# RSVDPACK: Subroutines for computing partial singular value decompositions via randomized sampling on single core, multi core, and GPU architectures

Sergey Voronin and Per-Gunnar Martinsson

February 17, 2015

### Abstract

This document describes an implementation in C of set of randomized algorithms for computing partial Singular Value Decompositions (SVDs). The techniques largely follow the prescriptions in the article *"Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," N. Halko, P.G. Martinsson, J. Tropp, SIAM Review, 53(2), 2011, pp. 217-288* but with some modifications to improve performance on three different sets of hardware (1) single core CPU, (2) multi core CPU, and (3) massively multicore GPU.

## 1 Introduction

We describe a software package that computes low-rank approximations to matrices. In other words, given a $m \times n$ matrix $A$, loaded from a binary format on disk into RAM, we seek to compute an approximation $A_{\mathrm{approx}}$ of rank $k < \min(m, n)$. To be precise, the software computes a partial singular value decomposition, which is known to result in a minimal error $||A - A_{\mathrm{approx}}||$ as measured in either the $\ell^2$-operator norm, or the Frobenius norm.

The problem addressed arises frequently in scientific computing and data analysis, and also in statistics, where it is frequently referred to as "principal component analysis (PCA)."

The algorithms used are based on randomized sampling, and are highly computationally efficient, in particular in environments where reducing communication costs is a primary objective. These techniques were originally published in [3], were later extended in [2] and analyzed and surveyed in [1]. In our development, we made minor modifications to previously published versions to improve performance on specific platforms, as detailed in Section 3.

The software is split into three codes, designed for three different hardware platforms: single core, multicore and GPU. Each code implements six different algorithms for computing the low rank SVD. The latest version of the software can be obtained from `https://github.com/sergeyvoronin`.

# 2 The singular value decomposition

This section introduces notation and lists some elementary facts about the SVD.

Let $A$ be an $m \times n$ matrix with real entries. Setting $p = \min(m, n)$, every such matrix admits a so called "singular value decomposition (SVD)" of the form

$$\underset{m \times n}{A} = \underset{m \times p}{U} \ \underset{p \times p}{\Sigma} \ \underset{p \times n}{V^T},$$

(2.1)

where $U$ and $V$ are orthonormal matrices and $\Sigma$ is a diagonal matrix. The columns $(u_j)_{j=1}^p$ and $(v_j)_{j=1}^p$ of $U$ and $V$ are called the left and right singular vectors of $A$, respectively, and the diagonal entries $(\sigma_j)_{j=1}^p$ of $\Sigma$ are the singular values of $A$. The singular values of $A$ are ordered so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$. In other words,

$$U = \begin{bmatrix} u_1 \ u_2 \ \cdots \ u_p \end{bmatrix}, \qquad V = \begin{bmatrix} v_1 \ v_2 \ \cdots \ v_p \end{bmatrix}, \qquad \text{and} \qquad \Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots \\ 0 & \sigma_2 & 0 & \cdots \\ 0 & 0 & \sigma_3 & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix}.$$

It is sometimes convenient to view the factorization (2.1) as a decomposition of $A$ as a sum of $p$ rank-one matrices

$$A = \sum_{j=1}^p \sigma_j \, u_j \, v_j^T.$$

(2.2)

In this note, we are primarily interested in the case where the singular values $\sigma_j$ decays relatively rapidly to zero, meaning that the sum (2.2) converges rapidly. In this case, it is often helpful to approximate $A$ using an approximation $A_k \approx A$ defined by the truncated sum

$$A_k = \sum_{j=1}^k \sigma_j \, u_j \, v_j^T = U_k \, \Sigma_k \, V_k^T,$$

(2.3)

where $k$ is a number less than $p$, and

$$U_k = \begin{bmatrix} u_1 \ u_2 \ \cdots \ u_k \end{bmatrix}, \qquad V_k = \begin{bmatrix} v_1 \ v_2 \ \cdots \ v_k \end{bmatrix}, \qquad \text{and} \qquad \Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ 0 & 0 & \sigma_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & & 0 \\ 0 & 0 & 0 & \cdots & \sigma_k \end{bmatrix}.$$

