Accelerated Newton Iteration for Roots of Black Box Polynomials

Anand Louis
Princeton University
Princeton, USA
alouis@princeton.edu

Santosh S. Vempala
Georgia Tech
Atlanta, USA
vempala@gatech.edu

***Abstract*—We study the problem of computing the largest root of a real rooted polynomial p(x) to within error 'z' given only black box access to it, i.e., for any x, the algorithm can query an oracle for the value of p(x), but the algorithm is not allowed access to the coefficients of p(x). A folklore result for this problem is that the largest root of a polynomial can be computed in O(n log (1/z)) polynomial queries using the Newton iteration. We give a simple algorithm that queries the oracle at only O(log n log(1/z)) points, where n is the degree of the polynomial. Our algorithm is based on a novel approach for accelerating the Newton method by using higher derivatives.**

***Keywords*-polynomial roots, Newton's method.**

I. Introduction

Computing the roots of a polynomial is a fundamental algorithmic problem. According to the folklore Abel-Ruffini theorem, polynomials of degree five or higher do not have any algebraic solution in general, and the roots of polynomials can be irrational. Therefore, the roots of a polynomial can be computed only to some desired precision. The classical Newton's method (also known as the Newton-Raphson method) is an iterative method to compute the roots of a real rooted polynomial. Starting with an initial upper bound $x^0 \in \mathbb{R}$ on the largest root of a polynomial $f(x)$ of degree n, the Newton's method iteratively computes better estimates to the largest root as follows

$$x^{t+1} := x^t - \frac{f(x^t)}{f'(x^t)} .$$

A folklore result is that after $\mathcal{O}\left(n \log(x^0/\varepsilon)\right)$ iterations, x^t will be ε-close to the largest root of the polynomial.

We study the problem of computing the largest root of a real rooted polynomial $p(x)$ given only blackbox access to it, i.e., for any $x \in \mathbb{R}$, the algorithm can query an oracle for the value of $p(x)$, but the algorithm is not allowed access to the coefficients of $p(x)$. This model is useful when the polynomial is represented implicitly, and each evaluation of the polynomial is computationally expensive. An important example is the characteristic polynomial, say $f(x)$, of a matrix, say A; each evaluation of $f(x)$ amounts to computing the determinant of the matrix $(A - xI)$. More generally, equations involving determinants of polynomial matrices fall into this category. A slightly modified Newton's method can be used to compute the largest root of a polynomial using $\mathcal{O}\left(n \log(x^0/\varepsilon)\right)$ black box queries; we review this in Section II.

Computational Model: The two most common ways of measuring the time complexity of an algorithm are its *arithmetic* complexity, and its *boolean* or *bit* complexity. Arithmetic complexity counts the number of basic arithmetic operations (i.e. addition, subtraction, multiplication, division) required to execute an algorithm, whereas boolean/bit complexity counts the number of bit operations required to execute the algorithm. For most algorithms for combinatorial optimization problems, these two notions of time complexity are roughly the same with arithmetic operations being done on $\mathcal{O}(\log n)$-bit numbers. However, for many numerical algorithms, they can differ vastly. For example, Gaussian elimination is usually said to take $\mathcal{O}\left(n^3\right)$ time, but this usually refers to the number of arithmatic operations, and if done naively, the intermediate bit complexity can be exponential in n [Bla66], [Fru77]. However, using a more careful variant of Gaussian elimination due to Edmonds [Edm67], the bit complexity is known to be $\tilde{\mathcal{O}}\left(n^4\right)$ (see also [Bar68], [Dix82], [Sch98]). In this paper, we will be bounding the bit complexity of our algorithms.

Polynomial Time Algorithms: An algorithm is said to have a polynomial running time if its running time is polynomial in the input description, i.e., the number of bits to describe the $p(x)$ (if $p(x)$ is given explicitly), and the parameter ε, the latter being $\log(1/\varepsilon)$ bits. Standard iterative optimization methods can take time that grows polynomially with $1/\varepsilon$, which is undesirable since they are not polynomial time algorithms. Given only black box access to $p(x)$, we seek algorithms that query the black box in $\mathcal{O}(\mathsf{poly}(n, \log(1/\varepsilon)))$ points in $\mathbb{R}$, where each queried point is represented using $\mathcal{O}(\mathsf{poly}(n, \log(1/\varepsilon)))$ bits.

A. Our results

In this paper we study an alternative approach, inspired by the classical Newton iteration for finding roots of polynomials. Applying the Newton iteration (see Section II), we see that the iterates converge within $\tilde{\mathcal{O}}(n)$ iterations. Can we do better than this? Our main idea is to accelerate the Newton iteration using higher derivatives of the polynomial. The standard generalization to Householder methods via higher derivatives [Hou70], [OR70] does not give any significant

0272-5428/16 $31.00 © 2016 IEEE
DOI 10.1109/FOCS.2016.83

benefit. In Section III, we give an new iteration based on higher derivatives that converges faster, yielding our following main result. The complete algorithm is described in Figure 1.

Theorem I.1. *Given black-box access to a monic real rooted polynomial f of degree n, an upper bound γ on the absolute value of its roots, and an error parameter $\varepsilon \in (0, 1/2]$, there exists a deterministic algorithm that queries f at $\mathcal{O}(\log n \log(\gamma/\varepsilon))$ locations, each having precision $\mathcal{O}(\log n \log(n\gamma/\varepsilon))$ bits, and outputs an $x \in \mathbb{Q}$ satisfying $\lambda_1 \leqslant x \leqslant \lambda_1 + \varepsilon$, where λ_1 is the largest root of f.*

Computing Matrix Eigenvalues: For a matrix $A \in \mathbb{Z}^{n \times n}$, its characteristic polynomial is defined as $f := \det(xI - A)$. The eigenvalues of a matrix are also the roots of its characteristic polynomial. We note that the algorithms for computing the roots of an explicit polynomial are not directly useful for computing the eigenvalues of a matrix, as computing the characteristic polynomial of a matrix is a computationally non-trivial task. To the best of our knowledge, the current best algorithm to compute the characteristic polynomial of a matrix is due to Kaltofen and Villard [KV05] achieving bit complexity $\tilde{\mathcal{O}}(n^{2.697})$. Computing the determinant of an integer matrix has asymptotic bit complexity $\mathcal{O}(n^{\omega} \log^2 n \log(\|A\|_F))$ for any integer matrix A [Sto05]. Using this determinant algorithm as a black box, we get the following result for computing the eigenvalues of matrices.

