution," *Ann. Math. Stat.*, vol. 25, no.2, pp. 389-394, 1954.

[141] J. W. Lindeberg, "Eine neue herleitung des exponentialgesetzes in der wahrscheinlichkeit-srechnung," *Math. Z*, vol. 15, no. 1, pp. 211-225, 1922.

[142] E. Hokenek, R.K. Montoye, and P.W. Cook, "Second-generation risc floating point with multiply-add fused," *IEEE J. Solid-State Circuits*, vol. 25, no. 5, pp. 1207-1213, 1990.

[143] S. Boldo, and J.M. Muller, "Exact and approximated error of the FMA," *IEEE Trans. Comput.* vol. 60, no. 2, pp. 157-164, 2011.

[144] M. R. Hestenes, and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Natl. Bur. Standards*, vol. 49, pp. 409-436, 1952.

[145] R. Fletcher, "Conjugate gradient methods for indefinite systems," *Numerical analysis*, Lecture Notes in Mathematics, Springer, pp. 73-89, 1976.

[146] R. Reddy, A. Lastovetsky, and P. Alonso, "Heterogeneous PBLAS: Optimization of PBLAS for Heterogeneous Computational Clusters," *in Proc. of ISPDC 2008*, pp. 73-80, 2008.

[147] J. Choi, et al., "ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance," *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, Springer Berlin Heidelberg, pp.95-106, 1995.

[148] H. Ji, Y. Li, and S. H. Weinberg, "Calcium ion fluctuations alter channel gating in a stochastic luminal calcium release site model," *in Proc. of ISBRA2015*, Norfolk, 2015.

[149] Z. Liu, and L. Vandenberghe, "Interior-point method for nuclear norm approximation with application to system identification," *SIAM J. Matrix Anal. Appl.,* vol. 31, pp. 1235– 1256, 2009.

[150] H. Ji, W. Yu, and Y. Li, "A Rank Revealing Randomized Singular Value Decomposition (R^3SVD) Algorithm for Low-rank Matrix Approximations," *arXiv:1605.08134,* 2016.

[151] H. Ji, and Y. Li, "Block Conjugate Gradient Algorithms for Least Squares Problems", *J. Comput. Appl. Math*., submitted, 2016.

[152] H. Ji, M. Mascagni, and Y. Li, "Gaussian Variant of Freivalds' Algorithm for Efficient and Reliable Matrix Product Verification," *Algorithmica*, submitted , 2016.

[153] H. Ji, M. Mascagni, and Y. Li, "Convergence Analysis of Markov Chain Monte Carlo Linear Solvers Using Ulam--von Neumann Algorithm," *SIAM J. Num. Anal.*, vol. 51, no. 4, pp. 2107-2122, 2013.

[154] H. Ji, and Y. Li, "Reusing random walks in Monte Carlo methods for linear systems," *in Proc. of ICCS 2012*, vol 9, pp. 383-392, 2012.

[155] H. Ji, S. H. Weinberg, M. Li, J. Wang, and Y. Li, "An Apache Spark Implementation of Block Power Method for Computing Dominant Eigenvalues and Eigenvectors of Large-Scale Matrices," *BDCloud 2016*, submitted, 2016.

[156] H. Ji, E. O'Saben, R. Lambi, and Y. Li, "Matrix Completion Based Model V2.0: Predicting the Winning Probabilities of March Madness Matches," *in Proc. of MSVESCC 2016*, in press, 2016.

APPENDIX A

ADDITIONAL PROOFS

Lemma 3.8. Suppose R_i is an $n \times s$ residual matrix of rank r_i $(r_i \leq s)$ at the ith iteration, then

$$rank\left(\tilde{P}_i^T R_i\right) = r_i.$$

Proof. Let $\tilde{P}_i$ denote an orthonormal basis of the search space $\mathcal{P}_i$, which is spanned by $Z_i + \tilde{P}_{i-1}\tilde{\beta}_{i-1}$ shown in Algorithm 3.2, then $Z_i + \tilde{P}_{i-1}\tilde{\beta}_{i-1}$ can be expressed as

$$Z_i + \tilde{P}_{i-1}\tilde{\beta}_{i-1} = \tilde{P}_i\delta, \qquad (1)$$

where δ is an $r_i \times s$ matrix of rank r_i. Left multiplying R_i^T to (1), we can get

$$R_i^T Z_i + R_i^T \tilde{P}_{i-1}\tilde{\beta}_{i-1} = R_i^T \tilde{P}_i \delta.$$

According to Corollary 3.6, $R_i^T \tilde{P}_{i-1} = 0$. Then,

$$R_i^T Z_i = R_i^T \tilde{P}_i \delta.$$

According to Proposition 3.4, we can obtain $rank\left(\left(R_i^{\mathrm{T}}\tilde{P}_i\right)\delta\right) = rank\left(R_i^T Z_i\right) = rank(R_i)$. Again, applying the basic rules of matrix rank, $rank\left(\tilde{P}_i^T R_i\right) = rank\left(\left(R_i^T \tilde{P}_i\right)\delta\right) = r_i$ is derived. □

Lemma 3.9. Z_{i+1} is conjugate to search spaces $\mathcal{P}_j$ where $j < i$.

