# (Dense Structured) Matrix Vector Multiplication

Atri Rudra[1]

January 27, 2023

[1] Department of Computer Science and Engineering, University at Buffalo, SUNY. Work supported by NSF CCF-1763481.

# Foreword

These are notes accompanying a short lecture series at the Open Ph.D. lecture series at the University of Warsaw. I thank them for their hospitality.

These notes are **not** meant to be a comprehensive survey of the vast literature on dense structured matrix-vector multiplication. Rather, these notes present a biased view of the literaure based on my own forays into this wonderful subject while working on the paper [1].

Thanks to Tri Dao, Chris De Sa, Rohan Puttagunta, Anna Thomas and in particular, Albert Gu and Chris Ré for many wonderful discussions related to matrix vector multiplication. Thanks also to Mahmoud Abo Khamis and Hung Ngo for work on a previous work that was my direct motivation to work on matrix-vector multiplication. Thanks to Matt Eichhorn for comments on the draft. Thanks also to all the following awesome folks for comments on the notes: Grzegorz Bokota, Marco Gaboardi, Alex Liu, Barbara Poszewiecka, Dan Suciu.

Finally some warning if you are planning to read through the notes:

- Chapter 3 is incomplete.
- **Bibliographic notes are completely missing.** We intend to put them in by the end of summer 2021.
- These notes are not polished. If you find any typos/bugs, please send them to atri@buffalo.edu.

My work on these notes was supported by NSF grant CCF-1763481.

©Atri Rudra, 2020.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.or
or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# What is matrix vector multiplication and why should you care?

## 1.1 What is matrix vector multiplication?

In these notes we will be working with matrices and vectors. Simply put, matrices are two dimensional arrays and vectors are one dimensional arrays (or the "usual" notion of arrays). We will be using notation that is consistent with array notation. In particular, a matrix $\mathbf{A}$ with $m$ rows and $n$ columns (also denoted as an $m \times n$ matrix) will in code be defined as `int [][] A = new int[m][n]` (assuming the matrix stores integers). So for example the following is a $3 \times 3$ matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & -3 \\ 2 & 9 & 0 \\ 6 & -1 & -2 \end{pmatrix}. \tag{1.1}$$

Also a vector $\mathbf{x}$ of size $n$ in code will be declared as `int [] x = new int[n]` (again assuming the vector contains integers). For example the following is a vector of length 3

$$\mathbf{z} = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}. \tag{1.2}$$

To be consistent with the array notations, we will denote the entry in $\mathbf{A}$ corresponding to the $i$th row and $j$th column as $\mathbf{A}[i, j]$ (or `A[i][j]`). Similarly, the $i$th entry in the vector $\mathbf{x}$ will be denoted as $\mathbf{x}[i]$ (or `x[i]`). We will follow the array convention assume that the indices $i$ and $j$ start at 0.

We are now ready to define the problem that we will study throughout the course of these notes:

- **Input**: An $m \times n$ matrix $\mathbf{A}$ and a vector $\mathbf{x}$ of length $n$
- **Output**: Their product, which is denoted by

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x},$$

where $\mathbf{y}$ is also a vector of length $n$ and its $i$th entry for $0 \le i < m$ is defined as follows:

$$\mathbf{y}[i] = \sum_{j=0}^{n-1} \mathbf{A}[i,j] \cdot \mathbf{x}[j].$$

For example, here is the worked out example for $\mathbf{A}$ and $\mathbf{z}$ defined in (1.1) and (1.2):

$$\begin{pmatrix} 1 & 2 & -3 \\ 2 & 9 & 0 \\ 6 & -1 & -2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \times 2 + 2 \times 3 + (-3) \times (-1) \\ 2 \times 2 + 9 \times 3 + 0 \times (-1) \\ 6 \times 2 + (-1) \times 3 + (-2) \times (-1) \end{pmatrix} = \begin{pmatrix} 11 \\ 31 \\ 11 \end{pmatrix}.$$

So far we have looked at matrices that are defined over integers. A natural question is whether there is something special about integers. As it turns out, the answer is no.

### There is nothing special about integers

It turns out that in these notes we will consider the matrix vector multiplication problem over any *field*. Informally a field is a set of numbers that is closed under addition, subtraction, multiplication and division. More formally,

**Definition 1.1.1.** A field $\mathbb{F}$ is given by a triple $(S, +, \cdot)$, where $S$ is the set of elements containing special elements 0 and 1 and $+, \cdot$ are functions $S \times S \to S$ with the following properties:

- Closure: For every $a, b \in S$, we have both $a + b \in S$ and $a \cdot b \in S$.
- Associativity: $+$ and $\cdot$ are associative, that is, for every $a, b, c \in S$, $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- Commutativity: $+$ and $\cdot$ are commutative, that is, for every $a, b \in S$, $a + b = b + a$ and $a \cdot b = b \cdot a$.
- Distributivity: $\cdot$ distributes over $+$, that is for every $a, b, c \in S$, $a \cdot (b + c) = a \cdot b + a \cdot c$.
- Identities: For every $a \in S$, $a + 0 = a$ and $a \cdot 1 = a$.
- Inverses: For every $a \in S$, there exists its unique *additive inverse* $-a$ such that $a + (-a) = 0$. Also for every $a \in S \setminus \{0\}$, there exists its unique *multiplicative inverse* $a^{-1}$ such that $a \cdot a^{-1} = 1$.

We note that our definition of the matrix vector multiplication problem is equally valid when elements in a matrix (and vector) come from a field as long as we associate the addition operator with the field operator $+$ and the multiplication operator with the $\cdot$ operator over the field. With the usual semantics for $+$ and $\cdot$, $\mathbb{R}$ (set of real number) is a field but $\mathbb{Z}$ (set of integers) is not a field as division of two integers can give rise to a rational number (the set of rational numbers itself is a field though– see Exercise 1.1).

**Definition 1.1.2.** Given a field $\mathbb{F}$, we will denote the space of all vector of length $n$ with elements from $\mathbb{F}$ as $\mathbb{F}^n$ and the space of all $m \times n$ matrices with elements from $\mathbb{F}$ as $\mathbb{F}^{m \times n}$.

The reader might have noticed that we talked about matrices over integers but integers do not form a field. It turns out that pretty much everything we cover in these notes also works for a weaker structure called *rings* (rings are like fields except they do not have multiplicative inverses) and $\mathbb{Z}$ it turns out is a ring (see Exercise 1.2). It also turns out that we can have *finite* fields. For example, the smallest finite field (on two elements) is defined as $\mathbb{F}_2 = (\{0,1\}, \oplus, \wedge)$ (where $\oplus$ is the XOR of addition $\mod 2$ operation and $\wedge$ is the Boolean AND operator). We will talk about more general finite fields in bit more detail soon.

As we will see shortly, the matrix-vector multiplication problem is a very fundamental computational task and the natural question is how efficiently one can perform this operation. In particular, here is the relevant question in its full generality:

**Question 1.1.1.** *Given a matrix* $\mathbf{A} \in \mathbb{F}^{m\times n}$ *and a vector* $\mathbf{x} \in \mathbb{F}^n$*, compute*

$$\mathbf{y} = \mathbf{A}\cdot\mathbf{x},$$

*where for* $0 \le i < m$*:*

$$\mathbf{y}[i] = \sum_{j=0}^{n-1} \mathbf{A}[i,j]\cdot\mathbf{x}[j]$$

*with as few (addition and multiplication) operations over* $\mathbb{F}$.

Note that we mention the number of additions and multiplications over $\mathbb{F}$ as our measure of complexity and not the time taken. We will revisit this aspect in the next chapter: for now it suffices to say that this choices makes it easier to talk about any field $\mathbb{F}$ uniformly.[1]

## 1.2 Complexity of matrix vector multiplication

After a moment's thought, one can see that we can can answer Question 1.1.1 for the worst-case scenario, at least with an upper bound. In particular, consider the obvious Algorithm 1.

**Algorithm 1** Naive Matrix Vector Multiplication Algorithm

```
INPUT: $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{A} \in \mathbb{F}^{m\times n}$
OUTPUT: $\mathbf{A}\cdot\mathbf{x}$

FOR $0 \le i < m$ DO
    $\mathbf{y}[i] \leftarrow 0$
    FOR $0 \le j < n$ DO
        $\mathbf{y}[i] \leftarrow \mathbf{y}[i] + \mathbf{A}[i,j]\cdot\mathbf{x}[j]$.
RETURN $\mathbf{y}$
```

One can easily verify that Algorithm 1 takes $O(mn)$ operations in the worst-case. Further, if the matrix $\mathbf{A}$ is arbitrary, one would need $\Omega(mn)$ time (see Exercise 1.3). Assuming that each operation for an field $\mathbb{F}$ can be done in $O(1)$ time, this implies that the worst-case complexity of matrix-vector multiplication is $\Theta(mn)$.

[1] E.g. this way we do not have to worry about precision issues while storing elements from infinite fields such as $\mathbb{R}$.

**So are we done?**

If we just cared about worst-case complexity, we would be done. However, since there is a fair bit of notes after this spot it is safe to assume that this is not we care about. It turns out that in a large number of practical applications, the matrix $\mathbf{A}$ is fixed (or more appropriately has some structure). Thus, when designing algorithms to compute $\mathbf{A}\cdot\mathbf{x}$ (for arbitrary $\mathbf{x}$), we can exploit the structure of $\mathbf{A}$ to obtain a complexity that is asymptotically better than $O(mn)$.

The basic insight is that in many applications, one needs to compute specific *linear functions*, which are defined as follows:

**Definition 1.2.1.** A function $f : \mathbb{F}^n \to \mathbb{F}^m$ is said to be *linear* if for every $a, b \in \mathbb{F}$ and $\mathbf{x}, \mathbf{y} \in \mathbb{F}^n$, we have

$$f(a\cdot\mathbf{x} + b\cdot\mathbf{y}) = a\cdot f(\mathbf{x}) + b\cdot f(\mathbf{y}).$$

It turns out that linear functions are *equivalent* to some matrix $\mathbf{A}$. In particular, we claim that a linear function $f : \mathbb{F}^n \to \mathbb{F}^m$ is uniquely determined by a matrix $\mathbf{A}_f \in \mathbb{F}^{m\times n}$ such that for every $\mathbf{x} \in \mathbb{F}^n$, $f(\mathbf{x}) = \mathbf{A}_f\cdot\mathbf{x}$. (See exercise 1.4.) Thus, evaluating a linear function $f$ at a point $\mathbf{x}$ is exactly the same as the matrix-vector multiplication $\mathbf{A}_f\cdot\mathbf{x}$. Next, we present two applications that crucially use linear functions.

## 1.3 Two case studies

### 1.3.1 Error-correcting codes

Consider the problem of communicating $n$ symbols from $\mathbb{F}$ over a noisy channel that can corrupt transmitted symbols. While studying various model of channels (which correspond to different ways in which transmitted symbols can get corrupted) is a fascinating subject, for the sake of not getting distracted, we will focus on the following noise model. We think of the channel as an adversary who can arbitrarily corrupt up to $\tau$ symbols.[2] A moment's though reveals that sending $n$ symbols over a channel that can corrupt even one symbol is impossible. Thus, a natural "work-around" is to instead send $m > n$ symbols over the channel so that even if $\tau$ symbols are corrupted during transmission, the receiver can still recover the original $n$ symbols.

We formalize the problem above as follows. We want to construct an *encoding* function[3]

$$E : \mathbb{F}^n \to \mathbb{F}^m,$$

and a *decoding* function

$$D : \mathbb{F}^m \to \mathbb{F}^n$$

such that the following holds for any $\mathbf{x} \in \mathbb{F}^n$ and $\mathbf{e} \in \mathbb{F}^m$ such that $\mathbf{e}$ has at most $\tau$ non-zero values:

$$D(E(\mathbf{x}) + \mathbf{e}) = \mathbf{x}.$$

Here is how we relate the above formal setting to the communication problem we talked about. A sender Alice wants to send $\mathbf{x} \in \mathbb{F}^n$ to Bob over a noisy channel that can corrupt up to $\tau$ transmitted symbols. In other words, the adversarial channel with the full knowledge $E(\mathbf{x})$ (and the encoding and decoding functions $E$ and $D$) computes an *error pattern* $\mathbf{e} \in \mathbb{F}^m$ with at most $\tau$ non-zero locations[4] and sends

[2] Corrupting a symbol $a \in \mathbb{F}$ means changing to some symbol in $F \setminus \{a\}$.

[3] The range of the encoding function $E$, i.e the set of vectors $\{E(\mathbf{x})|\mathbf{x} \in \mathbb{F}^n\}$ is called an *error-correcting code* or just a *code*.

[4] These are the locations where the adversary introduces errors.

$\mathbf{y} = E(\mathbf{x}) + \mathbf{e}$ to Bob. Bob then computes $D(\mathbf{y})$ to (hopefully) get back $\mathbf{x}$. A natural question is for a given $\tau$, how much *redundancy* (i.e. the quantity $m - n$) do we need so that in the above scenario Bob can always recover $\mathbf{x}$?

We begin with the simplest adversarial channel: where the channel can corrupt up to $\tau = 1$ error. And to begin with consider the simple problem of *error detection*: i.e. Bob on receiving $\mathbf{y}$ needs to decide if there is an $\mathbf{x}$ such that $\mathbf{y} = E(\mathbf{x})$ (i.e. is $\mathbf{e} = \mathbf{0}$?).[5] Consider this problem over $\mathbb{F} = \mathbb{F}_2$ (i.e. the symbols are now bits). Consider the following encoding function:

$$E_{\oplus}(x_0, \dots, x_{n-1}) = \left( x_0, \cdots, x_{n-1}, \sum_{i=0}^{n} x_i \right). \tag{1.3}$$

In other words, the encoding function $E_{\oplus}$ adds a *parity* at the end of the $n$ bits (so $m = n + 1$).[6] There exists a simple decoding function $D_{\oplus}$ that detects if at most one error occurred during transmission (see Exercise 1.5).

Let us now concentrate on $E_{\oplus}$: one can show this is a linear function. By Exercise 1.4, we know that there exists an equivalent $\mathbf{A}_{\oplus}$. For example, here is the matrix for $n = 3$:

$$\mathbf{A}_{\oplus} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Now note that by Exercise 1.3, we can compute $E_{\oplus}(\mathbf{x})$ with $O(n^2)$ operations (recall $m = n + 1$). However, it is also easy to see that from definition of $E_{\oplus}$ from (1.3), we can compute $E_{\oplus}(\mathbf{x})$ with $O(n)$ operations over $\mathbb{F}_2$ (see Exercise 1.6). Note that this is possible because the matrix $\mathbf{A}_{\oplus}$ is not an arbitrary matrix but has some structure: in particular, it has only $O(n)$ non-zero entries. (See Exercise 1.7 for the general observation on this front.) This leads to the following observation, which is perhaps the main insight from practice needed for these notes:

Applications in practice need matrices $\mathbf{A}$ that are not arbitrary, and one can use the structure of $\mathbf{A}$ to perform matrix-vector multiplication in $o(mn)$ operations.[a]

[a] We say that a function $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.

In particular, Exercise 1.7 implies that any matrix $\mathbf{A} \in \mathbb{F}^{m\times n}$ that is $o(mn)$ sparse also has an $o(mn)$ operation matrix-vector multiplication operation. It is worth noting that

To obtain an $o(mn)$ complexity, we have assume a succinct representation. If e.g. matrix $\mathbf{A}$ with at most $s$ non-zero elements is still represented as an $m \times n$ matrix, then there is no hope of beating the $\Omega(mn)$ complexity. For sparse matrices, we will assume any reasonable listing (e.g. list of triples $(i, j, \mathbf{A}[i, j])$ for all non-zero locations). We will come back to this issue in the next chapter.

As we progress in these notes, we will see more non-trivial conditions on the matrix $\mathbf{A}$, which leads to faster matrix-vector multiplication than the trivial algorithm. Next, we look at our next case study.

[5] Much of error correction on the Internet actually only performs error detection and the receiver asks the sender to re-send the information if an error is detected.

[6] A variant of the parity code is what is used on the Internet to perform error detection.

### 1.3.2 Deep learning

**WARNING:** We do not claim to have any non-trivial knowledge (deep or otherwise) of deep learning. Thus, we will only consider a very simplified model of neural networks and our treatment of neural networks should in no way be interpreted as being representative of the current state of deep learning.

We consider a toy version of neural networks in use today: we will consider the so called *single layer* neural network:

**Definition 1.3.1.** We define a *single layer neural network* with input $\mathbf{x} \in \mathbb{F}^n$ and output $\mathbf{y} \in \mathbb{F}^m$ where the output is related to input as follows:

$$\mathbf{y} = g\left(\mathbf{W} \cdot \mathbf{x}\right),$$

where $\mathbf{W} \in \mathbb{F}^{m \times n}$ and $g : \mathbb{F}^m \to \mathbb{F}^m$ is a non-linear function.

Some remarks are in order: (1) In practice neural networks are defined for $\mathbb{F} = \mathbb{R}$ or $\mathbb{F} = \mathbb{C}$ but we abstracted out the problem to a general field (because it matches better with our setup for matrix-vector multiplication); (2) One of the common examples of non-linear function $g : \mathbb{R}^m \to \mathbb{R}^m$ is applying to so called $\mathsf{ReLu}$ function to each entry.[7] (3) The entries in the matrix $\mathbf{W}$ are typically called the *weights* in the layer.

Neural networks have two tasks associated with it: the first is the task of learning the network. For the network defined in Definition 1.3.1, this implies learning the matrix $\mathbf{W}$ given a set of training data $(\mathbf{x}_0, \mathbf{y}_0), (\mathbf{x}_1, \mathbf{y}_1), \cdots$ where we $\mathbf{y}_i$ is supposed to be a noisy version of $g(\mathbf{x})$. The second task is that once we have learned $g$, we use it to *classify* new data points $\mathbf{x}$ by computing $g(\mathbf{x})$. In practice, we would like the second step to be as efficient as possible.[8] In particular, ideally we should be able to compute $g(\mathbf{x})$ with $O(m+n)$ operations. The computational bottleneck in computing $g(\mathbf{x})$ is computing $\mathbf{W} \cdot \mathbf{x}$. Further, it turns out (as well will see later in these notes) that the complexity of the first step of learning the network is closely related to the complexity of the corresponding matrix-vector multiplication problem.

### 1.3.3 The main motivating question

This leads to the following (not very precisely defined) problem, which will be one of our guiding force for a large part of these notes:

**Question 1.3.1.** *What are the "interesting" class of matrices $\mathbf{A} \in \mathbb{F}^{m \times n}$ for which one can compute $\mathbf{A} \cdot \mathbf{x}$ for arbitrary $\mathbf{x} \in \mathbb{F}^n$ in $\tilde{O}(m+n)$ operations.*[a]

[a] For the rest of the notes we will use $\tilde{O}\left(f(n)\right)$ to denote the family of functions $O\left(f(n) \cdot \log^{O(1)} f(n)\right)$.

We note that any matrix $\mathbf{A}$ that has $\tilde{O}(m+n)$ non-zero entries will satisfy the above property (via Exercise 1.7). However, it turns out that in many practical application (including in deep learning application above) such sparse matrices are not enough. In particular, we are more interested in answering Question 1.3.1 when the matrix $\mathbf{A}$ has $\Omega(mn)$ non-zero entries, i.e. we are interested in *dense* matrices.

We collect some of the guiding principles about what aspects of the matrix-vector multiplication problem for specific matrices $\mathbf{A}$ are useful for practice (when answering Question 1.3.1):

[7] More precisely, we have $\mathsf{ReLu}(x) = \max(0, x)$ for any $x \in \mathbb{R}$ and $g(\mathbf{z}) = (\mathsf{ReLu}(z_0), \cdots, \mathsf{ReLu}(z_{m-1}))$.

[8] Ideally, we would also like the first step to be efficient but typically the learning of the network can be done in an offline step so it can be (relatively) more inefficient.

1. Dense structured matrices $\mathbf{A}$ are very useful.

2. The problem is interesting both over finite fields (e.g. for error-correcting codes) as well as infinite fields (e.g. neural networks).

We will return to Question 1.3.1 in Chapter 2. For the rest of the chapter, we will collect some background information that will be useful to understand the rest of the notes. (Though we will get distracted by some shiny and cute results along the way!)

## 1.4 Some background stuff

In this section, we collect some background material that will be useful in understanding the notes. In general, we will assume that the reader is familiar with these concepts and/or is willing to accepted the stated facts without hankering for a proof.

### 1.4.1 Vector spaces

We are finally ready to define the notion of linear subspace.

**Definition 1.4.1** (Linear Subspace)**.** A *non-empty* subset $S \subseteq \mathbb{F}^n$ is a linear subspace if the following properties hold:

1. For every $\mathbf{x}, \mathbf{y} \in S$, $\mathbf{x} + \mathbf{y} \in S$, where the addition is vector addition over $\mathbb{F}$ (that is, do addition component wise over $\mathbb{F}$).

2. For every $a \in \mathbb{F}$ and $\mathbf{x} \in S$, $a \cdot \mathbf{x} \in S$, where the multiplication is over $\mathbb{F}$.

