Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Callum Gilchrist (`cg3g22@soton.ac.uk`)
15th August 2025

Formal Verification of Fast Fourier Transforms

Supervisor: Artjoms Šinkarovs (`a.sinkarovs@soton.ac.uk`)
Second Examiner: Vahid Yazdanpanah (`v.yazdanpanah@soton.ac.uk`)

A Project Report submitted for the award of
BSc Computer Science

Abstract

Discrete Fourier Transforms (DFTs) are key operations within Digital Signal Processing and other fields, Fast Fourier Transforms (FFTs) allow for the time complexity of computing the DFT to be significantly reduced. Implementations of the FFT often comprise large, low-level libraries written with efficiency in mind, making verification of their correctness challenging. Agda is a dependently typed functional language which implements Martin-Löf type theory allowing proofs to be embedded within code. As a result of including these proofs, programs written in Agda can contain formal guarantees of their correctness, for the FFT this requires proving that the DFT is equal to the FFT for all cases.
In this project, I have thus far created an Agda definition of the DFT and FFT. I now plan to implement proof that the two definitions are equal for all possible inputs. Given such a proof I will then use the FFT definition to generate a low-level library of my own which can perform Fourier Transforms with strict guarantees on its correctness.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Contents

1 Introduction

The Discrete Fourier Transform (DFT) is a staple operation within Computer Science, Physics, and other fields with many applications. Fast Fourier Transforms are implementations of the DFT with improved performance characteristics. Most current implementations, such as WFFT[1], take the form of large libraries written in low-level languages. A key component of these libraries is the use of multiple implementations of the same algorithm, with each implementation (or kernel) containing optimisations suited towards specific input sizes and hardware profiles. When the user wants to compute the result of a Fourier Transform, the library chooses the optimal kernel based on the input size and the user's hardware.

The large number of kernels makes it very challenging to verify that a given FFT library provides the same result as the naïve DFT. This is because to do so would involve analysing the low-level implementation of each kernel, individually, and proving that it gives the same result as the naïve DFT for all possible inputs. An alternate approach is as follows. Instead of analysing existing code to confirm its correctness, we can create a single specification of the FFT such that it can be instantiated to any kernel, giving us a usable kernel and formal proof that said kernel computes the expected values.

Agda is a dependently typed functional language which allows for formal properties of programs written in it to be proven.[2] This paper discusses the use of Agda to create a general case implementation of the FFT which can be proven to always compute the same value as the naïve DFT. This general case definition can then be used to generate the kernels for any size in a low-level, efficient language. This allows the proof of correctness to be propagated down to any kernels generated from it allowing for a library of formally correct kernels to be generated with associated guarantees of its correctness.

2 Background

2.1 Fourier Transforms

Fourier Transforms are mathematical operations which transform functions from their time domain to their frequency domain. Fourier Transforms, and derivatives of, receive their name from the French mathematician and physicist Jean-Baptiste-Joseph Fourier who proposed in his 1822[3] treatise that any given function can be represented as a harmonic series.[4] While bearing Fourier's name, some early forms of the Discrete Fourier Transform (DFT), a Fourier Transform which works on evenly spaced samples of a function, can be found before Fourier's time. As discussed by Heideman and Johnson in "Gauss and the History of the Fast Fourier Transform"[5], the earliest known example of this can be found in work published by Alexis-Claude Clairaut in 1754[6]. Clairaut defined a variation of the DFT which exclusively used what we now refer to as the cosine component, thus restricting the input domain to the set of even functions[1].[5] Carl Friedrich Gauss extended Clairaut's definition to make use of both cosine and sine components, removing the need for the input domain to be restricted to the set of even functions and allowing for the analysis of any periodic function.[9][5] This definition was published posthumously in 1866, however, it is believed that it was originally written in 1805.[5]

We can use the historical definitions discussed above to create our modern definition for the DFT as follows. Given an input sequence $x = (x_0, x_1, \ldots, x_{n-1})$, where $x_i \in \mathbb{C}$, our transformed sequence $X = (X_0, X_1, \ldots, X_{n-1})$, where $X_i \in \mathbb{C}$, is given as follows.

$$X_j = \sum_{k=0}^{N-1} \omega_N^{jk} x_k \tag{1}$$

$$\text{where } \omega_N = e^{-\frac{2\pi i}{N}} = \cos\left(\frac{2\pi}{N}\right) - i \sin\left(\frac{2\pi}{N}\right) \tag{2}$$