Theorem I.2. *Given a symmetric matrix $A \in \mathbb{Q}^{n \times n}$, and a parameter $\varepsilon \in (0, 1/2]$, there exists a Las Vegas algorithm having bit complexity $\tilde{\mathcal{O}}(n^{\omega} \log^2(\|A\|_F/\varepsilon))$ that outputs an $x \in \mathbb{Q}$ satisfying $\lambda_1 \leqslant x \leqslant \lambda_1 + \varepsilon$, where λ_1 is the largest eigenvalue of A.*

A problem that arises naturally in the context of solving SDPs is that of determining whether a given matrix is PSD. Theorem I.2 yields an algorithm to check if a matrix is approximately PSD.

Remark I.3. Given a symmetric matrix $A \in \mathbb{Q}^{n \times n}$, and a parameter $\varepsilon \in (0, 1/2]$, there exists a Las Vegas algorithm having bit complexity $\tilde{\mathcal{O}}(n^{\omega} \log^2(\|A\|_F/\varepsilon))$ to check if $A \succeq -\varepsilon I$.

B. Related work

Accelerated Newton's methods: A folklore result about the Newton's iteration is that it has quadratic local convergence, i.e., if the initial estimate x^0 is "close" to a root a of the function f, then $(x^{t+1} - a)$ is roughly $\mathcal{O}\left((x^t - a)^2\right)$. Kou et. al.[KLW06] gave a modification to the Newton's method that has local cubic convergence. Gerlach [Ger94] (see also [FP96], [KKZN97], [KG00]) gave a way to modify the function f to obtain a function F_m (where $m \in \mathbb{Z}^+$ is a parameter) such that the Newton's method applied to F_m will yield local convergence of order m (the complexity of the computation of F_m increases with m). Ezquerro and Hernández [EH99], and Gutiérrez and Hernández [GH01] gave an acceleration of the Newton's method based on the convexity of the function. Many other modifications of the Newton's method have been explored in the literature, for e.g. see [OW08], [LR08], etc. None of these improve the asymptotic worst-case complexity of root-finding.

Explicit polynomials: A related problem is to compute the roots of an explicit polynomial of degree n, say $p(x)$, to within error ε. The Jenkins-Traub [JT70] algorithm for computing the roots of polynomials converges at better than quadratic rate. Pan [Pan96], [Pan97] gave an algorithm to compute all the roots of an explicit polynomial using $\tilde{\mathcal{O}}(n)$ arithmetic operations; the bit complexity of this algorithm is bounded by $\tilde{\mathcal{O}}(n^3)$. Bini and Pan [BP98] gave an algorithm to compute the roots of a real-rooted polynomial; the bit complexity of their algorithm is bounded by $\tilde{\mathcal{O}}(n^2)$. We note that this model is different from the blackbox model that we study; in the blackbox model of a polynomial p, the algorithm can query an oracle for the value of $p(x)$ for any $x \in \mathbb{R}$, but the algorithm is not allowed access to the coefficients of $p(x)$.

The Householder's methods [Hou70], [OR70] are a class of iterative root-finding algorithms. Starting with an initial guess x^0, the dth order iteration is defined as

$$x^{t+1} := x^t + d \frac{(1/f)^{(d-1)}(x^t)}{(1/f)^{(d)}(x^t)}$$

where $(1/f)^{(d)}$ is the dth derivative of $1/f$. If the initial guess x^0 is sufficiently close to a root, then this iteration has convergence of the order $d + 1$.

We refer the reader to [BP94], [McN07], [MP13] for a comprehensive discussion on algorithms for computing the roots of polynomials. In particular, Chapter 9 of [MP13] discusses methods involving higher derivatives.

Matrix Eigenvalues: The *Power Iteration* algorithm [MPG29] produces an ε-approximation to the top eigenvalue of a matrix in $(\log n)/\varepsilon$ iterations, thereby giving a running time bound of $\mathcal{O}(n^2(\log n)/\varepsilon)$. Other methods such as the *Lanczos* algorithm [Lan50], the *Arnoldi iteration* [Arn51], etc. also have a polynomial dependance on $1/\varepsilon$ in the running time [KW92], whereas methods such as the Jacobi Method [Rut71], the Householder method [Hou58], etc. have worst case running time $\mathcal{O}(n^3)$. Faster methods are known for special matrices such as diagonally dominant matrices (of which Laplacians are an important special case), but their dependence on $1/\varepsilon$ is again polynomial [Vis13].

An algorithm for computing the largest eigenvalue of a matrix can be obtained by checking *PSDness* of a sequence of matrices, namely, a binary search for x s.t. $xI - A$ is PSD. Checking whether a matrix is PSD can be done using Cholesky decomposition or Gaussian elimination in $\tilde{\mathcal{O}}(n^4)$ bit operations [Edm67]. Independently and concurrently, Ben-Or and Eldar [BOE15] gave an algorithm having bit complexity $\tilde{\mathcal{O}}\left(n^{\omega+\nu}\right)$ for any $\nu > 0$, to compute all the eigenvalues of a matrix.

Algorithms due to [PC99] (see also [NH13]) compute all the eigenvalues of a matrix in $\tilde{\mathcal{O}}\left(n^3\right)$ arithmetic operations. [DDH07] gave an algorithm to compute the eigenvalues in $\tilde{\mathcal{O}}\left(n^\omega\right)$ arithmetic operations. We refer the reader to [BP94], [PTVF07], [GL12] for a comprehensive discussion.

C. Preliminaries

Assumption I.4. Given a real rooted polynomial f of degree n and an upper bound a on the absolute value of its roots, the roots of the polynomial $f(4ax)/(4a)^n$ lie in $[-1/4, 1/4]$ and the roots of the polynomial $f(4ax - 1/4)/(4a)^n$ lie in $[0, 1/2]$. Therefore, we can assume without loss of generality that the given polynomial has all its roots in the range $[0, 1/2]$. Similarly, for a symmetric matrix A, $0 \preceq I/4 + A/(4\|A\|_F) \preceq I/2$. Note that in both these cases, we will need to scale the error parameter ε accordingly; since our algorithms will only have a logarithmic dependance on $1/\varepsilon$, this scaling will not be a problem.

