Computing fast and accurate convolutions

Huon Wilson

Supervisor: Assoc. Prof. Uri Keich

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Statistics

Faculty of Science
University of Sydney

June 2016

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Huon Wilson

Abstract

The analysis of data often models random components as a sum of independent random variables (RVs). These RVs are often assumed to be lattice-valued, either implied by the problem or for computational efficiency. Thus, such analysis typically requires computing, or, more commonly, approximating a portion of the distribution of that sum.

Computing the underlying distribution without approximations falls under the area of exact tests. These are becoming more popular with continuing increases in both computing power and the size of data sets. For the RVs above, exactly computing the underlying distribution is done via a convolution of their probability mass functions, which reduces to convolving pairs of non-negative vectors.

This is conceptually simple, but practical implementations must carefully consider both speed and accuracy. Such implementations fall prey to the round-off error inherent to floating point arithmetic, risking large relative errors in computed results. There are two main existing algorithms for computing convolutions of vectors: naive convolution (NC) has small bounds on the relative error of each element of the result but has quadratic runtime; while Fast Fourier Transform-based convolution (FFT-C) has almost linear runtime but does not control the relative error of each element, due to the accumulation of round-off error.

This essay introduces two novel algorithms for these problems: aFFT-C for computing convolution of two non-negative vectors, and sisFFT for computing p-values of sums of independent and identically-distributed lattice-valued RVs. Through a rigorous analysis of round-off error and its accumulation, both aFFT-C and sisFFT provide control of the relative error similar to NC, but are typically closer in speed to FFT-C by careful use of FFT-based convolutions and by aggressively discarding irrelevant values. Both accuracy and performance are demonstrated empirically with a variety of examples.

Contents

List of Algorithms

Introduction

Scientific analysis routinely requires performing statistical tests to determine the significance of conclusions drawn from observed data. Deriving accurate outcomes from such tests is important, as such testing and its consequences are near unavoidable in the modern world. This is only becoming even more true, especially in computational biology, as increasing amounts of biological data are being collected and analysed.

There is a large variety of possible tests and techniques, with many of them essentially computing a significance level by considering the likelihood of the observed data assuming a certain null hypothesis about the underlying distribution. These tests come in two broad classes: the first uses asymptotic approximations of the underlying distribution and thus generally perform best with large sample sizes and non-extreme observed data, while the second class, exact testing, avoids such approximations by working with the underlying distribution directly to allow obtaining accuracy in all circumstances.

The significance level for the tests is often exactly the tail probability of some random variable X, calculating the probability P of X being larger than some threshold s_0 derived from observed data, that is, $P = P(X \geq s_0)$.

Non-exact tests will approximate the distribution of X and hence P with some other function that can be computed easily. One example is approximating the distribution of the likelihood ratio statistic G^2 by the distribution of $\chi^2(n)$, which can be calculated efficiently [Din92].

On the other hand, an exact test must work with the distribution of X directly, often by computing its probability mass function (pmf). This cannot easily be done for an arbitrary random variable X, but there are certain classes of random variables for which there is a shared approach. One such class often encountered in practice occurs when the random variable X can be expressed as $X = \sum_{i=1}^{L} X_i$, for random variables $X_1, \ldots, X_L$, where the X_i are lattice valued and independent and identically distributed (iid). This class is of particular note because one can use the well-understood convolution operation to compute the pmf of a random variable $U + V$ from the pmfs of the independent random variables U and V. This thus allows us to express the pmf $\boldsymbol{q}$ of X as the convolution of the identical pmfs $\boldsymbol{p}$ of the X_i,

$$\boldsymbol{q} = \underbrace{\boldsymbol{p} * \ldots * \boldsymbol{p}}_{L \text{ times}} = \boldsymbol{p}^{*L}.$$

One example of this is that explored by Keich in [Kei05] and further by Nagarajan, Jones and Keich in [NJK05]. In the setting of computational biology, one may wish to find significant locations and a sequence motif, or a set of similar substrings, within a large set of observed sequences drawn from some alphabet consisting of A letters. This can be done by computing

a series of potential alignments of substrings, and calculating an alignment quality score called the entropy score I for each of those possibilities. The significance of the alignment is estimated by computing a p-value for this score based on the null hypothesis that each of the L elements of the aligned substrings are drawn independently from some multinomial distribution. Defined as such, it turns out that the exact null distribution of I is given by the L-fold convolution $\boldsymbol{p}^{*L}$ for a certain pmf $\boldsymbol{p}$.

This example is interesting because it is amenable to alternate strategies, relying on large deviation theory as well as asymptotic approximations. For example, it can be shown that $2I \to \chi^2(L(A-1))$ in distribution, as the number of aligned substrings approaches ∞ (e.g. [Ric95]). However, this asymptotic result breaks down particularly for test scores near the maximum value, which is precisely the region that is of most interest in this context. As [NJK05] discusses, other approximation schemes break down similarly, so even widely cited and used software designed for the problem, such as MEME [BE94], can compute p-values that are orders of magnitude smaller than the exact value.

This is just one example of why exact tests, and hence exact computations of pmfs, are of interest, but there are many others, both academic [Hir05] and commercial [MP92].

Applying exact tests in practice requires additional care, beyond understanding their theoretical properties. Computers generally manipulate real numbers as approximate floating point numbers, and significant errors in algorithms can be caused by round-off, destroying guarantees of accuracy unless controlled carefully. Convolutions are a good example of this: there are two main choices for computing $\boldsymbol{x} * \boldsymbol{y}$ for non-negative vectors $\boldsymbol{x}$, $\boldsymbol{y}$: the slow, accurate naive convolution (NC), and the faster, but potentially inaccurate Fast Fourier Transform-based convolution (FFT-C).

Consideration of these practical concerns is particularly relevant to fields like bioinformatics, where large data sets means one may perform very many tests, requiring large multiple testing corrections. This requires that even very small p-values be computed accurately, e.g. [BFT04; NJK05; KN06], and thus making direct use of the efficient FFT-C inappropriate in such cases: FFT-C computed approximations of small p-values can be many times larger than the true values. However, one can still create accurate algorithms benefiting from its speed, by understanding its error. The entropy score example above again serves as a demonstration: [NJK05] introduces a novel algorithm for computing the significance of such scores that has guarantees about its accuracy despite being based on FFT-C.

This essay

This essay is divided into 5 chapters, each of which builds on the previous ones, working through computing convolutions accurately to finish with

calculating accurate p-values. The results try to be general without sacrificing clarity: where sensible, complex vectors are preferred over non-negative ones, and these are preferred over pmfs.

Chapter 1 reviews the computational basics required for the rest of the thesis, including floating point numbers, asymptotic notation and the Fast Fourier Transform (FFT).

Chapter 2 reviews and analyses the error in existing algorithms, NC and FFT-C, for computing convolutions $\boldsymbol{x} * \boldsymbol{y}$. This chapter also reviews Keich's sFFT algorithm [Kei05] for computing p-values of iterated convolutions, since the concepts there are vital for later work. This second chapter is broadly a refinement of the work previously performed in [Kei05].

Chapter 3 introduces the novel aFFT-C algorithm for efficiently and accurately computing a pairwise convolution $\boldsymbol{p} * \boldsymbol{q}$ of non-negative $\boldsymbol{p}$ and $\boldsymbol{q}$. This algorithm achieves the best of both FFT-C and NC to ensure accuracy and maintain maximal performance. This chapter also includes an analysis of two ways to incorporate a lower bound into the computation: some applications can tolerate the inaccurate computation of very small values, which can be handled in two ways. This possibility allows doing considerably less work in certain cases.

Chapter 4 is the last chapter focusing on computing convolutions themselves. This chapter builds on the lower bound considerations of the previous chapter to analyse an algorithm for computing approximations to the iterated convolution $\boldsymbol{p}^{*L}$. The algorithm is initially designed to work with any method for computing $\boldsymbol{p}*\boldsymbol{q}$ with the appropriate guarantees. This algorithm is then made more concrete by applying the aFFT-C algorithm also from the previous chapter, which allows us to explore and justify the choices made in that initial design.

Chapter 5 uses the iterated convolution algorithm of the previous chapter to allow computing approximations to a tail probability $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$. As mentioned above, this tail probability is exactly the p-value of a sum of L independent lattice-valued random variables, for pmf $\boldsymbol{p}$. As is the standard in this essay, these approximations come with guaranteed bounds on their accuracy. This algorithm is designed to work with arbitrary pmfs $\boldsymbol{p}$, addressing issues with existing algorithms such as sFFT, which only works well for log-concave pmfs.

The majority of the ideas of Section 2.2.1 and Chapter 3, with the notable exception of the details of Section 3.4, have been published as [WK16]. The core analysis of Chapter 4 and all of Chapter 5 have been improved slightly and submitted as [WK].

The code

The new algorithms aFFT-C and sisFFT introduced in this thesis have been implemented in Python [Ros95] and R [R C15]. The code is available for download at `https://github.com/huonw/sisfft-py` and `https://github.`

com/huonw/sisfft, and has been published as `sisfft` on PyPI, a package repository for Python.

Acknowledgements

Thank you to my supervisor, Uri Keich, who has introduced me to the world of academia, whose advice and suggestions have been almost universally correct and useful, and who did the initial work with sFFT on which I build.

I also thank the Australian Mathematical Sciences Institute, as the Vacation Research Scholarship I received gave me my first introduction to this topic of accurate convolutions and exact p-value calculations.

Notation

The notation used often in this essay is listed here for clarity. Let $\boldsymbol{v}$, $\boldsymbol{w}$ be vectors.

$\boldsymbol{v}$	Bold font indicates a vector.
$\mathrm{rel}(x, y)$	The relative error of x as an approximation for y, see (1.1.3).
$\widetilde{x}$	The value of x as computed via floating point.
ε	The machine epsilon of the floating point type in use, see Definition 1.2.5.
$f = O(g)$, $f = \Omega(g)$	f is asymptotically bounded above (resp. below) by g, see Section 1.4.
D, D^{-1}	The discrete Fourier transform operator (1.5.1) and its inverse (1.5.2).
$\boldsymbol{v} * \boldsymbol{w}$	The convolution of $\boldsymbol{v}$ and $\boldsymbol{w}$, see (2.0.1).
$\boldsymbol{v} *_N \boldsymbol{w}$	The cyclic convolution of $\boldsymbol{v}$ and $\boldsymbol{w}$, when both have length N, see Definition 2.2.1.
$\boldsymbol{v}^{*L}$	The L-fold convolution of $\boldsymbol{v}$ with itself, $\underbrace{\boldsymbol{v} * \ldots * \boldsymbol{v}}_{L \text{ times}}$.
$\boldsymbol{v} \odot \boldsymbol{w}$	The element-wise product of $\boldsymbol{v}$ and $\boldsymbol{w}$, that is, $(\boldsymbol{v} \odot \boldsymbol{w})(i) = \boldsymbol{v}(i)\boldsymbol{w}(i)$, when $\boldsymbol{v}$ and $\boldsymbol{w}$ have the same length.
$\boldsymbol{v}^{\odot x}$	The element-wise exponentiation of $\boldsymbol{v}$ with by x, that is, $(\boldsymbol{v}^{\odot x})(i) = \boldsymbol{v}(i)^x$.
$\min_+ \boldsymbol{v}$	The smallest positive value in $\boldsymbol{v}$, see Section 3.1.3.
$\mathrm{filt}_{\boldsymbol{v},\boldsymbol{w}}(\boldsymbol{x})$	Filtering $\boldsymbol{x}$ as an FFT-C approximation for $\boldsymbol{v} * \boldsymbol{w}$, see (3.1.4).
1_f, $\mathbf{1}_{\boldsymbol{v}}$	The indicator function/vector for the function f/vector $\boldsymbol{v}$.
$\operatorname{supp} f$, $\operatorname{supp} \boldsymbol{v}$	The support of the function f/vector $\boldsymbol{v}$.
$\boldsymbol{v}_{<B}$, $\boldsymbol{v}_{\geq B}$	The vectors truncated above or below some bound B, that is, $\boldsymbol{v} \odot \mathbf{1}_{\boldsymbol{v}<B}$ and $\boldsymbol{v} \odot \mathbf{1}_{\boldsymbol{v}\geq B}$. See Definition 3.4.1.

Chapter 1

Computational Background

1.1. Errors

One of the key parts of working with and analysing numeric algorithms is understanding the error that they introduce, that is, the distance between the exact answer and the value computed. A perfect algorithm will compute the exact answer, but this is often either impossible or requires too much time or memory, and so one is usually working with approximations, and thus controlling or at least knowing the degree of approximation is very important.

For scalars there are two common measures of errors.

Definition 1.1.1. *Let $\widetilde{x}$ be an approximation to $x \in \mathbb{C}$. The* absolute error *is*

$$\left|\widetilde{x} - x\right|, \tag{1.1.2}$$

and the relative error *is*

$$\operatorname{rel}(\widetilde{x}, x) = \begin{cases} \left|\frac{\widetilde{x}-x}{x}\right| & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \text{ and } \widetilde{x} = 0 \\ \infty & \text{if } x = 0 \text{ and } \widetilde{x} \neq 0. \end{cases} \tag{1.1.3}$$

Measuring the error in an approximation of a vector is more complicated. For many applications, using the ℓ^p absolute and relative errors, for some $p \geq 1$, is sufficient and desirable. These are defined by simply replacing the absolute values in the scalar definition of absolute and relative error by the vector's p-norm, such as

$$\frac{\left\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\right\|_p}{\left\|\boldsymbol{x}\right\|_p}. \tag{1.1.4}$$

Typical choices are $p = 2$ or $p = \infty$.

However, these measures tend to summarise the error in the vector based on the largest values of $\boldsymbol{x}$, and there can be arbitrarily large *relative* errors in the approximation $\widetilde{\boldsymbol{x}}$ of the smallest elements of $\boldsymbol{x}$.

Example 1.1.5. Consider the vector $\boldsymbol{x} = (1000, 0)$ and an approximation $\widetilde{\boldsymbol{x}} = (1000, 1)$, the approximation has an ℓ^p relative error of just $1/1000$ for all $p \geq 1$, but the (scalar) relative error in the second element is ∞. □

In some applications we are only interested in parts of the computed vectors, for example, when computing tail probabilities, a special case of which is computing a p-value. These regions often do not include the large values that have the most influence over the ℓ^p error, and thus a small value for a measure like (1.1.4) does not necessarily correspond to fine control over

the error in the final computation. For such applications, the relative error in the individual elements is more useful, as it ensures that small values that are of interest are still computed accurately. In other words, controlling something like the following quantity is desirable,

$$\max_k \operatorname{rel}(\widetilde{\boldsymbol{x}}(k), \boldsymbol{x}(k)).$$

In practice, the exact condition may be more precise, for instance, maybe the relative error only needs to be controlled for k such that $\boldsymbol{x}(k) \geq B$ for some bound B, or maybe only for $k \geq k_0$, for some minimum index k_0.

Focusing on the relative error of elements automatically gives some measure of control over the ℓ^p error. Specifically, if $\operatorname{rel}(\widetilde{\boldsymbol{x}}(k), \boldsymbol{x}(k)) \leq \beta$ for all k, then, it is easy to see that

$$\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_p \leq \beta \|\boldsymbol{x}\|_p . \tag{1.1.6}$$

However, in many cases, including the algorithms introduced in this essay, the relative error in the largest elements of $\boldsymbol{x}$ is much smaller than that in the smallest ones, and so the actual ℓ^p errors can be much smaller than (1.1.6) implies.

There are two main classes of the cause of errors in numerical algorithms:

- *truncation error* or *algorithmic error*: error introduced by differences between the algorithm and the true mathematical result, for instance, approximating an infinite sum with a finite truncation: $\sum_i^\infty f(i) \approx \sum_i^N f(i)$.
- *rounding error* or *round-off error*: error introduced by rounding intermediate results.

Truncation error is very specific to the algorithm in question, while rounding error is pervasive, typically caused by the use of limited precision floating point numbers to approximate the full precision value, a subject we will dwell on next.

1.2. Floating Point

A computer is inherently limited: it can only store a finite amount of information in memory, and it can only do finitely many operations to process that information. Hence sets like $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ cannot be represented perfectly in their entirety, forcing us to resort to digital approximations of these sets, each with their own benefits, limitations and trade-offs.

One of the most common digital representation of a real number is floating-point, and, most usually, the `binary32` and `binary64` types[1] of the IEEE754-2008 standard [IEE08]. These types have widespread support in hardware and hence can be assumed to offer high performance. Additionally, the standard provides mathematically rigorous definitions of its types, so we can prove theorems about algorithms which apply to useful real-world

[1] `binary32` is also known as `single` or just `float`, and `binary64` as `double` in previous versions of the IEEE754 standard, and in many programming languages.

implementations running on most platforms. We will briefly look at the relevant properties of these formats, referring the reader to [Gol91], [BZ10] and [MBdD+10] for more detailed treatments.

Floating point formats use the idea behind "scientific notation": fix a base $B \in \mathbb{R}_+$ and write $x \in \mathbb{R}$ as

$$x = s \cdot m \cdot B^e \tag{1.2.1}$$

for $s \in \{-1, 1\}$, $m \in \mathbb{R}_+$ and $e \in \mathbb{Z}$.

As written, this encoding has problems: it is not unique, and allowing $m \in \mathbb{R}$ arbitrary requires a way to represent real numbers, which is exactly the problem we are trying to solve. For $x \neq 0$, these can be resolved by, respectively, restricting m to $[1, B)$, and quantising it with some precision $p \in \mathbb{N}_+$ via the approximation $m \approx m' B^{-p+1}$ with $m' \in \mathbb{N}$ satisfying $B^{p-1} \leq m' < B^p$. The case $x = 0$ needs to be considered separately due to the $m \in [1, B)$ restriction.

Definition 1.2.2. *Fix some $B \in \mathbb{R}_+$ and $p \in \mathbb{N}$ with $p \geq 2$, if $x = s \cdot m \cdot B^e \neq 0 \in \mathbb{R}$ is approximated as a* floating-point number *$\widetilde{x} = s \cdot m' B^{-p+1} \cdot B^e$, then*

- *B is the* radix *(sometimes* base*),*
- *p is the* precision*,*
- *$s \in \{-1, 1\}$ is the* sign *of x,*
- *$m \in [1, B)$ is the* infinitely precise significand *of x,*
- *$m' \in \mathbb{N}$ satisfying $B^{p-1} \leq m' < B^p$ is the* integral significand*, and $m' B^{-p+1}$ is the* significand *(sometimes* mantissa*) of $\widetilde{x}$,*
- *$e \in \mathbb{Z}$ is the* exponent *of x.*

Since computers and silicon circuits work in binary most easily, the radix is most commonly $B = 2$ (as in `binary32` and `binary64`). Beyond this, the radix is almost always either power of 2 or a power of 10 (IEEE754-2008 also specifies radix-10 types like `decimal32`). This thesis will focus on the $B = 2$ case.

For an arbitrary $x \in \mathbb{R}$ the floating point quantised form $\widetilde{x}$ will only be an approximation, but the error in the approximation can be controlled.

Proposition 1.2.3. *Fix a radix B and precision p, then for any $x \neq 0 \in \mathbb{R}$ there exists a floating point number $\widetilde{x} = s \cdot m' B^{-p+1} \cdot B^e$ as in Definition 1.2.2 with relative error bounded by $B^{-p+1}/2$, that is,*

$$\left| \frac{\widetilde{x} - x}{x} \right| \leq \frac{B^{-p+1}}{2}$$

Proof. Write $x = s \cdot m \cdot B^e$ for some $m \in [1, B)$, let $m^* = m B^{p-1}$ and denote $[m^*]$ the closest integer to m^*. This integer satisfies $B^{p-1} \leq [m^*] \leq B^p$. The precise direction of rounding when m^* is a half-integer is not important, since any choice will satisfy $|m^* - [m^*]| \leq \frac{1}{2}$.

If $[m^*] < B^p$, choose $m' = [m^*]$, that is, define $\widetilde{x} = s \cdot [m^*]B^{-p+1} \cdot B^e$, and hence

$$\left|\frac{\widetilde{x} - x}{x}\right| = \frac{1}{m}\left|[m^*]B^{-p+1} - m\right| = \frac{1}{m}\left|[m^*] - m^*\right| B^{-p+1} \leq \frac{B^{-p+1}}{2}$$

If $[m^*] = B^p$, then taking $m' = [m^*]$ is not valid, so we instead choose $m' = B^{p-1}$ and use a larger exponent, that is

$$\widetilde{x} = s \cdot B^{p-1}B^{-p+1} \cdot B^{e+1}$$

In this case, we have

$$\left|\frac{\widetilde{x} - x}{x}\right| = \frac{1}{m}\left|m'B - m^*\right| B^{-p+1} = \frac{1}{m}\left|[m^*] - m^*\right| B^{-p+1}$$

and the inequality above still applies. □

With an exact representation of 0, one can thus represent all $x \in \mathbb{R}$ with bounded relative error, and some can even be represented exactly.

Example 1.2.4. Fix the radix $B = 10$ and the precision $p = 5$, the closest floating point number to $x = \exp(\pi) = 23.14069\ldots$ is

$$\begin{aligned}\widetilde{x} &= 23.141\\ &= 1 \cdot 2.3141 \cdot 10^1\\ &= 1 \cdot 23141 \cdot 10^{-4} \cdot 10^1.\end{aligned}$$

That is, the sign is 1, the significand is 2.3141 and the exponent is 1. This $\widetilde{x}$ approximates x with relative error less than $\frac{1}{2}10^{-4}$ (indeed, the actual relative error is approximately $1.3 \cdot 10^{-5}$). □

The error bound is core to guarantees about floating point values, and features in most theorems manipulating them.

Definition 1.2.5. *The quantity* $\varepsilon = \frac{B^{-p+1}}{2}$ *is the* machine epsilon[2] *or* unit round-off *of the floating point system.*

The `binary32` format has precision $p = 24$ and hence $\varepsilon = 2^{-24}$, while the format we usually use, `binary64`, has $p = 53$ and thus $\varepsilon = 2^{-53} \approx 10^{-16}$.

Unfortunately, there are limitations to this result: it relies on the exponent e being unbounded, which is impossible in reality; and, it only applies for a single approximation step. If one is performing a series of operations where each returns a floating point approximation to the actual result, there is no guarantee that the overall result will be close to the true value of the computation performed with infinite precision. We will look at these two deficiencies, starting with the second.

[2]In some domains, the "machine epsilon" term and ε symbol refer to the quantity 2ε, that is, B^{-p+1}. This essay uses only $\varepsilon = B^{-p+1}/2$.

1.2.1. Round-off errors and catastrophic cancellation

Floating point formats have finite precision, and hence, as quantified by Proposition 1.2.3, information is lost as they are processed. This loss is inevitable, but it has unfortunate consequences for floating point as a model of $\mathbb{R}$: addition and multiplication are not associative, and errors can compound over the course of a computation, possibly resulting in arbitrarily large error.

Definition 1.2.6. Round-off error *occurs when the result of operation performed with finite-precision numbers differs to the result of the same operation performed in $\mathbb{R}$ (i.e. with infinite precision).*

Round-off error can occur even on primitive operations like addition and multiplication. For instance, if two numbers have different exponents, some information in the significand of that with the smallest exponent may be lost when they are added.

Example 1.2.7. Take the floating point system with $B = 2$ and $p = 2$, and compute $(1 + 16) - 16$. First $1 + 16 = 17 = 1.001 \cdot 2^4$, which rounds to $1.0 \cdot 2^4 = 16$, and hence the result as computed via floating point is 0, which has 100% relative error from the true answer of 1. □

This example demonstrates round-off error in the addition and also *catastrophic cancellation* in the subtraction. The latter happens when two numbers with very similar values are subtracted (or divided), which can dramatically amplify the relative error of any previous round-off error that has occurred.

Round-off error is unfortunate, but it is an unavoidable consequence of computers' finite precision. However, if one understands the behaviour of a floating point system, this source of error can be controlled and accounted for, allowing algorithms to compute approximate results with floating point while guaranteeing a bound on the error of these approximations.

The IEEE754 standard tackles this by specifying exactly how information is lost for the basic operations like arithmetic. Let $f : \mathbb{R}^n \to \mathbb{R}$ be the infinitely precise version of any of these operations. For floating point numbers $\widetilde{x_1}, \ldots, \widetilde{x_n}$ the IEEE754 approximation $\widetilde{f}$ to f is defined as the closest representable floating point number to $f(\widetilde{x_1}, \ldots, \widetilde{x_n})$.

Example 1.2.7 demonstrates floating point approximations to $f(x, y) = x + y$ and $g(x, y) = x - y$, with the application of the former requiring rounding, and that of the latter being exact. It is worth noting that the operation was computed as $\widetilde{g}(\widetilde{f}(1, 16), 16)$, with the intermediate rounding of $\widetilde{f}$ meaning the overall result is not the same as the closest floating point number to $g(f(1, 16), 1)$.

For many functions, we can efficiently compute the closest number to the infinitely precise result assuming their input is error-free or perfectly representable. Indeed, with only a small amount of extra precision over that

of the input, it is generally straight-forward to deduce the correctly rounded result of arithmetic operations and even more complicated ones like square roots. Unfortunately, this ease does not translate to all common functions. For example, there are no tight bounds on the extra precision required for correctly rounding floating point approximations to functions like exp and log, a fact called the Table-Maker's Dilemma [Kah04]. The interested reader should refer to [MBdD+10] or [BZ10] for more details on this aspect of floating point arithmetic.

1.2.2. Underflow and Overflow

The other problem with applying Proposition 1.2.3 in practice is that it relies on the exponent e being unbounded, which is of course impossible to implement on real hardware. Hence, floating point formats will limit the exponent to some interval $e_{\min} \leq e \leq e_{\max}$. If the format has radix B and precision p, these bounds on e mean Proposition 1.2.3 is only guaranteed to hold for real numbers x such that

$$min := B^{e_{\min}} \leq |x| \leq (B - B^{-p+1})B^{e_{\max}} =: max.$$

If $|x| < min$, the approximation may *underflow* to $\widetilde{x} = 0$, with relative error 1. If $|x| > max$, the approximation may *overflow* giving the special value $\widetilde{x} = \pm\infty$ as appropriate, with infinite relative error. Floating point formats include the values ∞ and $-\infty$ for this purpose, to have a somewhat reasonable interpretation of calculations that get too large.

Example 1.2.8. Consider a floating point format with $B = 2$, $p = 2$ and limits $-5 \leq e \leq 5$. If $x = 1.0 \cdot 2^{-3}$ and $y = -1.0 \cdot 2^{3}$, then

$$\widetilde{x/y} = \widetilde{-1.0 \cdot 2^{-6}} = 0,$$
$$\widetilde{y/x} = \widetilde{-1.0 \cdot 2^{6}} = -\infty.$$ □

One can extend the region of representable values closer to zero by relaxing the definition of a floating point format for small values. Specifically, by allowing the integral significand to be less than B^{p-1} when $e = e_{\min}$, the format can represent values as small as B^{e-p+1} while still ensuring that each representable value has a unique representation. These are called *subnormal* values, in contrast to the *normal* values in the regions $[-max, -min] \cup [min, max]$. However, the existence of these subnormal values does not guarantee Proposition 1.2.3 applies for all x as small as B^{e-p+1}, and furthermore, they are typically not suitable for practical applications: they are much slower to compute with than normal values, and are sometimes just truncated to zero.

For the types specified by IEEE754-2008 the corresponding bounds are: $e_{\max} = -126$ and $e_{\max} = 127$ for `binary32`, and $e_{\min} = -1022$ and $e_{\max} = 1023$ for `binary64`. Hence, the magnitudes of the corresponding normal values vary between 10^{-38} to 10^{38} and 10^{-308} to 10^{308} respectively.

If either underflow or overflow occurs during an operation, the relative error dramatically increases to 1 or ∞ instead of, typically, some small polynomial of ε. As such, most results involving floating point either explicitly or implicitly include an assumption that underflow and overflow does not occur. All theorems in Chapters 2 to 4 make the assumption that the floating point numbers involved are all sufficiently moderate.

This is a reasonable assumption to make in most cases, especially for `binary64`: 10^{-308} and 10^{308} are far more extreme than most numbers encountered in reality. However, at times we need to process more extreme values, and hence mitigations become important.

One way to postpone underflow and overflow is to move to a type with a larger exponent range, however this typically only delays the issue slightly. Moving from `binary32` to `binary64` or `binary128` only increases the exponent from 8 bits to 11 or 15, respectively. This small increase means that calculations that underflow or overflow in one format are likely to quickly do the same in a larger format, especially as extreme values typically occur when performing multiplications or computing exponentials. For example, let $x = 3.403 \cdot 10^{38}$, this is just larger than the largest value in the `binary32` format and hence overflows in that format, so that $\widetilde{x} = \infty$. On the other hand, x does not overflow in the `binary64` format, but it only takes eight further multiplications to exceed the largest value, that is, $\widetilde{x^8} = \infty$. Fortunately, domains that require extreme range often have extra assumptions that can inform a choice of an alternative representation.

1.3. Log space

A commonly occurring scenario is that inputs and outputs to an algorithm will be non-negative, which allows for representing numbers in *log space*: an $x \geq 0 \in \mathbb{R}$ is stored as a floating point approximation to $X = \log_b x$ for some base b, with $x = 0$ of course represented by $-\infty$. This gives *much* larger range: if `binary64` is used to store the natural logarithm, then $x > 0$ can range from approximately $10^{-7 \cdot 10^{307}}$ to $10^{7 \cdot 10^{307}}$. This range comes at the expense of some operations becoming much more time consuming, and offering less precision at the extremes.

Manipulating these values can be somewhat non-trivial, as simply reversing the logarithm may underflow or overflow. For whatever base b is chosen, arithmetic can be performed on these values in a way that avoids catastrophic underflow and overflow. Suppose $x, y \geq 0$ and $X = \log_b x$, $Y = \log_b y$, then

- Addition: $x \geq y$, $\log_b(x + y) = X + \log_b(1 + b^{Y-X})$
- Subtraction: $x \geq y$, $\log_b(x - y) = X + \log_b(1 - b^{Y-X})$
- Multiplication: $\log_b(x \cdot y) = X + Y$
- Division: $\log_b(x/y) = X - Y$

Underflow or overflow in multiplication or division represents the true value xy or x/y exceeding the bounds of the log space representation, which

is unfortunate but unavoidable. The addition and subtraction formula factor out the largest term from the naive definition

$$\log_b(x+y) = \log_b\left(b^X + b^Y\right) = \log_b\left[b^X\left(1 + b^{Y-X}\right)\right],$$

which ensures that the intermediate results never overflow or underflow in a catastrophic way. There may be underflow when computing b^{Y-X}, but, as long as the exponent range is sufficiently large, this can only occur when y is so much smaller than x that the finite precision means $\widetilde{x+y} = x$ anyway.

1.4. Asymptotic notation & computational complexity

When analysing algorithms, understanding their accuracy is not the only factor of importance: as mentioned above, some exact algorithms are very expensive, often to the point of being prohibitively so for everything but very small instances of a problem. Being able to quantify the performance of an algorithm is thus also useful.

This is typically done via asymptotic notation, which captures the number of operations (or some other proxy of resource demand, such as memory use rather than time) required by the algorithm as the parameters of the problem increase.

Definition 1.4.1. *Given functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, we say $f = O(g)$ if*

$$\limsup_{x\to\infty} \frac{f(x)}{g(x)} < \infty \tag{1.4.2}$$

and $f = \Omega(g)$ if $g = O(f)$.

In less precise terms, $f = O(g)$ when f does not grow faster than some constant multiple of g, and $f = \Omega(g)$ when f grows at least as fast as some constant multiple g. For instance, let $f(x) = 3x^2 + 2$, then $f = O(x^2)$, and $f = O(x^3)$, and even $f = O(e^x)$, however $f \neq O(x)$ nor $f \neq O(\log x)$. As just demonstrated, we will typically write the function g implicitly.

In computational terms, algorithms with resource cost linear in the size of the problem n, that is, $O(n)$, are highly desirable: anything much more than this, even simply quadratic $O(n^2)$, can quickly require infeasible amounts of resources as the problem size increases. Unfortunately, many problems either have no known linear algorithm or, worse, are known to have no linear algorithm. Not all is lost, as some of these problems can be solved by algorithms with complexity $O(n(\log n)^c)$ for some constant c, typically $c = 1$. Such algorithms are sufficiently efficient for most purposes, and are thus often described as "almost linear".

The $f = O(g)$ notation is sometimes reused for small values of the parameter to f and g, where the limit is changed to approach 0,

$$\limsup_{x\to 0} \frac{f(x)}{g(x)} < \infty$$

The target of the limit should be understood from context, based on whether the parameter is large or small. For instance, $O(\varepsilon)$ often appears, eliding terms like $\varepsilon + \varepsilon^2$, when they are considered to be so small as to be irrelevant.

1.5. Fast Fourier transform (FFT)

The discrete Fourier transform (DFT) D and its inverse D^{-1} are linear operators $\mathbb{C}^N \to \mathbb{C}^N$ that decompose a complex vector into a Fourier series, or reconstruct it from its Fourier series. These operators (and their continuous versions) are the cornerstones of many theoretical results and practical applications, from differential equations and quantum mechanics to arbitrary precision arithmetic and photo manipulation.

