A Randomized Algorithm for Approximating the Log Determinant of a Symmetric Positive Definite Matrix

Christos Boutsidis∗ Petros Drineas† Prabhanjan Kambadur‡ Anastasios Zouzias§

Abstract

We introduce a novel algorithm for approximating the logarithm of the determinant of a symmetric positive definite matrix. The algorithm is randomized and proceeds in two steps: first, it finds an approximation to the largest eigenvalue of the matrix after running a few iterations of the so-called “power method” from the numerical linear algebra literature. Then, using this information, it approximates the traces of a small number of matrix powers of a specially constructed matrix, using the method of Avron and Toledo [1]. From a theoretical perspective, we present strong worst-case analysis bounds for our algorithm. From an empirical perspective, we demonstrate that a C++ implementation of our algorithm can approximate the logarithm of the determinant of large matrices very accurately in a matter of seconds.

1 Introduction

Given a matrix \( A \in \mathbb{R}^{n \times n} \), the determinant of \( A \), denoted by \( \det(A) \), is one of the most important quantities associated with \( A \). Since its invention by Cardano and Leibniz in the late 16th century, the determinant has been a fundamental mathematical concept with countless applications in numerical linear algebra and scientific computing. The advent of Big Data, which are often represented by matrices, increased the applicability of algorithms that compute, exactly or approximately, matrix determinants; see, for example, [17, 27, 28, 4, 14] for machine learning applications (e.g., gaussian process regression) and [18, 16, 9, 20, 21] for several data mining applications (e.g., spatial-temporal time series analysis).

The first formal definition of the determinant for arbitrary square matrices \( A \) is often attributed to Leibniz. Let \( \pi \) be a permutation of the set \( \{1, 2, \ldots, n\} \) and let \( \Pi_n \) be the set of all \( n! \) such permutations. Let \( \text{sgn}(\pi) \) be the sign function of the permutation \( \pi \), defined as \( \text{sgn}(\pi) = (-1)^{N(\pi)} \), and let \( A_{i, \pi_i} \) denote the \((i, \pi_i)\)-th entry of \( A \) (here \( \pi_i \) is the \( i \)-th entry of \( \pi \)). Using this notation,

\[
\det(A) = \sum_{\pi \in \Pi_n} \text{sgn}(\pi) \cdot \prod_{i=1}^{n} A_{i, \pi_i}.
\]

A second formula for the determinant is attributed to Laplace and was derived in the latter part of the 17th century. The Laplace (or co-factor) expansion, is given by

\[
\det(A) = \sum_{i=1}^{n} A_{i,j} \cdot \det(C^{i,j}).
\]

− Yahoo Labs. New York, NY. Email: boutsidis@yahoo-inc.com
† Rensselaer Polytechnic Institute. Troy, NY. Email: drinep@cs.rpi.edu
‡ Bloomberg L.P., New York, NY. Email: pkambadur@bloomberg.net
§ IBM Research. Zurich, Switzerland. Email: azo@zurich.ibm.com

Here \( N(\pi) \) denotes the number of inversions in \( \pi \). Formally, let \( \pi = (\pi_1, \ldots, \pi_n) \) be a sequence of \( n \) distinct numbers. If \( i < j \) and \( \pi_i > \pi_j \), then the pair \((i, j)\) is called an inversion of \( \pi \).
In the above formula, \( j \) is any index in the set \( \{1, 2, ..., n\} \) and \( C^{[i*,j]} \) denotes the \((n - 1) \times (n - 1)\) matrix derived from \( A \) by removing its \( i \)-th row and its \( j \)-th column. Notice that the Laplace formula is a recursive definition of the determinant.

However, neither the Laplace nor the Leibniz formula can be used to design an efficient, polynomial-time, algorithm to compute the determinant of \( A \). To achieve this goal, one should rely on other properties of the determinant. Two well known such properties are connections between the determinant and the matrix eigenvalues, as well as the connection between the determinant and the so-called \( LU \) matrix decomposition (or the Cholesky decomposition for symmetric matrices). More specifically, it is well known that

\[
\det(A) = \prod_{i=1}^{n} \lambda_i(A),
\]

where \( \lambda_1(A) \geq \lambda_2(A) \geq ... \geq \lambda_n(A) \) are the eigenvalues of \( A \). This property implies an \( O(n^3) \) deterministic algorithm to compute the determinant via the eigendecomposition of \( A \). Alternatively, one can leverage the \( LU \) decomposition of \( A \), e.g., the fact that any matrix \( A \in \mathbb{R}^{n \times n} \) can be expressed as

\[
A = LU,
\]

where \( L \in \mathbb{R}^{n \times n} \) is an upper triangular matrix with diagonal elements equal to one and \( U \in \mathbb{R}^{n \times n} \) is a lower triangular matrix. Using \( \det(XY) = \det(X) \det(Y) \) for any \( X, Y \in \mathbb{R}^{n \times n} \) and the fact that the determinant of an upper or lower triangular matrix is equal to the product of its diagonal elements, we get

\[
\det(A) = \det(LU) = \det(L) \cdot \det(U) = \prod_{i=1}^{n} L_{i,i} \cdot \prod_{i=1}^{n} U_{i,i} = \prod_{i=1}^{n} U_{i,i}.
\]

Since the computation of the \( LU \) decomposition takes \( O(n^3) \) time, the above derivation implies an \( O(n^3) \) deterministic algorithm to compute the determinant of \( A \).

In this paper, we are interested in approximating the logarithm of the determinant of a symmetric positive definite (SPD) matrix \( A \). (Recall that an SPD matrix is a symmetric matrix with strictly positive eigenvalues.) The logarithm of the determinant, instead of the determinant itself, is important in several settings [17, 27, 28, 4, 14, 18, 16, 9, 20, 21].

Definition 1. [LOGDET PROBLEM DEFINITION] Given an SPD matrix \( A \in \mathbb{R}^{n \times n} \), compute (exactly or approximately) \( \log \det(A) \).

The best exact algorithm for the above problem simply computes the determinant of \( A \) in cubic time and takes its logarithm. Few approximation algorithms have appeared in the literature, but they either lack a proper theoretical convergence analysis or do not work for all SPD matrices. We will discuss prior work in detail in Section 1.3.