It is well known that the truncated SVD $A_k$ is the most accurate of all rank-$k$ approximations to $A$, in the following sense:

**Theorem 2.1 (Eckart-Young)** *Let $A$ be an $m \times n$ matrix with singular value decomposition (2.1). Then for any $k$ such that $1 \leq k \leq \min(m, n)$, the truncated SVD $A_k$, as defined by (2.3) is the optimal approximation to $A$ in the sense that*

$$||A - A_k|| = \inf\{||A - B|| : \ B \ \text{has rank } k\},$$

2

where $||\cdot||$ is either the $\ell^2$-operator norm or the Frobenius norm. Moreover,

$$||A - A_k||_2 = \sigma_{k+1}$$

when the error is measured in the $\ell^2$ operator norm, and

$$||A - A_k||_F = \left( \sum_{j=k+1}^{\min(m,n)} \sigma_j^2 \right)^{1/2}$$

when the error is measured in the Frobenius norm.

# 3   Randomized Algorithm

The factors $U_k$, $\Sigma_k$, and $V_k$ in the partial SVD of a matrix $A$, cf. (2.3), can be computed by constructing the *full* SVD of $A$ using standard software routines such as, e.g, those available in a LAPACK implementation. However, this is quite costly, with an asymptotic cost of $O(mnp)$ where $p = \min(m, n)$. The techniques presented here, in contrast, have an asymptotic cost of $O(mnk)$, which represents a substantial savings when $k \ll \min(m, n)$. To be precise, we follow the strategy based on randomized sampling outlined in [1], which roughly consists of the following steps:

- Take $k$ samples of the range of $A$ by multiplying $A$ by random Gaussian vectors to form sample matrix $Y$ of size $m \times k$ such that range $Y \approx$ range $A$.

- Orthogonalize this set of samples forming the matrix $Q$.

- Project the original matrix into a lower dimensional one: $B = Q^T A$ where $B$ is $k \times n$, substantially smaller than $A$, which is $m \times n$.

- Take the SVD of the smaller matrix $B = \tilde{U}_k \Sigma_k V_k^T$.

- Take as low rank SVD of $A$ the product $U_k \Sigma_k V_k^T$ with $U_k = Q\tilde{U}_k$ (since $QQ^T A \approx A$).

The rank $k$ parameter must be chosen such that the range of $Q$ is approximately the same as the range of $A$. In that case, $QQ^T A \approx A$ based on the following simple lemma:

**Lemma 3.1** *Let $A \in \mathbb{R}^{m \times n}$ and let $Q \in \mathbb{R}^{m \times r}$ be an orthonormal matrix. Then the following are equivalent:*

*(1) $\mathcal{R}(A) \subseteq \mathcal{R}(Q)$ (the range of $A$ is a subset of the range of $Q$)*

*(2) $A = QQ^T A$*

**Proof.** Assume (1) holds. Then this implies that there exists a matrix $S \in \mathbb{R}^{r \times n}$ such that $A = QS$. It follows that:

$$QQ^T A = QQ^T QS = QS = A$$

3

since $Q^T Q = I$. Hence, (1) $\implies$ (2).

Next, assume (2) holds. Then:

$$A = QQ^T A = Q\left(Q^T A\right) \implies \mathcal{R}(A) \subseteq \mathcal{R}(Q)$$

Hence, (2) $\implies$ (1). $\square$

We proceed to describe several different variants of the randomized algorithm below, which are implemented in our software package.