There are several definitions of the DFT, differing in if/where the vectors are normalised. This essay uses a common definition, given by

$$(D\boldsymbol{v})(k) = \sum_{j=0}^{N-1} \boldsymbol{v}(j) e^{-2i\pi kj/N} \tag{1.5.1}$$

$$(D^{-1}\boldsymbol{w})(k) = \frac{1}{N} \sum_{j=0}^{N-1} \boldsymbol{w}(j) e^{2i\pi kj/N} \tag{1.5.2}$$

where $k = 0, \ldots, N-1$.

A major reason for the widespread use of the DFT is that it can be evaluated in almost linear time, making it the building block for efficient versions of other algorithms.

More precisely, evaluating the definitions of the operators as written above takes $O(N^2)$ operations, but they can also be computed via the Fast Fourier Transform (FFT). The FFT takes only $O(N \log N)$ operations, which makes it applicable and usable even for large N. For instance, the FFTW3 library [FJ05] can perform a DFT of length $2^{25} \approx 34 \cdot 10^6$ in less than a second on a modern CPU[3].

The FFT was first demonstrated by Gauss [HJB85], but it only became widely known and popular much later. That popularity was driven by Cooley and Tukey [CT65] who demonstrated a general purpose Fast Fourier Transform designed for calculation with a computer and which was applicable vectors with lengths factoring into only small prime factors.

[3]An Intel i7-4900MQ.

Chapter 2

Existing Convolution Algorithms

The convolution operation is an important component of many theoretical results and practical applications, and thus algorithms to compute it are valuable. The convolution of two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ is defined as the vector $\boldsymbol{v} * \boldsymbol{w}$ where

$$(\boldsymbol{v} * \boldsymbol{w})(k) = \sum_{i+j=k} \boldsymbol{v}(i)\boldsymbol{w}(j) = \sum_{i=0}^{k} \boldsymbol{v}(i)\boldsymbol{w}(k-i) \tag{2.0.1}$$

with $\boldsymbol{v}(i) = 0$ and similarly $\boldsymbol{w}(j) = 0$ if the index is out of bounds. If $\boldsymbol{v}$ has length m and $\boldsymbol{w}$ has length n, $\boldsymbol{v} * \boldsymbol{w}$ has length $m + n - 1$.

We will examine the two common ways to compute $\boldsymbol{v} * \boldsymbol{w}$ in this section, starting with naive convolution and then looking at a Fourier transform based version.

2.1. Naive Convolution

One way to compute a convolution is to use the definition directly: each element is a sum across two input vectors. An implementation of this *naive convolution* (NC) is shown in Algorithm 2.1.

Algorithm 2.1 An algorithm to convolve via naive convolution vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ with length m and n respectively.

```
1: procedure NC(v, w)
2:     N ← m + n − 1
3:     r ← (0, 0, ..., 0) where there are N zeros
4:     for k ← 0, N − 1 do
5:         for i ← max(0, k − m + 1), min(n − 1, k) do
6:             r(k) ← r(k) + v(i)w(k − i)
7:         end for
8:     end for
9:     return r.
10: end procedure
```

Unfortunately, computing $\boldsymbol{v} * \boldsymbol{w}$ in this manner involves $O(nm)$ operations: (2.0.1) demonstrates that each element of $\boldsymbol{v}$ is multiplied by each element of $\boldsymbol{w}$, with subsets of these products summed. The quadratic behaviour of this algorithm makes naive convolution infeasibly slow for many problems that require the convolution of large vectors.

2.1.1. Error analysis

The main advantage of NC is that it is guaranteed to compute the entries of a convolution of non-negative vectors with controlled, small relative error.

For a real vector $\boldsymbol{p}$, we let $\widetilde{\boldsymbol{p}}$ denote the floating point approximation of $\boldsymbol{p}$, where $\widetilde{\boldsymbol{p}}(k)$ is the closest floating point value to $\boldsymbol{p}(k)$ (with ties assumed to be broken by the ties-to-even rule of IEEE754-2008). More generally, throughout this essay, we denote $\widetilde{f(\boldsymbol{p},\boldsymbol{q},\ldots)}$ a machine approximation of $f(\boldsymbol{p},\boldsymbol{q},\ldots)$, where the exact method by which the approximation is computed is either explicitly stated or clear from context.

We are typically interested in analysing the accuracy of each step of the computation and do so by assuming that the input is exactly representable, that is, $\boldsymbol{p} = \widetilde{\boldsymbol{p}}$ and $\boldsymbol{q} = \widetilde{\boldsymbol{q}}$. This is indeed the setup for our first theorem.

Theorem 2.1.1. *Suppose $\boldsymbol{p}$ and $\boldsymbol{q}$ are non-negative vectors of length m and n respectively such that $\widetilde{\boldsymbol{p}} = \boldsymbol{p}$ and $\widetilde{\boldsymbol{q}} = \boldsymbol{q}$. Let $N = m + n - 1$ be the length of the convolution $\boldsymbol{c} = \boldsymbol{p} * \boldsymbol{q}$ and let $\widetilde{\boldsymbol{c}} = \widetilde{\boldsymbol{p} * \boldsymbol{q}}$ as computed by NC, then*

$$\operatorname{rel}(\widetilde{\boldsymbol{c}}(k), \boldsymbol{c}(k)) \leq (N+1)\varepsilon(1+N\varepsilon) \tag{2.1.2}$$

for each k.

Proof. By definition, we have

$$\boldsymbol{c}(k) = \sum_{i+j=k} \boldsymbol{p}(i)\boldsymbol{q}(j), \qquad \widetilde{\boldsymbol{c}}(k) = \sum_{i+j=k}^{\sim} \widetilde{\boldsymbol{p}(i)\boldsymbol{q}(j)}.$$

Hence, [Kei05, Lemma 2] implies that

$$|\widetilde{\boldsymbol{c}}(k) - \boldsymbol{c}(k)| \leq \boldsymbol{c}(k) \cdot (n_k + 1)\varepsilon(1 + n_k\varepsilon)$$

where n_k is number of entries in the NC sum where both $0 \leq i, j \leq N-1$. The proof is completed by noticing $n_k \leq N$, and that the inequality implies $\widetilde{\boldsymbol{c}}(k) = 0$ if $\boldsymbol{c}(k) = 0$. □

In other words, using `binary64` (or anything equally or more precise), NC is highly accurate for all vectors for which it can reasonably be used: the quadratic complexity means that it would take unreasonably long to use NC to convolve vectors where the result has length a significant fraction of $1/\varepsilon = 2^{53}$.

We saw previously in (1.1.6) that a bound of β on the relative error of elements translates into the same bound on the relative error of an ℓ^p norm. Hence, for NC, we have

$$\left\|\widetilde{\boldsymbol{p} * \boldsymbol{q}} - \boldsymbol{p} * \boldsymbol{q}\right\|_p \leq (N+1)\varepsilon(1+N\varepsilon)\left\|\boldsymbol{p} * \boldsymbol{q}\right\|_p. \tag{2.1.3}$$

Example 2.1.4. Consider the vector $\boldsymbol{p} = (0, a, b, c)$ with $a = 1 - b - c$, $b = 10^{-5}$ and $c = 10^{-20}$. The convolution $\boldsymbol{p} * \boldsymbol{p}$ can be computed exactly via

NC with rational arithmetic, with the following approximation:

$$\begin{aligned}\boldsymbol{p} * \boldsymbol{p} &= (0, 0, a^2, 2ab, 2ac + b^2, 2bc, c^2) \\ &\approx (0, 0, 1, 2 \cdot 10^{-5}, 10^{-10}, 2 \cdot 10^{-25}, 10^{-40}).\end{aligned}$$

The values of the convolution as computed by NC using the `binary64` and `binary128` floating point formats are close to the exact values: the vectors of relative error for each entry, $\boldsymbol{r}_{\mathrm{NC},b}$, where b is the size of the floating point format, are

$$\begin{aligned}\boldsymbol{r}_{\mathrm{NC},64} &= (0, 0, 2 \cdot 10^{-16}, 1 \cdot 10^{-20}, 2 \cdot 10^{-16}, 1 \cdot 10^{-16}, 3 \cdot 10^{-17}), \\ \boldsymbol{r}_{\mathrm{NC},128} &= (0, 0, 2 \cdot 10^{-35}, 3 \cdot 10^{-35}, 3 \cdot 10^{-35}, 3 \cdot 10^{-35}, 4 \cdot 10^{-35}).\end{aligned}$$

These values are of course less than the bound (2.1.2), which is $5.5 \cdot 10^{-16}$ and $4.8 \cdot 10^{-34}$ respectively for the two formats. □

2.2. Direct FFT

Stockham [Sto66] demonstrated that one can compute a convolution far more quickly via a Fourier Transform, leveraging the so-called convolution theorem. This theorem does not quite compute the convolution, but rather the variant defined as follows.

Definition 2.2.1. *If $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{C}^Q$, then the* cyclic convolution $\boldsymbol{v} *_Q \boldsymbol{w} \in \mathbb{C}^Q$ *is defined as*

$$(\boldsymbol{v} *_Q \boldsymbol{w})(k) = \sum_{i=0}^{Q-1} \boldsymbol{v}(i)\boldsymbol{w}((k-i) \bmod Q) \tag{2.2.2}$$

where $0 \leq x \bmod Q < Q$ is the positive remainder of dividing x by Q.

The DFT can be used to compute this convolution directly, efficiently.

Theorem 2.2.3. *Given vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{C}^Q$, then*

$$\boldsymbol{v} *_Q \boldsymbol{w} = D^{-1}(D\boldsymbol{v} \odot D\boldsymbol{w}) \tag{2.2.4}$$

where $\odot$ is the pointwise product of two vectors of equal length.

As we saw in Section 1.5, the fast Fourier transform allows one to compute D and D^{-1} of Q-dimensional complex vectors in $O(Q \log Q)$ time. Hence, since (2.2.4) performs three such operations, along with the $O(Q)$ pointwise product, the overall run time is $O(Q \log Q)$. However, this cyclic convolution is not the exact vector we require.

Fortunately, the cyclic convolution can be used to deduce the conventional convolution by extending vectors with a suffix of zeros. Suppose $\boldsymbol{v} \in \mathbb{C}^m$ and $\boldsymbol{w} \in \mathbb{C}^n$, where $m, n \geq 1$, and let $N = m + n - 1$ be the length of the convolution $\boldsymbol{v} * \boldsymbol{w}$. For any $Q \geq N$, define two vectors $\boldsymbol{v}_Q, \boldsymbol{w}_Q \in \mathbb{C}^Q$ by

$$\boldsymbol{v}_Q(k) = \begin{cases} \boldsymbol{v}(k) & \text{if } k < m \\ 0 & \text{otherwise} \end{cases} \tag{2.2.5}$$

and similarly for $\boldsymbol{w}_Q$ in terms of $\boldsymbol{w}$ and n. It is a standard result that the cyclic convolution (2.2.2) of these extended vectors is essentially the conventional convolution (2.0.1). In particular, we have, via (2.2.4),

$$\boldsymbol{v} * \boldsymbol{w} = \text{trunc}(\boldsymbol{v}_Q *_Q \boldsymbol{w}_Q) = \text{trunc}(D^{-1}(D\boldsymbol{v}_Q \odot D\boldsymbol{w}_Q)), \tag{2.2.6}$$

where trunc : $\mathbb{C}^Q \to \mathbb{C}^N$ is the projection of a vector in $\mathbb{C}^Q$ onto $\mathbb{C}^N$ that consists of just taking the vector's first N coordinates. In fact, those trailing elements have exact value zero, so some applications may be better served by omitting the trunc operation. Algorithm 2.2 lists FFT-C, which implements this procedure.

Algorithm 2.2 An algorithm to compute a convolution via FFT of vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ of length m and n respectively.

1: **procedure** FFT-C($\boldsymbol{v}$, $\boldsymbol{w}$)
2: Set $N \leftarrow m + n - 1$ and choose some $Q \geq N$.
3: Extend $\boldsymbol{v}, \boldsymbol{w}$ to $\boldsymbol{v}_Q, \boldsymbol{w}_Q \in \mathbb{C}^Q$ by padding with zeros.
4: **return** $D^{-1}(D\boldsymbol{v}_Q \odot D\boldsymbol{w}_Q)$, truncated to length N.
5: **end procedure**

We use the length $Q \geq N$ rather than N directly because Fourier transforms can typically be performed with a smaller constant factor in the almost linear complexity when the length of the vectors is composed of only small prime factors, for example, $Q = 2^a 3^b 5^c$. Even if Q is restricted to just having form $Q = 2^a$, there is such a Q satisfying $N \leq Q < 2N$, and hence the complexity of (2.2.6) is $O((n + m)\log(n + m))$ with this choice. This almost-linear complexity implies that we can apply FFT-based convolution to vectors far longer than we can when using NC.

Fourier transform based convolutions do have guarantees about their accuracy, and indeed can be more accurate than the naive method in some senses. For instance, quoting [GO77], Schatzman [Sch96] writes "Transform methods normally give no appreciable amplification of roundoff errors. In fact, the evaluation of convolution-like sums using FFTs often gives results with much smaller roundoff error than would be obtained if the convolution sums were evaluated directly".

However, these analyses focus on the accuracy of the vector as a whole, via ℓ^p norms, which means there is not much regard to the accuracy of individual elements of a transform and/or convolution. In particular, there are no guarantees about the relative accuracy of the elements, as there are for NC.

For example, Figure 2.1 demonstrates the computation of a convolution via FFT-C, contrasting it with the NC-computed accurate value. The FFT-C convolution is only accurate for values larger than approximately 10^{-16}, once the exact convolution falls below this level, the numeric noise generated by catastrophic cancellation is much larger than the true values. Indeed, the

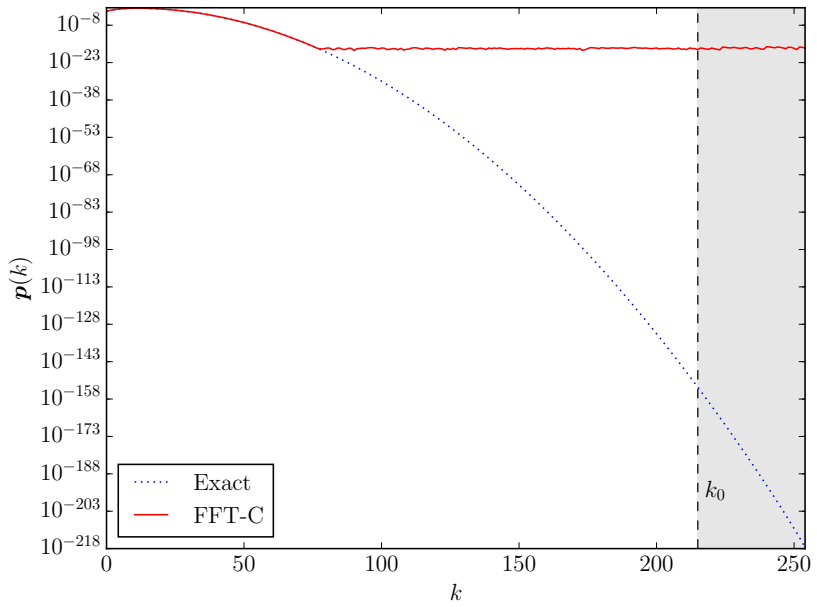

Figure 2.1 – The values of the pmf $\boldsymbol{p}*\boldsymbol{p}$ (solid line) as computed via FFT-C of Algorithm 2.2, are compared with the values computed via naive-convolution (dotted line). Here $\boldsymbol{p}(k) = A\exp\left(\frac{1}{60}k(10-k)\right)$ for $k = 0, 1, \ldots, 127$, where A guarantees $\boldsymbol{p}$ is a pmf, that is, $\|\boldsymbol{p}\|_1 = 1$. In the context of this essay, interest in convolutions is motivated by computing p-values, and in this case, the p-value of $k_0 = 215$, that is, the sum of the shaded region, $P = \sum_{k\geq k_0}(\boldsymbol{p}*\boldsymbol{p})(k)$ is computed to be $\widetilde{P} \approx 3\cdot 10^{-16}$ when the convolution is performed via FFT-C, which is more than 138 orders of magnitude larger than the true value of $P \approx 6\cdot 10^{-154}$.

value of the smallest element of the convolution as computed via FFT-C is approximately 10^{200} times larger than the exact value.

Example 2.2.7. Let us return to Example 2.1.4, and do the same convolutions with FFT-C instead of NC. We used the FFTW3 library [FJ05] for the transforms with both `binary64` and `binary128`, but any implementation would similarly have relative errors much larger than those of NC:

$$\boldsymbol{r}_{\text{FFT-C,64}} = (0, \infty, 4\cdot 10^{-17}, 5\cdot 10^{-12}, 2\cdot 10^{-7}, 3\cdot 10^{8}, 8\cdot 10^{22}),$$
$$\boldsymbol{r}_{\text{FFT-C,128}} = (\infty, \infty, 4\cdot 10^{-17}, 1\cdot 10^{-20}, 1\cdot 10^{-16}, 7\cdot 10^{-12}, 3\cdot 10^{5}). \qquad \square$$

FFT-C with fixed-precision floating point formats will always suffer from such catastrophic cancellation, when operating on vectors with sufficiently large dynamic range.

2.2.1. Error Analysis

The first step to having guarantees on the precision of an algorithm is understanding the error it introduces. We now analyse the round-off error introduced by performing convolution with a Fourier transform and its inverse

implemented with floating-point arithmetic. Denote these approximations to D and D^{-1} as $\widetilde{D}$ and $\widetilde{D^{-1}}$ respectively.

The goal of this section is to prove the following theorem and hence its corollary. For this purpose, we suppose that we are working with a floating point system with machine precision $\varepsilon < 2^{-5}$ with the rounding mode set to round to nearest (rounding ties to even).

Theorem 2.2.8. *Suppose $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{C}^Q$ are vectors of length $Q = 2^K$ such that $\widetilde{\boldsymbol{v}} = \boldsymbol{v}$ and $\widetilde{\boldsymbol{w}} = \boldsymbol{w}$, and $\widetilde{\boldsymbol{v} *_Q \boldsymbol{w}}$ is an approximation to the cyclic convolution $\boldsymbol{v} *_Q \boldsymbol{w}$ computed via (2.2.4) with $\widetilde{D}$ and $\widetilde{D^{-1}}$ in place of D and D^{-1}, then*

$$\left\| \widetilde{\boldsymbol{v} *_Q \boldsymbol{w}} - \boldsymbol{v} *_Q \boldsymbol{w} \right\|_\infty < cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2 (1 + O(\varepsilon)) \tag{2.2.9}$$

where $c = 15$ for $K \geq 1$ and $c = 13.5$ for $K \geq 5$.

Our main goal is to understand conventional, not cyclic, convolutions, and so it is really the following result that is most interesting. It follows directly from (2.2.6), which demonstrates that the elements of the convolution are a subset of the elements of a cyclic convolution.

Corollary 2.2.10. *Suppose $\boldsymbol{x} \in \mathbb{C}^m$ and $\boldsymbol{y} \in \mathbb{C}^n$ such that $\boldsymbol{x} = \widetilde{\boldsymbol{x}}$ and $\boldsymbol{y} = \widetilde{\boldsymbol{y}}$. Let $Q = 2^K \geq m + n - 1$, and let $\widetilde{\boldsymbol{x} * \boldsymbol{y}}$ be an approximation to the convolution $\boldsymbol{x} * \boldsymbol{y}$ computed via (2.2.6) by padding to length Q with $\widetilde{D}$ and $\widetilde{D^{-1}}$ in place of D and D^{-1}, then*

$$\left\| \widetilde{\boldsymbol{x} * \boldsymbol{y}} - \boldsymbol{x} * \boldsymbol{y} \right\|_\infty < cK\varepsilon \left\|\boldsymbol{x}\right\|_2 \left\|\boldsymbol{y}\right\|_2 (1 + O(\varepsilon))$$

where $c = 15$ for $K \geq 1$ and $c = 13.5$ for $K \geq 5$.

Let $\overline{\boldsymbol{v}} = D\boldsymbol{v}$ and $\widetilde{\overline{\boldsymbol{v}}} = \widetilde{D}\boldsymbol{v}$, and similarly $\overline{\boldsymbol{w}}$ and $\widetilde{\overline{\boldsymbol{w}}}$ for $\boldsymbol{w}$. Our line of attack to prove Theorem 2.2.8 is to focus on the the left-hand side of (2.2.9)

$$\left\| \widetilde{\boldsymbol{v} *_Q \boldsymbol{w}} - \boldsymbol{v} *_Q \boldsymbol{w} \right\|_\infty = \left\| \widetilde{D^{-1}}(\widetilde{\widetilde{\boldsymbol{v}} \odot \widetilde{\overline{\boldsymbol{w}}}}) - D^{-1}(\overline{\boldsymbol{v}} \odot \overline{\boldsymbol{w}}) \right\|_\infty$$

and bound it by working through how the transforms interact with norms. Thus, we first detail some preliminary results in this area.

Let $x, y \in \mathbb{C}$ and $\boldsymbol{x} \in \mathbb{C}^Q$ be such that $\widetilde{x} = x$, $\widetilde{y} = y$ and $\widetilde{\boldsymbol{x}} = \boldsymbol{x}$. Under the assumptions of the behaviour of the floating point system we have made, [BPZ07] proved that

$$|\widetilde{xy} - xy| < \sqrt{5}\varepsilon \left|xy\right|. \tag{2.2.11}$$

In [Kei05, Lemma 4], Keich proves the following bound on the error of a transform:

$$\left\| (D - \widetilde{D})\boldsymbol{x} \right\|_\infty \leq ((\mu + 2)K + O(\varepsilon))\varepsilon \left\|\boldsymbol{x}\right\|_1 \tag{2.2.12}$$

$$\left\| (D^{-1} - \widetilde{D^{-1}})\boldsymbol{x} \right\|_\infty \leq \frac{(\mu + 2)K + O(\varepsilon)}{Q} \varepsilon \left\|\boldsymbol{x}\right\|_1 \tag{2.2.13}$$

for $\mu = 3$. This constant μ is derived from a less precise analysis in the manner of (2.2.11), which only deduced $3\varepsilon \left|xy\right|$ as the bound, hence, we can take $\mu = \sqrt{5}$ here.

Theorem 4.1 of [TZ01] shows a similar bound on the ℓ^2-norm of the error of $\widetilde{D}$

$$\left\|(\widetilde{D} - D)\boldsymbol{x}\right\|_2 < ((\mu+2)K + O(\varepsilon))\sqrt{Q}\varepsilon \left\|\boldsymbol{x}\right\|_2. \tag{2.2.14}$$

We also need the following standard results about vector norms and the Fourier transform. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^Q$ be arbitrary. The Cauchy-Schwartz inequality states

$$\left\|\boldsymbol{x} \odot \boldsymbol{y}\right\|_1 \leq \left\|\boldsymbol{x}\right\|_2 \left\|\boldsymbol{y}\right\|_2, \tag{2.2.15}$$

and Parseval's theorem states

$$\left\|D\boldsymbol{x}\right\|_2 = \sqrt{Q}\left\|\boldsymbol{x}\right\|_2, \qquad \left\|D^{-1}\boldsymbol{y}\right\|_2 = \frac{1}{\sqrt{Q}}\left\|\boldsymbol{y}\right\|_2. \tag{2.2.16}$$

Lemma 2.2.17. *For arbitrary $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^Q$,*

$$\left\|D\boldsymbol{x}\right\|_\infty \leq \left\|\boldsymbol{x}\right\|_1, \tag{2.2.18}$$

$$\left\|D^{-1}\boldsymbol{y}\right\|_\infty \leq \frac{1}{Q}\left\|\boldsymbol{y}\right\|_1, \tag{2.2.19}$$

$$\left\|D^{-1}(\boldsymbol{x} \odot \boldsymbol{y})\right\|_\infty \leq \frac{\left\|\boldsymbol{x}\right\|_2}{\sqrt{Q}}\frac{\left\|\boldsymbol{y}\right\|_2}{\sqrt{Q}}. \tag{2.2.20}$$

Proof. It follows from the definition that

$$\left\|D\boldsymbol{x}\right\|_\infty = \max_k \left|\sum_j \boldsymbol{x}(j)e^{-2\pi ijk}\right| \leq \left\|\boldsymbol{x}\right\|_1,$$

proving (2.2.18). Identical reasoning proves (2.2.19), just needing to match the definition of D^{-1} by negating the -2π to 2π and including the additional factor of $\frac{1}{Q}$.

The Cauchy-Schwartz inequality and (2.2.19) imply (2.2.20). □

Lemma 2.2.21. *For arbitrary $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^Q$,*

$$\left\|\widetilde{\overline{\boldsymbol{x}}}\right\|_2 \leq (1 + O(\varepsilon))\left\|\overline{\boldsymbol{x}}\right\|_2, \tag{2.2.22}$$

$$\left\|\widetilde{\widetilde{\overline{\boldsymbol{x}}} \odot \widetilde{\overline{\boldsymbol{y}}}} - \overline{\boldsymbol{x}} \odot \overline{\boldsymbol{y}}\right\|_1 \leq (2(\mu+2)K + \mu + O(\varepsilon))\varepsilon \left\|\overline{\boldsymbol{x}}\right\|_2 \left\|\overline{\boldsymbol{y}}\right\|_2, \tag{2.2.23}$$

$$\left\|\widetilde{\widetilde{\overline{\boldsymbol{x}}} \odot \widetilde{\overline{\boldsymbol{y}}}}\right\|_1 \leq (1 + O(\varepsilon))\left\|\overline{\boldsymbol{x}}\right\|_2 \left\|\overline{\boldsymbol{y}}\right\|_2. \tag{2.2.24}$$

Proof. Deducing (2.2.22) is immediate from the reverse triangle inequality applied to (2.2.14).

We break (2.2.23) into parts,

$$\begin{aligned}\left\|\widetilde{\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}}-\boldsymbol{x}\odot\boldsymbol{y}\right\|_1 &= \left\|\widetilde{\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}}-\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}+\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}-\widetilde{\boldsymbol{x}}\odot\boldsymbol{y}+\widetilde{\boldsymbol{x}}\odot\boldsymbol{y}-\boldsymbol{x}\odot\boldsymbol{y}\right\|_1\\ &\le \underbrace{\left\|\widetilde{\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}}-\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}\right\|_1}_{\gamma_1}+\underbrace{\left\|\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}-\widetilde{\boldsymbol{x}}\odot\boldsymbol{y}\right\|_1}_{\gamma_2}\\ &\quad+\underbrace{\left\|\widetilde{\boldsymbol{x}}\odot\boldsymbol{y}-\boldsymbol{x}\odot\boldsymbol{y}\right\|_1}_{\gamma_3}\end{aligned}$$

Together, (2.2.11), the Cauchy-Schwartz inequality (2.2.15), and (2.2.22) imply that

$$\begin{aligned}\gamma_1 &\le \mu\varepsilon\left\|\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}\right\|_1\\ &\le \mu\varepsilon\left\|\widetilde{\boldsymbol{x}}\right\|_2\left\|\widetilde{\boldsymbol{y}}\right\|_2\\ &\le \mu\varepsilon(1+O(\varepsilon))\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2\end{aligned}$$

Using (2.2.16) and both the Cauchy-Schwartz inequality and (2.2.14) again we can see

$$\begin{aligned}\gamma_2 &= \left\|\widetilde{\boldsymbol{x}}\odot(\widetilde{\boldsymbol{y}}-\boldsymbol{y})\right\|_1\\ &\le \left\|\widetilde{\boldsymbol{x}}\right\|_2\left\|\widetilde{\boldsymbol{y}}-\boldsymbol{y}\right\|_2\\ &\le (1+O(\varepsilon))\left\|\boldsymbol{x}\right\|_2((\mu+2)K+O(\varepsilon))\varepsilon\sqrt{Q}\left\|\boldsymbol{y}\right\|_2\\ &= ((\mu+2)K+O(\varepsilon))\varepsilon\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2\end{aligned}$$

Similarly $\gamma_3 \le ((\mu+2)K+O(\varepsilon))\varepsilon\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2$, and combining these three bounds yields (2.2.23).

Finally, (2.2.24) follows from (2.2.23) and (2.2.15):

$$\begin{aligned}\left\|\widetilde{\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}}\right\|_1 &\le \left\|\widetilde{\widetilde{\boldsymbol{x}}\odot\widetilde{\boldsymbol{y}}}-\boldsymbol{x}\odot\boldsymbol{y}\right\|_1+\left\|\boldsymbol{x}\odot\boldsymbol{y}\right\|_1\\ &\le O(\varepsilon)\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2+\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2\\ &= (1+O(\varepsilon))\left\|\boldsymbol{x}\right\|_2\left\|\boldsymbol{y}\right\|_2\end{aligned}$$ □

We can now complete the analysis.

Proof of Theorem 2.2.8. We write the left-hand side of (2.2.9) as,

$$\begin{aligned}\left\|\widetilde{\boldsymbol{v}*_Q\boldsymbol{w}}-\boldsymbol{v}*_Q\boldsymbol{w}\right\|_\infty &= \left\|\widetilde{D^{-1}}(\widetilde{\widetilde{\boldsymbol{v}}\odot\widetilde{\boldsymbol{w}}})-D^{-1}(\boldsymbol{v}\odot\boldsymbol{w})\right\|_\infty\\ &\le \underbrace{\left\|D^{-1}(\widetilde{\widetilde{\boldsymbol{v}}\odot\widetilde{\boldsymbol{w}}}-\boldsymbol{v}\odot\boldsymbol{w})\right\|_\infty}_{\alpha}+\underbrace{\left\|(\widetilde{D^{-1}}-D^{-1})\widetilde{\widetilde{\boldsymbol{v}}\odot\widetilde{\boldsymbol{w}}}\right\|_\infty}_{\beta}\end{aligned}$$

The first term α can be bounded by the combination of (2.2.19), (2.2.23) and Parseval's theorem (2.2.16),

$$\begin{aligned}\alpha &\leq \frac{1}{Q}\left\|\widetilde{\widetilde{\boldsymbol{v}}\odot\widetilde{\boldsymbol{w}}}-\overline{\boldsymbol{v}}\odot\overline{\boldsymbol{w}}\right\|_1\\ &\leq \frac{1}{Q}(2(\mu+2)+\mu+O(\varepsilon))\varepsilon\left\|\overline{\boldsymbol{v}}\right\|_2\left\|\overline{\boldsymbol{w}}\right\|_2\\ &= (2(\mu+2)+\mu+O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_2\left\|\boldsymbol{w}\right\|_2\end{aligned}$$

This leaves β, which falls to (2.2.13), (2.2.24) and Parseval's theorem again,

$$\begin{aligned}\beta &\leq \frac{(\mu+2)K+O(\varepsilon)}{Q}\varepsilon\left\|\widetilde{\widetilde{\boldsymbol{v}}\odot\widetilde{\boldsymbol{w}}}\right\|_1\\ &\leq \frac{(\mu+2)K+O(\varepsilon)}{Q}\varepsilon\left\|\overline{\boldsymbol{v}}\right\|_2\left\|\overline{\boldsymbol{w}}\right\|_2\\ &= ((\mu+2)K+O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_2\left\|\boldsymbol{w}\right\|_2.\end{aligned}$$

Hence, bringing the parts together gives

$$\left\|\widetilde{\boldsymbol{v}*_Q\boldsymbol{w}}-\boldsymbol{v}*_Q\boldsymbol{w}\right\|_\infty \leq (3(\mu+2)K+\mu+O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_2\left\|\boldsymbol{w}\right\|_2,$$

and the constants mentioned in Theorem 2.2.8 follow from $\mu=\sqrt{5}$. □

2.3. Shifted FFT

Keich [Kei05] introduced the sFFT algorithm (shifted FFT) that guarantees the accuracy of the tail sum of certain FFT-based convolutions. Specifically, given a log concave non-negative pmf $\boldsymbol{p}$, sFFT allows one to accurately compute a region of the repeated self-convolution $\boldsymbol{p}^{*L}$ around some index s_0. The algorithm uses this to accurately compute the tail sum of $\boldsymbol{p}^{*L}$ from s_0 to the end of the vector,

$$P=\sum_{s\geq s_0}\boldsymbol{p}^{*L}(s). \tag{2.3.1}$$

The values near s_0 will typically dominate that sum, so one can retrieve an accurate estimate of P without knowing the complete value of the convolution $\boldsymbol{p}^{*L}$.

The motivation for sFFT is calculating small p-values for performing exact tests: $\boldsymbol{p}^{*L}$ is the pmf of the random variable $X=\sum^L X_i$ where the X_i are independent and identically distributed (iid) lattice-valued random variables with pmf $\boldsymbol{p}$, and thus P is exactly the p-value of s_0. sFFT gives an efficient way to compute an approximation to P, with guarantees on its accuracy.

