석 사 학 위 논 문
Master's Thesis

실수에서의 지수 크기 선형대수학 문제의 계산 복잡도

Computational Complexity of Linear Algebra Problems with Exponential Size and Real Entries

2020

코스아라 이반 아드리안 (Koswara, Ivan Adrian)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

실수에서의 지수 크기 선형대수학 문제의 계산 복잡도

2020

코스아라 이반 아드리안

한 국 과 학 기 술 원

전산학부

실수에서의 지수 크기 선형대수학 문제의 계산 복잡도

코스아라 이반 아드리안

2020년 12월 14일

Computational Complexity of Linear Algebra Problems with Exponential Size and Real Entries

Ivan Adrian Koswara

Advisor: Martin Ziegler

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
December 14, 2020

Approved by

Martin Ziegler
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS 코스아라 이반 아드리안. 실수에서의 지수 크기 선형대수학 문제의 계산 복잡도. 전산학부 . 2020년. 30+i 쪽. 지도교수: 마틴 지글러. (영문 논문)
Ivan Adrian Koswara. Computational Complexity of Linear Algebra Problems with Exponential Size and Real Entries. School of Computing . 2020. 30+i pages. Advisor: Martin Ziegler. (Text in English)

초 록
지수 크기 문제는 복잡도 이론에서 잘 연구되지는 않았지만 몇몇 응용에서 나타난다. 예를 들어 편미분 방정식의 해를 근사하려면 지수 크기 행렬의 큰 거듭제곱을 계산해야 한다. 우리는 입력 벡터와 행렬이 어떤 모수에 대하여 지수 크기를 갖고 실수 성분을 가지는 선형대수 문제들을 형식적으로 정의한다. 우리는 해당 문제들의 계산복잡도 또한 조사한다. 우리는 지수 크기 내적 계산이 #P이고 지수 크기 행렬 거듭제곱 계산이 FPSPACE이며 이들이 모두 최적임을 보인다. 또한 편미분방정식에서 얻은 영감으로 우리는 일부 행렬들은 다항식으로 변환될 수 있고 다항식 거듭제곱 계산은 #P임을 보인다. 이는 해당 행렬들에 대해 행렬 거듭제곱 계산에 대한 결과를 강화한다. 우리는 또한 참신한 방법으로 일부의 다항식 거듭제곱은 FP임을 보이고 (지수 크기가 아닌) 내적과 행렬 거듭제곱이 선형 미만 공간에서 계산될 수 있음을 보인다.

핵 심 낱 말 계산복잡도, 지수 크기 문제, 정확한 실수 계산

Abstract
Exponential-size problems have not been investigated very well in complexity theory even though they appear in several applications. For example, approximating the solution of a partial differential equation requires one to raise an exponential-size matrix to a large power. We formally define linear algebra problems when the input vectors and matrices have size that is exponential in a parameter and have real number entries. We also investigate the computational complexity of these problems. We show that computing exponential-size inner product is in #P and computing exponential-size matrix powering is in FPSPACE, and these are both optimal. Furthermore, inspired from the partial differential equation motivation, we show some matrices can be converted into polynomials, and that computing polynomial powering is in #P, improving the matrix powering result for these matrices. We also show that some cases of polynomial powering are in FP with novel methods, and computing (not exponential-size) inner product and matrix powering can be done in sublinear space.

Keywords Computational complexity, exponential-size problem, exact real computation

Contents

Chapter 1. Introduction

There has been much research on the computational complexity of linear algebra problems. Perhaps one of the most famous problems is computing the matrix multiplication. The naive definition of matrix multiplication suggests an algorithm with time complexity of $O(n^3)$; however, in 1969, Volker Strassen proved this exponent is not optimal by providing an $O(n^{\log_2 7}) = O(n^{2.80736})$ algorithm [9]. Further research has subsequently reduced the exponent down to $O(n^{2.3728639})$ [5], and it is an open question whether an $O(n^2)$ algorithm exists. In the same 1969 paper, Strassen also proved that computing matrix determinant and matrix inverse are both asymptotically equivalent to matrix multiplication.

There is one subtle point, though; these algorithms usually treat an arithmetic operation as being exact and having a constant cost. This is fine in many practical situations, but problematic once we try to apply it to broader cases. The following describes two examples that are often encountered in computational linear algebra, along with other programming contexts involving non-integral numbers.

The numerical science community usually works with floating-point number data types, which have a fixed precision. A common problem arising from this is adding two numbers of very different magnitudes: if $x \gg y$, then $x + y$ is often approximated by x, as the number closest to $x + y$ that can be represented in the data type is in fact x itself. However, since $x + y$ and x are now equal, we have $(x + y) - x = 0$; in other words, taking the difference loses all information about y. In applications where it is crucial to distinguish 0 from a nonzero number, such as performing Gaussian elimination, this is clearly an issue.

Occasionally, instead of using floating-point numbers, some programmers work with exact rational numbers; since rational numbers are just pairs of integers, computers can manipulate them exactly. However, it can also be shown that, in general, each arithmetic operation can double the lengths of the numerators and denominators. As standard integral data types such as `int` and `long` have limited size, we will have to work with arbitrary-precision integers. Now the lengths of the numbers matter, as adding n-bit numbers in general takes time $O(n)$. With numbers that quickly grow very long, this per-operation cost quickly becomes prohibitive. This approach also only works with rational numbers; if we wish to use real numbers, this approach does not work.

Due to these shortcomings, people that are particularly concerned about rigor, such as mathematicians, began to develop a formal theory on computational complexity, but with real numbers. In his 1991 book, Ker-I Ko collected various results forming the foundation of this theory [3]. For example, a common rule of thumb in the numerical science community is to never compare two floating-point numbers for equality, because floating-point numbers are imprecise, and instead to use approximate equality [1]. It can in fact be proven that determining whether two real numbers are equal is an undecidable problem.

There have also been implementations of this formal theory, often called *exact real computation*, such as a C++ package called `iRRAM` [6]. The idea behind exact real computation is to track the uncertainty of each number computed. If the uncertainty gets too large, the entire computation is restarted, but with inputs that are more precise, i.e. with smaller uncertainty. The goal of exact real computation is to guarantee that all computations are correct, as opposed to using floating-point numbers that may lead to useless results.

This rigorous theory of computability and complexity of real numbers is still in active research. There have been a number of results, such as computing Riemann integral of bounded and poly-time computable functions being in #P [3, p.182]. Still, the vast amount of mathematical problems means there are always new avenues to pursue.

This thesis contributes to solving some such problems: the computational complexity of linear algebra problems, in particular matrix multiplication, matrix powering, and polynomial powering, where the input vector/matrix is exponential-size and the entries are real numbers. By exponential-size, we mean the complexity will be parametrized by $\log k$, where k is the dimension of the input vectors/matrices. These problems are motivated by their applications to approximating partial differential equations, further expanded in Section 4.

The paper is organized into several parts. Section 2 defines the formal definitions of real computation and real complexity classes. Section 3 states the main results of the paper, about the complexity of such problems; many of the proofs are postponed to the appendix. Section 4 provides a number of applications of these results to other problems, including the aforementioned motivation from approximating partial differential equations. Section 5 gives a conclusion with a summary of our results.

In this entire paper, log stands for base-2 logarithm while ln stands for natural logarithm.

Chapter 2. Definitions

2.1 Real computation

While integers and rational numbers can be represented exactly, in general, we cannot do the same to real numbers; we need to give infinitely many digits following the decimal point. Therefore real computation theory has been developed to formalize what it means for computers to work with real numbers. In this paper, we use the following definition of computable real numbers among several equivalent ones.

Definition 2.1.1. *A real number x is* ***computable*** *if there exist a computable integer function p and an algorithm $\mathcal{A}$, satisfying the following: when $\mathcal{A}$ takes input an integer n, it produces output an integer a_n satisfying*

$$\left| x - \frac{a_n}{2^{p(n)}} \right| \leq 2^{-n}.$$

We say $\mathcal{A}$ ***approximates*** *x. When n is given, we say $\mathcal{A}$ approximates x to (absolute) error 2^{-n}.*

We also say a real number y is an approximation of x to (absolute) error 2^{-n} if $|y - x| \leq 2^{-n}$. Therefore, $\mathcal{A}$ approximates x if it can output an approximation of x to arbitrarily small error.

Intuitively, x is computable if we can approximate the real number x arbitrarily well by dyadic fractions: given any n, we can give an approximation with (absolute) error 2^{-n}. This definition closely matches the definition by Ko based on Cauchy functions [3, Def2.1a p.42]. In Ko's definition, the function p has to be the identity, i.e. the denominator is 2^n instead of $2^{p(n)}$. However, from the view of computation theory, our definition is equivalent; once we obtain a_n, we simply "round" it to the appropriate number of bits.

Similarly, we can define computable real functions. We provide the definition for a univariate function, although it is easy to extend it to multivariate functions; we provide an all-encompassing general definition later on.

Definition 2.1.2. *A function $F : \mathbb{R} \to \mathbb{R}$ is* ***computable*** *if there exist computable integer functions q, p and an algorithm $\mathcal{A}$, satisfying the following: when $\mathcal{A}$ takes input integers n, b_n satisfying*

$$\left| x - \frac{b_n}{2^{q(n)}} \right| \leq 2^{-q(n)}$$

it produces output an integer a_n satisfying

$$\left| F(x) - \frac{a_n}{2^{p(n)}} \right| \leq 2^{-n}.$$

Note that $\mathcal{A}$ is allowed to behave arbitrarily for inputs not satisfying the hypothesis.

Intuitively, F is computable if we can approximate $F(x)$ arbitrarily well, given a sufficiently precise approximation of x. Note that we require the approximation to x to be given in advance, with an error bound of $2^{-q(n)}$ that may only depend on the accuracy n. This is in contrast to Ko [3, Def2.11 p.51], which provides approximations of x as an oracle that the algorithm can query; theoretically such algorithm may query an approximation of x with different error bounds depending on the value of x.

However, even this definition is lacking. We wish to be able to include classical discrete inputs, i.e. integer arguments. More importantly, while the definition above can be generalized to any arbitrary arity, F still has a fixed arity. Since in linear algebra we are interested in vectors and matrices as both inputs and outputs, which may have arbitrary dimensions, we wish to modify this definition.

The following definition is the most general form we consider. Instead of considering a single function F, we consider a sequence of functions $\{F_k\}$, each one with fixed arity that may depend on k. We include not only real arguments but also integer arguments to the input. Finally, instead of asking for an entire vector as the output, we can instead define F_k to take an additional integer argument as an index to indicate which element of the output we seek. This lets us to keep the codomain $\mathbb{R}$.

Definition 2.1.3. *A sequence of functions $\{F_k\}$ where $F_k : \mathbb{Z}^{d_k} \times \mathbb{R}^{d_k} \to \mathbb{R}$ is **computable** if there exist computable integer functions q, p and an algorithm $\mathcal{A}$, satisfying the following: when $\mathcal{A}$ takes input integers $k, n, \vec{m}_{k,n}, \vec{b}_{k,n}$ satisfying*

$$\left| x_i - \frac{b_{k,n,i}}{2^{q(\lambda)}} \right| \leq 2^{-q(\lambda)} \quad \textit{for all } i = 1, \ldots, d_k,$$

it produces output an integer $a_{k,n}$ satisfying

$$\left| F_k(\vec{m}_{k,n}, \vec{x}) - \frac{a_{k,n,j}}{2^{p(\lambda)}} \right| \leq 2^{-n}$$

where $\lambda = k + n + \ell(\vec{m}_{k,n})$. Note that $\mathcal{A}$ is allowed to behave arbitrarily for inputs not satisfying the hypothesis. Here $\vec{m}, \vec{b}$ are understood to have d_k elements.

Note that the argument to p, q is $\lambda = k + n + \ell(\vec{m}_{k,n})$. This reflects that the values of k, n, and $\vec{m}$ may determine how much accuracy we demand from the real inputs.

2.2 Complexity classes

Before proceeding to complexity classes, we wish to note a definition from classical discrete computability and complexity theory, namely one of function problems. There are two common definitions of function problems in literature, one that has exactly one valid output to every input and one that does not. Here, we use the latter definition; the reason will be made clear after we define real numbers.