## 3.1 Version I

The first modification of the original algorithm computes $U_k$ and $V_k$ without having to take the SVD of the $k \times n$ matrix $B$. Instead, the eigendecomposition of the smaller $k \times k$ matrix $BB^T$ is used. We use the relations:

$$B = \tilde{U}_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T \quad ; \quad B^T = V_k \Sigma_k \tilde{U}_k^T \quad ; \quad Bv_i = \sigma_i \tilde{u}_i$$

$$BB^T = \left(\sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T\right)\left(\sum_{j=1}^{k} \sigma_j \tilde{u}_j v_j^T\right)^T = \sum_{i,j=1}^{k} \sigma_i \sigma_j \tilde{u}_i v_i^T v_j \tilde{u}_j^T = \sum_{i=1}^{k} \sigma_i^2 \tilde{u}_i \tilde{u}_i^T = \tilde{U}_k D \tilde{U}_k^T$$

To compute the right eigenvectors $v_i$, we can use the following relations:

$$B^T \tilde{U}_k = V_k \Sigma_k \tilde{U}_k^T \tilde{U}_k = V_k \Sigma_k \implies V_k = B^T \tilde{U}_k \Sigma^{-1}$$

assuming all the singular values in $\Sigma_k$ are above zero (which is the case for $k$ smaller than the numerical rank). The most complicated operation from a programming standpoint is the eigendecomposition of the symmetric matrix $BB^T$ (which can be easily done with a library call or hand coded Lanczos routine). Since this matrix is small ($k \times k$), the eigendecomposition is not expensive. On the other hand, a simple to implement, but possibly difficult step is to carry out the matrix matrix multiplication $BB^T$. When $B$ is large, this step can be done in a column by column fashion via multiplication with standard basis vectors $B(B^T e_j)$. The disadvantage of this version is that working with the matrix $BB^T$ essentially squares the condition number of $A$, such that small singular values near machine precision may not be properly resolved. This is an issue only if $A$ is expected to have very small singular values amongst $\sigma_1, \ldots, \sigma_k$.

## 3.2 Version II

Here, we use a QR factorization of $B^T$ instead of forming $BB^T$. Out of this we get a $k \times k$ matrix $R$ on which we perform the SVD. This version is based on the following calculations. Let us take the (economic form of the) QR factorization $B^T = \hat{Q}\hat{R}$. Then $\hat{R}$ is $k \times k$ and taking the SVD yields $\hat{R} = \hat{U}\hat{\Sigma}\hat{V}^T$.

$$A \approx QQ^T A = QB = Q\hat{R}^T \hat{Q}^T = Q\hat{V}\hat{\Sigma}\hat{U}^T \hat{Q}^T$$

Thus, the low rank SVD components are $U_k = Q\hat{V}, \Sigma_k = \hat{\Sigma}, V_k = \hat{Q}\hat{U}$.

4

## 3.3 Version III

This version of the algorithm is a slight modification of Version II and is effective in cases when the singular values of the matrix decrease slowly. This does not necessarily imply that the matrix is well conditioned. Rather, imagine a matrix $A$ with singular values $\sigma_1, \ldots, \sigma_n$ where the ratio between the first and last singular value is large. However, there maybe a sequence of singular values $\sigma_{i_1}, \ldots, \sigma_{i_p}$ such that $1 \leq i_1 \leq i_p \leq n$ where the singular values fall of slowly in value. In such case, it is better to use a different method in order to approximate the range of the matrix $A$. In Versions I and II, the matrix $Q$ which we use to project the matrix $A$ into a lower dimensional space is constructed as follows:

$$Y = AR \rightarrow Q = orth(Y)$$

If we don't perform any orthogonalization, then $Q = Y$ and we sample the range of the original matrix $A$. The upper bound on the approximation error with the low rank SVD in this case can be rather large if $\sigma_{k+1}$ is not sufficiently small [1]:

$$||A - U_k \Sigma_k V_k^T||_2 \leq \sqrt{kn} \sigma_{k+1}$$

However, if we instead work with the matrix $\left( (AA^T)^q A \right) R$, where $q$ is some integer greater than one and orthogonalize it to form $Q$, the error bound changes to:

