The Fast Fourier Transform and its Applications

Gillian Smith

August 2019

Contents

1 Introduction

The Fast Fourier Transform (commonly abbreviated as FFT) is a fast algorithm for computing the discrete Fourier transform of a sequence. The purpose of this project is to investigate some of the mathematics behind the FFT, as well as the closely related discrete sine and cosine transforms. I will produce a small library of MATLAB code which implements the algorithms discussed, and I will also look into two real-world applications of the FFT: solving partial differential equations and JPEG compression.

2 Methods

I began by studying the Fourier transform and the discrete Fourier transform, by reading the textbook [1], which has a chapter dedicated to the FFT and related concepts. From here I gained an understanding of how the discrete Fourier transform is related to the continuous Fourier transform, as well as how the FFT works.

The Fourier transform is an integral transform given by the formula

$$\mathcal{F}\{f(t)\} = \hat{f}(k) = \int_{-\infty}^{\infty} e^{-2\pi ikt} f(t)\,\mathrm{d}t.$$

It takes the function $f(t)$ as input and outputs the function $\hat{f}(k)$. We usually think of f as a function of time t and $\hat{f}$ as a function of frequency k. The Fourier transform has various properties which allow for simplification of ODEs and PDEs. For example, if $f^{(n)}$ denotes the nth derivative of f, then $\mathcal{F}\{f^{(n)}(t)\}(k) = (2\pi ik)^n \mathcal{F}\{f(t)\}(k) = (2\pi ik)^n \hat{f}(k)$ [2].

The discrete Fourier transform (DFT) of an array of N complex numbers $f_0, f_1, \ldots, f_{N-1}$ is another array of N complex numbers $F_0, F_1, \ldots, F_{N-1}$, defined by

$$F_n = \sum_{j=0}^{N-1} f_j e^{-2\pi inj/N}.$$

The DFT can provide an approximation to the continuous Fourier transform of a function f. Suppose we take N samples $f_j = f(t_j)$, where $t_j = jh$, $j = 0, 1, \ldots, N-1$, where h is the sampling interval. Then we can estimate that $\hat{f}(k_n) \approx hF_n$ at the frequencies $k_n = \frac{n}{Nh}$, $n = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, \frac{N}{2}$.

Taking inspiration from the algorithms at [4] and [5], I implemented the FFT in two slightly different ways using the programming language MATLAB. I then used a MATLAB script to compare the runtimes of both algorithms.

Given an ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$ and an initial condition $y(t_0) = y_0$, we can approximate the solution y with Euler's method. Fix a step size h, then let $t_n = t_0 + hn$, for $n = 0, 1, \ldots, N$. Then perform the recurrence relation $y_{n+1} = y_n + hf(t_n, y_n)$ for $n = 0, 1, \ldots, N$, to obtain the approximation $y(t_n) \approx y_n$. This is the simplest method for numerically solving initial value problems, but with a small enough step size, it can be very accurate [6].

One final technique, which is the central idea behind the compression of JPEG image files, is the discrete cosine transform. The DCT is similar to the DFT, but one main difference is that it uses real numbers only. The reason the DCT is well-suited to compression is because a signal can be reconstructed with reasonable accuracy from just a few low-frequency components of its DCT.

3 Results

If we consider the array of f_js as a vector $\vec{f}$, and its DFT as a vector $\vec{F}$, then we can compute the DFT via a matrix-vector multiplication: $\vec{F} = \mathbf{W}\vec{f}$, where

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & w^3 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \cdots & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & \cdots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \cdots & w^{(N-1)^2} \end{pmatrix}.$$

Multiplying a vector by a matrix requires $\mathcal{O}(N^2)$ operations, which can be very slow for large N. The Danielson-Lanczos Lemma states that the DFT of $\vec{f}$ can be rewritten in terms of two DFTs of length $N/2$. If we impose the restriction that N is a power of 2, then this lemma can be applied recursively until we are left with the N transforms of length 1, and the DFT of length 1 is just the identity function. Calculating the DFT in this manner reduces the number of operations to $\mathcal{O}(N \log_2 N)$, and is the main idea behind the Fast Fourier Transform algorithm. The steps in the algorithm are discussed in detail in appendix B.