1.1 Our contributions: theory

We present a fast approximation algorithm for the problem of Definition 1. Our algorithm (Algorithm 3) is randomized and its running time is \( O(\text{nnz}(A) \cdot (m\epsilon^{-2} + \log(n))) \), where \( \text{nnz}(A) \) denotes the number of non-zero elements in \( A \) and (integer) \( m > 0 \) and (real) \( \epsilon > 0 \) are user-controlled accuracy parameters that are specified in the input of the algorithm. The first step of our approximation algorithm uses the power method to compute an approximation to the top eigenvalue of \( A \). This value will be used in a normalization (preconditioning) step in order to compute a convergent matrix-Taylor expansion. The second step of our algorithm leverages a truncated matrix-Taylor expansion of a suitably constructed matrix in order to compute an approximation of the log determinant. This second step leverages randomized trace estimation algorithms [1].
Let \( \hat{\logdet}(A) \) be the value returned by our approximation algorithm (Algorithm 3) and let \( \logdet(A) \) be the true log determinant of \( A \). Then, in Lemma 7 we prove that with constant probability,

\[
| \hat{\logdet}(A) - \logdet(A) | \leq (\varepsilon + (1 - \gamma)^m) \cdot \Gamma,
\]

where

\[
\gamma = \frac{\lambda_n(A)}{\lambda_1(A)},
\]

and

\[
\Gamma = \sum_{i=1}^{n} \log \left( 5 \cdot \frac{\lambda_1(A)}{\lambda_i(A)} \right).
\]

We now take a more careful look at the above approximation bound, starting with the term \((1 - \gamma)^m\). For this term to be “small”, the smallest eigenvalue of \( A \) should be sufficiently large so that the quantity \((1 - \gamma)^m\) converges to zero fast as \( m \) grows. Formally, in order to guarantee \((1 - \gamma)^m \leq \varepsilon\) it suffices to set

\[
m \geq \frac{\log \left( \frac{1}{\varepsilon} \right)}{\log \left( \frac{1}{1 - \gamma} \right)} = \Omega \left( \log \left( \frac{1}{\varepsilon} \right) \cdot \kappa(A) \right),
\]

where \( \kappa(A) = \lambda_1(A)/\lambda_n(A) \) is the condition number of \( A \). The error of our algorithm also scales with \( \Gamma \), which cannot be immediately compared to \( \logdet(A) \), namely the quantity that we seek to approximate.

It is worth noting that the \( \Gamma \) term increases logarithmically with respect to the ratios \( \lambda_1(A)/\lambda_i(A) \geq 1 \); how large these ratios are depends also on the condition number of \( A \) (those ratios are “small” when the condition number is “small”). For example, using majorization, one upper bound for this term is

\[
\Gamma = \sum_{i=1}^{n} \log \left( 5 \cdot \frac{\lambda_1(A)}{\lambda_i(A)} \right) \leq n \cdot \log (5 \kappa(A)).
\]

Note that the above upper bound is, in general, an overestimate of the error of our algorithm. We now state our main theorem.

**Theorem 1.** Fix some accuracy parameter \( \tilde{\varepsilon} \) with \( 0 < \tilde{\varepsilon} < 1 \). Run Algorithm 3 on inputs \( A \),

\[
m = \Omega \left( \log \left( \frac{\log(\kappa(A))}{\tilde{\varepsilon}} \right) \cdot \kappa(A) \right),
\]

and

\[
\varepsilon = \frac{\tilde{\varepsilon}}{2 \log(5 \kappa(A))}.
\]

Let \( \hat{\logdet}(A) \) be the output of Algorithm 3 Then, with probability at least \( 3/16 - 0.01 \),

\[
| \hat{\logdet}(A) - \logdet(A) | \leq \tilde{\varepsilon} \cdot n.
\]

The running time of the algorithm is

\[
\mathcal{O} \left( \text{nnz}(A) \cdot \kappa(A) \cdot \log^2(\kappa(A)) \cdot \frac{1}{\tilde{\varepsilon}^2} \cdot \log \left( \frac{1}{\tilde{\varepsilon}} \right) \cdot \log(n) \right).
\]

### 1.2 Our contributions: practice

We implemented our algorithm in C++ and tested it on several large dense and sparse matrices. Our dense implementation runs on top of Elemental [23], a linear algebra library for distributed matrix computations with dense matrices. Our sparse implementation runs on top of Eigen [24], a software library for sparse matrix computations. This C++ implementation is accessible in both C++ and Python environments (through Python bindings), and it is also available to download on Github (see Section 4.2 for more details).

[23] http://elemental.tuxfamily.org/

[24] http://eigen.tuxfamily.org/
1.3 Related Work

The most relevant result to ours is the work in [2]. Barry and Pace [2] described a randomized algorithm for approximating the logarithm of the determinant of a matrix with special structure that we will describe below. They show that in order to approximate the logarithm of the determinant of a matrix \( A \), it suffices to approximate the traces of \( D^k \) for \( k = 1, 2, 3 \ldots \) for a suitably constructed matrix \( D \). Specifically, [2] deals with approximations to SPD matrices \( A \) of the form \( A = I_n - \alpha D \), where \( 0 \leq \alpha < 1 \) and all eigenvalues of \( D \) are in the interval \([-1, 1]\). Given such a matrix \( A \), the authors of [2] seek to derive an estimator \( \hat{\logdet}(A) \) that is close to \( \logdet(A) \). [2] proved (using the so-called Martin expansion [19]) that

\[
\log(det(A)) = -\sum_{k=1}^{\infty} \frac{\alpha^k}{k} \text{tr}(D^k) = -\sum_{k=m}^{\infty} \frac{\alpha^k}{k} \text{tr}(D^k).
\]

They considered the following estimator:

\[
\hat{\logdet}(A) = \frac{1}{p} \sum_{i=1}^{p} \left( -n \sum_{k=1}^{m} \frac{\alpha^k}{k} \sum_{\alpha \in \mathbb{R}^n} z_i^T D^k z_i \right).
\]

All \( V_i \) for \( i = 1 \ldots p \) are random variables and the value of \( p \) controls the variance of the estimator. The algorithm in [2] constructs vectors \( z_i \in \mathbb{R}^n \) whose entries are independent identically distributed standard Gaussian random variables. The above estimator ignores the trailing terms of the Martin expansion and only tries to approximate the first \( m \) terms. [2] presented the following approximation bound:

\[
|\hat{\logdet}(A) - \logdet(A)| \leq \frac{n \cdot \alpha^{m-1}}{(m+1)(1-\alpha)} + 1.96 \cdot \sqrt{\frac{\sigma^2}{p}},
\]

where \( \sigma^2 \) is the variance of the random variable \( V_i \). The above bound fails with probability at most 0.05.

We now compare the results in [2] with ours. First, the idea of using the Martin expansion [19] to relate the logarithm of the determinant and traces of matrix powers is present in both approaches. Second, the algorithm of [2] is applicable to SPD matrices that have special structure, while our algorithm is applicable to any SPD matrix. Intuitively, we overcome this limitation of [2] by estimating the top eigenvalue of the matrix in the first step of our algorithm. Third, our error bound is much better that the error bound of [2]. To analyze our algorithm, we used the theory of randomized trace estimators of Avron and Toledo [1], which relies on powerful matrix-Chernoff bounds, while [2] uses the weaker Chebyshev’s inequality.

A similar idea using Chebyshev polynomials appeared in the paper [22], to our best understanding, there are no theoretical convergence properties of the proposed algorithm. Applications to Gaussian process regression appeared in [17][27][28]. The work of [24] uses an approximate matrix inverse to compute the \( n \)-th root of the determinant of \( A \) for large sparse SPD matrices; the error bounds in this work are a posteriori and thus not directly comparable to our bounds.