Like FFT-C, sFFT relies on the convolution theorem, which for a pmf $\boldsymbol{p}$ of length n, in this case has the form

$$\boldsymbol{p}^{*L}=D^{-1}\left((D\boldsymbol{p})^{\odot L}\right), \tag{2.3.2}$$

where $(\boldsymbol{v}^{\odot L})(k) = \boldsymbol{v}(k)^L$ and D and D^{-1} are $Q \geq L(n-1)+1$ dimensional operators. As before, the vector $\boldsymbol{p}$ must first be extended with zeros to have length Q to account for the DFT naturally computing a cyclic convolution. The FFT allows D and D^{-1} to be computed in $O(Q \log Q)$ time, and hence the run time complexity of this algorithm is $O(Ln \log Ln)$.

Of course, as with FFT-C, implementing (2.3.2) with the finite precision operators $\widetilde{D}$ and $\widetilde{D^{-1}}$ means small values of $\boldsymbol{p}^{*L}$ will be calculated with very high relative error, and the extra operations implied by the exponentiation by L only exacerbates this problem.

The key to sFFT guaranteeing accuracy is performing an exponential shift of the vector $\boldsymbol{p}$. The statistical literature often refers to this as "exponential tilting" (for instance, [BC89]), and it is the procedure that underlies saddlepoint approximations. Conceptually, the shift acts as a rotation of $\log \boldsymbol{p}$, to ensure that values in the area of interest are large enough to be computed accurately via FFT-C.

Specifically, given some factor θ one can compute the shifted $\boldsymbol{p}_\theta$ defined by

$$\boldsymbol{p}_\theta(k) = \boldsymbol{p}(k)e^{\theta k}/M_{\boldsymbol{p}}(\theta), \tag{2.3.3}$$

where $M_{\boldsymbol{p}}(\theta) = \sum_k \boldsymbol{p}(k)e^{k\theta}$ is the moment-generating function that ensures that $\boldsymbol{p}_\theta$ is also normalised to be a pmf. Ignoring the normalisation, which is multiplication of the vector by a constant and hence easy to compensate for, this operation commutes with convolution:

$$(\boldsymbol{p}_\theta * \boldsymbol{q}_\theta)(k) = \sum_{i+j=k} \boldsymbol{p}(i)\boldsymbol{q}(j)e^{(i+j)\theta} = (\boldsymbol{p} * \boldsymbol{q})(k)e^{\theta k}. \tag{2.3.4}$$

In other words, $(\boldsymbol{p}_\theta * \boldsymbol{q}_\theta) = (\boldsymbol{p} * \boldsymbol{q})_\theta$.

The θ by which $\boldsymbol{p}$ should be shifted is chosen based on an index s_0 selected ahead of time, so that, heuristically, the maximum of $\boldsymbol{p}_\theta^{*L}$ lies at or very close to index s_0. Specifically, with the choice

$$\theta_0 = \arg\min_\theta \log M_{\boldsymbol{p}}(\theta) - \theta s_0/L, \tag{2.3.5}$$

the expectation of $\boldsymbol{p}_\theta^L$ is s_0, and, moreover, this choice minimizes the following bound on the approximation error of the tail sum P [Kei05, Section 4.5],

$$\left|P - \widetilde{P_\theta}\right| \leq \gamma e^{-\theta s_0 + L \log M_{\boldsymbol{p}}(\theta)} \tag{2.3.6}$$

where γ is a factor that does not depend on θ, and $\widetilde{P_\theta}$ represents P with the convolution $\boldsymbol{p}^{*L}$ computed via sFFT with a fixed exponential shift of θ.

The log concavity assumption on $\boldsymbol{p}$ guarantees that the largest values of $\boldsymbol{p}_{\theta_0}^{*L}$ will lie around s_0, and thus will be accurately computed by FFT-C. Once the shift is inverted, these accurately computed elements near s_0 will be the largest elements $\boldsymbol{p}_{\theta_0}^{*L}(s)$ for $s \geq s_0$, and hence the largest contributors to the tail sum P.

The remaining piece of the algorithm is ensuring that inaccurate calculation of the smaller values does not influence the final result. As we saw above, FFT-C may calculate values many orders of magnitude larger than

the true value when the true value is very small. Thus, to avoid this somewhat uncontrolled noise being incorrectly included in the sum, sFFT will zero any elements that are likely to be miscomputed. It does this by an analysis of the error in (2.3.2), the conclusion of which is setting the entry $\widetilde{\boldsymbol{p}^{*L}}(s)$ to zero if it is such that

$$\widetilde{\boldsymbol{p}^{*L}}(s) < LcK\varepsilon, \tag{2.3.7}$$

where c is a small universal constant, and $Q = 2^K$ was the dimension of the Fourier transform operators. This bound is justified by [Kei05, Lemma 5], which we look at in more detail below.

Algorithm 2.3 lists the sFFT algorithm as defined by [Kei05], which computes an approximation to the tail sum P. Two plots showing the various vectors computed in the process of sFFT are displayed in Figure 2.2. The first demonstrates how sFFT resolves the problem in Figure 2.1 and hence correctly computes the p-value, while the second gives an example where sFFT fails to compute an accurate approximation to the p-value, due to a non-log-concave pmf.

Algorithm 2.3 The sFFT algorithm, for computing the region of $\boldsymbol{p}^{*L}$ around some index s_0, where $\boldsymbol{p}$ is a log concave pmf, with length n.

procedure sFFT($\boldsymbol{p}$, L, s_0)
  Choose θ_0 via (2.3.5), and set $M \leftarrow \sum_k \boldsymbol{p}(k)e^{k\theta_0}$.
  Shift $\boldsymbol{p}$ to $\boldsymbol{p}_{\theta_0}$ via $\boldsymbol{p}_{\theta_0}(k) \leftarrow \boldsymbol{p}(k)e^{k\theta_0}/M$.
  Choose K such that $2^K \geq (n-1)L + 1$.
  Pad $\boldsymbol{p}_{\theta_0}$ with zeros so that it has length 2^K.
  Compute $\widetilde{\boldsymbol{p}^{*L}_{\theta_0}}$ via (2.3.2).
  Zero any element of $\widetilde{\boldsymbol{p}^{*L}_{\theta_0}}$ below the error threshold $LcK\varepsilon$ (see Lemma 2.3.8).
  Deduce $\widetilde{\boldsymbol{p}^{*L}}$ by inverting the shift $\widetilde{\boldsymbol{p}^{*L}}(k) \leftarrow \widetilde{\boldsymbol{p}^{*L}_{\theta_0}}(k)e^{-k\theta_0}M^L$.
  return $\sum_{s \geq s_0} \widetilde{\boldsymbol{p}^{*L}}(s)$.
end procedure

2.3.1. Error analysis

The analysis of the error in sFFT in [Kei05] gives two key results, quoted here with the assumption that the quantity c_δ defined in [Kei05] is zero, since we are not performing any lattice approximations, which c_δ quantifies.

The first result focuses on the actual error in a convolution computed via (2.3.2), under the assumption that the length of the final result is $Q = 2^K$. This gives the bound (2.3.7) for filtering $\widetilde{\boldsymbol{p}^{*L}_{\theta_0}}$.

Lemma 2.3.8 ([Kei05, Lemma 5])**.** *Let $\boldsymbol{p} \in \mathbb{R}^n$ be a pmf (i.e., $\boldsymbol{p}(k) \geq 0$ and $\|\boldsymbol{p}\|_1 = 1$), and let $\widetilde{\boldsymbol{p}^{*L}}$ be computed via* (2.3.2) *with floating point, then*

$$\left\|\widetilde{\boldsymbol{p}^{*L}} - \boldsymbol{p}^{*L}\right\|_\infty \leq LcK\varepsilon$$

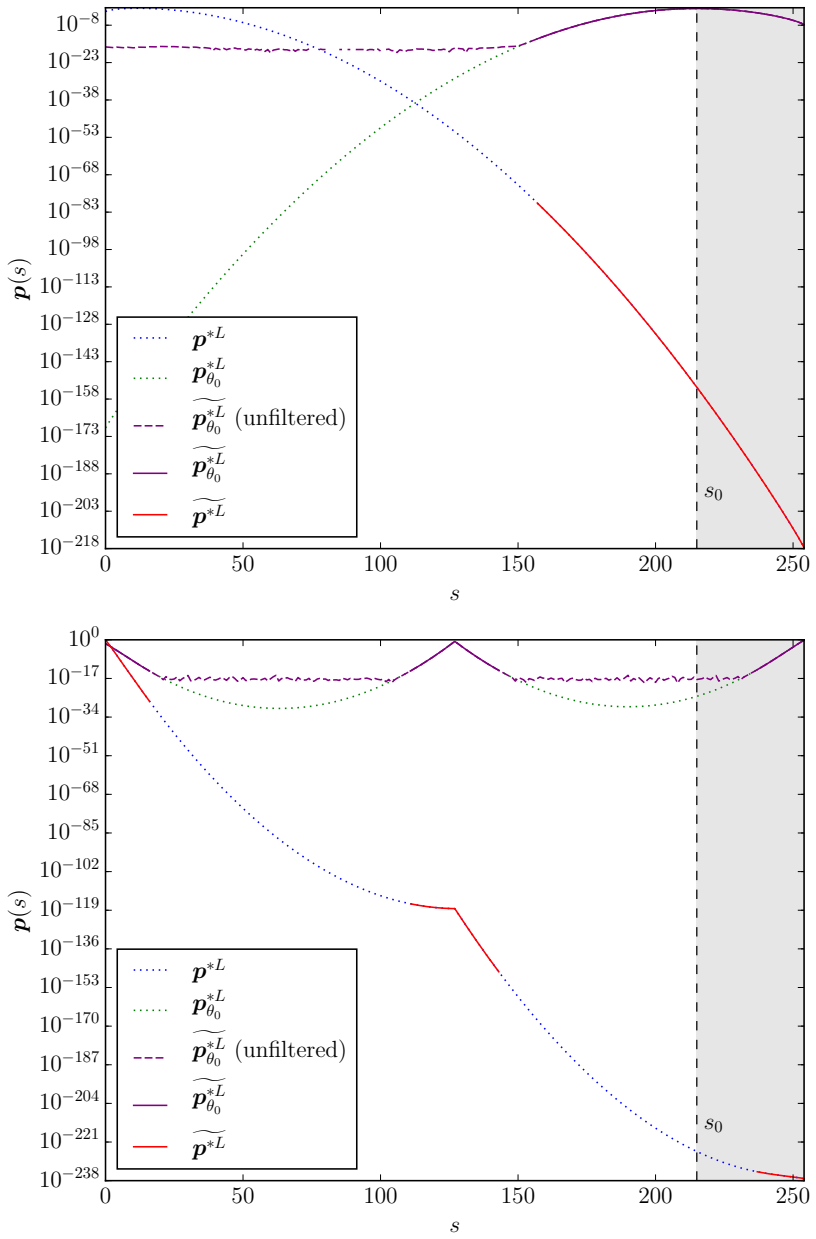

Figure 2.2 – A breakdown of the sFFT algorithm when used to compute an approximation $\widetilde{P}$ to $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ for $s_0 = 215$ and $L = 2$ with two different pmfs $\boldsymbol{p}$. The first plot uses $\boldsymbol{p}$ from Figure 2.1, while the second has $\boldsymbol{p}(s) = A \exp\left(\frac{1}{60} s(s-256)\right)$ for $s = 0, \ldots, 127$, with A such that $\|\boldsymbol{p}\|_1 = 1$. Both show the vectors computed at each stage of the algorithm: the dotted lines show exact values of the convolutions, solid lines show the values computed by sFFT after filtering errors out, and the dashed purple line shows the noise in the FFT-C computation that was removed. As can be seen, the solid red line matches the exact value of $\boldsymbol{p} * \boldsymbol{p}$ very closely around s_0 in the first example, and, indeed, $\mathrm{rel}(\widetilde{P}, P) < 10^{-13}$. The second one demonstrates how sFFT can fail with non-log-concave pmfs: the region around s_0, which contributes most to P, is still small in the shifted pmf and hence is not calculated accurately by FFT-C. This results in $\widetilde{P} \approx 2 \cdot 10^{-234}$ with the true value $P \approx 1 \cdot 10^{-225}$ many times larger.

where c is a small universal constant, $Q = 2^K$ is the size of the Fourier transforms used, and $Q \geq (n-1)L + 1$, the length of $\boldsymbol{p}^{*L}$.

The second result is the error in the result of sFFT as an approximation to P, and implies the bound (2.3.6) that supports (2.3.5) for choosing θ_0.

Lemma 2.3.9 ([Kei05, Claim 2])**.** *Let P be the tail sum* (2.3.1)*, and $\widetilde{P_\theta}$ be the approximation to it computed via sFFT with fixed $\theta_0 = \theta$ instead of computing it via* (2.3.5)*, then*

$$\left|P - \widetilde{P_\theta}\right| \leq LcK\varepsilon \sum_{s \geq s_0} e^{-\theta s + L \log M(\theta)},$$

where c and K are as in Lemma 2.3.8.

Chapter 3

Accurate FFT Convolutions

Now that we understand the error in an FFT-based convolution, we can try to control and manipulate it, to design FFT-based convolution algorithms that have guarantees about the values of individual elements.

3.1. Checked FFT-C

Our first step toward accurate FFT-based computation of a convolution is the identification of values that are or can be accurately computed via FFT-C. Given some vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ and the approximation $\widetilde{\boldsymbol{v} * \boldsymbol{w}}$ to $\boldsymbol{v} * \boldsymbol{w}$, identifying accurately computed values can be done in two ways: either by examining the computed values $(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k)$ to retroactively find those that are possibly inaccurate, or instead, by reasoning about the exact $\boldsymbol{v} * \boldsymbol{w}$. The practicality of the latter approach is not so obvious, because the convolution $\boldsymbol{v} * \boldsymbol{w}$ is exactly what we want to compute, but one can still prove theoretical results about it and hence deduce practical conditions.

For this section, unless otherwise stated, $\boldsymbol{v} \in \mathbb{C}^m$ and $\boldsymbol{w} \in \mathbb{C}^n$ are complex vectors with $\boldsymbol{v} = \widetilde{\boldsymbol{v}}$ and $\boldsymbol{w} = \widetilde{\boldsymbol{w}}$. Further assume that $\widetilde{\boldsymbol{v} * \boldsymbol{w}}$ is computed via FFT-C, with $Q = 2^K \geq m + n - 1$ and c referring to the values used in Corollary 2.2.10.

3.1.1. Guaranteeing FFT-C accuracy retroactively

Corollary 2.2.10 gave bounds on the maximal *absolute* error of elements of a convolution computed via FFT-C, which can be used to bound the relative error of sufficiently large elements, as described in the following corollary of Corollary 2.2.10.

Lemma 3.1.1. *Suppose $\alpha \geq 2$, then if the index k is such that*

$$\left|(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k)\right| \geq (\alpha + 1) \cdot cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2, \tag{3.1.2}$$

the computed value at k has bounded relative error,

$$\operatorname{rel}((\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k), (\boldsymbol{v} * \boldsymbol{w})(k)) < \frac{1}{\alpha}.$$

Proof. Let $x = (\boldsymbol{v} * \boldsymbol{w})(k)$ and $\widetilde{x} = (\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k)$, then Corollary 2.2.10 implies

$$|\widetilde{x} - x| \leq cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2,$$

which, combined with the reverse triangle inequality, gives,

$$\begin{aligned} |x| &> |\widetilde{x}| - cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2 \\ &\geq \alpha \cdot cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2. \end{aligned}$$

Putting them together completes the proof

$$\operatorname{rel}(\widetilde{x}, x) = \frac{|\widetilde{x} - x|}{|x|} < \frac{cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2}{\alpha \cdot cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2} = \frac{1}{\alpha}. \qquad \square$$

This result allows for retroactively evaluating which values of an FFT-C convolution are guaranteed to be accurate, to any chosen level of accuracy. This theorem has the notable practical quality of only needing knowledge of the computed convolution $\widetilde{\boldsymbol{v} * \boldsymbol{w}}$, that is, the approximation, rather than the (unknown) exact value $\boldsymbol{v} * \boldsymbol{w}$.

3.1.2. Guaranteeing FFT-C accuracy prospectively

Corollary 2.2.10 implies that, if $(\boldsymbol{v} * \boldsymbol{w})(k) = 0$, the FFT-C computed value $(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k)$ may be as large as $cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2$. By setting to zero computed values that are smaller than this bound, we can guarantee that we compute the zeros of a convolution accurately. Note that some of those values set to zero may in fact be non-zero, however ensuring that zeros are computed accurately in this manner reduces the maximum possible relative error in an estimate from ∞ to 1, or even further, as we will see below.

Definition 3.1.3. *Given a vector $\widetilde{\boldsymbol{z}} \in \mathbb{C}^{m+n-1}$ taken as an approximation to $\boldsymbol{v} * \boldsymbol{w}$, the filtering function* $\operatorname{filt}_{\boldsymbol{v},\boldsymbol{w}} : \mathbb{C}^{m+n-1} \to \mathbb{C}^{m+n-1}$ *is defined as*

$$\operatorname{filt}_{\boldsymbol{v},\boldsymbol{w}}(\widetilde{\boldsymbol{z}})(k) = \begin{cases} \widetilde{\boldsymbol{z}}(k) & \text{if } |\widetilde{\boldsymbol{z}}(k)| \geq cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2 \\ 0 & \text{otherwise.} \end{cases} \tag{3.1.4}$$

We will usually write just filt without its subscripts, when they are clear from context.

The control in relative error that this function gives can be quantified as follows.

Lemma 3.1.5. *Suppose $\alpha \geq 2$, and, an index k is such that either*

$$\begin{cases} |(\boldsymbol{v} * \boldsymbol{w})(k)| \geq \alpha \cdot cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2, & \text{or} \\ (\boldsymbol{v} * \boldsymbol{w})(k) = 0, \end{cases} \tag{3.1.6}$$

where c is that defined in Corollary 2.2.10, then the approximation $\widetilde{\boldsymbol{v} * \boldsymbol{w}}$ computed via FFT-C satisfies,

$$\operatorname{rel}(\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k), (\boldsymbol{v} * \boldsymbol{w})(k)) < \frac{1}{\alpha}. \tag{3.1.7}$$

Proof. Let $x = (\boldsymbol{v} * \boldsymbol{w})(k)$, $\widetilde{x} = (\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k)$ and $\widetilde{x}' = \operatorname{filt}(\boldsymbol{v} * \boldsymbol{w})(k)$. There are two choices, either $|x| > \alpha cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2$ or $|x| = 0$. For both cases

$$|\widetilde{x} - x| < cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2. \tag{3.1.8}$$

In the former case, we have

$$|\widetilde{x}| > |x| - cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2 \geq cK\varepsilon \|\boldsymbol{v}\|_2 \|\boldsymbol{w}\|_2,$$

and hence $\widetilde{x}' = \widetilde{x} \neq 0$. In this case,

$$\operatorname{rel}(\widetilde{x}', x) = \frac{|\widetilde{x} - x|}{|x|} < \frac{cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2}{\alpha \cdot cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2} = \frac{1}{\alpha}.$$

In the latter case, $x = 0$, and hence (3.1.8) implies $\widetilde{x}' = 0$, giving $\operatorname{rel}(\widetilde{x}', x) = 0$. □

Unlike Lemma 3.1.1, this result relies on the values of the exact convolution $\boldsymbol{v} * \boldsymbol{w}$ and so cannot be used directly to retroactively verify the values of a convolution $\widetilde{\boldsymbol{v} * \boldsymbol{w}}$. However, its utility will become clearer in the next sections as it is the critical building block.

3.1.3. Support of a convolution

As we next argue, Lemma 3.1.5 implies that it is possible to efficiently calculate the support of convolutions of even very long non-negative vectors, and hence, all the zeros of the convolution can be guaranteed to be computed accurately.

This result is best given in terms of the smallest non-zero element of a convolution, which we denote $\min_+ X = \min\{x \in X \mid x > 0\}$. This allows us to phrase it more clearly by focusing on the first case of (3.1.6). We hence have the following corollary of Lemma 3.1.5, with c and K as in that lemma.

Corollary 3.1.9. *If* (3.1.6) *holds with* $\alpha = 2$ *for all* k*, that is, if*

$$\min_{k+} |(\boldsymbol{v} * \boldsymbol{w})(k)| \geq 2 \cdot cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2, \tag{3.1.10}$$

then the support of the vectors $\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})$ *and* $\boldsymbol{v} * \boldsymbol{w}$ *are identical.*

Proof. Since (3.1.6) holds, the conclusion of Lemma 3.1.5 applies to every index k. That is, for all k, we have

$$\operatorname{rel}(\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k), (\boldsymbol{v} * \boldsymbol{w})(k)) < \frac{1}{2}.$$

If $(\boldsymbol{v} * \boldsymbol{w})(k) = 0$, this forces $\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k) = 0$. On the other hand, if $(\boldsymbol{v} * \boldsymbol{w})(k) \neq 0$, then $\operatorname{rel}(0, (\boldsymbol{v} * \boldsymbol{w})(k)) = 1 \geq 1/2$, and thus we must have $\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})(k) \neq 0$. □

Given non-negative and non-zero vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ with convolution $\boldsymbol{p} * \boldsymbol{q}$ of length Q, define $\boldsymbol{v} = \mathbf{1}_{\boldsymbol{p}>0}$ and $\boldsymbol{w} = \mathbf{1}_{\boldsymbol{q}>0}$. The vectors are non-negative, so the support of the exactly computed $\boldsymbol{p} * \boldsymbol{q}$, $\operatorname{supp}(\boldsymbol{p} * \boldsymbol{q})$, is identical to the support of the convolution of the supports, $\operatorname{supp}(\boldsymbol{v} * \boldsymbol{w})$.

We have $\left\|\boldsymbol{v}\right\|_2 \leq \sqrt{Q}$ and similarly for $\left\|\boldsymbol{w}\right\|_2$, and, since every element of $\boldsymbol{v}$ and $\boldsymbol{w}$ is either 1 or 0,

$$\min_{k+} |(\boldsymbol{v} * \boldsymbol{w})(k)| = 1.$$

Hence, as long as $QK = Q \log_2 Q \leq (2c\varepsilon)^{-1}$ we have

$$2 \cdot cK\varepsilon \left\|\boldsymbol{v}\right\|_2 \left\|\boldsymbol{w}\right\|_2 \leq 2cK\varepsilon Q \leq 1,$$

and thus it follows from Corollary 3.1.9 that

$$\operatorname{supp}(\operatorname{filt}(\widetilde{\boldsymbol{v} * \boldsymbol{w}})) = \operatorname{supp}(\boldsymbol{v} * \boldsymbol{w}) = \operatorname{supp}(\boldsymbol{p} * \boldsymbol{q}).$$

This allows computing the support of all practice convolutions: for the $\varepsilon = 2^{-53}$ of `binary64`, the condition $Q \log_2 Q \leq (2c\varepsilon)^{-1}$ is satisfied by

$$Q \leq 7 \cdot 10^{12} =: Q_{\max}. \tag{3.1.11}$$

Such long vectors would rarely be encountered in practice: even just storing a vector of $7 \cdot 10^{12}$ `binary64`s requires 56 terabytes of memory. One could store a very sparse vector of this length in much less space, but, anyhow, other convolution algorithms are likely to be better suited to this class of vectors, even when Q is much smaller.

3.1.4. The algorithm

By combining this reasoning about the support and Lemma 3.1.1, we create Checked FFT-C, listed in Algorithm 3.1, that can accurately compute some convolutions of non-negative vectors, even when there are zeros in the final convolution. More generally, Checked FFT-C can identify elements that are possibly inaccurate when the convolution is not guaranteed to be accurate. Note that $(\boldsymbol{p} * \boldsymbol{q})(k) = 0$ can only happen for some k if at least one of $\boldsymbol{p}$ and $\boldsymbol{q}$ contains a zero.

Algorithm 3.1 The Checked FFT-C algorithm. Given non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and $\alpha \geq 2$, Checked FFT-C returns $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ as computed by FFT-C, and a set I of indices for which the relative error may be larger than $1/\alpha$.

1: **procedure** Checked FFT-C($\boldsymbol{p}$, $\boldsymbol{q}$, α)
2: Compute $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ via FFT-C, and note the Q of Corollary 2.2.10 that was used.
3: Let I be the set of indices for which (3.1.2) does not hold.
4: **if** $I \neq \emptyset$ and $Q \leq Q_{\max}$ (see (3.1.11)) and $\boldsymbol{p}(k) = 0$ or $\boldsymbol{q}(k) = 0$ for some k **then**
5: Use FFT-C to compute the support of $\boldsymbol{p} * \boldsymbol{q}$ via Corollary 3.1.9,

$$I_{\text{sup}} = \operatorname{supp}(\boldsymbol{p} * \boldsymbol{q}) = \operatorname{supp}(\operatorname{filt}(\widetilde{\mathbf{1}_{\boldsymbol{p}>0} * \mathbf{1}_{\boldsymbol{q}>0}})).$$

6: Zero any entry $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(i)$ with $i \notin I_{\text{sup}}$.
7: Set $I \leftarrow I \cap I_{\text{sup}}$.
8: **end if**
9: **return** $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$, I.
10: **end procedure**

3.1.5. Complexity

Let $\boldsymbol{p} \in \mathbb{R}^m$ and $\boldsymbol{q} \in \mathbb{R}^n$ be vectors, and $Q \geq m + n - 1$ such that $Q = O(n + m)$. Checked FFT-C runs in $O(Q \log Q)$ time: it consists of

one or two FFT-C computations, each taking $O(Q \log Q)$ time, along with checking (3.1.2), searching $\boldsymbol{p}$ and $\boldsymbol{q}$ for a zero, zeroing some entries of $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ and computing a set intersection, all of which can be performed in $O(Q)$ time.

3.2. psFFT-C

What happens when Checked FFT-C cannot guarantee that all the values of the convolution are computed accurately? One way to fill in the gaps where FFT-C is not guaranteed to be accurate is to use NC to recompute $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(i)$ for each $i \in I$. However for cases when FFT-C fails to accurately compute a significant number of elements this approach has the same quadratic asymptotic cost as a direct NC itself, and thus is undesirable as a general strategy.

Example 3.2.1. Define a vector $\boldsymbol{p}$ of length $n+1$,

$$\boldsymbol{p} = (1, \underbrace{\varepsilon, \varepsilon, \ldots, \varepsilon}_{n \text{ copies}}).$$

The exact convolution $\boldsymbol{p} * \boldsymbol{p}$ has length $2n+1$, and value

$$\boldsymbol{p} * \boldsymbol{p} = \left(1, 2\varepsilon, 2\varepsilon + \varepsilon^2, \ldots, 2\varepsilon + (n-1)\varepsilon^2, n\varepsilon^2, \ldots, 2\varepsilon^2, \varepsilon^2\right)$$

Choose some $\alpha \geq 2$, then, since most entries are so small, they will not satisfy (3.1.2), and Checked FFT-C$(\boldsymbol{p}, \boldsymbol{p}, \alpha)$ will compute

$$\widetilde{\boldsymbol{p} * \boldsymbol{p}} = (1 + O(\varepsilon), O(\varepsilon), \ldots, O(\varepsilon))$$
$$I = \{1, 2, \ldots, 2n\}$$

That is, every entry other than the $1 + O(\varepsilon)$ at index 0 will not be guaranteed to have relative accuracy $1/\alpha$, and hence NC will have to be used to recompute $2n$ entries, with an average cost for each of $O(n)$. □

3.2.1. Partitioning

An alternative strategy that uses FFT-C to try to maintain good asymptotic run time performance is hinted by Lemma 3.1.5. As long as the non-zero elements of $\boldsymbol{v} * \boldsymbol{w}$ are sufficiently large, FFT-C will compute a convolution accurately, so if a convolution $\boldsymbol{v} * \boldsymbol{w}$ can be converted into one or more convolutions that are guaranteed to satisfy (3.1.6), then it can be computed via the efficient FFT-C.

Convolution is a bilinear operator, so one conversion strategy is simple: for non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of length m and n respectively, write

$$\boldsymbol{p} = \sum_{i=1}^{n_{\boldsymbol{p}}} \boldsymbol{p}_i \quad \text{and} \quad \boldsymbol{q} = \sum_{j=1}^{n_{\boldsymbol{q}}} \boldsymbol{q}_j$$

for some vectors $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$. The convolution $\boldsymbol{p} * \boldsymbol{q}$ can then be written

$$\boldsymbol{p} * \boldsymbol{q} = \sum_{i=1}^{n_{\boldsymbol{p}}} \sum_{j=1}^{n_{\boldsymbol{q}}} \boldsymbol{p}_i * \boldsymbol{q}_j. \tag{3.2.2}$$

In a practice, each term $\boldsymbol{p}_i * \boldsymbol{q}_j$ can of course only be approximated by some vector $\widetilde{\boldsymbol{p}_i * \boldsymbol{q}_j}$, which could be computed via NC, or FFT-C, or maybe some alternate strategy. However, guarantees about the error in these approximations translates into guarantees of the error in the overall approximation

$$\widetilde{\boldsymbol{p} * \boldsymbol{q}} = \sum_{i=1}^{n_{\boldsymbol{p}}} \widetilde{\sum_{j=1}^{n_{\boldsymbol{q}}}} \widetilde{\boldsymbol{p}_i * \boldsymbol{q}_j}. \tag{3.2.3}$$

Of course, we wish to perform each $\widetilde{\boldsymbol{p}_i * \boldsymbol{q}_j}$ via FFT-C to benefit from its almost linear performance. Thus to have assurances about the error we should select $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ such that Lemma 3.1.5 holds for each $\boldsymbol{p}_i * \boldsymbol{q}_j$. The asymptotic cost of this strategy is $O(n_{\boldsymbol{p}} n_{\boldsymbol{q}}(m+n)\log(m+n))$. When $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$ is small, as is typical, this can be much less than the $O(mn)$ cost of NC, representing a significant speed-up.

There is one major unresolved question with the strategy outlined above: how does one guarantee (3.1.6) for every $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ pair, without knowing the exact value of each $\boldsymbol{p}_i * \boldsymbol{q}_j$? Furthermore, to be usable in an algorithm that replaces NC, it must be possible to compute the $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ efficiently, that is, in time less than $O(mn)$.

One way to answer that question for a convolution $\boldsymbol{v} * \boldsymbol{w}$ is to guarantee (3.1.6) holds for the entire convolution by a conservative approximation of its first condition. In particular, the *extended dynamic range* $R(\boldsymbol{v})$ and $R(\boldsymbol{w})$ should not be too large, where $R(\boldsymbol{v})$ is defined for non-negative $\boldsymbol{v}$ as

$$R(\boldsymbol{v}) = \log \frac{\|\boldsymbol{v}\|_2}{\min_+ \boldsymbol{v}}. \tag{3.2.4}$$

Note that $R(\boldsymbol{v})$ is a variant of the conventional dynamic range, which uses the ℓ^∞ norm in place of the ℓ^2 norm. Also, note that the logarithm can be computed to any base, as long as that base is used consistently.

Controlling this R quantity gives the following corollary of Lemma 3.1.5.

Corollary 3.2.5. *Suppose $\boldsymbol{v}$ and $\boldsymbol{w}$ are non-negative vectors and $\alpha \geq 2$, if*

$$R(\boldsymbol{v}) + R(\boldsymbol{w}) \leq -\log(\alpha c K \varepsilon) \tag{3.2.6}$$

then (3.1.6) *holds, and hence so does the conclusion of Lemma 3.1.5.*

Proof. The condition (3.2.6) can be rearranged into

$$\min_+ \boldsymbol{v} \cdot \min_+ \boldsymbol{w} \geq \alpha c K \varepsilon \, \|\boldsymbol{v}\|_2 \, \|\boldsymbol{w}\|_2 \,,$$

and by non-negativity, $\min_+(\boldsymbol{v} * \boldsymbol{w}) \geq \min_+ \boldsymbol{v} \cdot \min_+ \boldsymbol{w}$. □

As stated above, (3.2.6) is a conservative condition, since $\min_+(\boldsymbol{v} * \boldsymbol{w}) = \min_+ \boldsymbol{v} \min_+ \boldsymbol{w}$ typically only occurs if $\boldsymbol{v}$ and $\boldsymbol{w}$ both have a minimal value at the same end, or if the vectors are sparse. In other cases, the smallest positive value may be significantly larger than that computed from the estimate. Unfortunately, this means that some convolutions that do satisfy (3.1.6) are incorrectly flagged as invalid, possibly forcing more work than necessary.

Example 3.2.7. Take any $x \in (0, 1/2)$ and consider the vector of length 3,

$$\boldsymbol{p} = (1, x, 1).$$

We have $\min_+(\boldsymbol{p} * \boldsymbol{p}) = 2x$, but $(\min_+ \boldsymbol{p})^2 = x^2$, underestimation by a factor of $2/x$, which can be arbitrarily large. □

However, despite being conservative in some cases, this condition is very useful, since, by design, it does not require knowing the true value $\boldsymbol{p} * \boldsymbol{q}$.

Almost all the necessary pieces are laid out, except the choice of $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ needs to guarantee that (3.2.6) holds for all pairs of $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$. The easiest way to guarantee this is to consider each of $\boldsymbol{p}$ and $\boldsymbol{q}$ in isolation, and choose the vectors $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ such that

$$R(\boldsymbol{p}_i) \leq -\frac{1}{2}\log(\alpha c K \varepsilon) \qquad R(\boldsymbol{q}_j) \leq -\frac{1}{2}\log(\alpha c K \varepsilon). \tag{3.2.8}$$

This can mean the vectors chosen are even more conservative than necessary, but, as we will see, it does allow for an efficient method to find vectors that will satisfy (3.2.6).