I also investigated an even faster algorithm for computing the DFT in the special case where all of the f_j are real numbers. Making use of this, I have written a "fast sine transform" and "fast cosine transform" as explained in appendix C.

Having now written my own FFT, I went on to explore how the FFT can be used to numerically solve the PDE known as the heat equation, $\frac{\partial}{\partial t}u(x,t) = \alpha^2 \frac{\partial^2}{\partial x^2}u(x,t)$, given an initial condition $u(x,0) = f(x)$, and periodic boundary conditions. Taking the Fourier transform of both sides of the equation with respect to the spatial variable x reduces the problem to a first order ODE for each independent value of the frequency variable k. Taking the Fourier transform of the initial condition $u(x,0)$ turns this into an initial value problem, which is solvable via Euler's method. Figure 1 shows a surface plot of the numerical solution in the case where $u(x,0) = \sin x$. More detail is given in appendix D.1.

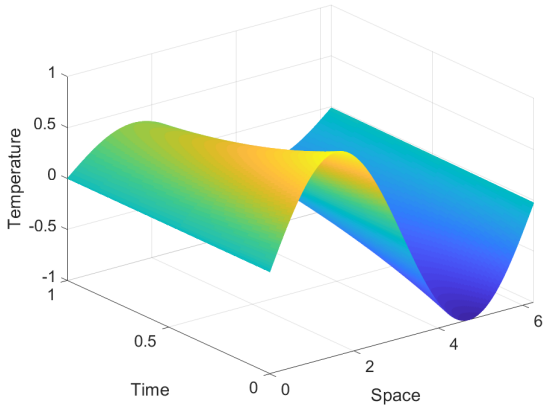


Figure 1: Solution to the heat equation for $x \in (0, 2\pi)$ and $u(x,0) = \sin x$

Finally, I investigated another real-world application of the FFT. The JPEG image file format works by removing some of the detail in an image which will be barely noticed by the human eye, allowing the image to take up significantly less space in memory. I studied the article [7] in order to find out how this is done. The idea is to use the DCT and a process called quantization which disregards some of the higher frequency components, as shown in appendix D.2. I have written a MATLAB script which demonstrates the main steps in this process.

The MATLAB code I created during the course of this project is available on GitHub: `https://github.com/gillian-smith/fft-project`.

4 Discussion/conclusion

The most difficult part of this project was figuring out how the FFT algorithms should be implemented, as well as the algorithms for the discrete sine and cosine transforms. I dealt with this by re-reading the textbook [1] and trying each of the steps on a few small examples, or by figuring it out for myself where there was a lack of explanation in the book. I also spent a lot of time finding where I had made mistakes in my code.

I chose to program in MATLAB because I have used it many times before for university coursework and for personal programming projects. It is also fairly user-friendly, so I did not have to worry too much about the low-level aspects I might have had to consider if I had used another programming language. Additionally, MATLAB makes it relatively easy to produce figures and plots since it comes with various in-built functions for doing so.

The biggest cultural impact of any of the topics covered in this project is that of the discrete cosine transform and JPEG compression. The JPEG format is one of the most popular image file formats,

due to its ability to store large photographs in great detail in a relatively small amount of memory. Furthermore, a modified version of the discrete cosine transform is used for encoding MP3 audio files.

Overall, the objectives of this project have been achieved. Given more time, I would have liked to try solving another more complicated PDE using Fourier methods, or explore convolution, which is another useful application of the Fourier transform.

5 Personal statement

This project has helped me to build on my existing programming skills and gain more experience with MATLAB. Writing this report has tested my skills in communicating mathematics. It was also useful to gain some experience of what it is like to do research. Additionally, I have had the opportunity to investigate an area of mathematics that I might not have studied in depth otherwise.

6 Summary

In this project I aimed to understand and implement the Fast Fourier Transform, an algorithm which has many important applications. I also investigated some related algorithms, and how to use the Fast Fourier Transform to solve the heat equation, a physics problem which describes the distribution of heat in a material over time. Finally, I found out how JPEG files can compress detailed images so that they take up less space in computer memory.