Recent work in [15] provides a strong worst-case theoretical result which is, however, only applicable to Symmetric Diagonally Dominant (SDD) matrices. The algorithm is randomized and guarantees that, with high probability, \(|\hat{\logdet}(A) - \logdet(A)| \leq \varepsilon \cdot n\) for a user specified error parameter \( \varepsilon > 0 \). This approach also uses the Martin expansion [19] as well as ideas from preconditioning systems of linear equations with Laplacian matrices [26]. The algorithm of [15] runs in time \( O(|\text{nnz}(A)| \varepsilon^{-2} \log^3 (n) \log^2 (n\kappa(A)/\varepsilon)) \). Comparing to our approach, we note that the above running time depends logarithmically on the condition number of the input matrix \( A \), whereas our algorithm has a linear dependency on the condition number. Notice, however, that our method is applicable to any SPD matrix while the method in [15] is applicable only to SDD matrices; given current state-of-the-art on Laplacian preconditioners it looks hard to extend the approach of [15] to general SPD matrices.

We conclude by noting that the aforementioned algorithms for the determinant computation assume floating point arithmetic and do not measure bit operations. If the computational cost is to be measured in bit operations, the situation is much more complicated and an exact computation of the determinant, even for integer matrices, is not trivial. We refer the interested reader to [8] for more details.
2 Preliminaries

This section summarizes notation and prior work that we will use in the paper.

2.1 Notation

\(A, B, \ldots\) denote matrices; \(a, b, \ldots\) denote column vectors. \(I_n\) is the \(n \times n\) identity matrix; \(0_{m \times n}\) is the \(m \times n\) matrix of zeros; \(\text{tr}(A)\) is the trace of a square matrix \(A\); the Frobenius and the spectral matrix-norms are: 
\[
\|A\|_F^2 = \sum_{i,j} A_{ij}^2 \quad \text{and} \quad \|A\|_2 = \max_{\|x\|=1} \|Ax\|_2.
\]
We denote the determinant of a matrix \(A\) by \(\det(A)\) and the logarithm (base two) of the determinant of \(A\) by \(\logdet(A)\). All logarithms in the paper are base two.

2.2 SPD matrices and eigenvalues

Let \(A\) be an \(n \times n\) symmetric matrix and let \(\lambda_i(A)\) denote the \(i\)-th eigenvalue of \(A\) for all \(i = 1, \ldots, n\) with 
\[
\|A\|_2 = \lambda_1(A) \geq \lambda_2(A) \geq \ldots \geq \lambda_n(A) \geq 0.
\]
A symmetric matrix \(A\) is called positive definite (SPD) if all its eigenvalues are positive (e.g., \(\lambda_n(A) > 0\)).

2.3 Matrix logarithm

For an SPD matrix \(A \in \mathbb{R}^{n \times n}\), \(\log[A]\) is an \(n \times n\) matrix defined as:
\[
\log[A] = UDU^T,
\]
where \(U \in \mathbb{R}^{n \times n}\) contains the eigenvectors of \(A\) and \(D \in \mathbb{R}^{n \times n}\) is diagonal:
\[
D = \begin{pmatrix}
\log(\lambda_1(A)) & & \\
& \log(\lambda_2(A)) & \\
& & \ddots \\
& & & \log(\lambda_n(A))
\end{pmatrix}.
\]

2.4 Matrix Taylor expansion

Let \(x\) be a scalar variable that satisfies \(|x| < 1\). Then,
\[
\ln(1 - x) = -\sum_{k=1}^{\infty} x^k / k.
\]
A matrix-valued generalization of this identity is the following statement.

**Lemma 2.** Let \(A \in \mathbb{R}^{n \times n}\) be a symmetric matrix whose eigenvalues all lie in the interval \((-1, 1)\). Then,
\[
\log(I_n - A) = -\sum_{k=1}^{\infty} A^k / k.
\]

2.5 Power method

The first step in our algorithm for approximating the determinant of an SPD matrix is to obtain an estimate for the largest eigenvalue of the matrix. To achieve this we employ the so-called “power method”. Given an SPD matrix \(A \in \mathbb{R}^{n \times n}\) we will use Algorithm 1 to obtain an accurate estimate of its largest eigenvalue. This estimated eigenvalue is denoted by \(\hat{\lambda}_1(A)\).

Algorithm 1 requires \(O(t(n + nnz(A)))\) arithmetic operations to compute \(\hat{\lambda}_1(A)\). The following lemma argues that, for a sufficiently large \(t\), \(\hat{\lambda}_1(A)\) is “close” to \(\lambda_1(A)\).

---

3See [http://www.eecs.berkeley.edu/~luca/cs359g/lecture07.pdf](http://www.eecs.berkeley.edu/~luca/cs359g/lecture07.pdf) for a proof.
Algorithm 1 Power method

- Input: SPD matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, integer $t > 0$
  1. Pick uniformly at random $\mathbf{x}_0 \in \{+1, -1\}^n$
  2. For $i = 1, \ldots, t$
     - $\mathbf{x}_i = \mathbf{A} \cdot \mathbf{x}_{i-1}$
     - $\mathbf{x}_i = \mathbf{x}_i / \|\mathbf{x}_i\|_2$
  3. $\tilde{\lambda}_1(\mathbf{A}) = \mathbf{x}_t^T \mathbf{A} \mathbf{x}_t$
- Return: $\tilde{\lambda}_1(\mathbf{A})$

Algorithm 2 Randomized Trace Estimation

- Input: SPD matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, accuracy parameter $0 < \varepsilon < 1$, and failure probability $0 < \delta < 1$
  1. Let $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_p$ be a set of independent Gaussian vectors in $\mathbb{R}^n$
  2. Let $p = 20 \ln(2/\delta)/\varepsilon^2$
  3. Let $\gamma = 0$
  4. For $i = 1, \ldots, p$
     - $\gamma = \gamma + \mathbf{g}_i^T \mathbf{A} \mathbf{g}_i$
  5. $\gamma = \gamma/p$
- Return: $\gamma$

Lemma 3. For any $t > 0, \varepsilon > 0$, with probability at least $3/16$:

$$\lambda_1(\mathbf{A}) \cdot (1 - \varepsilon) \cdot \frac{1}{1 + 4n(1 - \varepsilon)} \leq \tilde{\lambda}_1(\mathbf{A}) \leq \lambda_1(\mathbf{A}).$$

An immediate corollary to this lemma follows.

Corollary 4. Set $\varepsilon = 1/2$ and $t = \log(4n)$. Then, with probability at least $3/16$: $\lambda_1(\mathbf{A}) \cdot \frac{1}{4} \leq \tilde{\lambda}_1(\mathbf{A}) \leq \lambda_1(\mathbf{A})$.

