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.

ECS Statement of Originality Template, updated August 2018, Alex Weddell
`aiofficer@ecs.soton.ac.uk`

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[?], 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.[?] 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[?] treatise that any given function can be represented as a harmonic series.[?] 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"[?], the earliest known example of this can be found in work published by Alexis-Claude Clairaut in 1754[?]. 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].[?] 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.[?][?] This definition was published posthumously in 1866, however, it is believed that it was originally written in 1805.[?]

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.

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

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

The DFT Eq. 1 has applications in a variety of fields, such as digital signal processing[?], however, when implemented naïvely, it has poor performance scaling, requiring "$\mathcal{O}\left(n^2\right)$ complex operations" [?]. 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"[?] could be reduced.[?] 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.[?][?] 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 [?] and by Good [?]) were made to create Fast Fourier Transform algorithms (FFT algorithms) which could reduce the complexity of computation to $\mathcal{O}\left(n \log n\right)$. These algorithms, however, where only applicable to a

[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. [?][?]

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

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.[?][?] This FFT Algorithm allows for a given DFT to be computed with $\mathcal{O}\left(n \log n\right)$ complex operations through recursive splitting of the input.[?] Although other FFT Algorithms were discovered before and after the Cooley-Tukey FFT, it is commonly considered to be "the most important FFT"[?].

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.[?][?] 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.[?]

2.3 Related work

FFTW[?] 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.[?] It achieves this title by implementing its own "special-purpose compiler"[?], `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 [?] Eq .4) optimised for that sized transform and the current hardware.[?] 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[?] and Hol[?] with various methods and goals. In the paper "Certifying the Fast Fourier Transform with Coq"[?], Capretta makes use of binary trees to create a definition of the Cooley-Tukey FFT[?] 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",[?] Akbarpour and Tahar create two definitions of the Cooley-Tukey FFT[?] 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.[?] At one stage of this abstraction, Akbarpour and Tahar introduce floating and fixed point arithmetic, showing an analysis of the resultant errors.[?] Much like Capretta[?], 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.

3 Implementation

Before the DFT and FFT can be reasoned on, it is important to define a framework which can accurately encode all required data, as well as operations on that data. For the DFT and FFT, this requires the definition of a number format, and a structure in which these numbers can be represented.

3.1 Complex Numbers

The Agda Standard library does not provide definitions for Complex numbers, it is therefore necessary for us to design and decide upon an encoding.

One method defining this encoding would be to directly use the `Builtin` definition of floating point numbers, and create strict definitions for each of the basic operations. This method would, however, be non ideal. Unlike the definitions for other number systems in Agda, `Agda.Builtin.Float` exists only as an interface around IEEE754 floating point numbers and does not have a corresponding Agda definition. As this is not built up directly in Agda properties on these floating point numbers cannot be formed without assumptions. This would make any proof built upon them weaker.

Cite the IEEE754 c nition

cite https://ag ins.html

Instead, a record defining the operations and properties required of any Complex number can be created. This is equivalent to the definition of an interface in Java, and keeps definitions and proofs on the Fourier Transforms separate to the definition of *number* chosen.

As well as defining addition, multiplication, and other operations, any implementation of complex numbers requires some specific properties be present. Below is a minimal example of this defintion of Complex.

```
record Cplx : Set₁ where
```

Consider d scribing th is the carri

```
field
  ℂ : Set
  _+_ : ℂ → ℂ → ℂ
```

Addition, multiplication and negation must be proven to form a commutative ring.

```
  +-*-isCommutativeRing : IsCommutativeRing _+_ _*_ -_ 0ℂ 1ℂ
```

I should d scribe a) w a commuta ring is b) v we give a s as well as what a con tative ring

Roots of unity as described for Complex numbers in Equation 2, must be defined for some non-zero divisor N and some power K, along with some properties on them. The this nonZero property is an instance argument, allowing an instance resolution algorithm to perform automatic resolution on it, simplifying further proofs.