Example 3.2.9. Choose $K = 2$, so $Q = 4$ and choose $\alpha \geq 2$ such that $x = \sqrt{2\alpha c\varepsilon}/2 < 1$. Define vectors of length 2 as $\boldsymbol{p} = (1, x)$ and $\boldsymbol{q} = (1, 1)$. We have

$$\begin{aligned} R(\boldsymbol{p}) &= \log(\sqrt{1 + x^2}/x) \approx -\frac{1}{2}\log(2\alpha c\varepsilon) + \log 2 \\ R(\boldsymbol{q}) &= \frac{1}{2}\log 2 \end{aligned}$$

and hence $\boldsymbol{p}$ and $\boldsymbol{q}$ satisfy (3.2.6) as long as $\alpha c\varepsilon$ is sufficiently small, meaning $\boldsymbol{p} * \boldsymbol{q}$ can be computed accurately via FFT-C. On the other hand, $\boldsymbol{p}$ does not satisfy (3.2.8) in isolation, and hence $\boldsymbol{p}$ will be partitioned into, for instance, $(1, 0)$ and $(0, x)$, each of which does satisfy (3.2.8). Note however that this example is handled efficiently via Checked FFT-C. □

The conditions (3.2.8) can be evaluated efficiently, as (3.2.4) can be computed in linear time, much less than the quadratic complexity of NC. Furthermore, it can be evaluated in an incremental fashion: if $\boldsymbol{v}'$ differs to $\boldsymbol{v}$ at one known entry, then $R(\boldsymbol{v}')$ can be computed in $O(1)$ time from $R(\boldsymbol{v})$. This hints that it is an permissible strategy to build the splits $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$ in an iterative fashion.

The PITS (partition into thin stripes) procedure listed in Algorithm 3.2 uses condition (3.2.8) to partition a non-negative vector $\boldsymbol{p}$ into horizontal stripes that can be used to compute a convolution $\boldsymbol{p} * \cdot$ using FFT with relative accuracy α.

Specifically, given $\boldsymbol{p}$ of length n, it uses a greedy algorithm to compute a sequence $[b_1, \ldots, b_{n_{\boldsymbol{p}}}]$ of boundaries for partitions $\boldsymbol{p}_i$ which satisfy

$$\boldsymbol{p}_i(k) = \begin{cases} \boldsymbol{p}(k) & \text{if } b_{i+1} < \boldsymbol{p}(k) \leq b_i, \\ 0 & \text{otherwise,} \end{cases} \tag{3.2.10}$$

Algorithm 3.2 The PITS (Partition Into Thin Stripes) algorithm. It takes a pmf $\boldsymbol{p}$ of length n and returns a sequence of breakpoints b_i for partitions $\boldsymbol{p}_i$ defined by (3.2.10), each satisfying (3.2.8).

1: **procedure** PITS($\boldsymbol{p}$, α)
2: Find a permutation π so that $\boldsymbol{p} \circ \pi$ is non-increasing (sorted from largest to smallest).
3: Set $i \leftarrow 1$, $s_1 \leftarrow \boldsymbol{p}(\pi(1))^2$, $b_1 \leftarrow \boldsymbol{p}(\pi(1))$.
4: **for** $j \leftarrow 2 : n$ **do**
5: Set $x \leftarrow \boldsymbol{p}(\pi(j))$.
6: **if** $\log(\sqrt{s_i + x^2}/x) \leq -1/2 \log(\alpha c K \varepsilon)$ per (3.2.8) **then**
7: Update $s_i \leftarrow s_i + x^2$.
8: **else** ▷ *$\boldsymbol{p}_i$ can not be extended and still satisfy* (3.2.8).
9: Update $i \leftarrow i + 1$.
10: Set $s_i \leftarrow x^2$, $b_i \leftarrow x$.
11: **end if**
12: **end for**
13: Set $n_{\boldsymbol{p}} \leftarrow i$.
14: **return** $[b_1, \ldots, b_{n_{\boldsymbol{p}}}]$.
15: **end procedure**

where $b_{n_{\boldsymbol{p}}+1} = 0$.

The algorithm builds these partitions progressively. At any given iteration of the loop on line 4, the index i represents some partition $\boldsymbol{p}_i$ under consideration. Each partition starts as $\boldsymbol{p}_i = \boldsymbol{0}$, and the condition on line 6 is deciding whether to set $\boldsymbol{p}_i(\pi(j)) = \boldsymbol{p}(\pi(j))$. If this modified $\boldsymbol{p}_i$ is denoted as $\boldsymbol{p}_i'$, then the condition is checking if $\boldsymbol{p}_i'$ satisfies (3.2.8): by induction, it is clear that $s_i = \|\boldsymbol{p}_i\|_2^2$, and the non-increasing nature of the sequence of $\boldsymbol{p}(\pi(j))$ means that $x \leq \min_+ \boldsymbol{p}_i$, and hence $\|\boldsymbol{p}_i'\|_2 = \sqrt{s_i + x^2}$ and $\min_+ \boldsymbol{p}_i' = x$.

If the condition fails, that is $\boldsymbol{p}_i'$ does not satisfy (3.2.8), then the partition $\boldsymbol{p}_i$ is frozen, and the loop starts to build a new partition $\boldsymbol{p}_{i+1}$, starting with $\boldsymbol{p}_{i+1}(\pi(j)) = \boldsymbol{p}(\pi(j))$. Again, by the non-increasing nature of the sequence of elements of $\boldsymbol{p}$, this value will be the largest in $\boldsymbol{p}_{i+1}$, and hence serves as boundary b_{i+1}.

The partitions are treated abstractly in PITS, as this ensures it can run, and, in particular, the number of partitions $n_{\boldsymbol{p}}$ can be computed, in almost linear time for all vectors $\boldsymbol{p}$. If the partition vectors $\boldsymbol{p}_i$ were built concretely as the loop executes, then vectors with very large dynamic range would require creating many sparse splits, which, if each split $\boldsymbol{p}_i$ is stored in its entirety, requires $O(n)$ time to initialize to $\boldsymbol{0}$. Alternate strategies for storage may allow reducing this, but, as we will see later in aFFT-C, it is valuable to keep the cost of determining $n_{\boldsymbol{p}}$ as small as possible.

The $\boldsymbol{p}_i$ defined by (3.2.10) satisfy $\boldsymbol{p} = \sum_i \boldsymbol{p}_i$ as required to apply (3.2.3), and the supports of the $\boldsymbol{p}_i$ are pairwise disjoint, that is, $\operatorname{supp} \boldsymbol{p}_i \cap \operatorname{supp} \boldsymbol{p}_j = \emptyset$ for all $i \neq j$.

3.2.2. Shifting

As discussed, the complexity of computing a convolution $\boldsymbol{p} * \boldsymbol{q}$ via (3.2.3) is highly dependent on the product $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$, and minimising that product can result in large improvements to the performance of an algorithm based on partitioning.

The sFFT algorithm of [Kei05], discussed in Section 2.3, uses an exponential shift (2.3.3) for a similar purpose, and it applies equally well here: instead of splitting $\boldsymbol{p}$ and $\boldsymbol{q}$ and using them to compute $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$, one can instead perform an exponential shift with a factor θ, and split the resulting $\boldsymbol{p}_\theta$ and $\boldsymbol{q}_\theta$ into partitions $\boldsymbol{p}_{\theta,i}$ and $\boldsymbol{q}_{\theta,j}$. These partitions can be used in (3.2.3) to give $\widetilde{(\boldsymbol{p} * \boldsymbol{q})}_\theta$, and deducing an approximation to the true convolution just requires inverting the exponential shift.

In the context of general non-negative vectors $\boldsymbol{p}$, the normalisation of (2.3.3) by the moment-generating function of $\boldsymbol{p}$ (as if it were a pmf) is not strictly needed. However, as we will see later, working with vectors such that $\|\boldsymbol{v}\|_1 = 1$ makes things simpler, and so performing the normalisation helps there.

It is desirable to choose a θ that minimises the number of partitions required. One possibility would be to minimise the left hand side of (3.2.6), that is,

$$\theta_0 = \arg\min_\theta R(\boldsymbol{p}_\theta) + R(\boldsymbol{q}_\theta). \tag{3.2.11}$$

However, we found empirically that the number of partitions $n_{\boldsymbol{p}}$ is roughly proportional to $R(\boldsymbol{p})$ and hence

$$\theta_0 = \arg\min_\theta R(\boldsymbol{p}_\theta) \cdot R(\boldsymbol{q}_\theta) \tag{3.2.12}$$

generally results in a smaller $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$ implying a shorter run time.

The effect of the shifting is demonstrated visually in Figure 3.1. This shows the division of partitions computed by PITS for a pmf $\boldsymbol{p}$, and then for the shifted pmf $\boldsymbol{p}_{\theta_0}$ with θ_0 from (3.2.12). The reduction from 18 to just 6 corresponds to doing 9 times fewer FFT-C convolutions in (3.2.3).

For a more exhaustive examination we took a variety of pmfs $\boldsymbol{p}$ and calculated the number of partitions computed by PITS for the unshifted $\boldsymbol{p}$, and then for $\boldsymbol{p}_{\theta_0}$ with θ_0 computed both by (3.2.11) and for (3.2.12). A summary of the results is shown in Table 3.1, indicating that (3.2.12) works better in practice.

3.2.3. Computing the convolution

The final psFFT-C algorithm is listed in Algorithm 3.3. Given a desired relative accuracy $1/\alpha$, psFFT-C takes the output of PITS applied to two

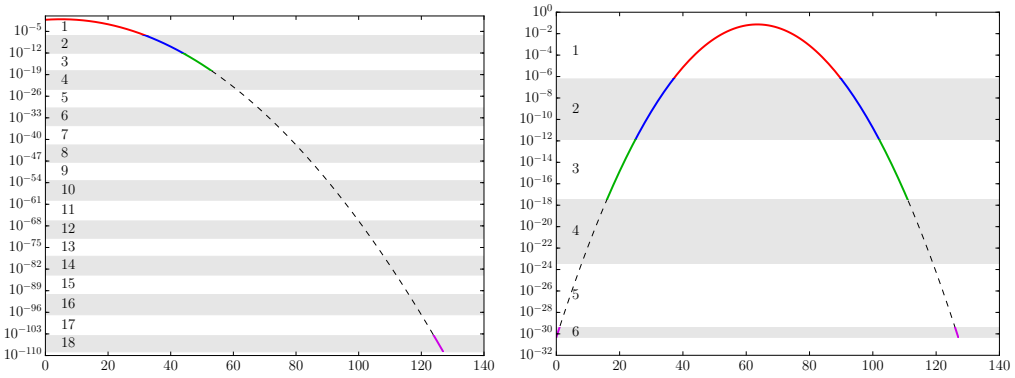

Figure 3.1 – The figures demonstrate the outcome of PITS applied to the same pmf $\boldsymbol{p}$ from Figure 2.1. The left plot shows the 18 components $\boldsymbol{p}_i$, each satisfying (3.2.8), that sum up to the original pmf $\boldsymbol{p}$. The plot on the right shows that, in contrast, when PITS is applied to $\boldsymbol{p}_\theta$ with θ determined from (3.2.12) (to optimize computing $\boldsymbol{p} * \boldsymbol{p}$) it yields only 6 components $\boldsymbol{p}_{\theta,i}$ each satisfying (3.2.8).

Minimization	Median	Mean	Maximum
No shifting	72	112	1156
(3.2.11)	65	110	756
(3.2.12)	60	87	424

Table 3.1 – Comparison of the number of partitions computed by PITS with no shifting and when shifting by θ_0 computed via (3.2.11) or (3.2.12) for the examples described in Section 3.5 of length $n = 4096$.

vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ shifted by θ_0 in the manner of (2.3.3), and uses them to compute $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ accurately.

psFFT-C does not bring together all its parts into a single procedure, because it can be far more efficient to combine psFFT-C with Checked FFT-C, which is best done by interleaving the choice of θ_0, the splitting of $\boldsymbol{p}$ and $\boldsymbol{q}$ via PITS and the final call to psFFT-C with other calculations, which will be carried out in detail below.

3.2.4. Complexity

Suppose psFFT-C is applied to $\boldsymbol{p}$ and $\boldsymbol{q}$ with lengths m and n respectively, and let $N = m + n - 1$ be the length of $\boldsymbol{p} * \boldsymbol{q}$ with $Q \geq N$ the length of the Fourier transforms used in FFT-C. The components of psFFT-C together have complexity $O(n_{\boldsymbol{p}} n_{\boldsymbol{q}} Q \log Q)$. There are three parts to analyse: the choice of θ_0, PITS (Algorithm 3.2) and psFFT-C listed in Algorithm 3.3.

The selection of θ_0 is performed via the one-dimensional optimisation problem (3.2.12), which typically requires only a few evaluations of the product $R(\boldsymbol{p}_\theta)R(\boldsymbol{q}_\theta)$, each of which takes $O(Q)$ time.

Algorithm 3.3 The psFFT-C (partitioned shifted FFT-C) algorithm. It takes two pmfs $\boldsymbol{p} \in (\mathbb{R}^+)^m$ and $\boldsymbol{q} \in (\mathbb{R}^+)^n$ that have had an exponential shift by θ_0 and have been partitioned via PITS, and computes a machine approximation to $\boldsymbol{p} * \boldsymbol{q}$ via FFT-C.

1: **procedure** psFFT-C(θ_0, $\boldsymbol{p}_{\theta_0}$, $[b_{\boldsymbol{p},i}]_{i=1}^{n_{\boldsymbol{p}}}$, $\boldsymbol{q}_{\theta_0}$, $[b_{\boldsymbol{q},j}]_{j=1}^{n_{\boldsymbol{q}}}$, α)
2: Slice $\boldsymbol{p}_{\theta_0,i}$ and $\boldsymbol{q}_{\theta_0,j}$ per (3.2.10), based on the boundaries $b_{\boldsymbol{p},i}$ and $b_{\boldsymbol{q},j}$ respectively.
3: For each i, j, compute $\widetilde{\boldsymbol{p}_{\theta_0,i} * \boldsymbol{q}}_{\theta_0,j}$ via FFT-C.
4: Compute

$$(\widetilde{\boldsymbol{p} * \boldsymbol{q}})_{\theta_0} = \sum_{i=1}^{n_{\boldsymbol{p}}} \sum_{j=1}^{n_{\boldsymbol{q}}} \mathrm{filt}(\widetilde{\boldsymbol{p}_{\theta_0,i} * \boldsymbol{q}}_{\theta_0,j}).$$

5: **return** $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$, deduced by undoing the θ_0 shift,

$$(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(k) = (\widetilde{\boldsymbol{p} * \boldsymbol{q}})_{\theta_0}(k) e^{-k\theta_0} M_{\boldsymbol{p}}(\theta_0) M_{\boldsymbol{q}}(\theta_0).$$

6: **end procedure**

An invocation of PITS on $\boldsymbol{p}$ consists of two parts: finding the permutation π (that is, sorting $\boldsymbol{p}$), which takes time $O(Q \log Q)$, and iterating over $\boldsymbol{p}$ to find the boundaries of the $n_{\boldsymbol{p}}$ stripes $\boldsymbol{p}_i$, which takes time $O(Q)$: each of the Q iterations of the loop is $O(1)$. Therefore the overall complexity of invoking PITS on both $\boldsymbol{p}_{\theta_0}$ and $\boldsymbol{q}_{\theta_0}$ is $O(Q \log Q)$.

The final component is of course psFFT-C in Algorithm 3.3. This first requires splitting the vectors into $\boldsymbol{p}_{\theta_0,i}$ and $\boldsymbol{q}_{\theta_0,j}$, requiring $O(Q(n_{\boldsymbol{p}} + n_{\boldsymbol{q}}))$ operations. It then performs $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$ FFT-C computations, each of which is $O(Q \log Q)$, followed by $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$ vector summations (each is $O(Q)$) and one shift inversion ($O(Q)$). The complexity of psFFT-C is hence $O(n_{\boldsymbol{p}} n_{\boldsymbol{q}} Q \log Q)$.

3.3. aFFT-C

Using psFFT-C directly can be quite expensive, since $n_{\boldsymbol{p}} n_{\boldsymbol{q}}$ can be large. Unfortunately, it can even have non-trivial overhead over a direct invocation of FFT-C when the latter is accurate, since the choice of the stripes $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$ via PITS is conservative. Fortunately, there is a middle ground for computing a convolution accurately, combining Checked FFT-C with both NC and psFFT-C to try to retain the performance of FFT-C when it is accurate, but still handling cases when it fails to deliver the desired accuracy as efficiently as possible.

An algorithm performing this balancing act, aFFT-C (accurate FFT-C), is listed in Algorithm 3.4. It takes two non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and computes an approximation $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ to $\boldsymbol{p} * \boldsymbol{q}$, such that all entries have a relative accuracy of $1/\alpha$ or better.

As mentioned above, the three components of psFFT-C are interleaved with other computations. Specifically, they are interleaved with calls to

Algorithm 3.4 The aFFT-C (accurate FFT convolution) algorithm. It takes two non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of length m and n respectively, and some $\alpha \geq 2$, and computes an approximation $\boldsymbol{c} = \widetilde{\boldsymbol{p} * \boldsymbol{q}}$, such that $\operatorname{rel}(\boldsymbol{c}(k), (\boldsymbol{p} * \boldsymbol{q})(k)) < 1/\alpha$ for each k. It builds on psFFT-C, but incorporates several fast paths based on Checked FFT-C (Algorithm 3.1) to try to ensure maximal performance.

procedure aFFT-C($\boldsymbol{p}$, $\boldsymbol{q}$, α)
  Set $N = m + n - 1$ and choose $Q = 2^K \geq N$.
  Compute $\boldsymbol{c}_1, I_1 \leftarrow$ Checked FFT-C$(\boldsymbol{p}, \boldsymbol{q}, \alpha)$.
  if $2CQ \log Q \geq N\,|I_1|$ **then** ▷ *Direct FFT-C is accurate for sufficiently many entries.*
    Recompute the entries $\boldsymbol{c}_1(i)$ via NC, for each $i \in I_1$.
    return $\boldsymbol{c}_1$.
  end if
  Compute θ_0 via (3.2.12) and then $\boldsymbol{p}_{\theta_0}$ and $\boldsymbol{q}_{\theta_0}$ as in (2.3.3).
  Compute $\boldsymbol{c}_2, I_2 \leftarrow$ Checked FFT-C$(\boldsymbol{p}_{\theta_0}, \boldsymbol{q}_{\theta_0}, \alpha)$.
  if $2CQ \log Q \geq N\,|I_2|$ **then** ▷ *FFT-C with a shift is accurate for sufficiently many entries.*
    Recompute the entries $\boldsymbol{c}_2(i)$ via NC, for each $i \in I_2$.
    Deduce $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ by undoing the θ_0 shift of $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})_{\theta_0} = \boldsymbol{c}_2$ like psFFT-C.
  else
    Compute the boundaries $[b_{\boldsymbol{p},i}]_{i=1}^{n_{\boldsymbol{p}}} \leftarrow \mathrm{PITS}(\boldsymbol{p}_{\theta_0}, \alpha)$ and $[b_{\boldsymbol{q},j}]_{j=1}^{n_{\boldsymbol{q}}} \leftarrow \mathrm{PITS}(\boldsymbol{q}_{\theta_0}, \alpha)$.
    if $Cn_{\boldsymbol{p}}n_{\boldsymbol{q}}Q \log Q \geq N\,|I_2|$ **then** ▷ *NC estimated to be faster.*
      Recompute the entries $\boldsymbol{c}_2(i)$ via NC, for each $i \in I_2$.
      Deduce $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ from $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})_{\theta_0} = \boldsymbol{c}_2$ as above.
    else
      $\widetilde{\boldsymbol{p} * \boldsymbol{q}} =$ psFFT-C$(\theta_0, \boldsymbol{p}_{\theta_0}, [b_{\boldsymbol{p},i}], \boldsymbol{q}_{\theta_0}, [b_{\boldsymbol{q},j}], \alpha)$.
    end if
  end if
  return $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$.
end procedure

Checked FFT-C and decision points about whether it is more efficient to use NC to recompute any potentially inaccurate entries of that FFT-C convolution, or to press ahead and finally end up in psFFT-C itself.

Both psFFT-C and the combination of Checked FFT-C and NC are able to accurately compute a convolution $\boldsymbol{p} * \boldsymbol{q}$, and so their performance is the main reason to choose between them. Therefore, the decisions are driven by estimating the cost of the two options (NC or psFFT-C) and choosing the fastest one. This is most easily done via the asymptotic behaviour of the two algorithms: $O(N\ell)$ for NC to recompute the ℓ potentially inaccurate entries compared to $O(n_{\boldsymbol{p}}n_{\boldsymbol{q}}Q \log Q)$ for psFFT-C with $n_{\boldsymbol{p}}$, $n_{\boldsymbol{q}}$ stripes. Since the

asymptotic behaviour notation ignores constant factors, we must empirically estimate a constant of proportionality C, the ratio of the constant factors, and thus compare $Cn_{\boldsymbol{p}}n_{\boldsymbol{q}}Q\log Q$ to $N\ell$.

The aFFT-C algorithm calls Checked FFT-C twice, first to ensure there is minimal overhead over FFT-C in cases when it is accurate, and then again to take advantage of the reduced dynamic range of the shift.

The first call to Checked FFT-C on line 3 is designed to handle the case when $\boldsymbol{p}$ and $\boldsymbol{q}$ have sufficiently small dynamic range or happen to have their values arranged such that direct FFT-C is accurate almost everywhere. This is the first non-trivial operation performed by aFFT-C, so that there is minimal overhead of aFFT-C over FFT-C in this case. In particular, the true value $n_{\boldsymbol{p}}n_{\boldsymbol{q}}$ has not yet been computed, and so a lower bound of 2 is used. This bound is motivated by the reasoning that if both Checked FFT-C invocations fail ($I_1, I_2 \neq \emptyset$ for $x = 1, 2$), then PITS will partition at least one of the pmfs at least once, making $n_{\boldsymbol{p}} \geq 2$ or $n_{\boldsymbol{q}} \geq 2$. This allows for a cheap decision on line 4 about whether sufficiently few entries $|I_1|$ will need to be recomputed via NC.

The second call to Checked FFT-C uses the exponential-shifted vectors $\boldsymbol{p}_{\theta_0}$ and $\boldsymbol{q}_{\theta_0}$. The shift is likely to decrease the dynamic range of the vectors, and so Checked FFT-C is more likely to be successful. The same estimation of $n_{\boldsymbol{p}}n_{\boldsymbol{q}}$ is used as in the previous part, giving the same easily computed heuristic for determining if $|I_2|$ is sufficiently small on line 10.

Note that in rare cases it may be that PITS would compute $n_{\boldsymbol{p}}n_{\boldsymbol{q}} = 1$, as splitting is not truly guaranteed when the two Checked FFT-C calls do not compute all entries accurately: the retroactive analysis needs an error threshold that is larger by a factor of $(\alpha + 1)/\alpha$ than the more precise prospective bound (3.1.6) used by psFFT-C. However, this factor is typically small (it has a maximum value of 3/2) and the approximations used to guarantee (3.1.6) are conservative, meaning splits will be required more often than strictly necessary.

If falling back to NC is still estimated to be more expensive than psFFT-C with $n_{\boldsymbol{p}}n_{\boldsymbol{q}} = 2$, the boundaries of stripes $\boldsymbol{p}_{\theta_0,i}$ and $\boldsymbol{q}_{\theta_0,j}$ are finally computed via PITS. This gives the exact values of $n_{\boldsymbol{p}}n_{\boldsymbol{q}}$ and hence a more accurate comparison of the cost of completing the psFFT-C calculation or filling in any missing values with NC. This final decision occurs on line 15.

We can take advantage of this condition to further reduce the work required in some cases, since it means the complete partitioning is only needed if the condition on line 15 chooses psFFT-C over NC. Let $M = N\,|I_2|\,/(CQ\log Q)$, and then if $n_{\boldsymbol{p}}n_{\boldsymbol{q}} > M$ we can be sure that the output of PITS is not used. Thus, we can pass this M to PITS to allow it to abort when $n_{\boldsymbol{p}}n_{\boldsymbol{q}} > M$, passing the baton straight to NC. When PITS is applied first to $\boldsymbol{p}$ we conservatively estimate $n_{\boldsymbol{q}} = 1$, but for the application to $\boldsymbol{q}$ we know $n_{\boldsymbol{p}}$ exactly. The estimation of the product $n_{\boldsymbol{p}}n_{\boldsymbol{q}}$ can be improved in some cases, such as when computing $\boldsymbol{p} * \boldsymbol{p}$: it is guaranteed that $n_{\boldsymbol{p}} = n_{\boldsymbol{q}}$

and hence a limit of $n_{\boldsymbol{p}}^2 < M$ works. Notably, this case also only needs to invoke PITS once, since all the partitions will be identical.

3.3.1. Complexity

Suppose the vector $\boldsymbol{p}$ has length m and the vector $\boldsymbol{q}$ has length n, and the FFT-C convolutions are performed on vectors of length $Q \geq m + n - 1$. As a whole, computing $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ via aFFT-C has complexity approximately $O(\min(n_{\boldsymbol{p}} n_{\boldsymbol{q}} Q \log Q, Q^2))$.

Firstly, all of the operations up to and including the invocations of PITS have complexity $O(Q \log Q)$ or less. We saw before, in Section 3.1.5, that the calls to Checked FFT-C have almost linear run time, $O(Q \log Q)$. Estimating the relative run times of the two algorithms is efficient, possibly as low as $O(1)$ but certainly no more $O(Q)$, depending on the representation of the sets I_x for $x \in \{1, 2\}$.

Computing the shift θ_0 requires some unknown number of computations of $R(\boldsymbol{p}_\theta)$ and $R(\boldsymbol{q}_\theta)$, which take $O(Q)$ time. This is not easy to bound directly, for arbitrary $\boldsymbol{p}$ and $\boldsymbol{q}$, but efficient algorithms such as Brent's method [Bre73], as implemented under the name `fminbound` in SciPy [JOP+01] or `optimize` in R, mean that in practice we have never seen this take a noticeable fraction of the overall run time.

The remaining code up to the invocations of PITS are first the fallback to NC when it is estimated to be more efficient, and then the call to PITS itself. The former only occur if the number of missing elements satisfies $|I_x| \leq C \log Q$, for either $x = 1, 2$, and hence the time required to recompute these missing elements is bounded by $O(Q \log Q)$. We saw in Section 3.2.4 that PITS also has complexity $O(m \log m)$ for $\boldsymbol{p}$ and $O(n \log n)$ for $\boldsymbol{q}$, making the two invocations $O(Q \log Q)$. Summing all of these gives the stated $O(Q \log Q)$ cost.

This just leaves the cost of the computations that depend on the number of partitions reported by PITS: the two possibilities are the $O(Q^2)$ cost of NC, or the $O(n_{\boldsymbol{p}} n_{\boldsymbol{q}} Q \log Q)$ cost of psFFT-C, based on which is estimated to be the least expensive. This separation gives the overall complexity mentioned above.

3.4. Convolving with a lower bound

In some situations, such as computing p-values in the manner of sFFT, one is only interested in the "larger" values of the convolution $\boldsymbol{p} * \boldsymbol{q}$. The precise way in which "large" is defined can differ, so we look at two. The first, "trim then convolve" (TtC), is the easiest to implement and as we will see in Section 4.3.1, works well as a component of algorithms that may do several pairwise convolutions. The second, "convolve then trim" (CtT), gives a result that is more intuitive when one is performing a single pairwise convolution, since, unlike TtC, it directly guarantees the accuracy of all values that are sufficiently large.

Before analysing these two options, we introduce a useful piece of notation.

Definition 3.4.1. *Let $\boldsymbol{v} \in \mathbb{R}^N$ be a vector, then define*

$$\boldsymbol{v}_{\geq \Delta} = \boldsymbol{v} \odot \mathbf{1}_{v \geq \Delta},$$
$$\boldsymbol{v}_{< \Delta} = \boldsymbol{v} \odot \mathbf{1}_{v < \Delta}.$$

These are truncations of $\boldsymbol{v}$, and satisfy $\boldsymbol{v} = \boldsymbol{v}_{\geq \Delta} + \boldsymbol{v}_{< \Delta}$.

3.4.1. Trim then convolve

One way to avoid computing small values of $\boldsymbol{p} * \boldsymbol{q}$ is to approximate it by computing something like $\boldsymbol{p}_{\geq \Delta} * \boldsymbol{q}_{\geq \Delta}$ for some lower bound Δ, that is, trimming the two vectors before performing the convolution.

Definition 3.4.2. *Let $\boldsymbol{p}$ and $\boldsymbol{q}$ be non-negative vectors, and take some $\alpha \geq 2$ and $\Delta \geq 0$, then an approximation $\boldsymbol{c}$ is said to have the* trim then convolve (TtC) *guarantee if it satisfies*

$$\left(1 - \frac{1}{\alpha}\right)(\boldsymbol{p}_{\geq \Delta} * \boldsymbol{q}_{\geq \Delta})(k) \leq \boldsymbol{c}(k) \leq \left(1 + \frac{1}{\alpha}\right)(\boldsymbol{p} * \boldsymbol{q})(k). \tag{3.4.3}$$

for all k.

This does not ensure that $\boldsymbol{c}$ accurately approximates every element such that $(\boldsymbol{p} * \boldsymbol{q})(k) > \Delta$: the trimming may mean that values close to Δ are computed very inaccurately.

Example 3.4.4. Fix some $\Delta \in (0, 1)$ and integer $n \geq 1$, define $x = \Delta - \delta$ for some small $0 < \delta < \Delta/2$ and

$$\boldsymbol{p} = (1, \underbrace{x, \ldots, x}_{n \text{ copies}}).$$

We have $\boldsymbol{p}_{\geq \Delta} = (1, 0, \ldots, 0)$, since $x < \Delta$, and hence

$$\boldsymbol{p} * \boldsymbol{p} = (1, 2x, 2x + x^2, \ldots, 2x + (n-1)x^2, nx^2, \ldots, x^2)$$
$$\boldsymbol{p}_{\geq \Delta} * \boldsymbol{p}_{\geq \Delta} = (1, 0, \ldots, 0).$$

There are at least n elements larger than Δ in the true convolution that are computed as zero in the truncated one, and thus have a relative error of 1, and, ignoring terms of order δ, the absolute error is as large as $2x + (n-1)x^2 \approx 2\Delta + (n-1)\Delta^2$. □

Fortunately, the lost mass can be quantified.

Lemma 3.4.5. *Given non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$, and a lower bound $\Delta \geq 0$, we have*

$$0 \leq (\boldsymbol{p} * \boldsymbol{q})(k) - (\boldsymbol{p}_{\geq \Delta} * \boldsymbol{q}_{\geq \Delta})(k) \leq \Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1) \tag{3.4.6}$$

for all k.

Proof. Since the vectors are non-negative and no element of the vectors $\boldsymbol{p}_{\geq\Delta}$ and $\boldsymbol{q}_{\geq\Delta}$ is larger than the corresponding element of $\boldsymbol{p}$ and $\boldsymbol{q}$ respectively, the left hand inequality of (3.4.6) is clear.

We have $\boldsymbol{p} = \boldsymbol{p}_{\geq\Delta} + \boldsymbol{p}_{<\Delta}$ and similarly for $\boldsymbol{q}$, and hence

$$\begin{aligned}\|\boldsymbol{p} * \boldsymbol{q} - \boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta}\|_\infty &= \|\boldsymbol{p}_{<\Delta} * \boldsymbol{q}_{\geq\Delta} + \boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{<\Delta} + \boldsymbol{p}_{<\Delta} * \boldsymbol{q}_{<\Delta}\|_\infty \\ &= \|\boldsymbol{p}_{<\Delta} * \boldsymbol{q} + \boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{<\Delta}\|_\infty \\ &\leq \|\boldsymbol{p}_{<\Delta}\|_\infty \|\boldsymbol{q}\|_1 + \|\boldsymbol{p}_{\geq\Delta}\|_1 \|\boldsymbol{q}_{<\Delta}\|_\infty \\ &\leq \Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1)\end{aligned}$$

where we used $\|\boldsymbol{x} * \boldsymbol{y}\|_\infty \leq \|\boldsymbol{x}\|_\infty \|\boldsymbol{y}\|_1$. This proves the right hand side of (3.4.6). □

This lemma is fairly precise, as indicated by Example 3.4.4: in that example, $\|\boldsymbol{p}\|_1 \approx 1 + n\Delta$, and hence the lemma implies $\boldsymbol{p}_{\geq\Delta} * \boldsymbol{p}_{\geq\Delta}$ may underestimate $\boldsymbol{p} * \boldsymbol{p}$ by at most $\Delta(2 + 2n\Delta)$, which differs from the quantity observed in that example by only $(n+1)\Delta^2$.

As we have seen before, in FFT-C, a bound like this on the absolute error can be translated into a bound on the relative accuracy.