2.6 Trace Estimation

Though computing the trace of a square $n \times n$ matrix requires only $O(n)$ arithmetic operations, the situation is more complicated when $\mathbf{A}$ is given through a matrix function, e.g., $\mathbf{A} = \mathbf{X}^2$, for some matrix $\mathbf{X}$ and the user only observes $\mathbf{X}$. For situations such as these, Avron and Toledo analyzed several algorithms to estimate the trace of $\mathbf{A}$. The following lemma presents a relevant result from their paper.

Lemma 5 (Theorem 5.2 in [1]). Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be an SPD matrix, let $0 < \varepsilon < 1$ be an accuracy parameter, and let $0 < \delta < 1$ be a failure probability. If $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_p \in \mathbb{R}^n$ are independent random standard Gaussian vectors, then for all $p > 0$ with probability at least $1 - e^{-p\varepsilon^2/20}$:

$$\left| \text{tr}(\mathbf{A}) - \frac{1}{p} \sum_{i=1}^{p} \mathbf{g}_i^T \mathbf{A} \mathbf{g}_i \right| < \varepsilon \cdot \text{tr}(\mathbf{A}).$$

The lemma indicates that choosing $p = 20 \ln(2/\delta)/\varepsilon^2$ and approximating the trace of $\mathbf{A}$ by $\frac{1}{p} \sum_{i=1}^{p} \mathbf{g}_i^T \mathbf{A} \mathbf{g}_i$ results in a relative error approximation with probability at least $1 - \delta$. Algorithm 2 is a detailed description of the above procedure.
3 Main Result

Lemma 6 is the starting point of our main algorithm for approximating the determinant of a symmetric positive definite matrix. The message in the lemma is that computing the log determinant of an SPD matrix \( A \) reduces to the task of computing the largest eigenvalue of \( A \) and the trace of all the powers of a matrix \( C \) related to \( A \).

Lemma 6. Let \( A \in \mathbb{R}^{n \times n} \) be an SPD matrix. For any \( \alpha \) with \( \lambda_1(A) < \alpha \), define

\[
B := A/\alpha \quad \text{and} \quad C := I_n - B.
\]

Then,

\[
\log \det(A) = n \log(\alpha) - \sum_{k=1}^{\infty} \frac{\text{tr}(C^k)}{k}.
\]

Proof. Observe that \( B \) is an SPD matrix with \( \|B\|_2 < 1 \). It follows that

\[
\log \det(A) = \log(\alpha^n \det(A/\alpha)) = n \log(\alpha) + \log(\prod_{i=1}^{n} \lambda_i(B)) = n \log(\alpha) + \sum_{i=1}^{n} \log(\lambda_i(B)) = n \log(\alpha) + \text{tr}(\log B).
\]

Here, we used standard properties of the determinant, standard properties of the logarithm function, and the fact that (recall that \( B \) is an SPD matrix),

\[
\text{tr}(\log B) = \sum_{i=1}^{n} \lambda_i(\log B) = \sum_{i=1}^{n} \log(\lambda_i(B)).
\]

Now,

\[
\text{tr}(\log B) = \text{tr}(\log [I_n - (I_n - B)]) = \text{tr}\left(-\sum_{k=1}^{\infty} (I_n - B)^k / k\right) = -\sum_{k=1}^{\infty} \text{tr}(C^k) / k.
\]

The second equality follows by Lemma 2 because all the eigenvalues of \( C = I_n - B \) are contained in \((0, 1)\) and the last equality follows by the linearity of the trace operator.

3.1 Algorithm

Lemma 6 indicates the following “high-level” procedure for computing the logdet of an SPD matrix \( A \):

1. Compute some \( \alpha \) with \( \lambda_1(A) < \alpha \).
2. Compute \( C = I_n - A/\alpha \).
3. Compute the trace of all the powers of \( C \).

\[\text{Indeed, } \lambda_i(C) = 1 - \lambda_i(B) \text{ and } 0 < \lambda_i(B) < 1 \text{ for all } i = 1 \ldots n.\]
To implement the first step in this procedure we use the “power iteration” from the numerical linear algebra literature (see Section 2.5). The second step is straightforward. To implement the third step, we keep a finite number of summands in the expansion \( \sum_{k=1}^{\infty} \text{tr}(C^k) \). This step is important since the quality of the approximation, both theoretically and empirically, depends on the number of summands (denoted with \( m \)) that will be kept. On the other hand, the running time of the algorithm increases with \( m \). Finally, to estimate the traces of the powers of \( C \), we use the randomized algorithm of Section 2.6. Our approach is described in detail in Algorithm 3.

**Algorithm 3 Randomized Log Determinant Estimation**

1. **INPUT:** \( A \in \mathbb{R}^{n \times n} \), accuracy parameter \( \varepsilon > 0 \), and integer \( m > 0 \).
2. Compute \( \lambda_1(A) \) using the method of Section 2.5 with \( t = \log(4n) \).
3. Pick \( \alpha = 5\lambda_1(A) \) (notice that \( \lambda_1(A) < \alpha \leq 5\lambda_1(A) \)).
4. Set \( C = I - A/\alpha \).
5. Set \( p = 20 \ln(200)/\varepsilon^2 \).
6. Let \( g_1, g_2, \ldots, g_p \in \mathbb{R}^n \) be i.i.d. random Gaussian vectors.
7. For \( i = 1, 2, \ldots, p \)
   - \( v_1^{(i)} = C g_i \) and \( \gamma_1^{(i)} = g_i^\top v_1^{(i)} \)
   - For \( k = 2, \ldots, m \)
     1. \( v_k^{(i)} := Cv_k^{(i-1)} \).
     2. \( \gamma_k^{(i)} = g_i^\top v_k^{(i)} \) (Inductively \( \gamma_k^{(i)} = g_i^\top C v_1^{(i)} \)).
   - EndFor
8. **OUTPUT:** \( \logdet(A) \) as
   \[
   \logdet(A) = n \log(\alpha) - \sum_{k=1}^{m} \left( \frac{1}{p} \sum_{i=1}^{p} \gamma_k^{(i)} \right) / k
   \]

Notice that step 7 in the above algorithm is an efficient way of computing
\[
\logdet(A) := n \log(\alpha) - \sum_{k=1}^{m} \left( \frac{1}{p} \sum_{i=1}^{p} g_i^\top C^k g_i \right) / k.
\]

### 3.2 Running time

Step 2 takes \( O(\log(n) \cdot \text{nnz}(A)) \) time. For each \( k > 0 \), \( v_k^{(i)} = C^k g_i \). The algorithm inductively computes \( v_k^{(i)} \) and \( g_i^\top C^k g_i = g_i^\top v_k^{(i)} \) for all \( k = 1, 2, \ldots, m \). Given \( v_{k-1}^{(i)} \), \( v_k^{(i)} \) and \( g_i^\top C^k g_i \) can be computed in \( \text{nnz}(C) \) and \( O(n) \) time, respectively. Notice that \( \text{nnz}(C) \leq n + \text{nnz}(A) \). Therefore, step 7 requires \( O(p \cdot m \cdot \text{nnz}(A)) \) time. Since \( p = O(\varepsilon^{-2}) \), the total cost is \( O((m\varepsilon^{-2} + \log(n)) \cdot \text{nnz}(A)) \).