Definition 2.2.1. *Given a domain X and codomain Y, a **(partial) function problem** F is a subset of $X \times Y$; we say $F : X \to Y$ to denote that the function problem F has domain X and codomain Y. An algorithm $\mathcal{A}$ **computes** a function problem F if, given any $x \in X$ as input such that there exists some $y \in Y$ where $(x, y) \in F$, $\mathcal{A}$ outputs one such y. Note that $\mathcal{A}$ is allowed to behave arbitrarily for x not having any corresponding y, including not halting.*

It is important that we distinguish "function problems" from "functions"; functions still have exactly one output for each input. Informally speaking, in this paper, we treat functions as the idealized form in mathematics, taking exact real numbers as inputs and producing exact real numbers as outputs. Function problems are defined to consider the inaccuracies of real numbers.

For example, a real number x is a function problem; it maps the desired accuracy represented as n to an approximation of x having the desired error 2^{-n}. Note that there are many possible outputs; for example, if $n = p(n)$, then we can "round" x either downward or upward to the closest multiple of 2^{-n}, and both are valid approximations of x. As another example, the problem computed by an algorithm $\mathcal{A}$ computing a real function f is also a function problem; f itself maps x to $f(x)$, but $\mathcal{A}$ maps approximations of x to approximations of $f(x)$ (in which there can be several).

With this peculiarity sorted out, we now proceed to define complexity classes. We first recall definitions of classical complexity classes, so that we can define their real versions accordingly.

Definition 2.2.2. *For a computable function t, a function problem $F : \mathbb{Z} \to \mathbb{Z}$ is in* $\mathsf{FTIME}(t)$ *(respectively* $\mathsf{FSPACE}(t)$*) if there exist a computable function t and an algorithm $\mathcal{A}$ computing F, and on input x, $\mathcal{A}$ takes time (resp. uses space) $\leq O(t(\ell(x)))$ where $\ell(x)$ is the length of the input.*

If there exists polynomial (respectively exponential) t such that $\mathcal{A}$ takes time $\leq t$, we say F is in FP *(resp.* FEXP*). If there exists polynomial t such that $\mathcal{A}$ uses space $\leq t$, we say F is in* $\mathsf{FPSPACE}$*.*

We leave "length of the input" intentionally not defined, because sometimes we wish to encode the input in binary (i.e. $\ell(x) = \log(1 + |x|)$) and sometimes in unary (i.e. $\ell(x) = O(x)$).

These definitions translate directly into definitions of real numbers and functions computable with a certain resource bound. For instance, we have the following definition for the complexity of a real number.

Definition 2.2.3. *A real number x is* ***poly-time/exp-time/poly-space computable*** *if the corresponding function problem of approximating x as per Definition 2.1.1 is in* $\mathsf{FP}, \mathsf{FEXP}, \mathsf{FPSPACE}$ *respectively, with n in unary.*

Note that, in Definition 2.1.1, the output of $\mathcal{A}$ has length $p(n) + O(1)$. We also want $p(n) \geq n - 1$; otherwise sometimes there is no valid output. Therefore the running time of $\mathcal{A}$ is at least $n + O(1)$. This motivates why we write n in unary, i.e. parametrize the complexity function in n instead of $\log n$; if n was in binary, the running time would be exponential in $\log n$ and so "poly-time computable real number" would not make sense.

It might seem straightforward to define the complexity of a real function, too. However, we have a problem of determining the parameter of the complexity function: what is "$\ell(x)$" in Definition 2.2.2? Here, we use the desired accuracy n as defined in Definition 2.1.2 as the parameter. Similar to the previous definition, n here is in unary.

This raises another issue: the magnitude of the input is not included in the parameter. For example, it is unreasonable to ask that the algorithm runs in the same time for $x = 0$ and $x = 2^{1000}$; after all, the function $f(x) = x$ will take time at least proportional to the length of x, the time taken to copy the input to the output. We solve this problem by restricting the domain into a fixed, bounded interval. Therefore the integer part has constant size, and the fractional part is already parametrized by n, leading to the following definition.

Definition 2.2.4. *For a fixed real c, a function $F : [-c, c] \to \mathbb{R}$ is in* $\mathbb{R}\mathsf{P}$ *if there exist a polynomial t and an algorithm $\mathcal{A}$ computing F in the sense of Definition 2.1.2, and $\mathcal{A}$ takes time $t(n)$.*

The definitions for F being in $\mathbb{R}\mathsf{EXP}$*,* $\mathbb{R}\mathsf{PSPACE}$*,* $\mathbb{R}\mathsf{TIME}$*,* $\mathbb{R}\mathsf{SPACE}$ *are similar.*

It follows that p, q in Definition 2.1.2 should be polynomial. If p is more than a polynomial, then $\mathcal{A}$ cannot output the required digits in polynomial time; if q is more than a polynomial, there are input bits that are not read and so q could have been made smaller.

We can also likewise extend this to a sequence of functions, like in Definition 2.1.3. The main difference from the previous definition is that the complexity function takes into account the function index k and all the integer inputs $\ell(\vec{m}_{k,n})$. This time, there is no particular justification why we choose k instead of $\log k$; it is just more convenient.

Definition 2.2.5. *For a fixed real c, a sequence of functions $\{F_k\}$ where $F_k : \mathbb{Z}^{d_k} \times [-c, c]^{d_k} \to \mathbb{R}$ is in* $\mathbb{R}\mathsf{P}$ *if there exists a polynomial t and an algorithm $\mathcal{A}$ computing $\{F_k\}$ in the sense of Definition 2.1.3, and $\mathcal{A}$ takes time $t(\lambda)$.*

The definitions for $\{F_k\}$ being in $\mathbb{R}\mathsf{EXP}$*,* $\mathbb{R}\mathsf{PSPACE}$*,* $\mathbb{R}\mathsf{TIME}$*,* $\mathbb{R}\mathsf{SPACE}$ *are similar.*

Remark 2.2.6. *It is crucial that the input to $\mathcal{A}$ is presented in a way such that $\mathcal{A}$ can seek to the i-th bit in time $O(\log i)$. For example, $\mathcal{A}$ may be a random-access machine, which can fetch the contents of any address in constant time, so that the cost of retrieving the bit is just the cost of computing the address.*

Note that $\mathcal{A}$ cannot be a Turing machine, as a Turing machine takes time $O(i)$ to seek to the i-th bit. This is important because our input may have size exponential in λ, and in particular, exponential in k.

Besides these classical complexity classes, we will also discuss the complexity class $\#\mathsf{P}$, defined by Valiant [10]. In the classical definition, a $\#\mathsf{P}$ problem counts the number of accepting paths of a nondeterministic Turing machine. This has the disadvantage that $\#\mathsf{P}$ is not closed under subtraction (since we cannot count a negative number of accepting paths), and so GapP has been introduced as the closure of $\#\mathsf{P}$ under subtraction. In this paper, we will take a different approach: since negative real numbers exist naturally, we will treat as if $\#\mathsf{P}$ is already complete under subtraction in the first place, providing the following definition.

Definition 2.2.7. *A function problem $F : \mathbb{Z} \to \mathbb{Z}$ is in $\#\mathsf{P}$ if there exist a polynomial r and an algorithm $\mathcal{A}$ satisfying the following: on input (x, w) where w is a binary string having length $r(\ell(x))$, $\mathcal{A}$ produces output a_w in time polynomial in $\ell(x)$, and*

$$\sum_w a_w = F(x).$$

Here the summation is over all w of length $r(\ell(x))$.

In other words, $\mathcal{A}$ takes an additional input w, called the witness. Note that $\mathcal{A}$ produces possibly different outputs on different witnesses, even with the same input x. The function $F(x)$ is the sum of all the outputs from different witnesses.

Intuitively, this means we can break down the computation of $F(x)$ into many pieces, each indexed by a different witness, and sum together all the answers. The power of this definition comes from the fact that there can be exponentially many witnesses: since r is a polynomial, there are $2^{r(\ell(x))}$ witnesses, which is exponential in $\ell(x)$. Naively summing over all witnesses would take exponential time; however, if each piece can be computed in polynomial time, the problem F is in $\#\mathsf{P}$. Another way to interpret $\#\mathsf{P}$ is that we get a final summation "for free", even if the summation is exponentially long.

We can also take the real equivalent of $\#\mathsf{P}$; this simply combines the definitions of computable real sequences of real functions and $\#\mathsf{P}$.

Definition 2.2.8. *For a fixed real c, a sequence of functions $\{F_k\}$ where $F_k : \mathbb{Z}^{d_k} \times [-c, c]^{d_k} \to \mathbb{R}$ is in $\mathbb{R}\#\mathsf{P}$ if there exist polynomials q, p, r and an algorithm $\mathcal{A}$ satisfying the following: on input $(k, n, \vec{m}_{k,n}, \vec{b}_{k,n}, w)$ where w is a binary string having length $r(\lambda)$ and*

$$\left| x_i - \frac{b_{k,n,i}}{2^{q(\lambda)}} \right| \leq 2^{-q(\lambda)} \quad \textit{for all } i = 1, \ldots, d_k,$$

$\mathcal{A}$ produces output $a_{k,n,w}$ in time polynomial in λ, and

$$\left| F(\vec{x}) - \sum_w \frac{a_{k,n,w}}{2^{p(\lambda)}} \right| \leq 2^{-n}$$

where $\lambda = k + n + \ell(\vec{m}_{k,n})$. Here the summation is over all w of length $r(\ell(x))$.

2.3 Linear algebra problems

We have defined the computability and complexity of sequences of real functions as in Definition 2.1.3, 2.2.5, and 2.2.8. Note that although the functions are defined to only take a single real vector as input, in practice we can also take multiple vectors as well as matrices in a straightforward way.

We will finally define the functions of our interest: exponential-size inner product, exponential-size matrix powering, and polynomial powering. The informal definitions are that we want to compute the inner product $\vec{u} \cdot \vec{v}$, any entry of the matrix power M^e, and the coefficients of polynomial power $p(\vec{x})^e$. Formally, they are defined as follows. Note that these are *functions*, i.e. the F_k's in the definitions mentioned above; their associated function problems follow accordingly.

Definition 2.3.1. *The function `EXPSIZE-INNER-PRODUCT` with index k is defined as follows: given two real vectors $\vec{u}, \vec{v}$ of 2^k elements, where each element is in $[-1, 1]$, compute*

$$\vec{u} \cdot \vec{v} := \sum_{i=1}^{2^k} u_i v_i.$$

Definition 2.3.2. *The function `EXPSIZE-MATRIX-POWER` with index k is defined as follows: given a $2^k \times 2^k$ real matrix M satisfying*

$$\|M\| := \max_j \sum_i |M_{i,j}| \leq 1,$$

a positive integer E, and positive integers I, J, compute the (I, J)-th element of M^E; that is, $(M^E)_{I,J}$. Here E, I, J are represented in binary.

Definition 2.3.3. *The function `POLYNOMIAL-POWER` with index v, d is defined as follows: given the coefficients of a polynomial P on $x_1, \ldots, x_v$ and degree d, satisfying*

$$\|P\| := \sum_{i_1,\ldots,i_v} |P[x_1^{j_1} \ldots x_v^{j_v}]| \leq 1,$$

a positive integer E, and nonnegative integers $I_1, \ldots, I_v$, compute the coefficient of $x_1^{I_1} \ldots x_v^{I_v}$ in the polynomial $P(x_1, \ldots, x_v)^E$. Here $E, I_1, \ldots, I_v$ are represented in binary.

Chapter 3. Main theorems

The main theorems in this paper are the complexity results of exponential-size inner product, exponential-size matrix power, and polynomial power. We will show that computing inner product is in $\mathbb{R}\#\mathsf{P}$, computing matrix powering is in $\mathbb{R}\mathsf{PSPACE}$, and computing polynomial powering is in $\mathbb{R}\#\mathsf{P}$ with specific cases that are in $\mathbb{R}\mathsf{P}$. We will also mention how the polynomial powering results can be used for certain cases of matrix powering that arise from solving differential equations. Moreover, we also show that, in a sense, computing inner product is "$\#\mathsf{P}$-hard" and computing matrix powering is "PSPACE-hard".