Corollary 3.4.7. *Suppose $\boldsymbol{p}$, $\boldsymbol{q}$ and Δ are as in Lemma 3.4.5 and $\alpha \geq 1$, then any element k such that*

$$(\boldsymbol{p} * \boldsymbol{q})(k) \geq \alpha\Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1)$$

satisfies

$$\operatorname{rel}\left((\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta})(k), (\boldsymbol{p} * \boldsymbol{q})(k)\right) \leq \frac{1}{\alpha}.$$

Proof. Let $x = (\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta})(k)$ and $y = (\boldsymbol{p} * \boldsymbol{q})(k)$, then

$$\operatorname{rel}(x, y) = \frac{|x - y|}{y} \leq \frac{\Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1)}{\alpha\Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1)} = \frac{1}{\alpha},$$

as desired. □

This result can be extended to give bounds on the relative error in an approximation to $\boldsymbol{p} * \boldsymbol{q}$ in a manner similar to Lemma 3.1.1.

Corollary 3.4.8. *Suppose $\boldsymbol{p}$, $\boldsymbol{q}$ and Δ are as in Lemma 3.4.5, and $\alpha, \alpha' \geq 2$ and $\boldsymbol{c}$ is a vector that satisfies* (3.4.3) *with coefficients α and Δ, then any element k such that*

$$(\boldsymbol{p} * \boldsymbol{q})(k) \geq \alpha'\Delta(\|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1)$$

satisfies

$$\operatorname{rel}(\boldsymbol{c}(k), (\boldsymbol{p} * \boldsymbol{q})(k)) \leq \frac{1}{\alpha} + \frac{1}{\alpha'} - \frac{1}{\alpha\alpha'} < \frac{1}{\alpha} + \frac{1}{\alpha'}$$

Proof. For our k, Corollary 3.4.7 implies (3.4.3) can be rewritten to

$$\left(1 - \frac{1}{\alpha}\right)\left(1 - \frac{1}{\alpha'}\right)(\boldsymbol{p} * \boldsymbol{q})(k) \leq \boldsymbol{c}(k) \leq \left(1 + \frac{1}{\alpha}\right)(\boldsymbol{p} * \boldsymbol{q})(k).$$

This gives

$$\begin{aligned}\operatorname{rel}(\boldsymbol{c}(k), (\boldsymbol{p} * \boldsymbol{q})(k)) &\le \max\left(\left|\left(1 - \frac{1}{\alpha}\right)\left(1 - \frac{1}{\alpha'}\right) - 1\right|, \frac{1}{\alpha}\right) \\ &= \frac{1}{\alpha} + \frac{1}{\alpha'} - \frac{1}{\alpha\alpha'}.\end{aligned}$$

□

This is a theoretical analysis of TtC, but how can it be implemented in practice? Both the Checked FFT-C and psFFT-C components of aFFT-C benefit from having a lower bound.

One way to compute a vector $\boldsymbol{c}$ satisfying (3.4.3) with psFFT-C is to literally compute $\widetilde{\boldsymbol{p}_{\ge\Delta} * \boldsymbol{q}_{\ge\Delta}}$, by replacing the input $\boldsymbol{p}$ and $\boldsymbol{q}$ with their trimmed versions. The guarantees of psFFT-C ensure that the computed approximation has relative error less than $1/\alpha$, ensuring that it satisfies (3.4.3). These vectors have smaller dynamic range and fewer non-zero elements, and so PITS may be able to perform fewer splits, making psFFT-C and hence aFFT-C faster.

On the other hand, handling a lower bound in Checked FFT-C can be a little more subtle. One possibility, Bounded Checked FFT-C, for doing so is listed in Algorithm 3.5. The original Checked FFT-C gives a vector $\boldsymbol{c}$ and a set of indices I where $\boldsymbol{c}$ is not guaranteed to be accurate, and Bounded Checked FFT-C is similar. It guarantees that (3.4.3) holds for $\boldsymbol{c}(k)$ such that $k \notin I$.

Algorithm 3.5 The Bounded Checked FFT-C algorithm with the TtC guarantee. Given non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and scalars $\alpha \ge 2$ and $\Delta \ge 0$, Bounded Checked FFT-C returns a vector $\boldsymbol{c} = \widetilde{\boldsymbol{p} * \boldsymbol{q}}$ as computed by FFT-C and a set I of indices such that (3.4.3) holds for $\boldsymbol{c}$ for all $k \notin I$.

procedure Bounded Checked FFT-C($\boldsymbol{p}$, $\boldsymbol{q}$, α, Δ)
  Compute $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ via FFT-C, and note the Q of Corollary 2.2.10 that was used.
  Let I be the set of indices for which (3.1.2) does not hold.
  if $I \ne \emptyset$ and $Q \le Q_{\max}$ and $\boldsymbol{p}(i) < \Delta$ or $\boldsymbol{q}(i) < \Delta$ for some i **then**
    Use FFT-C to compute the support of $\boldsymbol{p}_{\ge\Delta} * \boldsymbol{q}_{\ge\Delta}$

$$I_{\text{sup}} = \operatorname{supp}(\boldsymbol{p}_{\ge\Delta} * \boldsymbol{q}_{\ge\Delta}) = \operatorname{supp}(\operatorname{filt}(\widetilde{\mathbf{1}_{\boldsymbol{p}\ge\Delta} * \mathbf{1}_{\boldsymbol{q}\ge\Delta}})).$$

    Zero any entry $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(i)$ with $i \notin I_{\text{sup}}$.
    Set $I \leftarrow I \cap I_{\text{sup}}$.
  end if
  return $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$, I.
end procedure

One way to use a lower bound in Checked FFT-C is to trim the vectors before doing any other work, effectively computing

$$\text{Checked FFT-C}(\boldsymbol{p}_{\ge\Delta}, \boldsymbol{q}_{\ge\Delta}, \alpha).$$

However, doing this can be unhelpful for the comparison with (3.1.2). The trimming removes mass from the vectors and will hence result in smaller computed values, and making it possible that, for some k,

$$\begin{aligned}(\widetilde{\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta}})(k) &< (\alpha+1)cK\varepsilon \left\|\boldsymbol{p}_{\geq\Delta}\right\|_2 \left\|\boldsymbol{q}_{\geq\Delta}\right\|_2 \\ &< (\alpha+1)cK\varepsilon \left\|\boldsymbol{p}\right\|_2 \left\|\boldsymbol{q}\right\|_2 \leq (\widetilde{\boldsymbol{p} * \boldsymbol{q}})(k).\end{aligned}$$

That is, FFT-C applied to the untrimmed vectors satisfies (3.1.2), but does not when applied to the trimmed one. This argues in favour of just using Checked FFT-C on $\boldsymbol{p}$ and $\boldsymbol{q}$ themselves, which will satisfy (3.4.3).

On the other hand, trimming may reduce the support of the convolution, since the trimmed vectors will have smaller support: $\operatorname{supp}(\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta}) \subseteq \operatorname{supp}(\boldsymbol{p} * \boldsymbol{q})$. Decreasing that support means that we can identify more elements that satisfy (3.4.3), since we can efficiently compute exactly the k for which $(\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta})(k) = 0$, and the bound will hence be satisfied by setting corresponding element $\boldsymbol{c}(k) = 0$ in the computed approximation. This has the consequence of reducing the size of the set I of indices with unknown accuracy, and hence decreases how much of the vector may need to be computed via NC.

In summary, there are some benefits with using $\boldsymbol{p} * \boldsymbol{q}$, and benefits with $\boldsymbol{p}_{\geq\Delta} * \boldsymbol{q}_{\geq\Delta}$. Fortunately, there are no trade-offs needed, since we still achieve the goal by using the former when it is best (for finding the values of the convolution), and the latter when it is best (for finding the support).

Example 3.4.9. Assume $\varepsilon = 2^{-53}$ as for `binary64`. Define $\boldsymbol{p} = (1, 1/2, 1/4)$, and choose $\alpha = 10^{14}$ and $\Delta = 1/2$.

We will see that the choice to use both $\boldsymbol{p}$ and $\boldsymbol{p}_{\geq\Delta}$ at different times will give Bounded Checked FFT-C$(\boldsymbol{p}, \boldsymbol{p}, \alpha, \Delta) = (\boldsymbol{c}, \emptyset)$, for some $\boldsymbol{c} \approx (1, 1, 3/4, 0, 0)$, whereas using $\boldsymbol{p}$ everywhere or $\boldsymbol{p}_{\geq\Delta}$ everywhere would result in a non-empty set of potentially inaccurate indices.

We of course have $\boldsymbol{p}_{\geq\Delta} = (1, 1/2, 0)$, and hence

$$\begin{aligned}\boldsymbol{c}_1 = \boldsymbol{p} * \boldsymbol{p} &= (1, 1, 3/4, 1/4, 1/16) \\ \boldsymbol{c}_2 = \boldsymbol{p}_{\geq\Delta} * \boldsymbol{p}_{\geq\Delta} &= (1, 1, 1/4, 0, 0),\end{aligned}$$

with the corresponding FFT-C approximations $\widetilde{\boldsymbol{c}_1}$ and $\widetilde{\boldsymbol{c}_2}$. Including the trailing zeros, these convolutions are of length 5, and hence our transforms can be of length $8 = 2^3$, giving $K = 3$ and $c = 15$ in Lemma 3.1.1. The error bounds of FFT-C for computing these two convolutions are therefore, respectively,

$$\begin{aligned}E_1 &= 15 \cdot 3 \cdot 2^{-53} \cdot \left\|\boldsymbol{p}\right\|_2^2 \approx 6.6 \cdot 10^{-15} \\ E_2 &= 15 \cdot 3 \cdot 2^{-53} \cdot \left\|\boldsymbol{p}_{\geq\Delta}\right\|_2^2 \approx 6.2 \cdot 10^{-15},\end{aligned}$$

and hence the bounds for (3.1.2) are

$$\begin{aligned}B_1 &= (10^{14} + 1)E_1 \approx 0.66 \\ B_2 &= (10^{14} + 1)E_2 \approx 0.62.\end{aligned}$$

Since the E_i are sufficiently small, we are guaranteed to have $\widetilde{\boldsymbol{c}_1}(k) \geq B_1$ for $k = 0, 1, 2$ and $\widetilde{\boldsymbol{c}_1}(k) < B_1$ elsewhere, and at the same time $\widetilde{\boldsymbol{c}_2}(k) \geq B_2$ for only $k = 0, 1$ with $\widetilde{\boldsymbol{c}_2}(k) < B_2$ for the remaining $k = 2, 3, 4$. In other words, the set of indices for which (3.1.2) does not guarantee accuracy is $I_1 = \{3, 4\}$ for the first convolution but $I_2 = \{2, 3, 4\}$ for the second, demonstrating that using the untrimmed vector $\boldsymbol{p}$ is better.

The first convolution has full support $I_{\text{sup},1} = \operatorname{supp} \boldsymbol{c}_1 = \{0, \ldots, 4\}$, but the trimmed one does not, specifically, $I_{\text{sup},2} = \operatorname{supp} \boldsymbol{c}_2 = \{0, 1, 2\}$. Note that $I_1 \cap I_{\text{sup},2} = \emptyset$, guaranteeing that (3.4.3) holds for all $k = 0, \ldots, 4$. □

There is one additional complication: performing an exponential shift to reduce dynamic range will disturb the comparison. In general, if $\theta \neq 0$, then clearly neither $(\boldsymbol{p}_\theta)_{\geq\Delta} = \boldsymbol{p}_{\geq\Delta}$ nor $(\boldsymbol{p}_\theta)_{\geq\Delta} = (\boldsymbol{p}_{\geq\Delta})_\theta$ hold. Thus, convolutions with a lower bound must be careful to either not perform a shift, or to first handle the lower bound by trimming, and only after that perform the shift.

All of these parts can be drawn together to create Bounded aFFT-C, listed in Algorithm 3.6.

Algorithm 3.6 The Bounded aFFT-C algorithm, a version of aFFT-C that gives the TtC guarantee. It takes two non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ of length m and n respectively, and some $\alpha \geq 2$ and a bound $\Delta \geq 0$, and computes an approximation $\boldsymbol{c} = \widetilde{\boldsymbol{p} * \boldsymbol{q}}$, such that (3.4.3) holds. It is very similar to the original aFFT-C, listed in Algorithm 3.4, with the notable removal of the exponential shift.

procedure Bounded aFFT-C($\boldsymbol{p}$, $\boldsymbol{q}$, α, Δ)
  Set $N = m + n - 1$ and choose $Q = 2^K \geq N$.
  Compute $\boldsymbol{c}_1, I_1 \leftarrow$ Bounded Checked FFT-C$(\boldsymbol{p}, \boldsymbol{q}, \alpha, \Delta)$.
  if $2CQ \log Q \geq N\,|I_1|$ **then** ▷ *Direct FFT-C is accurate for sufficiently many entries.*
    Recompute the entries $\boldsymbol{c}_1(i)$ via NC, for each $i \in I_1$.
    return $\boldsymbol{c}_1$.
  end if
  Compute the boundaries $[b_{\boldsymbol{p},i}]_{i=1}^{n_{\boldsymbol{p}}} \leftarrow \text{PITS}(\boldsymbol{p}_{\geq\Delta}, \alpha)$ and $[b_{\boldsymbol{q},j}]_{j=1}^{n_{\boldsymbol{q}}} \leftarrow \text{PITS}(\boldsymbol{q}_{\geq\Delta}, \alpha)$.
  if $Cn_{\boldsymbol{p}}n_{\boldsymbol{q}}Q \log Q \geq N\,|I_2|$ **then** ▷ *NC estimated to be faster.*
    Recompute the entries $\boldsymbol{c}_2(i)$ via NC, for each $i \in I_2$.
    return $\boldsymbol{c}_2$.
  else
    $\widetilde{\boldsymbol{p} * \boldsymbol{q}} = \text{psFFT-C}(0, \boldsymbol{p}_{\geq\Delta}, [b_{\boldsymbol{p},i}], \boldsymbol{q}_{\geq\Delta}, [b_{\boldsymbol{q},j}], \alpha)$.
    return $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$.
  end if
end procedure

3.4.2. Convolve then trim

As discussed briefly earlier, the condition (3.4.3) is often not the most useful constraint on the accuracy of a convolution with a bound. It may be that one wishes to compute a convolution such that all elements larger than some $\Delta \geq 0$ are computed accurately, and smaller ones are either irrelevant or, for instance, only need to be guaranteed to not be larger than their true value. As usual, due to floating point calculations, these assurances of accuracy have to be considered as bounded relative error.

Definition 3.4.10. *Let $\boldsymbol{p}$ and $\boldsymbol{q}$ be non-negative vectors, and take some $\alpha \geq 2$ and $\Delta \geq 0$, then an approximation $\boldsymbol{c}$ is said to have the* convolve then trim (CtT) *guarantee if it satisfies*

$$\left(1 - \frac{1}{\alpha}\right)(\boldsymbol{p} * \boldsymbol{q})_{\geq \Delta}(k) \leq \boldsymbol{c}(k) \leq \left(1 + \frac{1}{\alpha}\right)(\boldsymbol{p} * \boldsymbol{q})(k) \tag{3.4.11}$$

for all k.

This is similar to Definition 3.4.2, except it uses $(\boldsymbol{p} * \boldsymbol{q})_{\geq \Delta}$ instead of $\boldsymbol{p}_{\geq \Delta} * \boldsymbol{q}_{\geq \Delta}$ in the lower bound. This has the consequence that $\operatorname{rel}(\boldsymbol{c}(k), (\boldsymbol{p} * \boldsymbol{q})(k)) \leq 1/\alpha$ for k such that $(\boldsymbol{p} * \boldsymbol{q})(k) \geq \Delta$, but is slightly stronger than just that condition, as it still controls the computed value of small elements: an element with true value $x < \Delta$ is never computed to be larger than $(1 + 1/\alpha)x$.

Neither (3.4.3) nor (3.4.11) imply the other for fixed α and Δ, but the former can be used to get the latter guarantee by choosing coefficients α' and Δ' for (3.4.3) appropriately smaller than α and Δ.

Lemma 3.4.12. *Suppose $\boldsymbol{p}$ and $\boldsymbol{q}$ are non-negative vectors, $\alpha \geq 2$ and $\Delta \geq 0$. Let $S = \|\boldsymbol{p}\|_1 + \|\boldsymbol{q}\|_1$ and*

$$\alpha' \geq 2\alpha \tag{3.4.13}$$

$$\Delta' \leq \frac{\Delta}{\alpha' S}, \tag{3.4.14}$$

then any vector $\boldsymbol{c}$ that satisfies (3.4.3) *with coefficients α' and Δ' will also satisfy* (3.4.11) *with coefficients α and Δ.*

Proof. Let k be an arbitrary index and take $x = \boldsymbol{c}(k)$ and $y = (\boldsymbol{p} * \boldsymbol{q})(k)$. The condition (3.4.11) can be broken down into requiring one of two separate conditions:

$$\begin{cases} \operatorname{rel}(x, y) \leq \frac{1}{\alpha} & \text{if } y \geq \Delta \\ 0 \leq x \leq \left(1 + \frac{1}{\alpha}\right) y & \text{if } y < \Delta \end{cases}$$

If y lies in the first case, then $y \geq \Delta \geq \alpha' \Delta' S$, and so Corollary 3.4.8 implies that

$$\operatorname{rel}(x, y) \leq \frac{2}{\alpha'} \leq \frac{1}{\alpha},$$

as desired. The second case is implied directly by (3.4.3) and the non-negativity of $\boldsymbol{p}$ and $\boldsymbol{q}$. □

In other words, getting results of $\boldsymbol{p} * \boldsymbol{q}$ accurate down to the level Δ may require elements in $\boldsymbol{p}$ and $\boldsymbol{q}$ smaller than Δ by a factor of $2S\alpha$, and requires each element to be computed with half the relative error to make up for any lost mass.

There is not much that can be done to make psFFT-C satisfy CtT, other than the technique just explored, building on the TtC guarantee. This of course will make psFFT-C slower to execute, since both the lower bound and the desired relative error are more stringent.

Algorithm 3.7 The Bounded Checked FFT-C algorithm that convolves then trims. Given non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ and scalars $\alpha \geq 2$ and $\Delta \geq 0$, Bounded Checked FFT-C returns a vector $\boldsymbol{c} = \widetilde{\boldsymbol{p} * \boldsymbol{q}}$ as computed by FFT-C and a set I of indices such that (3.4.11) holds for $\boldsymbol{c}$ for all $k \notin I$.

procedure Bounded Checked FFT-C($\boldsymbol{p}$, $\boldsymbol{q}$, α, Δ)
  Compute $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$ via FFT-C, and note the $Q = 2^K$ of Corollary 2.2.10 that was used.
  Zero any entry $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(k)$ smaller than $\Delta - cK\varepsilon \left\|\boldsymbol{p}\right\|_2 \left\|\boldsymbol{q}\right\|_2$.
  Let I be the set of any other indices for which (3.1.2) does not hold.
  Set $\Delta' = \Delta/(2\alpha(\left\|\boldsymbol{p}\right\|_1 + \left\|\boldsymbol{q}\right\|_1))$.
  if $I \neq \emptyset$ and $Q \leq Q_{\max}$ and $\boldsymbol{p}(i) < \Delta'$ or $\boldsymbol{q}(i) < \Delta'$ for some i **then**
    Use FFT-C to compute the support of $\boldsymbol{p}_{\geq\Delta'} * \boldsymbol{q}_{\geq\Delta'}$.

$$I_{\text{sup}} = \text{supp}(\boldsymbol{p}_{\geq\Delta'} * \boldsymbol{q}_{\geq\Delta'}) = \text{supp}(\text{filt}(\widetilde{\mathbf{1}_{\boldsymbol{p}\geq\Delta'} * \mathbf{1}_{\boldsymbol{q}\geq\Delta'}})).$$

    Zero any entry $(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(i)$ with $i \notin I_{\text{sup}}$.
    Set $I \leftarrow I \cap I_{\text{sup}}$.
  end if
  return $\widetilde{\boldsymbol{p} * \boldsymbol{q}}$, I.
end procedure

On the other hand, the CtT guarantee can be achieved with Checked FFT-C in a slightly more aggressive way, as demonstrated by Bounded Checked FFT-C listed in Algorithm 3.7. This version can utilise the idea explored in the previous section of using FFT-C to compute $\boldsymbol{p} * \boldsymbol{q}$ but computing the support of $\boldsymbol{p}_{\geq\Delta'} * \boldsymbol{q}_{\geq\Delta'}$ for some lower bound Δ'. An appropriate value of Δ' can be chosen via Lemma 3.4.12.

The algorithm only needs to ensure (3.1.2) holds for indices i such that

$$(\widetilde{\boldsymbol{p} * \boldsymbol{q}})(i) \geq \Delta - cK\varepsilon \left\|\boldsymbol{p}\right\|_2 \left\|\boldsymbol{q}\right\|_2 . \tag{3.4.15}$$

Any other index is guaranteed to have $(\boldsymbol{p} * \boldsymbol{q})(i) < \Delta$ by Corollary 2.2.10, and hence can be set to zero, to ensure that it is not larger than the right hand side of (3.4.11). Unfortunately, this is only useful when Δ is sufficiently large, or else the right hand side of (3.4.15) will be less than the bound of (3.1.2), making the comparison vacuous.

3.5. Empirical Results

We now look into the behavior of an implementation of aFFT-C compared to existing algorithms, supporting our assertion that it enjoys the best of both NC and FFT-C, ensuring accuracy with higher performance than NC. The implementation examined here is in Python, using the high performance NumPy array library [vWCV11] as much as possible. We used `binary64` and log space so that the algorithms are applicable to cases where overflow or underflow may be encountered.

3.5.1. Accuracy

In the previous chapter we introduced a vector $\boldsymbol{p}$ in Example 2.1.4, which was used to demonstrate the accuracy of NC and inaccuracy of FFT-C. Let $\boldsymbol{r}_{\mathrm{aFFT-C},\alpha}$ represent the relative errors of each element of the convolution $\boldsymbol{p} * \boldsymbol{p}$ as computed via aFFT-C guaranteeing a relative accuracy of $1/\alpha$, then

$$\boldsymbol{r}_{\mathrm{aFFT-C},10^3} = (0, 0, 1 \cdot 10^{-7}, 4 \cdot 10^{-12}, 2 \cdot 10^{-15}, 4 \cdot 10^{-12}, 1 \cdot 10^{-7}).$$
$$\boldsymbol{r}_{\mathrm{aFFT-C},10^9} = (0, 0, 8 \cdot 10^{-15}, 4 \cdot 10^{-12}, 2 \cdot 10^{-15}, 4 \cdot 10^{-12}, 1 \cdot 10^{-14}).$$

Even in such a small example, the adaptive nature of the aFFT-C can be seen: when less accuracy is requested, the computed values may be less accurate.

Another example we saw previously was the demonstration of the catastrophic cancellation of FFT-C in Figure 2.1, where some elements of $\widetilde{\boldsymbol{p} * \boldsymbol{p}}$ as computed by FFT-C were hundreds of orders of magnitude larger than the true value. As one can see in Figure 3.2, aFFT-C handles this case perfectly, computing all elements with relative error less than the requested bound $1/\alpha = 10^{-3}$.

Figure 3.2 also hints at the shifting and splitting strategy used by aFFT-C: the almost-parabolic region in the center is the convolution of the split containing the largest values of the shifted vector with itself $\widetilde{\boldsymbol{p}_{\theta_0,1} * \boldsymbol{p}_{\theta_0,1}}$, and the smaller sloping regions on either side are artifacts of the various $\widetilde{\boldsymbol{p}_{\theta_0,i} * \boldsymbol{p}_{\theta_0,j}}$ terms.

For evaluating the accuracy of the algorithms in a more exhaustive way, we created many random instances of several classes of vectors of several different lengths. Since the applications we have in mind are in statistics, we normalised each vector to be a pmf.

In the examples below $n \geq 2$ denotes the length of the vectors $\boldsymbol{p}^{(k)}$. The values $a^{(k)}, b^{(k)}, c^{(k)} \sim U(0,1)$ are independent, and fixed for each $\boldsymbol{p}^{(k)}$ but vary between them, and $u_i^{(k)} \sim U(0,1)$ are independent, for each entry $\boldsymbol{p}^{(k)}(i)$. Finally, $A^{(k)}$ is the normalising constant such that $\left\|\boldsymbol{p}^{(k)}\right\|_1 = 1$.

a) constant, $\boldsymbol{p}^{(k)}(i) \equiv A^{(k)}$
b) random, $\boldsymbol{p}^{(k)}(i) = A^{(k)} \exp(-40 u_i^{(k)})$.

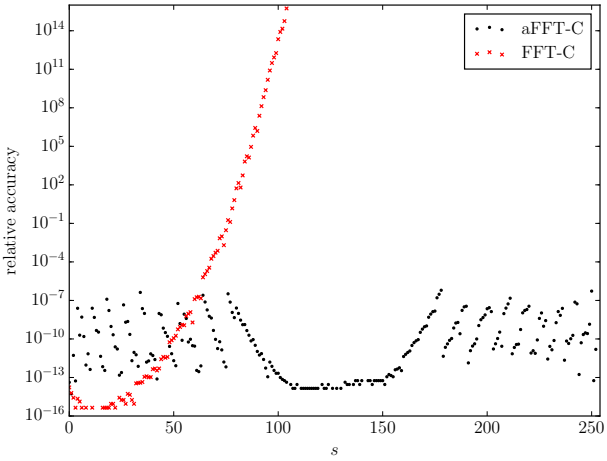

Figure 3.2 – The relative accuracy of aFFT-C vs. NC and FFT-C vs. NC for computing each entry of $\boldsymbol{q} := \boldsymbol{p} * \boldsymbol{p}$ with the pmf $\boldsymbol{p}$ from Figure 2.1 and $\alpha = 10^3$ for aFFT-C. The plot shows $\mathrm{rel}(\tilde{\boldsymbol{q}}_{\text{aFFT-C}}(s), \tilde{\boldsymbol{q}}_{\text{NC}}(s))$ and similarly for $\tilde{\boldsymbol{q}}_{\text{FFT-C}}$ where $\tilde{\boldsymbol{q}}_x(s) = \tilde{\boldsymbol{q}}(s)$ as computed by algorithm x, with $x = \text{NC}, \text{aFFT-C}, \text{FFT-C}$. The FFT-C plot is truncated because its relative error quickly increases.

	aFFT-C ($\alpha = 10^9$)		aFFT-C ($\alpha = 10^3$)		FFT-C	
n	Median	Maximum	Median	Maximum	Median	Maximum
8	$1 \cdot 10^{-13}$	$7 \cdot 10^{-11}$	$1 \cdot 10^{-8}$	$7 \cdot 10^{-5}$	$7 \cdot 10^{62}$	$4 \cdot 10^{303}$
16	$2 \cdot 10^{-12}$	$9 \cdot 10^{-11}$	$4 \cdot 10^{-7}$	$9 \cdot 10^{-5}$	$3 \cdot 10^{62}$	$2 \cdot 10^{303}$
32	$4 \cdot 10^{-12}$	$6 \cdot 10^{-11}$	$1 \cdot 10^{-6}$	$7 \cdot 10^{-5}$	$1 \cdot 10^{66}$	overflow
64	$5 \cdot 10^{-12}$	$7 \cdot 10^{-11}$	$2 \cdot 10^{-6}$	$6 \cdot 10^{-5}$	$7 \cdot 10^{65}$	overflow
128	$8 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$2 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	$2 \cdot 10^{69}$	overflow
256	$6 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$2 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	$1 \cdot 10^{68}$	overflow
512	$6 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$1 \cdot 10^{-6}$	$2 \cdot 10^{-5}$	$5 \cdot 10^{70}$	overflow
1024	$4 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$8 \cdot 10^{-7}$	$5 \cdot 10^{-5}$	$2 \cdot 10^{70}$	overflow
2048	$3 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$4 \cdot 10^{-7}$	$2 \cdot 10^{-5}$	$3 \cdot 10^{68}$	overflow
4096	$2 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$2 \cdot 10^{-7}$	$3 \cdot 10^{-5}$	$3 \cdot 10^{69}$	overflow
8192	$1 \cdot 10^{-12}$	$5 \cdot 10^{-11}$	$2 \cdot 10^{-7}$	$3 \cdot 10^{-5}$	$1 \cdot 10^{69}$	overflow

Table 3.2 – The maximum and median of the maximum relative error of the elements of $\widetilde{\boldsymbol{q}}^{(j,k)} = \boldsymbol{p}^{(j)} * \boldsymbol{p}^{(k)}$ as computed by aFFT-C and FFT-C where the $\boldsymbol{p}^{(j)}$ and $\boldsymbol{p}^{(k)}$ are selected exhaustively from 100 random instances of classes of the pmfs described in Section 3.5, and n is their length. That is, the third column is $\max_{j,k} \max_s \mathrm{rel}(\widetilde{\boldsymbol{q}}^{(j,k)}_{\text{aFFT-C}}(s), \widetilde{\boldsymbol{q}}^{(j,k)}_{\text{NC}}(s))$ where $\widetilde{\boldsymbol{q}}^{(j,k)}_x(s)$ is $\widetilde{\boldsymbol{q}}^{(j,k)}(s)$ as computed by algorithm x with $x = \text{NC}, \text{aFFT-C}$ ($\alpha = 10^9$). The second column is identical, just with the median in place of the outer max. The fourth and fifth use aFFT-C with $\alpha = 10^3$. The last two columns use FFT-C in place of aFFT-C and the entries of "overflow" indicate that the largest relative error is larger than $1.8 \cdot 10^{308}$, the largest `binary64` value. The same pmfs $\boldsymbol{p}^{(j)}$ and $\boldsymbol{p}^{(k)}$ were used for every column.

c) quadratic, $\boldsymbol{p}^{(k)}(i) = A^{(k)} \exp(-30(a^{(k)} + 1){x_i}^2 + 20(2b^{(k)} - 1)x_i + 20(2c^{(k)} - 1))$ where the x_i are evenly spaced in $[0, 1]$ with the first element $x_0 = 0$, and the last $x_{n-1} = 1$.

d) sinusoid, $\boldsymbol{p}^{(k)}(i) = A^{(k)} \exp(10(3a^{(k)} + 1) \sin(x_i + b^{(k)}/10) + 10(5c^{(k)} - 4)x_i)$ where x_i are evenly spaced in $[0, 3\pi]$ with $x_0 = 0$ and $x_{n-1} = 3\pi$ similar to above.

e) "multi-scaled I", a random fifth (rounded down) of the $\boldsymbol{p}^{(k)}(i)$ are of the form $A^{(k)} \exp(-30u_i^{(k)})$ and the remaining entries are of the form $A^{(k)} \exp(-100(u_i^{(k)} + 1))$.

f) "multi-scaled II", a random third (rounded down) of the $\boldsymbol{p}^{(k)}(i)$ are of the form $A^{(k)} \exp(-15(2a^{(k)} + 1)u_i^{(k)})$ and the remaining entries are of the form $A^{(k)} \exp[-50((2a^{(k)} + 1)u_i^{(k)} + 2b^{(k)} + 1)]$.

For each length $n \in \{2^3, 2^4, \ldots, 2^{13}\}$, we generated 100 instances of each class of vector (other than the constant vectors, where there is exactly one such vector), giving $k = 1, 2, \ldots, 501$. We then computed all pairwise convolutions $\boldsymbol{p}^{(j)} \tilde{*} \boldsymbol{p}^{(k)}$ with both aFFT-C and FFT-C, recording the relative error of each element of the convolution with respect to the convolution computed with NC. The results of this are summarised in Table 3.2, which shows that aFFT-C is accurate in all cases with error less than the requested $1/\alpha$ for both a very large and smaller values of α. In contrast, FFT-C can be wildly incorrect, as expected.

3.5.2. Run time

Accuracy is not the only aspect of aFFT-C that is interesting: it must also be fast, and indeed we devoted several paragraphs above to this topic, which we should justify with some measurements.

An emphasised point above was that aFFT-C is never significantly slower than FFT-C in cases when FFT-C is guaranteed to be accurate, a point demonstrated by Figure 3.3. This figure shows a summary of the time taken for computing convolutions of vectors with elements chosen independently as $U(0, 1)$, plotted against the length of those vectors. Such vectors are unlikely to be inaccurate when computed by FFT-C, and the plot demonstrates that aFFT-C accounts for this, with little overhead over direct FFT-C. Both aFFT-C and FFT-C are approximately $5 \cdot 10^4$ times faster than NC for vectors of length 2^{20}.