### 3.3 Error bound

The following lemma proves that Algorithm 3 returns an accurate approximation to the logdet of \( A \).

**Lemma 7.** Let \( \hat{\logdet}(A) \) be the output of Algorithm 3 on inputs \( A, m, \) and \( \varepsilon \). Then, with probability at least \( 3/16 - 0.01 \),
\[
\left| \logdet(A) - \hat{\logdet}(A) \right| \leq (\varepsilon + (1 - \gamma)^m) \cdot \Gamma,
\]
where
\[
\gamma = \frac{\lambda_n(A)}{\lambda_1(A)},
\]

8
and
\[
\Gamma = \sum_{i=1}^{n} \log \left( 5 \cdot \frac{\lambda_i(A)}{\lambda_1(A)} \right).
\]

**Proof.** We can derive our error bound as follows:
\[
\left| \log \det (A) - \log \det (X) \right| = \left| \sum_{k=1}^{m} \frac{1}{p} \frac{1}{p} \sum_{i=1}^{p} g_i^\top C_k g_i / k - \sum_{k=1}^{\infty} \frac{\text{tr} (C^k)}{k} \right|
\leq \sum_{k=1}^{m} \frac{1}{p} \frac{1}{p} \sum_{i=1}^{p} g_i^\top C_k g_i / k - \sum_{k=1}^{\infty} \frac{\text{tr} (C^k)}{k}
\]
\[
\left. + \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^k)}{k} \right|_{\Gamma_1}
\]
\[
\left. + \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^k)}{k} \right|_{\Gamma_2}.
\]

Below, we bound the two terms \(\Gamma_1\) and \(\Gamma_2\) separately. We start with \(\Gamma_1\): the idea is to apply Lemma 5 on the matrix \(\sum_{k=1}^{m} C_k / k\) with \(\delta = 10^{-2}\) and \(p = 20 \ln(200)/\varepsilon^2\). Hence, with probability at least 0.99:
\[
\Gamma_1 \leq \varepsilon \cdot \text{tr} \left( \sum_{k=1}^{m} C_k / k \right) \leq \varepsilon \cdot \text{tr} \left( \sum_{k=1}^{\infty} C_k / k \right).
\]

In the last inequality we used the fact that \(C\) is a positive matrix, hence for all \(k\), \(\text{tr} (C^k) > 0\). The second term \(\Gamma_2\) is bounded as:
\[
\Gamma_2 = \left| \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^k)}{k} \right|
\leq \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^k)}{k}
\leq \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^m \cdot C^{k-m})}{k}
\leq \sum_{k=m+1}^{\infty} \|C^m\|_2 \cdot \frac{\text{tr} (C^{k-m})}{k}
\leq \|C^m\|_2 \cdot \sum_{k=m+1}^{\infty} \frac{\text{tr} (C^{k-m})}{k}
\leq \|C^m\|_2 \cdot \sum_{k=1}^{\infty} \frac{\text{tr} (C^k)}{k}
\leq \left(1 - \frac{\lambda_n(A)}{\alpha} \right)^{\frac{m}{\alpha}} \cdot \sum_{k=1}^{\infty} \frac{\text{tr} (C^k)}{k}.
\]
In the first inequality, we used the triangle inequality and the fact that \( C \) is a positive matrix. In the second inequality, we used the following fact\(^5\): given two positive semidefinite matrices \( A, B \) of the same size, \( \text{tr}(AB) \leq \|A\|_2 \cdot \text{tr}(B) \). In the last inequality, we used the fact that 
\[
\lambda_1(C) = 1 - \lambda_n(B) = 1 - \frac{\lambda_n(A)}{\alpha}.
\]
The bound for \( \Gamma_2 \) is deterministic.

Combining the bounds for \( \Gamma_1 \) and \( \Gamma_2 \) gives that with probability at least \( 3/16 - 0.01 \),
\[
\left| \hat{\logdet}(A) - \logdet(A) \right| \leq \left( \epsilon + \left( 1 - \frac{\lambda_n(A)}{\alpha} \right)^m \right) \cdot \sum_{k=1}^{\infty} \frac{\text{tr}(C^k)}{k}.
\]
We have already proven in Lemma 3 that
\[
\sum_{k=1}^{\infty} \frac{\text{tr}(C^k)}{k} = -\text{tr}(\log|B|) = n \log(a) - \logdet(A).
\]

We further manipulate the last term as follows:
\[
n \log(a) - \logdet(A) = n \log(\alpha) - \log(\prod_{i=1}^{n} \lambda_i(A))
\]
\[
= n \log(\alpha) - \sum_{i=1}^{n} \log(\lambda_i(A))
\]
\[
= \sum_{i=1}^{n} (\log(\alpha) - \log(\lambda_i(A)))
\]
\[
= \sum_{i=1}^{n} \log \left( \frac{\alpha}{\lambda_i(A)} \right)
\]
Collecting our results together, we get:
\[
\left| \hat{\logdet}(A) - \logdet(A) \right| \leq \left( \epsilon + \left( 1 - \frac{\lambda_n(A)}{\alpha} \right)^m \right) \cdot \sum_{i=1}^{n} \log \left( \frac{\alpha}{\lambda_i(A)} \right).
\]
Using the relation \( \lambda_1(A) < \alpha \leq 5\lambda_1(A) \), concludes the proof.

4 Experiments

The goal of our experimental section is to establish that our approximation to \( \logdet(A) \) (Algorithm 3) is both accurate and fast for both dense and sparse matrices. The accuracy of Algorithm 3 is measured by comparing its result against the “exact” \( \logdet(A) \) computed using Cholesky factorization. The rest of this section is laid out as follows: in Section 4.1, we describe our software for approximating \( \logdet(A) \) and in Sections 4.3 and 4.4, we discuss experimental results for dense and sparse SPD matrices, respectively.

4.1 Software

We developed high-quality, shared- and distributed-memory parallel C++ code for the algorithms listed in this paper; python bindings for our code are in writing, and we hope to complete it soon. All of the code that was developed for this paper (which continues to be improved) is hosted at [https://github.com/pkambadu/ApproxLogDet](https://github.com/pkambadu/ApproxLogDet). In its current state, our software supports: (1) ingesting dense (binary and text

\(^5\)This is easy to prove using Von Neumann’s trace inequality.
Figure 1: Panels 1(a) and 1(b) depict the effect of $m$ (see Algorithm 3) on the accuracy of the approximation and the time to completion, respectively, for dense matrices generated by randSPDDense. For all the panels, $p = 60$, $t = \log 4n$. The baseline for all experiments was Cholesky factorization, which was used to compute the exact value of $\log det (A)$. For panels 1(a) and 1(b), the number of cores, $np$, was set to 1. The last panel 1(c) depicts the relative speed of the approximate algorithm when compared to the baseline (at $m = 4$). Elemental was used as the backend for these experiments. For the approximate algorithm, we report the mean and standard deviation of 10 iterations; each iteration used different random numbers.

format) matrices and sparse (binary, text, and matrix market format) matrices, (2) generating large random symmetric positive definite matrices, (3) computing both approximate and exact spectral norms of matrices, (4) computing both approximate and exact traces of powers matrices, and (5) computing both approximate and exact log determinant of matrices. Currently, we support both Eigen [11] and Elemental [23] matrices. Eigen package supports both dense and sparse matrices; Elemental strongly supports dense matrices and only recently added support for sparse matrices (pre-release). As we wanted the random SPD generation to be fast, we have used parallel random number generators from Random123 [25] in conjunction with Boost.Random.