Here is a (trivial) example of a linear subspace of $\mathbb{R}^3$:

$$S_1 = \{(a,a,a) | a \in \mathbb{R}\}. \tag{1.4}$$

Note that for example $(1,1,1) + (3,3,3) = (4,4,4) \in S_1$ and $2 \cdot (3.5, 3.5, 3.5) = (7,7,7) \in S_1$ as required by the definition. Here is another somewhat less trivial example of a linear subspace over $\mathbb{F}_2^3$:

$$S_2 = \{(0,0,0),(1,0,1),(1,1,0),(0,1,1)\}. \tag{1.5}$$

Note that $(1,0,1) + (1,1,0) = (0,1,1) \in S_2$ and $0 \cdot (1,0,1) = (0,0,0) \in S_2$ as required. Also note that $S_2$ is not a linear subspace over any other field $\mathbb{F}$.

*Remark* 1.4.1. Note that the second property implies that $\mathbf{0}$ is contained in every linear subspace. Further for any subspace over $\mathbb{F}_2$, the second property is redundant: see Exercise 1.8.

Before we state some properties of linear subspaces, we state some relevant definitions.

**Definition 1.4.2** (Span)**.** Given a set $B = \{\mathbf{v}_1, \ldots, \mathbf{v}_\ell\}$. The *span* of $B$ is the set of vectors

$$\left\{ \sum_{i=1}^{\ell} a_i \cdot \mathbf{v}_i | a_i \in \mathbb{F} \text{ for every } i \in [\ell] \right\}.$$

**Definition 1.4.3** (Linear independence of vectors)**.** We say that $\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_k$ are *linearly independent* if for every $1 \le i \le k$ and for every $k-1$-tuple $(a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k) \in \mathbb{F}^{k-1}$,

$$\mathbf{v}_i \neq a_1\mathbf{v}_1 + \ldots + a_{i-1}\mathbf{v}_{i-1} + a_{i+1}\mathbf{v}_{i+1} + \ldots + a_k\mathbf{v}_k.$$

In other words, $\mathbf{v}_i$ is not in the span of the set $\{\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}, \mathbf{v}_{i+1}, \ldots, \mathbf{v}_n\}$.

For example the vectors $(1,0,1), (1,1,1) \in S_2$ are linearly independent.

**Definition 1.4.4** (Rank of a matrix)**.** The *rank* of matrix in $\mathbb{F}^{m\times n}$ is the maximum number of linearly independent rows (or columns). A matrix in $\mathbb{F}^{m\times n}$ with rank $\min(m,n)$ is said to have *full* rank.

One can define the row (column) rank of a matrix as the maximum number of linearly independent rows (columns). However, it is a well-known theorem that the row rank of a matrix is the same as its column rank. For example, the matrix below over $\mathbb{F}_2$ has full rank (see Exercise 1.9):

$$G_2 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{1.6}$$

Any linear subspace satisfies the following properties (the full proof can be found in any standard linear algebra textbook).

**Theorem 1.4.1.** *If $S \subseteq \mathbb{F}^m$ is a linear subspace then*

1. *If $|\mathbb{F}| = q$, then $|S| = q^k$ for some $k \ge 0$. The parameter $k$ is called the* dimension *of $S$.*

2. *There exists $\mathbf{v}_1, \ldots, \mathbf{v}_k \in S$ called* basis elements *(which need not be unique) such that every $\mathbf{x} \in S$ can be expressed as $\mathbf{x} = a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \ldots + a_k\mathbf{v}_k$ where $a_i \in \mathbb{F}$ for $1 \le i \le k$. In other words, there exists a full rank $m \times k$ matrix $\mathbf{G}$ (also known as a* generator matrix*) with entries from $\mathbb{F}$ such that every $\mathbf{x} \in S$, $\mathbf{x} = \mathbf{G} \cdot (a_1, a_2, \ldots a_k)^T$ where*

$$\mathbf{G} = \begin{pmatrix} \uparrow & \uparrow & \cdots & \uparrow \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_k \\ \downarrow & \downarrow & \cdots & \downarrow \end{pmatrix}.$$

3. *There exists a full rank $(m-k) \times m$ matrix $H$ (called a* parity check matrix*) such that for every $\mathbf{x} \in S$, $H\mathbf{x} = \mathbf{0}$.*

4. *$G$ and $H$ are* orthogonal*, that is, $G \cdot H^T = \mathbf{0}$.*

The above implies the following connection (the proof is left as Exercise 1.10):

**Proposition 1.4.2.** *A linear function $f : \mathbb{F}^n \to \mathbb{F}^m$ is uniquely identified with a linear subspace.*

### Linear codes

We will now see an application of linear subspaces to the coding problem.

**Definition 1.4.5** (Linear Codes)**.** A linear code defined by an *linear* encoding function $E : \mathbb{F}^n \to \mathbb{F}^m$. Equivalently (via Proposition 1.4.2) $\{E(\mathbf{x}) | \mathbf{x} \in \mathbb{F}^n\}$ is a linear subspace of $\mathbb{F}^m$.

Theorem 1.4.1 then implies the following nice properties of linear codes:

**Proposition 1.4.3.** *Any linear code with encoding function $E : \mathbb{F}^n \rightarrow \mathbb{F}^m$ has the following two algorithmic properties:*

1. *The encoding problem (i.e. given $\mathbf{x} \in \mathbb{F}^n$, compute $E(\mathbf{x})$) can be computed with $O(mn)$ operations.*
2. *The error detection problem (i.e. given $\mathbf{y} \in \mathbb{F}^m$, does there exist $\mathbf{x} \in \mathbb{F}^n$ such that $E(\mathbf{x}) = \mathbf{y}$) can be solved with $O((m-n)n)$ operations.*

Recall that in the coding problem, we can design the encoding function and since encoding has to be done before any codeword is transmitted over the channel, we have the following question:

**Question 1.4.1.** *Can we come up with a "good" linear code for which we can solve the encoding problem with $O(m)$ or at least $\tilde{O}(m)$ operations?*

We purposefully leave the notion of "good" undefined for now but we will come back to this questions shortly.

### 1.4.2 Back to fields

The subject of fields should be studied in its own devoted class. However, since we do not have the luxury of time, we make some further observations on fields (beyond what has already been said in Section 1.1).

#### Infinite fields

We have already seen that the real numbers form a field $\mathbb{R}$. We now briefly talk about the field of complex numbers $\mathbb{C}$. Given any integer $n \geq 1$, define

$$\omega_n = e^{2\pi\iota/n},$$

where we use $\iota$ for the imaginary number (i.e. $\iota^2 = -1$). The complex number $\omega_n$ is a *primitive* root of unity: i.e. $\omega_n^j \neq 1$ for any $j < n$. Further, the $n$ roots of unity (i.e. the roots of the equation $X^n = 1$) are given by $\omega_n^j$ for $0 \leq j < n$. Further, by definition, these $n$ numbers are distinct.

Now consider the following *discrete Fourier matrix*:

**Definition 1.4.6.** The $n \times n$ *discrete Fourier* matrix $\mathbf{F}_n$ defined as follows (for $0 \leq i, j < n$):

$$F_n[i,j] = \omega_n^{ij}.$$

Let us unroll the following matrix-vector multiplication: $\hat{\mathbf{x}} = \mathbf{F}_n\mathbf{x}$. In particular, for any $0 \leq i < n$:

$$\hat{\mathbf{x}}[i] = \sum_{j=0}^{n-1} \mathbf{x}[j] \cdot e^{2\pi\iota ji/n}.$$

In other words, $\hat{\mathbf{x}}$ is the *discrete Fourier transform* of $\mathbf{x}$. It turns out that the discrete Fourier transform is incredibly useful in practice (and is used in applications such as image compression). One of the most celebrated algorithmic results is that the Fourier transform can be computed with $O(n\log n)$ operations:

**Theorem 1.4.4.** *For any $\mathbf{x} \in \mathbb{C}^n$, one can compute $\mathbf{F}_n \cdot \mathbf{x}$ in $O(n\log n)$ operations.*

We will prove this result in Chapter 3.

### Finite fields

As the name suggests these are fields with a finite size set of elements. (We will overload notation and denote the size of a field by $|\mathbb{F}|$.) The following is a well known result.

**Theorem 1.4.5** (Size of Finite Fields)**.** *The size of any finite field is $p^s$ for prime $p$ and integer $s \geq 1$.*

One example of finite fields that we have already seen is the field of two elements $\{0,1\}$, which we will denote by $\mathbb{F}_2$ (we have seen this field in the context of binary linear codes). For $\mathbb{F}_2$, addition is the XOR operation, while multiplication is the AND operation. The additive inverse of an element in $\mathbb{F}_2$ is the number itself while the multiplicative inverse of 1 is 1 itself.

Let $p$ be a prime number. Then the integers modulo $p$ form a field, denoted by $\mathbb{F}_p$ (and also by $\mathbb{Z}_p$), where the addition and multiplication are carried out $\mod p$. For example, consider $\mathbb{F}_7$, where the elements are $\{0,1,2,3,4,5,6\}$. So we have $4+3 \mod 7 = 0$ and $4 \cdot 4 \mod 7 = 2$. Further, the additive inverse of 4 is 3 as $3+4 \mod 7 = 0$ and the multiplicative inverse of 4 is 2 as $4 \cdot 2 \mod 7 = 1$.

More formally, we prove the following result.

**Lemma 1.4.6.** *Let $p$ be a prime. Then $\mathbb{F}_p = (\{0,1,\dots,p-1\}, +_p, \cdot_p)$ is a field, where $+_p$ and $\cdot_p$ are addition and multiplication $\mod p$.*

*Proof.* The properties of associativity, commutativity, distributivity and identities hold for integers and hence, they hold for $\mathbb{F}_p$. The closure property follows since both the “addition" and “multiplication" are done $\mod p$, which implies that for any $a, b \in \{0,\dots,p-1\}$, $a +_p b, a \cdot_p b \in \{0,\dots,p-1\}$. Thus, to complete the proof, we need to prove the existence of unique additive and multiplicative inverses.

Fix an arbitrary $a \in \{0,\dots,p-1\}$. Then we claim that its additive inverse is $p-a \mod p$. It is easy to check that $a + p - a = 0 \mod p$. Next we argue that this is the unique additive inverse. To see this note that the sequence $a, a+1, a+2, \dots, a+p-1$ are $p$ consecutive numbers and thus, exactly one of them is a multiple of $p$, which happens for $b = p - a \mod p$, as desired.

Now fix an $a \in \{1,\dots,p-1\}$. Next we argue for the existence of a unique multiplicative universe $a^{-1}$. Consider the set of numbers $\{a \cdot_p b\}_{b \in \{1,\dots,p-1\}}$. We claim that all these numbers are unique. To see this, note that if this is not the case, then there exist $b_1 \neq b_2 \in \{0,1,\dots,p-1\}$ such that $a \cdot b_1 = a \cdot b_2 \mod p$, which in turn implies that $a \cdot (b_1 - b_2) = 0 \mod p$. Since $a$ and $b_1 - b_2$ are non-zero numbers, this implies that $p$ divides $a \cdot (b_1 - b_2)$. Further, since $a$ and $|b_1 - b_2|$ are both at most $p-1$, this implies that factors of $a$ and $(b_1 - b_2) \mod p$ when multiplied together results in $p$, which is a contradiction since $p$ is prime. Thus, this implies that there exists a unique element $b$ such that $a \cdot b = 1 \mod p$ and thus, $b$ is the required $a^{-1}$. □

One might think that there could be different fields with the same number of elements. However, this is not the case:

**Theorem 1.4.7.** *For every prime power $q$ there is a* unique *finite field with $q$ elements (up to isomorphism[9]).*

Thus, we are justified in just using $\mathbb{F}_q$ to denote a finite field on $q$ elements.

It turns out that one can extend the discrete Fourier matrix to any finite field $\mathbb{F}_q$. This basically needs the following well-known result:

[9] An isomorphism $\phi: S \to S'$ is a map (such that $\mathbb{F} = (S, +, \cdot)$ and $\mathbb{F}' = (S', \oplus, \circ)$ are fields) where for every $a_1, a_2 \in S$, we have $\phi(a_1 + a_2) = \phi(a_1) \oplus \phi(a_2)$ and $\phi(a_1 \cdot a_2) = \phi(a_1) \circ \phi(a_2)$.

**Lemma 1.4.8.** *Let $q$ be a prime power and n be an integer that does not divide $q-1$. Then there exists an element $\omega_n \in \mathbb{F}_q^*$ such that $\omega_n^n = 1$ and $\omega_n^j \neq 1$ for every $1 \leq j < n$.*

Given the above, one can still define the discrete Fourier matrix $\mathbf{F}_n$ as in Definition 1.4.6 (assuming $n$ divides $q-1$). Further, Theorem 1.4.4 can also be extended to these matrices.

**Vandermonde matrix**

We finish off this section by describing a matrix that we will see a few times in these notes. Consider the following matrix:

**Definition 1.4.7.** For any $n \geq 1$ and any field $\mathbb{F}$ with size at least $m$ consider $m$ distinct elements $a_0, \dots, a_{m-1} \in F$, consider the matrix (where $0 \leq i < m$ and $0 \leq j < n$)

$$\mathbf{V}_n^{(\mathbf{a})}[i,j] = a_i^j,$$

where $\mathbf{a} = (a_0, \dots, a_{n-1})$.

We now state some interesting facts about these matrices:

1. The discrete Fourier matrix is a special case of a Vandermonde matrix (Exercise 1.12).

2. The Vandermonde matrix has full rank (Exercise 1.13).

3. It turns out that one can compute $\mathbf{V}_n \cdot \mathbf{x}$ for any $\mathbf{x} \in \mathbb{F}^n$ can be computed with $O(n \log^2 n)$ operations (see Chapter 3).

Finally, the fact that the Vandermonde matrix has full rank (see Exercise 1.13) allows us to define another error correcting code.

**Definition 1.4.8.** Let $m \geq n \geq 1$ be integers and let $q \geq m$ be a prime power. Let $\alpha_0, \dots, \alpha_{m-1}$ be distinct values. Then the *Reed-Solomon* code with *evaluation points* $\alpha_0, \dots, \alpha_{m-1}$ is a linear code whose generator matrix is given by $\mathbf{V}_n^{(\alpha_0, \dots, \alpha_{m-1})}$.

Reed-Solomon codes have very nice properties and we will come back to them at the end of the chapter (where we will also explain why we use the term evaluation points in the definition above).

## 1.5 Our tool of choice: polynomials

Polynomials will play a very integral part of the technical parts of these notes and in this section, we quickly review them and collect some interesting results about them.

### 1.5.1 Polynomials and vectors

We begin with the formal definition of a (univariate) polynomial.

**Definition 1.5.1.** A function $F(X) = \sum_{i=0}^{\infty} f_i X^i$, $f_i \in \mathbb{F}$ is called a polynomial.

For our purposes, we will only consider the finite case; that is, $F(X) = \sum_{i=0}^{d} f_i X^i$ for some integer $d > 0$, with coefficients $f_i \in \mathbb{F}$, and $f_d \neq 0$. For example, $2X^3 + X^2 + 5X + 6$ is a polynomial over $\mathbb{F}_7$ (as well as $\mathbb{R}$ and $\mathbb{C}$).

Next, we define some useful notions related to polynomials. We begin with the notion of degree of a polynomial.

**Definition 1.5.2.** For $F(X) = \sum_{i=0}^{d} f_i X^i$ ($f_d \neq 0$), we call $d$ the *degree* of $F(X)$.[10] We denote the degree of the polynomial $F(X)$ by $\deg(F)$.

For example, $2X^3 + X^2 + 5X + 6$ has degree 3.

We now state an obvious equivalence between polynomials and vector, which will be useful for these notes:

> There is a bijection between $\mathbb{F}^n$ and polynomials of degree at most $n-1$ over $\mathbb{F}$. In particular, we will use the following map: vector $\mathbf{f} \in \mathbb{F}^n$ maps to the polynomial $P_{\mathbf{f}}(X) = \sum_{i=0}^{n-1} \mathbf{f}[i] \cdot X^i$. Taking this a step further, the following is a bijection between a matrix $\mathbf{M} \in \mathbb{F}^n$ and a "family" of polynomials given by $\left\{P_i^{\mathbf{M}}(X) = P_{\mathbf{M}[i,:]}(X) : 0 \le i < n\right\}$

We will see another useful bijection in a little bit.

In addition to these syntactic relationships, we will also use operations over polynomials to design algorithms for certain structured matrix-vector multiplication.

### 1.5.2 Operations on polynomials

Let $\mathbb{F}[X]$ be the set of polynomials over $\mathbb{F}$, that is, with coefficients from $\mathbb{F}$. Let $F(X), G(X) \in \mathbb{F}[X]$ be polynomials. Then $\mathbb{F}[X]$ has the following natural operations defined on it:

**Addition:**

$$F(X) + G(X) = \sum_{i=0}^{\max(\deg(F),\deg(G))} (f_i + g_i) X^i,$$

where the addition on the coefficients is done over $\mathbb{F}_q$. For example, over $\mathbb{F}_2$, $X + (1+X) = X \cdot (1+1) + 1 \cdot (0+1)1 = 1$ (recall that over $\mathbb{F}_2$, $1+1=0$).[11] Note that since for every $a \in \mathbb{F}$, we also have $-a \in \mathbb{F}$, this also defined the subtraction operation.

**Multiplication:**

$$F(X) \cdot G(X) = \sum_{i=0}^{\deg(F)+\deg(G)} \left( \sum_{j=0}^{\min(i,\deg(F))} p_j \cdot q_{i-j} \right) X^i,$$

where all the operations on the coefficients are over $\mathbb{F}_q$. For example, over $\mathbb{F}_2$, $X(1+X) = X + X^2$; $(1+X)^2 = 1 + 2X + X^2 = 1 + X^2$, where the latter equality follows since $2 \equiv 0 \bmod 2$.

**Division:** We will state the division operation as generating a quotient and a reminder. In other words, we have

$$F(X) = Q(X) \cdot G(X) + R(X),$$

[10] We define the degree of $F(X) = 0$ to be 0.

[11] This will be a good time to remember that operations over a finite field are much different from operations over $\mathbb{R}$. For example, over $\mathbb{R}$, $X + (X+1) = 2X + 1$.

where $\deg(R) < \deg(G)$ and $Q(X)$ and $R(X)$ are unique. For example over $\mathbb{F}_7$ with $F(X) = 2X^3 + X^2 + 5X + 6$ and $G(X) = X + 1$, we have $Q(X) = 2X^2 + 6X + 6$ and $R(X) = 0$– i.e. $X + 1$ divides $2X^3 + X^2 + 5X + 6$.

**Evaluation:** Given a constant $\alpha \in \mathbb{F}$, define the evaluation of $F$ at $\alpha$ to be

$$F(\alpha) = \sum_{i=0}^{\deg(F)} f_i \cdot \alpha^i.$$

For example over $\mathbb{F}_7$, we have for $F = 2X^3 + X^2 + 5X + 6$, $F(2) = 2 \cdot 2^3 + 2^2 + 5 \cdot 2 + 6 = 2 + 4 + 3 + 6 = 1$.

### More on polynomial evaluation

We now make a simple observation. Let $f(X) = \sum_{i=0}^{n-1} f_i \cdot X^i$ and $\mathbf{f} = (f_0, \dots, f_{n-1})$. Then for any $\alpha_0, \dots, \alpha_{m-1} \in \mathbb{F}$, we have (see Exercise 1.14):

$$\begin{pmatrix} f(\alpha_0) \\ \vdots \\ f(\alpha_{m-1}) \end{pmatrix} = \mathbf{V}_n^{(\alpha_0,\dots,\alpha_{m-1})} \cdot \begin{pmatrix} f_0 \\ \vdots \\ f_{n-1} \end{pmatrix}. \tag{1.7}$$

Next, we make use of (1.7) to make two further observations. First we note that this implies the following alternate definition of Reed-Solomon codes (see Definition 1.4.8):

**Definition 1.5.3** (Polynomial view of Reed-Solomon Codes)**.** Let $m \geq n \geq 1$ be integers and let $q \geq m$ be a prime power. Let $\alpha_0, \dots, \alpha_{m-1}$ be distinct elements of $\mathbb{F}_q$. Then the *Reed-Solomon* code with *evaluation points* $\alpha_0, \dots, \alpha_{m-1}$ is defined via the following encoding function $E_{\text{RS}}$. Given a "message" $\mathbf{f} \in \mathbb{F}_q^n$:

$$E_{\text{RS},(\alpha_0,\dots,\alpha_{m-1})}(\mathbf{f}) = (P_{\mathbf{f}}(\alpha_0), \dots, P_{\mathbf{f}}(\alpha_{m-1})).$$

Note that the above definition justifies the use of the term evaluation points for $\alpha_0, \dots, \alpha_{m-1}$ in Definition 1.4.8.

Second we note that (1.7) along with the fact that Vandermonde matrices are full rank (Exercise 1.13) implies a bijection between polynomials of degree at most $n-1$ and evaluating such polynomials over $n$ distinct points in the field. The following alternate representation of vectors and matrices.