Examples for which Lemma 3.1.1 does not guarantee the accuracy of FFT-C do not see aFFT-C being quite so much faster than NC, but aFFT-C is still offers a large decrease in run time. Figure 3.4 shows the time taken for one example convolution to be performed with NC and with aFFT-C at several different accuracy levels α. In one case we used Algorithm 3.5, the variant of aFFT-C that satisfies the TtC guarantee. As can be seen, aFFT-C has some overhead over NC for small convolutions, but grows much less slowly, so that a convolution of size 2^{20} can be performed approximately 150

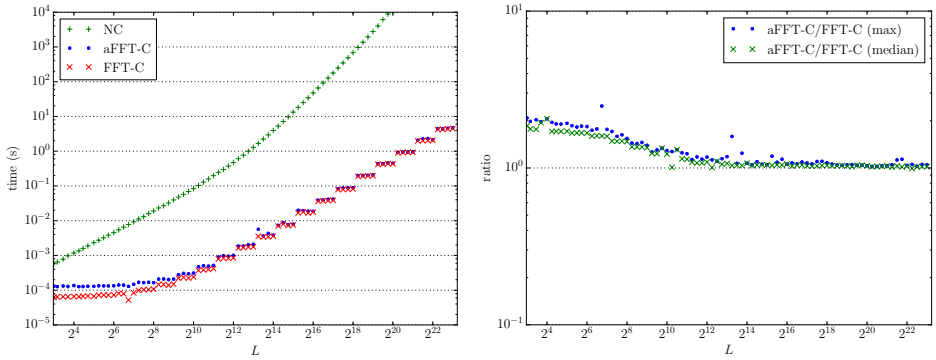

Figure 3.3 – Summary of the absolute and relative timings of a log-space implementation of aFFT-C vs. a log-space implementation of naive convolution vs. a direct FFT-C computations for many random examples of pmfs. The left plot shows the execution time of the convolution for each length $n = 2^{\lfloor k/4 \rfloor}$ which has the largest ratio between the run times of aFFT-C and FFT-C, to demonstrate the worst case for aFFT-C. Each convolution is of the form $\boldsymbol{p}_{2i}^{(n)} * \boldsymbol{p}_{2i+1}^{(n)}$ with $i = 1, \ldots, 100$, where each $\boldsymbol{p}_j^{(n)}$ is a vector of n iid $U(0,1)$ entries normalised so that $\left\|\boldsymbol{p}_j^{(n)}\right\| = 1$. The right plot shows that maximum ratio, and also the median ratio. None of the instances tested required splits, demonstrating that aFFT-C suffers from little overhead compared with direct FFT-C in cases where FFT-C is guaranteed to be accurate. The relative timing only shows aFFT-C vs. FFT-C because NC is orders of magnitude slower.

times faster with aFFT-C than with NC, and about 550 times faster when using a bound of $\Delta = 10^{-30}$.

Changing the accuracy level α influenced the time required, but was only highly significantly in the more extreme case $\alpha = 10^7$. The change from $\alpha = 10^1$ to $\alpha = 10^3$ resulted in a slow-down of approximately 1.7 times, while $\alpha = 10^7$ is more than 4 times slower than $\alpha = 10^3$ for longer vectors.

One could consider applying FFT-C to these convolutions, but they are wildly inaccurate. Performing the convolutions with FFT-C results in relative errors larger than 10^{100}. Furthermore, while we can identify when some values of the FFT-C computation are accurate, we can only do so for the largest ones, certainly ones that are larger than $\varepsilon = 2^{-53}$. These elements form a very small subset of the entire convolution in Figure 3.4.

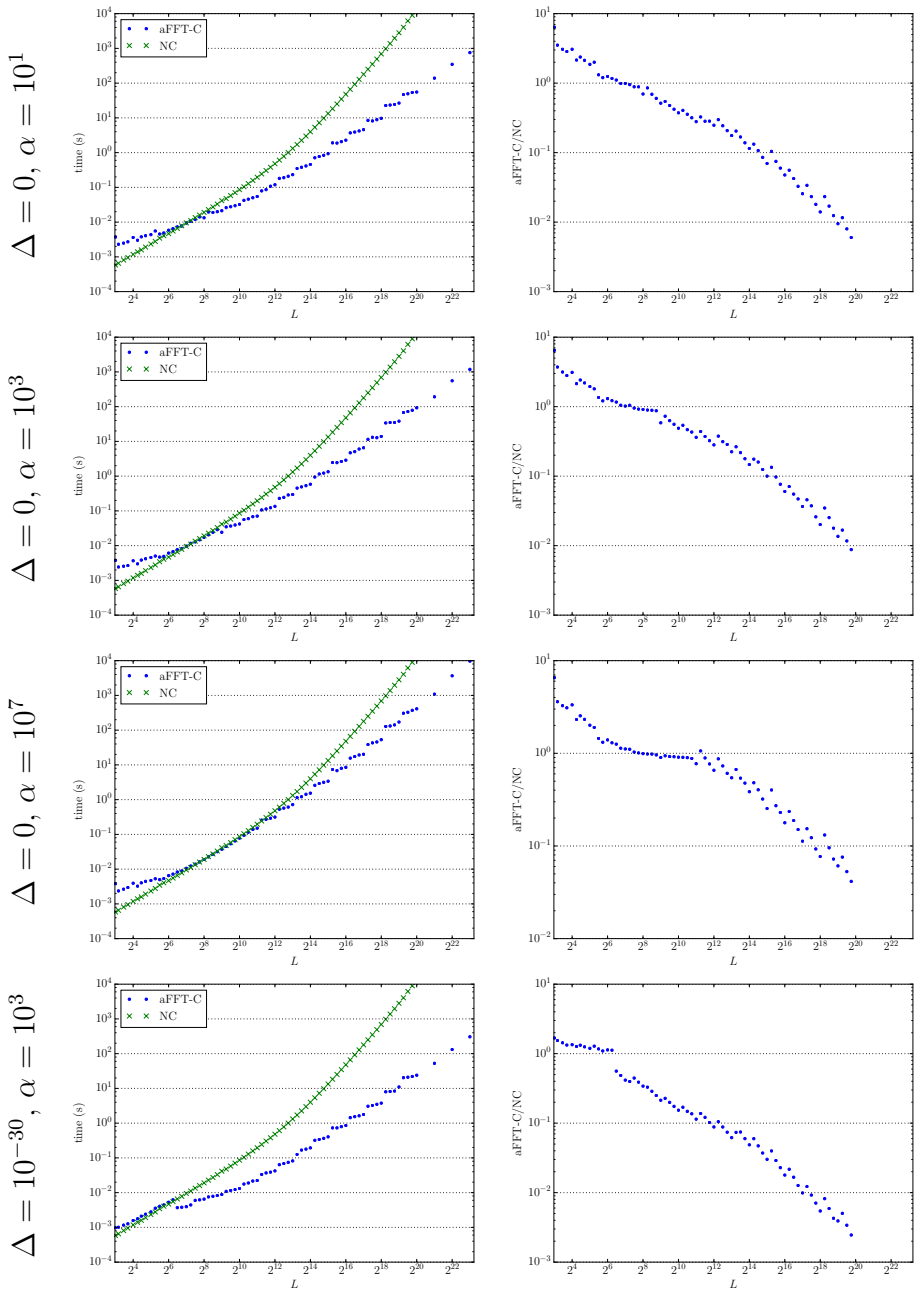

Figure 3.4 – The absolute and relative timings of a log-space implementation of aFFT-C vs. a log-space implementation of naive convolution for computing $\boldsymbol{p}^{(n)} * \boldsymbol{p}^{(n)}$, where $\boldsymbol{p}^{(n)}$ is given by $\boldsymbol{p}^{(n)}(k) = A\exp(60\sin s_k - 10 s_k)$ and $s_0 = 0$, $s_{n-1} = 3\pi$ with the remaining $n-2$ values of s_k evenly spaced between them. The upper three pairs of panels show the time taken for each algorithm (left) as well as the ratio of that time (right) for aFFT-C without a bound (represented as $\Delta = 0$), for several different values of α. The lower pair of panels give the same information for the same convolutions performed via Algorithm 3.6, satisfying (3.4.3) with $\Delta = 10^{-30}$ and $\alpha = 10^3$. We chose $n = \lfloor 2^{k/4} \rfloor$ for $k \in \{12, 13, \ldots, 79, 80, 84, 88, 92\}$. The steps after each power of two are caused by rounding up to $Q = 2^K$ for the FFT-C calculations.

Chapter 4

Accurate Iterated FFT Convolutions

A common use of convolution in statistical settings is computing the pmf of sums of independent random variables defined on lattices. A special case that often occurs in practice is sums of the form

$$X = \sum_{i=1}^{L} X_i$$

where X_i are iid. If $\boldsymbol{p}$ is the pmf of each X_i, then $\boldsymbol{p}^{*L}$ is the pmf of X.

4.1. FFT-C for iterated convolutions

The convolution $\boldsymbol{p}^{*L}$ can be performed via (2.3.2), restated here:

$$\boldsymbol{p}^{*L} = D^{-1}\left((D\boldsymbol{p})^{\odot L}\right), \tag{4.1.1}$$

where the transforms are performed with length at least as large as the length of the final result, with $\boldsymbol{p}$ padded with zeros to this length.

This suffers from the same potentially catastrophic errors as FFT-C, quantified in Lemma 2.3.8. The sFFT algorithm of Section 2.3 is designed to compensate for this case, but with limitations: it is focusing on computing a p-value of an observed value of X, rather than computing its entire pmf, and its accuracy is typically compromised if the pmf of X is not log-concave.

4.1.1. Error analysis

Iterated convolution via FFT-C is an integral part of sFFT, and thus analysis of the error in sFFT also included analysis of (4.1.1). The results of this are listed in Section 2.3.1 in this essay, and the result of particular interest is quoted as Lemma 2.3.8. However, the previous analysis built on a less precise version of Corollary 2.2.10. Hence, we have the following extension of Lemma 5 of [Kei05] which applies to arbitrary complex vectors, rather than just pmfs, and also quantifies the constant c.

Theorem 4.1.2. *Let $n, L \geq 2$ be natural numbers, $\boldsymbol{v} \in \mathbb{C}^n$ be a vector with $\widetilde{\boldsymbol{v}} = \boldsymbol{v}$ and $\widetilde{\boldsymbol{v}^{*L}}$ be an approximation to $\boldsymbol{v}^{*L}$ as computed via (4.1.1). Define $N = (n-1)L + 1$, the length of $\boldsymbol{v}^{*L}$, and choose $Q = 2^K \geq N$, then*

$$\left\|\widetilde{\boldsymbol{v}^{*L}} - \boldsymbol{v}^{*L}\right\|_\infty \leq LcK\varepsilon \left\|\boldsymbol{v}\right\|_1^L \tag{4.1.3}$$

where $c = 5$ if $L \geq 10$ and $K \geq 6$, and $c = 7$ otherwise.

Proof. This proof is similar to the proof of Theorem 2.2.8, and hence we use the same notation, specifically: let $\overline{\boldsymbol{x}}$ denote $D\boldsymbol{x}$, $\widetilde{\overline{\boldsymbol{x}}}$ denote $\widetilde{D\boldsymbol{x}}$ and μ denote the coefficient of ε in a bound on the relative error in complex multiplication, such as the $\sqrt{5}$ of (2.2.11).

The core difference between this proof and the proof of Theorem 2.2.8 is Lemma 2.2.21. The relevant equivalent results for this case are, for arbitrary $\boldsymbol{x} \in \mathbb{C}^Q$,

$$\left\| \widetilde{\widetilde{\boldsymbol{x}}^{\odot L}} - \overline{\boldsymbol{x}}^{\odot L} \right\|_1 \leq (L(\mu+2)K + (L-1)\mu + O(\varepsilon))\varepsilon \left\| \overline{\boldsymbol{x}}^{\odot L} \right\|_1 \tag{4.1.4}$$

$$\left\| \widetilde{\widetilde{\boldsymbol{x}}^{\odot L}} \right\|_1 \leq (1 + O(\varepsilon)) \left\| \overline{\boldsymbol{x}}^{\odot L} \right\|_1 \tag{4.1.5}$$

The second follows from (4.1.4) and the triangle inequality. Hence, we only need to focus on (4.1.4), which we do by breaking it into parts:

$$\begin{aligned} \left\| \widetilde{\widetilde{\boldsymbol{x}}^{\odot L}} - \overline{\boldsymbol{x}}^{\odot L} \right\|_1 &= \left\| \widetilde{\widetilde{\boldsymbol{x}}^{\odot L}} - \widetilde{\boldsymbol{x}}^{\odot L} + \widetilde{\boldsymbol{x}}^{\odot L} - \overline{\boldsymbol{x}}^{\odot L} \right\|_1 \\ &\leq \underbrace{\left\| \widetilde{\widetilde{\boldsymbol{x}}^{\odot L}} - \widetilde{\boldsymbol{x}}^{\odot L} \right\|_1}_{\gamma_1} + \underbrace{\left\| \widetilde{\boldsymbol{x}}^{\odot L} - \overline{\boldsymbol{x}}^{\odot L} \right\|_1}_{\gamma_2} \end{aligned}$$

Computing $\widetilde{\widetilde{\boldsymbol{x}}^{\odot L}}$ can possibly be done in a way that rounds to the closest floating point complex number, which bounds $\gamma_1 \leq \sqrt{2}\varepsilon \left\| \widetilde{\boldsymbol{x}}^{\odot L} \right\|_1$, or, if not the closest, some bounded distance away, replacing $\sqrt{2}$. However this can be inefficient, and so a more flexible bound that still works for our purposes can be deduced simply from the direct algorithm of repeated pointwise multiplications of $\widetilde{\boldsymbol{x}}$ with itself. For some element $\widetilde{x} = \widetilde{\boldsymbol{x}}(i)$, it is easy to see by induction that

$$\operatorname{rel}(\widetilde{\widetilde{x}^k}, \widetilde{x}^k) \leq ((k-1)\mu + O(\varepsilon))\varepsilon.$$

This gives a bound

$$\gamma_1 \leq ((L-1)\mu + O(\varepsilon))\varepsilon \left\| \widetilde{\boldsymbol{x}}^{\odot L} \right\|_1. \tag{4.1.6}$$

Both of the powers in γ_2 are exact, and hence just inflate the existing error (2.2.12), implying,

$$\gamma_2 \leq L((\mu+2)K + O(\varepsilon))\varepsilon \left\| \overline{\boldsymbol{x}}^{\odot L} \right\|_1.$$

This gives $\left\| \widetilde{\boldsymbol{x}}^{\odot L} \right\|_1 \leq (1 + O(\varepsilon)) \left\| \overline{\boldsymbol{x}}^{\odot L} \right\|_1$, and, hence, returning to (4.1.6) we have a version using $\overline{\boldsymbol{x}}$ instead of $\widetilde{\boldsymbol{x}}$,

$$\gamma_1 \leq ((L-1)\mu + O(\varepsilon))\varepsilon(1 + O(\varepsilon)) \left\| \overline{\boldsymbol{x}}^{\odot L} \right\|_1,$$

and thus we have proved (4.1.4).

The quantity $\left\|\overline{\boldsymbol{x}}^{\odot L}\right\|_1$ can be broken down with the combination of the convolution theorem, (2.2.18) and the Young inequality,

$$\begin{aligned}\left\|\overline{\boldsymbol{x}}^{\odot L}\right\|_1 &\leq Q\left\|\overline{\boldsymbol{x}}^{\odot L}\right\|_\infty \\ &= Q\left\|D(\boldsymbol{x}^{*L})\right\|_\infty \\ &\leq Q\left\|\boldsymbol{x}^{*L}\right\|_1 \\ &\leq Q\left\|\boldsymbol{x}\right\|_1^L. \end{aligned} \tag{4.1.7}$$

Returning to (4.1.3), the left hand side can also be separated into parts:

$$\begin{aligned}\left\|\widetilde{\boldsymbol{v}^{*L}} - \boldsymbol{v}^{*L}\right\|_\infty &= \left\|\widetilde{D^{-1}}\widetilde{\overline{\boldsymbol{v}}^{\odot L}} - D^{-1}\overline{\boldsymbol{v}}^{\odot L}\right\|_\infty \\ &\leq \underbrace{\left\|D^{-1}(\widetilde{\overline{\boldsymbol{v}}^{\odot L}} - \overline{\boldsymbol{v}}^{\odot L})\right\|_\infty}_{\alpha} + \underbrace{\left\|(\widetilde{D^{-1}} - D)\widetilde{\overline{\boldsymbol{v}}^{\odot L}}\right\|_\infty}_{\beta}\end{aligned}$$

First, α can be bounded, by (2.2.19), (4.1.4) and (4.1.7),

$$\begin{aligned}\alpha &\leq \frac{1}{Q}\left\|\widetilde{\overline{\boldsymbol{v}}^{\odot L}} - \overline{\boldsymbol{v}}^{\odot L}\right\|_1 \\ &\leq \frac{1}{Q}(L(\mu+2)K + (L-1)\mu + O(\varepsilon))\varepsilon\left\|\overline{\boldsymbol{v}}^{\odot L}\right\|_1 \\ &\leq (L(\mu+2)K + (L-1)\mu + O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_1^L\end{aligned}$$

To finish, β, bounded by (2.2.13), (4.1.7) and (4.1.5),

$$\begin{aligned}\beta &\leq \frac{(\mu+2)K + O(\varepsilon)}{Q}\varepsilon\left\|\widetilde{\overline{\boldsymbol{v}}^{\odot L}}\right\|_1 \\ &\leq \frac{(\mu+2)K + O(\varepsilon)}{Q}\varepsilon(1+O(\varepsilon))Q\left\|\boldsymbol{v}\right\|_1^L \\ &= ((\mu+2)K + O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_1^L.\end{aligned}$$

Bringing α and β together gives

$$\left\|\widetilde{\boldsymbol{v}^{*L}} - \boldsymbol{v}^{*L}\right\|_\infty \leq ((L+1)(\mu+2)K + (L-1)\mu + O(\varepsilon))\varepsilon\left\|\boldsymbol{v}\right\|_1^L.$$

The values of c stated in the theorem follow from this, since the assumption on n and L ensures $K \geq 2$. □

4.2. Convolution by squaring

An alternative to the direct FFT of (4.1.1) is to compute $\boldsymbol{p}^{*L}$ with $O(\log L)$ pairwise convolutions, via a squaring procedure based on the fact that $\boldsymbol{x}^{*K} * \boldsymbol{x}^{*L} = \boldsymbol{x}^{*(K+L)}$. This is inspired by the "exponentiation by squaring" algorithms often used for computing x^n, see, for instance, [Gor98].

Algorithm 4.1 lists one way to compute $\boldsymbol{v}^{*L}$ for arbitrary vectors $\boldsymbol{v}$ and $L \in \mathbb{N}$ in terms of pairwise convolutions, with any algorithm for computing

them. The goal is to break the computation down as

$$\boldsymbol{v}^{*L} = \boldsymbol{v}^{*2^{i_1}} * \boldsymbol{v}^{*2^{i_2}} * \ldots * \boldsymbol{v}^{*2^{i_\ell}} \tag{4.2.1}$$

where $0 \leq i_1 < \ldots < i_\ell \leq \lfloor \log_2 L \rfloor$ correspond to the $\ell = \sum_i b_i$ bits with value 1 in the binary expansion of L:

$$L = \sum_{i=0}^{\lfloor \log_2 L \rfloor} b_i 2^i = \sum_{i_j : b_{i_j} = 1} 2^{i_j} \tag{4.2.2}$$

where $b_i \in \{0, 1\}$. The $\boldsymbol{x}_i = \boldsymbol{v}^{*2^i}$ can be computed iteratively, via $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i * \boldsymbol{x}_i$.

Algorithm 4.1 Compute $\boldsymbol{v}^{*L}$ for an arbitrary vector $\boldsymbol{v}$ with $O(\log L)$ pairwise convolutions.

```
1: procedure SQUARING-C(v, L)
2:     Set w ← (1) and x_0 = v.
3:     Consider L in binary as in (4.2.2), L = Σ_i b_i 2^i.
4:     for i ← 0 : ⌊log_2 L⌋ do
5:         if b_i = 1 then
6:             Set w ← w * x_i.
7:         end if
8:         Compute x_{i+1} ← x_i * x_i.
9:     end for
10:    return w.
11: end procedure
```

This algorithm will of course give an exact answer when computed using infinite precision, but it needs a careful consideration when the pairwise convolutions $\boldsymbol{w} * \boldsymbol{x}_i$ on line 6 and $\boldsymbol{x}_i * \boldsymbol{x}_i$ on line 8 are computed with finite precision floating point. Those lines instead become approximations to the convolutions of approximate vectors:

$$\widetilde{\boldsymbol{w}} \leftarrow \widetilde{\widetilde{\boldsymbol{w}} * \widetilde{\boldsymbol{x}_i}},$$
$$\widetilde{\boldsymbol{x}_{i+1}} \leftarrow \widetilde{\widetilde{\boldsymbol{x}_i} * \widetilde{\boldsymbol{x}_i}}.$$

The error in the approximations $\widetilde{\boldsymbol{w}}$ and $\widetilde{\boldsymbol{x}_i}$ compounds with the error introduced by the finite-precision convolution $\widetilde{\cdot * \cdot}$, even with a precise algorithm such as NC or aFFT-C. Getting guarantees about the accuracy of the result requires quantifying and managing these sources of error.

4.2.1. Error Analysis (TtC)

We will analyse Squaring-C for computing $\boldsymbol{p}^{*L}$ where $\boldsymbol{p}$ is non-negative, and, without loss of generality, $\|\boldsymbol{p}\|_1 = 1$ (that is, $\boldsymbol{p}$ is a pmf). Choose $\alpha \geq 2$ and $\Delta \geq 0$, we assume that one is using a convolution algorithm $C_{\alpha,\Delta}$ that

trims then convolves, in the manner of Definition 3.4.2. That is, $C_{\alpha,\Delta}$ will compute an approximation to $\boldsymbol{v} * \boldsymbol{w}$ such that

$$\left(1 - \frac{1}{\alpha}\right)(\boldsymbol{v}_{\geq\Delta} * \boldsymbol{w}_{\geq\Delta})(k) \leq C_{\alpha,\Delta}(\boldsymbol{v}, \boldsymbol{w})(k) \leq \left(1 + \frac{1}{\alpha}\right)(\boldsymbol{v} * \boldsymbol{w})(k), \quad (4.2.3)$$

for all k.

By having the accuracy being conditional on the Δ term we can apply this analysis to situations such as the p-value computation in Chapter 5 where one can tolerate some lost mass in the computed vector. In that case, one can precompute some approximation to the exact p-value that allows working out just how much of the final convolution $\widetilde{\boldsymbol{p}^{*L}}$ is needed to get an accurate estimate of the tail sum. This allows the convolution to be computed much more efficiently as values that are known to be unimportant can be discarded.

The following theorem provides bounds on how error propagates through Squaring-C. This will allow us to deduce an α and Δ that achieve a desired overall accuracy in computing the convolution $\widetilde{\boldsymbol{p}^{*L}}$.

Theorem 4.2.4. *Suppose $\boldsymbol{p}$ is a pmf such that $\widetilde{\boldsymbol{p}} = \boldsymbol{p}$, $L \geq 2$ is an integer, $\alpha \geq 2$ and $\Delta \in [0, 1)$. Write L in binary, that is, $L = \sum_i b_i 2^i$ and let $\ell = \sum_i b_i$ and $0 < i_1 < \ldots < i_\ell = \lfloor \log_2 L \rfloor$ the indices i such that $b_i = 1$. Finally, denote $\beta = 1/\alpha$. Define*

$$L_j = \sum_{k=1}^{j} b_{i_k} 2^{i_k} \quad (4.2.5)$$

$$r_i = (1 + \beta)^{2^i - 1} - 1 \quad (4.2.6)$$

$$\bar{r}_j = (1 + \beta)^{L_j - 1} - 1 \quad (4.2.7)$$

$$\delta_i = \Delta \sum_{k=0}^{i-1} 2^{i-k}(1 + r_k) \quad (4.2.8)$$

$$\bar{\delta}_j = \sum_{k=1}^{j} \delta_{i_k} + \Delta \sum_{k=1}^{j-1} \left(2 + \bar{r}_k + r_{i_{k+1}}\right) \quad (4.2.9)$$

If $\widetilde{\boldsymbol{p}^{*L}}$ is computed via Squaring-C *with a TtC convolution algorithm $C_{\alpha,\Delta}$ as in* (4.2.3)*, then*

$$-\bar{r}_\ell \boldsymbol{p}^{*L}(k) - \bar{\delta}_\ell \leq \widetilde{\boldsymbol{p}^{*L}}(k) - \boldsymbol{p}^{*L}(k) \leq \bar{r}_\ell \boldsymbol{p}^{*L}(k) \quad (4.2.10)$$

for all k.

Note that $L_\ell = L$ and hence $\bar{r}_\ell = (1 + \beta)^L - 1$.

Proving this has three main steps:

(1) analyse $\widetilde{\widetilde{\boldsymbol{v}} * \widetilde{\boldsymbol{w}}}$ as an approximation to $\boldsymbol{v} * \boldsymbol{w}$,
(2) use this to quantify the error in squaring, that is, in $\widetilde{\boldsymbol{v}_{i+1}} = \widetilde{\widetilde{\boldsymbol{v}_i} * \widetilde{\boldsymbol{v}_i}}$ as an approximation to $\boldsymbol{v}_{i+1} = \boldsymbol{v}_i * \boldsymbol{v}_i$,
(3) join both parts to deduce the error in $\widetilde{\boldsymbol{p}^{*L}}$.

The separation of the last two steps is in fact visible in four of the sequences: r_i & δ_i come from step (2) and $\bar{r}_j$ & $\bar{\delta}_j$ come from step (3).

Step (1). The first step is to start with quantified approximations to two pmfs, and use these to derive a quantification of an approximation to their convolution.

Lemma 4.2.11. *Suppose $\boldsymbol{v}$ and $\boldsymbol{w}$ are pmfs and $\widetilde{\boldsymbol{v}}$ and $\widetilde{\boldsymbol{w}}$ are non-negative approximations such that, for each k,*

$$(1-e_1)\boldsymbol{v}(k)-\varepsilon_1 \le \widetilde{\boldsymbol{v}}(k) \le (1+e_1)\boldsymbol{v}(k)$$
$$(1-e_2)\boldsymbol{w}(k)-\varepsilon_2 \le \widetilde{\boldsymbol{w}}(k) \le (1+e_2)\boldsymbol{w}(k)$$

where $e_i, \varepsilon_i \in [0,1)$. Then, for each k,

$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}})(k) \le (1+\beta)(1+e_1)(1+e_2)(\boldsymbol{v}*\boldsymbol{w})(k) \tag{4.2.12}$$
$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}})(k) \ge (1-\beta)(1-e_1)(1-e_2)(\boldsymbol{v}*\boldsymbol{w})(k) - (\varepsilon_1+\varepsilon_2+\Delta(2+e_1+e_2)) \tag{4.2.13}$$

where $\beta = 1/\alpha$.

Proof. Since the vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are non-negative, we have

$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}})(k) \le (1+\beta)(\widetilde{\boldsymbol{v}}*\widetilde{\boldsymbol{w}})(k) \le (1+\beta)(1+e_1)(1+e_2)(\boldsymbol{v}*\boldsymbol{w})(k)$$

for each k, proving (4.2.12).

Define the vector $\boldsymbol{\varepsilon}_1'$ by

$$\boldsymbol{\varepsilon}_1'(k) = \min((1-e_1)\boldsymbol{v}(k), \varepsilon_1) \tag{4.2.14}$$

and similarly define $\boldsymbol{\varepsilon}_2'$ in terms of $\boldsymbol{w}$, e_2 and ε_2. Since $\widetilde{\boldsymbol{v}}$ is non-negative, we have $\widetilde{\boldsymbol{v}}(k) \ge (1-e_1)\boldsymbol{v}(k) - \boldsymbol{\varepsilon}_1'(k)$ and similarly for $\widetilde{\boldsymbol{w}}$. Therefore,

$$\begin{aligned}(\widetilde{\boldsymbol{v}}*\widetilde{\boldsymbol{w}})(k) &\ge (1-e_1)(1-e_2)(\boldsymbol{v}*\boldsymbol{w})(k) - (1-e_1)(\boldsymbol{v}*\boldsymbol{\varepsilon}_2')(k) - \\ &\qquad (1-e_2)(\boldsymbol{\varepsilon}_1'*\boldsymbol{w})(k) + (\boldsymbol{\varepsilon}_1'*\boldsymbol{\varepsilon}_2')(k) \\ &\ge (1-e_1)(1-e_2)(\boldsymbol{v}*\boldsymbol{w})(k) - (\boldsymbol{v}*\boldsymbol{\varepsilon}_2')(k) - (\boldsymbol{\varepsilon}_1'*\boldsymbol{w})(k)\end{aligned} \tag{4.2.15}$$

Looking at the subtracted terms, we can bound them from above:

$$(\boldsymbol{v}*\boldsymbol{\varepsilon}_2')(k) \le \|\boldsymbol{v}\|_1 \|\boldsymbol{\varepsilon}_2'\|_\infty \le 1\cdot\varepsilon_2$$
$$(\boldsymbol{\varepsilon}_1'*\boldsymbol{w})(k) \le \|\boldsymbol{\varepsilon}_1'\|_\infty \|\boldsymbol{w}\|_1 \le \varepsilon_1\cdot 1$$

By the assumption (4.2.3), any entry $C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}})(k)$ may be as small as the corresponding entry $(1-\beta)(\widetilde{\boldsymbol{v}}_{\ge\Delta}*\widetilde{\boldsymbol{w}}_{\ge\Delta})(k)$, a relationship which is better written as

$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}})(k) \ge (1-\beta)(\widetilde{\boldsymbol{v}}*\widetilde{\boldsymbol{w}} + \underbrace{\widetilde{\boldsymbol{v}}_{\ge\Delta}*\widetilde{\boldsymbol{w}}_{\ge\Delta} - \widetilde{\boldsymbol{v}}*\widetilde{\boldsymbol{w}}}_{\gamma})(k) \tag{4.2.16}$$

The term $\boldsymbol{\gamma}$ can be bounded via Lemma 3.4.5, which implies that

$$\begin{aligned}\boldsymbol{\gamma}(k) = (\widetilde{\boldsymbol{v}}_{\geq\Delta} * \widetilde{\boldsymbol{w}}_{\geq\Delta} - \widetilde{\boldsymbol{v}} * \widetilde{\boldsymbol{w}})(k) &\geq -\Delta(\|\widetilde{\boldsymbol{v}}\|_1 + \|\widetilde{\boldsymbol{w}}\|_1) \\ &\geq -\Delta\left[(1+e_1)\|\boldsymbol{v}\|_1 + (1+e_2)\|\boldsymbol{w}\|_1\right] \\ &= -\Delta(2+e_1+e_2).\end{aligned}$$

Bringing together this and (4.2.15) allows us to complete the reasoning of (4.2.16),

$$\begin{aligned}C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})(k) &\geq (1-\beta)\left[(\widetilde{\boldsymbol{v}} * \widetilde{\boldsymbol{w}})(k) - \Delta(2+e_1+e_2)\right] \\ &\geq (1-\beta)\left[(1-e_1)(1-e_2)(\boldsymbol{v}*\boldsymbol{w})(k) - \right. \\ &\qquad\qquad \left.(\varepsilon_1+\varepsilon_2+\Delta(2+e_1+e_2))\right] \\ &\geq (1-\beta)(1-e_1)(1-e_2)(\boldsymbol{v}*\boldsymbol{w})(k) - \\ &\qquad\qquad [\varepsilon_1+\varepsilon_2+\Delta(2+e_1+e_2)]\end{aligned}$$

which is exactly (4.2.13). □

These bounds are almost in the form that allows iterated use of the result; the only problem is that the lower and upper bound on $C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})(k)$ do not have matching coefficients. However, it is easy to see that

$$(1-\beta)(1-e_1)(1-e_2) - 1 \geq 1 - (1+\beta)(1+e_1)(1+e_2),$$

giving the following result.

Corollary 4.2.17. *Suppose $\boldsymbol{v}$ and $\boldsymbol{w}$ are pmfs with approximations $\widetilde{\boldsymbol{v}}$ and $\widetilde{\boldsymbol{w}}$ as in Lemma 4.2.11, then with*

$$e_3 = (1+\beta)(1+e_1)(1+e_2) - 1 \tag{4.2.18}$$

$$\varepsilon_3 = \varepsilon_1 + \varepsilon_2 + \Delta(2+e_1+e_2) \tag{4.2.19}$$

we have

$$\begin{aligned}(1-e_3)(\boldsymbol{v}*\boldsymbol{w})(k) - \varepsilon_3 \leq C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})(k) \\ \leq (1+e_3)(\boldsymbol{v}*\boldsymbol{w})(k).\end{aligned} \tag{4.2.20}$$

Specifically, e_3 and ε_3 are coefficients for $C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})$ as an approximation to $\boldsymbol{v} * \boldsymbol{w}$ in the form required to use as an input to Lemma 4.2.11.

Step (2). For the vector $\boldsymbol{p}$ define the sequence of convolutions

$$\widetilde{\boldsymbol{x}_0} = \boldsymbol{p} \qquad \widetilde{\boldsymbol{x}_i} = C_{\alpha,\Delta}(\widetilde{\boldsymbol{x}_{i-1}}, \widetilde{\boldsymbol{x}_{i-1}}) \tag{4.2.21}$$

The $\widetilde{\boldsymbol{x}_i}$ are approximations to $\boldsymbol{x}_i = \boldsymbol{p}^{*2^i}$, but how approximate are they?

Lemma 4.2.22. *Take r_i from (4.2.6) and δ_i from (4.2.8), then, for each k,*

$$-r_i\boldsymbol{x}_i(k) - \delta_i \leq \widetilde{\boldsymbol{x}_i}(k) - \boldsymbol{x}_i(k) \leq r_i\boldsymbol{x}_i(k) \tag{4.2.23}$$

Proof. We prove the result by induction on i using Corollary 4.2.17. The relation (4.2.23) is clearly true for $i = 0$.