CITE

```
  -ω : (N : ℕ) → .⦃ nonZero-n : NonZero N ⦄ → (k : ℕ) → ℂ
  ω-N-0    : -ω N 0             ≡ 1ℂ
  ω-N-mN   : -ω N (N *ₙ m)      ≡ 1ℂ
```

```
ω-r₁x-r₁y   : -ω (r₁ *ₙ x) (r₁ *ₙ y) ≡ -ω x y
ω-N-k₀+k₁ : -ω N (k₀ +ₙ k₁)           ≡ (-ω N k₀) * (-ω N k₁)
```

As well as avoiding the difficulties which would comes from floating point arithmetic, isolating the definition of the Complex numbers allows for any proof made upon them to be lifted onto any finite field with complex roots of unity which holds the required properties. In turn this means that any implementations of the FFT using this definition can be utilised upon such a field allowing, for example, for fast multiplication to be performed on this field.

uld do
etter ex-
ion

3.2 Matricies

In Equations 1 and 4, the DFT and FFT are both defined for any input vector x of length N and length $r_1 \times r_2$ respectively. This implies that it would be possible to represent the input structure for both the DFT and the FFT in vector form, possibly using the Agda standard libraries functional vector definition, `Data.Vec.Functionals`.

Although this structure is ideal for the DFT, the FFTs reliance on index splitting, as described in Equation 3, would mean any such definition would require a large amount of low level index manipulation. This would make an kind of reasoning on the FFT, as well as any generalisation where the FFT is called iteratively difficult as both would be pulled down to require the same low level of index manipulation.

The need for this low level manipulation can be removed, by creating some definition for shaped, multi-dimensional matrices, and allowing the FFT to accept these shaped matrices as inputs. As well as removing the need these low level manipulations, using this definition will also abstract the splitting of the input vector out of the FFT making any definition radix independent.

Matrix shapes take the form of tensor products, meaning any shape is either a leaf, or a product of two shapes. Each leaf, ι n, is constructed from a natural number, one leaf can be considered to add one dimension to the overall shape. Each product is then constructed on two shapes and takes the form s ⊗ p. This allows shapes to form binary trees which are able to describe the structure of any multidimensional matrix.

```
data Shape : Set where
  ι : ℕ → Shape
  _⊗_ : Shape → Shape → Shape
```

Matrices can then be inductively defined as a dependant type on Shapes. This definition takes the same form as that of shapes and defines the position of a non-leaf nodes as being constructed by the positions of its two children position nodes, while leaf nodes are bound by the length of that leaf. This binding on the length of the leaf, allows the type checker to require evidence that the length is not greater than the length, removing the possibility for runtime out of bounds errors.

```
data Position : Shape → Set where
  ι : Fin n → Position (ι n)
  _⊗_ : Position s → Position p → Position (s ⊗ p)
```

Position can then be used to define the matrix data encoding, such that matrices form indexed types accepting a position and returning the value at that position.

cite agda.readthedocs.io data-types.html indexed-datatypes

```
Ar : Shape → Set → Set
Ar s X = Position s → X
```

This means any given matrix of Shape s and type X accept a Position of shape s and returns a value of type X.

TODO: Ta about how is related t AS's defini in the bloc multiplicat paper

3.2.1 Methods on one dimension

Given the definition of matrices, some basic operations upon them can be described. The first of these definitions can be restricted to operate only upon the one dimensional case

Head and Tail allow for the deconstruction of any matrix of shape ι (suc n). head_1 returns the first element of the matrix, while tail_1 returns all following elements in a matrix of shape ι n. These operations allow recursion over single dimensional matrices to be defined.

```
head₁ : Ar (ι (suc n)) X → X
head₁ ar = ar (ι fzero)

tail₁ : Ar (ι (suc n)) X → Ar (ι n) X
tail₁ ar (ι x) = ar (ι (fsuc x))
```

One feature of Agda which I make use of regularly is seen here, pattern matching. This is a feature taken from Haskell which allows us to break down some types of input fields to the types they are built on. In the above example ι x is of type Position (suc n), which is deconstructed to expose x of type Fin (suc n).

CITE this

Sum can then be defined over the one dimensional case. One interesting observation of the implementation of sum here, is that it is defined for any carrier set X, and commutative monoid _·_. Although in this case sum is only used on the addition of Complex numbers, the same definition could, for example, be used to produce an implementation of Π over the Natural numbers. As sum is defined in this general manor, ϵ is used to represent the identity element, for our summation (Σ), this is bound to 0 while _·_ is bound to _+_.[3]

Add citati what a mo is, and a d tion

```
sum : (xs : Ar (ι n) A) → A
sum {zero}   xs = ε
sum {suc n} xs = (head₁ xs) · (sum ∘ tail₁) xs
```