$$||A - U_k \Sigma_k V_k^T||_2 \leq (kn)^{\frac{1}{2(2q+1)}} \sigma_{k+1}$$

Note that $A$ and $(AA^T)^q A$ have the same eigenvectors and related eigenvalues. Plugging in the SVD $A = U\Sigma V^T$ we have:

$$AA^T = U\Sigma^2 U^T \implies (AA^T)^2 = U\Sigma^2 U^T U\Sigma^2 U^T = U\Sigma^4 U^T \implies (AA^T)^q = U\Sigma^{2q} U^T$$
$$\implies (AA^T)^q A = U\Sigma^{2q} U^T U\Sigma V^T = U\Sigma^{2q+1} V^T$$

When $A$ is such that its trailing singular values decay slowly, the matrix $(AA^T)^q A$ has much more rapid decay of singular values. The matrix $Z = (AA^T)^q AR$ can be built up by means of the following iterative procedure:

```
Z = A R;
for j=1:q
    Z = A A^T Z
end
```

In practice, we need to orthonormalize before multiplications with $A$ to prevent computing $M^j$ with $M$ a matrix having singular values greater than one. Thus, the construction of $Q$ used to project the matrix $A$ into a lower dimensional space can proceed as follows:

```
Y = A R
for j=1:q
    Yorth = orth(Y)
    Z = A^T Yorth
```

```
5       Zorth = orth(Z)
6       Y = A Zorth
7   end
8   Q = orth(Y);
```

where the orth operation is performed via a QR factorization (constructing only the factor Q, i.e. stabilized Gram-Schmidt). In many cases, orthogonalization does not need be performed twice at each iteration. In the software, we use a parameter $s$ which controls how often the orthogonalization is done ($s = 1$ corresponds to the above case, $s = 2$ corresponds to forming Yorth but not Zorth, and greater values for $s$ correspond to less frequent orthogonalization).

## 3.4 Autorank Methods

Often, a suitable rank $k$ parameter value is tricky to determine without advance knowledge of the singular value behavior of the matrix. We propose two different methods which do not rely on the rank $k$ parameter and instead determine this parameter automatically using a supplied tolerance parameter. In our software, we include two different strategies for automatically determining the rank $k$ to use for the low rank SVD. In the first approach, we start with lots of samples (via the maxdim parameter) and form a large sample matrix $Y$. Then, we run the Gram-Schmidt orthogonalization procedure until the norm of the projection vectors becomes small. We then scale down the size of our $Q$ matrix and use that to project $A$ to a lower dimensional space.

```
1   R = randn(n,maxdim);
2   Ybig = A*R;
3   Qbig = Ybig;
4   p1 = zeros(maxdim,1);
5   for j=1:maxdim
6       vj = Qbig(:,j);
7       for i=1:(j-1)
8           vi = Qbig(:,i);
9           p = project_vec(vj,vi);
10          vj = vj - p;
11          if norm(p) < TOL && norm(p1) < TOL
12              break;
13              good_rank = j;
14          end
15          p1 = p;
16      end
17      vj = vj/norm(vj);
18      Qbig(:,j) = vj;
19  end
20  Y = Qbig(:,1:good_rank);
21  [Q,~] = qr(Y,0);
```

The disadvantage of this approach is that it can be slow when the singular values fall of slowly. In the second autorank approach, we add blocks of samples at a time to the sample matrix $Y$ and stop adding samples when $\frac{||QQ^TA-A||}{||A||}$ becomes sufficiently small, which by Lemma 3.1 indictates that $Q$ captures the range of $A$.

```
R = randn(n,kblock);
Y = A*R;

exit_loop = 0;
while exit_loop~=1
    [Q,~]=qr(Y,0);
    norm_tol = norm(Q*Q'*A - A)/norm(A);
    if norm_tol > TOL
        R = randn(n,kblock);
        Y_new = A*R;
        Y = [Y, Y_new];
    else
        exit_loop = 1;
    end
end
```

On output, we use the $Q$ to project $A$ to lower dimensional space. The advantage of this scheme is that the block size can be large so this strategy can be efficient even when a large $k$ is ultimately needed.