Let $\alpha_0, \dots, \alpha_{n-1} \in F$ be distinct points in the field. Then there exists a bijection between $\mathbb{F}^n$ and evaluations of polynomials of degree at most $n-1$ on $\alpha = (\alpha_0, \cdots, \alpha_{n-1})$. In particular, we can map any $\mathbf{z} \in \mathbb{F}^n$ to $\left(P_{\mathbf{z},\alpha}(\alpha_0), \dots, P_{\mathbf{z},\alpha}(\alpha_{n-1})\right)$, where for any $0 \leq j < n$, we have $P_{\mathbf{z},\alpha}(\alpha_j) = \mathbf{z}[j]$.

In particular, the above implies the following *polynomial transform* view of matrices. Specifically, for any $n \times n$ matrix $\mathbf{A}$, we have a family of $n$ polynomials $A_0(X), \dots, A_{n-1}(X)$, where for every $0 \leq i, j < n$, we have $P_i(\alpha_j) = \mathbf{A}[i, j]$.

Note that the Vandermonde matrix corresponds to the case when the polynomial $A_i(X) = X^i$.

### Finite field representation

Next, we define the notion of a root of a polynomial.

**Definition 1.5.4.** $\alpha \in \mathbb{F}$ is a root of a polynomial $F(X)$ if $F(\alpha) = 0$.

For instance, 1 is a root of $1 + X^2$ over $\mathbb{F}_2$ (but that $1 + X^2$ does not have any roots over $\mathbb{R}$). We will also need the notion of a special class of polynomials, which are like prime numbers for polynomials.

**Definition 1.5.5.** A polynomial $F(X)$ is irreducible if for every $G_1(X), G_2(X)$ such that $F(X) = G_1(X)G_2(X)$, we have $\min(\deg(G_1), \deg(G_2)) = 0$

In these notes, we will almost exclusively focus on irreducible polynomials over finite fields. For example, $1 + X^2$ is not irreducible over $\mathbb{F}_2$, as $(1 + X)(1 + X) = 1 + X^2$. However, $1 + X + X^2$ is irreducible, since its non-trivial factors have to be from the linear terms $X$ or $X + 1$. However, it is easy to check that neither is a factor of $1 + X + X^2$. (In fact, one can show that $1 + X + X^2$ is the only irreducible polynomial of degree 2 over $\mathbb{F}_2$– see Exercise 1.15.) A word of caution: if a polynomial $E(X) \in \mathbb{F}_q[X]$ does not have any root in $\mathbb{F}_q$, it does *not* mean that $E(X)$ is irreducible. For example consider the polynomial $(1 + X + X^2)^2$ over $\mathbb{F}_2$– it does not have any root in $\mathbb{F}_2$ but it obviously is not irreducible.

Just as the set of integers modulo a prime is a field, so is the set of polynomials modulo an irreducible polynomial:

**Theorem 1.5.1.** *Let $E(X)$ be an irreducible polynomial with degree $\geq 2$ over $\mathbb{F}_p$, $p$ prime. Then the set of polynomials in $\mathbb{F}_p[X]$ modulo $E(X)$, denoted by $\mathbb{F}_p[X]/E(X)$, is a field.*

The proof of the theorem above is similar to the proof of Lemma 1.4.6, so we only sketch the proof here. In particular, we will explicitly state the basic tenets of $\mathbb{F}_p[X]/E(X)$.

- Elements are polynomials in $\mathbb{F}_p[X]$ of degree at most $s-1$. Note that there are $p^s$ such polynomials.
- Addition: $(F(X)+G(X)) \mod E(X) = F(X) \mod E(X) + G(X) \mod E(X) = F(X) + G(X)$. (Since $F(X)$ and $G(X)$ are of degree at most $s - 1$, addition modulo $E(X)$ is just plain simple polynomial addition.)
- Multiplication: $(F(X) \cdot G(X)) \mod E(X)$ is the unique polynomial $R(X)$ with degree at most $s - 1$ such that for some $A(X)$, $R(X) + A(X)E(X) = F(X) \cdot G(X)$
- The additive identity is the zero polynomial, and the additive inverse of any element $F(X)$ is $-F(X)$.
- The multiplicative identity is the constant polynomial 1. It can be shown that for every element $F(X)$, there exists a unique multiplicative inverse $(F(X))^{-1}$.

For example, for $p = 2$ and $E(X) = 1 + X + X^2$, $\mathbb{F}_2[X]/(1 + X + X^2)$ has as its elements $\{0, 1, X, 1 + X\}$. The additive inverse of any element in $\mathbb{F}_2[X]/(1 + X + X^2)$ is the element itself while the multiplicative inverses of $1, X$ and $1 + X$ are $1, 1 + X$ and $X$ respectively.

A natural question to ask is if irreducible polynomials exist. Indeed, they do for every degree:

**Theorem 1.5.2.** *For all $s \geq 2$ and $\mathbb{F}_p$, there exists an irreducible polynomial of degree $s$ over $\mathbb{F}_p$. In fact, the number of such irreducible polynomials is $\Theta\left(\frac{p^s}{s}\right)$.*[12]

Note that the above and Theorems 1.4.5 and 1.5.1 imply that we now know how any finite field can be represented.

[12] The result is true even for general finite fields $\mathbb{F}_q$ and not just prime fields but we stated the version over prime fields for simplicity.

### 1.5.3 Diversions: Why polynomials are da bomb

We conclude this section by presenting two interesting things about polynomials: one that holds for all polynomials and one for a specific family of polynomials.

#### The degree mantra

We will prove the following simple result on polynomials:

**Proposition 1.5.3** ("Degree Mantra"). *A nonzero polynomial $f(X)$ over $\mathbb{F}$ of degree $t$ has at most $t$ roots in $\mathbb{F}$.*

*Proof.* We will prove the theorem by induction on $t$. If $t = 0$, we are done. Now, consider $f(X)$ of degree $t > 0$. Let $\alpha \in \mathbb{F}$ be a root such that $f(\alpha) = 0$. If no such root $\alpha$ exists, we are done. If there is a root $\alpha$, then we can write

$$f(X) = (X - \alpha)g(X)$$

where $\deg(g) = \deg(f) - 1$ (i.e. $X - \alpha$ divides $f(X)$). Note that $g(X)$ is non-zero since $f(X)$ is non-zero. This is because by the fundamental rule of division of polynomials:

$$f(X) = (X - \alpha)g(X) + R(X)$$

where $\deg(R) \le 0$ (as the degree cannot be negative this in turn implies that $\deg(R) = 0$) and since $f(\alpha) = 0$,

$$f(\alpha) = 0 + R(\alpha),$$

which implies that $R(\alpha) = 0$. Since $R(X)$ has degree zero (i.e. it is a constant polynomial), this implies that $R(X) \equiv 0$.

Finally, as $g(X)$ is non-zero and has degree $t - 1$, by induction, $g(X)$ has at most $t - 1$ roots, which implies that $f(X)$ has at most $t$ roots. □

It can be show easily that the degree mantra is tight (see Exercise 1.16). The reason the above result is interesting is that it implies some very nice properties of Reed-Solomon codes, which we discuss next.

#### Back to Reed-Solomon codes

It turns out that the degree mantra implies a very interesting result about Reed-Solomon codes. Before we do that we setup a notation: we will call a linear code with an encoding function $E : \mathbb{F}_q^n \to \mathbb{F}_q^m$ to be an $[m, n]_q$ code (for notational brevity).

**Lemma 1.5.4.** *Let $E_{\text{RS}}$ be the encoding function of an $[m, n]_q$ Reed-Solomon code. Define $\tau_{\text{RS}} = \frac{n-k}{2}$. Then there exists a decoding function such that for every message $\mathbf{x} \in \mathbb{F}_q^n$ and error pattern $\mathbf{e} \in \mathbb{F}^m$ (where the error pattern has at most $\tau_{\text{RS}}$ non-zeros) on input $E_{\text{RS}}(\mathbf{x}) + \mathbf{e}$ outputs $\mathbf{x}$. Further, no other code can "correct" strictly more than $\tau_{\text{RS}}$ many errors.*

The proof of the above result follows from a sequence of basic results in coding theory. Before we lay those out we would like to point out that the decoding function mentioned in Lemma 1.5.4 can be implemented in polynomially (and indeed near-linear) many operations over $\mathbb{F}_q$. However, this algorithm is non-trivial so in our proof we will provide with a decoding functions with exponential complexity.

We begin with the proof of the existence of the decoding function in Lemma 1.5.4. Towards this end, we quickly introduce some more basic definitions from coding theory.

**Definition 1.5.6.** The *Hamming distance* of two vector $\mathbf{y},\mathbf{z} \in \mathbb{F}^m$, denoted by $\Delta(\mathbf{y},\mathbf{z})$ is defined to be the number of locations $0 \le i < n$ where they differ (i.e. $\mathbf{y}[i] \neq \mathbf{z}[i]$). The *distance of a code* is the minimum Hamming distance between all pairs of codewords. The *Hamming weight* of a vector $\mathbf{x} \in \mathbb{F}^m$ is the number of non-zero locations in $\mathbf{x}$.

The following observation ties the distance of a code to its error correcting capabilities.

**Proposition 1.5.5.** *Let the $[m,n]_q$ code corresponding to the encoding function $E : \mathbb{F}_q^n \to \mathbb{F}_q^m$ have distance $d$. Define $\tau = \frac{d-1}{2}$. Then there exists a decoding function such that for every message $\mathbf{x} \in \mathbb{F}_q^n$ and error pattern $\mathbf{e} \in \mathbb{F}^m$ (where the error pattern has at most $\tau$ non-zeros) on input $E(\mathbf{x}) + \mathbf{e}$ outputs $\mathbf{x}$. Further, there does not exist any decoding function that can correct strictly more than $\tau$ errors.*

The above result follows from fairly simple arguments so we will relegate them to Exercise 1.17.

To prove Lemma 1.5.4, we will argue the following:

**Proposition 1.5.6.** *An $[m,n]_q$ Reed-Solomon code has distance $m - n + 1$.*

*Proof.* We first claim that for any linear code (defined by the encoding function $E$), the distance of the code is exactly the minimum Hamming weight of $E(\mathbf{x})$ over all non-zero $\mathbf{x} \in \mathbb{F}^n$ (see Exercise 1.18). We will prove the claim using this equivalent description.

Consider an non-zero "message" $\mathbf{f} \in \mathbb{F}^n$. Then note that $\deg(P_{\mathbf{f}}(X)) \le n - 1$. Since $\mathbf{f}$ (and hence $P_{\mathbf{f}}(X)$) is non-zero, the degree mantra implies that $P_{\mathbf{f}}(X)$ has at most $n-1$ zeroes. In other words $E_{\mathrm{RS},(\alpha_0,\alpha_{m-1})}(\mathbf{f})$ will have at least $m - (n-1) = m - n + 1$ non-zero locations. On the other hand, Exercise 1.16 shows that there does exists a non-zero $\mathbf{f}$, such that $E_{\mathrm{RS},(\alpha_0,\alpha_{m-1})}(\mathbf{f})$ has Hamming weight exactly $m - n + 1$. Thus, Exercise 1.18 shows that the Reed-Solomon code has distance $m - n + 1$. □

We note that Proposition 1.5.5 and 1.5.6 prove the positive part of Lemma 1.5.4. The negative part of Lemma 1.5.4 follows from the negative part of Proposition 1.5.5 and the additional fact that any code that maps $n$ symbols to $m$ has distance at most $n - m + 1$ (see Exercise 1.19).

### Chebyshev polynomials

We now change tracks and will talk about a specific family of code. In particular,

**Definition 1.5.7.** Define $T_0(X) = 1$, $T_1(X) = X$ and for any $\ell \ge 2$:

$$T_\ell(X) = 2X \cdot T_{\ell-1}(X) - T_{\ell-2}(X). \tag{1.8}$$

For example, $T_2(X) = 2X^2 - 1$ and $T_3(X) = 4X^3 - 3X$.

Chebyshev polynomials have many cool properties but for now we will state the following:

**Proposition 1.5.7.** *Let $\theta \in [-\pi,\pi]$. Then for any integer $\ell \ge 0$,*

$$T_\ell(\cos\theta) = \cos(\ell\theta).$$

*Proof.* We will prove this by induction on $\ell$. The claim for $\ell = 0,1$ follows from definition of $T_0(X)$ and $T_1(X)$. Now assume the claim holds for $\ell - 1$ and $\ell - 2$ for some $\ell \ge 2$. We will prove the claim for $\ell$.

To do this we will use the following well-known trigonometric inequality:

$$\cos(A \pm B) = \cos A \cos B \mp \sin A \sin B.$$

This implies that we have

$$\cos(\ell\theta) = \cos((\ell-1)\theta)\cos\theta + \sin((\ell-1)\theta)\sin\theta$$

and

$$\cos((\ell-2)\theta) = \cos((\ell-1)\theta)\cos\theta - \sin((\ell-1)\theta)\sin\theta.$$

Adding the two identities above, we get

$$\cos(\ell\theta) + \cos((\ell-2)\theta) = 2\cos\theta\cos((\ell-1)\theta).$$

The by the inductive hypothesis we have

$$\cos(\ell\theta) = 2\cos\theta\, T_{\ell-1}(\cos\theta) - T_{\ell-2}(\cos\theta),$$

which along with (1.8) completes the proof. □

Proposition 1.5.7 and the fact that $T_\ell(X)$ is a polynomial in $X$ implies the following:

**Corollary 1.5.8.** *Let $\theta \in [-\pi, \pi]$. Then for any integer $\ell \geq 0$, $\cos(\ell\theta)$ is a polynomial in $\cos\theta$.*

Later in the notes, we will come back to Chebyshev polynomials (and the more general *orthogonal polynomial* family).

## 1.6 Exercises

*Exercise* 1.1. Prove that the set of rationals (i.e. the set of reals of the form $\frac{a}{b}$, where both $a$ and $b \neq 0$ are integers), denoted by $\mathbb{Q}$, is a field.

*Exercise* 1.2. Prove that the set of integers $\mathbb{Z}$ with the usual notion of addition and multiplication forms a ring (i.e. $(\mathbb{Z}, +, \cdot)$ satisfies all properties of Definition 1.1.1 except the one on the existence of multiplicative inverse for non-zero integers).

*Exercise* 1.3. First argue that Algorithm 1 takes $O(mn)$ operations over $\mathbb{F}$.

Also argue that for arbitrary $\mathbf{A}$, in the worst-case *any* algorithm has to read all entries of $\mathbf{A}$. Thus, conclude that any algorithm that solves $\mathbf{Ax}$ for arbitrary $\mathbf{A}$ needs $\Omega(mn)$ time.[13]

*Exercise* 1.4. Show that a function $f : \mathbb{F}^n \to \mathbb{F}^m$ is linear if and only if there exists a matrix $\mathbf{A}_f \in \mathbb{F}^{m\times n}$ such that $f(\mathbf{x}) = \mathbf{A}_f \cdot \mathbf{x}$ for every $\mathbf{x} \in \mathbb{F}^n$.
*Hint:* Consider the vectors $f(\mathbf{e}_i)$ for every $0 \leq i < n$.[14]

*Exercise* 1.5. Show that there exists a decoding function $D_\oplus$ that given $E_\oplus(\mathbf{x}) + \mathbf{e}$ can determine if $\mathbf{e} = \mathbf{0}$ or $\mathbf{e}$ has one non-zero element.

*Exercise* 1.6. Argue that one can compute $E_\oplus(\mathbf{x})$ for any $\mathbf{x} \in \mathbb{F}_2^n$ with $O(n)$ operations over $\mathbb{F}_2$.

*Exercise* 1.7. Let $\mathbf{A} \in \mathbb{F}^{m\times n}$ be such that it has at most $s$ non-zero entries in it. Then argue that one can compute $\mathbf{A}\cdot\mathbf{x}$ with $O(s)$ operations over $\mathbb{F}$.

*Exercise* 1.8. Argue that $S \subseteq \mathbb{F}_2^n$ is a linear subspace if and only if for every $\mathbf{x}, \mathbf{y} \in S$, $\mathbf{x} + \mathbf{y} \in S$.

*Exercise* 1.9. Argue that the matrix $G_2$ has full rank over $\mathbb{F}_2$.

[13] This also implies $\Omega(mn)$ operations for any reasonable model of arithmetic computation but we'll leave this as is for now.

[14] $\mathbf{e}_i \in \mathbb{F}^n$ are vectors that are all 0s except in position $i$, where it is 1.

*Exercise* 1.10. Prove Proposition 1.4.2.
*Hint:* Use Exercise 1.4.

*Exercise* 1.11. Prove Proposition 1.4.3.
*Hint:* Exercise 1.3 would be useful.

*Exercise* 1.12. Show that $\mathbf{F}_n$ is a special case of a Vandermonde matrix.

*Exercise* 1.13. Show that for any $\mathbf{V}_n^{(\mathbf{a})}$ where **a** has all of its $m$ values being different has full rank.
*Hint:* Let $N = \min(n, m)$. Then prove that the sub-matrix $\mathbf{V}_N^{(a_0,\ldots,a_{N-1})}$ has determinant $\prod_{i=0}^{N}\prod_{0\le j\ne i<N}(a_i - a_j)$.

*Exercise* 1.14. Prove (1.7).

*Exercise* 1.15. Prove that $1 + X + X^2$ is the only irreducible polynomial of degree 2 over $\mathbb{F}_2$.

*Exercise* 1.16. For every $t \ge 1$ and field $\mathbb{F}$ of size at least $t$, show that there exists a polynomial with exactly $t$ (distinct) roots. Further, this is true even if the roots have to be from an arbitrary subset of $\mathbb{F}$ of size at least $t$.

*Exercise* 1.17. Prove Proposition 1.5.5.
*Hint:* For the existence argument consider the decoding function that outputs the codeword closest to $E(\mathbf{x})+\mathbf{e}$. For the negative result, think of an error vector based on two codewords that are closest to each other.

*Exercise* 1.18. Let the encoding function of an $[m, n]_q$ linear code be $E$. Then prove that the distance of the code is exactly the minimum Hamming weight of $E(\mathbf{x})$ over all non-zero $\mathbf{x} \in \mathbb{F}^n$.
*Hint:* Use the fact that $\Delta(\mathbf{y},\mathbf{z})$ is exactly the Hamming weight of $\mathbf{y}-\mathbf{z}$.

*Exercise* 1.19. Argue that any code with encoding function $E : \mathbb{F}_q^n \to \mathbb{F}_q^m n$ has distance at most $m - n + 1$.
*Hint:* Let the code have distance $d$. Now consider the "projection" of the code where we only consider the first $n - d + 1$ positions in the codeword. Can you prove a lower bound on the distance of this projected down code?

# Chapter 2

# Setting up the technical problem

In this chapter, we will setup the technical problem we would like to solve. We will start off with the 'pipe dream' question. Then we will realize soon enough that dreams generally do not come true and we will then re-calibrate out ambitions and then state a weaker version of our pipe dream. This weaker version will be the main focus of the rest of the notes.

## 2.1 Arithmetic circuit complexity

As mentioned in Chapter 1, we are interested in investigating the complexity of matrix-vector multiplication. In particular, we would like to answer the following question:

**Question 2.1.1.** *Given an $m \times n$ matrix $\mathbf{A}$, what is the optimal complexity of computing $\mathbf{A} \cdot \mathbf{x}$ (for arbitrary $\mathbf{x}$)?*

Note that to even begin to answer the question above, we need to fix our "machine model." One natural model is the so called RAM model on which we analyze most of our beloved algorithms. However, we do not understand the power of RAM model (in the sense that we do not have a good handle on what problems can be solved by say linear-time or quadratic-time algorithms[1]) and answering Question 2.1.1 in the RAM model seems hopeless.

So we need to consider a mode restrictive model of computation. Instead of going through a list of possible models, we will just state the model of computation we will use: *arithmetic circuit* (also known as the straight-line program). In the context of an arithmetic circuit that computes $\mathbf{y} = \mathbf{A}\mathbf{x}$, there are $n$ inputs gates (corresponding to $\mathbf{x}[0], \dots, \mathbf{x}[n-1]$) and $m$ output gates (corresponding to $\mathbf{y}[0], \dots, \mathbf{y}[m-1]$. All the internal gates correspond to the addition, multiplication, subtraction and division operator over the underlying field $\mathbb{F}$. The circuit is also allowed to use constants from $\mathbb{F}$ for "free." The complexity of the circuit will be its size: i.e. the number of addition, multiplication, subtraction and division gates in the circuit. Let us record this choice:

**Definition 2.1.1.** For any function $f : \mathbb{F}^n \to \mathbb{F}^m$, its *arithmetic circuit complexity* is the minimum number of addition, multiplication, subtraction and division operations over $\mathbb{F}$ to compute $f(\mathbf{x})$ for any $\mathbf{x} \in \mathbb{F}^n$.

Given the above, we have the following more specific version of Question 2.1.1:

[1] The reader might have noticed that we are ignoring the P vs. NP elephant in the room.

**Question 2.1.2.** *Given a matrix* $\mathbf{A} \in \mathbb{F}^{m\times n}$*, what is the arithmetic circuit complexity of computing* $\mathbf{A}\cdot\mathbf{x}$ *(for arbitrary* $\mathbf{x} \in \mathbb{F}^n$*)?*

In this chapter, we will delve into the above question a bit more. However, before we proceed, let us list what we gain from using arithmetic circuit complexity as our notion of complexity (and what do we lose).