3.1 Exponential-size inner product is $\mathbb{R}\#\mathsf{P}$-complete

The main results of this section are the following two theorems related to computing exponential-size inner product. The first simply states that inner product is in $\mathbb{R}\#\mathsf{P}$. The second says, given any function in $\#\mathsf{P}$, we can reduce it into an instance of exponential-size inner product. However, since the instance will be exponential-size, the reduction instead provides an algorithm that generates the input.

Theorem 3.1.1. *EXPSIZE-INNER-PRODUCT is in* $\mathbb{R}\#\mathsf{P}$.

Theorem 3.1.2. *For any function $F \in \#\mathsf{P}$, there exists a polynomial-time algorithm $\mathcal{R}_F$ with the following property:*

Given integer input x, algorithm $\mathcal{R}_F$ returns $(k, \mathcal{B})$ where k is the index to EXPSIZE-INNER-PRODUCT and $\mathcal{B}$ is an algorithm that approximates the input $\vec{u}, \vec{v}$ for EXPSIZE-INNER-PRODUCT in the following manner:

Given accuracy parameter n and index w, i where $w \in \{u, v\}$ and $i \in [1, 2^k]$, $\mathcal{B}$ outputs an approximation of w_i to error 2^{-n}. Moreover, the vectors $\vec{u}, \vec{v}$ approximated by $\mathcal{B}$ have the property $\vec{u} \cdot \vec{v} = F(x)$.

3.1.1 Proof of Theorem 3.1.1

The idea for the proof is very straightforward. Since

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^{2^k} u_i v_i,$$

we can construct an algorithm $\mathcal{A}$ as follows: given a witness $w \in [1, 2^k]$, $\mathcal{A}$ seeks the approximations of u_w, v_w and outputs an approximation of $u_w v_w$. Then the sum of the outputs over all witnesses is exactly $\vec{u} \cdot \vec{v}$. The problem is to determine whether we can do this with sufficient accuracy. This section will show that, yes, we can obtain sufficient accuracy without much trouble.

Lemma 3.1.3. ***(a)*** *If x, y are real numbers and $\hat{x}, \hat{y}$ are approximations of x, y to error δ_x, δ_y respectively, i.e.*

$$|x - \hat{x}| \le \delta_x \quad \text{and} \quad |y - \hat{y}| \le \delta_y,$$

then $\hat{x} + \hat{y}$ is an approximation of $x + y$ to error $\delta_x + \delta_y$.

(b) *If x, y are real numbers satisfying $|x| \le c_x, |y| \le c_y$ for some c_x, c_y, and $\hat{x}, \hat{y}$ are approximations of x, y to error δ_x, δ_y respectively and also satisfying $|\hat{x}| \le c_x, |\hat{y}| \le c_y$, then $\hat{x}\hat{y}$ is an approximation of xy to error $c_x\delta_y + c_y\delta_x$.*

The proof is simple and largely mechanical.

Proof of Lemma 3.1.3. **(a)** The error of $\hat{x}+\hat{y}$ is given by

$$\begin{aligned} |(x+y)-(\hat{x}+\hat{y})| &= |(x-\hat{x})+(y-\hat{y})| \\ &\le |x-\hat{x}|+|y-\hat{y}| && \text{by triangle inequality} \\ &\le \delta_x+\delta_y && \text{by assumption} \end{aligned}$$

(b) The error of $\hat{x}\hat{y}$ is given by

$$\begin{aligned} |xy-\hat{x}\hat{y}| &= |(xy-x\hat{y})+(x\hat{y}-\hat{x}\hat{y})| \\ &= |x(y-\hat{y})+\hat{y}(x-\hat{x})| \\ &\le |x|\cdot|y-\hat{y}|+|\hat{y}|\cdot|x-\hat{x}| && \text{by triangle inequality} \\ &\le c_x\delta_y+c_y\delta_x && \text{by assumption} \end{aligned}$$

□

Equipped with these results, we can now determine the required accuracy.

Let x be an input real number satisfying $|x|\le 1$. Suppose we guarantee that the approximation $\hat{x}$ of x has absolute error ≤ 1. Then it follows $|\hat{x}|\le|x-2^{-n}|\le 2$. Therefore, all input real numbers and their approximations have absolute value ≤ 2.

Fix the accuracy parameter n. Fix a real number δ_n to be determined later. For each input real number u_i, let $\hat{u}_i$ be an approximation of it with absolute error $\delta_n\le 1$; define $\hat{v}_i$ similarly. Then by Lemma 3.1.3b, $\hat{u}_i\hat{v}_i$ is an approximation of u_iv_i with absolute error $2\delta_n+2\delta_n=4\delta_n$, and

$$\sum_{i=1}^{2^k}\hat{u}_i\hat{v}_i$$

is an approximation of $\vec{u}\cdot\vec{v}$ with absolute error $2^k\cdot 4\delta_n=2^{k+2}\delta_n$. Since we want this approximation to have error $\le 2^{-n}$, we can take $\delta_n=2^{-k-n-2}$.

More formally, using Definition 2.2.5, we take $\lambda=k+n$ (as we do not have any integer input to `EXPSIZE-INNER-PRODUCT`). We also take $q(\lambda)=\lambda+2$, $p(\lambda)=2\lambda+4$, $r(\lambda)=\lambda$. Map the integers in $[1,2^k]$ to the binary strings of length $r(\lambda)$ such that the mapping is one-to-one. Suppose $\mathcal{A}$ receives input $(k,n,\vec{b}_{u,k,n},\vec{b}_{v,k,n},w)$ satisfying the following condition: $\vec{b}_{u,k,n}$ is a 2^k-element vector where, if we define

$$\hat{u}_i=\frac{b_{u,k,n,i}}{2^{k+n+2}},$$

we have

$$|u_i-\hat{u}_i|=\left|u_i-\frac{b_{u,k,n,i}}{2^{k+n+2}}\right|\le 2^{-(k+n+2)}\quad\text{for all } i=1,\dots,2^k.$$

Similar condition applies for $\vec{b}_{v,k,n}$. If w is not one of the strings mapped by $[1,2^k]$, $\mathcal{A}$ outputs 0. Otherwise, $\mathcal{A}$ outputs $b_{u,k,n,w}\cdot b_{v,k,n,w}$. According to the proof above, we know

$$\begin{aligned} &\left|\vec{u}\cdot\vec{v}-\sum_w\frac{\vec{b}_{u,k,n,w}\cdot\vec{b}_{v,k,n,w}}{2^{2(k+n)+4}}\right| \\ &=\left|\vec{u}\cdot\vec{v}-\sum_w\frac{\vec{b}_{u,k,n,w}}{2^{k+n+2}}\cdot\frac{\vec{b}_{v,k,n,w}}{2^{k+n+2}}\right| \\ &=\left|\vec{u}\cdot\vec{v}-\sum_w\hat{u}_w\hat{v}_w\right| && \text{by definition of } \hat{u}_w,\hat{v}_w \\ &\le 2^{-n} && \text{by the proof above} \end{aligned}$$

Clearly $\mathcal{A}$ takes polynomial time, as $b_{u,k,n,w}, b_{v,k,n,w}$ have length $k+n+3$ bits (1 bit for the integer part as the absolute value is $\leq 2^1$, plus $k+n+2$ bits for the fractional part), which is polynomial in $k+n$, so they can be multiplied in polynomial time. This shows that `EXPSIZE-INNER-PRODUCT` is in $\mathbb{R}$#P. $\square$

3.1.2 Proof of Theorem 3.1.2

We will in fact avoid the real part of this problem altogether: the vectors $\vec{u}, \vec{v}$ we construct will only have integer entries. This way, not only $\mathcal{B}$ can output an approximation of w_i to error 2^{-n} as desired, but $\mathcal{B}$ can output w_i exactly.

Since $F \in \#\mathsf{P}$, there exists a polynomial r and algorithm $\mathcal{A}$ satisfying Definition 2.2.2. Recall that the length of the witness w is $r(\ell(x))$. Let $\mathcal{A}$ take time $t(\ell(x))$, where t is a polynomial. Then the output a_w of $\mathcal{A}$ satisfies $|a_w| \leq 2^{t(\ell(x))}$.

Let $k = r(\ell(w)) + t(\ell(x))$. Let $v_i = 1$ for all $i \in [1, 2^k]$. Define u_i as follows. Let

$$i = 2^{t(\ell(x))} \cdot r + t + 1$$

where $r \in [0, 2^{r(\ell(x))} - 1], t \in [0, 2^{t(\ell(x))} - 1]$; we can treat r as a binary string of length $r(\ell(x))$. Then consider the output a_r of $\mathcal{A}$ with witness r. If $|a_r| > t$, then $u_i = 1$ if $a_r > 0$ and $u_i = -1$ otherwise. Otherwise, $u_i = 0$.

Note that $u_i, v_i \in [-1, 1]$, so this is an instance of `EXPSIZE-INNER-PRODUCT`. We claim $\vec{u} \cdot \vec{v} = F(x)$.

Fix r, and consider all i having the same r. The number of nonzero u_i's is exactly $|a_r|$ (these are the ones where $t = 0, 1, \ldots, |a_r| - 1$), and the values of all these u_i's are the same, 1 if $a_r > 0$ and -1 otherwise; equivalently, $u_i = \text{sgn}(a_r)$ for the nonzero u_i's. Therefore, the sum of all u_i's for a given r is exactly $|a_r| \cdot \text{sgn}(a_r) = a_r$. Since $v_i = 1$ for all i, it follows $\sum_i u_i v_i = a_r$ for the i's giving the same r. Therefore,

$$\sum_i u_i v_i = \sum_r a_r = F(x)$$

by definition of F, as it is the sum of all a_r's.

Finally, we need to show this can be computed in time polynomial in $\ell(x)$. But this is straightforward. If $\mathcal{B}$ is asked to compute v_i, it can immediately output 1. Otherwise, $\mathcal{B}$ computes the necessary $r(\ell(x)), t(\ell(x))$ as well as the decomposition $i = 2^{t(\ell(x))} \cdot r + t + 1$, then computes $\mathcal{A}$ on input x and witness r which takes time polynomial in x (specifically $t(\ell(x))$), and finally output the appropriate u_i. All these steps take time polynomial in $\ell(x)$. $\square$

3.2 Exponential-size matrix powering is $\mathbb{R}$PSPACE-complete

Similar to the previous section, the main results of this section are the following two theorems related to computing exponential-size matrix power. The first states that computing matrix power is in $\mathbb{R}$PSPACE. The second says we can reduce a known PSPACE-complete problem, `LBA-HALTING`, into an instance of exponential-size matrix powering. Again, the reduction instead provides an algorithm that generates the input.

Theorem 3.2.1. *`EXPSIZE-MATRIX-POWER` is in* $\mathbb{R}$PSPACE.

Theorem 3.2.2. *Let `LBA-HALTING` be the following decision problem: given a Turing machine $\mathcal{T}$ and a natural number t in unary, determine whether $\mathcal{T}$ halts without using more than t cells of the tape. There exists a polynomial-time algorithm $\mathcal{R}$ with the following property:*

Given a Turing machine $\mathcal{T}$ and a natural number t in unary, the algorithm $\mathcal{R}$ returns $(k, E, I, J, \mathcal{B})$ where k is the index to `EXPSIZE-MATRIX-POWER`, *E is the exponent of the matrix, I, J are the indices of the desired entry, and $\mathcal{B}$ is an algorithm that approximates the input M for* `EXPSIZE-MATRIX-POWER` *in the following manner:*

Given accuracy parameter n and index $i', j' \in [1, 2^k]$, $\mathcal{B}$ outputs an approximation of $M_{i',j'}$ to error 2^{-n}. Moreover, the matrix M approximated by $\mathcal{B}$ has the property $(M^E)_{I,J} = 1$ if $\mathcal{T}$ halts without using more than t cells of the tape, and 0 otherwise.