7 Itinerary

Approximately 5 weeks were spent on researching the topics covered and writing MATLAB code, and the final 1 week was spent writing the report, although part of the report was written during the research phase. The financial support was used for subsistence.

8 Acknowledgements

I would like to thank my supervisor, Dr Ben Goddard, for his help and guidance throughout the project. I would also like to thank the University of Edinburgh's College of Science and Engineering for awarding me a College Vacation Scholarship.

Appendices

A The Fourier transform and discrete Fourier transform

A.1 Defining the Fourier transform

The Fourier transform of an integrable function $f : \mathbb{R} \to \mathbb{C}$ is an integral transform, defined as

$$\mathcal{F}\{f(t)\} = \hat{f}(k) = \int_{-\infty}^{\infty} e^{-2\pi ikt} f(t)\,\mathrm{d}t, \tag{1}$$

and the inverse Fourier transform (when it exists) is defined as

$$\mathcal{F}^{-1}\{\hat{f}(k)\} = f(t) = \int_{-\infty}^{\infty} e^{2\pi ikt} \hat{f}(k)\,\mathrm{d}k. \tag{2}$$

One can think of the Fourier transform as changing a function of time into a function of frequency. In other words, if $f(t)$ tells us the amplitude of a signal at time t, then $\hat{f}(k)$ tells us "how much" of each frequency is present in the signal. The functions f and $\hat{f}$ are known as a Fourier transform pair.

It is worth mentioning that there are a few different ways to define the Fourier transform. One possibility is to let $2\pi k = \omega$, so that $\mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} e^{-i\omega t} f(t)\,\mathrm{d}t$ and $\mathcal{F}^{-1}\{\hat{f}(k)\} = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{i\omega t}\hat{f}(k)\,\mathrm{d}k$. Another option is to multiply both the Fourier transform and its inverse by $\frac{1}{\sqrt{2\pi}}$: then $\mathcal{F}\{f(t)\} = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{-i\omega t} f(t)\,\mathrm{d}t$, and $\mathcal{F}^{-1}\{\hat{f}(k)\} = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{i\omega t}\hat{f}(k)\,\mathrm{d}k$ [2].

These various choices of definition are purely conventions and no particular definition is any more correct than the others, however most people have one that they prefer to use. The ω convention is commonly used in physics because it represents angular frequency. I will stick to definitions 1, 2 for the rest of this report.

A.2 Existence of the Fourier transform and inverse Fourier transform

Since the Fourier transform is defined as an improper integral, it is only defined under certain conditions. If f is absolutely integrable, meaning that $\int_{-\infty}^{\infty} |f(t)|\,\mathrm{d}t < \infty$, then it has a Fourier transform. This is because $|\hat{f}(k)| = |\int_{-\infty}^{\infty} e^{-2\pi ikt} f(t)\,\mathrm{d}t| \leq \int_{-\infty}^{\infty} |e^{-2\pi ikt} f(t)|\,\mathrm{d}t = \int_{-\infty}^{\infty} |e^{-2\pi ikt}||f(t)|\,\mathrm{d}t = \int_{-\infty}^{\infty} |f(t)|\,\mathrm{d}t$. So if f is absolutely integrable, then $|\hat{f}(k)| < \infty$ for all k [2].

It is also possible (and useful) to define Fourier transforms of sine and cosine functions, as well as complex exponentials, although the result is defined in terms of delta functions [3] [6, Chapter 6.5]. A list of properties of the Fourier transform, as well as a list of common Fourier transform pairs, can be found on Wikipedia `https://en.wikipedia.org/wiki/Fourier_transform`.

A.3 The discrete Fourier transform

The discrete Fourier transform (DFT) of a finite sequence of N complex numbers $f_0, f_1, \ldots, f_{N-1}$ is another sequence of N complex numbers $F_0, F_1, \ldots, F_{N-1}$, where

$$F_n = \sum_{j=0}^{N-1} f_j e^{-2\pi inj/N}. \tag{3}$$

We can recover the f_js from the F_ns via the inverse discrete Fourier transform:

$$f_j = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{2\pi inj/N}. \tag{4}$$

To make intuitive sense of equation 3, suppose we want to estimate the Fourier transform of a function f from a finite number N of samples $f_j = f(t_j)$, where $t_j = jh$, for $j = 0, 1, 2, \ldots, N-1$. The sampling interval h is the distance between consecutive t_j. The sampling frequency, the number of samples per second, is $1/h$. Since we have N input samples we will be able to produce no more than N independent outputs.