### 2.1.1 What we gained and what we lost due to Definition 2.1.1

As it will turn out, all the gains in using arithmetic circuit complexity will come from nice theoretical implications and the (main) drawback will be from practical considerations.

We will start with the positives:

- As was mentioned in Section 1.3.2, we are interested in solving Question 2.1.2 for both finite and infinite fields. Using arithmetic circuit complexity allows us to abstract out the problem to general fields. Indeed many of the results we will talk about in these notes will work seamlessly for *any* field (or at least for large enough fields).

- As we will see in Chapter 4, the arithmetic circuit view is useful in that it allows us to prove 'meta-theorems' that might have been harder (or not even possible) to prove in a more general model.

- As we will see shortly, the arithmetic circuit complexity view allows us to talk about a concrete notion of optimal complexity, which would have been harder to argue with say the RAM model. In particular, this allows us to characterize this complexity in a purely combinatorial manner (instead of stating the optimal complexity in a purely computational manner), which is aesthetically pleasing.

- For finite fields $\mathbb{F}_q$, we can implement each basic operation over $\mathbb{F}_q$ in $\tilde{O}(\log q)$ in say the RAM model (see Exercise 2.1). In other words, when the fields are polynomially large in $mn$, this is not an issue.

- (This is more of a personal reason.) The author personally likes dealing with abstract fields and not having to worry about representation and numerical stability issues especially when dealing with $\mathbb{R}$ or $\mathbb{C}$.

The main disadvantage is the flip side of the last point:

- Ignoring representation and (especially) numerical stability issues generally spells doom for practical implementation of algorithms developed for arithmetic circuit complexity.

Given the above (biased set of) pros and cons, we will stick with arithmetic complexity and note that even though for practical implementations (over $\mathbb{R}$ or $\mathbb{C}$) algorithms designed for the arithmetic circuit complexity, they can still be a good starting point to then adapt these generic algorithms to handle the real issues of numerical stability over $\mathbb{R}$ or $\mathbb{C}$. Of course from a theoretical point of view, it still makes sense to talk about arithmetic circuit complexity over $\mathbb{R}$ or $\mathbb{C}$.

## 2.2 What do we know about Question 2.1.2

We now recall what we already know about Question 2.1.2 thanks to what we already covered in Chapter 1:

- If we want to answer Question 2.1.2 but in the worst-case sense (i.e. figure out the optimal complexity over *all* matrices in $\mathbb{F}^{m\times n}$), then the answer is $\Theta(mn)$ (see Section 1.2).
- On the other hand, if we know more about the matrix **A**, then we can do better. For example, if we know that **A** is $s$-sparse (i.e. it has at most $s$ non-zero entries), then the answer is $\Theta(s+n)$. (For the upper bound, see Exercise 1.7. The lower bound will follow from Exercise 2.2.)

The above two extremes provides more justification for why we stated Question 2.1.2 in the manner we did: the first items above shows that worst-case complexity is not that interesting, and the second item shows that it is possible to have a better result for certain sub-classes of matrices. However, pay close attention to what Question 2.1.2 is asking: we are seeking to *completely characterize* the complexity of matrix vector multiplication just based on matrix **A**– i.e. we are looking for the ultimately per-instance guarantee.

At this point, some of you might be up in arms or wondering what the author is smoking since this looks like a "pipe dream." If so, you would be mostly right– we're not going to get any close to satisfactorily resolving Question 2.1.2 in its full generality. But this quest will fall into the 'the journey is more important than the destination' cliche: along the way we will discover some really nice results.

### 2.2.1 Representation Issues

Before we proceed further with Question 2.1.2, let us pause and consider the issue of how we represent the matrices **A**. If we insist that the input to our algorithm has to be an $m \times n$ matrix, then there is not much hope in improving upon the $\Omega(mn)$ complexity (a similar argument to that of Exercise 1.3 will still work). In other words, we need a way to represent matrices that allow for $o(mn)$ complexity for matrix-vector multiplication. Here is a general way to do this:

Consider a *compression* function $f : \mathbb{F}^{m\times n} \to \mathbb{F}^*$ and a *decompression* function $D : \mathbb{F}^* \to \mathbb{F}^{m\times n}$. In other words, $C(\mathbf{A})$ for any $\mathbf{A} \in \mathbb{F}^{m\times n}$ is a compressed version of **A** and we can assume that our algorithm is given $C(\mathbf{A})$ and **x** as its inputs.

In the most general case both $(C, D)$ can depend on **A**. However, this turns out to be very unwieldy to handle so we will instead focus on families of matrices that are defined by a given $(C, D)$ pair of compressor and decompressor. For example, for sparse matrices the compressor would be an algorithm that for every non-zero entry $\mathbf{A}[i, j]$ outputs an entry $(i, j, \mathbf{A}[i, j])$ in the output (and the decompressor does the obvious reverse process).

Now consider a given compressor $C : \mathbb{F}^{m\times n} \to \mathbb{F}^s$: i.e. the compressed version is of size $s$. We first note that for such a family of matrices, the best complexity one can hope for is $O(s)$ (see Exercise 2.2). So now our next goal is:

**Question 2.2.1.** *What is the most general class of compressors we can design for which we can say something interesting about the corresponding complexity of matrix-vector multiplication?*

We will start off with a very natural notion of compression, which surprisingly will turn out to be powerful enough to capture Question 2.1.2 in its full generality (though only for very very large fields).

### 2.2.2 Linear circuit complexity

The main idea here is instead of considering the general arithmetic circuit complexity of **Ax**, let us consider the linear arithmetic circuit complexity. A linear arithmetic circuit only uses linear operations:

**Definition 2.2.1.** A linear arithmetic circuit (over $\mathbb{F}$) only allows operations of the form $\alpha X + \beta Y$, where $\alpha, \beta \in \mathbb{F}$ are constants while $X$ and $Y$ are the inputs to the operation. The *linear arithmetic circuit complexity* of **Ax** is the size of the smallest linear arithmetic circuit that computes **Ax** (where **x** are the inputs and the circuit depends on **A**). Sometimes we will overload terminology and the linear arithmetic circuit complexity of **Ax** as the linear arithmetic circuit complexity of (just) **A**.

We note that while the above is stated as a circuit to compute **Ax**, it also is a compressor for **A** (see Exercise 2.3).

We first remark that the linear arithmetic circuit complexity seems to be a very natural model to consider the complexity of computing **Ax** (recall this defines a linear function over **x**). In fact one could plausibly conjecture that going from general arithmetic circuit complexity to linear arithmetic circuit complexity of computing **Ax** should be without loss of generality (the intuition being: "What else can you do?").

It turns out that for infinite fields, the above intuition is correct:

**Theorem 2.2.1.** *Let* **F** *be an infinite field and* $\mathbf{A} \in \mathbb{F}^{m\times n}$. *Let* $\mathscr{C}(\mathbf{A})$ *and* $\mathscr{C}^L(\mathbf{A})$ *be the arithmetic circuit complexity and linear arithmetic circuit complexity of computing* **Ax** *(for arbitrary* **x***). Then* $\mathscr{C}^L(\mathbf{A}) = \Theta(\mathscr{C}(\mathbf{A}))$.

We defer the proof of Theorem 2.2.1 till later.

We first make some observations. First, it turns out that Theorem 2.2.1 can be proved for finite fields that are exponentially large (see Exercise 2.4). Second, it is a natural question to try and prove a version of Theorem 2.2.1 for small finite fields (say over $\mathbb{F}_2$). This question is very much open:

**Open Question 2.2.1.** *Prove (or disprove) Theorem 2.2.1 for* $\mathbb{F}_2$.

Next, we take a detour to talk about derivatives, which will be useful in proving Theorem 2.2.1 (as well as in Chapter 4).

#### (Partial) Derivatives

It turns out that we will only be concerned with studying derivatives of polynomials. For this, we can define the notion of a formal derivative (over univariate polynomials):

**Definition 2.2.2.** The formal derivative $\nabla_X(\cdot) : \mathbb{F}[X] \to \mathbb{F}[X]$ as follows. For every integer $i$,

$$\nabla_X\left(X^i\right) = i \cdot X^{i-1}.$$

The above definition can be extended to all polynomials in $\mathbb{F}[X]$ by insisting that $\nabla_X(\cdot)$ be a linear map. That is for every $\alpha, \beta \in \mathbb{F}$ and $f(X), g(X) \in \mathbb{F}[X]$ we have

$$\nabla_X\left(\alpha f(X) + \beta g(X)\right) = \alpha \nabla_X\left(f(X)\right) + \beta \nabla_X\left(g(X)\right).$$

We note that over $\mathbb{R}$, the above definition when applied to polynomials over $\mathbb{R}[X]$ gives the same result as the usual notion of derivatives.

We will actually need to work with derivatives of multi-variate polynomials. We will use $\mathbb{F}[X_1,\dots,X_m]$ to denote the set of multivariate polynomials with variables $X_1,\dots,X_m$. For example, $3XY + Y^2 + 1.5X^3Y^4$ is in $\mathbb{R}[X,Y]$. We extend the definition of derivatives from Definition 2.2.2 to the following (which also called a *gradient*)

**Definition 2.2.3.** Let $f(X_1,\dots,X_n)$ be a polynomial in $\mathbb{F}[X_1,\dots,X_n]$. Then define its derivative as (where we use $\mathbf{X} = (X_1,\dots,X_n)$ to denote the vector of variables):

$$\nabla_{\mathbf{X}}\left(f(\mathbf{X})\right) = \left(\nabla_{X_1}\left(f(\mathbf{X})\right),\dots,\nabla_{X_n}\left(f(\mathbf{X})\right)\right),$$

where $\nabla_{X_i}\left(f(\mathbf{X})\right)$ we think of $f(\mathbf{X})$ as being a polynomial in $X_i$ with coefficients in $\mathbb{F}[X_1,\dots,X_{i-1},X_{i+1},\dots,X_n]$.

Finally note that $\nabla_{X_i}\left(f(\mathbf{X})\right)$ is again a polynomial and we will denote its evaluation at $\mathbf{a} \in \mathbb{F}^n$ as $\nabla_{X_i}\left(f(\mathbf{X})\right)_{|\mathbf{a}}$. We extend this notation to the gradient by

$$\nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\mathbf{a}} = \left(\nabla_{X_1}\left(f(\mathbf{X})\right)_{|\mathbf{a}},\dots,\nabla_{X_n}\left(f(\mathbf{X})\right)_{|\mathbf{a}}\right).$$

For example

$$\nabla_{X,Y}\left(3XY + Y^2 + 1.5X^3Y^4\right) = \left(3Y + 4.5X^2Y^4, 3X + 2Y + 6X^3Y^3\right).$$

We will use the fact that the derivative above satisfies the *product rule*:

**Lemma 2.2.2.** *For any two polynomials $f(\mathbf{X}), g(\mathbf{X}) \in \mathbb{F}[\mathbf{X}]$, it holds that*

$$\nabla_{\mathbf{X}}\left(f(\mathbf{X}) \cdot g(\mathbf{X})\right) = f(\mathbf{X}) \cdot \nabla_{\mathbf{X}}\left(g(\mathbf{X})\right) + g(\mathbf{X}) \cdot \nabla_{\mathbf{X}}\left(f(\mathbf{X})\right).$$

The proof pretty much follows from definition: see Exercise 2.5.

### Why multivariate polynomials?

Before we dive into the proof of Theorem 2.2.1, we state an obvious connection between an arithmetic circuit with addition, subtraction and multiplication[2] and multivariate polynomials.

**Proposition 2.2.3.** *Consider an arithmetic circuit (with addition, subtraction and multiplication) $\mathcal{C}$ that computes $\mathbf{Ax}$ for $\mathbf{A} \in \mathbb{F}^{m\times n}$ and $\mathbf{x} \in \mathbb{F}^n$. Then every gate $g$ in $\mathcal{C}$ computes a function $g(\mathbf{x})$. Further, if one thinks of $\mathbf{x}[i]$ as variable $X_i$, then $g(X_0,\dots,X_{n-1})$ is a multivariate polynomial in $\mathbb{F}[X_0,\dots,X_{n-1}]$.*

The above follows from a straightforward inductive argument and is left to Exercise 2.6.

[2] Note there is no division.

### Proof of Theorem 2.2.1

*Proof.* We first note that by definition, $\mathscr{C}(\mathbf{A}) \le \mathscr{C}^L(\mathbf{A})$ for any matrix $\mathbf{A} \in \mathbb{F}^{m\times n}$. For the rest of the proof, we will argue that given a general arithmetic circuit for $\mathscr{C}_{\mathbf{A}}$ to compute $\mathbf{Ax}$, we can compute a linear circuit $\mathscr{C}^L_{\mathbf{A}}$ that only has a constant factor many gates. We will make the assumption that $\mathscr{C}_{\mathbf{A}}$ does not have any division gate.[3]

For the rest of the proof we will use $\mathbf{x} \in \mathbb{F}^n$ to denote the input vector and $\mathbf{X}$ to denote the vector of variables $X_0, \dots, X_{n-1}$ (where think of $X_i$ corresponding to the input symbols $\mathbf{x}[i]$). Let $\mathbf{y} = \mathbf{Ax}$ and let $Y_i(\mathbf{X})$ correspond to the polynomial representing the output gate for $\mathbf{y}[i]$ (by Proposition 2.2.3 such a polynomial exists). We first claim that $Y_i(\mathbf{X})$ is a linear polynomial (i.e. of the form $\sum_{i=0}^{n-1} c_i X_i$).[4]

Given that $Y_i(\mathbf{X})$ is a linear polynomial, we have that

$$\mathbf{y}[i] = \left\langle \nabla_{\mathbf{X}} (Y_i(\mathbf{X}))_{|\mathbf{0}}, \mathbf{x} \right\rangle. \tag{2.1}$$

Indeed, by Exercise 2.9, we have that $Y_i(\mathbf{X}) = \sum_{j=0}^{n-1} \mathbf{A}[i,j]\cdot X_j$ and hence we have $\nabla_{X_j}(Y_i(\mathbf{X})) = \mathbf{A}[i,j]$, which implies the above.[5] Next we will argue how we can convert the general arithmetic circuit $\mathscr{C}_{\mathbf{A}}$ into an equivalent linear arithmetic circuit by induction on the size of the circuit (i.e. the number of gates). In particular, we will argue that any sub-circuit of $\mathscr{C}_{\mathbf{A}}$ with size $s$ is equivalent to a linear circuit of size $s$.

Consider the base case of each output gate is just an input gate $X_i$. In this case the argument is trivial. For the inductive argument pick any $0 \le i < m$ such that the output $\mathbf{y}[i]$ is computed at a non-trivial gate. We have three cases to consider:

- *Case 1:* The output gate is an addition operator. That is,

$$Y_i(\mathbf{X}) = f(\mathbf{X}) + g(\mathbf{X}).$$

  In this case since the derivative is a linear operator, we have

$$\nabla_{\mathbf{X}} (Y_i(\mathbf{X}))_{|\mathbf{0}} = \nabla_{\mathbf{X}} \left(f(\mathbf{X})\right)_{|\mathbf{0}} + \nabla_{\mathbf{X}} \left(g(X)\right)_{|\mathbf{0}}.$$

  In other words, we have

$$\left\langle \nabla_{\mathbf{X}} (Y_i(\mathbf{X}))_{|\mathbf{0}}, \mathbf{x} \right\rangle = \left\langle \nabla_{\mathbf{X}} \left(f(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x} \right\rangle + \left\langle \nabla_{\mathbf{X}} \left(g(X)\right)_{|\mathbf{0}}, \mathbf{x} \right\rangle. \tag{2.2}$$

  Inductively, we know that both $\left\langle \nabla_{\mathbf{X}} \left(f(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x} \right\rangle$ and $\left\langle \nabla_{\mathbf{X}} \left(g(X)\right)_{|\mathbf{0}}, \mathbf{x} \right\rangle$ can be computed with linear circuits of the same size as needed to compute $f(\mathbf{X})$ and $g(\mathbf{X})$. Since we only need one linear operator to compute (2.2), induction completes the proof in this case.

- *Case 2:* The output gate is a subtraction operator. The argument in this case is basically the same as in the previous one and is omitted.

- *Case 3:* The output gate is multiplication by a scalar $\alpha \in \mathbb{F}$, i.e.

$$Y_i(\mathbf{X}) = \alpha \cdot f(\mathbf{X}).$$

[3] This assumption can be removed: see Exercise 2.7.

[4] This part uses the fact that $\mathbb{F}$ is infinite. Also while the statement seems "obvious," it needs a proof– see Exercise 2.9.

[5] The choice to evaluate $\nabla_{\mathbf{X}}\left(Y_i(\mathbf{X})\right)$ at $\mathbf{0}$ is arbitrary: any fixed vector in $\mathbb{F}^n$ would suffice for the proof.

Again since the derivative is a linear operator, we have

$$\left\langle \nabla_{\mathbf{X}}\left(Y_i(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle = \alpha \cdot \left\langle \nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle.$$

Since $\alpha \in \mathbb{F}$ is a constant independent of $\mathbf{x}$, we only need one linear operator to compute the above and induction completes the proof in this case.

- *Case 4:* The output gate is a multiplication operator. That is,

  $$Y_i(\mathbf{X}) = f(\mathbf{X}) \cdot g(\mathbf{X}).$$

  Proposition 2.2.2 implies that

  $$\nabla_{\mathbf{X}}\left(Y_i(\mathbf{X})\right)_{|\mathbf{0}} = g(\mathbf{0}) \cdot \nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\mathbf{0}} + f(\mathbf{0}) \cdot \nabla_{\mathbf{X}}\left(g(X)\right)_{|\mathbf{0}}.$$

  In other words, we have

  $$\left\langle \nabla_{\mathbf{X}}\left(Y_i(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle = g(\mathbf{0}) \left\langle \nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle + f(\mathbf{0}) \cdot \left\langle \nabla_{\mathbf{X}}\left(g(X)\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle.$$

  Again, inductively we know that both $\left\langle \nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle$ and $\left\langle \nabla_{\mathbf{X}}\left(g(X)\right)_{|\mathbf{0}}, \mathbf{x}\right\rangle$ can be computed with linear circuits of the same size as needed to compute $f(\mathbf{X})$ and $g(\mathbf{X})$. Since both $f(\mathbf{0})$ and $g(\mathbf{0})$ are constants *independent* on $\mathbf{x}$, the above can me implemented with one linear operator. Again induction, completes the proof.

□

## 2.3 Let's march on

We now return to Question 2.1.2. Note that Theorem 2.2.1 implies that for infinite $\mathbb{F}$, the answer to Question 2.1.1 is the "minimum linear arithmetic circuit complexity." Even for finite fields, it is a natural to ask Question 2.1.1 but for linear circuit complexity. However, it seems like even though we have identified a natural notion of complexity for computing $\mathbf{Ax}$, we have not made any tangible progress towards getting concrete bounds. Further, this notion of linear arithmetic circuit complexity is not satisfactory since it does not seem to give an more structural information about the matrix $\mathbf{A}$. It turns out that for the latter, there is a nice equivalent combinatorial representation of this complexity: see Exercise 2.10.

One natural question, given the situation above, is to ask how easy is to compute the linear arithmetic circuit complexity of a given matrix $\mathbf{A}$? It turns out that one can show it is NP-hard. One could instead think of coming up with an *approximation*: i.e. coming up with a linear arithmetic circuit to compute $\mathbf{Ax}$ with size at most some factor $\alpha > 1$ of the optimal. It is known that it is NP-hard to do this for some (small) constant $\alpha_0 < 1$. The best known polynomial time algorithm achieves $\alpha = O(n/\log n)$. In other words, the following is wide open:

**Open Question 2.3.1.** *Have tighter (upper and lower) bounds on $\alpha$ for which one has a polynomial time algorithm (or some hardness result) to solve the above problem.*

Given the above, making direct progress on Question 2.1.2 seems some way off. Hence, for the rest of the notes, we will consider the following dual version of Question 2.1.2:

**Question 2.3.1.** *What is the largest class of family of matrices* $\mathbf{A} \in \mathbb{F}^{m\times n}$ *for which one can guarantee that the (linear) arithmetic circuit complexity of* $\mathbf{Ax}$ *is* $o(mn)$*. More dramatically, we would like the complexity to be* $\tilde{O}(m+n)$*.*

In particular, for the rest of the notes, we will aim for $\tilde{O}(m+n)$ complexity. Also to make our lives simpler (and have to worry about one less parameter), we will make the following assumption:

**Assumption 2.3.1.** *We will consider square matrices, i.e.* $m = n$*. In other words, for Question 2.3.1, we are aiming for* $\tilde{O}(n)$ *arithmetic circuit complexity.*

## 2.4 What do we know about Question 2.3.1?

### 2.4.1 Some obvious cases

We start with some very simple conditions on $\mathbf{A}$ for which we can answer Question 2.3.1 in the affirmative.