Remark 3.2.3. *Note that, unlike the result for inner products, here we only show we can reduce languages, i.e. decision problems, into matrix powering. The main reason is that our definition of matrix powering requires the matrix to have bounded powers, so that all the entries will be restricted in the range $[-1, 1]$; otherwise, due to the exponential power, the entries can blow up exponentially. We enforce this by requiring the absolute row sums to be in the range $[-1, 1]$. Decision problems, whose outputs are in $\{0, 1\}$, form a natural candidate to be reduced into our problem; meanwhile, function problems, with unrestricted output, seem difficult to adapt.*

3.2.1 Proof of Theorem 3.2.1

The idea is to perform the algorithm known as exponentiation-by-squaring. In essence, we use the following identity:

$$M^e = \begin{cases} (M^{e/2})^2 & \text{if } e \text{ is even,} \\ M \cdot M^{e-1} & \text{if } e \text{ is odd} \end{cases}$$

By recursively applying the algorithm, it can be shown that this algorithm requires at most $2 \log e$ multiplications. Therefore, although the exponent e is exponential in the parameter $\lambda = k + n + \log e$, the number of multiplications is polynomial in λ.

Another issue is that M is also exponential-size; it is of order $2^k \times 2^k$. It is clear that we cannot hope to compute M^e in polynomial time. However, it turns out we can compute it in polynomial space. The sketch of the proof is that computing inner product requires little space, and matrix product is just a composition of several levels of inner products. The trick is, each level of inner product can reuse the same space by recomputing entries as needed, leading to little space requirement. This same trick is used in the proof of Savitch's theorem [8].

Before going into the proof, we will prove a quick lemma about approximating an approximation.

Lemma 3.2.4. ***(a)*** *Let x be a real number. Let $\hat{x}_1$ be an approximation of x to error δ_1. Let $\hat{x}_2$ be an approximation of $\hat{x}_1$ to error δ_2. Then $\hat{x}_2$ is an approximation of x to error $\delta_1 + \delta_2$.*

(b) *Let x be a real number in $[a, b]$ for some reals a, b. Let $\hat{x}_1$ be an approximation of x to error δ. Let $\hat{x}_2$ be a "clamping" of $\hat{x}_1$ into the interval $[a, b]$; that is,*

$$\hat{x}_2 = \begin{cases} a & \text{if } \hat{x}_1 < a, \\ \hat{x}_1 & \text{if } a \leq \hat{x}_1 \leq b, \\ b & \text{if } b < \hat{x}_1 \end{cases}$$

Then $\hat{x}_2$ is an approximation of x to error δ.

Proof of Lemma 3.2.4. **(a)** Immediate from triangle inequality:

$$|x - \hat{x}_2| = |(x - \hat{x}_1) + (\hat{x}_1 - \hat{x}_2)| \leq |x - \hat{x}_1| + |\hat{x}_1 - \hat{x}_2| \leq \delta_1 + \delta_2.$$

(b) If $\hat{x}_2 = \hat{x}_1$ the conclusion immediately follows. Otherwise, without loss of generality suppose $\hat{x}_2 = b$. Then $x \le \hat{x}_2 \le \hat{x}_1$, so $|x - \hat{x}_2| \le |x - \hat{x}_1| \le \delta$. □

The proof is divided into two parts. In the first part, we take the unrealistic assumption that we can do real arithmetic operations exactly without loss of accuracy, in constant time and space. We construct an algorithm to establish correctness. In the second part, we drop the assumption and construct an algorithm working with approximations of real numbers, and translate the proof of correctness into the new algorithm.

Part 1: Ideal algorithm

For now, we assume real arithmetic operations can be done exactly. Fix k. To implement the idea of exponentiation-by-squaring, we will use a subroutine $\mathcal{P}$ that acts as one "level" of the recursion. $\mathcal{P}$ takes integer inputs e, i, j, where e is the exponent and i, j are the indices. The subroutine works as follows. Here, $\mathcal{P}(e, i, j)$ means the output of $\mathcal{P}$ on input (e, i, j).

```
1. If $e = 1$, return $M_{i,j}$ from the input.
2. Set $s_0 \leftarrow 0$. This will be an accumulator for the inner product.
3. For each $m = 1, \dots, 2^k$,
   3.1. If $e$ is even, let $M'_{i,m} = \mathcal{P}(e/2, i, m)$ and $M''_{m,j} = \mathcal{P}(e/2, m, j)$.
   3.2. If $e$ is odd, let $M'_{i,m} = \mathcal{P}(1, i, m)$ and $M''_{m,j} = \mathcal{P}(e-1, m, j)$.
   3.3. Set $s_m \leftarrow s_{m-1} + M'_{i,m} \cdot M''_{m,j}$.
4. Return $s_{2^k}$.
```

We claim that $\mathcal{P}(e, i, j)$ outputs $(M^e)_{i,j}$, by induction on e. When $e = 1$, this is obvious from step 1. Otherwise, $\mathcal{P}$ runs step 3.

In both step 3.1 and step 3.2, $\mathcal{P}$ generates numbers $M'_{i,m}$ and $M''_{m,j}$, and in step 3.3, $\mathcal{P}$ adds the result to s. Therefore we can induct on h to get

$$s_h = \sum_{m=1}^{h} M'_{i,m} \cdot M''_{m,j}$$

and so s_{2^k} is the inner product of the vectors $M'_{i,\cdot} = (M'_{i,1}, \dots, M'_{i,2^k})$ and $M''_{\cdot,j} = (M''_{1,j}, \dots, M''_{2^k,j})$. Therefore, if M' and M'' are interpreted as $2^k \times 2^k$ matrices, s_{2^k} is the (i, j)-th entry of the product $M' \cdot M''$.

Finally, using the inductive hypothesis, in step 3.1 we have $M' = M'' = M^{e/2}$ so $M' \cdot M'' = M^e$, while in step 3.2 we have $M' = M$ and $M'' = M^{e-1}$ so $M' \cdot M'' = M^e$. Therefore, in either case, $\mathcal{P}$ outputs $s_{2^k} = (M^e)_{i,j}$.

Part 2: Approximation algorithm

Now we will construct a subroutine $\mathcal{P}'$ based on $\mathcal{P}$, but working with approximations of real numbers instead. Fix n; let δ_n be a real number in the form 2^{-q}, for some integer q to be determined later. We also assume $\delta_n \ll 1$. $\mathcal{P}'$ takes integer inputs e, i, j just like $\mathcal{P}'$, where e is the exponent and i, j are the indices, and returns $(M^e)_{i,j}$.

The subroutine $\mathcal{P}'$ works as follows. Assume the element $M_{i,j}$ from the input is approximated as $b_{k,n,i,j}/2^q$.

```
1. If $e = 1$, return $b_{k,n,i,j}$ from the input.
2. Set $s_0 \leftarrow 0$. This will be an accumulator for the inner product.
3. For each $m = 1, \dots, 2^k$,
    3.1. If $e$ is even, let $\hat{M}'_{i,m} = \mathcal{P}'(e/2, i, m)$ and $\hat{M}''_{m,j} = \mathcal{P}(e/2, m, j)$.
    3.2. If $e$ is odd, let $\hat{M}'_{i,m} = \mathcal{P}'(1, i, m)$ and $\hat{M}''_{m,j} = \mathcal{P}(e-1, m, j)$.
    3.3. Set $s_m \leftarrow s_{m-1} + \hat{M}'_{i,m} \cdot \hat{M}''_{m,j}$.
    3.4. Remove $\hat{M}'_{i,m}, \hat{M}''_{m,j}, s_{m-1}$ from memory.
4. Set $s \leftarrow \lfloor s_{2^k}/2^q \rfloor$.
5. If $s < -2^q$, set $s \leftarrow -2^q$. If $s > 2^q$, set $s \leftarrow 2^q$.
6. Return $s$.
```

We claim $\mathcal{P}'(e, i, j)$ outputs an integer s such that $s/2^q$ is an approximation of $(M^e)_{i,j}$ to error $2^{(k+3)\cdot r(e)}\delta_n$, where r is a function from $\{1, 2, \ldots\}$ to the naturals, satisfying the following recursion:

$$r(e) = \begin{cases} 0 & \text{if } e = 1, \\ 1 + r(e/2) & \text{if } e \text{ is even,} \\ 1 + r(e-1) & \text{if } e > 1 \text{ is odd} \end{cases}$$

Equivalently, $r(e) + 2$ is equal to the number of digits of e in binary plus the number of 1's when writing e in binary. It follows from this equivalence that $r(e) \leq 2\lceil \log e \rceil$.

We will prove the claim on $\mathcal{P}'$ by induction on e. For $e = 1$, we have $r(1) = 0$, so the claim is that $\mathcal{P}'(1, i, j)$ outputs s where $s/2^q$ is an approximation of $M_{i,j}$ to error δ_n. This is true by definition, as $\delta_n = 2^{-q}$.

Otherwise, by the inductive claim, the output of $\mathcal{P}'$ is the numerator of a fraction with denominator 2^q. Therefore, $\hat{M}'_{i,m}/2^q$ is an approximation of a real number $M'_{i,m}$; similarly, $\hat{M}''_{m,j}/2^q$ is an approximation of a real number $M''_{m,j}$. Therefore,

$$\begin{aligned} \frac{s_h/2^q}{2^q} &= \frac{1}{2^{2q}} \cdot \sum_{m=1}^{h} \hat{M}'_{i,m} \cdot \hat{M}''_{m,j} \\ &= \sum_{m=1}^{h} \frac{\hat{M}'_{i,m}}{2^q} \cdot \frac{\hat{M}''_{m,j}}{2^q} \end{aligned}$$

and so $\frac{s_{2^k}/2^q}{2^q}$ is an approximation of $M'_{i,\cdot} \cdot M''_{\cdot,j}$.

When e is even, $M' = M'' = M^{e/2}$ from the inductive claim. Therefore $\hat{M}'_{i,m}/2^q$ is an approximation of $M'_{i,m}$ with error $2^{(k+3)\cdot r(e/2)}\delta_n$, and $\hat{M}''_{m,j}/2^q$ is also an approximation of $M''_{m,j}$ with error $2^{(k+3)\cdot r(e/2)}\delta_n$. By Lemma 3.1.3,

$$\left| \frac{s_{2^k}}{2^{2q}} \right| = \left| \sum_{m=1}^{2^k} \frac{\hat{M}'_{i,m}}{2^q} \cdot \frac{\hat{M}''_{m,j}}{2^q} \right| \leq 2^{k+2} \cdot 2^{(k+3)\cdot r(e/2)}\delta_n.$$

Since $s = \lfloor s_{2^k}/2^q \rfloor$, it follows $|s - \frac{s_{2^k}}{2^q}| \leq 1$. Therefore $s/2^q$ is an approximation of $\frac{s_{2^k}/2^q}{2^q}$ with error 2^{-q}, and by Lemma 3.2.4, $s/2^q$ is an approximation of $M'_{i,\cdot} \cdot M''_{\cdot,j}$ with error

$$2^{k+2} \cdot 2^{(k+3)\cdot r(e/2)}\delta_n + \delta_n \leq 2^{(k+3)\cdot(1+r(e/2))}\delta_n = 2^{(k+3)\cdot r(e)}\delta_n.$$

The same approach works when e is odd. The only difference is that the error of $s_{2^k}/2^{2q}$ is

$$2^{k+1} \cdot (2^{(k+3)\cdot r(e-1)} + 1)\delta_n$$

and so the error of $s/2^q$ is

$$2^{k+1} \cdot (2^{(k+3)\cdot r(e-1)} + 1)\delta_n + \delta_n \leq 2^{(k+3)\cdot(1+r(e-1))}\delta_n = 2^{(k+3)\cdot r(e)}\delta_n.$$

Therefore, $\mathcal{P}'(e,i,j)$ is an approximation of $(M^e)_{i,j}$ to error $2^{(k+3)\cdot r(e)}\delta_n$, proving the claim. Furthermore, using $r(e) \leq 2\lceil \log e \rceil$, we can pick $q = n + 2(k+3)\lceil \log E \rceil$ and $\delta_n = 2^{-q}$ so that $\mathcal{P}'(E,I,J)$ is an approximation of $(M^E)_{I,J}$ to error 2^{-n}.