Notation: For an $x \in \mathbb{R}$, we use $\mathcal{B}(x)$ to denote the bit complexity of x, i.e., the number of bits need to represent x. For a function g, we use $\tilde{\mathcal{O}}(g)$ to denote $\mathcal{O}\left(g \log^c g\right)$ for absolute constants c. For a function g, we use $g^{(k)}(x)$ to denote its k^{th} derivative w.r.t. x.

II. The basic Newton iteration

For finding the root of a polynomial function $f(\cdot) : \mathbb{R} \to \mathbb{R}$, the basic Newton iteration is the following: initialize $x^0 = 1$, and then

$$x^{t+1} := x^t - \frac{f(x^t)}{f'(x^t)} .$$

If $x^0 \geqslant \lambda_1$, then this iteration maintains $x^t \geqslant \lambda_1 \ \forall t$ and reduces $x^t - \lambda_1$ by a factor of at least $\left(1 - \frac{1}{n}\right)$ from the following observation.

Proposition II.1. *For any t, the Newton iterate x^t satisfies $x^t \geqslant \lambda_1$ and*

$$x^t - \lambda_1 \geqslant \frac{f(x^t)}{f'(x^t)} \geqslant \frac{x^t - \lambda_1}{n}$$

Proof: Since $f(x) = \Pi_{i\in[n]}(x - \lambda_i)$ we have

$$\frac{f(x)}{f'(x)} = \frac{1}{\sum_{i\in[n]} \frac{1}{x-\lambda_i}},\ x - \lambda_1 \geqslant \frac{1}{\sum_{i\in[n]} \frac{1}{x-\lambda_i}} \geqslant \frac{x - \lambda_1}{n} .$$

■

Along with the next elementary lemma, we get a bound of $\mathcal{O}\left(n \log\left(1/\varepsilon\right)\right)$ on the number of iterations needed for x^t to be ε close to λ_1.

Lemma II.2. *Let $x^0, x^1, \ldots$ be iterates satisfying $x^0 \geqslant \lambda_1$ and*

$$x^{t+1} \leqslant x^t - \frac{x^t - \lambda_1}{q(n)}.$$

Then for all $t \geqslant q(n)\ln(1/\varepsilon)$, we have $0 \leqslant x^t - \lambda_1 \leqslant \varepsilon$.

Proof: Suppose the condition is satisfied. Then,

$$\frac{x^{t+1} - \lambda_1}{x^t - \lambda_1} \leqslant 1 - \frac{1}{q(n)} .$$

Therefore,

$$\left(x^t - \lambda_1\right) \leqslant \left(1 - \frac{1}{q(n)}\right)^t \left(x^0 - \lambda_1\right) \leqslant \left(1 - \frac{1}{q(n)}\right)^t .$$

Hence, for all $t \geqslant q(n) \log\left(1/\varepsilon\right)$, we have $0 \leqslant x^t - \lambda_1 \leqslant \varepsilon$.

■

This leaves the task of computing $f'(x)$. We can simply use the approximation $(f(x+\delta) - f(x))/\delta$ for a suitably small δ. Thus the modified iteration which only needs evaluation of f (i.e., determinant computations when f is the characteristic polynomial of a matrix), is the following: initialize $x^0 = 1$, and then

$$x^{t+1} := x^t - \frac{\delta}{2} \frac{f(x^t)}{f(x^t + \delta) - f(x^t)}$$

with $\delta = \varepsilon^2$.

When $f(\cdot)$ is the characteristic polynomial of a matrix A, evaluation of $f(x)$ reduces to computing $\det(A - xI)$ which can be done using Theorem III.10. This gives an overall bit complexity of $\tilde{\mathcal{O}}\left(n^{\omega+1}\right)$ for computing the top eigenvalue.

III. Accelerating the Newton iteration

To see the main idea, consider the following family of functions. For any $k \in \mathbb{Z}$, define

$$g_k(x) := \sum_{i\in[n]} \frac{1}{\left(x - \lambda_i\right)^k} .$$

We define the k'th order iteration to be

$$\boxed{x^{t+1} := x^t - \frac{1}{n^{1/k}} \frac{g_{k-1}(x^t)}{g_k(x^t)} \qquad (1)}$$

Note that $g_1(x) = f'(x)/f(x)$ and for $k = 1$ we get the Newton iteration, as $g_0(x) = n$. Viewing the $g_k(x)$ as the k'th moment of the vector $\left(\frac{1}{x-\lambda_1}, \frac{1}{x-\lambda_2}, \ldots, \frac{1}{x-\lambda_n}\right)$, we can use the following basic norm inequality.

Lemma III.1. *For any vector $X \in \mathbb{R}^n$,*

$$\frac{1}{n^{1/k}} \|X\|_{k-1}^{k-1} \|X\|_\infty \leqslant \|X\|_k^k \leqslant \|X\|_{k-1}^{k-1} \|X\|_\infty .$$

Proof: Using Holder's Inequality, we get

$$\begin{aligned}\|X\|_{k-1}^{k-1} &= \sum_i X(i)^{k-1} \leqslant \left(\sum_i X(i)^k\right)^{\frac{k-1}{k}} n^{\frac{1}{k}} \\ &= \|X\|_k^{k-1} n^{\frac{1}{k}} .\end{aligned}$$

By the monotonicity of norms, we have $\|X\|_\infty \leqslant \|X\|_k$. Therefore,

$$\frac{1}{n^{1/k}} \|X\|_{k-1}^{k-1} \|X\|_\infty \leqslant \|X\|_k^k .$$

Next,

$$\begin{aligned}\|X\|_k^k &= \sum_i X(i)^k \leqslant \|X\|_\infty \sum_i X(i)^{k-1} \\ &= \|X\|_{k-1}^{k-1} \|X\|_\infty \,.\end{aligned}$$

■

Lemma III.1 implies that

$$x^t - \lambda_1 \geqslant \frac{1}{n^{1/k}} \frac{g_{k-1}(x^t)}{g_k(x^t)} \geqslant \frac{x^t - \lambda_1}{n^{1/k}}\,.$$

Therefore, the distance to λ_1 shrinks by a factor of $(1 - 1/n^{1/k})$ in each iteration, thereby needing only $\tilde{\mathcal{O}}\left(n^{1/k}\right)$ iterations in total.

This brings us to question of how to implement the iteration, i.e., how to compute $g_k(x)$? We first note that these can be rewritten in terms of higher derivatives of $g_1(x)$. Let $g_k^{(i)}(x)$ be the i'th derivative of $g_k(x)$.