- We already have seem that an $O(n)$-sparse matrix $\mathbf{A}$ allows for $O(n)$ arithmetic circuit complexity for $\mathbf{Ax}$ (recall Exercise 1.7). Indeed one can do this with $O(n)$- linear arithmetic circuit complexity (see Exercise 2.11).
- Another well-studied case is that of low rank matrices: if $\mathbf{A}$ has rank $O(1)$, then again one can achieve $O(n)$ linear arithmetic circuit complexity for $\mathbf{Ax}$ (see Exercise 2.12).
- Given the above (fairly simple) results, it seems we should modify Question 2.3.1 to the following one:

  **Question 2.4.1.** *What is the largest class of family of* dense *and* full rank *matrices* $\mathbf{A} \in \mathbb{F}^{n\times n}$ *for which one can guarantee that the (linear) arithmetic circuit complexity of* $\tilde{O}(n)$*.*

  We have already seen some examples of matrices for which the above is true, which we will defer to the next subsection. But before that consider the following simpler case: let $\mathbf{U}_n \in \mathbb{F}^{n\times n}$ be defined as

  $$\mathbf{U}[i,j] = \begin{cases} 1 & \text{if } i \le j \\ 0 & \text{otherwise} \end{cases}.$$

  For example

  $$\mathbf{U}_3 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

  It is fairly easy to see that the linear arithmetic circuit complexity of $\mathbf{U}_n\mathbf{x}$ is $O(n)$ (see Exercise 2.13).

Now, we will consider some more examples where the linear complexity being $\tilde{O}(n)$ is not obvious: indeed some of the algorithms developed are some of the most influential algorithms ever designed (even beyond matrix-vector multiplication).

### 2.4.2 Some non-obvious cases

We now list a varied (and curated) list of matrices **A** for which answering Question 2.4.1 in the affirmative (should) look non-obvious.

**Discrete Fourier matrix**

We have seen this in Section 1.4.2 (recall Definition 1.4.6 and Theorem 1.4.4).

**Vandermonde matrix**

We have seen this in Section 1.4.2 (recall Definition 1.4.7 and its matrix-vector linear arithmetic circuit complexity is $O(n\log^2 n)$– see Chapter 3).

**Boolean Hadamard matrix**

Consider the following matrix:

**Definition 2.4.1.** Let $n = 2^m$ and we index each row and columns as vectors in $\{0,1\}^m$. Then consider the matrix (where $\mathbf{i},\mathbf{j} \in \mathbb{F}_2^m$):

$$\mathbf{H}_m[\mathbf{i},\mathbf{j}] = \langle \mathbf{i},\mathbf{j} \rangle .$$

Note that we have defined the matrix over $\mathbb{F}_2$ but you might have seen it defined equivalently over $\mathbb{R}$, where the $(\mathbf{i},\mathbf{j})$'th entry is given by $(-1)^{\langle \mathbf{i},\mathbf{j} \rangle}$. It can be shown that this matrix has rank $2^m - 1$ (see Exercise 2.14.

The following result is known:

**Theorem 2.4.1.** *One can compute* $\mathbf{H}_m \cdot \mathbf{x}$ *for any* $\mathbf{x} \in \mathbb{F}_2^n$ *with linear arithmetic circuit complexity of* $O(n\log n)$.

**Toeplitz matrix**

Consider the following matrix:

**Definition 2.4.2.** Arbitrarily fix $\mathbf{A}[i,0]$ and $\mathbf{A}[0,j]$ for every $0 \le i,j < n$. Then the rest of the entries are defined as

$$\mathbf{T}_n[i,j] = \mathbf{T}_n[i-1,j-1].$$

It can be shown that this matrix has full rank (under certain conditions– see Exercise 2.15).

The following result is known:

**Theorem 2.4.2.** *One can compute* $\mathbf{T}_n \cdot \mathbf{x}$ *for any* $\mathbf{x} \in \mathbb{F}^n$ *with linear arithmetic circuit complexity of* $O(n\log^2 n)$.

**Cauchy matrix**

Consider the following matrix:

**Definition 2.4.3.** Arbitrarily fix $\mathbf{s},\mathbf{t} \in \mathbb{F}^n$ such that for every $0 \le i,j < n$, $\mathbf{s}[i] \neq \mathbf{t}[j]$, $\mathbf{s}[i] \neq \mathbf{s}[j]$ and $\mathbf{t}[i] \neq \mathbf{t}[j]$ and

$$\mathbf{C}_n[i,j] = \frac{1}{\mathbf{s}[i] - \mathbf{t}[j]}.$$

It can be shown that this matrix has full rank (see Exercise 2.16).

The following result is known:

**Theorem 2.4.3.** *One can compute* $\mathbf{C}_n \cdot \mathbf{x}$ *for any* $\mathbf{x} \in \mathbb{F}^n$ *with linear arithmetic circuit complexity of* $O(n\log^2 n)$.

**Discrete Chebyshev matrix**

Consider the following matrix:

**Definition 2.4.4.** Arbitrarily fix $\alpha_0, \dots, \alpha_{n-1} \in \mathbb{F}^n$ such that for every $0 \le i, j < n$, we have

$$\hat{\mathbf{C}}_n[i, j] = T_i(\alpha_j),$$

where $T_i(X)$ is the $i$'th Chebyshev polynomial (see Definition 1.5.7).

It can be shown that this matrix has full rank (see Exercise 2.17.
The following result is known:

**Theorem 2.4.4.** *One can compute* $\hat{\mathbf{C}}_n \cdot \mathbf{x}$ *for any* $\mathbf{x} \in \mathbb{F}^n$ *with linear arithmetic circuit complexity of* $O(n \log^2 n)$.

**What's the point here?**

A natural question to ask is why we listed the matrices that we did in this section. The first answer is that they are all mathematically pleasing. Second (and perhaps most importantly), these matrices have found uses in practice (and in theory) and are very well-studied matrices. Indeed, a lot of work over the years have gone into arguing the theorems about the linear arithmetic circuit complexity of doing matrix-vector multiplication for such matrices. These algorithm can look a bit ad-hoc, though curiously they all ultimately reduce to Theorem 1.4.4. A natural question to ask is

**Question 2.4.2.** *Is it possible to show that all the matrices that we have seen in this chapter have* $\tilde{O}(n)$ *linear arithmetic circuit complexity of computing the corresponding matrix-vector multiplication via a* single *algorithm?*

By the end of these notes, we would have answered this question in the affirmative (and then some).

## 2.5 Exercises

*Exercise* 2.1. Argue that adding, subtracting, multiplying and dividing two elements in $\mathbb{F}_q$ can be done in $\tilde{O}(\log q)$ time.
*Hint:* For multiplication and division, for now assume that one can multiply and divide two polynomials of degree $d$ over $\mathbb{F}$ can be done with $\tilde{O}(d)$ operations over $\mathbb{F}$. We will argue these claims in Chapter 3.

*Exercise* 2.2. Assume that we fix a family of matrices in $\mathbb{F}^{m \times n}$ that can be presented with $s$ parameters from $\mathbb{F}$. Then any algorithms that computes $\mathbf{Ax}$ for worst-case $\mathbf{x} \in \mathbb{F}^n$ and worst-case $\mathbf{A}$ from the chosen family has arithmetic complexity $\Omega(s)$.
*Hint:* Generalize the argument for Exercise 1.3.

*Exercise* 2.3. Let $\mathscr{C}$ be an arithmetic circuit that computes $\mathbf{Ax}$ for arbitrary $\mathbf{x}$. Then $\mathscr{C}$ is a compressor for $\mathbf{A}$.
*Hint:* Try to evaluate $\mathscr{C}$ in special inputs.

*Exercise* 2.4. Prove Theorem 2.2.1 for finite field of size at least $2^{\Omega(mn)}$.
*Hint:* Define an appropriate notion of formal derivative over finite fields and then re-do the proof over infinite fields. Do you see where you need the field size to be exponentially large?

*Exercise* 2.5. Prove Lemma 2.2.2.

*Exercise* 2.6. Prove Proposition 2.2.3.

*Exercise* 2.7. Prove Theorem 2.2.1 for the most general case when arithmetic circuits can have division gates.

*Exercise* 2.8. Let $f(X_0, X_1, \dots, X_m)$ is a polynomial such that $f(\mathbf{X})$ has degree $< d$ in each $X_i$. The consider the following univariate polynomial:

$$f_K(Y) = f(Y, Y^d, \dots, Y^{d^m}),$$

i.e. the polynomial obtained by substituting each $X_i$ by $Y^{d^i}$. Argue that there is a one of one correspondence between a polynomial $f(\mathbf{X})$ and its corresponding *Kronecker substitution* $f_K(Y)$. In particular, $f(\mathbf{X})$ is a non-zero polynomial if and only if $f_k(Y)$ is.

*Exercise* 2.9. Argue that the polynomial $Y_i(X_0, \dots, X_{n-1})$ defined in proof of Theorem 2.2.1 is a linear polynomial. In particular, it is exactly the linear polynomial $\sum_{j=0}^{n-1} \mathbf{A}[i, j] \cdot X_j$.

*Hint:* Use Exercise 2.8 and the degree mantra.

*Exercise* 2.10. Prove that for any matrix $\mathbf{A} \in \mathbb{F}^{m\times n}$, $\mathsf{spw}(\mathbf{A})$ (to be defined shortly) is the same up to constant factors of the linear arithmetic complexity of $\mathbf{A}$ plus the number of non-zero rows of $\mathbf{A}$.

In the rest of the exercise, we will define $\mathsf{spw}(\mathbf{A})$. First, we define the notation of a *factorization* of $\mathbf{A}$, which is a sequence of matrices $\mathbf{A}_i \in \mathbb{F}^{k_i \times k_{i-1}}$ for $1 \le i \le p$ for some integer $p \ge 1$ such that $k_0 = n$ and $k_p = m$ with

$$\mathbf{A} = \mathbf{A}_p \cdot \mathbf{A}_{p-1} \cdots \mathbf{A}_2 \cdot \mathbf{A}_1.$$

A *proper factorization* has the additional constraint that none of the $\mathbf{A}_i$ (except for $i = p$) has a zero row and none of the $\mathbf{A}_i$ (except for $i = 1$) has a zero column. The *sparse product width* of a factorization is defined as

$$\mathsf{spw}(\mathbf{A}_1, \dots, \mathbf{A}_p) = \sum_{i=1}^{p} \|\mathbf{A}_i\|_0 - \sum_{i=1}^{p-1} k_i,$$

where $\|\mathbf{M}\|_0$ denotes the sparsity of $\mathbf{M}$ (i.e. the number of non-zero elements in $\mathbf{M}$). Finally, $\mathsf{spw}(\mathbf{A})$ is defined as the minimum of $\mathsf{spw}$ of any proper factorization of $\mathbf{A}$.

*Exercise* 2.11. Let $\mathbf{A} \in \mathbb{F}^{m\times n}$ be such that it has at most $s$ non-zero entries in it. Then argue that $\mathbf{Ax}$ has *linear* arithmetic circuit complexity of $O(s)$.

*Exercise* 2.12. Let $\mathbf{A} \in \mathbb{F}^{n\times n}$ be such that it has rank $r$. Then argue that $\mathbf{Ax}$ has *linear* arithmetic circuit complexity of $O(rn)$.

*Hint:* First prove that $\mathbf{A}$ has rank $r$ if and only if there exists matrices $\mathbf{G}, \mathbf{H} \in \mathbb{F}^{n\times r}$ such that $\mathbf{A} = \mathbf{G}\cdot\mathbf{H}^T$ and then use it.

*Exercise* 2.13. Show that the linear arithmetic circuit complexity of $\mathbf{U}_n\mathbf{x}$ is $O(n)$.

*Exercise* 2.14. Show that $\mathbf{H}_m$ has rank $2^m - 1$.

*Exercise* 2.15. Consider the following special case of Toeplitz matrix called the *circulant matrix* $\mathbf{C} \in \mathbb{F}^{n\times n}$, which is defined as follows. Let $\mathbf{c} \in \mathbb{F}^n$ be a vector and let $\mathbf{C}[0,:] = \mathbf{c}$ and every subsequent row is right rotation of the previous row, i.e. $\mathbf{C}[i, j] = \mathbf{C}[i-1, (j-1) \mod n]$ for every $1 \le i < n$.

First argue that $\mathbf{C}$ is a Toeplitz matrix. Then argue that $\mathbf{C}$ has full rank if and only if $\mathbf{F}_n\mathbf{c}$ is all non-zero.

*Hint:* First argue that $\mathbf{C} = \mathbf{F}_n^{-1}\mathrm{diag}(\mathbf{F}_n\mathbf{c})\mathbf{F}_n$ and then use it to argue full rank.

*Exercise* 2.16. Show that $\mathbf{C}_n$ has rank $n$.

*Exercise* 2.17. Show that $\hat{\mathbf{C}}_n$ has rank $n$.

# Chapter 3

# How to efficiently deal with polynomials

# Chapter 4

# Some neat results you might not have heard of

In this chapter, we will present two results related to matrix-vector multiplication that are very natural but you *might* not have heard about. As a bonus, these results also highlight why choosing arithmetic circuit complexity can provide a nice lens with which one can prove interesting 'meta theorems.' Finally, we will mention some other popular linear algebra operations that are related to matrix-vector multiplication but unfortunately have been given very little real estate in these notes.

## 4.1 Back to (not so) deep learning

To motivate the first 'meta theorem' (which actually holds for *any* arithmetic circuit and not just those computing **Ax**), we go back to the single layer neural network that we studied earlier in Section 1.3.2. In particular, we will consider a single layer neural network that is defined by

$$\mathbf{y} = g\left(\mathbf{W}\cdot\mathbf{x}\right), \tag{4.1}$$

where $\mathbf{W} \in \mathbb{F}^{m\times n}$ and $g : \mathbb{F}^m \to \mathbb{F}^m$ is a non-linear function. Further,

**Assumption 4.1.1.** *We will assume that non-linear function $g : \mathbb{F}^m \to \mathbb{F}^m$ is obtained by applying the same function $g : \mathbb{F} \to \mathbb{F}$ to each of the $m$ elements.*

In other words, (4.1) is equivalently stated as for every $0 \le i < m$:

$$\mathbf{y}[i] = g\left(\langle \mathbf{W}[i,:], \mathbf{x}\rangle\right).$$

Recall that in Section 1.3.2, we had claimed (without any argument) that the complexity of learning the weight matrix **W** given few samples is governed by the complexity of matrix-vector multiplication for **W**. In this section, we will rigorously argue this claim. To do this, we define the learning problem more formally:

**Definition 4.1.1.** Given $L$ *training data* $\left(\mathbf{y}^{(\ell)}, \mathbf{x}^{(\ell)}\right)$ for $\ell \in [L]$, we want to compute a matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ that minimizes the error

$$E(\mathbf{W}) = \sum_{\ell=1}^{L} \left\| \mathbf{y}^{(\ell)} - g\left(\mathbf{W}\cdot\mathbf{x}^{(\ell)}\right)\right\|_2^2.$$

Note that in the above the training searches for the 'best' weight matrix from the set of all matrices in $\mathbb{R}^{m\times n}$. However, we are interested in searching for the best weight matrix with a certain class.[1] To abstract this we will assume that

**Assumption 4.1.2.** *Given a vector $\boldsymbol{\theta} \in \mathbb{F}^s$ for some $s = s(m,n)$ such that the vector $\boldsymbol{\theta}$ completely specifies a matrix in our chosen family. We will use $\mathbf{W}_{\boldsymbol{\theta}}$ to denote the class of matrix family parameterized by $\boldsymbol{\theta}$.*

For example, if $s = mn$, then we get the set of all matrices in $\mathbb{F}^{m\times n}$. On the other hand, for say the Vandermonde matrix (recall Definition 1.4.7), we have $s(m,n) = m$ and $\boldsymbol{\theta} = (\alpha_0,\dots,\alpha_{m-1})$ for distinct $\alpha_i$'s. Given this, we generalize Definition 4.1.1

**Definition 4.1.2.** Given $L$ *training data* $\left(\mathbf{y}^{(\ell)},\mathbf{x}^{(\ell)}\right)$ for $\ell \in [L]$, we want to compute the parameters of an $m \times n$ matrix $\boldsymbol{\theta} \in \mathbb{R}^{s(m,n)}$ that minimizes the error

$$E(\boldsymbol{\theta}) = \sum_{\ell=1}^{L} \left\| \mathbf{y}^{(\ell)} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \mathbf{x}^{(\ell)}\right)\right\|_2^2.$$

While there exists techniques to solve the above problem theoretically, in practice *Gradient Descent* is commonly used to solve the above problem. In particular, one starts off with an initial state $\boldsymbol{\theta} = \boldsymbol{\theta}_0 \in \mathbb{R}^s$ and one keeps changing $\boldsymbol{\theta}$ is opposite direction of $\nabla_{\boldsymbol{\theta}}(E(\boldsymbol{\theta}))$ till the error is below a pre-specified threshold (or one goes beyond a pre-specified number of iterations. Algorithm 2 has the details.

**Algorithm 2** Gradient Descent

```
INPUT: $\eta > 0$ and $\varepsilon > 0$
OUTPUT: $\boldsymbol{\theta}$

$i \leftarrow 0$
Pick $\boldsymbol{\theta}_0$                    ▷ This could be arbitrary or initialized to something more specific
WHILE $|E(\boldsymbol{\theta}_i)| \ge \varepsilon$ DO          ▷ One could also terminate based on number of iterations
    $\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \eta \cdot (\nabla_{\boldsymbol{\theta}}(E(\boldsymbol{\theta})))_{|\boldsymbol{\theta}_i}$          ▷ $\eta$ is the 'learning rate'
    $i \leftarrow i+1$
RETURN $\boldsymbol{\theta}_i$
```

### 4.1.1 Computing the gradient

It is clear from Algorithm 2, that the most computationally intensive part is computing the gradient. We first show that if one can compute a related gradient, then we could implement Algorithm 2. In Section 4.2 we will show that this latter gradient computation is closely tied to computing $\mathbf{Wx}$. We first argue:

**Lemma 4.1.1.** *If for every $\mathbf{z} \in \mathbb{R}^m$ and $\mathbf{u} \in \mathbb{R}^n$, one can compute $\left(\nabla_{\boldsymbol{\theta}}\left(\mathbf{z}^T\mathbf{W}_{\boldsymbol{\theta}}\mathbf{u}\right)\right)_{|\mathbf{a}}$ for any $\mathbf{a} \in \mathbb{R}^s$ in $T_1(m,n)$ operations and $\mathbf{Wu}$ in $T_2(m,n)$ operations, then one can compute $(\nabla_{\boldsymbol{\theta}}(E(\boldsymbol{\theta})))_{|\boldsymbol{\theta}_0}$ for a fixed $\boldsymbol{\theta}_0 \in \mathbb{R}^s$ in $O(L(T_1(m,n) + T_2(m,n)))$ operations.*

[1] The hope here is that this class would be restricted enough so that computing $\mathbf{Wx}$ would be efficient but the class would be expressive enough to capture 'interesting' functions. We will ignore the latter aspect completely in these notes.

*Proof.* For notational simplicity define

$$\mathbf{W} = \mathbf{W}_{\boldsymbol{\theta}_0}$$

and

$$E_\ell(\boldsymbol{\theta}) = \left\| \mathbf{y}^{(\ell)} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \mathbf{x}^{(\ell)}\right) \right\|_2^2 .$$

Fix $\ell \in [L]$. We will show that we can compute $\nabla_{\boldsymbol{\theta}} \left(E_\ell(\boldsymbol{\theta})\right)_{|\boldsymbol{\theta}_0}$ with $O(T_1(m,n) + T_2(m,n))$ operations, which would be enough since $\nabla_{\boldsymbol{\theta}} \left(E(\boldsymbol{\theta})\right) = \sum_{\ell=1}^{L} \nabla_{\boldsymbol{\theta}} \left(E_\ell(\boldsymbol{\theta})\right)$.

For notational simplicity, we will use $\mathbf{y}, \mathbf{x}$ and $E(\boldsymbol{\theta})$ to denote $\mathbf{y}^{(\ell)}, \mathbf{x}^{(\ell)}$ and $E_\ell(\boldsymbol{\theta})$ respectively. Note that

$$\begin{aligned} E(\boldsymbol{\theta}) &= \left\| \mathbf{y} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \mathbf{x}\right) \right\|_2^2 \\ &= \sum_{i=0}^{m-1} \left( \mathbf{y}[i] - g\left( \sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \mathbf{x}[j] \right) \right)^2 . \end{aligned}$$

Applying the chain rule of the gradient on the above, we get (where $g'(x)$ is the derivative of $g(x)$):

$$\nabla_{\boldsymbol{\theta}} \left(E(\boldsymbol{\theta})\right) = -2 \sum_{i=0}^{m-1} \left( \mathbf{y}[i] - g\left( \sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \mathbf{x}[j] \right) \right) g'\left( \sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \mathbf{x}[j] \right) \sum_{j=1}^{n-1} \left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{W}_{\boldsymbol{\theta}}[i,j] \right) \mathbf{x}[j] \right). \tag{4.2}$$

Define a vector $\mathbf{z} \in \mathbb{R}^m$ such that for any $0 \le i < m$,

$$\mathbf{z}[i] = -2 \left( \mathbf{y}[i] - g\left( \langle \mathbf{W}[i,:], \mathbf{x} \rangle \right) \right) g'\left( \langle \mathbf{W}[i,:], \mathbf{x} \rangle \right).$$

Note that once we compute $\mathbf{W}\mathbf{x}$ (which by assumption we can do in $T_2(m,n)$ operation, we can compute $\mathbf{z}$ with $O(T_2(m,n))$ operations.[2] Further, note that $\mathbf{z}$ is independent of $\boldsymbol{\theta}$.