Finally, we analyze the memory used by $\mathcal{P}'$. The recursion depth of $\mathcal{P}'$ is exactly $r(E) = O(\log E)$. In each level of recursion, $\mathcal{P}'$ only needs to remember $m, s_{m-1}, \hat{M}'_{i,m}, \hat{M}''_{m,j}$; when we advance to the next value of m, we can forget the previous values as explicitly directed in step 3.4, since we no longer need their values.

Since we know $\|M^e\| \leq 1$, it follows $|\hat{M}'_{i,m}|, |\hat{M}''_{m,j}| \leq 2^q$ and $|s_{m-1}| \leq 2^{2q}$. It is also clear $m \leq 2^k$. Therefore the total memory needed to store the numbers of one level is $O(\log 2^q + \log 2^q + \log 2^{2q} + \log 2^k) = O(k+q) = O(n + k\log E)$. As there are $O(\log E)$ levels in total, the memory needed in total is $O(n\log E + k\log^2 E)$; calls on the same level reuse the same memory space. This amount is polynomial in $\lambda = k + n + \log E$.

Finally, $\mathcal{A}$ just calls $\mathcal{P}'(E,I,J)$ and returns the result. This gives us a polynomial-space algorithm to compute $(M^E)_{I,J}$, therefore showing `EXPSIZE-MATRIX-POWER` $\in \mathbb{R}$PSPACE. □

Remark 3.2.5. *Savitch's algorithm [8] is very similar to this proof. The main differences are as follows: the entries are only 0 and 1 instead of real numbers; the inner product of two vectors is 1 if some component is 1 in both vectors, and 0 otherwise; and there is no restriction in matrix norm. Therefore, the input matrix is the adjacency matrix of a directed graph; Step 3.3 of the algorithm changes to $S_m \leftarrow \max\{S_{m-1}, M'_{i,m} \cdot M''_{m,j}\}$; and the meaning of $(M^e)_{i,j}$ is changed to the following: there exists a path from i to j with length e. This shows that graph reachability is in* PSPACE.

3.2.2 Proof of Theorem 3.2.2

Just like Theorem 3.1.2, we will ignore the real part; our matrix will be a 0-1 matrix. Not only that, but each row has at most one element with value 1. Then this matrix clearly satisfies the bounded norm condition.

For the Turing machine $\mathcal{T}$, define a "basic configuration" to be a tuple (T,h,s) where T is the current tape contents, h is the position of the head, and s is the state of the head. Note that a basic configuration completely describes the current state of $\mathcal{T}$. We also define two "special configurations": "accept" and "reject". The configuration "accept" means the machine halted without overstepping the memory limit; the configuration "reject" means the machine tried to step over the memory limit.

Let C be the set of configurations. We also define a transition relation $\to$ on C as follows: if in one step, $\mathcal{T}$ moves from configuration c_1 to c_2, then we say $c_1 \to c_2$. Note that since $\mathcal{T}$ is deterministic, from any basic configuration c_1 there is exactly one c_2 such that $c_1 \to c_2$. We also define accept $\to$ accept and reject $\to$ reject, therefore turning $\to$ into a function.

Since $\mathcal{T}$ is limited to t cells of memory, the total number of basic configurations is $O(1)^t \times t \times O(1)$: there are $O(1)^t$ possible tape contents as there are a constant number of symbols for each of the t cells, there are t possible positions for the head, and there are a constant number of states. There are also $2 = O(1)$ special configurations. Therefore, in total there are $2^{O(t)}$ possible configurations. That means we can compute, in polynomial time, an integer k such that the total number of configurations is $\leq 2^k$. This k is the index to `EXPSIZE-MATRIX-POWER`.

Enumerate the configurations as $c_1, c_2, \ldots, c_{|C|}$. Define the input matrix M as follows:

$$M_{i,j} = \begin{cases} 1 & \text{if } i,j \le |C| \text{ and } c_i \to c_j, \\ 0 & \text{if } i,j \le |C| \text{ and } c_i \not\to c_j, \\ 1 & \text{if } i,j > |C| \text{ and } i = j, \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Each entry of this matrix can be computed in polynomial time as follows. First, if either of i, j is $> |C|$, we simply check for equality. Otherwise, we can convert them to the configurations c_i, c_j, then we take a step of $\mathcal{T}$ from configuration c_i and check if the result is c_j; simulating a step of the Turing machine takes polynomial time. For simplicity, in the rest of this proof, we will assume that M has size $|C| \times |C|$.

We claim $(M^e)_{i,j} = 1$ if and only if $\mathcal{T}$ moves from configuration c_i to c_j in exactly e steps, by induction on e. When $e = 1$, this is obvious by definition. Otherwise, we know

$$(M^e)_{i,j} = \sum_{m=1}^{|C|} M_{i,m} \cdot (M^e)_{m,j}.$$

Since $\mathcal{T}$ is deterministic, there is exactly one m such that $c_i \to c_m$. Let m_0 be such m; then $M_{i,m_0} = 1$ and $M_{i,m} = 0$ for all other m, and so we have

$$(M^e)_{i,j} = (M^{e-1})_{m_0,j}.$$

But this just says, since the first step of $\mathcal{T}$ is moving from configuration c_i to c_{m_0}, the only way $\mathcal{T}$ moves from c_i to c_j in e steps is if $\mathcal{T}$ moves from c_{m_0} to c_j in $e-1$ steps. This proves the claim.

Let c_s be the starting configuration of $\mathcal{T}$ and c_a be the accepting configuration. Consider 2^k steps of $\mathcal{T}$ starting from c_s; let the execution be $c_s = c'_0 \to c'_1 \to \ldots \to c'_{2^k}$. Since there are $|C| \le 2^k$ configurations in total but the execution has $2^k + 1$ configurations visited, at least one configuration is repeated twice. Since $\mathcal{T}$ is deterministic, this forms a cycle; therefore, the execution ends in a cycle. If this cycle is $c_a \to c_a$, then $\mathcal{T}$ halts within the memory bounds. Otherwise, either $\mathcal{T}$ is stuck in the rejecting configuration for having stepped over memory bounds, or $\mathcal{T}$ is stuck in an infinite loop; either way, it means $\mathcal{T}$ does **not** halt within the memory limit of t cells of tape.

Therefore, $(M^{2^k})_{c_s,c_a} = 1$ if and only if $\mathcal{T}$ halts without using more than t cells of tape. Therefore $\mathcal{R}$ works as follows:

1. Compute the value of k (based on $\mathcal{T}$ and t).
2. Fix some enumeration of the configurations. Let c_s, c_a be the starting configuration and accepting configuration respectively.
3. Compute $E = 2^k, I = c_s, J = c_a$.
4. Define $\mathcal{B}$ as follows: on input i', j', it computes $M_{i',j'}$ as defined in Equation 3.1.

Finally, $\mathcal{R}$ runs in time poly(t), and $\mathcal{B}$ also runs in time poly(k) = poly(t). So this is a correct polynomial-time reduction. □

3.3 Polynomial powering is in $\mathbb{R}\#\mathsf{P}$

The main result of this section is that computing polynomial powering is in $\mathbb{R}\#\mathsf{P}$. In Section 4 we show that some forms of matrices, which occur from solving differential equations using difference schemes, can be converted into polynomials; therefore, for those matrices, the matrix powering result can be improved from $\mathbb{R}$PSPACE to $\mathbb{R}$#P.

Theorem 3.3.1. *For any fixed v, d, POLYNOMIAL-POWER is in $\mathbb{R}$#P.*

3.3.1 Proof of Theorem 3.3.1

We first recall the following theorems.

Fact 3.3.2 (Cauchy's integral formula for polynomials)**.** *If $P(x_1, \dots, x_v)$ is a polynomial and $I_1, \dots, I_v$ are natural numbers, then*

$$P[x_1^{I_1} \dots x_v^{I_v}] = \frac{1}{(2\pi i)^v} \oint_v \dots \oint_1 \frac{P[z_1, \dots, z_v]}{z_1^{I_1+1} \dots z_v^{I_v+1}} \, dz_1 \dots dz_v$$

where i is the imaginary unit and each $\oint_j$ is a complex integral taken on a loop around the origin with winding number 1.

Fact 3.3.3 (Integration is in $\mathbb{R}$#P)**.** *If $f : [0,1] \to \mathbb{R}$ is bounded and polynomial-time computable, then the function*

$$x \mapsto \int_0^x f(t) \, dt$$

is in $\mathbb{R}$#P. [3, Thm5.32 p.184]

For Fact 3.3.3, we can in fact say more. Although it says we need f to be polynomial-time computable and have codomain $\mathbb{R}$, we can generalize this: f just needs to be in $\mathbb{R}$#P and have codomain $\mathbb{C}$. In Ko [3], the proof of Fact 3.3.3 approximates f using step functions $\{f_n\}$ that converge to f; here, step functions are piecewise constant functions, which are easy to integrate. We can still approximate $\mathbb{R}$#P-computable functions using step functions; the f_n's will be in $\mathbb{R}$#P, but this just means the whole integration remains in $\mathbb{R}$#P. We can also use codomain $\mathbb{C}$ as we can as easily approximate complex-valued functions using step functions.

The proof now follows immediately. Using Fact 3.3.2, we can express our desired coefficient as a series of complex integrals. For each $\oint_j$, we take the path $|z_j| = 1$, and we also transform the polynomial with polar substitution $z_j \mapsto w_j \exp(\theta_j 2\pi i)$ so that $\oint_j$'s path becomes $\int_0^1$. The function in the integral is $\mathbb{R}$#P-computable (in fact polynomial-time computable), thus by Fact 3.3.3, the entire integral is in $\mathbb{R}$#P. □

3.4 Certain cases of polynomial powering is in $\mathbb{R}$P

We show that specific cases of polynomial powering are in $\mathbb{R}$P. While the restrictions are rather severe, leading to limited utility, the ideas used for the proof are rather novel. It might be possible to extend these ideas for a larger class of cases of polynomial powering.

Theorem 3.4.1. *Consider the following restriction of POLYNOMIAL-POWER: $v = d = 1$, the polynomial P is fixed, and $n = \lfloor c \log E \rfloor$ for a fixed real $c > 0$, where E is the exponent. Then this problem is in $\mathbb{R}$P.*

Remark 3.4.2. *By "the polynomial P is fixed", it means for different polynomials, we have different algorithms. The only inputs to the problem are the exponent E, the index I of the indeterminate, and sufficiently accurate approximations of the coefficients of P. (Recall that an algorithm should receive sufficiently accurate approximations of its input real numbers, depending on the desired output accuracy;*

since our output accuracy n depends on E, which is part of the input, we do need the input real numbers to be more precise as E gets larger.)

3.4.1 Proof of Theorem 3.4.1

Before we proceed to the proof, we will first recall the following theorems.

Fact 3.4.3 (Chernoff–Hoeffding theorem [2])**.** *Let $X_1, \dots, X_n$ be independent and identically distributed random variables taking values on $\{0,1\}$. Let $X = \sum X_i/n$ and $p = E[X]$. Then, for any real $\varepsilon > 0$,*

$$\Pr(|X-p| \geq \varepsilon) \leq 2 \cdot \exp(-2\varepsilon^2 n).$$

Fact 3.4.4 (Stirling series)**.** *For every natural number t, there exist rational numbers $a_1, \dots, a_{t-1}, b_t$ such that, for all complex number z with positive real part,*

$$\ln \Gamma(z) = z \ln z - z + \frac{1}{2} \ln \frac{2\pi}{z} + \sum_{i=1}^{t-1} \frac{a_i}{z^{2i-1}} + R_t(z)$$

where Γ is the Gamma function and

$$|R_t(z)| \leq \frac{b_t}{|z|^{2t-1}}.$$

Remark 3.4.5. *In this proof we will only use Stirling series where z is a positive integer; therefore, $\Gamma(z) = (z-1)!$. The rational numbers $a_1, \dots, a_{t-1}, b_t$ are related to Bernoulli numbers and can be computed exactly, but it is not clear if they can be computed in time polynomial in t.*

We also prove the following lemmas about approximating $\log x$ and $\exp(x)$.