Lemma III.2. *For any $k \in \mathbb{Z}$,*

$$g_k'(x) = -k\, g_{k+1}(x)\,.$$

$$g_k^{(i)}(x) = (-1)^i g_{k+i}(x) \prod_{j=0}^{i-1}(k+j)\,.$$

Proof:

$$\begin{aligned} g_k'(x) &= \frac{\mathrm{d}}{\mathsf{dx}}\left(\sum_{i=1}^n \frac{1}{(x-\lambda_i)^k}\right) \\ &= \sum_{i=1}^n -k \cdot \frac{1}{(x-\lambda_i)^{k+1}} = -k\, g_{k+1}(x)\,.\end{aligned}$$

The second part is similar. ■

Therefore the terms in iteration (1) are simply a ratio of higher derivatives of $g_1(x)$. In the complete algorithm below (Figure 1), which only needs evaluations of $f(\cdot)$, we approximate $g_l(x)$ using finite differences. The folklore finite difference method says that for any function $f : \mathbb{R} \to \mathbb{R}$, its k^{th} derivative can be estimated using

$$\frac{1}{\delta^k}\left(\sum_{i=0}^k (-1)^i \binom{k}{i} f\left(x + (k-i)\delta\right)\right)$$

for small enough δ. We prove this rigorously in our setting in Lemma III.7.

Discussion: While it is desirable for x^t to be very close to λ_1, for $\tilde{g}_k(x^t)$ to be a good approximation of $g_k(x^t)$, we need α and δ to be sufficiently smaller than $x^t - \lambda_1$. Equivalently, we need a way to detect when x^t gets "very close" to λ_1; step 2c does this for us (Lemma III.9). We also want to keep the bit complexity of x^t bounded; step 2d ensures this by retaining only a small number of the most significant bits of u^t.

The analysis of the algorithm can be summarised as follows.

Algorithm III.3 (Higher-order Newton Iteration)**.**
Input: A real rooted monic polynomial f of degree n such that all its roots lie in $[0, 1/2]$ (Assumption I.4), error parameter $\varepsilon \in (0, 1/2]$, iteration depth k.
Output: A real number λ satisfying $0 \leqslant \lambda - \lambda_1 \leqslant \varepsilon$, where λ_1 is the largest root of f.

1) Initialize $x^0 = 1$,
$$\varepsilon' := \frac{\varepsilon}{8n^{1/k}}, \delta := \frac{\varepsilon'}{16(2e)^k k}, \delta' := \delta^{k+1}, \alpha := \frac{\delta' \varepsilon'^2}{2n^2}$$
2) Repeat for $t = 1$ to $\lceil 16 n^{1/k} \log(1/\varepsilon) \rceil$ iterations:
   a) Compute $\tilde{g}_k(x^t)$ as follows.
   $$\tilde{g}_1(x) := \frac{1}{f(x)}\left(\frac{f(x+\alpha) - f(x)}{\alpha}\right)$$
   and $\tilde{g}_{k+1}(x) :=$
   $$\frac{(-1)^k}{k!} \frac{1}{\delta^k} \sum_{i=0}^k (-1)^i \binom{k}{i} \tilde{g}_1\left(x + (k-i)\delta\right)\,. \tag{2}$$
   b) Compute the update
   $$u^t := \frac{1}{4n^{1/k}} \frac{\tilde{g}_{k-1}(x^t)}{\tilde{g}_k(x^t)}\,.$$
   c) If $u^t \leqslant \varepsilon'$, then Stop and output x^t.
   d) If $u^t > \varepsilon'$, then round down u^t to an accuracy of ε'/n to get $\tilde{u}^t$ and set $x^{t+1} := x^t - \tilde{u}^t$.
3) Output x^t.

Figure 1. The Accelerated Newton Algorithm

Theorem III.4. *Given a monic real rooted polynomial $f : \mathbb{R} \to \mathbb{R}$ of degree n, having all its roots in $[0, 1/2]$, Algorithm III.3 outputs a λ satisfying*

$$0 \leqslant \lambda - \lambda_1 \leqslant \varepsilon$$

while evaluating f at $\mathcal{O}\left(kn^{1/k}\log(1/\varepsilon)\right)$ locations on $\mathbb{R}$. Moreover, given access to a blackbox subroutine to evaluate $f(x)$ which runs in time[1] *$\mathcal{T}_f\left(\mathcal{B}(x)\right)$, Algorithm III.3 has overall time complexity $\tilde{\mathcal{O}}\left(kn^{1/k}\log(1/\varepsilon)\mathcal{T}_f\left(k^2 + k\log(n/\varepsilon)\right)\right)$.*

A. Analysis

We start with a simple fact about the derivaties of polynomials.

Fact III.5. *For a degree n polynomial $f(x) = \Pi_{i\in[n]}(x - \lambda_i)$,*

[1]We assume that $\mathcal{T}_f(cn) = \mathcal{O}\left(\mathcal{T}_f(n)\right)$ for absolute constants c, and that $\mathcal{T}_f(n_1) \leqslant \mathcal{T}_f(n_2)$ if $n_1 \leqslant n_2$.

and for $k \in \mathbb{Z}_{\geqslant 0}$, $k \leqslant n$, we have

$$f^{(k)}(x) = k! f(x) \left(\sum_{\substack{S \subset [n] \\ |S|=k}} \Pi_{i \in S} \frac{1}{x - \lambda_i} \right) .$$

Proof: We prove this by induction on k. For $k = 1$, this is true. We assume that this statement holds for $k = l$ ($l < n$), and show that it holds for $k = l + 1$.

$$\begin{aligned}
f^{(l+1)}(x) &= \frac{\mathrm{d}}{\mathsf{dx}} f^{(l)}(x) \\
&= l! \frac{\mathrm{d}}{\mathsf{dx}} \left(\sum_{\substack{S \subset [n] \\ |S|=l}} f(x) \Pi_{i \in S} \frac{1}{x - \lambda_i} \right) \\
&= l! \sum_{\substack{S \subset [n] \\ |S|=l}} \sum_{j \in [n] \setminus S} \left(f(x) \Pi_{i \in S} \frac{1}{x - \lambda_i} \right) \frac{1}{x - \lambda_j} \\
&= l! \sum_{\substack{S \subset [n] \\ |S|=l+1}} (l+1) f(x) \Pi_{i \in S} \frac{1}{x - \lambda_i} \\
&= (l+1)! f(x) \sum_{\substack{S \subset [n] \\ |S|=l+1}} \Pi_{i \in S} \frac{1}{x - \lambda_i} .
\end{aligned}$$

■

Next, we analyze $\tilde{g}_1(\cdot)$.