The DFT Eq. 1 has applications in a variety of fields, such as digital signal processing[10], however, when implemented naïvely, it has poor performance scaling, requiring "$\mathcal{O}(n^2)$ complex operations" [11]. Methods to reduce the number of complex operations required when computing the DFT were first investigated by Gauss in his 1805 treatise such that the "tediousness of mechanical calculations"[9] could be reduced.[5] In part due to his lack of research into the complexity scaling factor of his method, Gauss's research into how computation complexity could be reduced was not widely recognised until 1977 when H. H. Goldstine highlighted Gauss's research in an article for the Journal of Applied Mathematics and Mechanics.[5][12] While the DFT continued to be of great use to mathematicians through the 20th century, and with Gauss's work on complexity remaining hidden, some attempts (such as those by Danielson and Lanczos [13] and by Good [14]) were made to create Fast Fourier Transform algorithms (FFT algorithms) which could reduce the complexity of computation to $\mathcal{O}(n \log n)$. These algorithms, however, where only applicable to a subset of the

[1]The term "even function" refers to the set of functions $f(x)$ such that $f(-x) = f(x)$, that is to say, the set of functions which are symmetric over the y-axis. [7][8]

domain[14], succeeded only in reducing the constant on $\mathcal{O}(n^2)$, or did not directly perform the computational complexity[13].

In 1965 James William Cooley and John Tukey succeeded in discovering an FFT algorithm through the inadvertent reinvention of Gauss's algorithm for fast computation of the DFT; This would henceforth be known as the Cooley-Tukey FFT Algorithm.[15][5] This FFT Algorithm allows for a given DFT to be computed with $\mathcal{O}(n \log n)$ complex operations through recursive splitting of the input.[15] Although other FFT Algorithms were discovered before and after the Cooley-Tukey FFT, it is commonly considered to be "the most important FFT"[1].

The Cooley-Tukey FFT can be derived from the DFT Eq. 1 by splitting any non-prime input n into the composite $n = r_1 r_2$. and expressing the indices k and j as follows.

$$\begin{aligned} & j = j_1 r_1 + j_0 & & k = k_1 r_2 + k_0 \\ \text{where } & j_0 = (0, 1, \ldots, r_1 - 1) & \text{where } & k_0 = (0, 1, \ldots, r_2 - 1) \\ & j_1 = (0, 1, \ldots, r_2 - 1) & & k_1 = (0, 1, \ldots, r_1 - 1) \end{aligned} \tag{3}$$

Eq. 1 can then be arranged to take the following form.

$$X_{j_1 r_1 + j_0} = \sum_{k_0=0}^{r_2-1} \left[\left(\sum_{k_1=0}^{r_1-1} x_{k_1 r_2 + k_0} \omega_{r_1}^{k_1 j_0} \right) \omega_{r_1 r_2}^{k_0 j_1} \right] \omega_{r_2}^{k_0 j_1} \tag{4}$$

When written in this form our recursive step, and thus the core idea of the Cooley-Tukey FFT, be easily observed by noting that the inner sum takes the form of a DFT of length r_1.

2.2 Agda

Agda[2] is a functional programming language which implements Martin-Löf Type Theory.[2][16] Martin-Löf type theory provides the definition of, and Agda allows for the construction of, dependent types, these allow Agda to act as a proof assistant meaning programs constructed with it can contain proofs asserting their correctness. These proofs allow systems to be built which are provably correct allowing for a high confidence in their reliability.[2]

2.3 Related work

FFTW[1] is a `C` code library which is generally accepted within academia and industry as the fastest method with which the FFT can be correctly computed.[17] It achieves this title by implementing its own "special-purpose compiler"[17], `genfft`, this compiler accepts the size of the transform as input and outputs a kernel - a `C` code implementations of some known algorithm (i.e. the Cooley-Tukey FFT [15] Eq .4) optimised for that sized transform and the current hardware.[17] Although it is known through rigorous testing and real-world use that FFTW is correct, there is no formal verification of its correctness.