Lemma 3.4.6. ***(a)*** *Let x be a real number satisfying $|x| \leq \frac{1}{2}$, and $\hat{x}$ be an approximation of x to error δ. Then for any natural number k, it is possible to approximate $\ln(1+x)$ to error $k\delta + 2^{-k}$. In particular, if $\delta \leq 2^{-(n+\log n+2)}$, it is possible to approximate $\ln(1+x)$ to error 2^{-n}.*

(b) *Let $x > 0$ be a real number, and $\hat{x}$ be an approximation of x to error δx where $\delta \in (0, 1/2)$. Then $\ln \hat{x}$ is an approximation of $\ln x$ with error 2δ.*

Proof of Lemma 3.4.6. **(a)** By Taylor expansion, we have

$$\ln(1+x) = \sum_{i=1}^{\infty} \frac{(-1)^{i+1} x^i}{i} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

which is valid for $|x| < 1$. In particular, since our assumption is that $|x| \leq \frac{1}{2}$, this expansion holds. By truncating it on the k-th term and using $\hat{x}$ instead of x, we obtain an approximation of $\ln(1+x)$ as

$$\sum_{i=1}^{k} \frac{(-1)^{i+1} \hat{x}^i}{i}.$$

The error of this approximation can be written as follows.

$$\begin{aligned} &\left| \sum_{i=1}^{\infty} \frac{(-1)^{i+1} x^i}{i} - \sum_{i=1}^{k} \frac{(-1)^{i+1} \hat{x}^i}{i} \right| \\ &\leq \left| \sum_{i=1}^{k} \frac{(-1)^{i+1} (x^i - \hat{x}^i)}{i} \right| + \left| \sum_{i=k+1}^{\infty} \frac{(-1)^{i+1} x^i}{i} \right| \\ &\leq \sum_{i=1}^{k} \frac{|x^i - \hat{x}^i|}{i} + \sum_{i=k+1}^{\infty} \frac{|x^i|}{i} \end{aligned}$$

The first summation is the error obtained by approximating x as $\hat{x}$, and the second summation is the error obtained by removing all but the first k terms.

The first summation can be bounded as follows. Since $|x| < 1$, we can use Lemma 3.1.3 to show that $\hat{x}^i$ is an approximation of x^i to error $i\delta$. Therefore the first summation becomes

$$\sum_{i=1}^{k} \frac{|x^i - \hat{x}^i|}{i} \leq \sum_{i=1}^{k} \frac{i\delta}{i} = k\delta$$

The second summation can be bounded by noting $|x| \leq \frac{1}{2}$, so this series decays faster than a geometric series, as follows.

$$\sum_{i=k+1}^{\infty} \frac{|x^i|}{i} \leq \sum_{i=k+1}^{\infty} |x^i| \leq \sum_{i=k+1}^{\infty} \left(\frac{1}{2}\right)^i = 2^{-k}$$

Therefore, the error of the approximation is $k\delta + 2^{-k}$. By using $\delta \leq 2^{-(n+\log n+2)}$ and $k = n+1$, this error becomes

$$k\delta + 2^{-k} \leq (n+1) \cdot 2^{-(\log 2n)} \cdot 2^{-n-1} + 2^{-n-1} \leq 2^n.$$

(b) Let $\hat{x} = \hat{c}x$. We have

$$|\ln \hat{x} - \ln x| = |(\ln \hat{c} + \ln x) - \ln x| = |\ln \hat{c}|.$$

By definition of $\hat{c}$, $\hat{c}$ is an approximation of 1 to error δ. Therefore we have

$$1 - \delta \leq \hat{c} \leq 1 + \delta \quad \Longrightarrow \quad \ln(1-\delta) \leq \ln \hat{c} \leq \ln(1+\delta)$$

We again use Taylor expansion, but this time going for a different estimate. When $|z| \leq 1/2$,

$$\begin{aligned} |\ln(1+z)| &= \left| \sum_{i=1}^{\infty} \frac{(-1)^{i+1} z^i}{i} \right| \\ &\leq \sum_{i=1}^{\infty} \left| \frac{(-1)^{i+1} z^i}{i} \right| \\ &\leq \sum_{i=1}^{\infty} |z|^i \\ &\leq \sum_{i=1}^{\infty} |z| \cdot \left(\frac{1}{2}\right)^{i-1} = 2|z| \end{aligned}$$

Therefore, applying this with $z = \delta$ and $z = -\delta$ gives

$$-2\delta \leq \ln \hat{c} \leq 2\delta \quad \Longrightarrow \quad |\ln \hat{c}| \leq 2\delta.$$

Therefore, $\ln \hat{x}$ is an approximation of $\ln x$ with error $|\ln \hat{c}| \leq 2\delta$. □

Remark 3.4.7. *We can also compute* $\ln \hat{x}$ *in part (b) of the lemma. First, multiply or divide* $\hat{x}$ *by* $e = \exp(1)$ *until it falls in the interval* $(1/2, 3/2)$*; this must happen, even with approximated* $\hat{x}$*, since* $\frac{3/2}{1/2} = 3 > e$*. Each multiplication by e corresponds to adding 1 to the logarithm; each division corresponds to subtracting 1 from the logarithm. Once* $\hat{x}$ *is in the interval* $(1/2, 3/2)$*, we use part (a) of the lemma to approximate* $\ln \hat{x}$*, and add/subtract the appropriate number of 1's.*

Lemma 3.4.8. *For any natural number n, there exists* δ *satisfying the following. Let* $x \leq 0$ *be a real number, and* $\hat{x}$ *be an approximation of* x *to error* δ*. Then* $\exp(x)$ *can be approximated to error* 2^{-n}*, in time polynomial in* $n + \ell(x)$*.*

Proof of Lemma 3.4.8. First, note that if $-x > \ln 2 \cdot n$, then $\exp(x) < 2^{-n}$ and thus 0 is an approximation of $\exp(x)$. Thus assume $-x \leq \ln 2 \cdot n$.

Using Taylor expansion,

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

Just like in Lemma 3.4.6(a), we approximate this by truncating at the k-th term and using x instead of $\hat{x}$. We assume $k > en$ and write $k = en + k'$. Then the error is

$$\leq \sum_{i=0}^{k} \frac{|x^i - \hat{x}^i|}{i!} + \sum_{i=k+1}^{\infty} \frac{|x^i|}{i!}.$$

The first part of the error is bounded by

$$\sum_{i=0}^{k} \frac{|x^i - \hat{x}^i|}{i!} \leq \sum_{i=0}^{k} \frac{i\delta}{i!} \leq e\delta.$$

The second part of the error is bounded by

$$\sum_{i=k+1}^{\infty} \frac{|x^i|}{i!} \leq \frac{|x|^k}{k!} \cdot \sum_{i=1}^{\infty} \frac{(\ln 2 \cdot n)^i}{(en)^i} \leq 2 \cdot \frac{|x|^k}{k!}.$$

In turn, using $k = en + k'$, we can further bound it as

$$2 \cdot \frac{|x|^k}{k!} \leq 2 \cdot \frac{(\ln 2 \cdot n)^{en}}{(en)!} \cdot \frac{(\ln 2 \cdot n)^{k'}}{(en)^{k'}}$$

By Stirling's approximation (Fact 3.4.4), $(en)! \geq n^{en}$, thus the second term is bounded by $\leq (\ln 2)^{en} \leq 1$. Since $(\ln 2)/e < 1/2$, the third term is bounded by $2^{-k'}$. Therefore this error is bounded by $2^{-k'+1}$.

In total, the error of approximating $\exp(x)$ using truncated Taylor series is

$$e\delta + 2^{-k'+1}.$$

By picking $\delta = 2^{-n-\log e-1}$ and $k' = n + 2$, we obtain the desired error 2^{-n}. This approximation runs in time $\text{poly}(k) = O(\text{poly}(n))$ as $k = en + (n+2) = O(n)$. (The $\ell(x)$ part of the running time is only used at the beginning, to check if $-x > \ln 2 \cdot n$.) $\square$

We will now begin the proof. Due to the restriction $v = d = 1$, the polynomial is $P(X) = aX + b$ for some reals a, b satisfying $|a| + |b| \leq 1$. We will first prove the claim when $a \in (0, 1/2]$ and $b = 1 - a$; then we will generalize the result.

Part 1: $P(X) = aX + (1-a)$ **with** $a \in [0, 1/2]$

Then the coefficient of X^I in P^E is given by

$$P^E[X^I] = \binom{E}{I} a^I (1-a)^{E-I}.$$

Note that $P^E[X^I] \in (0, 1]$ as long as $I \in [0, E]$.

Taking the logarithm gives

$$\log P^E[X^I] = \log E! - \log I! - \log(E-I)! + I \log a + (E-I)\log(1-a) \tag{3.2}$$

The idea of the algorithm is the following.

1. If $\left|a - \frac{I}{E}\right|$ is sufficiently large, by using Chernoff–Hoeffding theorem, we can approximate $P^E[X^I]$ with 0.
2. Otherwise, by using Stirling's formula, we can approximate $\log E!, \log I!, \log(E-I)!$ with a sufficiently small error.
3. Using Lemma 3.4.6, $\log a$ and $\log(1-a)$ can be approximated with small error. Therefore, $\log P^E[X^I]$ can be approximated with small error. Using Lemma 3.4.8, we also can approximate $P^E[X^I]$ with small error.

Step 1: Approximating with 0 if $P^E[X^I]$ is too small in the first place

Consider the following probabilistic interpretation. Let $Y_1, \ldots, Y_E$ be independent and identically distributed random variables, such that $Y_i = 1$ with probability a and $Y_i = 0$ with probability $1-a$. Let $Y = \sum Y_i$. Then the distribution of Y is given by the binomial distribution with parameters E and a; in particular,

$$\Pr(Y = I) = \binom{E}{I} a^I (1-a)^{E-I} = P^E[X^I].$$

At the same time, by using Fact 3.4.3 and taking $\varepsilon = \frac{1}{2}a$,

$$\Pr\left(\left|\frac{Y}{E} - a\right| \geq \frac{1}{2}a\right) \leq 2 \cdot \exp\left(-\frac{1}{2}a^2 E\right).$$

Assume I is such that $\left|a - \frac{I}{E}\right| \geq \frac{1}{2}a$. Then

$$\Pr(Y = I) \leq \Pr\left(\left|\frac{X}{E} - a\right| \geq \frac{1}{2}a\right)$$

as the event $Y = I$ is a subset of the event $\left|\frac{Y}{E} - a\right| \geq \frac{1}{2}a$. Therefore,

$$0 \leq P^E[X^I] \leq 2 \cdot \exp\left(-\frac{1}{2}a^2 E\right).$$

Since a is fixed, the right hand side decays exponentially in E. On the other hand, the required accuracy

$$2^{-n} = 2^{-\lfloor c \log E \rfloor} = O(E^{-c})$$

decays polynomially in E. Therefore, there exists some E_0 such that, for all $E' \geq E_0$,

$$0 \leq P^E[X^I] \leq 2 \cdot \exp\left(-\frac{1}{2}a^2 E'\right) < 2^{-\lfloor c \log E' \rfloor}.$$

Therefore, if $E \geq E_0$ and I satisfies $\left|a - \frac{I}{E}\right| \geq \frac{1}{2}a$, we can output 0.

Step 2: Approximating $\log E!, \log I!, \log(E-I)!$ with small error

Suppose $\left|a - \frac{I}{E}\right| < \frac{1}{2}a$. Therefore $\frac{I}{E} > \frac{1}{2}a$. Also, since we assume $a \leq \frac{1}{2}$, we have $\frac{I}{E} \leq \frac{3}{4} < 1 - \frac{1}{2}a$, therefore $\frac{E-I}{E} > \frac{1}{2}a$.

By using Fact 3.4.4, for some t to be determined later, there exists some real number b_t such that, for every natural number m, we can approximate $\ln m!$ with some real number $S_t(m)$ with error

$$\leq \frac{b_t}{(m+1)^{2t-1}}.$$

Since $\log m! = \log e \cdot \ln m!$, it follows we can also approximate $\log m!$ with some real number $S_t'(m)$ with error

$$\leq \frac{\log e \cdot b_t}{(m+1)^{2t-1}}.$$

Moreover, it can also be seen from Fact 3.4.4 that $S'_t(m)$ is computable in time $\mathrm{poly}(\log m)$.