Consider r ing or shri the below tion

[3] The definition of sum described here differs slightly from that used in the final proof which contains some minor optimisations to ease proofs

Index's in a single dimension . As defined above, Position encodes the bounds on a given index, as well as the index itself. When calculating the DFT some arithmetic on this index is required, this arithmetic would be overly complex if performed while the index is wrapped in a position, and so helper functions are required to convert a given position to its index value. This helper function for the single dimensional case is shown below.

```
iota : Ar (ι N) ℕ
iota (ι i) = toℕ i
```

3.3 DFT

Given the above definition of the complex numbers, matrices, and methods on one dimensional matrices, the formation of the DFT is now trivial. First a function DFT' is formed, this is of the same shape as Equation 1, but requires that the length of any input vector is non zero, as to satisfy this same condition on the divisor of -ω as defined in 3.1.

```
DFT' : ⦃ nonZero-N : NonZero N ⦄ → Ar (ι N) ℂ → Ar (ι N) ℂ
DFT' {N} xs j = sum λ k → xs k * -ω N (iota k *ₙ iota j)
```

It is then trivial to form the DFT without this restriction, by checking if a given array is of length zero, and returning that same array of length zero when this is the case.

3.4 Reshape

For any definition of FFT, some operations such as transpose, are be required to modify the shape of the input and intermediate matrix. As many such operations exist, a language of reshapes can be created, allowing the creation of general rules across all reshape operations.

For this language, each reshape operations can be considered as a bijective function from shape s to shape p. As this ensures that no matrix can loose or gain data, creating a strict reshape language will strengthen any reasoning in future proofs. This also means that any reshape operation is reversible which will allow for the formation of rules which are general to all operations in the reshape language.

The reshape language is defined as a set of operations from shape to shape as follows.

```
data Reshape : Shape → Shape → Set where
  eq    : Reshape s s                                        -- Identity
  _∙_   : Reshape p q → Reshape s p → Reshape s q            -- Composition of Reshapes
  _⊕_   : Reshape s p → Reshape q r → Reshape (s ⊗ q) (p ⊗ r) -- Left/ Right application
  split : Reshape (ι (m * n)) (ι m ⊗ ι n)                    -- "Vector" → 2D Matrix
  flat  : Reshape (ι m ⊗ ι n) (ι (m * n))                    -- 2D Matrix → "Vector"
  swap  : Reshape (s ⊗ p) (p ⊗ s)                            -- Transposition
```

Using this definition of reshape and some standard library methods on Fin, it is then possible do define the application of reshape to positions and matrices.

```
_⟨_⟩ : Position p → Reshape s p → Position s
i           ⟨ eq     ⟩ = i
i           ⟨ r ∙ r₁ ⟩ = i ⟨ r ⟩ ⟨ r₁ ⟩
(i ⊗ j)     ⟨ r ⊕ r₁ ⟩ = (i ⟨ r ⟩) ⊗ (j ⟨ r₁ ⟩)
(ι i ⊗ ι j) ⟨ split  ⟩ = ι (combine i j)
ι i         ⟨ flat   ⟩ = let a , b = remQuot _ i in ι a ⊗ ι b
(i ⊗ j)     ⟨ swap   ⟩ = j ⊗ i

reshape : Reshape s p → Ar s X → Ar p X
reshape r a ix = a (ix ⟨ r ⟩ )
```

3.4.1 Reverse

As each reshape operation is a bijective function, it is trivial to define a reverse method.

```
rev : Reshape s p → Reshape p s
rev eq = eq
rev (r ⊕ r₁) = rev r ⊕ rev r₁
rev (r ∙ r₁) = rev r₁ ∙ rev r
rev split = flat
rev flat = split
rev swap = swap
```

From this operation, rules on reshape can be defined, allow for formation of relations between reshape operations. This allows for the reduction of the reshape language when operations such as split ∙ flat occur.

```
rev-eq :
  ∀ (r : Reshape s p)
    (i : Position p )
  ---------------------
  → i ⟨ r ∙ rev r ⟩ ≡ i

rev-rev :
  ∀ (r : Reshape s p)
    (i : Position p )
  ----------------------------
  → i ⟨ rev (rev r) ⟩ ≡ i ⟨ r ⟩
```

3.4.2 Recursive Reshaping

While the above operations of reshape can be applied to matrices of a fixed shape this language of reshapes can be improved with the creation of recursive reshape operations.