If the result is true for some i, then we can take $\boldsymbol{v} = \boldsymbol{w} = \boldsymbol{x}_i$. and $\widetilde{\boldsymbol{v}} = \widetilde{\boldsymbol{w}} = \widetilde{\boldsymbol{x}}_i$ in Corollary 4.2.17, which, by definition, satisfy $\boldsymbol{v} * \boldsymbol{w} = \boldsymbol{x}_{i+1}$

and $C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}},\widetilde{\boldsymbol{w}}) = \widetilde{\boldsymbol{x}}_{i+1}$. This gives coefficients $e_1 = e_2 = r_i$ and $\varepsilon_1 = \varepsilon_2 = \delta_i$. It is easy to check that the e_3 and ε_3 of the corollary are exactly r_{i+1} and δ_{i+1} respectively, that is,

$$\begin{aligned} e_3 = r_{i+1} &= (1+\beta)(1+r_i)(1+r_i) - 1 \\ \varepsilon_3 = \delta_{i+1} &= \delta_i + \delta_i + \Delta(2 + r_i + r_i). \end{aligned} \tag{4.2.24}$$

The proof is completed by substituting into (4.2.20). □

Step (3). Recall that we represented L using its binary expansion $L = \sum_i b_i 2^i$ and denoted the ℓ indices i of the b_i such that $b_i = 1$ as $i_1 < i_2 < \ldots < i_\ell$ and that $L_k = \sum_{j=1}^{k} b_{i_j} 2^{i_j}$ is the truncation of the binary expansion after the kth 1.

Define a recurrence of vectors as follows:

$$\widetilde{\boldsymbol{v}_1} = \widetilde{\boldsymbol{x}_{i_1}} \qquad \widetilde{\boldsymbol{v}_j} = C_{\alpha,\Delta}(\widetilde{\boldsymbol{x}_{i_j}}, \widetilde{\boldsymbol{v}_{j-1}}). \tag{4.2.25}$$

Each $\widetilde{\boldsymbol{v}_j}$ is the approximation to the $\boldsymbol{v}_j = \boldsymbol{p}^{*L_j}$ which are computed iteratively by Squaring-C, and so the approximation $\widetilde{\boldsymbol{p}^{*L}}$ which is the focus of Theorem 4.2.4 is exactly $\widetilde{\boldsymbol{v}_\ell}$.

Lemma 4.2.26. *Take $\bar{r}_j$ from* (4.2.7) *and $\bar{\delta}_j$ from* (4.2.9)*, then, for each k,*

$$-\bar{r}_j \boldsymbol{v}_j(k) - \bar{\delta}_j \le \widetilde{\boldsymbol{v}_j}(k) - \boldsymbol{v}_j(k) \le \bar{r}_j \boldsymbol{v}_j(k) \tag{4.2.27}$$

Proof. This follows by induction using Corollary 4.2.17, in a similar manner to the proof of Lemma 4.2.22. We in fact fact use that lemma for the error in the $\widetilde{\boldsymbol{x}_{i_j}}$. Since $2^{i_1} = L_1$, it follows that $\bar{r}_1 = (1+\beta)^{2^{i_1}-1} = r_{i_1}$ and similarly $\bar{\delta}_1 = \delta_{i_1}$, thus Lemma 4.2.22 implies that (4.2.27) holds for $j = 1$.

The inductive step hinges on noticing that

$$\begin{aligned} \bar{r}_{j+1} &= (1+\beta)(1+\bar{r}_j)(1+r_{i_{j+1}}) - 1 \\ \bar{\delta}_{j+1} &= \bar{\delta}_j + \delta_{i_{j+1}} + \Delta(2 + \bar{r}_j + r_{i_{j+1}}) \end{aligned} \tag{4.2.28}$$

are the coefficients e_3 and ε_3 in Corollary 4.2.17. □

This last lemma is precisely what is needed to complete the proof.

Proof of Theorem 4.2.4. Set $j = \ell$ in Lemma 4.2.26, then $\widetilde{\boldsymbol{v}_\ell} = \widetilde{\boldsymbol{p}^{*L}}$ and so (4.2.10) is exactly (4.2.27). □

4.2.2. Error Analysis (CtT)

In Section 3.4, we also analysed convolutions $C_{\alpha,\Delta}$ that convolve then trim. Specifically, with the guarantee of Definition 3.4.10: for $\alpha \ge 2$ and $\Delta \ge 0$, $C_{\alpha,\Delta}$ satisfies

$$\left(1 - \frac{1}{\alpha}\right)(\boldsymbol{v} * \boldsymbol{w})_{\ge\Delta}(k) \le C_{\alpha,\Delta}(\boldsymbol{v},\boldsymbol{w})(k) \le \left(1 + \frac{1}{\alpha}\right)(\boldsymbol{v} * \boldsymbol{w})(k), \tag{4.2.29}$$

for all k.

This class of convolution algorithms gives a very similar result for iterated convolutions to Theorem 4.2.4, although with simpler bounds.

Theorem 4.2.30. *Suppose $\boldsymbol{p}$ is a pmf, $L \geq 2$ is an integer, $\alpha \geq 2$ and $\Delta \in [0, 1)$. Let $\bar{r}_\ell = (1+\beta)^{L-1} - 1$ and $\bar{\delta}_\ell = \Delta(L-1)$. If $\widetilde{\boldsymbol{p}^{*L}}$ is computed via* Squaring-C *with a CtT convolution algorithm $C_{\alpha,\Delta}$, then*

$$-\bar{r}_\ell \boldsymbol{p}^{*L}(k) - \bar{\delta}_\ell \leq \widetilde{\boldsymbol{p}^{*L}}(k) - \boldsymbol{p}^{*L}(k) \leq \bar{r}_\ell \boldsymbol{p}^{*L}(k) \tag{4.2.31}$$

for all k.

Notably, the quantity $\bar{r}_\ell$ is the same in both Theorem 4.2.30 and Theorem 4.2.4. The proof of this theorem is essentially identical to that of that previous theorem, and can be deduced by following through the consequences of a change to Lemma 4.2.11, which allows the multiplicative and subtractive error coefficients to be independent.

Lemma 4.2.32. *Suppose $\boldsymbol{v}$ and $\boldsymbol{w}$ are pmfs and $\widetilde{\boldsymbol{v}}$ and $\widetilde{\boldsymbol{w}}$ are non-negative approximations such that, for each k,*

$$(1-e_1)\boldsymbol{v}(k) - \varepsilon_1 \leq \widetilde{\boldsymbol{v}}(k) \leq (1+e_1)\boldsymbol{v}(k)$$
$$(1-e_2)\boldsymbol{w}(k) - \varepsilon_2 \leq \widetilde{\boldsymbol{w}}(k) \leq (1+e_2)\boldsymbol{w}(k)$$

where $e_i, \varepsilon_i \in [0, 1)$. Then, for each k,

$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})(k) \leq (1+\beta)(1+e_1)(1+e_2)(\boldsymbol{v} * \boldsymbol{w})(k) \tag{4.2.33}$$
$$C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}})(k) \geq (1-\beta)(1-e_1)(1-e_2)(\boldsymbol{v} * \boldsymbol{w})(k) - (\varepsilon_1 + \varepsilon_2 + \Delta) \tag{4.2.34}$$

where $\beta = 1/\alpha$.

In particular, the $\Delta(2 + e_1 + e_2)$ term of (4.2.13) can be reduced to Δ in (4.2.34). This makes the equivalent recurrences to (4.2.24) and (4.2.28) become, respectively,

$$\delta_{i+1} = \Delta + 2\delta_i,$$
$$\bar{\delta}_{j+1} = \Delta + \delta_{i_{j+1}} + \bar{\delta}_j,$$

and hence the sequences of coefficients that appear in the statement of Theorem 4.2.4 simplify under this alternate scheme to just

$$\delta_i = \Delta(2^i - 1),$$
$$\bar{\delta}_j = \Delta(L_j - 1).$$

4.2.3. Choosing the accuracy parameters

The behaviour and propagation of errors through Squaring-C is now understood, so Theorem 4.2.4 and similarly Theorem 4.2.30 allow deducing parameters required for computing an iterated convolution with a certain desired accuracy. This is straightforward for both analyses, since the equations defining the coefficients $\bar{r}_\ell$ and $\bar{\delta}_\ell$ are easily manipulated, as shown next.

Suppose $C_{\alpha,\Delta}$ is a convolution algorithm that satisfies either (4.2.3) or (4.2.29), and is used in Algorithm 4.1 to compute $\widetilde{\boldsymbol{p}^{*L}}$, an approximation to

the L-fold convolution $\boldsymbol{p}^{*L}$. Both Theorem 4.2.4 and Theorem 4.2.30 state that

$$-\bar{r}_\ell \boldsymbol{p}^{*L}(k) - \bar{\delta}_\ell \le \left(\widetilde{\boldsymbol{p}^{*L}} - \boldsymbol{p}^{*L}\right)(k) \le \bar{r}_\ell \boldsymbol{p}^{*L}(k) \tag{4.2.35}$$

for some coefficients $\bar{r}_\ell$ and $\bar{\delta}_\ell$, for all k.

The coefficient $\bar{r}_\ell = (1+\beta)^{L-1} - 1$ in both cases. The two versions of the coefficient $\bar{\delta}_\ell$ are not equal, but they has the same form: $\bar{\delta}_\ell = \Delta M$ for some M independent of Δ,

$$M = \begin{cases} \displaystyle\sum_{k=1}^{\ell} \sum_{j=0}^{i_k - 1} 2^{i_k - j}(1 + r_j) + \sum_{k=1}^{\ell-1} 2 + \bar{r}_k + r_{i_{k+1}} & \text{if } C_{\alpha,\Delta} \text{ satisfies (4.2.3)} \\ L - 1 & \text{if } C_{\alpha,\Delta} \text{ satisfies (4.2.29)} \end{cases} \tag{4.2.36}$$

where ℓ, i_k, r_i and $\bar{r}_j$ are from the statement of Theorem 4.2.4.

Corollary 4.2.37. *Given a pmf $\boldsymbol{p}$, an integer $L \ge 2$, a desired accuracy parameter $\alpha \ge 2$ and a desired lower bound $\Delta \in [0, 1)$, let*

$$\alpha' \ge \left(\left(1 + \frac{1}{\alpha}\right)^{1/(L-1)} - 1\right)^{-1} \tag{4.2.38}$$

$$\Delta' \le \frac{\Delta}{M}, \tag{4.2.39}$$

where M is defined in (4.2.36)*. Then, using a convolution algorithm $C_{\alpha',\Delta'}$ that satisfies either* (4.2.3) *or* (4.2.29) *in* Squaring-C*, we can compute an approximation $\widetilde{\boldsymbol{p}^{*L}}$ to $\boldsymbol{p}^{*L}$ such that*

$$-\frac{1}{\alpha}\boldsymbol{p}^{*L}(k) - \Delta \le \widetilde{\boldsymbol{p}^{*L}}(k) - \boldsymbol{p}^{*L}(k) \le \frac{1}{\alpha}\boldsymbol{p}^{*L}(k) \tag{4.2.40}$$

for each k.

In practice, one simply chooses α' and Δ' to be equal to the right hand side of their inequality.

4.2.4. Choosing the order of convolutions

A reasonable question to ask is if the order in which the $\widetilde{\boldsymbol{x}_j}$ are combined with $C_{\alpha,\Delta}$ influences the accuracy of the result. It might be that, say, computing $\widetilde{\boldsymbol{p}^{*L}}$ by starting with the highest power $\widetilde{\boldsymbol{x}_{i_\ell}}$ and working downward to end with $\widetilde{\boldsymbol{x}_{i_1}}$ gives a more accurate result than the reverse order of starting from the smallest power, which is the scheme described in (4.2.25) and used to prove Lemma 4.2.26.

It turns out that the order has no influence if the convolution algorithm $C_{\alpha,\Delta}$ satisfies CtT, but the error bound *is* affected by changing the order of operations with a TtC convolution algorithm. We will focus on the latter first.

Suppose $C_{\alpha,\Delta}$ is a convolution TtC algorithm with lower bound Δ, and $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ are vectors with approximations $\widetilde{\boldsymbol{u}}$, $\widetilde{\boldsymbol{v}}$ and $\widetilde{\boldsymbol{w}}$ that satisfy the

conditions of Lemma 4.2.11 with coefficients $e_{\boldsymbol{u}}$ and $\varepsilon_{\boldsymbol{u}}$, and similarly for $\boldsymbol{v}$ and $\boldsymbol{w}$. Without loss of generality, suppose $e_{\boldsymbol{u}} \leq e_{\boldsymbol{v}} \leq e_{\boldsymbol{w}}$. We are interested in both $\boldsymbol{c}_1 = C_{\alpha,\Delta}(C_{\alpha,\Delta}(\widetilde{\boldsymbol{u}}, \widetilde{\boldsymbol{v}}), \widetilde{\boldsymbol{w}})$ and $\boldsymbol{c}_2 = C_{\alpha,\Delta}(\widetilde{\boldsymbol{u}}, C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}}, \widetilde{\boldsymbol{w}}))$ as approximations to $\boldsymbol{u} * \boldsymbol{v} * \boldsymbol{w}$. Two applications of Corollary 4.2.17 allow us to derive error coefficients e_1 and ε_1 for a bound in the manner of that corollary on the error in $\boldsymbol{c}_1$,

$$\begin{aligned} e_1 &= (1+\beta)^2(1+e_{\boldsymbol{u}})(1+e_{\boldsymbol{v}})(1+e_{\boldsymbol{w}}) - 1 \\ \varepsilon_1 &= \Delta\left[3 + e_{\boldsymbol{u}} + e_{\boldsymbol{v}} + e_{\boldsymbol{w}} + (1+\beta)(1+e_{\boldsymbol{u}})(1+e_{\boldsymbol{v}})\right] + \varepsilon_{\boldsymbol{u}} + \varepsilon_{\boldsymbol{v}} + \varepsilon_{\boldsymbol{w}}, \end{aligned}$$

and similarly coefficients e_2 and ε_2 for $\boldsymbol{c}_2$,

$$\begin{aligned} e_2 &= (1+\beta)^2(1+e_{\boldsymbol{u}})(1+e_{\boldsymbol{v}})(1+e_{\boldsymbol{w}}) - 1 \\ \varepsilon_2 &= \Delta\left[3 + e_{\boldsymbol{u}} + e_{\boldsymbol{v}} + e_{\boldsymbol{w}} + (1+\beta)(1+e_{\boldsymbol{v}})(1+e_{\boldsymbol{w}})\right] + \varepsilon_{\boldsymbol{u}} + \varepsilon_{\boldsymbol{v}} + \varepsilon_{\boldsymbol{w}}, \end{aligned}$$

We have $e_1 = e_2$, but $\varepsilon_1 \neq \varepsilon_2$, and hence the order of convolutions does influence the error bound. We have

$$\varepsilon_2 - \varepsilon_1 = \Delta(1+\beta)(1+e_{\boldsymbol{v}})(e_{\boldsymbol{w}} - e_{\boldsymbol{u}}) \geq 0,$$

and thus $\boldsymbol{c}_1$ has a smaller bound on its error than $\boldsymbol{c}_2$. In other words, an approximation to a sequence of convolutions will have the smallest error bound if it is computed by starting from the most accurate vectors. The order used in Lemma 4.2.26 is close to this, as the error in the $\widetilde{\boldsymbol{x}_i}$ increases with i. That said, the difference is on the order of $\Delta(e_{\boldsymbol{w}} - e_{\boldsymbol{u}})$ and is hence typically negligible.

The assertion above about the case when $C_{\alpha,\Delta}$ has the CtT guarantee follows from noticing that $\varepsilon_1 = \varepsilon_2 = 2\Delta + \varepsilon_{\boldsymbol{u}} + \varepsilon_{\boldsymbol{v}} + \varepsilon_{\boldsymbol{w}}$.

4.2.5. Complexity

As usual, we analyse the complexity of Squaring-C. Let $C(n, m)$ represent the cost of performing a convolution of vectors of length n and m, with short-hand $C(n) = C(n, n)$. For FFT-C, the cost is $C(n, m) = O((n + m)\log(n + m))$, and we analyse this specific case.

Denote n the length of the pmf $\boldsymbol{p}$, and, as usual, let $L = \sum_i b_i 2^i$ have $\ell = \sum_i b_i$ set bits, with indices $0 \leq i_1 < \ldots < i_\ell = \lfloor \log_2 L \rfloor$, and denote $L_j = \sum_{k=1}^{j} b_{i_k} 2^{i_k}$ the truncation of the binary expansion of L at j. The iterated convolution $\boldsymbol{p}^{*M}$ has length $M(n-1)+1$.

There are two classes of convolutions performed by Squaring-C: the $\widetilde{\boldsymbol{x}_i}$ of (4.2.21), and the $\widetilde{\boldsymbol{v}_j}$ of (4.2.25).

Computing the $b = \lfloor \log_2 L \rfloor$ convolutions of the first class requires computing $C_{\alpha,\Delta}(\widetilde{\boldsymbol{x}_i}, \widetilde{\boldsymbol{x}_i})$ for $i = 0, 1, \ldots, b-1$. Each $\widetilde{\boldsymbol{x}_i}$ has length $2^i(n-1)+1$,

and so the total time required is

$$\begin{aligned}\sum_{i=0}^{b-1} C\left(2^i(n-1)+1\right) &= \sum_{i=0}^{b-1} O(2^{i+1}n\log(2^{i+1}n)) \\ &\le \sum_{i=0}^{b-1} O(2^{i+1}n\log(2^b n)) \\ &= O(2^b n\log(2^b n)).\end{aligned}$$

The second class requires $\ell-1$ convolutions. If these are performed in the order of (4.2.39), then each $\widetilde{\boldsymbol{v}_j}$ has length $L_j(n-1)+1$, and so computing $\widetilde{\boldsymbol{v}_{j+1}} = C_{\alpha,\Delta}(\widetilde{\boldsymbol{v}_j}, \widetilde{\boldsymbol{x}_{i_{j+1}}})$ for $j=1,\ldots,\ell-1$ has cost

$$C_j = C\left(L_j(n-1)+1, 2^{i_{j+1}}(n-1)+1\right) = O(L_{j+1}n\log(L_{j+1}n))$$

giving a total cost of

$$\sum_{j=1}^{\ell-1} C_j \le O(L_\ell n\log(L_\ell n)).$$

Thus, combining these, we see that the complexity of Squaring-C with an almost linear pairwise convolution algorithm such as FFT-C is

$$O(2^b n\log(2^b n)) + O(L_\ell n\log(L_\ell n)) = O(Ln\log(Ln)).$$

This is, in fact, the same complexity as using FFT-C directly in the manner of (4.1.1), but, as we know, both forms of FFT-C are inaccurate, especially so with the compounding nature of errors with Squaring-C.

For the $C(n,m) = O(nm)$ cost of the accurate NC, one can show in a similar manner that the complexity is $O(L^2n^2)$.

With aFFT-C, the complexity will lie somewhere between these two extremes, depending on the values of $\boldsymbol{p}$, and, more specifically, its dynamic range.

4.3. Squaring aFFT-C

The Squaring-C was more of an algorithm template than an actual algorithm, since the method of convolution and even the guarantees needed for it were left unspecified. One could choose any of the three pairwise convolution algorithms discussed so far: NC, FFT-C or aFFT-C, with the latter of the most interest.

4.3.1. Choosing the accuracy guarantee

As we saw in Algorithm 3.6, the aFFT-C algorithm can be easily modified to do less computation while still having the TtC guarantee assumed by Theorem 4.2.4, and as such it is an appropriate choice of $C_{\alpha,\Delta}$ for that theorem. Lemma 3.4.12 also demonstrated that this guarantee could be used to give the CtT guarantee assumed by Theorem 4.2.30 by choosing smaller parameters, and thus aFFT-C can also be used as the $C_{\alpha,\Delta}$ for this

second theorem. We proved that both TtC and CtT algorithms can be used to get similar guarantees of accuracy, so the choice between them is one of performance: aFFT-C runs faster with a larger lower bound, so which of the two gives the largest bound for each pairwise convolution?

The optimal choice is not necessarily immediately obvious: the use of Lemma 3.4.12 in Theorem 4.2.30 means each pairwise convolution must increase its accuracy and decrease its own lower bound, but the propagated errors are larger in Theorem 4.2.4. Intuitively, the latter should be better, since it is the natural setting of aFFT-C and so the analysis under that assumption captures the behaviour of the algorithm more closely, and, indeed, that is what we will see from inspecting the bounds required for each.

Choose a pmf $\boldsymbol{p}$, an integer $L \geq 2$, and some accuracy parameters $\alpha \geq 2$ and $\Delta \in [0, 1)$, and recall Corollary 4.2.37. Let $\beta' = 1/\alpha'$, and assume that it is small, so that we can ignore terms of order $O(\beta'^2)$. This assumption is reasonable for purposes of this analysis, especially because $\alpha' \approx (L-1)\alpha$, and thus, since $L \neq 2$ typically, it is usually larger than the original α.

If we are using aFFT-C to obtain the CtT guarantee, then we require each pairwise convolution to be computed with relative accuracy α' defined by (4.2.38), to a lower bound of $\Delta/(L-1)$. Combined with Lemma 3.4.12, we find that each invocation of aFFT-C needs the TtC guarantee with a bound of at most

$$\Delta_{\mathrm{CtT}} = \frac{\Delta}{2(L-1)\alpha''} = \frac{\Delta}{4(L-1)\alpha'}, \tag{4.3.1}$$

where α'' is derived from α' via (3.4.13).

On the other hand, if we use aFFT-C to get the TtC guarantee, then each pairwise convolution needs to use a bound of $\Delta_{\mathrm{TtC}} = \Delta/M$, with the M defined in (4.2.36). Using $(1+\beta')^n = 1 + n\beta' + O(\beta'^2) \approx 1 + n\beta'$ we can estimate an upper bound on the quantity M, to understand Δ_{TtC},

$$\begin{aligned}
M &= \sum_{k=1}^{\ell} \sum_{j=0}^{i_k-1} 2^{i_k-j}(1+r_j) + \sum_{k=1}^{\ell-1}(2+\bar{r}_k + r_{i_{k+1}}) \\
&= \sum_{k=1}^{\ell} \sum_{j=0}^{i_k-1} 2^{i_k-j}(1+\beta')^{2^j-1} + \sum_{k=1}^{\ell-1}\left((1+\beta')^{L_k-1} + (1+\beta')^{2^{i_{k+1}}-1}\right) \\
&\approx \sum_{k=1}^{\ell} \sum_{j=0}^{i_k-1} 2^{i_k-j}(1+(2^j-1)\beta') + \sum_{k=1}^{\ell-1}\left(2 + (L_k + 2^{i_{k+1}} - 2)\beta'\right) \\
&= \sum_{k=1}^{\ell} \sum_{j=0}^{i_k-1} \left(2^{i_k-j} + 2^{i_k-j}(2^j-1)\beta'\right) + \sum_{k=1}^{\ell-1}(2 + (L_{k+1}-2)\beta')
\end{aligned}$$

We have $\ell \leq \log_2 L$, and hence we can bound it by $2(\ell-1)$, as well as omitting the subtractive terms,

$$\leq \sum_{k=1}^{\ell} \sum_{j=0}^{i_k-1} \left(2^{i_k-j} + 2^{i_k}\beta'\right) + 2\log_2 L + \sum_{k=1}^{\ell-1} L_{k+1}\beta'$$

We can extend the second sum to include the $k = 0$ term, and relabel the summation variable to simplify that sum,

$$\begin{aligned}
&\approx 2\log_2 L + \sum_{k=1}^{\ell} 2^{i_k} \sum_{j=0}^{i_k-1} (2^{-j} + \beta') + \sum_{k=1}^{\ell} L_k\beta' \\
&\leq 2\log_2 L + \sum_{k=1}^{\ell} \left(2^{i_k}(2 + \log_2 L\beta') + L_k\beta'\right)
\end{aligned}$$

Finally, we have by definition $L = \sum_{k=1}^{\ell} 2^{i_k}$ and $L_k \leq L$, giving

$$\leq 2(L + \log_2 L) + 2L\log_2 L\beta'.$$

Thus, we can estimate $\Delta_{\mathrm{TtC}} \geq \frac{\Delta}{2L+\ldots}$. This is larger than Δ_{CtT} by a factor of approximately $2\alpha'$.

In conclusion, in addition to not requiring each convolution to be performed with higher accuracy, the use of Theorem 4.2.4 for pairwise convolution algorithms like aFFT-C that only guarantee (4.2.3) allows the use of a better lower bound for each pairwise convolution than the combination of Theorem 4.2.30 and Lemma 3.4.12.

4.3.2. Empirical Results

A typical use of iterated convolution where small values are vital is for computing p-values. As such, we chose to investigate this algorithm in the context of computing p-values in the next chapter, which gives us additional benefits, such as being able to compute lower bounds and perform exponential shifts that reduce execution time.

Chapter 5

Computing Accurate P-values

One major application of convolution and particularly iterated convolutions is computing p-values. Given $X = \sum_{i=1}^{L} X_i$ with X_i iid lattice-valued random variables, the p-value $P = P(X \geq s_0)$ of an observed value s_0 is often of interest. If $\boldsymbol{p}$ is the pmf of X_i, then

$$P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s). \tag{5.0.1}$$

The sFFT algorithm of [Kei05], described in detail in Section 2.3 allows efficient FFT-based computation of an approximation to P with guarantees about the relative accuracy, but it is generally only useful when $\boldsymbol{p}$ is log-concave. As we saw in Figure 2.2, if $\boldsymbol{p}$ is not log-concave, sFFT often fails to guarantee reasonable accuracy, leaving one to either use approximations without concrete guarantees such as saddlepoint methods [But07], or computing $\boldsymbol{p}^{*L}$ using the significantly slower method of NC.

5.1. The sisFFT algorithm

A better way to compute $\boldsymbol{p}^{*L}$ is to use Squaring-C with the newly designed aFFT-C as the pairwise convolution algorithm. This would ensure that the convolution could be calculated accurately and thus P can be calculated accurately via (5.0.1). Moreover, as we have argued here, aFFT-C is typically significantly faster than NC. However we can further improve the efficiency by noticing it is often not necessary to compute the full $\boldsymbol{p}^{*L}$ convolution. Indeed, as exploited by sFFT, getting an accurate estimate of P typically only requires the largest values $\boldsymbol{p}^{*L}(s)$ with $s \geq s_0$: neither indices $s < s_0$, nor negligible values relative to P are required to be accurately computed. This observation is the driving force behind our segmented iterated shifted FFT (sisFFT) algorithm.

The goals of sisFFT and sFFT are identical, and hence it is not surprising that they share two of the same core techniques: introducing an exponential shift to reduce the dynamic range of the pmf, and reasoning about a lower bound on the p-value P.

5.1.1. Shifting

As mentioned above and explored in earlier sections, performing an exponential shift on a pmf flattens it, particularly around the chosen s_0. This allows focusing effort on just the region of interest $s \geq s_0$, especially when combined with some lower bound.

Recall that sFFT relies on the exponential shift (2.3.3):

$$\boldsymbol{p}_\theta(s) = \boldsymbol{p}(s)e^{\theta s}/M_{\boldsymbol{p}}(\theta),$$

where $M_{\boldsymbol{p}}(\theta) = E[e^{\theta X}] = \sum_s \boldsymbol{p}(s)e^{s\theta}$ is the moment-generating function of X. Normalising by $M_{\boldsymbol{p}}(\theta)$ ensures that $\boldsymbol{p}_\theta$ is a pmf. This operation commutes with convolution, up to multiplication by a known constant, and hence is easy to invert after performing an L-fold convolution:

$$\boldsymbol{p}^{*L}(s) = \boldsymbol{p}_\theta^{*L}(s)e^{-s\theta}M_{\boldsymbol{p}}(\theta)^L. \tag{5.1.1}$$

In sFFT, θ is chosen as

$$\theta_0 = \arg\min_\theta \kappa(\theta) - \theta s_0/L, \tag{5.1.2}$$

where $\kappa(\theta) = \log M_{\boldsymbol{p}}(\theta)$ is the cumulant generating function of X.

As explained in Section 2.3, this is the canonical choice of θ in large deviation studies, and is motivated by the fact that $E(X_{\theta_0}) = s_0$ where X_{θ_0} is the random variable with pmf $\boldsymbol{p}_{\theta_0}^{*L}$. We also saw in that section that it is the θ that minimises an explicit bound on the relative error in the sFFT-computed p-value, which provided further motivation for this choice.

5.1.2. Lower bound

After shifting, the second way to reduce the work performed is to only accurately compute the largest values in the sum (5.0.1), since the smaller elements contribute little to P. This can be done by deducing a lower bound in the manner of Corollary 4.2.37: the nature of the bound $1/\alpha$ on the relative error and the lower bound Δ in that result guarantee that large values are accurately computed, and avoids the problems of FFT-C of significantly overestimating small values. To be useful, this lower bound must apply even in the presence of an exponential shift θ.

Suppose $\boldsymbol{p}$ has length n, and, for any integer ℓ, define $N_\ell = \ell(n-1)+1$, which is the length of $\boldsymbol{p}^{*\ell}$.

The main component of computing a lower bound is motivated by sFFT: the largest values of an iterated convolution can be computed efficiently and accurately via FFT-C. In general, however, just knowing the largest values will not give a tight lower bound on the p-value: we saw an example of this in Figure 2.2, where sFFT computed a p-value many times smaller than the true one. This failure comes from the combination of FFT-C and the exponential shift: FFT-C can only compute the largest values of the shifted result accurately, leaving gaps that may become significant when the shift is reversed. We can derive a more accurate lower bound by using FFT-C for most of a convolution, but filling in the gaps that it may leave.

The easiest way to do this is by noticing that the output of a convolution is most influenced by the largest values of the inputs. We can apply this observation by writing $\boldsymbol{p}^{*L} = \boldsymbol{p}^{*(L-1)} * \boldsymbol{p}$: a single pairwise convolution of the largest values of $\boldsymbol{p}^{*(L-1)}$ with the full $\boldsymbol{p}$ will generally give an estimate of $\boldsymbol{p}^{*L}$ without sizable gaps. FFT-C truncates its result in a way compatible with this observation, and thus we can approximate $\boldsymbol{p}^{*L}$ via

$$\widetilde{\boldsymbol{p}^{*L}} = \widetilde{\boldsymbol{p}^{*(L-1)}} * \boldsymbol{p} \tag{5.1.3}$$

where the $(L-1)$-fold convolution is computed and filtered in the manner of sFFT, and the pairwise convolution is computed with some accurate algorithm.

For our use case, we have three additional considerations: we need a lower bound, rather than an estimate; we only need a lower bound for the sum $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$, not the full vector; and, it is most beneficial for the $\boldsymbol{p}^{*(L-1)}$ convolution to be performed on a shifted $\boldsymbol{p}_\theta$ vector, to benefit from the manner in which a shift emphasises the relevant part of the pmf.

Not needing the full vector is particularly important for performance: if one wished to compute (5.1.3) as written, NC would require $O(Ln^2)$ time, while aFFT-C would offer asymptotic performance somewhere between that $O(Ln^2)$ and the almost linear $O(Ln \log Ln)$. As we will see below, only needing a tail sum allows the pairwise convolution between $\widetilde{\boldsymbol{p}_\theta^{*(L-1)}}$ and $\boldsymbol{p}_\theta$ to be computed via NC in $O(Ln)$ time and computing the $(L-1)$-fold convolution via FFT-C takes only $O(Ln \log Ln)$ time, making the latter the overall complexity of deducing a lower bound for P.

The following result is the formalisation of the above, driven in particular by its introduction of the vector $\overline{\boldsymbol{p}}$ of tail sums of $\boldsymbol{p}$. The result also resolves the other two factors above, ensuring that we are guaranteed to have a lower bound, and taking into account the possibility of an exponential shift by some θ.

Lemma 5.1.4. *Let $\widetilde{\boldsymbol{v}}'_\theta = \widetilde{\boldsymbol{p}_\theta^{*(L-1)}}$ as computed via (4.1.1) and take the $Q = 2^K \geq N_{L-1}$ used as the dimension of the DFTs performed. Let $E = (L-1) \cdot cK\varepsilon$ where c is defined in Theorem 4.1.2. Define*

$$\widetilde{\boldsymbol{v}}_\theta(k) = \begin{cases} \widetilde{\boldsymbol{v}}'_\theta(k) - E & \text{if } \widetilde{\boldsymbol{v}}'_\theta(k) > E \\ 0 & \text{otherwise} \end{cases} \tag{5.1.5}$$

and $\overline{\boldsymbol{p}}(k) = \sum_{j \geq k} \boldsymbol{p}(j)$. Then,

$$P_B = P_B(\theta) = \sum_{k \geq 0} \widetilde{\boldsymbol{v}}_\theta(k) e^{-k\theta + (L-1)\kappa(\theta)} \overline{\boldsymbol{p}}(s_0 - k) \leq P. \tag{5.1.6}$$

where, for any vector $\boldsymbol{x}$, we take $\boldsymbol{x}(i) = 0$ when $i < 0$ or $i \geq \text{length}(\boldsymbol{x})$, which ensures the sums defining both $\overline{\boldsymbol{p}}$ and P_B are finite.