4.2 Environment

All experiments were run on “Nadal”, which is a 60-core machine, where each core is an Intel® Xeon® E7-4890 machine running at 2.8 Ghz. Nadal has 1 tera-byte of RAM and runs Linux kernel version 2.6-32. For compilation, we used GCC 4.9.2. We used Eigen 3.2.4, OpenMPI 1.8.4, Boost 1.55.7, and the latest version of Elemental at https://github.com/elemental For experiments with Elemental, we used OpenBlas, which is an extension of GotoBlas [10], for its parallel prowess; Eigen provides built-in BLAS and LAPACK.

4.3 Dense Matrices

Data Synthesis In our experiments, we used two types of synthetic SPD matrices. The first kind were diagonally dominant and were generated as follows. First, we create $X \in \mathbb{R}^{n \times n}$ by drawing $n^2$ entries from a uniform sphere with center 0.5 and radius 0.25. We generate symmetric $Y$ by setting

$$Y = 0.5 * (X + X^T).$$

Finally, we ensure that the desired matrix $A$ is positive definite by adding $n$ to each of the diagonal entries of $Y$ [8]:

$$A = Y + n \cdot I_n.$$ 

We call this method randSPDDenseDD.

The second approach generates SPD matrices that are not diagonally dominant. We create $X, D \in \mathbb{R}^{n \times n}$ by drawing $n^2$ and $n$ entries, respectively, from a uniform sphere with center 0.5 and radius 0.25; $D$ is a
Table 1: Accuracy and sequential running times (at \( p = 60, m = 4, t = \log 4n \)) for dense random matrices generated using \texttt{randSPDDense}. Baselines were computed using Cholesky factorization; mean and standard deviation are reported for 10 iterations; each iteration reset the random processes in Algorithm 3.

| \( n \) | \( \log \det (A) \) | time (secs) |
|-------|----------------|-------------|
|       | exact | mean | std | exact | mean | std |
| 5000  | -3717.89 | -3546.920 | 8.10 | 2.56 | 1.15 | 0.0005 |
| 7500  | -5474.49 | -5225.152 | 8.73 | 7.98 | 2.53 | 0.0015 |
| 10000 | -7347.33 | -7003.086 | 7.79 | 18.07 | 4.47 | 0.0006 |
| 12500 | -9167.47 | -8734.956 | 17.43 | 34.39 | 7.00 | 0.0030 |
| 15000 | -11100.9 | -10575.16 | 15.09 | 58.28 | 10.39 | 0.0102 |

Figure 2: Panels 2(a) and 2(b) depict the effect of \( m \) (see Algorithm 3) on the accuracy of the approximation and the time to completion, respectively, for diagonally dominant dense random matrices generated by \texttt{randSPDDenseDD}. For all the panels, \( p = 60, t = \log 4n \). The baseline for all experiments was Cholesky factorization, which was used to compute the exact value of \( \log \det (A) \). For panels 2(a) and 2(b), the number of cores, \( np \), was set to 1. The last panel 2(c) depicts the relative speed of the approximate algorithm when compared to the baseline (at \( m = 2 \)). Elemental was used as the backend for these experiments. For the approximate algorithm, we report the mean and standard deviation of 10 iterations; each iteration used different random numbers.

Evaluation To compare the runtime of Algorithm 3, we use Cholesky decomposition to compute a factorization and then use the diagonal elements of the factor to compute \( \log \det (A) \); that is, we compute \( L = \text{chol}(A) \) and \( \log \det (A) = 2 \times \log \det (L) \). As Elemental provides distributed and shared memory parallelism, we restrict ourselves to experiments with Elemental matrices throughout this section. Note that we measure accuracy of the approximate algorithm in terms of relative error to ensure that we have numbers of the same scale for matrices with vastly different values for \( \log \det (A) \). We define the relative error \( e \) thusly: \( e = 100 \times \frac{x - \hat{x}}{x} \), where \( x \) is the true value and \( \hat{x} \) is the approximation. Similarly, we define the speedup \( s \) thusly: \( s = t_x / t_{\hat{x}} \), where \( t_x \) is the time to compute \( x \) and \( t_{\hat{x}} \) is the time to compute the approximation \( \hat{x} \); this definition of speedup is used both for parallel speedup and for speedup resulting from the approximation.

Results For dense matrices, we exclusively used synthetic data generated using both \texttt{randSPDDense} and \texttt{randSPDDenseDD}. We first discuss the results of the relatively “ill-conditioned” matrices with \( n \), the number...
of rows and columns being sizes 5000, 7500, 10000, 12500, and 15000 that were generated using randSPDDenseDD. The three key points pertaining to these matrices are shown in Figure 1. First, we discuss the effect of \( m \), the number of terms in the Taylor series used to approximate \( \logdet(A) \); panel 1(a) depicts our results for the sequential case. On the y-axis, we see the relative error, which is measured against the exact \( \logdet(A) \) as computed using Cholesky factorization. We see that for these “ill-conditioned” matrices, for small values of \( m < 4 \), the relative error (in %) is high. However, for all values of \( m >= 4 \), we see that the error drops down significantly and stabilizes. Note that in each iteration, all random processes were re-seeded with new values; we have plotted the error bars throughout Figure 1. However, the standard deviation in both accuracy and time was consistently small; indeed, it is not visible to the naked eye at scale. To see the benefit of approximation, we look at Figure 1(b) together with Figure 1(a). For example, at \( m = 4 \), for all matrices, we get at least a factor of 2 speedup. However, as can be seen, as \( n \) gets larger, the speedups afforded by the approximation also increase. For example, for \( n = 15000 \), the speedup at \( m = 4 \) is nearly 6x; from panel 1(a) we see that at \( m = 4 \), the relative error between the exact and approximate quantities of \( \logdet(A) \) is around 4%. This speedup is expected as Cholesky factorization requires \( O(n^3) \) operations; the approximation only relies on matrix-matrix products where one of the matrices have a small number of columns \( (p) \). Notice that \( p \) depends entirely on \( \delta \) and \( \varepsilon \), and is subsequently independent from \( n \); this makes Algorithm 3 scalable. Finally, we discuss the parallel speedup in panel 1(c) which shows the relative speedup of the approximate algorithm with respect to the baseline Cholesky algorithm. For this, we set \( m = 4 \) and varied the number of processes, \( np \), from 1 to 60. The main take away from panel 1(c) is that the approximate algorithm provides nearly the same or increasingly better speedups relative to a parallelized version of the exact (Cholesky) algorithm. For example, for \( n = 15000 \), the speedups for using the approximate algorithm are consistently better than 6.5x. The absolute values for \( \logdet(A) \) and timing along with the baseline numbers for this experiment are given in Table 1. We report the numbers in Table 1 at \( m = 4 \) at which point, we have low relative error.