## 4 Developed Software

In this section, we describe the developed software which has been written to implement Versions I, II, and III of the randomized SVD (RSVD) algorithm along with three autorank versions (approach 1 using the Gram-Schmidt vectors with version II of the algorithm and the block based approach 2 using versions II and III of the algorithm). We have developed three different codes: for single core, multi-core, and GPU architectures. In each case, we have used well known software libraries to implement BLAS (matrix-matrix and matrix-vector operations) and write wrappers for QR, eigendecomposition, and SVD operations. Each of the codes is written using the C programming language. Each code can load a matrix from disk stored using the following simple binary format:

```
num_rows (int)
num_columns (int)
nnz (double)
...
nnz (double)
```

where the nonzeros are listed in the order of a double loop over the rows and columns of the matrix. Note that even zero values are written in this format. It is of course not difficult to extend

7

the codes to support arbitrary matrix formats. The matrix can be loaded using the supplied matrix_load_from_binary_file(char *fname) function. For each code, the low rank SVD can be computed by version I, II, or III or the autorank methods via the following basic call sequence (illustrated here for the multicore MKL code, but very similar for all the 3 codes):

```
// declare matrices
mat *M, *U, *S, *V;

// load matrix M from file
M = matrix_load_from_binary_file(mfile);
m = M->nrows; n = M->ncols;

// set svd rank (< min(m,n))
k = 300;

// call random SVD (use one of these variants)
randomized_low_rank_svd1(M, k, &U, &S, &V);
randomized_low_rank_svd2(M, k, &U, &S, &V);
randomized_low_rank_svd3(M, k, q, s, &U, &S, &V);
randomized_low_rank_svd2_autorank1(M, frac, TOL, &U, &S, &V);
randomized_low_rank_svd2_autorank2(M, blocksize, TOL, &U, &S, &V);
randomized_low_rank_svd3_autorank2(M, blocksize, q, s, TOL, &U, &S, &V);

// write results to disk
matrix_write_to_binary_file(U, "data/U.bin");
matrix_write_to_binary_file(S, "data/S.bin");
matrix_write_to_binary_file(V, "data/V.bin");
```

## 4.1 Single Core Code

The single core code is written using functions from the GNU Scientific Library (GSL). It is a commonly used well documented library that has well optimized functions. The code uses gsl_vector and gsl_matrix objects from the library and corresponding operations with these objects. Matrix-matrix and matrix-vector operations are done using the included BLAS functionality of GSL. This code performs well and is somewhat faster than a similar Octave/Matlab implementation. However, it does not take advantage of multi-processor architectures and runs on a single core of a CPU. It is likely however, that the library will support multiprocessor architectures in the future with the same interface.

## 4.2 Multi Core Code

The multi-core code is written using the Intel Math Kernel Library (MKL) which includes multi-threaded functions for various matrix-vector operations. We define our own matrix and vector objects using the following simple constructions:

```
1  typedef struct {
2      int nrows, ncols;
3      double * d;
4  } mat;
5
6  typedef struct {
7      int nrows;
8      double * d;
9  } vec;
10
11 /* initialize new matrix and set all entries to zero */
12 mat * matrix_new(int nrows, int ncols)
13 {
14     mat *M = malloc(sizeof(mat));
15     M->d = (double*)calloc(nrows*ncols, sizeof(double));
16     M->nrows = nrows;
17     M->ncols = ncols;
18     return M;
19 }
20
21 /* initialize new vector and set all entries to zero */
22 vec * vector_new(int nrows)
23 {
24     vec *v = malloc(sizeof(vec));
25     v->d = (double*)calloc(nrows,sizeof(double));
26     v->nrows = nrows;
27     return v;
28 }
```

Access to matrix and vector elements is then provided via the functions:

```
1  // column major format
2  double matrix_get_element(mat *M, int row_num, int col_num){
3      return M->d[col_num*(M->nrows) + row_num];
4  }
5
6  double vector_get_element(vec *v, int row_num){
7      return v->d[row_num];
8  }
```