Lemma III.6. *For $x \in [\lambda_1 + \varepsilon', 1]$, $\tilde{g}_1(x)$ defined in Algorithm III.3 satisfies $g_1(x) \leqslant \tilde{g}_1(x) \leqslant g_1(x) + \delta'$.*

Proof:

Using $\xi := 1/(x - \lambda_1)$ for brevity,

$$\begin{aligned}
&\frac{f(x+\alpha) - f(x)}{\alpha} \\
&= \frac{1}{\alpha} \left(\sum_{j=0}^{\infty} \frac{\alpha^j}{j!} f^{(j)}(x) - f(x) \right) \\
&\qquad \text{(Taylor series expansion of } f(\cdot) \text{)} \\
&= f'(x) + \frac{1}{\alpha} \left(\sum_{j=2}^{\infty} \alpha^j f(x) \sum_{\substack{S \subset [n] \\ |S|=j}} \Pi_{i \in S} \frac{1}{x - \lambda_i} \right) \\
&\qquad \text{(Using Fact III.5)} \\
&\leqslant f'(x) + \frac{f(x)}{\alpha} \left(\sum_{j=2}^{\infty} \alpha^j n^j \xi^j \right) \\
&\qquad \left(\text{Using} \frac{1}{x - \lambda_i} \leqslant \xi \right) \\
&\leqslant f'(x) + f(x) \frac{\alpha (n\xi)^2}{1 - \alpha n \xi} \\
&\leqslant f'(x) + \delta' f(x) \\
&\qquad \text{(Using definition of } \alpha) .
\end{aligned}$$

Since all the roots of $f(x)$ are in $[0, 1/2]$ and $x \in [\lambda_1 + \varepsilon', 1]$, we have $f(x) \leqslant 1$. Therefore,

$$\tilde{g}_1(x) = \frac{\frac{1}{\alpha} \left(f(x+\alpha) - f(x) \right)}{f(x)} \leqslant g_1(x) + \delta' .$$

Next, since $f(x), x - \lambda_i \geqslant 0$,

$$\begin{aligned}
&\frac{f(x+\alpha) - f(x)}{\alpha} \\
&= f'(x) + \frac{1}{\alpha} \left(\sum_{j=2}^{\infty} \alpha^j f(x) \sum_{\substack{S \subset [n] \\ |S|=j}} \Pi_{i \in S} \frac{1}{x - \lambda_i} \right) \\
&\geqslant f'(x) .
\end{aligned}$$

Therefore, $\tilde{g}_1(x) \geqslant g_1(x)$.

■

The crux of the analysis is to show that $\tilde{g}_l(x)$ is "close" to $g_l(x)$. This is summarised by the following lemma.

Lemma III.7 (Main Technical Lemma)**.** *For $x \in [\lambda_1 + \varepsilon', 1]$, $\tilde{g}_{k+1}(x)$ defined in Algorithm III.3 satisfies*

$$|\tilde{g}_{k+1}(x) - g_{k+1}(x)| \leqslant \frac{1}{4} g_{k+1}(x) .$$

Proof: We first bound the quantity $h_{k+1}(x)$ defined as follows.

$$
\begin{aligned}
&h_{k+1}(x)\\
&:= \frac{(-1)^k}{k!\delta^k}\left(\sum_{i=0}^{k}(-1)^i\binom{k}{i}g\left(x+(k-i)\delta\right)\right)\\
&= \frac{(-1)^k}{k!\delta^k}\left(\sum_{i=0}^{k}(-1)^i\binom{k}{i}\sum_{j=0}^{\infty}\frac{((k-i)\delta)^j}{j!}g^{(j)}(x)\right)\\
&\qquad\text{(Taylor series expansion of } g(\cdot)\text{)}\\
&= \frac{(-1)^k}{k!\delta^k}\left(\sum_{j=0}^{\infty}\frac{\delta^j g^{(j)}(x)}{j!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^j\right)\\
&\qquad\text{(Rearranging summations)}\\
&= \frac{(-1)^k}{k!\delta^k}\left(\sum_{j=0}^{\infty}\frac{\delta^j g^{(j)}(x)}{j!}(-1)^k\sum_{i=0}^{k}(-1)^i\binom{k}{i}i^j\right)\\
&\qquad\text{(Rearranging summation)}\\
&= g_{k+1}(x)\\
&+ \frac{(-1)^k}{k!\delta^k}\left(\sum_{j=k+1}^{\infty}\frac{\delta^j g^{(j)}(x)}{j!}(-1)^k\sum_{i=0}^{k}(-1)^i\binom{k}{i}i^j\right)\\
&\qquad\text{(Using Fact III.8)}\,.
\end{aligned}
$$

Using $\xi := 1/(x-\lambda_1)$ for brevity,

$$
\begin{aligned}
&|h_{k+1}(x)-g_{k+1}(x)|\\
&= \left|\frac{(-1)^k}{k!}\left(\sum_{j=k+1}^{\infty}\frac{\delta^{j-k}g^{(j)}(x)}{j!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}i^j\right)\right|\\
&\leqslant \frac{1}{k!}\left(\sum_{j=k+1}^{\infty}\delta^{j-k}g_{j+1}(x)\left(\sum_{i=0}^{k}\binom{k}{i}i^j\right)\right)\\
&\qquad\text{(Using Lemma III.2)}\\
&\leqslant \frac{1}{k!}\sum_{j=k+1}^{\infty}\delta^{j-k}\left(g_{k+1}(x)\xi^{j-k}\right)k^j 2^k\\
&\qquad\text{(Using } g_{j+1}(x)\leqslant \xi^{j-k}g_{k+1}(x)\text{)}\\
&= g_{k+1}(x)\frac{(2k)^k}{k!}\sum_{p=1}^{\infty}(k\delta\xi)^p\\
&\qquad\text{(Substituting } p \text{ for } j-k\text{)}\\
&= g_{k+1}(x)\frac{(2k)^k}{k!}\frac{k\delta\xi}{1-k\delta\xi}\,.
\end{aligned}
$$