Proof. Since R_{j+1} is generated by

$$R_{j+1} = R_j - A\tilde{P}_j\tilde{\alpha}_j, \qquad (2)$$

left multiplying (2) by ${Z_{i+1}}^T$ and we have

$${Z_{i+1}}^T R_{j+1} = {Z_{i+1}}^T R_j - {Z_{i+1}}^T A\tilde{P}_j\tilde{\alpha}_j.$$

When $j < i$, according to Corollary 3.7, ${Z_{i+1}}^T R_j = 0$ and ${Z_{i+1}}^T R_{j+1} = 0$. Thus, ${Z_{i+1}}^T A\tilde{P}_j\tilde{\alpha}_j = 0$ for all $j < i$.

Based on Theorem 3.5, $\tilde{\alpha}_j = \left({\tilde{P}_j}^T A\tilde{P}_j\right)^{-1} {\tilde{P}_j}^T R_j$, we have

$$Z_{i+1}{}^T A\tilde{P}_j \left(\tilde{P}_j{}^T A\tilde{P}_j\right)^{-1} \tilde{P}_j{}^T R_j = 0$$

Due to the facts that $\tilde{P}_j{}^T A\tilde{P}_j$ is an $r_j \times r_j$ matrix with full rank, $\tilde{P}_j{}^T R_j$ is a $r_j \times s$ matrix with rank r_j by Lemma 3.8, and $r_j \leq s$, $Z_{i+1}{}^T A\tilde{P}_j = 0 \quad (j < i)$ is obtained. □

APPENDIX B

SYSTEMS OF LINEAR EQUATIONS

The coefficient matrix A and the right-hand side matrix B used in Section 3.2.5.3 are presented below.

$A =$

$$\left[\begin{matrix}
121.164272268116 & 17.8757971682236 & 8.91160049194292 & 14.8013917105125 & 12.7809854465276 \\
17.8757971682236 & 123.317477848499 & 15.3784313056350 & 13.9826052710372 & 8.99969320736193 \\
8.91160049194292 & 15.3784313056350 & 114.944841846832 & 14.0212707850901 & 18.3077854261355 \\
14.8013917105125 & 13.9826052710372 & 14.0212707850901 & 112.181615127048 & 6.98849991034816 \\
12.7809854465276 & 8.99969320736193 & 18.3077854261355 & 6.98849991034816 & 112.325588245555 \\
14.7026896764278 & 12.0449732854894 & 14.5027264215831 & 15.6000866949538 & 9.62150309966732 \\
16.6581592592829 & 14.8004340755219 & 12.0968055881465 & 15.6521797575797 & 9.91145010305920 \\
17.7634024982886 & 6.47584364565590 & 16.0640864509010 & 17.5486159107531 & 7.70562137370861 \\
13.6288388058580 & 6.74411759716316 & 14.2012968909398 & 11.8376249537951 & 19.8270052266075 \\
12.6121271222844 & 11.8855805421895 & 6.14231789000198 & 13.0217718614950 & 17.6325001345481
\end{matrix}\right.$$

$$\left.\begin{matrix}
14.7026896764278 & 16.6581592592829 & 17.7634024982886 & 13.6288388058580 & 12.6121271222844 \\
12.0449732854894 & 14.8004340755219 & 6.47584364565590 & 6.74411759716316 & 11.8855805421895 \\
14.5027264215831 & 12.0968055881465 & 16.0640864509010 & 14.2012968909398 & 6.14231789000198 \\
15.6000866949538 & 15.6521797575797 & 17.5486159107531 & 11.8376249537951 & 13.0217718614950 \\
9.62150309966732 & 9.91145010305920 & 7.70562137370861 & 19.8270052266075 & 17.6325001345481 \\
102.912724285154 & 13.9592217854341 & 6.65030637918623 & 15.6299996128245 & 16.3450359607153 \\
13.9592217854341 & 106.578021459235 & 7.87956983883075 & 10.6885329046362 & 7.63921835027445 \\
6.65030637918623 & 7.87956983883075 & 113.736168684120 & 8.30021275368670 & 18.8043098692214 \\
15.6299996128245 & 10.6885329046362 & 8.30021275368670 & 108.648495445339 & 20.4504103006764 \\
16.3450359607153 & 7.63921835027445 & 18.8043098692214 & 20.4504103006764 & 117.313312551535
\end{matrix}\right]$$

$$\mathrm{B} = \begin{bmatrix}
0.719862394959852 & 7.19862399356066 \\
0.298498062508485 & 2.98498066864206 \\
0.719943073352362 & 7.19943077821203 \\
0.470645548592634 & 4.70645553655237 \\
0.213065120059020 & 2.13065123100835 \\
0.635136176538378 & 6.35136184705153 \\
0.338215520218286 & 3.38215526612211 \\
0.274120126028595 & 2.74120129843795 \\
0.243954498892080 & 2.43954507177449 \\
0.630536116636262 & 6.30536119819008
\end{bmatrix}$$

VITA

Hao Ji was born in Anhui, China, on August 03, 1985. He received his Bachelor's degree in Mathematics and Applied Mathematics and his Master's degree in Computer Software and Theory from Hefei University of Technology, Hefei, China, in 2007 and 2010, respectively. After that, he began to pursue his Ph.D. degree in the Department of Computer Science at Old Dominion University, Norfolk, VA, USA. His research interests include Monte Carlo Methods for Big Data Analysis, Large-Scale Linear Algebra, and High Performance Scientific Computing.