Flatten and Unflatten enable the recursive application of flat and split respectively. This allows for an N-dimensional matrix to be flattened, and for any single dimensional matrix of size length s to be unflattened into a matrix of shape s.

$$\begin{aligned}
&\flat : \text{Reshape } s\ (\iota\ (\text{length } s))\\
&\flat\ \{\iota\ x\ \} = \text{eq}\\
&\flat\ \{s \otimes s_1\} = \text{flat} \cdot \flat \oplus \flat
\end{aligned}$$

$$\begin{aligned}
&\text{-- Unflatten is free from flatten}\\
&\sharp : \text{Reshape } (\iota\ (\text{length } s))\ s\\
&\sharp = \text{rev } \flat
\end{aligned}$$

Recursive transpose defines an application of transposition for any multi dimensional matrix. Recursive transpose applies swap to any non leaf nodes, allowing for any given function designed to operate on multi dimensional matrices, such as the FFT, to do the same.

this is
worded
silly,
e maybe?

$$\begin{aligned}
&\text{recursive-transpose} : \text{Shape} \to \text{Shape}\\
&\text{recursive-transpose}\ (\iota\ x\ ) = \iota\ x\\
&\text{recursive-transpose}\ (s \otimes s_1) = \text{recursive-transpose}\ s_1 \otimes \text{recursive-transpose}\ s
\end{aligned}$$

3.5 Multi dimensional matrix operations

text here
needing
before
g FFT

Zip With performs pointwise application of a given function f over two matrices of the same shape.

$$\begin{aligned}
&\text{zipWith} : (X \to Y \to Z) \to \text{Ar}\ s\ X \to \text{Ar}\ s\ Y \to \text{Ar}\ s\ Z\\
&\text{zipWith}\ f\ arr_1\ arr_2\ pos = f\ (arr_1\ pos)\ (arr_2\ pos)
\end{aligned}$$

This has many uses, below is shown one example where zipWith is used over matrices x and y, of shape (ι n $\otimes$ ι m), to add the values at each position.

this is
rrect way
, syntax
ht

$$\text{zipWith}\ \_+\_ \begin{bmatrix} x_{1,1} & \dots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,n} \end{bmatrix} \begin{bmatrix} y_{1,1} & \dots & y_{1,n} \\ \vdots & \ddots & \vdots \\ y_{m,1} & \dots & y_{m,n} \end{bmatrix} \equiv \begin{bmatrix} x_{1,1}+y_{1,1} & \dots & x_{1,n}+y_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1}+y_{m,1} & \dots & x_{m,n}+y_{m,n} \end{bmatrix}$$

Map is similar to zipWith, but operates over a singular matrix, applying a function f to each element.

$$\begin{aligned}
&\text{map} : (f : X \to Y) \to \text{Ar}\ s\ X \to \text{Ar}\ s\ Y\\
&\text{map}\ f\ arr\ i = f\ (arr\ i)
\end{aligned}$$

The functions nest and unnest can then be defined to create an isomorphism between matrices of the form Ar (s $\otimes$ p) X and nested matrices of the form A s (Ar p X).

This allows for the definition of a new function mapRows which can apply a given function to the rows of a given matrix.

```
nest : Ar (s ⊗ p) X → Ar s (Ar p X)
nest arr i j = arr (i ⊗ j)

unnest : Ar s (Ar p X) → Ar (s ⊗ p) X
unnest arr (i ⊗ j) = arr i j

mapRows : ∀ {s p t : Shape} → (Ar p X → Ar t Y) → Ar (s ⊗ p) X → Ar (s ⊗ t) Y
mapRows f arr = unnest (map f (nest arr))
```

3.6 FFT

Given the above operations, it is now possible to begin forming a definition for the FFT. For these initial definitions, it is assumed that all input vectors are non zero. In the final implementation this property is a requirement for each function, however, it decreases readability and as such is not shown here.