We will apply this for $m = E, I, E-I$. We know $E, I, E-I > \frac{1}{2}aE$, therefore the error is

$$\leq \frac{\log e \cdot b_t (2/a)^{2t-1}}{E^{2t-1}}.$$

It follows that $S'_t(E) - S'_t(I) - S'_t(E-I)$ is an approximation of $\log E! - \log I! - \log(E-I)!$ with error

$$3 \cdot \frac{\log e \cdot b_t (2/a)^{2t-1}}{E^{2t-1}}.$$

Step 3: Approximating $P^E[X^I]$ with small error

Using Equation 3.2, we approximate

$$\log P^E[X^I] = \log E! - \log I! - \log(E-I)! + I \log a + (E-I)\log(1-a)$$

using the approximation

$$\hat{A} := \left(S_t(E) - S_t(I) - S_t(E-I)\right) + \left(I \log e \cdot \ln \hat{a} + (E-I)\log e \cdot \ln(1-\hat{a})\right).$$

The error of this approximation comes from two sources. The first part, by Step 2, is bounded by

$$3 \cdot \frac{b_t (2/a)^{2t-1}}{E^{2t-1}}.$$

The numerator is a constant for any fixed t. When $t = c$, this error is of order $O(E^{-2c-1})$, so this decays faster than $2^{-n} = O(E^{-c})$.

It remains to investigate the error from the second part. By Lemma 3.4.6 and Remark 3.4.7, $\ln \hat{a}$ and $\ln(1-\hat{a})$ can be approximated to any error $2^{-n'}$ as long as $\hat{a}$ approximates a with error $2^{-\mathrm{poly}(n')}a$. In particular, taking $n' = n + \log E + O(1)$ allows us to approximate

$$I \log e \cdot \ln \hat{a} + (E-I)\log e \cdot \ln(1-\hat{a})$$

with error

$$I \log e \cdot 2^{-n'-O(1)} + (E-I)\log e \cdot 2^{-n'-O(1)} = E \log e \cdot 2^{-O(1)} \cdot 2^{-n} \cdot 2^{-\log E}.$$

The factors E and $2^{-\log E}$ cancel, so this error is $\leq 2^{-n-O(1)}$.

Therefore, the total error of $\hat{A}$ is $\leq 2^{-n-O(1)}$.

Finally, we are looking for $P^E[X^I] = 2^{\hat{A}} = \exp(\ln 2 \cdot \hat{A})$. By using Lemma 3.4.8, since $\ln 2 \cdot \hat{A}$ can be approximated to error $2^{-n-O(1)}$, we also can approximate $P^E[X^I]$ to error $2^{-n-O(1)}$, in particular to error 2^{-n}. Each part of this approximation can be computed in polynomial time, so this gives a polynomial-time algorithm. □

Part 2: $P(X) = aX + b$

We now generalize this to a general linear polynomial. The coefficient is

$$\binom{E}{I} a^I b^{E-I}.$$

Note that the sign can be easily determined: $\binom{E}{I}$ is always positive, a^I contributes the sign $\mathrm{sgn}(a)^I$, and b^{E-I} contributes the sign $\mathrm{sgn}(b)^{E-I}$. Moreover, the sign does not affect the magnitude of the coefficient. So for simplicity, assume $a, b \geq 0$.

We scale the polynomial as

$$a = \frac{a'}{a+b} \qquad b = \frac{b'}{a+b}$$

so the coefficient becomes

$$(a+b)^E \cdot \binom{E}{I} (a')^I (b')^{E-I}.$$

Since $a + b \leq 1$, we can approximate the right term with Part 1 normally with error $2^{-n'}$ where $n' = (c+1)\log E$. We can also easily approximate the left term with error $2^{-n'}$; therefore, we can approximate the coefficient with error $2 \cdot 2^{-n'} \leq 2^{-n}$. $\square$

Chapter 4. Applications

The main motivation for the main results is from solving differential equations by using difference schemes (also known as finite difference method). Given an ordinary or partial differential equation, we divide all spatial and temporal dimensions into small discrete time steps, and convert the differential equation into a difference scheme. When the differential equation is linear and has constant coefficients, the difference scheme is a system of linear recurrences in the temporal dimension. A system of linear recurrences in general is just the matrix equation

$$\vec{u}_{t+1} = M \cdot \vec{u},$$

which by induction on t gives

$$\vec{u}_t = M^t \cdot \vec{u}_0.$$

Therefore, by computing matrix powers and inner products, we can evaluate $\vec{u}_t$ at any t, therefore approximating the solution of the system at any time.

4.1 Linear recurrences and linear ordinary differential equations

Consider the following system of linear recurrences:

$$\begin{pmatrix} a_{t+1} \\ b_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a_t \\ b_t \end{pmatrix}.$$

This system describes the Fibonacci sequence: with $a_0 = 1, b_0 = 0$, we have $a_t = F_{t+1}, b_t = F_t$ where F_t is the t-th Fibonacci number. We can also express other sequences that are similarly defined recursively in this manner, as a matrix equation. This is a typical use of a system of linear recurrences, with small dimensions but entries that grow arbitrarily large.

Similarly, consider the following linear ordinary differential equation:

$$-f''(x) = f(x).$$

Using difference schemes, we may approximate

$$f''(x) \approx \frac{f(x+2h) - 2f(x+h) + f(x)}{2h^2},$$

and so for a time step of h, we obtain the linear recurrence

$$f(x+2h) \approx 2f(x+h) + (-1-2h^2)f(x).$$

This in turn can be converted into the following system of linear recurrences, by defining $a_t = f(th)$ and $b_t = f((t+1)h)$:

$$\begin{pmatrix} a_{t+1} \\ b_{t+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1-2h^2 & 2 \end{pmatrix} \cdot \begin{pmatrix} a_t \\ b_t \end{pmatrix}.$$

In general, linear ordinary differential equations can be converted into systems of linear recurrences with small dimensions; to be precise, ordinary differential equations of order k can be converted into systems of k linear recurrences.

Consider a system of linear recurrences defined by the matrix equation

$$\vec{u}_{t+1} = M \cdot \vec{u}_t,$$

where $\vec{u}_t$ has dimension k and M has dimension $k \times k$. Then by induction on t, we have

$$\vec{u}_t = M^t \cdot \vec{u}_0,$$

so given the initial values $\vec{u}_0$ and t, we can obtain $\vec{u}_t$. From the motivation above, we are interested to investigate when k is small (so the matrix can be read off in full), M is unrestricted (so we do not have any stability constraints), and t is small (so the assumption that M is not necessarily stable is not a problem). We can thus define two problems based on our exponential-size problems.

Definition 4.1.1. *The function `INNER-PRODUCT` with index k is defined to be exactly the same as the function `EXPSIZE-INNER-PRODUCT` in Definition 2.3.1, except that the dimensions of $\vec{u}, \vec{v}$ are k instead of 2^k.*

The function `MATRIX-POWER` with index k is defined to be the function `EXPSIZE-MATRIX-POWER` in Definition 2.3.2, except that the dimension of M is $k \times k$ instead of $2^k \times 2^k$.

Since `EXPSIZE-INNER-PRODUCT` and `EXPSIZE-MATRIX-POWER` are computable in polynomial space, it follows `INNER-PRODUCT` and `MATRIX-POWER` are computable in poly-log space. We make it more precise as follows. These results were previously published by the author [4].

Theorem 4.1.2. ***(a)*** *`INNER-PRODUCT` is in $\mathbb{R}\mathsf{SPACE}(\log)$.*

(b) *If the exponent E is written in unary, `MATRIX-POWER` is in $\mathbb{R}\mathsf{SPACE}(\log^2)$.*

(c) *If the exponent E is written in binary, `MATRIX-POWER` is in $\mathbb{R}\mathsf{P}$.*

Remark 4.1.3. *Here, the definition of using sublinear space such as $\mathbb{R}\mathsf{SPACE}(\log)$ follows the standard convention: the input is random-access read-only and the output is write-only, but neither is charged against the memory consumption. Only the working memory is charged.*

4.1.1 Proof of Theorem 4.1.2

(a) We will first solve a related problem: $\vec{u}, \vec{v}$ are integer vectors of k elements each, where each element is in the interval $[0, 2^p)$, and we wish to compute $\vec{u} \cdot \vec{v}$. The complexity parameter is $\lambda = k + p$; therefore, we want to use $O(\log \lambda) = O(\log(k+p))$ space.

First note that $\vec{u} \cdot \vec{v} \leq k \cdot 2^{2p}$. Therefore the length of the output is $O(\log k + p)$. Write each u_i in binary:

$$u_i = \sum_{j=0}^{p-1} u_{i,j} 2^j.$$

Note that this is possible since we assumed $u_i \in [0, 2^p)$. Define $u_{i,j} = 0$ for $j \geq p$. Also define $v_{i,j}$

similarly. Then

$$\begin{aligned}\vec{u}\cdot\vec{v} &= \sum_{i=1}^{k}\left(\sum_{j=0}^{p}u_{i,j}2^j\right)\cdot\left(\sum_{j=0}^{p}v_{i,j}2^j\right)\\ &= \sum_{i=1}^{k}\sum_{b=0}^{2p}\sum_{j=0}^{b}u_{i,j}2^j\cdot v_{i,b-j}2^{b-j}\\ &= \sum_{i=1}^{k}\sum_{b=0}^{2p}2^b\sum_{j=0}^{b}u_{i,j}v_{i,b-j}\\ A &:= \sum_{b=0}^{2p}2^b\cdot\left(\sum_{i=1}^{k}\sum_{j=0}^{b}u_{i,j}v_{i,b-j}\right)\end{aligned}$$

Here, the second line is derived by rearranging the terms in the product of sums. We take the terms in the form $u_{i,j}2^j\cdot v_{i,b-j}2^{b-j}$ into the b-th group; this is motivated by $2^j\cdot 2^{b-j}=2^b$ for all the terms in the group, so that we can take the 2^b out as in the third line. And furthermore, we can then interchange the summations on i and b as the only terms that depend on either i or b are the terms inside the double sum.

Note that for any b', the b'-th bit of A is also obtained as the b'-th bit of $A \mod 2^{b'+1}$, where $\mod$ is the modulo operation. Moreover, if $b > b'$, then the b-th term of the outer-most summation of A does not contribute to $A \mod 2^{b'+1}$, as 2^b divides $2^{b'+1}$ so the contribution is always 0. Therefore $A \mod 2^{b'+1}$ can be determined from the following:

$$A_{b'} := \sum_{b=0}^{b'}2^b\cdot\left(\sum_{i=1}^{k}\sum_{j=0}^{b}u_{i,j}v_{i,b-j}\right)$$

Moreover, the terms corresponding to $b < b'$ make up $A_{b'-1}$ by definition, so we have

$$A_{b'} = A_{b'-1} + 2^{b'}\cdot\sum_{i=1}^{k}\sum_{j=0}^{b'}u_{i,j}v_{i,b'-j}.$$

This gives rise to the following algorithm. Our algorithm will compute the bits of the output from the least significant bit; therefore, we assume that bits $0,1,\dots,b-1$ have been computed and we are computing bit b of the output.

Let $C_b = \lfloor |A_b|/2^b\rfloor$; in other words, C_b is what we get when we remove the last b bits of A_b. Put in another way, C_b is the part from computing A_b that still matter. (The last b bits of C_b are bits $0,1,\dots,b-1$, which no longer matter as they have been output. The b-th bit is what we are working on, and the $(b+1)$-th bit and above will matter later as carry.) Then by dividing the previous equation by $2^{b'}$ and taking floor,

$$C_{b'} = \left\lfloor\frac{C_{b'-1}}{2}\right\rfloor + \sum_{i=1}^{k}\sum_{j=0}^{b'}u_{i,j}v_{i,b'-j}.$$

Therefore $C_{b'}$ is obtained from the previous $C_{b'-1}$, plus a sum. We will investigate the memory requirements of $C_{b'}$.