For efficiency, we avoid the use of double arrays for matrices, keeping all data in a one dimensional array. This is useful also to parallelize the code for multi-core processors. For this, we use simple OpenMP constructs. For example, the Frobenius norm of a matrix can be calculated using:

```
1  /* matrix frobenius norm */
```

```c
double matrix_frobenius_norm(mat *M){
    int i;
    double val, normval = 0;
    #pragma omp parallel shared(M,normval) private(i,val)
    {
    #pragma omp for reduction(+:normval)
    for(i=0; i<((M->nrows)*(M->ncols)); i++){
        val = M->d[i];
        normval += val*val;
    }
    }
    return sqrt(normval);
}
```

Here we have used OpenMP pragmas to split the large for loop into several smaller independently executed loops, speeding up the calculation when several cores are available. The call to the low rank SVD is performed using a similar construction as in the GSL code. The code automatically distributes work amongst the number of available cores. Alternatively one can specify the number of threads to use via an environmental variable setting:

```
export OMP_NUM_THREADS=6
```

The number of threads should generally be set to the number of available physical cores for best performance. This code is significantly faster than the single core code for larger matrices.

## 4.3   GPU Code

The GPU code is written using the CULA library which implements BLAS and LAPACK functionality using the CUDA toolkit for NVIDIA graphics cards. The code is very similar to the MKL code and shares many of the same functions and constructs, with minor changes to interface with the CULA functions. CULA provides two versions of each function: a host version which transfers data between host and GPU and performs the calculation on the GPU, then transfers the result back to the host; and a device version, which assumes data is already on the GPU and keeps the result on the GPU after computation. The default code uses the host functions and openMP host code. That is for example, the Frobenius norm of the matrix can still be performed with openMP multi-threading on the host. On the other hand expensive functions like QR and SVD are evaluated on the GPU. We have also implemented an experimental device code, which moves data manually between CPU and GPU and calls the device versions of the CULA functions. Our current device code does not offer big performance improvements even for moderately large matrices and we make available only the host code in our software distribution.

# 5 Performance Comparisons

We now present some performance comparisons. First, we compare the approximation error of the three versions of the RSVD algorithm. In Figure 1, we plot $\log \frac{\|A - U_k \Sigma_k V_k^T\|_2}{\|A\|_2}$ versus $k$ for two matrices with different rates of decay of singular values. We observe that versions I and II give nearly identical errors, while version III gives a smaller error for the same value of $k$. When the fall off in singular values is rapid all versions give acceptable results. However, when the tail end singular values are significant, version III is recommended.