Looking at the basic derivation of the Cooley Tukey FFT over an input vector defined in Equation 4, three distinct sections can be observed.

$$X_{j_1 r_1 + j_0} = \underbrace{\sum_{k_0=0}^{r_2-1} \left[\underbrace{\left(\underbrace{\sum_{k_1=0}^{r_1-1} x_{k_1 r_2 + k_0} \omega_{r_1}^{k_1 j_0}}_{Section A} \right) \omega_{r_1 r_2}^{k_0 j_1}}_{Section B} \right] \omega_{r_2}^{k_0 j_1}}_{Section C} \tag{5}$$

Section A takes the form of a DFT of length r_1. In vector form, the first element of the input for this DFT is located at index k_0, each subsequent input is then found taken by making a step of r_2, r_1 times. In vector form this is a relatively complex input to reason upon, when we can instead consider our input in matrix form, initially, as a matrix of shape $\iota\ r_1 \otimes \iota\ r_2$. In this form, Section A can be considered to apply the DFT to each column of the input matrix. Similar to Section A, Section C then takes the form of a DFT of length r_2. In our $\iota\ r_1 \otimes \iota\ r_2$ matrix form, this is equivalent to the application of the DFT over the rows of the result of section B.

Section B differs to section A and C, and applies what are generally referred to as, the twiddle factors. In matrix form this section is equivalent to a point wise multiplication over each element from Section A. This step can be represented in Agda as zipWith _*_, on a matrix containing these "twiddle factors".

n that k_0
are the
variable
here

```
2D-twiddles : Ar (ι r₂ ⊗ ι r₁) ℂ
2D-twiddles {r₁} {r₂} (k₀ ⊗ j₁) = -ω (r₂ *ₙ r₁) (iota k₀ *ₙ iota j₁ )
```

Using this twiddle matrix, the definition for the two dimensional FFT is generated by forming each section into its own step. Of note in the definition below are the three uses of swap. The first swap allows DFT' to map over the columns of the input array, while the next allows it to instead map over the rows. The final swap is performed because, given an input in row major order, the result of the FFT is read in column major order, and so the output matrix should be the transposition of the input matrix.

```
2D-FFT : Ar (ι r₁ ⊗ ι r₂) ℂ → Ar (ι r₂ ⊗ ι r₁) ℂ
2D-FFT {r₁} {r₂} arr =
  let
    innerDFTapplied        = mapRows (DFT' {r₁}) (reshape swap arr)
    twiddleFactorsApplied  = zipWith _*_ innerDFTapplied 2D-twiddles
    outerDFTapplied        = mapRows (DFT' {r₂}) (reshape swap twiddleFactorsApplied)
  in reshape swap outerDFTapplied
```

Given knowledge that the DFT should be equivalent to the FFT, the two dimensional definition can then be improved by applying the FFT at each step. This requires the slight modification of the FFT implementation such that it accepts a matrix of any shape s as input.

The definition for the twiddle factors must also be redefined, such that twiddles can be computed for any shape with more than two dimensions. It is easy to see, that the previous base of the roots of unity, $r_1 \times r_2$, maps directly to the flat length of any given matrix. To calculate the power of the root of unity, we can define offset-prod to multiply the flattened index values of the left and right position trees.

This expli
doesn't sit
with me

```
offset-prod : Position (s ⊗ p) → ℕ
offset-prod (k ⊗ j) = iota (k ⟨ ♯ ⟩) *ₙ iota (j ⟨ ♯ ⟩)

twiddles : Ar (s ⊗ p) ℂ
twiddles {s} {p} i = -ω (length (s ⊗ p)) (offset-prod i)
```

The definition of this general twiddle matrix now allows for FFT' to be defined for an input of any shape. Where the two dimensional FFT returned the transposition of the input, the new general case FFT now returns the recursive transposition, as whenever the FFT is applied the sub matrices are transposed.

```
FFT' : Ar s ℂ → Ar (recursive-transpose s) ℂ
FFT' {ι N} arr = DFT' arr
FFT' {r₁ ⊗ r₂} arr =
  let
    innerDFTapplied        = mapRows FFT' (reshape swap arr)
    twiddleFactorsApplied  = zipWith _*_ innerDFTapplied twiddles
    outerDFTapplied        = mapRows FFT' (reshape swap twiddleFactorsApplied)
  in reshape swap outerDFTapplied
```

Because of its generallity, this expression of the FFT is very good. As time was invested at the start of the project into a the creation of a language on matrices and reshaping, every case of the Cooley Tukey algorithm can be represented within the three lines shown above. Given a proof of correctness, this generalitlty makes way for further experiments into different radix sizes, and combination of radix sizes, to be easily undertaken.