We will seek estimates of $\mathcal{F}\{f\} = \hat{f}$ at frequencies $k_n = \frac{n}{Nh}$, for $n = -\frac{N}{2}, -\frac{N}{2}+1, \cdots, \frac{N}{2}$. Note that we have $N+1$ values of n rather than N, but the existence of the extra output will be resolved later. Approximating the integral in the Fourier transform as a sum, we estimate that

$$\hat{f}(k_n) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i k_n t}\,\mathrm{d}t \approx \sum_{j=0}^{N-1} h f_j e^{-2\pi i k_n t_j} = h\sum_{j=0}^{N-1} f_j e^{-2\pi i \frac{n}{Nh} jh} = h\sum_{j=0}^{N-1} f_j e^{-2\pi i nj/N} = hF_n.$$

Furthermore, the DFT is periodic in n with period N:

$$F_{n+N} = \sum_{j=0}^{N-1} f_j e^{-2\pi i (n+N) j/N} = \sum_{j=0}^{N-1} f_j e^{-2\pi i nj/N} e^{-2\pi i j} = \sum_{j=0}^{N-1} f_j e^{-2\pi i nj/N} = F_n$$

since $e^{-2\pi i j} = 1$ for any integer j. In particular, $F_{-N/2} = F_{N/2}$ and we have only N independent outputs after all. Due to the periodicity, we can choose to instead let n vary from 0 to $N-1$.

We can also represent an N-point DFT as multiplication by an $N \times N$ matrix. Let $w = e^{-2\pi i/N}$, and represent the f_js and F_ns as vectors, $\vec{f} = (f_0, f_1, \ldots, f_{N-1})^\top$ and $\vec{F} = (F_0, F_1, \ldots, F_{N-1})^\top$. Define the matrix $\mathbf{W}$ by $\mathbf{W}_{jk} = w^{jk}$ (where $0 \le j, k \le N-1$), or in other words

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & w^3 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \cdots & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & \cdots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \cdots & w^{(N-1)^2} \end{pmatrix}.$$

Then $\vec{F} = \mathbf{W}\vec{f}$.

Algorithm 1 Computing the DFT and inverse DFT by matrix multiplication

```
1:  procedure SLOW_DFT(f⃗, λ)
2:      N ← LENGTH(f⃗)
3:      θ ← 2πi · λ/N
4:      W ← N × N matrix of ones
5:      for i ← 1, 2, 3, ..., N − 1 do
6:          for j ← 1, 2, 3, ..., i − 1 do
7:              W_ij ← exp(ijθ)
8:              W_ji ← W_ij
9:          end for
10:     end for
11:     F⃗ ← W f⃗
12:     if λ = 1 then F⃗ ← (1/N) F⃗
13:     end if
14: return F⃗
15: end procedure
```

B The Fast Fourier Transform

B.1 The FFT algorithm

If we compute the DFT of an N-point sequence directly from the definition (equation 3) or with algorithm 1, the number of operations required is $\mathcal{O}(N^2)$. This gets very slow as N increases, so we should try to find a better option. Fast Fourier Transforms (FFT) are a family of algorithms for computing the DFT in just $\mathcal{O}(N \log_2 N)$ operations. This section of the report will explain a simple version of the variant known as the Cooley-Tukey FFT.