Next, using Lemma III.6 and (2), we have

$$
\begin{aligned}
|\tilde{g}_{k+1}(x)-h_{k+1}(x)| &\leqslant \left|\frac{1}{k!\delta^k}\left(\sum_{i=0}^{k}(-1)^i\binom{k}{i}\delta'\right)\right|\\
&\leqslant \frac{\delta' 2^k}{k!\delta^k}\,.
\end{aligned}
$$

$$
\begin{aligned}
&|\tilde{g}_{k+1}(x)-g_{k+1}(x)|\\
&\leqslant |h_{k+1}(x)-g_{k+1}(x)|+|\tilde{g}_{k+1}(x)-h_{k+1}(x)|\\
&\leqslant g_{k+1}(x)\frac{(2k)^k}{k!}\frac{k\delta\xi}{1-k\delta\xi}+\frac{\delta' 2^k}{k!\delta^k}\\
&\leqslant g_{k+1}(x)4k\delta\xi\frac{(2k)^k}{k!}\\
&\qquad\text{(Using } g_{k+1}(x)\geqslant 1 \text{ and } \delta'=\delta^{k+1}\text{)}\\
&\leqslant g_{k+1}(x)(2e)^k 4k\delta\xi\\
&\text{(Using Stirling's approximation for } k!\text{)}\\
&\leqslant \frac{1}{4}g_{k+1}(x)\,.
\end{aligned}
$$

■

Fact III.8. *For* $j,k\in\mathbb{Z}_{\geqslant 0}$,

$$
\sum_{i=0}^{k}(-1)^i\binom{k}{i}i^j = \begin{cases}0 & \text{for } j<k\\ (-1)^k k! & \text{for } j=k\end{cases}\,.
$$

Proof: Define the polynomial $S_j(x)$ to be

$$
S_j(x) := \underbrace{x\frac{\mathrm{d}}{\mathsf{dx}}\ldots x\frac{\mathrm{d}}{\mathsf{dx}}}_{j\text{ times}}(1+x)^k\,. \tag{3}
$$

Then,

$$
S_j(x)=\sum_{i=0}^{k}\binom{k}{i}i^j x^i \quad\text{and}\quad S_j(-1)=\sum_{i=0}^{k}(-1)^i\binom{k}{i}i^j\,.
$$

Now, for $j<k$, $(1+x)$ will be a factor of the polynomial in (3). Therefore, $S_j(-1)=0$ for $j<k$. For $j=k$, out of the $k+1$ terms of $S_k(x)$ in (3), the only term that does not have a multiple of $(1+x)$ is $x^k k!$. Therefore, $S_k(-1)=(-1)^k k!$. ■

Next we show that the update step in Algorithm III.3 (step 2b) makes sufficient progress in each iteration.

Lemma III.9. *For* $x^t\in[\lambda_1+\varepsilon',1]$,

$$
\frac{x^t-\lambda_1}{8n^{1/k}}\leqslant u^t\leqslant\frac{x^t-\lambda_1}{2}\,.
$$

Proof: We first prove the upper bound.

$$
\begin{aligned}
u^t &= \frac{1}{4n^{1/k}}\frac{\tilde{g}_{k-1}(x^t)}{\tilde{g}_k(x^t)}\leqslant\frac{1}{4n^{1/k}}\frac{(1+1/4)g_{k-1}(x^t)}{(1-1/4)g_k(x^t)}\\
&\leqslant\frac{x^t-\lambda_1}{2}\,.
\end{aligned}
$$

Here, the first inequality uses Lemma III.7 and the second inequality uses Lemma III.1 with the vector $\left(\frac{1}{x^t-\lambda_1},\ldots,\frac{1}{x^t-\lambda_n}\right)$. Next,

$$
\begin{aligned}
u^t &= \frac{1}{4n^{1/k}}\frac{\tilde{g}_{k-1}(x^t)}{\tilde{g}_k(x^t)}\geqslant\frac{1}{4n^{1/k}}\frac{(1-1/4)g_{k-1}(x^t)}{(1+1/4)g_k(x^t)}\\
&\geqslant\frac{x^t-\lambda_1}{8n^{1/k}}\,.
\end{aligned}
$$

Here again, the first inequality uses Lemma III.7 and the second inequality uses Lemma III.1 with the vector $\left(\frac{1}{x^t-\lambda_1},\ldots,\frac{1}{x^t-\lambda_n}\right)$.

■

Putting it together: We now have all the ingredients to prove Theorem III.4. Theorem I.2 follows from Theorem III.4 by picking $k = \log n$.

Proof of Theorem III.4: We first analyze the output guarantees of Algorithm III.3, and then we bound its bit complexity.

Invariants and Output Guarantees: W.l.o.g., we may assume that $x^0 - \lambda_1 \geqslant \varepsilon'$. We will assume that $x^t - \lambda_1 \geqslant \varepsilon'$, and show that if the algorithm does not stop in this iteration, then $x^{t+1} - \lambda_1 \geqslant \varepsilon'$, thereby justifying our assumption.

Since we do step 2d only when $u^t > \varepsilon'$, we get using Lemma III.9 that

$$\tilde{u}^t \geqslant u^t - \frac{\varepsilon'}{n} > \frac{u^t}{2} \geqslant \frac{x^t-\lambda_1}{16n^{1/k}}\,.$$

Using Lemma II.2, we get that for some iteration $t \leqslant 16n^{1/k}\log(1/\varepsilon)$ of Algorithm III.3, we will have $x^t - \lambda_1 \leqslant \varepsilon$. Therefore, if the algorithm does not stop at step 2c, and terminates at step 3, the λ output by the algorithm will satisfy $0 \leqslant \lambda - \lambda_1 \leqslant \varepsilon$. If the algorithm does stop at step 2c, i.e., $u^t \leqslant \varepsilon'$, then from Lemma III.9 we get

$$x^t - \lambda_1 \leqslant 8n^{1/k}\, u^t \leqslant 8n^{1/k}\,\varepsilon' \leqslant \varepsilon\,.$$

Therefore, in both these cases, the algorithm outputs a λ satisfying $0 \leqslant \lambda - \lambda_1 \leqslant \varepsilon$.