If this was instead written in C, or a C style language, this level of generality would be almost impossible. Any such general, C style implementation would require many, low level, index manipulations. Without structures such as those defined for here for position, these index manipulations become increasingly complex as the radix sizes, and levels of nesting, increase. This complexity makes it difficult to reason upon any such implementation meaning garuntees are more challenging to achive.

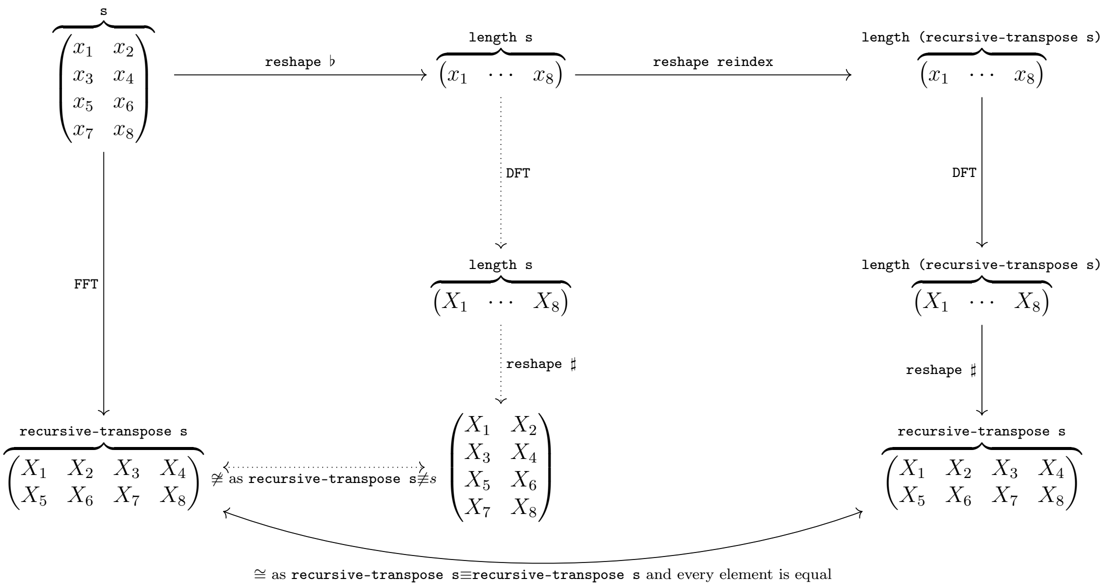
4 Proof of correctness

Given the above defintion of the FFT, and our previous definition of the DFT, a proof of equality between the two can be formed. This proof consists of two component parts, the first of which is the proposition. The proposition describes the relationship between the FFT and DFT and is shown below.

```
fft'≅dft' :
      ⟦ nz-s : NonZero_s s ⟧
    → ∀ (arr : Ar s ℂ)
    → FFT' arr
      ≅
      ( (reshape ♯)
      ∘ (DFT' ⟦ nz-# (nz^t nz-s) ⟧ )
      ∘ (reshape (reindex (|s|≡|s^t| {s}) · ♭))) arr
```

To define this relation, pointwise equality _≅_ is used. This defines equality between two matrices of shape s to hold when ∀ (i : Position s) → xs i ≡ ys i. This allows for proofs to be defined for a general position i.

As the DFT operates on the vector form, reshape operations are used to flatten the input matrix and unflatten the output for comparison. Not mentioned previously, is the reindex operation. As the output of the FFT must be read in column major order, it is of the form recursive-transpose s. When flattened this gives a matrix of shape ι (length (recursive-transpose s)). Meanwhile, without the use of reindex, the output of the DFT is of shape ι (length s). Reindexing allows this to be modeled as ι (length (recursive-transpose s)) without changing the ordering of the results in this matrix. This allows us to make use of pointwise equality.


s
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
reshape ♭
length s
$(x_1 \cdots x_8)$
reshape reindex
length (recursive-transpose s)
$(x_1 \cdots x_8)$
DFT
DFT
FFT
length s
$(X_1 \cdots X_8)$
length (recursive-transpose s)
$(X_1 \cdots X_8)$
reshape ♯
reshape ♯
recursive-transpose s
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8
≇ as recursive-transpose s≢s
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8
recursive-transpose s
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8
≅ as recursive-transpose s≡recursive-transpose s and every element is equal

4.1 Chain of Reasoning

While the proposition defines what we wish to prove, the chain of reasoning is used to justify that the proof holds. The full proof can be found in the attached files, while the most important sections are discussed here. It is important to note that as proofs must hold every invariant, at every step a large amount of complexity is held within this chain of reasoning. As done previously to hide NonZero, as much complexity as possible is hidden in the below extracts from the main chain of reasoning as to improve readability. This complexity remains important, however, as it what allows the strict guarantees provided by Agda to hold.