The Danielson-Lanczos Lemma will help us to develop a divide-and-conquer algorithm for computing the DFT. It states that a DFT of length N, where N is an even number, can be rewritten in terms of two DFTs of length $N/2$. The proof is as follows [1, Chapter 12.2]:

$$\begin{aligned}
F_n &= \sum_{j=0}^{N-1} e^{-2\pi ijn/N} f_j \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{-2\pi i(2j)n/N} f_{2j} + \sum_{j=0}^{\frac{N}{2}-1} e^{-2\pi i(2j+1)n/N} f_{2j+1} \\
&= \sum_{j=0}^{\frac{N}{2}-1} e^{-2\pi ijn/\frac{N}{2}} f_{2j} + e^{-2\pi in/N} \sum_{j=0}^{\frac{N}{2}-1} e^{-2\pi ijn/\frac{N}{2}} f_{2j+1} \\
&= F_n^e + w^n F_n^o
\end{aligned}$$

where F^e denotes the DFT of the even components f_{2j}, F^o is the DFT of the odd components f_{2j+1}, and $w = e^{-2\pi i/N}$.

The following observation enables us to compute F_n and $F_{n+\frac{N}{2}}$ at the same time:

$$\begin{aligned}
F_{n+N/2} &= \sum_{j=0}^{N-1} e^{-2\pi ij(n+N/2)/N} f_j \\
&= \sum_{j=0}^{N-1} e^{-2\pi ijn/N} e^{-2\pi ijN/2N} f_j \\
&= \sum_{j=0}^{N-1} e^{-2\pi ijn/N} e^{-\pi ij} f_j \\
&= \sum_{j=0}^{N-1} e^{-2\pi ijn/N} (-1)^j f_j \\
&= \sum_{j=0}^{N/2-1} e^{-2\pi i(2j)n/N} f_{2j} - \sum_{j=0}^{N/2-1} e^{-2\pi i(2j+1)n/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{-2\pi ijn/\frac{N}{2}} f_{2j} - e^{-2\pi in/N} \sum_{j=0}^{N/2-1} e^{-2\pi ijn/\frac{N}{2}} f_{2j+1} \\
&= F_n^e - w^n F_n^o
\end{aligned}$$

We can calculate the DFT of the whole sequence by using the Danielson-Lanczos lemma several times. Now that the problem has been reduced to computing F_n^e and F_n^o, we can repeat the same argument to reduce the problem to computing F_n^{ee}, F_n^{eo}, F_n^{oe}, and F_n^{oo}, the transforms of length $N/4$.

In the case where N is a power of 2, we can continue to subdivide the transforms until we reach the N transforms of length 1. It follows from the equation 3 that a 1-point DFT is simply the identity operation. There exist Fast Fourier Transforms for various cases where N is not a power of 2, but I will not discuss them in this report. One option is to 'pad' the vector with zeroes, that is, add zeroes to the end of the vector so that its length is a power of 2, and then take the FFT of that vector, however a slightly different result is produced. For the purposes of this project, it is easiest to just assume that N is always a power of 2.

Once we have done all of the subdividing, we find that each 1-point transform corresponds to a unique pattern of $\log_2 N$ es and os. We can find the value of n for which $F_n^{eoeooee\ldots oeoe} = f_n$ as follows: reverse the pattern of es and os, replace e with 0 and o with 1, and here we have the value of n in binary. This means we will have to reorder the input data using what is known as bit-reversal permutation [1, Chapter 12.2].

Now we have an algorithm for computing the DFT: first, permute the $\vec{f}$ vector into bit-reversed order and store the result in $\vec{F}$. This gives the set of 1-point DFTs. Then use a loop to double the length each time in order to compute the transforms of length $2, 4, 8, \ldots, N$. At each stage, use the

Danielson-Lanczos lemma and the transforms already computed to construct the next set of transforms in-place.

The inverse DFT (equation 4) can be calculated via essentially the same algorithm, since the only differences are the sign of the exponent and multiplication by $1/N$ in the inverse.

I have explored two slightly different approaches to the FFT, based on algorithms found on the internet. The first is iterative, using 'for' and 'while' loops [5], and the second is recursive, calling itself repeatedly until it reaches the 'base case' $N = 1$ [4]. Algorithms 2 and 3 were implemented in MATLAB, as functions `fft_iterative` and `fft_recursive`, as well as algorithm 1 for naively computing the DFT directly from the definition, as a function called `slow_dft`. All of the procedures take an additional parameter $\lambda = \pm 1$, which should be set to -1 for the forward DFT and 1 for the inverse.