From (4.2), we get that

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \left(E(\boldsymbol{\theta})\right)_{|\boldsymbol{\theta}_0} &= -2 \sum_{i=0}^{m-1} \left( \mathbf{y}[i] - g\left( \langle \mathbf{W}[i,:], \mathbf{x} \rangle \right) \right) g'\left( \langle \mathbf{W}[i,:], \mathbf{x} \rangle \right) \sum_{j=0}^{n-1} \left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{W}_{\boldsymbol{\theta}}[i,j] \right)_{|\boldsymbol{\theta}_0} \cdot \mathbf{x}[j] \right) \\ &= \sum_{i=0}^{m-1} \mathbf{z}[i] \cdot \sum_{j=0}^{n-1} \left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{W}_{\boldsymbol{\theta}}[i,j] \right)_{|\boldsymbol{\theta}_0} \cdot \mathbf{x}[j] \right) \\ &= \left( \nabla_{\boldsymbol{\theta}} \left( \sum_{i=0}^{m-1} \mathbf{z}[i] \cdot \sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \cdot \mathbf{x}[j] \right) \right)_{|\boldsymbol{\theta}_0} \\ &= \left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{z}^T \mathbf{W}_{\boldsymbol{\theta}} \mathbf{x} \right) \right)_{|\boldsymbol{\theta}_0} \end{aligned}$$

In the above, the first equality follows from our notation that $\mathbf{W} = \mathbf{W}_{\boldsymbol{\theta}_0}$, the second equality follows from the definition of $\mathbf{z}$ and the third equality follows from the fact that $\mathbf{z}$ is independent of $\boldsymbol{\theta}$. The proof is complete by noting that we can compute $\left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{z}^T \mathbf{W}_{\boldsymbol{\theta}} \mathbf{x} \right) \right)_{|\boldsymbol{\theta}_0}$ in $T_1(m,n)$ operations. □

Thus, to efficiently implement gradient descent, we have to efficiently compute $\left( \nabla_{\boldsymbol{\theta}} \left( \mathbf{z}^T \mathbf{W}_{\boldsymbol{\theta}} \mathbf{x} \right) \right)_{|\boldsymbol{\theta}_0}$ for any fixed $\mathbf{z} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$. Next, we will show that the arithmetic complexity of this operation is the same (up to constant factors) as the arithmetic complexity of computing $\mathbf{z}^T \mathbf{W} \mathbf{x}$ (which in turn has complexity no worse than that of computing our old friend $\mathbf{W}\mathbf{x}$). In the next section, not only will we show that this result is true but it is true for *any* function $f : \mathbb{R}^s \to \mathbb{R}$. As a bonus, we will present a simple (but somewhat non-obvious) algorithmic proof.

[2] Here we have assumed that one can compute $g(x)$ with $O(1)$ operations and assumed that $T_2(m,n) \ge m$.

## 4.2 Computing gradients very fast

In this section we consider the following general problem:

- **Input**: Function an arithmetic circuit $\mathcal{C}$ that computes a function $f : \mathbb{F}^s \to \mathbb{F}$ and an evaluation point $\mathbf{a} \in \mathbb{F}^s$.
- **Output**: $\nabla_{\boldsymbol{\theta}} \left(f(\boldsymbol{\theta})\right)_{|\mathbf{a}}$.

Recall that in the previous section, we were interested in solving the above problem for the function $f_{\mathbf{z},\mathbf{x}}(\boldsymbol{\theta}) = \mathbf{z}^T \mathbf{W}_{\boldsymbol{\theta}} \mathbf{x}$ where $\mathbf{W}_\theta \in \mathbb{F}^{m\times n}, \mathbf{z} \in \mathbb{F}^m$ and $\mathbf{x} \in \mathbb{F}^n$.

The way we will tackle the above problem if given the arithmetic circuit $\mathcal{C}$ for $f(\boldsymbol{\theta})$, we will try to come up with an arithmetic circuit $\mathcal{C}'$ to compute $\nabla_{\boldsymbol{\theta}} \left(f(\boldsymbol{\theta})\right)$. We first note that given a fixed $0 \le \ell < s$, it is fairly easy compute a circuit $\mathcal{C}'_\ell$ that on input $\mathbf{a} \in \mathbb{F}^s$ computes $\nabla_{\boldsymbol{\theta}[\ell]} \left(f(\boldsymbol{\theta})\right)_{|\mathbf{a}}$ with essentially the same size (see Exercise 4.1. In particular, the most natural way to do this is to follow a similar argument that we used in the proof of Theorem 2.2.1. This implies that one can compute $\nabla_{\boldsymbol{\theta}} \left(f(\boldsymbol{\theta})\right)$ with arithmetic circuit complexity $O(m \cdot |\mathcal{C}|)$ (where $|\mathcal{C}|$ denotes the size of $\mathcal{C}$).

We will now prove the Baur-Strassen theorem, which states that the gradient can be computed in the same (up to constant factors) arithmetic circuit complexity as evaluating $f$.

**Theorem 4.2.1** (Baur-Strassen Theorem)**.** *Let $f : \mathbb{F}^s \to \mathbb{F}$ be a function that has an arithmetic circuit $\mathcal{C}$ such that given $\boldsymbol{\theta} \in \mathbb{F}^s$ it computes $f(\boldsymbol{\theta})$. Then there exists another arithmetic circuit $\mathcal{C}'$ that computes for any given $\mathbf{a} \in \mathbb{F}^s$, the gradient $\nabla_{\boldsymbol{\theta}} \left(f(\boldsymbol{\theta})\right)_{|\mathbf{a}}$. Further,*

$$|\mathcal{C}'| \le O(|\mathcal{C}|).$$

Before we prove Theorem 4.2.1, we recall the following version of chain rule for multi-variable function.

**Lemma 4.2.2.** *Let $f : \mathbb{F}^s \to \mathbb{F}$ be a function composition of a polynomial $g \in \mathbb{F}[H_1, \dots, H_k]$ and polynomials $h_i \in \mathbb{F}[X_1, \dots, X_s]$ for every $i \in [k]$, i.e.*

$$f(\mathbf{X}) = g\left(h_1(\mathbf{X}), \dots, h_k(\mathbf{X})\right).$$

*Then for every $0 \le \ell < s$, we have*

$$\nabla_{X_\ell}\left(f(\mathbf{X})\right) = \sum_{j=1}^{k} \nabla_{H_j}\left(g(H_1, \dots, H_k)\right) \cdot \nabla_{X_\ell}\left(h_j(\mathbf{X})\right).$$

We note that over $\mathbb{R}$ the above is known as the *high-dimensional chain rule* (and it holds for more general classes of functions). It turns out that if $g$ and $h_i$ are polynomials, then the high-dimensional chain rule pretty much follows from Definition 2.2.2– see Exercise 4.2).

We are now ready to prove Theorem 4.2.1:

*Proof of Theorem 4.2.1.* For simplicity, we will assume that $\mathcal{C}$ does not have any division operator.[3]

For notational simplicity, let us denote the inputs to the function $f$ by $\Theta_1, \dots, \Theta_s$. By Proposition 2.2.3, we have that any gate $g$ in the arithmetic circuit corresponds to a multi-variate polynomials $g(\Theta_1, \dots, \Theta_s) \in$

[3] The proof can be extended to include division– see Exercise 4.3.

$\mathbb{F}[\Theta_1,\dots,\Theta_s]$. By overloading notation, let us denote the output gate[4] by $f$ and the corresponding polynomial by $f(\Theta_1,\dots,\Theta_s)$.

Consider an arbitrary gate $g$ and all the gates $h_1,\dots,h_k$ that $g$ feeds into (as an input gate). Then note that we can think of $f$ as

$$f(\boldsymbol{\theta}) = f(h_1(G,\boldsymbol{\theta}),\dots,h_k(G,\boldsymbol{\theta}),\boldsymbol{\theta}),$$

where we think of $G$ as a new variable corresponding to the gate $g$. In other words, consider the new circuit where $g$ is an input gate (and we drop all the wires connecting the input gates of $g$ to $g$).[5]

The main insight if to them apply the (high-dimension) chain rule to (4.2). In particular, note that Lemma 4.2.2 implies that

$$\nabla_g \left(f(\Theta_1,\dots,\Theta_s)\right) = \sum_{i=1}^{k} \nabla_g \left(h_i(g,\Theta_1,\dots,\Theta_s)\right) \cdot \nabla_{h_i} \left(f(\Theta_1,\dots,\Theta_s)\right).$$

In other words, one can compute the derivative of the final output gate with respect to any gate $g$ in terms of the derivative of $f$ with respect to the 'parent' functions of $g$. Thus, if we start from the output gate (and note that $\nabla_f\left(f\right) = 1$) and move *backwards* to the input gate $\Theta_i$, we get $\nabla_{\Theta_i}\left(f\right)$– i.e. the after this process is done the gradient 'resides' at the input gates. Finally, we exploit the fact that $\mathscr{C}$ is an arithmetic circuit to pin down the derivative for parents of $g$ with respect to $g$. In particular, for any $i \in [k]$:

$$\nabla_g \left(h_i\right) = \begin{cases} \alpha & \text{if } h_i = \alpha \cdot g + \beta \cdot \bar{g}_i \\ \bar{g}_i & \text{if } h_i = g \cdot \bar{g}_i, \end{cases} \tag{4.3}$$

where the two inputs to the operation at gate $h_i$ are $g$ and $\bar{g}_i$. The full details are in Algorithm 3. It is not too hard to argue that (i) Algorithm 3 is correct (see Exercise 4.5), (ii) it implicitly defines an arithmetic circuit $\mathscr{C}'$ that computes $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)$ (see Exercise 4.6) and (iii) $|\mathscr{C}'| \le 5|\mathscr{C}|$ (see Exercise 4.7). This completes the proof. □

**Algorithm 3** Back-propagation Algorithm

INPUT: $\mathscr{C}$ that computes a function $f : \mathbb{F}^s \to \mathbb{F}$ and an evaluation point $\mathbf{a} \in \mathbb{F}^s$
OUTPUT: $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)_{|\mathbf{a}}$

```
Let $\sigma$ be an ordering of gates of $\mathscr{C}$ in reverse topological sort with output gate first  ▷ This is possible since the graph of $\mathscr{C}$ is a DAG
WHILE Next gate $g$ in $\sigma$ has not been considered DO
    Let the parent gates of $g$ be $h_1,\dots,h_k$        ▷ $k = 0$ is allowed and implies no parents
    IF $k = 0$ THEN
        $\mathbf{d}[g] \leftarrow 1$
    ELSE
        $\mathbf{d}[g] \leftarrow 0$
        FOR $i \in [k]$ DO
            $\mathbf{d}[g] \leftarrow \mathbf{d}[g] + \nabla_g\left(h_i\right)_{|\mathbf{a}}$        ▷ Use (4.3)
RETURN $(\mathbf{d}[\Theta_i])_{0 \le i < s}$
```

[4] Recall that $f(\boldsymbol{\theta}) \in \mathbb{F}$ so we will have only one output gate.

[5] See Exercise 4.4 for more on this claim.

### 4.2.1 Automatic Differentiation

It turns out that Algorithm 3 can be extended to work beyond arithmetic circuits (at least over $\mathbb{R}$). This uses that fact that the high dimensional chain rule (Lemma 4.2.2) holds for any differentiable functions $g, h_1, \dots, h_k$. In other words, we can consider circuits that compute $f$ where each gate computes a differentiable function of its input. In other words, given a circuit for $f$ with 'reasonable' gates, one can automatically compile another circuit for its gradient. This idea has lead to the creation of the field of *automatic differentiation* (or *auto diff*) and is at the heart of many recent machine learning progress. (In particular, those familiar with neural networks would notice that Algorithm 3 is the well-known *back-propagation algorithm* (and hence the title of Algorithm 3). However, for these notes, we will not need the full power of auto diff in these notes.

## 4.3 Multiplying by the transpose

We first recall the definition of the transpose of a matrix:

**Definition 4.3.1.** The *transpose* of a matrix $\mathbf{A} \in \mathbb{F}^{m\times n}$, denoted by $\mathbf{A}^T \in \mathbb{F}^{n\times m}$ is defined as follows (for any $0 \le i < n, 0 \le j < m$:

$$\mathbf{A}^T[i,j] = \mathbf{A}[j,i].$$

It is natural to ask (since the transpose it so closely related to the original matrix):

**Question 4.3.1.** *Is the (arithmetic circuit) complexity of computing* $\mathbf{A}^T\mathbf{x}$ *related to the (arithmetic circuit) complexity of computing* $\mathbf{Ax}$ *for* every *matrix* $\mathbf{A} \in \mathbb{F}^{m\times n}$*? E.g. are they within* $\tilde{O}(1)$ *of each other?*

We will address the above question in the rest of this section. Before we provide a general answer, let us consider some specific examples.

### 4.3.1 Some examples

Let us consider the matrix $\mathbf{A}_\oplus \in \mathbb{F}_2^{(n+1)\times n}$ corresponding to (1.3). We saw that $\mathbf{Ax}$ for any $\mathbf{x} \in \mathbb{F}^n$ can be computed with $n$ operations over $\mathbb{F}_2$. Now consider $\mathbf{A}_\oplus^T$. For example, here is the transpose for $n = 3$:

$$\mathbf{A}_\oplus^T = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

More generally each row in $\mathbf{A}_\oplus$ with have exactly two ones in it, which again can be computed with $n$ operations over $\mathbb{F}_2$.

In fact, a moment's reflection reveals that for an $s$-sparse matrix $\mathbf{A} \in \mathbb{F}^{m\times n}$, one can compute $\mathbf{A}^T\mathbf{y}$ for any $\mathbf{y} \in \mathbb{F}^m$ with $O(s)$ (linear) arithmetic operations. This matches the bound for $\mathbf{Ax}$ (see Exercises 1.7 and 2.11).

As another general class consider the case when $\mathbf{A} \in \mathbb{F}^{n\times n}$ has rank $r$. Then since $\mathbf{A}^T$ also has rank $r$, it implies that one can compute $\mathbf{A}^T\mathbf{y}$ for any $\mathbf{y} \in \mathbb{F}^n$ with $O(rn)$ linear arithmetic complexity as well.

We should now start thinking about dense full rank matrices that we have seen so far. We start with the discrete Fourier matrix $\mathbf{F}_n$ (recall Definition 1.4.6). In this case note that $\mathbf{F}_n^T = \mathbf{F}_n$ and hence (very trivially) computing $\mathbf{F}_n^T\mathbf{y}$ has the same arithmetic complexity as computing $\mathbf{F}_n\mathbf{x}$.

The next non-trivial example we considered is the Vandermonde matrix (recall Definition 1.4.7). This probably the first example that might give us pause and let us wonder if there is any hope of answering Question 4.3.1 in the affirmative. Indeed, it turns out that one can also compute the multiplication of the transpose of the Vandermonde with an arbitrary vector can be done with arithmetic complexity $O(n\log^2 n)$ (which matches the complexity of $\mathbf{T}\mathbf{x}$ (see Chapter 3)).

And after that close shave, we mention that indeed for all the dense full rank matrix $\mathbf{A}$ that we have considered in these notes so far the arithmetic complexity of computing $\mathbf{A}\mathbf{x}$ is the same (up to some constant factors) as computing $\mathbf{A}^T\mathbf{y}$.

### 4.3.2 Transposition principle

It turns out that the 'co-incidences' of the arithmetic complexity of $\mathbf{A}^T\mathbf{y}$ matching that of $\mathbf{A}\mathbf{x}$ is not a co-incidence. In particular, it turns out that the answer to Question 4.3.1 is an emphatic *yes*:

**Theorem 4.3.1** (Transposition Principle)**.** *Fix a matrix $\mathbf{A} \in \mathbb{F}^{n\times n}$ such that there exists an arithmetic circuit of size $s$ that computes $\mathbf{A}\mathbf{x}$ for arbitrary $\mathbf{x} \in \mathbb{F}^n$. Then there exists an arithmetic circuit of size $O(s+n)$ that computes $\mathbf{A}^T\mathbf{y}$ for arbitrary $\mathbf{y} \in \mathbb{F}^n$.*

The above result was surprising to the author when he first came to know about it. Indeed, the author could have saved more than a year's worth of plodding while working on the paper [1]. For whatever reason, this reason is not as well-known. One of the reasons to write these notes was to shine more light on this hidden gem!

One might wonder if the additive $n$ term in the bound in the transposition principle is necessary: it turns out that it is (see Exercise 4.8).

There exists proofs of the transposition principle that are very structural in the sense that the consider the circuit for computing $\mathbf{A}\mathbf{x}$ and then directly change it to compute a circuit for $\mathbf{A}^T\mathbf{y}$.[6] For these notes we will present a much slicker proof that directly uses the Baur-Strassen theorem 4.2.1. For this the following alternate view of $\mathbf{A}^T\mathbf{y}$ will be very useful (see Exercise 4.9):

$$\mathbf{y}^T\mathbf{A} = \left(\mathbf{A}^T\mathbf{y}\right)^T. \tag{4.4}$$

*Proof of Theorem 4.3.1.* Thanks to (4.4), we will consider the computation of $\mathbf{y}^T\mathbf{A}$ for any $\mathbf{y} \in \mathbb{F}^n$. We first claim that (see Exercise 4.10):

$$\mathbf{y}^T\mathbf{A} = \nabla_{\mathbf{x}}\left(\mathbf{y}^T\mathbf{A}\mathbf{x}\right). \tag{4.5}$$

Note that the function $\mathbf{y}^T\mathbf{A}\mathbf{x}$ is exactly the same product we have encountered before in Lemma 4.1.1.[7] Then note that given an arithmetic circuit of size $s$ to compute $\mathbf{A}\mathbf{x}$ one can design an arithmetic circuit that computes $\mathbf{y}^T\mathbf{A}\mathbf{x}$ of size $s+O(n)$ (by simply additionally computing $\langle \mathbf{y}, \mathbf{A}\mathbf{x}\rangle$, which takes $O(n)$ operations.).

Now, by the Baur-Strassen theorem, there is a circuit that computes $\nabla_{\mathbf{x}}\left(\mathbf{y}^T\mathbf{A}\mathbf{x}\right)$ with arithmetic circuit of size $O(s+n)$.[8] Equation (4.5) completes the proof. □

[6] At a very high level this involves "reversing" the direction of the edges in the DAG corresponding to the circuit.

[7] However, earlier we where taking the gradient with respect to (essentially) $\mathbf{A}$ whereas here it is with respect to $\mathbf{x}$.

[8] Here we consider $\mathbf{A}$ as given and $\mathbf{x}$ and $\mathbf{y}$ as inputs. This implies that we need to prove the Baur-Strassen theorem when we only take derivatives with respect to part of the inputs– but this follows trivially since one can just read off $\nabla_{\mathbf{x}}\left(\mathbf{y}^T\mathbf{A}\mathbf{x}\right)$ from $\nabla_{\mathbf{x},\mathbf{y}}\left(\mathbf{y}^T\mathbf{A}\mathbf{x}\right)$.

## 4.4 Other matrix operations

Given that we have shown that the arithmetic circuit complexity of computing $\mathbf{Ax}$ is essentially the same as $\mathbf{A}^T\mathbf{y}$, one should get greedy and ask for what other matrices $\mathbf{B}$ related to $\mathbf{A}$ can we show that the arithmetic circuit complexity of computing $\mathbf{Bz}$ is essentially the same as computing $\mathbf{Ax}$.

A very natural such related matrix is the inverse of a (full rank) matrix:

**Definition 4.4.1.** Let $\mathbf{A} \in \mathbb{F}^{n\times n}$ be a full rank matrix (i.e. with rank $n$). Then $\mathbf{A}^{-1}$ is the unique matrix in $\mathbb{F}^{n\times n}$ such that

$$\mathbf{A}\cdot\mathbf{A}^{-1} = \mathbf{A}^{-1}\cdot\mathbf{A} = \mathbf{I}_n,$$

where $\mathbf{I}_n \in \mathbb{F}^{n\times n}$ is the identity matrix.

Matrix inverse is a very useful matrix related to $\mathbf{A}$ and has numerous applications. We present one such example. Consider the problem of solving $n$ linear equations equations, which can be represented as solving for

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{y},$$

where $\mathbf{y}$ and $\mathbf{A}$ are given and form the linear equations in the variables in the unknown vector $\mathbf{x}$. A (unique) solution of $\mathbf{x}$ exists if and only if $\mathbf{A}$ is full rank and is given by

$$\mathbf{x} = \mathbf{A}^{-1}\cdot\mathbf{y}.$$

Thus, given $\mathbf{A}$ is the very useful to be able to quickly compute $\mathbf{A}^{-1}\mathbf{y}$ for arbitrary $\mathbf{y} \in \mathbb{F}^n$.