For the second set of dense experiments, we generated diagonally dominant matrices using randSPDDenseDD; we were able to quickly generate and run benchmarks on matrices of \( n = 10000, 20000, 30000, 40000 \) due to the relatively simpler procedure involved in matrix generation. In this set of experiments, due to the diagonal dominance, all matrices were “well-conditioned”. The results of our experiments on these “well-conditioned” matrices are presented in Figure 2 and show a marked improved over the results presented in Figure 1. First, notice that very few terms of the Taylor series (i.e., small \( m \)) are sufficient to get high accuracy approximations; this is apparent from panel 2(a). In fact, we see that even at \( m = 2 \), we are near convergence and at \( m = 3 \), for most of the matrices, we have near-zero relative error. This experimental result, when taken together with panel 2(b) is encouraging; at \( m = 2 \), we seem to not only have near-exact approximation of \( \logdet(A) \), but also have at least 5x speedup. Like in Figure 1, the speedups are better for larger matrices. For example, for \( n = 40000 \), the speedup at \( m = 2 \) is nearly 20x. We conclude our analysis by presenting panel 2(c) which like panel 1(c) drives home the point that at any level of parallelism, Algorithm 3 maintains its relative performance over the exact (Cholesky) factorization. The absolute values for \( \logdet(A) \) and timing along with the baseline for this experiment are given in Table 2. We report the numbers in Table 2 at \( m = 2 \) at which point, we have low relative error.

| \( n \) | \( \logdet(A) \) | \( \text{time (secs)} \) |
|-------|----------------|-------------------|
|       | exact | mean | std | exact | mean | std |
| 10000 | 92103.1 | 92269.5 | 5.51 | 18.09 | 2.87 | 0.01 |
| 20000 | 198069.0 | 198397.4 | 9.60 | 135.92 | 12.41 | 0.02 |
| 30000 | 309268.0 | 309763.8 | 20.04 | 448.02 | 30.00 | 0.12 |
| 40000 | 423865.0 | 424522.4 | 14.80 | 1043.74 | 58.05 | 0.05 |

Table 2: Accuracy and sequential running times (at \( p = 60, m = 2, t = \log(4n) \)) for diagonally dominant dense random matrices generated using randSPDDenseDD. Baselines were computed using Cholesky factorization; mean and standard deviation are reported for 10 iterations; each iteration reset the random processes in Algorithm 3.
### Table 3: Description of the SPD matrices from the University of Florida sparse matrix collection \[7\] that were used in our experiments. All experiments were run sequentially \((np = 1)\) using Eigen. Accuracy for the approximation (Algorithm 3) are the mean and standard deviation of 10 iterations at \(t = 5\) and \(p = 5\); only the mean for the time is reported as the standard deviation was negligible. The exact \(\logdet(A)\) was computed using Cholesky factorization.

| name            | \(n\)   | \(nnz\) | area of origin | \(\logdet(A)\)     | time (sec) | \(m\) |
|-----------------|---------|---------|----------------|---------------------|------------|-------|
|                 | exact   | approx  | exact | approx | mean | std | exact | approx | mean | std |                      |
| thermal2        | 1228045 |         | Thermal   | 1.3869e6          | 964.79     | 31.28 | 31.24 | 149    |
| ecology2        | 999999  |         | 2D/3D    | 3.3943e6          | 1212.8     | 18.5  | 10.47 | 125    |
| ldoor           | 952203  | 42493817| Structural | 1.4429e7          | 1683.5     | 117.91| 17.60 | 33     |
| thermomech_TC   | 102158  | 711558  | Thermal   | -546829.4         | 18.56      | 553.12| 247.14| 33     |
| boneS01         | 127224  | 5516602 | Engineered | 1.106e6          | 247.14     | 130.4 | 8.48  | 77     |

### 4.4 Sparse Matrices

#### Data Synthesis

To generate a sparse synthetic matrix \(A \in \mathbb{R}^{n \times n}\), with \(nnz\) non-zeros, we use a Bernoulli distribution to determine the location of the non-zero entries and a uniform distribution to generate the values. First, we completely fill the \(n\) principle diagonal entries. Next, we generate \(\frac{nnz - n}{2}\) index positions in the upper triangle for the non-zero entries by sampling from a Bernoulli distribution with \(p = \frac{nnz - n}{n^2 - n}\). We reflect each entry about the principle diagonal for symmetricity. Finally, we add \(n\) to each diagonal entry to ensure positive-definiteness. This also makes the generated matrix diagonally dominant.

#### Real Data

To demonstrate the prowess of Algorithm 3 on real-world data, we use some SPD matrices from the University of Florida’s sparse matrix collection \[7\]. The complete list of matrices from this collection used in our experiments along with a brief description is given in columns 1–4 of Table 3.

#### Evaluation

It is tricky to pick any single method as the “exact method” to compute the \(\logdet(A)\) for a sparse SPD matrix \(A\). One approach would be to use direct methods such as Cholesky decomposition of \(A\) \[6, 12\]. For direct methods, it is difficult to derive an analytical solution for the number of operations required for factorization as a function of the number of non-zeros \((nnz)\) as this is highly dependent on the structure of the matrix \[13\]. In case of distributed computing, one also needs to consider the volume of communication involved, which is often the bottleneck; we omit a detailed discussion for brevity. Alternatively, we can use iterative methods to compute the eigenvalues of \(A\) \[5\] and use the eigenvalues to compute \(\logdet(A)\). It is clear that the worst case performance of both the direct and iterative methods is \(O(n^3)\). However, iterative methods are typically used to compute a few eigenvalues and eigenvectors: therefore, we use direct Cholesky factorization based on matrix reordering to compute the exact value of \(\logdet(A)\). It is important to note that both the direct and iterative methods are notoriously hard to implement, which contracts starkly with the relatively simple implementation of Algorithm 3 which also parallelizes readily in the case of sparse matrices. However, we omit further discussion of parallelism in the sparse case for brevity and discuss results for sequential performance.

#### Results

The true power of Algorithm 3 lies in its ability to approximate \(\logdet(A)\) for sparse \(A\). The exact method — Cholesky — can introduce \(O(n^2)\) non-zeros during factorization due to fill-in; for many problems, there is insufficient memory to factorize a large but sparse matrix. In our first set of experiments, we wanted to show the effect of \(m\) on: (1) convergence of \(\logdet(A)\) and (2) cost of the solution. To this end, we generated sparse, diagonally dominant SPD matrices of size \(n = 10^6\) and varied the sparsity from 0.1% to 1% in increments of 0.25%. We did not attempt to compute the exact \(\logdet(A)\) for these synthetic matrices — our aim was to merely study the speedup with \(m\) for different sparsities while \(t\) and \(p\) were held constant at \(\log(4n)\) and 60 respectively. The results are shown in Figure 3.