Proof. By Theorem 4.1.2, we have

$$\left\| \widetilde{\boldsymbol{v}}'_\theta - \boldsymbol{p}_\theta^{*(L-1)} \right\|_\infty \leq E \left\| \boldsymbol{p}_\theta \right\|^{L-1} = E$$

and hence $\widetilde{\boldsymbol{v}}_\theta(k) \leq \boldsymbol{p}_\theta^{*(L-1)}(k)$ for all k.

The sum defining P_B has two core terms. The first is reverting the θ shift of $\widetilde{\boldsymbol{v}}_\theta$ as an approximation to $\boldsymbol{p}_\theta^{*(L-1)}$, as follows

$$\widetilde{\boldsymbol{v}}_\theta(k) e^{-k\theta + (L-1)\kappa(\theta)} \leq \boldsymbol{p}_\theta^{*(L-1)}(k) e^{-k\theta + (L-1)\kappa(\theta)} = \boldsymbol{p}^{*(L-1)}(k)$$

by (5.1.1). The second is of course the tail sum

$$\overline{\boldsymbol{p}}(s_0 - k) = \sum_{j \geq s_0 - k} \boldsymbol{p}(j).$$

Thus, (5.1.6) can be bounded by reordering the terms of the sums to reach the tail sum that defines P:

$$\begin{aligned} P_B &\leq \sum_{k \geq 0} \sum_{j \geq s_0 - k} \boldsymbol{p}^{*(L-1)}(k) \boldsymbol{p}(j) \\ &= \sum_{k \geq 0} \sum_{i \geq s_0} \boldsymbol{p}^{*(L-1)}(k) \boldsymbol{p}(i-k) \\ &= \sum_{i \geq s_0} \sum_{k \geq 0} \boldsymbol{p}^{*(L-1)}(k) \boldsymbol{p}(i-k) \\ &= \sum_{i \geq s_0} \boldsymbol{p}^{*L}(i) = P \end{aligned}$$ □

Notably, this last step demonstrates how only needing a tail sum allows us to use NC efficiently by not computing the entirety of (5.1.3). The key is that a tail sum of a convolution can be written as nested summation and reordering terms allows extracting common factors in the form of $\overline{\boldsymbol{p}}$.

This P_B is a lower bound for the final quantity we aim to compute, but it is not tuned for the exponential shift. In particular, we wish to compute $\boldsymbol{p}_\theta^{*L}$, not $\boldsymbol{p}^{*L}$ itself, and so we need to translate this lower bound into the shifted domain to ensure that it is in fact a lower bound and, more commonly, to avoid using a bound smaller than necessary.

Theorem 5.1.7. *Choose an accuracy level $0 < \beta \leq 1/2$, and fix $\theta \in \mathbb{R}$. Take $P_B = P_B(\theta)$ from Lemma 5.1.4 and define the lower bound,*

$$\begin{aligned} B_\theta &= P_B(\theta) \left(\sum_{s=s_0}^{N_L - 1} e^{-s\theta + L\kappa(\theta)} \right)^{-1} \\ &= P_B e^{\theta s_0 - L\kappa(\theta)} \frac{1 - e^{-\theta}}{1 - e^{-(N_L - s_0)\theta}}. \end{aligned} \tag{5.1.8}$$

Let $\widetilde{\boldsymbol{p}_\theta^{*L}}$ be an approximation of $\boldsymbol{p}_\theta^{*L}$ computed according to Corollary 4.2.37 with accuracy parameters $\alpha = 2/\beta$ and $\Delta = B_\theta \beta / 2$. Define the approximation

$$\widetilde{P} = \sum_{s \geq s_0} \widetilde{\boldsymbol{p}_\theta^{*L}}(s) e^{-\theta s + L\kappa(\theta)},$$

then

$$\operatorname{rel}(\widetilde{P}, P) \leq \beta.$$

Proof. The exact p-value P can be written in terms of the shifted pmf $\boldsymbol{p}_\theta$ via (5.1.1):

$$P = \sum_{s \geq s_0} \boldsymbol{p}_\theta^{*L}(s) e^{-s\theta + L\kappa(\theta)}$$

and hence the difference $\widetilde{P} - P$ can be expressed as

$$\widetilde{P} - P = \sum_{s \geq s_0} \left(\widetilde{\boldsymbol{p}_\theta^{*L}}(s) - \boldsymbol{p}_\theta^{*L}(s) \right) e^{-s\theta + L\kappa(\theta)}.$$

Therefore, by Corollary 4.2.37 and our choice of α and Δ for it,

$$\widetilde{P} - P \leq \sum_{s \geq s_0} \frac{\beta}{2} \boldsymbol{p}_\theta^{*L}(s) e^{-s\theta + L\kappa(\theta)} = \frac{\beta}{2} P.$$

At the same time, using Corollary 4.2.37 again, along with (5.1.8) and (5.1.6), we have

$$\begin{aligned}
\widetilde{P} - P &\geq \sum_{s \geq s_0} \left(-\frac{\beta}{2} \boldsymbol{p}_\theta^{*L}(s) - \frac{\beta}{2} B_\theta \right) e^{-s\theta + L\kappa(\theta)} \\
&= -\frac{\beta}{2} \left(P + P_B \left(\sum_{s=s_0}^{N_L - 1} e^{-s\theta + L\kappa(\theta)} \right)^{-1} \sum_{s=s_0}^{N_L - 1} e^{-s\theta + L\kappa(\theta)} \right) \\
&= -\frac{\beta}{2} \left(P + P_B \right) \geq -\beta P
\end{aligned}$$

Thus,

$$-\beta P \leq \widetilde{P} - P \leq \frac{\beta}{2} P$$

and so the result is proved. □

5.1.3. The algorithm

Pseudo-code describing sisFFT is listed in Algorithm 5.1. This implementation brings together all the elements of Sections 5.1.1 and 5.1.2. The results of the latter work with any θ, but, as discussed in the former, there is a clear choice θ_0.

5.1.4. Complexity

For computing $\sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$, where $\boldsymbol{p}$ has length n, this procedure has complexity $O(Ln \log Ln + C)$, where C is the complexity of Squaring-C.

As with aFFT-C, the parameter θ_0 used for performing the exponential shift can be computed by a numeric minimisation procedure, such as Brent's method. Each evaluation of the expression $\kappa(\theta) - \theta s_0 / L$ requires $O(n)$ time to compute $\kappa(\theta)$. A typical invocation will not require many such evaluations to find an appropriate θ_0 to a useful tolerance: in practice, we have not observed any case where computing θ_0 took a non-trivial amount of time.

The only non-trivial operation performed other than Squaring-C itself is computing P_B, which also takes $O(Ln \log Ln)$ time: the computation of $\boldsymbol{v}'$ takes $O(Ln \log Ln)$, followed by the $O(Ln)$ operations to compute $\boldsymbol{v}$, and $O(N_{L-1} - s_0) = O(Ln)$ to compute the sum defining P_B. The vector $\overline{\boldsymbol{p}}$ can be computed in $O(n)$ time by the relation $\overline{\boldsymbol{p}}(k) = \boldsymbol{p}(k) + \overline{\boldsymbol{p}}(k+1)$.

Algorithm 5.1 The segmented iterated shifted FFT (sisFFT) algorithm. Given a pmf $\boldsymbol{p}$, an index s_0, integer L and accuracy parameter $\beta \in (0, 1/2]$, sisFFT computes an approximation $\widetilde{P}$ to the tail sum $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ such that $\mathrm{rel}(\widetilde{P}, P) \leq \beta$.

1: **procedure** sisFFT($\boldsymbol{p}$, s_0, L, β)
2: Compute $\theta_0 = \arg\min_\theta \kappa(\theta) - \theta s_0/L$, where $\kappa(\theta)$ is the cumulant generating function of $\boldsymbol{p}$.
3: Shift $\boldsymbol{p}$ to $\boldsymbol{p}_{\theta_0}$ via (2.3.3).
4: Use (4.1.1) to compute $\boldsymbol{v}' = \widetilde{\boldsymbol{p}_{\theta_0}^{*(L-1)}}$, and hence $\boldsymbol{v}$ defined by (5.1.5).
5: Let $\overline{\boldsymbol{p}}(k) = \sum_{j \geq k} \boldsymbol{p}(j)$, and use this to compute $P_B(\theta_0)$ per (5.1.6) and hence a lower bound B_{θ_0} per (5.1.8).
6: Compute an approximation $\widetilde{\boldsymbol{p}_{\theta_0}^{*L}}$ via Corollary 4.2.37 with $\alpha = 2/\beta$ and $\Delta = B\beta/2$.
7: **return** $\widetilde{P} = \sum_{s \geq s_0} \widetilde{\boldsymbol{p}_\theta^{*L}}(s) e^{-\theta s + L\kappa(\theta)}$.
8: **end procedure**

Once the convolution $\widetilde{\boldsymbol{p}_\theta^{*L}}$ has been computed, it takes $O(N_L - s_0) = O(Ln)$ time to invert the shift and sum to retrieve $\widetilde{P}$.

For Squaring-C using aFFT-C, its complexity is often also approximately $C = O(Ln \log Ln)$, especially so with the lower bound that restricts the dynamic range of the vectors. This allows us to heuristically estimate that the overall complexity of sisFFT is often $O(Ln \log Ln)$.

5.2. Empirical Results

As with aFFT-C we now look at the empirical behaviour of our implementation of sisFFT. As before, there are two aspects of this to examine: accuracy and run time.

5.2.1. Accuracy

As with aFFT-C, we analysed the accuracy of sisFFT with a series of random tests. However, before looking at more exhaustive tests like that, we return to Figures 2.1 and 2.2.

Figure 2.1 and the upper panel of Figure 2.2 both computed the p-value P of $s_0 = 215$ in $\boldsymbol{p}^{*2}$, where $\boldsymbol{p}(s) = Ae^{\frac{1}{60}s(10-s)}$ for $s = \{0, 1, \ldots, 127\}$. We saw there the exact p-value is approximately $6 \cdot 10^{-154}$. Figure 2.1 mentions that FFT-C, which in this case is equivalent to the iterated version (4.1.1) with $L = 2$, calculated a vector that gave a p-value more than 10^{138} times larger than that, while Figure 2.2 showed that sFFT gives an accurate value. As one would expect, sisFFT is similar to sFFT in that it successfully computes an accurate value: with $\beta = 10^{-3}$, the approximation $\widetilde{P}$ satisfies $\mathrm{rel}(\widetilde{P}, P) < 10^{-13}$.

The lower panel of Figure 2.2 shows a more interesting example: computing the p-value P of $s_0 = 215$ in $\boldsymbol{p}^{*2}$ again, but for the pmf $\boldsymbol{p}(s) = A\exp\left(\frac{1}{60}s(s-256)\right)$ for $s = 0, \ldots, 127$, with A such that $\|\boldsymbol{p}\|_1 = 1$. This pmf $\boldsymbol{p}$ is not log concave, and sFFT fails: it computes $\widetilde{P} \approx 2 \cdot 10^{-234}$ while the true value is $P \approx 1 \cdot 10^{-225}$. On the other hand, sisFFT handles this case flawlessly, with $\beta = 10^{-3}$ as above, the approximation $\widetilde{P}$ satisfies $\mathrm{rel}(\widetilde{P}, P) < 10^{-11}$. Figure 5.1 shows how sisFFT computes the approximation for this example and the previous one, similar to Figure 2.2.

Table 5.1 lists the results of the exhaustive tests of sisFFT. For a variety of lengths n, we generated 100 instances of the same classes of vectors as listed in Section 3.5.1, and computed approximations to $\sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ for a several different convolution counts L and indices s_0. The former was chosen up to $L = 256$, via

$$L \in \left\{ 2^k \,\middle|\, 3 \leq k \leq 8 \right\} \cup \left\{ \lfloor 1.5^k \rfloor \,\middle|\, 2 \leq k \leq 13 \right\} \tag{5.2.1}$$

where, for $x \in \mathbb{R}$, $\lfloor x \rfloor$ is the largest integer not greater than x. The value of s_0 was chosen as a fraction of the length $N_L = L(n-1)+1$ of $\boldsymbol{p}^{*L}$, as

$$s_0 \in \left\{ \lfloor r N_L \rfloor \,|\, r \in \{0.6, 0.7, 0.8, 0.9, 0.95, 0.99\} \right\}. \tag{5.2.2}$$

As can be seen from the summary statistics in Table 5.1, the p-values computed via sisFFT always have a relative error less than the requested bound of β, validating our analysis. Furthermore, the adaptive nature of the algorithm is demonstrated by comparing the results with the extreme value of $\beta = 10^{-9}$ with the far more moderate value of $\beta = 10^{-3}$.

5.2.2. Run time

We also looked at the run time of sisFFT. The lower bound and shift combine to make it run efficiently, even as it still gives guaranteed accuracy. This is in contrast to the other method with guarantees of accuracy: computing the entire $\boldsymbol{p}^{*L}$ via Squaring-C with NC, and computing the relevant tail sum.

Figure 5.2 returns to the pmf used in Figure 3.4, and demonstrates the run time of sisFFT for computing an estimate to $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$, with several different levels of accuracy and values of L. As can be seen, in all cases, sisFFT is never slower than NC, even for the smallest case of $L = 2$, and the margin progressively decreases until sisFFT is more than 20,000 faster for $\beta = 10^{-1}$ and $L = 2^{15}$, and still more than 2000 faster times at this L for both $\beta = 10^{-3}$ and $\beta = 10^{-7}$.

Additionally, as decreasing relative plot especially indicates, the sisFFT algorithm appears to be asymptotically faster than NC as L increases.

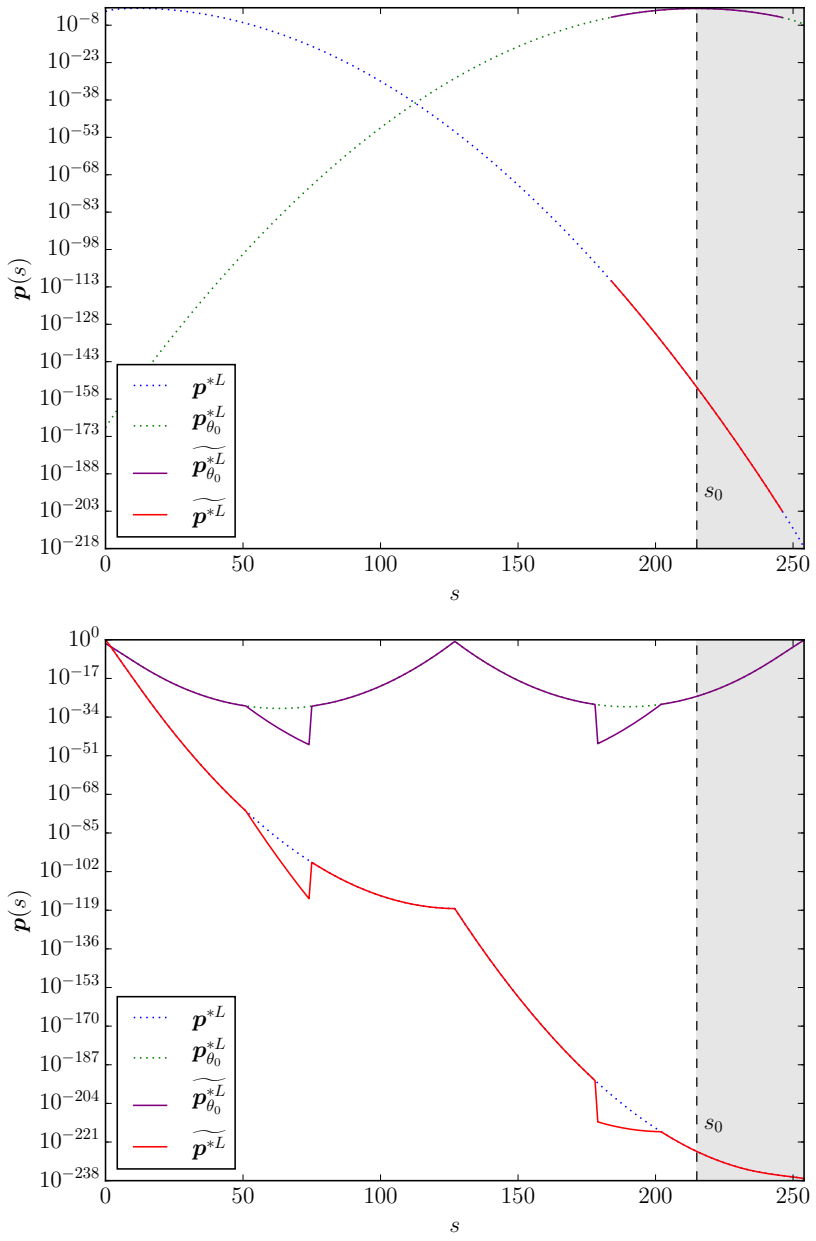

Figure 5.1 – A breakdown of the sisFFT algorithm when used to compute an approximation $\widetilde{P}$ to $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ for the two problems in Figure 2.2. As in that figure, the dotted lines indicate exact values and the solid lines are the values computed by sisFFT. One can see in the first plot that sisFFT has a large lower bound, only needed to compute a very small region around s_0. The jagged line of the second shows that the lower bound was large enough that the smallest values of $\boldsymbol{p}_{\theta_0}$ could be ignored when partitioning $\boldsymbol{p}_{\theta_0}$ inside aFFT-C. As designed, the computed values $\widetilde{\boldsymbol{p}^{*L}}(s)$ match the largest of the exact values $\boldsymbol{p}^{*L}(s)$ closely in the shaded region $s \geq s_0$, and thus, as discussed in Section 5.2.1, the p-values $\widetilde{P}$ are computed accurately.

	$\beta = 10^{-9}$		$\beta = 10^{-3}$	
n	Median	Maximum	Median	Maximum
8	$1 \cdot 10^{-13}$	$3 \cdot 10^{-10}$	$6 \cdot 10^{-12}$	$3 \cdot 10^{-4}$
16	$1 \cdot 10^{-13}$	$3 \cdot 10^{-10}$	$8 \cdot 10^{-9}$	$3 \cdot 10^{-4}$
32	$2 \cdot 10^{-13}$	$3 \cdot 10^{-10}$	$2 \cdot 10^{-7}$	$2 \cdot 10^{-4}$
64	$2 \cdot 10^{-13}$	$5 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$2 \cdot 10^{-4}$
128	$3 \cdot 10^{-13}$	$5 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$1 \cdot 10^{-4}$
256	$4 \cdot 10^{-13}$	$7 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$3 \cdot 10^{-4}$
512	$4 \cdot 10^{-13}$	$5 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$1 \cdot 10^{-4}$
1024	$4 \cdot 10^{-13}$	$5 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$2 \cdot 10^{-4}$
2048	$3 \cdot 10^{-13}$	$6 \cdot 10^{-10}$	$1 \cdot 10^{-6}$	$6 \cdot 10^{-5}$
4096	$3 \cdot 10^{-13}$	$3 \cdot 10^{-11}$	$8 \cdot 10^{-7}$	$4 \cdot 10^{-5}$
8192	$3 \cdot 10^{-13}$	$2 \cdot 10^{-11}$	$5 \cdot 10^{-7}$	$3 \cdot 10^{-5}$

Table 5.1 – The accuracy of sisFFT. As with the similar Table 3.2 for aFFT-C, we performed many tests of pmfs described in Section 5.2.1 of different lengths. For each pmf $\boldsymbol{p}$ with length n, we computed the tail sum $\sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ with L per (5.2.1) and s_0 per (5.2.2). This tail sum was computed via sisFFT with $\beta = 10^{-9}$ and $\beta = 10^{-3}$, and by computing $\boldsymbol{p}^{*L}$ with Squaring-C using NC, with no trimming, for pairwise convolutions and summing the relevant values. For each length n, the table lists summary statistics of the relative error of each of the sisFFT-computed values as compared to the corresponding NC-computed value.

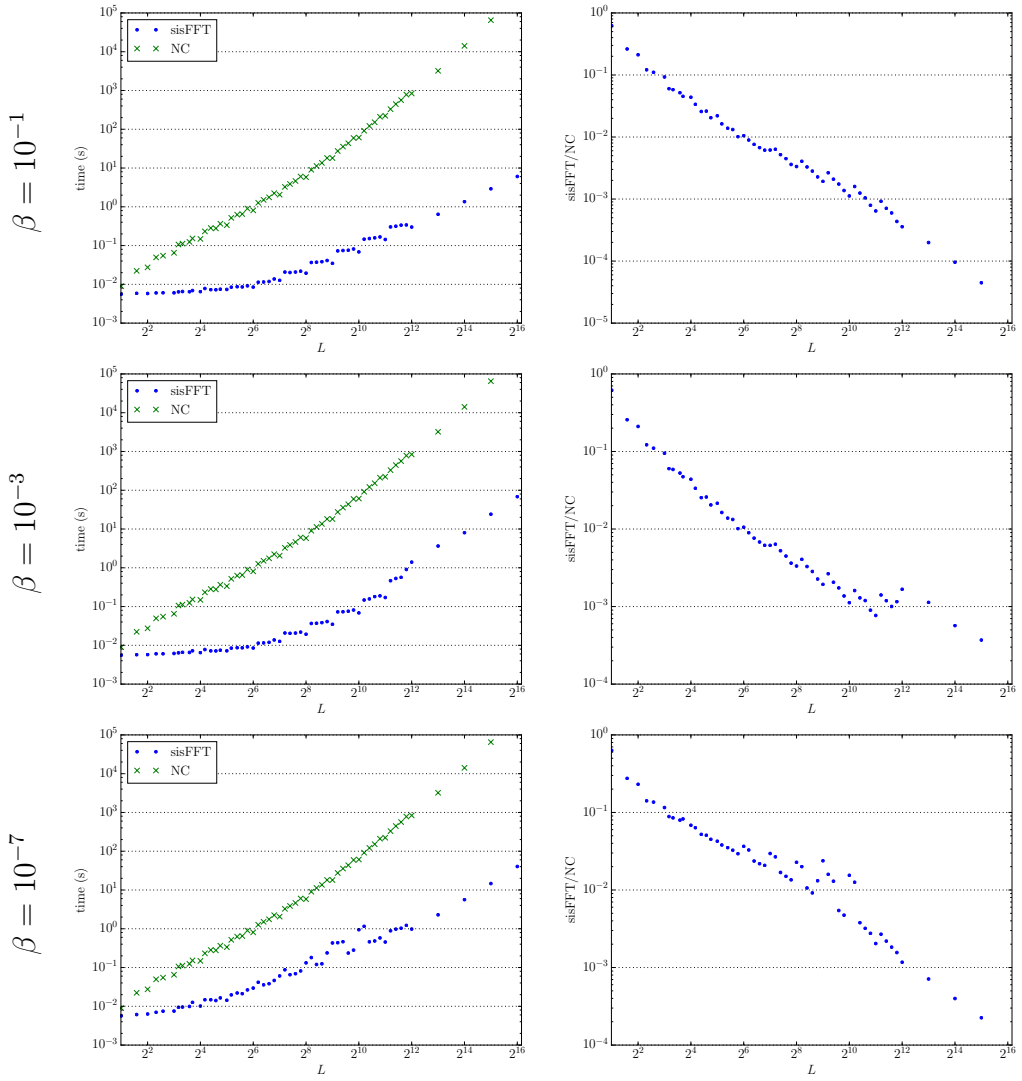

Figure 5.2 – A plot of the time required to compute an estimate to $P = \sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ for $\boldsymbol{p}$ from Figure 3.4 with length $n = 128$, by sisFFT and by computing $\boldsymbol{p}^{*L}$ via Squaring-C using NC for the pairwise convolutions. Both were implemented in log-space. We chose $L = \lfloor 2^{k/5} \rfloor$ for $k \in \{5, 6, \ldots, 39, 40, 45, \ldots, 65\}$, and $s_0 = \lfloor 0.95 N_L \rfloor$ where $N_L = L(n-1)+1$ is the length of $\boldsymbol{p}^{*L}$. Each plot shows both the time required to compute an estimate of P using NC as described above and the time required for sisFFT. The sisFFT computation is shown for three different accuracy thresholds β.

Conclusion

This essay introduces two novel algorithms that offer significant improvements over current methods for computing convolutions. The first algorithm, aFFT-C, is designed to accurately compute $\boldsymbol{p} * \boldsymbol{q}$ with non-negative vectors $\boldsymbol{p}$ and $\boldsymbol{q}$. The computation guarantees the accuracy of all computed values to the user-specified level of relative precision. The second algorithm, sisFFT, is similarly designed to guarantee the accuracy of tail sums $\sum_{s \geq s_0} \boldsymbol{p}^{*L}(s)$ with $\boldsymbol{p}$ a pmf vector.

Based on rigorous mathematical analysis accompanied by careful algorithmic considerations, both algorithms maintain the desired accuracy level by accounting for the accumulated effect of floating point round-off errors. They are designed to ensure that a user need not worry about inaccuracy, much like when using NC, while retaining most of the speed of FFT-based convolutions in most circumstances.

This essay develops the mathematical theory on which these new algorithms are based. It goes further to analyse the performance of both algorithms theoretically, and contrasts this analysis with empirical analysis of their accuracy and runtime performance. Importantly, we implemented both algorithms in both Python and R, and made them available at `https://github.com/huonw/sisfft-py` and `https://github.com/huonw/sisfft` respectively.

As mentioned, the aFFT-C algorithm introduced as Algorithm 3.4 in Chapter 3 allows the user to select the appropriate precision for their task, trading performance for accuracy. Based on a novel analysis of the accumulation of round-off error in FFT-C, this algorithm is also designed to detect many cases when FFT-C is sufficiently accurate. In such cases, aFFT-C has minimal overhead over FFT-C, giving it almost-linear performance; in other cases, it is designed to never be significantly slower than a direct application of NC, and is usually much faster.

We extend aFFT-C to allow incorporating a lower bound in two ways. Using a lower bound allows the algorithm to be faster, by ignoring values that are considered uninteresting while still retaining its accuracy guarantees for all sufficiently large values of the result. The first method, trim then convolve, is later shown to be well-suited for the iterated convolutions required by sisFFT, with the modified algorithm listed as Algorithm 3.6. The second, convolve then trim, is designed to be more intuitive for single pairwise convolutions. As before, the accuracy as well as the lower bound can be selected by the user of the algorithm.

The sisFFT algorithm is designed for computing a p-value of a sum of lattice-valued iid random variables. It builds on aFFT-C and its ability to use a lower bound to do this efficiently. As part of this, we describe and, importantly, analyse errors in so-called "convolution by squaring", for

computing iterated convolutions $\boldsymbol{p}^{*L}$, as well as refining our earlier analysis of the round-off errors in FFT-C to apply to iterated convolutions more directly. The analysis of the former is careful to handle propagation of error, and incorporates consideration of a lower bound. Both iterated aFFT-C and iterated FFT-C are used as part of sisFFT, with FFT-C used to estimate a lower bound that allows aFFT-C to automatically discard irrelevant values.

Importantly, the techniques developed here to create aFFT-C and especially sisFFT can potentially be used for accelerating exact solutions to other problems, such as evaluating the significance of the Pearson χ^2 multinomial goodness-of-fit test.

References

[BC89] O. Barndorff-Nielsen and D. Cox, *Asymptotic Techniques for Use in Statistics*, ser. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1989. DOI: 10.1002/bimj.4710330330.

[BE94] T. Bailey and C. Elkan, "Fitting a mixture model by expectation maximization to discover motifs in biopolymers", in *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, Menlo Park, California, 1994, pp. 28–36.

[BFT04] G. Bejerano, N. Friedman and N. Tishby, "Efficient exact p-value computation for small sample, sparse and surprising categorical data", *J. Comput. Biol.*, vol. 11, pp. 867–886, 2004. DOI: 10.1089/cmb.2004.11.867.

[BPZ07] R. Brent, C. Percival and P. Zimmermann, "Error bounds on complex floating-point multiplication", *Mathematics of Computation*, vol. 76, no. 259, pp. 1469–1481, 2007. DOI: 10.1090/S0025-5718-07-01931-X.

[Bre73] R. Brent, *Algorithms for Minimization Without Derivatives*, ser. Prentice-Hall series in automatic computation. Englewood Cliffs, NJ: Prentice-Hall, 1973, ISBN: 0130223352.

[But07] R. Butler, *Saddlepoint Approximations with Applications*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2007. DOI: 10.1017/CBO9780511619083.

[BZ10] R. Brent and P. Zimmermann, *Modern Computer Arithmetic*, ser. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511921698.

[CT65] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. DOI: 10.1090/S0025-5718-1965-0178586-1.

[Din92] C. G. Ding, "Algorithm AS 275: Computing the non-central χ^2 distribution function", *Applied Statistics*, pp. 478–482, 1992. DOI: 10.2307/2347584.

[FJ05] M. Frigo and S. Johnson, "The design and implementation of FFTW3", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. DOI: 10.1109/JPROC.2004.840301.

[GO77] D. Gottlieb and S. A. Orszag, *Numerical Analysis of Spectral Methods: Theory and Applications*. Philadelphia, PA: Society

for Industrial and Applied Mathematics, 1977.
DOI: 10.1137/1.9781611970425.

[Gol91] D. Goldberg, "What every computer scientist should know about floating point arithmetic", *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.
DOI: 10.1145/103162.103163.

[Gor98] D. M. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.
DOI: 10.1006/jagm.1997.0913.

[Hir05] K. F. Hirji, *Exact Analysis of Discrete Data*. Boca Raton, FL: Chapman and Hall/CRC, 2005.
DOI: 10.1201/9781420036190.

[HJB85] M. T. Heideman, D. H. Johnson and C. S. Burrus, "Gauss and the history of the fast Fourier transform", *Archive for History of Exact Sciences*, vol. 34, no. 3, pp. 265–277, 1985.
DOI: 10.1007/BF00348431.

[IEE08] IEEE Computer Society, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York, NY: IEEE, Aug. 2008.
DOI: 10.1109/IEEESTD.2008.4610935.

[JOP+01] E. Jones, T. Oliphant, P. Peterson *et al.* (2001–). SciPy: Open source scientific tools for Python, [Online]. Available: `http://www.scipy.org/` (visited on 12/01/2016).

[Kah04] W. Kahan. (2004). A Logarithm Too Clever by Half, [Online]. Available: `https://www.cs.berkeley.edu/~wkahan/LOG10HAF.TXT` (visited on 11/01/2016).

[Kei05] U. Keich, "sFFT: A faster accurate computation of the p-value of the entropy score", *Journal of Computational Biology*, vol. 12, no. 4, pp. 416–430, 2005.
DOI: 10.1089/cmb.2005.12.416.

[KN06] U. Keich and N. Nagarajan, "A fast and numerically robust method for exact multinomial goodness-of-fit test.", *Journal of Computational and Graphical Statistics*, Lecture Notes in Computer Science, vol. 15, no. 4, I. Jonassen and J. Kim, Eds., 2006.
DOI: 10.1198/106186006X159377.

[MBdD+10] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé and S. Torres, *Handbook of Floating-Point Arithmetic*. Boston, MA: Birkhäuser, 2010, p. 572.
DOI: 10.1007/978-0-8176-4705-6.

[MP92] C. R. Mehta and N. R. Patel, *StatXact: User manual: Statistical software for exact nonparametric inference*, CYTEL Software Corporation, 1992.

[NJK05] N. Nagarajan, N. Jones and U. Keich, "Computing the P-value of the information content from an alignment of multiple sequences", *Bioinformatics*, vol. 21 Suppl 1, no. ISMB 2005, pp. i311–i318, Jun. 2005.
DOI: 10.1093/bioinformatics/bti1044.

[R C15] R Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2015.

[Ric95] J. A. Rice, *Mathematical statistics and data analysis*, 2nd. Belmont, CA: Duxbury Press, 1995, ISBN: 9780534209346.

[Ros95] G. Rossum, "Python reference manual", Amsterdam, The Netherlands, Tech. Rep., 1995.

[Sch96] J. Schatzman, "Accuracy of the discrete Fourier transform and the fast Fourier transform", *SIAM J. Sci. Comput.*, vol. 17, no. 5, pp. 1150–1166, 1996.
DOI: 10.1137/S1064827593247023.

[Sto66] T. Stockham Jr., "High-speed convolution and correlation", in *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*, ser. AFIPS '66 (Spring), Boston, Massachusetts: ACM, 1966, pp. 229–233.
DOI: 10.1145/1464182.1464209.

[TZ01] M. Tasche and H. Zeuner, "Worst and average case roundoff error analysis for FFT", *BIT*, vol. 41, no. 3, pp. 563–581, 2001.
DOI: 10.1023/A:1021923430250.

[vWCV11] S. van der Walt, S. Colbert and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation", *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011.
DOI: 10.1109/MCSE.2011.37.

[WK] H. Wilson and U. Keich, "Accurate small tail probabilities of sums of iid lattice-valued random variables via FFT", Submitted.

[WK16] H. Wilson and U. Keich, "Accurate pairwise convolutions of non-negative vectors via FFT", *Computational Statistics & Data Analysis*, vol. 101, pp. 300–315, 2016.
DOI: 10.1016/j.csda.2016.03.010.