4.1.1 Inductive Step

The core of this proof which allows its application to any shape is the inductive step, this is also the first section of the proof.

```
    fft'≅dft' {ι N} ⟦ ι nz-N ⟧ arr i = refl                                  (1)
    fft'≅dft' {r₁ ⊗ r₂} ⟦ nz-r₁ ⊗ nz-r₂ ⟧ arr (j₁ ⊗ j₀) =                     (2)
```

These first two lines of this chain of reasoning split the proof on the shape of the input matrix. 1 pattern matches the case where the shape is one dimensional, as FFT on such a shape is equal by definition to the DFT, no chain of reasoning is required to prove this case. This is the base case of the induction. 2 pattern matches on the alternate case, and precedes the remainder of the proof, where r₁ and r₂ represent the left and right sub shapes.

```
        begin
          FFT' {r₂} (λ k₀ →
               FFT' {r₁} (λ k₁ → _) j₀ * _                                   (3)
          ) j₁
        ≡⟨ fft'≅dft' _ j₁ ⟩
          DFT' {# r₂ ᵗ} (λ k₀ →
               FFT' {r₁} (λ k₁ → _) j₀ * _                                   (4)
          ) (j₁ ⟨ ♯ ⟩)
        ≡⟨ DFT'-cong (λ x → cong₂ _*_ (fft'≅dft' _ j₀) refl) (j₁ ⟨ ♯ ⟩ ) ⟩
          DFT' {# r₂ ᵗ} (λ k₀ →
               DFT' {# r₁ ᵗ} (λ k₁ → _) (j₀ ⟨ ♯ ⟩) * _                       (5)
          ) (j₁ ⟨ ♯ ⟩)
        -- ...
```

Splitting upon the shape allows the left hand side to take the form shown in step 3. Step 4 and 5 are then able to apply the proposition currently being proven to the outer and inner instances of FFT'. This allows both instances to be represented as DFT', which in turn allows for the representation of the left hand side in sum form.

```
        ≡⟨⟩
          sum {# r₂ ᵗ} (λ k →
               sum {# r₁ ᵗ} (λ k₁ →
                    arr _
                 *
                   -ω _ _ -- Inner DFT -ω
               )
            *
               -ω _ _        -- Twiddle Factor -ω
            *
               -ω _ _        -- Outer DFT -ω
          )
```

4.1.2 Cooley Tukey Derivation

Using the rule that $c \times \sum_{i=0}^{n} x_i \equiv \sum_{i=0}^{n} cx_i$, the two instances of -$\omega$ in the outer sum, can be moved into the inner sum. With all instances of -ω gathered, the rules of the roots of unity can be used, following the inverse of the initial Cooley Tukey derivation to represent all roots of unity as one root of unity.

en like
ear old
heir pen
e, reword

```
cooley-tukey-derivation :
  ∀ (r₁ r₂ k₀ k₁ j₀ j₁ : ℕ)
  → ⦃ nonZero-r₁ : NonZero r₁ ⦄
  → ⦃ nonZero-r₂ : NonZero r₂ ⦄
  →
    -ω
      (r₂ *ₙ r₁)
      ⦃ m*n≢0 r₂ r₁ ⦄
      (
        (r₂ *ₙ k₁ +ₙ k₀)
        *ₙ
        (r₁ *ₙ j₁ +ₙ j₀)
      )
    ≡
      -ω (r₁) (k₁ *ₙ j₀)
    * -ω (r₂ *ₙ r₁) ⦃ m*n≢0 r₂ r₁ ⦄ (k₀ *ₙ j₀)
    * -ω (r₂) (k₀ *ₙ j₁)
cooley-tukey-derivation r₁ r₂ k₀ k₁ j₀ j₁ ⦃ nonZero-r₁ ⦄ ⦃ nonZero-r₂ ⦄
  = rearrange-ω-power
  ⊡ split-ω
  ⊡ remove-constant-term
  ⊡ simplify-bases
```

4.1.3 Nesting of Sums

5 Review of Implementation

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.