Panel 3(a) depicts the convergence of \(\logdet(A)\) measured as relative error of the current estimate from the final estimate. As can be seen — for well conditioned matrices — convergence is quick. Panel 3(b)
Figure 3: Panels 3(a) and 3(b) depict the effect of the number of terms in the Taylor expansion, \( m \), (see Algorithm 3) on the convergence to the final solution and the time to completion of the approximation. The matrix size was fixed at \( n = 10^9 \) and sparsity was varied as 0.1\%, 0.25\%, 0.5\%, 0.75\%, and 1\%. Experiments were run sequentially (\( np = 1 \)) and we set \( p = 60, t = \log 4n \). For panel 3(a), the baseline is the final value of \( \log \det(A) \) at \( m = 25 \). For panel 3(b), the baseline is the time to completion of approximation of \( \log \det(A) \) at \( m = 1 \). Eigen was used as the backend for these experiments.

shows the relative cost of increasing \( m \); here the baseline is \( m = 1 \). As can be seen, the additional cost incurred by increasing \( m \) is linear when all other parameters are held constant.

The results of running Algorithm 3 on the UFL matrices are shown in Table 3. The numbers reported for the approximation are the mean and standard deviation of 10 iterations \( t = 5 \), and \( p = 5 \). The value of \( m \) was varied from 1 to 150 in increments of 5 to select the best (mean) accuracy. The matrices shown in Table 3 have nice structure, which lends itself to nice reorderings and therefore efficient Cholesky factorization. We see that even in such cases, the performance of Algorithm 3 is commendable due to the strong theoretical guarantees and the relatively lower algorithmic complexity; ldoor is the only exception as the approximation takes longer than the factorization. In the case of thermomech_T, we achieve good accuracy while getting a 22x speedup. The most expensive operations in Algorithm 3 are the (sparse-)matrix vector products, where the number of vectors is \( p \), a parameter which is independent of \( n \). Note that, for efficiency, we rewrite the \( p \) matrix-vector products as a single symmetric matrix-matrix product. As we show, it is possible to improve performance of Algorithm 3 by tuning \( p, t, \) and \( m \).

5 Conclusions

There are few approximation algorithms for the logarithm of the determinant of a symmetric positive definite matrix. Unfortunately, those algorithms either do not work for all SPD matrices, or do not admit a worst-case theoretical analysis, or both. In this work, we presented the first approximation algorithm for the logarithm of the determinant of a matrix that comes with strong theoretical worst-case analysis bounds and can be applied to any SPD matrix. Using state-of-the-art C++ numerical linear algebra software packages for both dense and sparse matrices, we demonstrated that the proposed approximation algorithm performs remarkably well in practice in serial and parallel environments.

Acknowledgements

We thank Ahmed El Alaoui for many useful suggestions and comments on an early draft of this paper.

---

6 We tried to run with a few larger SPD matrices from UFL collection (eg., Flan_1565, Hook_1498, Geo_1438, StocF-1465), but we were unable to finish Cholesky factorization in a reasonable amount of time when running sequentially.

7 We experimented with different \( p, t \) and settled on the smallest values that did not result in loss in accuracy.
References

[1] H. Avron and S. Toledo. Randomized Algorithms for Estimating the Trace of an Implicit Symmetric Positive Semi-definite Matrix. *J. ACM*, 58(2):8, 2011.

[2] R. P. Barry and R. K. Pace. Monte carlo estimates of the log determinant of large sparse matrices. *Linear Algebra and its applications*, 289(1):41–54, 1999.

[3] P. F. Curran. On a variation of the gershgorin circle theorem with applications to stability theory. 2009.

[4] A. d’Aspremont, O. Banerjee, and L. El Ghaoui. First-order methods for sparse covariance selection. *SIAM Journal on Matrix Analysis and Applications*, 30(1):56–66, 2008.

[5] E. R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics*, 17(1):87–94, 1975.

[6] T. A. Davis. *Direct methods for sparse linear systems*, volume 2. SIAM, 2006.

[7] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[8] W. Eberly, M. Giesbrecht, and G. Villard. On computing the determinant and smith form of an integer matrix. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 675–685. IEEE, 2000.

[9] J. Friedman, T. Hastie, and R. Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

[10] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.

[11] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[12] A. Gupta. Wsmp: Watson sparse matrix package (part-i: direct solution of symmetric sparse systems). *IBM TJ Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC*, 21886, 2000.

[13] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *Parallel and Distributed Systems, IEEE Transactions on*, 8(5):502–520, 1997.

[14] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, P. K. Ravikumar, and R. Poldrack. Big & quic: Sparse inverse covariance estimation for a million variables. In *Advances in Neural Information Processing Systems*, pages 3165–3173, 2013.

[15] T. Hunter, A. E. Alaoui, and A. Bayen. Computing the log-determinant of symmetric, diagonally dominant matrices in near-linear time. *arXiv preprint arXiv:1408.1693*, 2014.

[16] P. Kambadur and A. Lozano. A parallel, block greedy method for sparse inverse covariance estimation for ultra-high dimensions. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 351–359, 2013.

[17] W. Leithead, Y. Zhang, and D. Leith. Efficient gaussian process based on bfgs updating and logdet approximation. In *the 16th IFAC world congress*, 2005.

[18] J. P. LeSage and R. K. Pace. Spatial dependence in data mining. In *Data Mining for Scientific and Engineering Applications*, pages 439–460. Springer, 2001.

[19] R. Martin. Approximations to the determinant term in gaussian maximum likelihood estimation of some spatial models. *Communications in Statistics-Theory and Methods*, 22(1):189–205, 1992.

[20] R. K. Pace and R. Barry. Quick computation of spatial autoregressive estimators. *Geographical analysis*, 29(3):232–247, 1997.
[21] R. K. Pace, R. Barry, O. W. Gilley, and C. Sirmans. A method for spatial–temporal forecasting with an application to real estate prices. *International Journal of Forecasting*, 16(2):229–246, 2000.

[22] R. K. Pace and J. P. LeSage. Chebyshev approximation of log-determinants of spatial weight matrices. *Computational Statistics & Data Analysis*, 45(2):179–196, 2004.

[23] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[24] A. Reusken. Approximation of the determinant of large sparse symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 23(3):799–818, 2002.

[25] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[26] D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90. ACM, 2004.

[27] Y. Zhang, W. Leithead, D. Leith, and L. Walshe. Log-det approximation based on uniformly distributed seeds and its application to gaussian process regression. *Journal of Computational and Applied Mathematics*, 220(1):198–214, 2008.

[28] Y. Zhang and W. E. Leithead. Approximate implementation of the logarithm of the matrix determinant in gaussian process regression. *Journal of Statistical Computation and Simulation*, 77(4):329–348, 2007.