With the motivation out of the way, we return to our greedy ways and we ask the natural version of Question 4.3.1:

**Question 4.4.1.** *Let* $\mathbf{A} \in \mathbb{F}^{n\times n}$ *be of full rank. Is the (arithmetic circuit) complexity of computing* $\mathbf{A}^{-1}\mathbf{x}$ *related to the (arithmetic circuit) complexity of computing* $\mathbf{Ax}$ *for* every *matrix* $\mathbf{A} \in \mathbb{F}^{m\times n}$*? E.g. are they within an* $\tilde{O}(1)$ *factor of each other?*

The main issue with resolving the above question is that unlike $\mathbf{A}^T$, $\mathbf{A}^{-1}$ seems like a completely different beast than $\mathbf{A}$. For example, $\mathbf{A}$ can be sparse but $\mathbf{A}^{-1}$ need not be (see Exercise 4.11). So even the following question is open:

**Open Question 4.4.1.** *What is the largest class of full rank matrices in* $\mathbb{F}^{n\times n}$ *so that the arithmetic complexity of computing* $\mathbf{A}^{-1}\mathbf{y}$ *is within a* $\tilde{O}(1)$ *factor of the arithmetic circuit complexity of computing* $\mathbf{Ax}$*?*

Partial answers to the above is known and we will see in Chapter 5 some reasonably large class of matrices for which the answer to the above question is yes.

### And the street is littered with unattended matrix operations

There are many matrix operations that are closely related to matrix-vector multiplication but would not even get a mention in these notes. Not because they are not important (both theoretically and/or practically) but because we have kept the focus of these notes on the narrower problem of matrix-vector multiplication.

However, we will conclude this chapter by considering the problem of *matrix-matrix multiplication*:

**Definition 4.4.2.** Given a matrix $\mathbf{A} \in \mathbb{F}^{m \times n}$ and $\mathbf{B} \in \mathbb{F}^{n \times p}$, their product $\mathbf{C} \in \mathbb{F}^{m \times p}$ as follows. For every $0 \le j < p$:

$$\mathbf{C}[:, j] = \mathbf{A} \cdot \mathbf{B}[:, j].$$

The trivial arithmetic complexity of the above problem is $O(mnp)$. The surprising thing[9] is that one can do better for *arbitrary* matrices **A** and **B**. This is an extremely well-studied problem: so much so that when $m = p = n$, the exponent $\omega$ is defined as the smallest constant such that two $n \times n$ matrices can be multiplied in $\tilde{O}(n^{\omega})$ operations. However, we would like to clarify that

The results in these notes do not imply *anything* about the matrix-matrix multiplication problem for *arbitrary* **A** and **B**.

Of course if **A** has an $\tilde{O}(n)$ arithmetic circuit complexity for computing **Ax** for arbitrary then $\mathbf{A} \cdot \mathbf{B}$ for arbitrary **B** has arithmetic circuit complexity of $\tilde{O}\left(n^2\right)$, which is the best possible up to a $\tilde{O}(1)$ factor.[10]

## 4.5 Exercises

*Exercise* 4.1. Given that $f : \mathbb{F}^s \to \mathbb{F}$ can be computed with an arithmetic circuit of size $t$, show that one can compute for any fixed $0 \le i < s$, $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)$ with an arithmetic circuit of size $O(t)$.
*Hint:* Try to compute the answer from the leaves of the circuit from $f$ to its root (like in the proof of Theorem 2.2.1).

*Exercise* 4.2. Prove Lemma 4.2.2.
*Hint:* Use Definition 2.2.2 and the linearity of $\nabla_{X_\ell}(\cdot)$.

*Exercise* 4.3. Prove Theorem 4.2.1 for the most general case when $\mathscr{C}$ also includes the division operator.
*Hint:* See how one can extend the given proof of Theorem 4.2.1.

*Exercise* 4.4. . Prove that $f$ in proof of Theorem 4.2.1 can be re-written as in (4.2).

*Exercise* 4.5. Argue that Algorithm 3 is correct.

*Exercise* 4.6. Argue that Algorithm 3 defines a circuit $\mathscr{C}'$ for $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)$ given a circuit $\mathscr{C}$ for $f$.

*Exercise* 4.7. Let $\mathscr{C}$ and $\mathscr{C}'$ be defined as in Exercise 4.6. Then prove that

$$|\mathscr{C}'| \le 5 \cdot |\mathscr{C}|.$$

*Hint:* Note that to use (4.3) one has to evaluate $f$ itself first.

*Exercise* 4.8. Show that there exists a matrix $\mathbf{A} \in \mathbb{F}^{n \times n}$ such that the arithmetic circuit complexity of **Ax** is zero but the arithmetic circuit complexity of $\mathbf{A}^T\mathbf{y}$ is $\Omega(n)$.
*Hint:* You can assume that the inner product $\langle \mathbf{w}, \mathbf{u} \rangle$ for arbitrary $\mathbf{u}, \mathbf{w} \in \mathbb{F}^n$ has arithmetic circuit complexity of $\Omega(n)$. Can you prove this claim?

*Exercise* 4.9. Prove (4.4).

*Exercise* 4.10. Prove (4.5).

*Exercise* 4.11. Show that there is an $O(n)$-sparse full rank matrix $\mathbf{A} \in \mathbb{F}^{n \times n}$ such that $\mathbf{A}^{-1}$ is $\Omega(n^2)$ sparse.

[9] Well if you have made it so far into the notes, then you already know this so perhaps not as surprising.

[10] This is because the output size of $\Theta(n^2)$.

# Chapter 5

# Combining two prongs of known results

In this chapter, we return to Question 2.4.2, with the goal of talking about results from [1]. In particular, recall the definition of the Chebyshev transform (Definition 2.4.4) and Cauchy matrix (Definition 2.4.3). In both cases, we know how to perform matrix vector multiplication with arithmetic circuit complexity of $\tilde{O}(n)$ (recall Theorem 2.4.4 and 2.4.3). It turns out that Chebyshev polynomials are a special class of *orthogonal polynomials* and Cauchy matrices are special cases of *low displacement rank matrices.* Next we define these two special classes of matrices and then we will see how we can recover algorithms for both with basically the same algorithm.

Before we proceed, it would be a good time to recall the equivalence between polynomials and vectors (Section 1.5.1) and that between matrices and polynomial transforms (Section 1.5.2).

## 5.1 Orthogonal Polynomials

We start with the notion of a family of orthogonal polynomials. There are two (equivalent) definitions of orthogonal polynomials. We begin with the "traditional" definition that also explains the moniker "orthogonal":

**Definition 5.1.1.** We say that a family of polynomials $p_0(X), p_1(X), \ldots$ (over $\mathbb{R}$) where each $p_i(X)$ is a polynomial of degree $i$ is *orthogonal* with respect to the measure $w(X)$ and the interval $[\ell, u]$ if for every pair of integers $i, j \geq 0$:

$$\int_{\ell}^{u} p_i(x) p_j(x) w(x) dx = \delta_{i,j}, \tag{5.1}$$

where $\delta_{i,j}$ is the Kronecker delta function is defined as

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise.} \end{cases}$$

(There are no condition on $w(X)$ other than that (5.1) holds.)

It turns out that the above is equivalent to the following definition (when $\mathbb{F} = \mathbb{R}$):

**Definition 5.1.2.** We say that a family of polynomials $p_0(X), p_1(X), \ldots$ (over $\mathbb{F}$) where each $P_i(X)$ is a polynomial of degree $i$ is *orthogonal* if $p_0(X)$ and $p_1(X)$ are some degree 0 and 1 polynomial respectively. Further for every integer $i \geq 2$, there exists values $a_i, b_i, c_i \in \mathbb{F}$ such that

$$p_i(X) = (a_i X + b_i) \cdot p_{i-1}(X) + c_i \cdot p_{i-2}(X). \tag{5.2}$$

Recall the definition of Chebyshev polynomials (Definition 1.5.7). Note that it fits Definition 5.1.2 with $a_i = 2, b_i = 0, c_i = -1$ for every $i \geq 2$. Further, it turns out that in the language of Definition 5.1.1, Chebyshev polynomials are orthogonal over $[-1,1]$ with respect to the measure $\frac{1}{\sqrt{1-x^2}}$ (see Exercise 5.1).

### 5.1.1 Orthogonal Polynomials transforms

Specializing the definition of polynomial transforms (Section 1.5.2) to the case of an orthogonal polynomial family, we get the following:

**Definition 5.1.3.** Let $p_0(X), \ldots, P_{n-1}(X)$ be the first $n$ polynomials in an orthogonal polynomial family. Let $\alpha_0, \cdots, \alpha_{n-1}$ be $n$ distinct points in $\mathbb{F}$. Then the corresponding *orthogonal polynomial transform* $\mathbf{A} \in \mathbb{F}^{n \times n}$ is defined as follows. For every $0 \leq i, j < n$:

$$\mathbf{A}[i,j] = p_i(\alpha_j).$$

If we define the matrix $\mathbf{P} \in \mathbb{F}^{n \times n}$ to be the matrix where the $i$th row has the coefficients of $p_i(X)$. In other words, if $p_i(X) = \sum_{j=0}^{i} p_{i,j} \cdot X^j$, then

$$\mathbf{P}[i,j] = p_{i,j}.$$

Further, note that then the corresponding orthogonal polynomial transform is given by

$$\mathbf{A} = \mathbf{P} \cdot \left(\mathbf{V}_n^{(\alpha_0,\ldots,\alpha_{n-1})}\right)^T.$$

As we have shown in Chapter 3 that we can compute $\mathbf{V}_n^{(\alpha_0,\ldots,\alpha_{n-1})} \cdot \mathbf{x}$ for any $\mathbf{x} \in \mathbb{F}^n$ with arithmetic circuit complexity$\tilde{O}(n)$, this implies that the arithmetic circuit complexity of computing $\mathbf{Px}$ is the same as that of computing $\mathbf{Az}$ (for arbitrary $\mathbf{x}, \mathbf{z} \in \mathbb{F}^n$) (up to an additive factor of $\tilde{O}(n)$ (see Exercise 5.2).

Due to the above, from now on when we talk about orthogonal polynomial transforms, we will instead talk about the coefficient matrix $\mathbf{P}$. In particular, when we talk about computing the orthogonal polynomial transform (i.e. compute $\mathbf{Ax}$), we will focus on the arithmetic circuit complexity of computing $\mathbf{Pz}$. It turns out that this equivalent view makes it easier to design our algorithms.

### 5.1.2 Fun with roots of orthogonal polynomials

It turns out that the traditional definition of orthogonal polynomials (Definition 5.1.1) has some nice implications, specifically about roots of the $n$'th degree polynomial. We present two of these in this subsection.

However, before we present these results, we first present an immediate consequence of Definition 5.1.1:

**Lemma 5.1.1.** *Let $\{p_i(X)\}_{i \geq 0}$ be a family of polynomials that is orthogonal on $[\ell, u]$ with measure $w(X)$. Fix any $0 \leq d < m$ and let $f(X)$ be any polynomial of degree $d$. Then*

$$\int_{\ell}^{u} f(x) p_m(x) w(x) dx = 0.$$

(See Exercise 5.6.)

### Orthogonal polynomials have distinct real roots

We first argue that an orthogonal polynomial of degree $d$ has $d$ distinct roots in $\mathbb{R}$.

**Lemma 5.1.2.** *Let $\{p_i(X)\}_{i\ge 0}$ be a family of orthogonal polynomials on $[\ell,u]$ with measure $w(X)>0$. Let $d\ge 0$ be any integer. Then $p_d(X)$ has $d$ distinct roots in $[\ell,u]$.*

*Proof.* For the sake of contradiction, let us assume that $p_d(X)$ has distinct roots $a_1,\dots,a_m$ in $[\ell,u]$. If $m=d$ we are done. For the sake of contradiction, let us assume not. This implies that $m<d$. Indeed, if $m>d$, then the degree mantra would be violated. Now consider the polynomial

$$p_d(X)\cdot\prod_{i=1}^{m}(X-a_i). \tag{5.3}$$

We first claim that the polynomial above never changes sign in $[\ell,u]$ (see Exercises 5.7). Since $w(x)>0$ and the polynomial in (5.3) is non-zero, we have

$$\int_{\ell}^{u} p_d(x)\cdot\prod_{i=1}^{m}(x-a_i)\,w(x)\,dx>0.$$

However, the above contradicts Lemma 5.1.1, which means we have $m=n$, as desired. □

### Quadrature

We now state a really neat results about orthogonal polynomials, which allows us to *exactly* represent certain integral involving polynomial by a discrete sum.[1] In particular, we will argue the following:

**Theorem 5.1.3.** *Let $\{p_i(X)\}_{i\ge 0}$ be a family of orthogonal polynomials on $[\ell,u]$ with measure $w(X)>0$. Then for any integer $n\ge 0$ and any polynomial $f(X)$ of degree at most $2n-1$ the following holds. Let $\alpha_0,\dots,\alpha_{n-1}$ be the roots[2] of $p_n(X)$. Then there exists $n$ numbers $w_0,\dots,w_{n-1}$ such that*

$$\int_{\ell}^{u} f(x)\,w(x)\,dx=\sum_{i=0}^{n-1} w_i\cdot f(\alpha_i).$$

*Proof.* Define the weights as follows. For any $0\le i<n$:

$$w_i=\int_{\ell}^{u}\Delta_i(x)\,w(x)\,dx,$$

where $\Delta_i(X)$ is the unique polynomial of degree $n-1$ defined by (see Exercise 5.8):

$$\Delta_i(\alpha_j)=\delta_{i,j}. \tag{5.4}$$

Let $q(X)$ and $r(X)$ be the unique polynomials such that

$$f(X)=p_n(X)q(X)+r(X).$$

Note that $\deg(r(X))\le n-1$. Further, since $\deg(f(X))\le 2n-1$, we also have $\deg(q(X))\le n-1$. Thus, we have

$$\int_{\ell}^{u} f(x)\,w(x)\,dx=\int_{\ell}^{u} p_n(X)q(X)\,w(x)\,dx+\int_{\ell}^{u} r(x)\,w(x)\,dx$$

[1] Quadrature is old-speak for integral.

[2] Note that Lemma 5.1.2 shows that these are all distinct.

$$= \int_{\ell}^{u} r(x)w(x)dx \tag{5.5}$$

$$= \sum_{i=0}^{n-1} r(\alpha_i) \int_{\ell}^{u} \Delta_i(x)w(x)dx \tag{5.6}$$

$$= \sum_{i=0}^{n-1} w_i \cdot f(\alpha_i). \tag{5.7}$$

In the above, (5.5) follows from Lemma 5.1.1 (and the fact that $\deg(q(X)) < n$), (5.6) follows from Exercise 5.9 and (5.7) follows since (where below the second equality uses the fact that $\alpha_i$ is a root of $p_n(X)$):

$$f(\alpha_i) = p_n(\alpha_i)q(\alpha_i) + r(\alpha_i) = r(\alpha_i).$$

Equation (5.7) completes the proof. □

## 5.2 Low displacement rank

We begin with the definition of a matrix having a displacement rank of $r$:

**Definition 5.2.1.** A matrix $\mathbf{A} \in \mathbb{F}^{n\times n}$ has a *displacement rank* with respect to $\mathbf{L}, \mathbf{R} \in \mathbb{F}^{n\times n}$, if the *residual*

$$\mathbf{E} = \mathbf{LA} - \mathbf{AR}$$

has rank $r$.

We first argue that the Cauchy matrix (Definition 2.4.3) has displacement rank 1 with respect to $\mathbf{L} = \text{diag}(s_0,\dots,s_{n-1})$ and $\mathbf{R} = \text{diag}(t_0,\dots,t_{n-1})$, where diag($\mathbf{x}$) is the diagonal matrix with $\mathbf{x}$ on its diagonal. Indeed note that in this case we have $\text{diag}(\mathbf{s})\mathbf{C}_n - \mathbf{C}_n\text{diag}(\mathbf{t})$ is the all ones matrix (see Exercise 5.3).

Further, it turns out that $\mathbf{V}_n^{(\mathbf{a})}$ for any $\mathbf{a} \in \mathbb{F}^n$ has displacement rank 1 with respect to $\mathbf{L} = \text{diag}(\mathbf{a})$ and $\mathbf{R}$ being the *shift* matrix as defined next (see Exercise 5.4):

**Definition 5.2.2.** The *shift* matrix $\mathbf{Z} \in \mathbb{F}^{n\times n}$ is defined by

$$\mathbf{Z}[i,j] = \begin{cases} 1 & \text{if } i = j-1 \\ 0 & \text{otherwise.} \end{cases}$$

(The reason the matrix $\mathbf{Z}$ is called the shift matrix is because when applied to the left or right of a matrix it shift the row (or columns respectively) of the matrix– see Exercise 5.5.) It is known how to compute $\mathbf{Ax}$ with arithmetic circuit complexity $\tilde{O}(rn)$, where $\mathbf{A}$ has displacement rank at most $r$ with respect to $\mathbf{L}, \mathbf{R}$ where these "operators" are either shift or diagonal matrices. We will see later on how to recover these results (as well as prove more general results).

We now return to the matrix vector multiplication problem for the orthogonal polynomial transform case.

## 5.3 Matrix vector multiplication for orthogonal polynomial transforms

Given a matrix $\mathbf{P}$ corresponding to an family of orthogonal polynomials $\left\{p_i(X) = \sum_{j=0}^{i} p_{i,j}X^j\right\}_{i\geq 0}$, we can want to efficiently compute $\mathbf{Px}$ for arbitrary $\mathbf{x} \in \mathbb{F}^n$. Towards this end, we will use the transposition principle (Theorem 4.3.1) it suffices to bound the arithmetic circuit complexity of compute $\mathbf{P}^T\mathbf{y}$. As we shall see shortly this is a reasonably simple problem to solve.

Indeed, the author "wasted" over a year trying to understand how to compute $\mathbf{Px}$ efficiently once he and his co-authors figured out the algorithm for $\mathbf{P}^T\mathbf{y}$. Again, if the transposition principle were known, life would have been much easier.

The first thing we observe is that $\mathbf{P}^T[:.j]$ is the coefficients of $p_j(X)$. In other words we can think of the $j$'th column as the polynomial $p_j(X)$, like so

$$\begin{pmatrix} \uparrow & \cdots & \uparrow & \cdots & \uparrow \\ p_0(X) & \vdots & p_j(X) & \vdots & p_{n-1}(X) \\ \downarrow & \cdots & \downarrow & \cdots & \downarrow \end{pmatrix}.$$

This means that computing $\mathbf{Py}^T$ is the same as computing

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot p_j(X). \tag{5.8}$$

Given the above, here is the obvious algorithm to compute (5.8): However, it is not too hard to check

**Algorithm 4** Naive Algorithm to compute $\mathbf{P}^T\mathbf{y}$

```
INPUT: $\mathbf{y} \in \mathbb{F}^n, p_0(X), p_i(X)$ and $a_i, b_i, c_i \in \mathbb{F}$ for every $1 < i < n$
OUTPUT: $\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot p_j(X)$

$q(X) \leftarrow \mathbf{y}[0] \cdot p_0(X) + \mathbf{y}[1] \cdot p_1(X)$
FOR $2 \le i < n$ DO
    $p_i(X) \leftarrow (a_i \cdot X + b_i) \cdot p_{i-1}(X) + c_i \cdot p_{i-2}(X)$
    $q(X) \leftarrow q(X) + \mathbf{y}[i] \cdot p_i(X)$
RETURN $q(X)$
```

that Algorithm 4 has arithmetic circuit complexity $\Theta(n^2)$. We of course want to do better and in particular, (1) we do not want to explicitly generate all the entries of $\mathbf{P}$ and (2) use the fact that the $\{p_i(X)\}_{i \ge 0}$ for an orthogonal family to achieve (1).

For the rest of the chapter we will assume that $n$ is a power of 2. This simplifies some of the expressions while not affecting the asymptotic complexity.

### 5.3.1 The case of $c_i = 0$

To improve upon the complexity of Algorithm 4, let us first consider the special case when $c_i = 0$ for every $i > 1$. In other words for $i \ge 2$, we have

$$p_i(X) = g_i(X) \cdot p_{i-1}(X),$$

where $g_i(X) = a_i X + b_i$ is a linear polynomial.

We will use a divide and conquer strategy. The obvious way to do this with the sum in (5.8) is to divide up the sum into two equal parts, like so:

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot p_j(X) = \sum_{j=0}^{n/2-1} \mathbf{y}[j] \cdot p_j(X) + \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot p_{j+n/2}(X).$$

Note that the first term is *exactly* the same sum as in (5.8) but half the size. Hence, we can just recurse to compute the first sum. The second does not as is look like the first sum. However, for now

assume that $p_{n/2}(X)$ is given as part of the input.