Algorithm 2 An iterative Fast Fourier Transform

procedure FFT_ITERATIVE($\vec{f}$, λ)
  $N \leftarrow$ LENGTH($\vec{f}$)
  $\vec{F} \leftarrow$ BITREVERSEPERMUTE($\vec{f}$)
  $\theta \leftarrow 2\pi i \cdot \lambda / N$
  $\vec{w} \leftarrow \left(1, \quad \exp\theta, \quad \exp 2\theta, \quad \ldots, \quad \exp\left(\frac{N}{2} - 1\right)\theta\right)$
  $currN \leftarrow 2$
  while $currN \leq N$ **do**
    $halfcurrN \leftarrow currN/2$
    $tablestep \leftarrow N/currN$
    for all $i \in \{0, currN, 2currN, \ldots, N - currN\}$ **do**
      $k \leftarrow 0$
      for $j \leftarrow i, i+1, \ldots, i + halfcurrN$ **do**
        $temp \leftarrow w_k F_{j+halfcurrN}$
        $F_{j+halfcurrN} \leftarrow F_j - temp$
        $F_j \leftarrow F_j + temp$
        $k \leftarrow k + tablestep$
      end for
    end for
    $currN \leftarrow 2currN$
  end while
return $\vec{F}$
end procedure

Algorithm 3 A recursive Fast Fourier Transform

procedure FFT_RECURSIVE($\vec{f}$, λ)
  $N \leftarrow$ LENGTH($\vec{f}$)
  $\theta \leftarrow 2\pi i \cdot \lambda / N$
  $\vec{w} \leftarrow \left(1, \quad \exp\theta, \quad \exp 2\theta, \quad \ldots, \quad \exp\left(\frac{N}{2} - 1\right)\theta\right)$
  $\vec{F} \leftarrow \vec{0} \in \mathbb{C}^N$
  if N = 1 **then**
    $F_0 \leftarrow f_0$
  else
    $\left(F_0, \quad F_1, \quad \ldots, \quad F_{N/2-1}\right) \leftarrow$ FFT_RECURSIVE($(f_0, f_2, f_4, \ldots, f_{N-2})$, λ)
    $\left(F_{N/2}, \quad F_{N/2+1}, \quad \ldots, \quad F_{N-1}\right) \leftarrow$ FFT_RECURSIVE($(f_1, f_3, f_5, \ldots, f_{N-1})$, λ)
    for $k \leftarrow 0, 1, \ldots, N/2 - 1$ **do**
      $temp = F_k$
      $F_k = temp + w_k F_{k+N/2}$
      $F_{k+N/2} = temp - w_k F_{k+N/2}$
    end for
  end if
return $\vec{F}$
end procedure

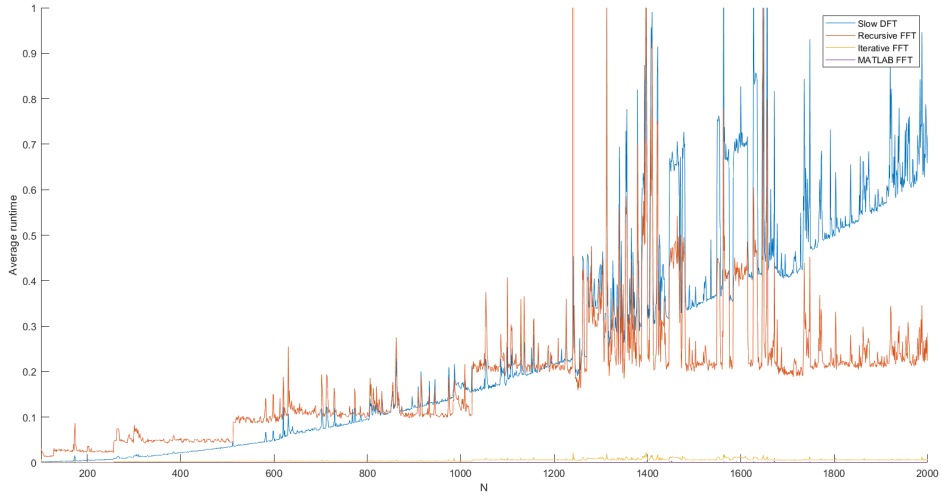


Figure 2: Comparison of the runtimes of four DFT/FFT algorithms, for arrays of lengths from 100 to 2000