[2]Reference to "Agda" throughout this report will always refer to version 2 unless explicitly stated otherwise

As FFTW does not come with formal guarantees separate definitions of various FFTs have been created before in proof assistance such as Coq[18] and Hol[19] with various methods and goals. In the paper "Certifying the Fast Fourier Transform with Coq"[20], Capretta makes use of binary trees to create a definition of the Cooley-Tukey FFT[15] for the radix-2 case (when $r_1 = 2$). This definition is then proven to be extensionally equal to that of the DFT. This provides a good definition for the radix-2 case of the FFT, allowing for it to be built on to create future proofs should they require the FFT, however, it does not cover the generalisation on the radix restricting the proof to specific splitting strategies. In another paper, "A Methodology for the Formal Verification of FFT Algorithms in HOL",[21] Akbarpour and Tahar create two definitions of the Cooley-Tukey FFT[15] in Hol for the radix-2 and radix-4 cases. With a primary focus on the radix-2 case Akbarpour and Tahar go on to show equivalence to the DFT across various levels of abstraction.[21] At one stage of this abstraction, Akbarpour and Tahar introduce floating and fixed point arithmetic, showing an analysis of the resultant errors.[21] Much like Capretta[20], this paper also does not make use of a general radix, however, it does highlight how its methodology can be used to analyse general radix FFT implementations.

References

[1] M. Frigo and S. G. Johnson, “The design and implementation of fftw3,” *Proceedings of the IEEE*, vol. 93, pp. 216–231, 2005.

[2] U. Norell, *Towards a practical programming language based on dependent type theory.* Chalmers University of Technology, 2007, vol. 32.

[3] J.-B.-J. Fourier, *Théorie analytique de la chaleur.* F. Didot, 1822.

[4] L. Saribulut, A. Teke, and M. Tümay, “Journal of electrical and electronics engineering research fundamentals and literature review of fourier transform in power quality issues,” vol. 5, pp. 9–22, 2013. [Online]. Available: http://www.academicjournals.org/JEEER

[5] 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, pp. 265–277, 1985. [Online]. Available: https://doi.org/10.1007/BF00348431

[6] A.-C. Clairaut, “Mémoire sur l’orbite apparente du soleil autour de la terre, en ayant égard aux perturbations produites par des actions de la lune et des planètes principales,” *Hist. Acad. Sci. Paris*, pp. 52–564, 1754.

[7] I. M. Gelʹfand, E. G. Glagoleva, and E. E. Shnol, *Functions and graphs.* Springer Science & Business Media, 1990, vol. 1.

[8] G. P. Tolstov, *Fourier Series.* Prentice-Hall, 1962.

[9] C. F. Gauss, “Nachlass: Theoria interpolationis methodo nova tractata,” *Carl Friedrich Gauss Werke*, vol. 3, pp. 265–327, 1866.

[10] M. Bellanger and B. A. Engel, *Digital signal processing : theory and practice*, 10th ed. John Wiley & Sons, Inc., 2024. [Online]. Available: https://ieeexplore.ieee.org/book/10480650

[11] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform.* SIAM, 1992, vol. 10.

[12] H. Heinrich, “Goldstine, h. h., a history of numerical analysis from the 16th through the 19th century.” *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 60, p. 445, 1980. [Online]. Available: http://dx.doi.org/10.1002/zamm.19800600914

[13] G. C. Danielson and C. Lanczos, “Some improvements in practical fourier analysis and their application to x-ray scattering from liquids,” *Journal of the Franklin Institute*, vol. 233, pp. 365–380, 1942. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0016003242907671

[14] I. J. Good, “The interaction algorithm and practical fourier analysis,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 20, pp. 361–372, 1958. [Online]. Available: http://www.jstor.org/stable/2983896

[15] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of Computation*, vol. 19, pp. 297–301, 1965. [Online]. Available: https://dx.doi.org/10.2307/2003354

[16] P. Martin-Löf and G. Sambin, *Intuitionistic type theory.* Bibliopolis Naples, 1984, vol. 9.

[17] M. Frigo, “A fast fourier transform compiler,” in *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, vol. 34. ACM, 5 1999, pp. 169–180.

[18] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, and H. Herbelin, “The coq proof assistant reference manual,” *INRIA, version*, vol. 6, 1999.

[19] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[20] V. Capretta, “Certifying the fast fourier transform with coq,” in *Theorem Proving in Higher Order Logics*, P. B. B. R. J. and Jackson, Eds. Springer Berlin Heidelberg, 2001, pp. 154–168.

[21] B. Akbarpour and S. Tahar, “A methodology for the formal verification of fft algorithms in hol,” in *Formal Methods in Computer-Aided Design*, A. J. Hu and A. K. Martin, Eds. Springer Berlin Heidelberg, 2004, pp. 37–51.