Next, if the algorithm does not stop in step 2c, then we get from Lemma III.9 that

$$\varepsilon' < u^t \leqslant \frac{x^t-\lambda_1}{2} \tag{4}$$

and since $\tilde{u}^t \leqslant u^t$,

$$\begin{aligned} x^{t+1} - \lambda_1 &= \left(x^t - \tilde{u}^t\right) - \lambda_1 \geqslant \left(x^t - u^t\right) - \lambda_1 \\ &\qquad \text{(from step 2b of Algorithm III.3)} \\ &\geqslant \frac{x^t-\lambda_1}{2} \\ &\qquad \text{(from Lemma III.9)} \\ &> \varepsilon' \qquad \text{(from (4))}\,. \end{aligned}$$

Therefore, if we do not stop in interation t, then we ensure that $x^{t+1} - \lambda_1 \geqslant \varepsilon'$.

Bit Complexity: We now bound the number of bit operations performed by the algorithm. We will show by induction on t that the bit complexity of each x^t is

$$\mathcal{B}\left(x^t\right) \leqslant \log(n/\varepsilon')\,. \tag{5}$$

We will assume that $\mathcal{B}(x^t) \leqslant \log(n/\varepsilon')$. We use this to bound the number of bit operations performed in each step of Algorithm III.3 and to show that $\mathcal{B}\left(x^{t+1}\right) \leqslant \log(n/\varepsilon')$.

Each computation of $\tilde{g}_1(\cdot)$ involves two computations of $f(\cdot)$ and one division by $f(\cdot)$. The bit complexity of the locations at which $f(\cdot)$ is computed can be upper bounded by

$$\mathcal{B}\left(x^t\right) + \mathcal{B}\left(k\delta\right) + \mathcal{B}\left(\alpha\right) = \mathcal{O}\left(\mathcal{B}\left(\alpha\right)\right)\,.$$

From our assumption that $\mathcal{T}_f\left(n_1\right) \leqslant \mathcal{T}_f\left(n_2\right),\ \forall n_1 \leqslant n_2$, and that $\mathcal{T}_f\left(cn\right) = \mathcal{O}\left(\mathcal{T}_f\left(n\right)\right)$, we get that the bit complexity of each of these $f(\cdot)$ computations can be bounded by $\mathcal{O}\left(\mathcal{T}_f\left(\mathcal{B}\left(\alpha\right)\right)\right)$. Since, division can be done in nearly linear time [SS71], the bit complexity of the computation of $\tilde{g}_1(\cdot)$ is $\tilde{\mathcal{O}}\left(\mathcal{T}_f\left(\mathcal{B}\left(\alpha\right)\right)\right)$.

The computation of the $\tilde{g}_k(\cdot)$ involves k computations of $\tilde{g}_1(\cdot)$ and one division by δ^k, and therefore can be done using $\tilde{\mathcal{O}}\left(k\mathcal{T}_f\left(\mathcal{B}\left(\alpha\right)\right)\right)$ bit operations. Next, the computation of u^t involves computing the ratio of $\tilde{g}_{k-1}(x^t)$ and $\tilde{g}_k(x^t)$, both of which have bit complexity $\tilde{\mathcal{O}}\left(k\mathcal{T}_f\left(\mathcal{B}\left(\alpha\right)\right)\right)$. Therefore, u^t can be computed in $\tilde{\mathcal{O}}\left(k\mathcal{T}_f\left(\mathcal{B}\left(\alpha\right)\right)\right)$ bit operations [SS71]. Finally, since $x^{t+1} = x^t - \tilde{u}^t$, we get that $\mathcal{B}\left(x^{t+1}\right) = \mathcal{B}\left(x^t - \tilde{u}^t\right) \leqslant \log(n/\varepsilon')$. For our choice of parameters

$$\begin{aligned} \mathcal{B}\left(\alpha\right) &= \log\left(\frac{2n^2(16k)^{k+1}(2e)^{k^2+k}}{\varepsilon'^{k+3}}\right) \\ &= \mathcal{O}\left(k^2 + k\log(n/\varepsilon)\right)\,. \end{aligned}$$

Finally, since the number of iterations in the algorithm is at most $16n^{1/k}\log\left(1/\varepsilon\right)$, the overall query complexity of the algorithm is $\mathcal{O}\left(n^{1/k}\log\left(1/\varepsilon\right)\cdot k\right)$, and the overall bit complexity (running time) is

$$\tilde{\mathcal{O}}\left(n^{1/k}\log\left(1/\varepsilon\right)\cdot k\mathcal{T}_f\left(k^2 + k\log(n/\varepsilon)\right)\right)\,.$$

■

B. Computing the top eigenvalue of a matrix

Our algorithm (Theorem I.2) uses an algorithm the compute the determinant of a matrix as a subroutine. Computing the determinant of a matrix has many applications in theoretical computer science and is a well studied problem. We refer the reader to [KV04] for a survey. The algorithm for computing the determinant of a matrix with the current fastest asymptotic running time is due to Storjohann [Sto05].

Theorem III.10 ([Sto05])**.** *Let $A \in \mathbb{Z}^{n\times n}$. There exists a Las Vegas algorithm that computes $\det(A)$ using an expected number of $\mathcal{O}\left(n^\omega \log^2 n \log\|A\|_F\right)$ bit operations.*

Proof of Theorem I.2: Using Theorem III.10, each computation of $f(x) = \det(xI - A)$ can be done in time

$$\begin{aligned} &\mathcal{O}\left(n^\omega \log^2 n \log\left(\|A\|_F/\alpha\right)\right) \\ &\qquad = \mathcal{O}\left(n^\omega \log^2 n\left(k^2\log\left(\|A\|_F/\varepsilon\right)\right)\right) \end{aligned}$$

Using Theorem III.4 with $k = \lceil\log n\rceil$, the overall bit complexity (running time) of Algorithm III.3 is

$$\mathcal{O}\left(n^\omega \log^5 n \log^2\left(\|A\|_F/\varepsilon\right)\right)\,.$$

■

Acknowledgement: Anand Louis was supported in part by the Simons Collaboration on Algorithms and Geometry. Santosh S. Vempala was supported in part by NSF awards CCF-1217793 and EAGER-1555447. The authors are grateful to Ryan O' Donnell for helpful discussions, to Yin Tat Lee, Prasad Raghavendra, Aaron Schild and Aaron Sidford for pointing us to the finite difference method for approximating higher derivatives efficiently, and to Victor Pan and Nisheeth Vishnoi for helpful comments on a previous version of this paper.

References

[Arn51] Walter Edwin Arnoldi, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quarterly of Applied Mathematics **9** (1951), no. 1, 17–29.