A natural question to ask is what the above buys us. The main observation is that *relative* to $p_{n/2}(X)$, the polynomials $p_{j+n/2}(X)$ satisfy a similar recurrence as the original one but of half the size. In particular, for every $0 \le j < n/2$, define

$$q_j(X) = \frac{p_{j+n/2}(X)}{p_{n/2}(X)}.$$

We first note the following properties of these new polynomials (see Exercise 5.10):

**Lemma 5.3.1.** *We have the following base cases:*

$$q_0(X) = 1 \text{ and } q_1(X) = g_{1+n/2}(X).$$

*Further, for any* $1 < i < n/2$, *we have*

$$q_i(X) = g_{i+n/2}(X) \cdot q_{i-1}(X).$$

Lemma 5.3.1 then suggests the divide and conquer strategy as outlined in Algorithm 5:

**Algorithm 5** RecursiveTransposeSpecial

```
Input: $\mathbf{y} \in \mathbb{F}^n, p_0(X), p_i(X), p_{n/2}(X)$ and $a_i, b_i \in \mathbb{F}$ for every $1 < i < n$
Output: $\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot p_j(X)$

$L(X) \leftarrow$ RecursiveTransposeSpecial($\mathbf{y}[0 : n/2-1], p_0(X), p_1(X), p_{n/4}(X), \{a_i, b_i\}_{1<i<n/2}$)
$R(X) \leftarrow$ RecursiveTransposeSpecial($\mathbf{y}[n.2 : n-1]1, g_{1+n/2}(X), q_{n/4}(X), \{a_{i+n/2}, b_{i+n/2}\}_{1<i<n/2}$)
return $L(X) + p_{n/2}(X) \cdot R(X)$
```

The statement of RecursiveTransposeSpecial is not complexity because for each call to input of size $n$, one needs to compute $p_{n/2}(X)$. However, it turns out that we can compute all of them in a pre-processing step with $\tilde{O}(n)$ complexity (also see Exercise 5.11):

**Lemma 5.3.2.** *Over the entire run of Algorithm 5.3.1 one needs to compute the following product over all* $0 \le \ell < \log n$ *and* $0 \le j < n/2^i$:

$$\prod_{k=0}^{2^\ell} g_{k+j \cdot 2^\ell}.$$

*Further all of these products can be computed in* $\tilde{O}(n)$ *arithmetic circuit complexity.*

Recall from Chapter 3 that we can multiple two polynomials of degree at most $d$ in $\tilde{O}(d)$ complexity. If $T(n)$ denotes the arithmetic circuit complexity of RecursiveTransposeSpecial with input of size $n$, then we have

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \tilde{O}(n), \tag{5.9}$$

which shows the following (see Exercise 5.12):

**Theorem 5.3.3.** *Algorithm 5 is correct and has arithmetic circuit complexity of* $\tilde{O}(n)$.

### 5.3.2 The general case

We now consider the general case when we can also have $c_i \neq 0$. In this case, we have the following recursion

$$p_i(X) = g_{i,1} \cdot p_{i-1}(X) + g_{i,2}(X) \cdot p_{i-2}(X), \tag{5.10}$$

where $g_{i,j}(X)$ for $j \in \{1,2\}$ is of degree at most $j$.[3]

Let us decompose the sum (5.8) as we did in the previous sub-section:

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot p_j(X) = \sum_{j=0}^{n/2-1} \mathbf{y}[j] \cdot p_j(X) + \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot p_{j+n/2}(X).$$

As before the first sum is again the same problem but of size $n/2$ and hence, we can recurse on that part. The intuition in Algorithm 5 was that if we knew $p_{n/2}(X)$ then the second sum is "independent" of the first sum. After a bit of thinking (and staring at (5.10)) the analogous assumption is to assume we know $p_{n/2}(X)$ and $p_{n/2+1}(X)$. For now, let us

assume that $p_{n/2}(X)$ and $p_{n/2+1}(X)$ are part of the input.

However, here we run into a bit of a hitch. For the previous case since all $p_{j+n/2}(X)$ was a multiple of $p_{n/2}(X)$, we divided each of the polynomials in the second sum by $p_{n/2}(X)$. However, now we have that $p_{j+n/2}$ (for any $j > 1$) depends on *both* $p_{n/2}(X)$ and $p_{n/2+1}(X)$ and one cannot "divide" $p_{j+n/2}(X)$ by $p_{n/2}(X)$ and $p_{n/2+1}(X)$. The trick is to think of $p_{n/2}(X)$ and $p_{n/2+1}(X)$ as a unit. This is reminiscent of how one can show that the $n$th Fibonacci number can be computed with arithmetic circuit complexity$O(\log n)$. For the benefit of those who have not seen this before, we walk through that trick below.

#### Computing the $n$th Fibonacci numbers quickly

Recall that the Fibonacci numbers are defined as follows:

$$f_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ f_{i-1} + f_{i-2} & i \geq 2. \end{cases}$$

Note that the recurrence has some similarity to the recurrence in (5.10) (except we are dealing with integers instead of polynomials).

[3] Note that for orthogonal polynomials, we have $\deg(g_{i,2}) = 0$ but our algorithm will seamlessly handle this generalization so we will go with case. Note that we still have $\deg(p_i) \leq i$ and further, we can also insist $\deg(p_i) = i$ if we want but this is not required for the algorithm.

Given an integer $n \geq 0$ it is easy to see that one can compute $f_n$ in $O(n)$ operations. However, one can do better as follows. We first write the recurrence $f_i = f_{i-1} + f_{i-2}$ in the following matrix form:

$$\begin{pmatrix} f_i \\ f_{i-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} f_{i-1} \\ f_{i-2} \end{pmatrix}.$$

Note that the above implies that for $\mathbf{T} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, we have

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \mathbf{T}^{n-1} \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}.$$

In other words, if we can compute $\mathbf{T}^{n-1}$, we can compute $f_n$ with $O(1)$ additional operations. However, one can compute $\mathbf{T}^m$ for any integer $m$ with arithmetic circuit complexity $O(\log m)$ (see Exercise 5.13).

**On our way back to** (5.10)

We now use inspiration from the Fibonacci number computation above and define for every $i \geq 1$, the vector

$$\mathbf{p}_i(X) = \begin{pmatrix} p_i(X) \\ p_{i-1}(X) \end{pmatrix}.$$

Now we can re-write (5.10) equivalently as

$$\mathbf{p}_i(X) = \mathbf{T}_i(X) \cdot \mathbf{p}_{i-1}(X), \tag{5.11}$$

where

$$\mathbf{T}_i(X) = \begin{pmatrix} g_{i,1}(X) & g_{i,2}(X) \\ 1 & 0 \end{pmatrix}.$$

Before we proceed we make the following assumption:

**Assumption 5.3.1.** *Let us assume that for every $i \geq 2$, we have $\mathbf{T}_i(X) = \mathbf{T}(X)$.*

We note that the above is still interesting: e.g. for the Chebyshev polynomial Assumption 5.3.1 is satisfied with

$$\mathbf{T}(X) = \begin{pmatrix} 2X & -1 \\ 1 & 0 \end{pmatrix}.$$

Given the above, note that (5.11) implies that for every $i \geq 2$,

$$\mathbf{p}_i(X) = (\mathbf{T}(X))^{i-1} \mathbf{p}_1.$$

Now note that if we can compute the $2 \times 1$ vector

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X), \tag{5.12}$$

then we can compute (5.8). Now consider the following equality:

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X) = \sum_{j=0}^{n/2-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X) + \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{p}_{j+n/2}(X).$$

As before the first first is the original problem of size $n/2$. For the second sum, note that all of $\mathbf{p}_{j+n/2}(X)$ has the common term of $(\mathbf{T}(X))^{n/2-1}$. More precisely, define a new recurrence as follows

$$\mathbf{q}_1(X) = \mathbf{p}_1(X)$$

and for every $i \geq 2$

$$\mathbf{q}_i(X) = \mathbf{T} \cdot \mathbf{q}_{i-1}(X).$$

Then we have

$$\sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{p}_{j+n/2}(X) = (\mathbf{T}(X))^{n/2-1} \cdot \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{q}_j(X).$$

The above then implies the generalization of Algorithm 5 in Algorithm 6

**Algorithm 6** RECURSIVETRANSPOSE

```
INPUT: $\mathbf{y} \in \mathbb{F}^n, \mathbf{p}_1(X)$ and $\mathbf{T}(x) \in \mathbb{F}[X]^{2\times 2}$ for every $1 < i < n$
OUTPUT: $\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X)$

$L(X) \leftarrow$ RECURSIVETRANSPOSE($\mathbf{y}[0 : n/2-1], \mathbf{p}_1(X), \mathbf{T}(X)$)
$R(X) \leftarrow$ RECURSIVETRANSPOSE($\mathbf{y}[n/2 : n-1], \mathbf{p}_1(X), \mathbf{T}(X)$)
RETURN $L(X) + (\mathbf{T}(X))^{n/2-1} \cdot R(X)$
```

It can be shown that $\mathbf{T}^{n/2-1}(X)$ can be computed with arithmetic circuit complexity$\tilde{O}(n)$ (see Exercise 5.14). This then implies the following (see Exercise 5.15):

**Theorem 5.3.4.** *Algorithm 6 is correct and has arithmetic circuit complexity of* $\tilde{O}(n)$.

### Getting rid of Assumption 5.3.1.

We now get rid of Assumption 5.3.1 and generalize Algorithm 6 to solve the general case of (5.10) or more precisely that of (5.11).

Before we begin we add some notation:

**Definition 5.3.1.** For any $2 \leq \ell \leq r < n$, define

$$\mathbf{T}_{[\ell,r]} = \mathbf{T}_r \cdot \mathbf{T}_{r-1} \cdots \mathbf{T}_\ell.$$

Then note that for any $i > j \geq 1$, we have from (5.11):

$$\mathbf{p}_i(X) = \mathbf{T}_{[j+1:i]} \cdot \mathbf{p}_j(X). \tag{5.13}$$

As before we split up the final sum of interest into two parts:

$$\sum_{j=0}^{n-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X) = \sum_{j=0}^{n/2-1} \mathbf{y}[j] \cdot \mathbf{p}_j(X) + \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{p}_{j+n/2}(X).$$

As before we can recurse directly on the second part. Let us consider the second sum. Note that (5.13) implies that we can re-write the second sum as

$$\sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{p}_{j+n/2}(X) = \sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2] \cdot \mathbf{T}_{[n/2+1:j+n/2]} \cdot \mathbf{p}_{n/2}(X).$$

However, unlike the previous cases we cannot "pull" $\mathbf{p}_{n/2}(X)$ outside of the sum to the "left." However, we can pull it to the "right," like so:

$$\left(\sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2]\cdot\mathbf{T}_{[n/2+1:j+n/2]}\right)\cdot\mathbf{p}_{n/2}(X).$$

However, in the above the recursive sum is not of the same form as what we started out with. There is a simple fix to this. We now consider the sum of 2 × 2 matrices, like so:

$$\sum_{j=2}^{n-1} \mathbf{y}[j]\cdot\mathbf{T}_{[2:j]}, \tag{5.14}$$

where note that the sum start at $j=2$. It is easy to see that we can compute $\mathbf{y}[0]p_0(X)+\mathbf{y}[1]p_1(X)$ with $O(1)$ operations. Further, the sum $\sum_{j=2}^{n-1}\mathbf{y}[j]\cdot p_j(X)$ can be computed from

$$\left(\sum_{j=2}^{n-1} \mathbf{y}[j]\cdot\mathbf{T}_{[2:j]}\right)\mathbf{p}_1(X).$$

We now return to (5.14) and note that it is the same as

$$\sum_{j=2}^{n/2-1} \mathbf{y}[j]\cdot\mathbf{T}_{[2:j]} + \left(\sum_{j=0}^{n/2-1} \mathbf{y}[j+n/2]\cdot\mathbf{T}_{[n/2:j+n/2]}\right)\cdot\mathbf{T}_{[2:n/2-1]}.$$

Nnow note that the two sum in the above are the same problem as the original but of size at mots $n/2$ and hence, we can recurse. In particular, one can generalize Algorithm 6 to handle this more general case (see Exercise 5.16) in $\tilde{O}(n)$ operations.

### 5.3.3 More generalizations

Finally, we present two generalizations of the recurrence in (5.10) that can be handled with algorithms very similar to those that we have considered so far. We mention two here.

**Going from 2 to $t$.**

Consider the following generalization of the recurrence in (5.10) for $i \ge t$:

$$p_i(X) = \sum_{j=1}^{t} g_{i,j}(X)\cdot p_{i-j}(X), \tag{5.15}$$

where deg($g_{i,j} \le j$, deg($p_i(X) = i$ and polynomials $p_0(X),\cdots,p_{t-1}(X)$ are given. It can be shown that the algorithms for the case of $t=2$ can be extended to the general case with $\tilde{O}\left(t^2 n\right)$ arithmetic circuit complexity (and $\tilde{O}(t^{\omega}n)$ pre-processing)– see Exercise 5.17.

**Handling error terms**

Consider the following generalization of (5.15):

$$p_i(X) = \sum_{j=1}^{t} g_{i,j}(X)\cdot p_{i-j}(X) + E_i(X), \tag{5.16}$$

where the matrix whose rows are $E_i(X)$ has rank $r$. It turns out that even this generalization is not too hard to handle: see Exercise 5.16.

## 5.4 Exercises

*Exercise* 5.1. Argue that Chebyshev polynomials as defined in Definition 5.1.2 are orthogonal over $[-1,1]$ with respect to the measure $\frac{1}{\sqrt{1-x^2}}$.

*Exercise* 5.2. Argue that the arithmetic circuit complexity of computing $\mathbf{Px}$ is the same as that of computing $\mathbf{Az}$ (for arbitrary $\mathbf{x},\mathbf{z}\in\mathbb{F}^n$) (up to an additive factor of $\tilde{O}(n)$.

*Exercise* 5.3. Let $\mathbf{C}_n$ be as defined in Definition 2.4.3. Then show that $\text{diag}(\mathbf{s})\mathbf{C}_n-\mathbf{C}_n\text{diag}(\mathbf{t})$ is the all ones matrix.

*Exercise* 5.4. Prove that $\mathbf{V}_n^{\mathbf{a}}$ has displacement rank 1 with respect to $\mathbf{L}=\text{diag}(\mathbf{a})$ and $\mathbf{R}=\mathbf{Z}$.

*Exercise* 5.5. Let $\mathbf{B}=\mathbf{ZA}$ and $\mathbf{C}=\mathbf{AZ}$. Then argue the following:

- $\mathbf{B}[0,:]=\mathbf{0}^T$ and for every $0<i<n$: $\mathbf{B}[i,:]=\mathbf{A}[i-1,:]$.
- $\mathbf{C}[:,n-1]=\mathbf{0}$ and for every $0\le j<n-1$: $\mathbf{C}[:,j]=\mathbf{A}[:,j+1]$.

*Exercise* 5.6. Prove Lemma 5.1.1.
*Hint:* First argue that $\left\{p_i(X)\right\}_{i\ge 0}$ forms a basis for $\mathbb{R}[X]$ and then use that fact.

*Exercise* 5.7. Prove that the polynomial in (5.3) does not change signs in the interval $[\ell,u]$.

*Exercise* 5.8. Give a closed-formed expression for $\Delta_i(X)$ defined by (5.4). Also argue that this polynomial is unique.

*Exercise* 5.9. Let $p(X)$ be any polynomial of degree $d$ over $\mathbb{F}$. Let $\alpha_0,\dots,\alpha_{d-1}\in F$ be $d$ distinct points. Then we have

$$p(X)=\sum_{i=0}^{d-1}p(\alpha_i)\Delta_i(X),$$

where $\Delta_i(X)$ is as in Exercise 5.8.

*Exercise* 5.10. Prove Lemma 5.3.1.
*Hint:* Use induction.

*Exercise* 5.11. Prove Lemma 5.3.2.
*Hint:* Use the fact that two polynomials of degree at most $d$ can be computed with arithmetic circuit complexity$\tilde{O}(d)$.

*Exercise* 5.12. Prove Theorem 5.3.3.

*Exercise* 5.13. Let $\mathbf{T}\in\mathbb{F}^{t\times t}$. Argue that one can compute $\mathbf{T}^m$ with arithmetic circuit complexity $O\left(t^\omega\cdot\log m\right)$. (Here $\omega$ is the matrix-matrix multiplication exponent for $\mathbb{F}$.)
*Hint:* Use repeated squaring.

*Exercise* 5.14. Let $\mathbf{T}(X)\in\mathbb{F}[X]^{t\times t}$, where entry is a polynomial in $\mathbb{F}[X]$ of degree at most $d$. Argue that one can compute $(\mathbf{T}(X))^m$ with arithmetic circuit complexity $\tilde{O}(dmt^\omega)$. (Here $\omega$ is the matrix-matrix multiplication exponent for $\mathbb{F}[X]$, i.e. multiplying two matrices in $(\mathbb{F}[X])^{N\times N}$ can be computed with $O(N^\omega)$ operations over $\mathbb{F}[X]$.)
*Hint:* Use repeated squaring and keep track of the degree of the polynomials in the intermediate results.

*Exercise* 5.15. Prove Theorem 5.3.4.

*Exercise* 5.16. Design an algorithm with $\tilde{O}(n)$ arithmetic circuit complexity the computes the sum in (5.14).

*Exercise* 5.17. Consider the matrix **P** defined by (5.15). Then show that after $\tilde{O}(t^\omega n)$ operations worth of pre-processing, the arithmetic circuit complexity og computing **px** is $\tilde{O}\left(t^2 n\right)$.

*Hint:* Define $\mathbf{p}_i(X)$ to be the vector $\begin{pmatrix} p_i(X) \\ \vdots \\ p_{i-t+1}(X) \end{pmatrix}$.

*Exercise* 5.18. Consider the matrix **P** defined by (5.16). Then show that after $\tilde{O}(t^\omega n)$ operations worth of pre-processing, the arithmetic circuit complexity og computing **px** is $\tilde{O}\left(t^2 n\right)$.

*Hint:* Define $\mathbf{p}_i(X)$ to be the vector $\begin{pmatrix} p_i(X) \\ \vdots \\ p_{i-t+1}(X) \end{pmatrix}$.

# Bibliography

[1] Christopher De Sa, Albert Gu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1060–1079, 2018.

# Appendix A

# Notation Table

| | | |
|---|---|---|
| $\mathbb{F}$ | A field | |
| $\mathbb{R}$ | The field of real numbers | |
| $\mathbb{C}$ | The field of complex numbers | |
| $\iota$ | The imaginary number, i.e. $\iota^2 = -1$ | |
| $\mathbb{F}^n$ | The set of all length $n$ vectors where each entry is from $\mathbb{F}$ | Definiton 1.1.2 |
| $\mathbb{F}^{m\times n}$ | The set of all $m \times n$ matrices where each entry is from $\mathbb{F}$ | Definiton 1.1.2 |
| $\mathbf{A}[i, j]$ | The $(i, j)$th entry of the matrix $\mathbf{A}$ | |
| $\mathbf{A}[i, :]$ | The $i$'th row of $\mathbf{A}$ | |
| $\mathbf{A}[:, j]$ | The $j$'th column of $\mathbf{A}$ | |
| $\mathbf{x}[i]$ | The $i$th entry of the vector $\mathbf{x}[i]$ | |
| $\mathbf{I}_n$ | The $n \times n$ matrix defined as $\mathbf{I}[i, i] = 1$ and zero everywhere else | |
| $\mathbf{A}^T$ | The transpose of $\mathbf{A}$ defined by $\mathbf{A}^T[i, j] = \mathbf{A}[j, i]$ | |
| $\mathbf{A}-1$ | The inverse of $\mathbf{A}$ (if it is full rank) | |
| $\log x$ | Logarithm to the base 2 | |
| $\tilde{O}\big(f(n)\big)$ | family of functions $O\big(f(n) \cdot \log^{O(1)} f(n)\big)$ | |
| $\mathbf{v}$ | A vector | |
| $\mathbf{0}$ | The all zero vector | |
| $P_{\mathbf{f}}(X)$ | The polynomial $\sum_{i=0}^{n-1} \mathbf{f}[i] \cdot X^i$ | |
| $\mathbf{e}_i$ | The $i$th standard vector, i.e. 1 in position $i$ and 0 everywhere else | |
| $\mathbf{v}_S$ | Vector $\mathbf{v}$ projected down to indices in $S$ | |
| $\langle \mathbf{u}, \mathbf{v} \rangle$ | Inner-product of vectors $\mathbf{u}$ and $\mathbf{v}$ | |
| $[a, b]$ | $\{x \in \mathbb{R} \mid a \le x \le b\}$ | |
| $[x]$ | The set $\{1, \ldots, x\}$ | |
| $\mathbb{F}_q$ | The finite field with $q$ elements ($q$ is a prime power) | |
| $\mathbb{F}^*$ | The set of non-zero elements in the field $\mathbb{F}$ | |
| $\mathbb{F}_q[X_1, \ldots, X_m]$ | The set of all $m$-variate polynomials with coefficients from $\mathbb{F}_q$ | |
| $\mathbb{E}[V]$ | Expectation of a random variable $V$ | |
| $\mathbb{1}_E$ | Indicator variable for event $E$ | |
| $\deg(P)$ | Degree of polynomial $P(X)$ | |
| $\mathbb{F}[X]$ | The set of all univariate polynomials in $X$ over $\mathbb{F}$ | |
| $\mathbb{F}[X_1, \ldots, X_m]$ | The set of all univariate polynomials in $X_1, \ldots, X_m$ over $\mathbb{F}$ | |
| $\nabla_{\mathbf{X}}\big(f(X)\big)$ | The vector of partial derivates of $f(\mathbf{X})$ with respect to $X_i$ where $\mathbf{X} = X_1, \ldots, X_m$ | Definition 2.2.3 |

$|\mathscr{C}|$ For an arithmetic circuit $\mathscr{C}$ over $\mathbb{F}$, $|\mathscr{C}|$ is the number of $\mathbb{F}$ operations in $\mathscr{C}$