We also present tables illustrating sample runtimes of the different codes. We perform our runs on a high-end 6 core PC containing an Intel Xeon E5-2440 chip (6 cores, up to 2.90 Ghz) and an NVIDIA Tesla M2070-Q graphics card. In the tables below, matrix I refers to a dense $2000 \times 4000$ matrix, constructed such that the singular values experience moderate decay. Matrix II is a matrix of the same kind but of larger size $6000 \times 12000$. We first present results for the 3 codes using version I and version II of the randomized algorithm for the two matrices. We notice that the GSL single core code is much slower than the other two codes for the larger matrix II. We also notice that the CULA GPU code outperforms the multi-threaded MKL code for the larger matrix. The CULA GPU code offers even better performance for Version III of the algorithm, when $q > 1$, since the matrix-matrix multiplications required scale well over the many GPU cores. In the third table, we present the runtimes for Version III of the algorithm using the RSVD with $k = 1500$ and different values of $q$ (with $s = 1$, for the most re-orthogonalizations between matrix multiplications). We see that the GPU code has lower runtimes. However, for regular GPU cards, the limited GPU memory (usually under 8 GB) is an issue for larger matrices. If the matrices are so large that they do not entirely fit in GPU memory, performance will be degraded. In such cases, the MKL code would be more applicable.
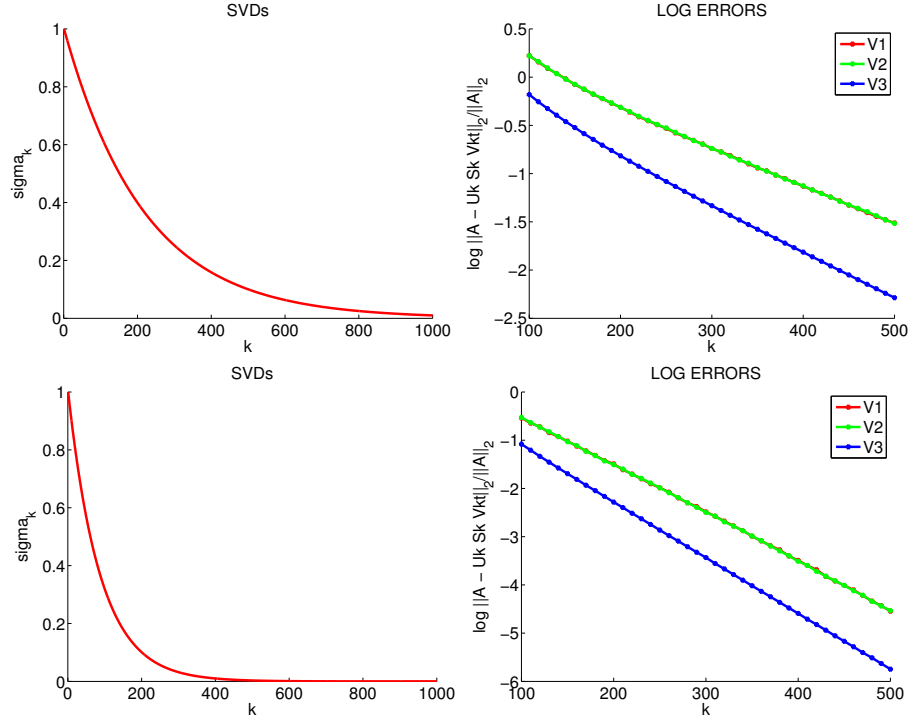
Figure 1: Logarithmic errors vs $k$ with the three RSVD algorithms for two differently conditioned $1000 \times 1500$ matrices.

| $k$ | Single Core (s) | Multi Core (s) | GPU (s) |
|---|---|---|---|
| 300 | 6 | $< 1$ | $< 1$ |
| 600 | 18 | 1 | 1 |
| 1200 | 52 | 2 | 2 |

Runtimes for V1 (matrix I)

| $k$ | Single Core (s) | Multi Core (s) | GPU (s) |
|---|---|---|---|
| 300 | 7 | $< 1$ | $< 1$ |
| 600 | 22 | 1 | 1 |
| 1200 | 83 | 2 | 2 |

Runtimes for V2 (matrix I)

| $k$ | Single Core (s) | Multi Core (s) | GPU (s) |
|---|---|---|---|
| 1500 | 260 | 13 | 4 |
| 3000 | $> 300$ | 31 | 11 |
| 6000 | $> 300$ | 72 | 43 |

Runtimes for V1 (matrix II)

| $k$ | Single Core (s) | Multi Core (s) | GPU (s) |
|---|---|---|---|
| 1500 | 273 | 12 | 5 |
| 3000 | $> 300$ | 34 | 15 |
| 6000 | $> 300$ | 138 | 65 |

Runtimes for V2 (matrix II)

| $q$ | Multi Core (s) | GPU (s) |
|---|---|---|
| 5 | 40 | 12 |
| 10 | 65 | 21 |
| 20 | 102 | 38 |

Runtimes for V3 (matrix II)

# 6 Conclusions

This article presents the mathematical details of RSVDPACK: an open source software package for efficiently computing low rank SVD factorizations of matrices. The package currently implements six different algorithms across three distinct architectures: single core, multi core, and GPU. The provided routines should be suitable for a wide range of applications and the interface can be easily extended to match the necessary input/output requirements.