We first look into the sum. Since i ranges in $[1,k]$ and j ranges in $[0,b']$, we need $O(\log k)$ space for i and $O(\log b')$ space for j. Each term $u_{i,j}v_{i,b'-j}$ is in the range $\{0,1\}$ (since $u_{i,j},v_{i,b'-j}\in\{0,1\}$), so the total sum is at most $k(b'+1)$; therefore, an accumulator for this sum takes space $O(\log(kb'))$. Similar to the proof of Theorem 3.2.1, we emphasize that once we compute a term and add it to the accumulator, we forget the term; this is how we can reduce the amount of memory needed. In total, we require

$$O(\log k) + O(\log b') + O(\log(kb')) = O(\log k + \log b')$$

memory. However, since $b' \leq 2p + \log k$, the total memory reduces to $O(\log k + \log p) = O(\log \lambda)$.

Now, since the sum is at most $k(b' + 1) \leq 2kp$, we can prove by induction that $C_{b'} \leq 2 \cdot 2kp$. Therefore $C_{b'}$ only requires space $O(\log(kp)) = O(\log \lambda)$ as well. It is also worth emphasizing that we only store $C_{b'}$, not $A_{b'}$, in memory. From $C_{b'}$ we can obtain the b'-th bit; this is the least significant bit of $C_{b'}$. And we only need to remember $\lfloor C_{b'}/2 \rfloor$ for the next iteration $b = b' + 1$.

Therefore, each iteration takes space $O(\log \lambda)$, which we can reuse. The amount of memory required to remember the current iteration is $O(\log b) = O(\log \lambda)$, so in total, we can compute inner product of natural number vectors in log-space.

We now wish to move to the real version. But first, we will show how to compute inner product of integer vectors (with possibly negative entries) in log-space.

We assume each element u_i, v_i also has a sign bit $u_{i,+}, v_{i,+}$ indicating its sign. The idea is to split the inner product into positive and negative contributions. During computation of $C_{b'}$ above, when the algorithm wants to compute $u_{i,j}v_{i,b'-j}$, it first checks if the sign bits are correct: for the positive contribution, we want $u_{i,+} = v_{i,+}$, while for the negative contribution, we want $u_{i,+} \neq v_{i,+}$. If the sign bits are not appropriate, we skip the product; this corresponds to skipping the component because it is considered in the other contribution.

Using this, the algorithm can compute the positive and negative contributions in log-space. It then simply compares which one is larger to determine the sign of the output, and computes the difference to determine the magnitude of the output. Both of these can be done in log-space as well.

The real version is now straightforward. The accuracy needed has already been settled by the proof of Theorem 3.1.1. By approximating the input numbers to the required number of digits and then scaling by a multiple of 2^p (where $p = 2\log k + 2n + 4$ is the parameter for the input accuracy), we may assume that all entries are integers. Then using inner product for integers above, we obtain a log-space algorithm to compute the output to the correct approximation. Finally, since $p = O(\log k + n) = O(\lambda)$, it follows our algorithm takes space $O(\log(k + p)) = O(\log(k + n)) = O(\log \lambda)$. $\square$

(b) This is simply a combination of the proof in (a) and the proof of Theorem 3.2.1. We use the same idea of using accuracy from Theorem 3.2.1 and then scaling them so that we work with integers as in (a). We can compute inner product in $O(\log(k + n))$, and exponentiation-by-squaring requires $O(\log E)$ levels of recursion. Therefore the total memory required is $O(\log(k + n)\log E) = O(\log^2 \lambda)$.

As a remark, this statement even holds without the condition $\|M\| \leq 1$, as long as the magnitude plays a part in the complexity parameter: if the elements in M satisfy $|M_{i,j}| \leq 2^c$, then the complexity parameter λ should be $k + n + E + c$. $\square$

(c) The proof in (b) holds, but now the complexity parameter is $k + n + \log E$ instead of $k + n + E$ since E is written in binary instead of unary. Therefore, a memory cost of $O(\log(k+n)\log E)$ is no longer $\log^2$-space. Instead, we now optimize for time. Instead of recomputing entries as needed, whenever we need to compute a matrix such as $M^{e/2}$ for M^e, we remember the entire matrix. Since each entry is an inner product and hence can be computed in poly-time, the whole matrix can be computed in poly-time. Since there are $O(\log E)$ levels of recursion, which is polynomial in the parameter λ, the total time to compute the matrix power is also polynomial. $\square$

4.2 Linear partial differential equations

Partial differential equations can also be approximated using difference schemes by dividing both temporal and spatial dimensions into time steps. The main difference is that, instead of having $\vec{u}$ to represent only the last few time steps (thus leading to a matrix of small size), we use $\vec{u}$ to represent the approximate value of the function at every spatial point. For example, given the differential equation

$$f_t(x,t) = f_x(x,t)$$

defined on $[0,1] \times [0,\infty)$, we can divide the spatial dimension x into time steps of size k and the temporal dimension t into time steps of size h to obtain

$$f(x,t+h) \approx \frac{h}{k} f(x+k,t) + \left(1 - \frac{h}{k}\right) f(x,t).$$

Since our vector $\vec{u}_t$ is only indexed by t, we need to include all the spatial points in order to hold all the necessary information; in other words,

$$\vec{u}_t = (f(0,t), f(k,t), f(2k,t), \ldots, f(1,t)).$$

This means the dimensions of $\vec{u}$ and the matrix M are $1/h$. To improve the approximation, we want to take a smaller h; this leads to the exponential-size dimensions, hence why we have been studying of exponential-size problems.

However, the matrices produced by this kind of difference schemes are not just random matrices; they have a specific structure. In general, the matrices are very sparse, and nonzero elements are placed close to the main diagonal, to capture the intuition that far-away points cannot influence each other.

For linear partial differential equations of one spatial dimension with periodic boundary condition and constant coefficients, such as the above example of $f_t(x,t) = f_x(x,t)$, it can be shown that the matrix corresponding to it is a circulant matrix with low band-width. In more general situations with more than one spatial dimension (but still with periodic boundary condition and constant coefficients), the matrix is in the form

$$M := \sum_{|i_1|,\ldots,|i_v| \le d} B_{i_1,\ldots,i_v} \otimes C_{N_1}^{i_1} \otimes \ldots \otimes C_{N_v}^{i_v}$$

where $\otimes$ indicates the Kronecker product, $B_{i_1,\ldots,i_v}$ are $b \times b$ matrices that pairwise commute, C_N is an $N \times N$ circulant matrix with 1's on the off-diagonal below the main diagonal, $N_1, \ldots, N_v$ are dimensions of the "components" of the matrix, and d is a small constant indicating the bandwidth.

Any element of M^E can be represented as an element of the block

$$\mathcal{B} \otimes C_{N_1}^{I_1} \otimes \ldots \otimes C_{N_v}^{I_v}$$

where $\mathcal{B}$ is some $b \times b$ matrix obtained as a product of powers of the $B_{i_1,\ldots,i_v}$ matrices, and $I_1, \ldots, I_v$ are some integers. Therefore, we wish to determine $\mathcal{B}$ for a specific choice of $I_1, \ldots, I_v$.

Treating $C_{N_1}, \ldots, C_{N_v}$ as indeterminates, we obtain a Laurent polynomial in v variables and degree d

$$p_M = \sum_{|i_1|,\ldots,|i_v| \le d} B_{i_1,\ldots,i_v} x_1^{i_1} \ldots x_v^{i_v}.$$

Raising this to the E-th power and seeking for the coefficient of $x_1^{I_1} \ldots x_v^{I_v}$ gives us the desired matrix $\mathcal{B}$. This motivates why we consider the function `POLYNOMIAL-POWER` treated in Section 3.3, as computing powers of this polynomial allows us to compute powers of our matrix.

As a remark, note that there are some differences between p_M and the polynomials we consider in `POLYNOMIAL-POWER`, but these are of no particular importance.

First, p_M is a Laurent polynomial, possibly having negative powers. But this is fine; if the degree of p_M is d, then multiply p_M by $x_1^d \dots x_v^d$ and seek for the coefficient of $x_1^{I_1+dE} \dots x_v^{I_v+dE}$.

Second, the coefficients of p_M are real matrices instead of just real numbers. However, each element of $(p_M)^E$ is a polynomial, so our result still holds.

Third, the same block $\mathcal{B} \otimes C_{N_1}^{I_1} \otimes \dots \otimes C_{N_v}^{I_v}$ of M^E can be obtained from multiple choices of $I_1, \dots, I_v$. However, since `POLYNOMIAL-POWER` is in $\#\mathsf{P}$, we can simply include an additional witness: choose the indices $I_1, \dots, I_v$ as part of the witness, check if it contributes to the block we seek (and return 0 otherwise), and if yes, we proceed to `POLYNOMIAL-POWER`. Therefore computing the block is in $\#\mathsf{P}$.

Chapter 5. Conclusion

We discussed the definition of some linear algebra problems when they involve exponential-size inputs and real number entries. We took an approach used by Ko [3], where computing a real number means computing arbitrarily accurate approximations of it, and getting a real number as an input to a real function means getting a sufficiently accurate approximation of it. We also developed a way to formalize exponential-size inputs by instead treating a sequence of functions whose arities grow along the sequence. In addition, we defined real complexity classes by generalizing existing classical complexity classes; in particular, we defined $\mathbb{R}\#\mathsf{P}$.

We also investigated the computational complexity of the linear algebra problems. In the case of exponential-size inner product, the complexity is in $\mathbb{R}\#\mathsf{P}$, and this is in a sense optimal by showing that a $\#\mathsf{P}$-complete problem can be reduced to this. For exponential-size matrix powering, the complexity is in $\mathbb{R}\mathsf{PSPACE}$, and this is also optimal in the sense that a PSPACE-complete problem is reducible to exponential-size matrix powering. We also looked at several restrictions of the matrix powering problem, turning it into a polynomial powering problem, which we showed to be in $\mathbb{R}\#\mathsf{P}$. A particular special case of the polynomial powering problem is in polynomial time with a novel approach. We also showed an application of these results, namely the application of polynomial powering to approximate solutions to differential equations using difference schemes. These theorems contribute to the growing field of real complexity theory by providing new results in relation to exponential-size problems, previously poorly researched.

Some of our results are not tight yet. For example, polynomial powering does not yet have a lower bound. The input to this problem is no longer exponential-size, since we fix the degree and arity of the polynomial; therefore, it seems reasonable that we can have a polynomial-time algorithm. Does such algorithm exist or is our $\mathbb{R}\#\mathsf{P}$ result optimal? And what about the restriction discussed in Section 3.4; does this poly-time result hold for a larger class of polynomials?

There are also plenty of other linear algebra problems, such as matrix determinant and inverse, matrix decomposition, and more, which have not been discussed in this paper. With a matrix as an input, it is reasonable to consider the exponential-size versions of them just like we discussed here; while there have been results on the "small"-sized versions where the entire matrix can be read off, the exponential-size versions are new. Their complexity can be researched and discussed in the future.

Bibliography

[1] B. M. Bush. “The Perils of Floating Point.” http://www.lahey.com/float.htm. 1996.

[2] W. Hoeffding. “Probability Inequalities for Sums of Bounded Random Variables.” *Journal of the American Statistical Association* 58.301, p.13–30. 1963.

[3] K. Ko. *Complexity Theory of Real Functions*. Birkhäuser Basel. 1991.

[4] I. Koswara, S. Selivanova, M. Ziegler. “Computational Complexity of Real Powering and Improved Solving Linear Differential Equations.” *Computer Science – Theory and Applications (CSR 2019)*. 2019.

[5] F. Le Gall. “Powers of Tensors and Fast Matrix Multiplication.” *39th International Symposium on Symbolic and Algebraic Computation (ISSAC 2014)*. arXiv: 1401.7714. 2014.

[6] N. T. Müller. “The The iRRAM: Exact Arithmetic in C++.” *Computability and Complexity in Analysis (CCA 2000)*, p.222–252. 2000.

[7] G. Nemes. “On the coefficients of the asymptotic expansion of $n!$.” arXiv: 1003.2907. 2010.

[8] W. J. Savitch. “Relationships between nondeterministic and deterministic tape complexities.” *Journal of Computer and System Sciences* 4 (2), p.177–192. 1970.

[9] V. Strassen. “Gaussian elimination is not optimal.” *Numerische Mathematik* 13, p.354–356. 1969.

[10] L. Valiant. “The complexity of computing the permanent.” *Theoretical Computer Science* 8 (2), p. 189–201. 1979.