[Bar68] Erwin H Bareiss, *Sylvesters identity and multistep integer-preserving gaussian elimination*, Mathematics of computation **22** (1968), no. 103, 565–578.

[Bla66] WA Blankinship, *Matrix triangulation with integer arithmetic*, Communications of the ACM **9** (1966), no. 7, 513.

[BOE15] Michael Ben-Or and Lior Eldar, *The quasi-random perspective on matrix spectral analysis with applications*, arXiv preprint arXiv:1505.08126 (2015).

[BP94] Dario Bini and Victor Y. Pan, *Polynomial and matrix computations (vol. 1): Fundamental algorithms*, Birkhauser Verlag, 1994.

[BP98] ______, *Computing matrix eigenvalues and polynomial zeros where the output is real*, SIAM J. Comput. **27** (1998), 1099–1115.

[DDH07] James Demmel, Ioana Dumitriu, and Olga Holtz, *Fast linear algebra is stable*, Numerische Mathematik **108** (2007), no. 1, 59–91.

[Dix82] John D Dixon, *Exact solution of linear equations usingp-adic expansions*, Numerische Mathematik **40** (1982), no. 1, 137–141.

[Edm67] Jack Edmonds, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Standards, Sect. B **71** (1967), no. 4, 241–245.

[EH99] J. A. Ezquerro and M. A. Hernández, *On a convex acceleration of newton's method*, J. Optim. Theory Appl. **100** (1999), no. 2, 311–326.

[FP96] William F. Ford and James A. Pennline, *Accelerated convergence in newton's method*, SIAM Rev. **38** (1996), no. 4, 658–659.

[Fru77] Michael A Frumkin, *Polynomial time algorithms in the theory of linear diophantine equations*, Fundamentals of Computation Theory, Springer, 1977, pp. 386–392.

[Ger94] Jürgen Gerlach, *Accelerated convergence in newton's method*, SIAM Rev. **36** (1994), no. 2, 272–276.

[GH01] J. M. Gutiérrez and M. A. Hernández, *An acceleration of newton's method: Super-halley method*, Appl. Math. Comput. **117** (2001), no. 2-3, 223–239.

[GL12] Gene H. Golub and Charles F. Van Loan, *Matrix computations (4th ed.)*, Johns Hopkins University Press, 2012.

[Hou58] Alston S. Householder, *Unitary triangularization of a nonsymmetric matrix*, J. ACM **5** (1958), no. 4, 339–342.

[Hou70] Alston Scott Householder, *The numerical treatment of a single nonlinear equation*, McGraw-Hill New York, 1970.

[JT70] M. A. Jenkins and J. F. Traub, *A three-stage variable-shift iteration for polynomial zeros and its relation to generalized rayleigh iteration*, Numer. Math. **14** (1970), 252–263.

[KG00] Bahman Kalantari and Jürgen Gerlach, *Newton's method and generation of a determinantal family of iteration functions*, J. Comput. Appl. Math. **116** (2000), no. 1, 195–200.

[KKZN97] Bahman Kalantari, Iraj Kalantari, and Rahim Zaare-Nahandi, *A basic family of iteration functions for polynomial root finding and its characterizations*, J. Comput. Appl. Math. **80** (1997), no. 2, 209–226.

[KLW06] Jisheng Kou, Yitian Li, and Xiuhua Wang, *A modification of newton method with third-order convergence*, Appl. Math. Comput. **181** (2006), no. 2, 1106–1111.

[KV04] Erich Kaltofen and Gilles Villard, *Computing the sign or the value of the determinant of an integer matrix, a complexity survey*, Journal of Computational and Applied Mathematics **162** (2004), no. 1, 133–146.

[KV05] ______, *On the complexity of computing determinants*, Computational complexity **13** (2005), no. 3-4, 91–130.

[KW92] Jacek Kuczynski and Henryk Wozniakowski, *Estimating the largest eigenvalue by the power and lanczos algorithms with a random start*, SIAM journal on matrix analysis and applications **13** (1992), no. 4, 1094–1122.

[Lan50] Cornelius Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, United States Governm. Press Office, 1950.

[LR08] Tibor Lukić and Nebojša M Ralević, *Geometric mean newtons method for simple and multiple roots*, Applied Mathematics Letters **21** (2008), no. 1, 30–36.

[McN07] John M. McNamee, *Numerical methods for roots of polynomials*, Elsevier, 2007.

[MP13] John M. McNamee and Victor Pan, *Numerical methods for roots of polynomials, part II*, Elsevier, 2013.

[MPG29] RV Mises and Hilda Pollaczek-Geiringer, *Praktische verfahren der gleichungsauflösung.*, ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik **9** (1929), no. 1, 58–77.

[NH13] Yuji Nakatsukasa and Nicholas J Higham, *Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the svd*, SIAM Journal on Scientific Computing **35** (2013), no. 3, A1325–A1349.

[OR70] James M Ortega and Werner C Rheinboldt, *Iterative solution of nonlinear equations in several variables*, vol. 30, Siam, 1970.

[OW08] Christina Oberlin and Stephen J. Wright, *An accelerated newton method for equations with semismooth jacobians and nonlinear complementarity problems*, Math. Program. **117** (2008), no. 1, 355–386.

[Pan96] Victor Y Pan, *Optimal and nearly optimal algorithms for approximating polynomial zeros*, Computers & Mathematics with Applications **31** (1996), no. 12, 97–138.

[Pan97] ______, *Solving a polynomial equation: some history and recent progress*, SIAM review **39** (1997), no. 2, 187–220.

[PC99] Victor Y Pan and Zhao Q Chen, *The complexity of the matrix eigenproblem*, Proceedings of the thirty-first annual ACM symposium on Theory of computing, ACM, 1999, pp. 507–516.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*, 3 ed., Cambridge University Press, 2007.

[Rut71] Heinz Rutishauser, *The jacobi method for real symmetric matrices*, Linear algebra, Springer, 1971, pp. 202–211.

[Sch98] Alexander Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, 1998.

[SS71] A. Schönhage and V. Strassen, *Schnelle multiplikation grosser zahlen*, Computing **7** (1971), no. 3-4, 281–292.

[Sto05] Arne Storjohann, *The shifted number system for fast linear algebra on integer matrices*, Journal of Complexity **21** (2005), no. 4, 609–650.

[Vis13] Nisheeth K. Vishnoi, $Lx = b$, Foundations and Trends in Theoretical Computer Science **8** (2013), 1–141.