# 7 Appendix

We present the three different versions of the RSVD algorithm. The autorank strategies discussed in this article can be used with any of these versions of the algorithm.

---

**Algorithm 1:** RSVD Algorithm Version I

---

**Input** : $A \in \mathbb{R}^{m \times n}$ and rank parameter $k < \min(m, n)$.
**Output**: Matrices $U_k \in \mathbb{R}^{m \times k}$, $\Sigma_k \in \mathbb{R}^{k \times k}$, and $V_k \in \mathbb{R}^{k \times n}$.

Initialize a matrix with Gaussian random entries $R \in \mathbb{R}^{n \times k}$;
$Y = AR$;
$[Q, \cdot] = qr(Y, 0)$ (orthonormalize $Y$);
$B = Q^T A$;
$BBt = BB^T$ (this step can be done column by column);
$[\hat{U}, D] = eig(BBt)$ (perform eigendecomposition of $k \times k$ $BBt$ matrix);
$\Sigma_k = \sqrt{D}$;
$U_k = Q\hat{U}$;
$V_k = B^T \hat{U} \Sigma_k^{-1}$ (this step can be done column by column);

---

---

**Algorithm 2:** RSVD Algorithm Version II

---

**Input** : $A \in \mathbb{R}^{m \times n}$ and rank parameter $k < \min(m, n)$.
**Output**: Matrices $U_k \in \mathbb{R}^{m \times k}$, $\Sigma_k \in \mathbb{R}^{k \times k}$, and $V_k \in \mathbb{R}^{k \times n}$.

Initialize a matrix with Gaussian random entries $R \in \mathbb{R}^{n \times k}$;
$Y = AR$;
$[Q, \cdot] = qr(Y, 0)$ (orthonormalize $Y$);
$Bt = A^T Q$;
$[\hat{Q}, \hat{R}] = qr(Bt, 0)$ (compact QR factorization);
$[\hat{U}, \Sigma_k, \hat{V}] = svd(\hat{R})$ (SVD of $k \times k$ matrix);
$U_k = Q\hat{V}$;
$V_k = \hat{Q}\hat{U}$;

---

---

**Algorithm 3:** RSVD Algorithm Version III

---

**Input** : $A \in \mathbb{R}^{m \times n}$, rank parameter $k < \min(m, n)$, and re-orthogonalization amount parameter $s \geq 1$.
**Output**: Matrices $U_k \in \mathbb{R}^{m \times k}$, $\Sigma_k \in \mathbb{R}^{k \times k}$, and $V_k \in \mathbb{R}^{k \times n}$.

Initialize a matrix with Gaussian random entries $R \in \mathbb{R}^{n \times k}$;
$Y = AR$;
**for** $j = 1$ **to** $q$ **do**
    **if** mod $((2j - 2), s) == 0$ **then**
        | $[Y, \cdot] = qr(Y, 0)$;
    **end**
    $Z = A^T Y$;
    **if** mod $((2j - 1), s) == 0$ **then**
        | $[Z, \cdot] = qr(Z, 0)$;
    **end**
    $Y = AZ$;
**end**
$[Q, \cdot] = qr(Y, 0)$ (orthonormalize $Y$);
$Bt = A^T Q$;
$[\hat{Q}, \hat{R}] = qr(Bt, 0)$ (compact QR factorization);
$[\hat{U}, \Sigma_k, \hat{V}] = svd(\hat{R})$ (SVD of $k \times k$ matrix);
$U_k = Q\hat{V}$;
$V_k = \hat{Q}\hat{U}$;

---

# References

[1] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[2] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proc. Natl. Acad. Sci. USA*, 104(51):20167–20172, 2007.

[3] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the approximation of matrices. Technical Report Yale CS research report YALEU/DCS/RR-1361, Yale University, Computer Science Department, 2006.