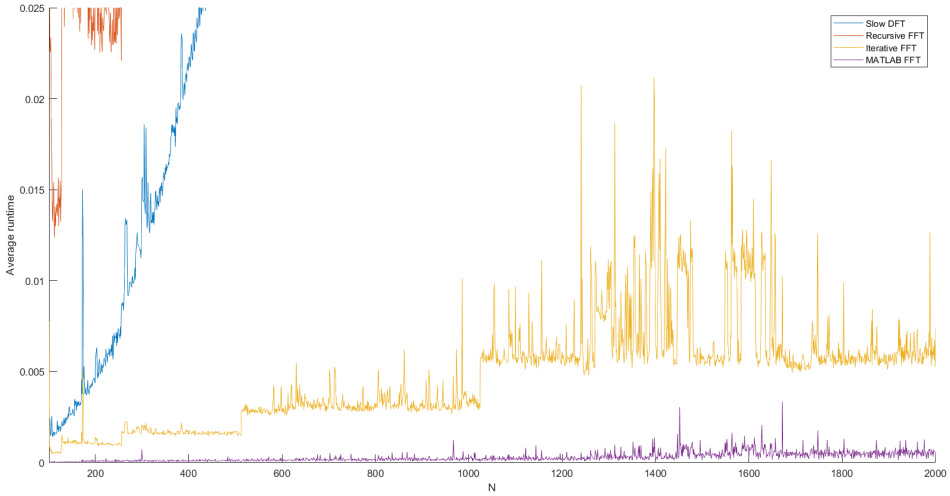


Figure 3: Same plot as figure 2, but with a smaller upper bound on the runtime axis in order to show the difference in speed between the iterative FFT and MATLAB's in-built FFT

Most importantly, the functions `slow_dft`, `fft_iterative` and `fft_recursive` all give the same output, and they all agree with MATLAB's built-in `fft` function (up to small errors).

The runtimes of all three functions, as well as MATLAB's built-in `fft`, were compared, taking the average runtime over five random vectors in $\mathbb{C}^N$ for $N = 2, 4, 8, \ldots, 2^{18}$. The naive approach is slightly faster than algorithms 2 and 3 up to about $N = 32$, but in most real-world applications the data set is much larger. For larger N, the iterative algorithm is significantly faster than the recursive one, despite its three nested loops, although this is probably a consequence of the way MATLAB handles recursion.

Figures 2 and 3 show the difference in runtimes for values of N from 100 to 2000. For each N, ten random vectors of length N were generated and padded with zeroes, and the average runtime of performing each FFT algorithm on each of these vectors was found and plotted. On figure 2, the runtimes of algorithm 2 and MATLAB's in-built FFT are so much faster that they are barely visible.

Given more time and/or computational power, I could have taken the average over more than just ten vectors for each N. I could have also explored values of N larger than 2000, but by this point it is fair to assume that the iterative algorithm will be the fastest of the three. Due to the difference in speed, I will use algorithm 2 for the rest of the project.

I have written wrapper functions `my_fft` and `my_ifft` which call `fft_iterative` with $\lambda = -1$ and $\lambda = 1$ respectively, for ease of use. Also, `my_ifft` multiplies $\vec{F}$ by $1/N$ before returning, as is consistent with equation 4. Both functions will pad the vector $\vec{f}$ with zeroes in the case where its length is not a power of 2.

B.2 FFT of real-valued vectors

It turns out there is an even more efficient way to compute the DFT in the special case where the input vector contains real values only. This method tends to be significantly faster, since we do some clever manipulation of the data and perform an FFT of length $N/2$ instead of one of length N.

Suppose we have a real-valued vector $\vec{f} \in \mathbb{R}^N$. We can consider the even-indexed elements as one array and the odd-indexed elements as another, and define the vector $\vec{h} \in \mathbb{C}^{N/2}$ by $h_j = f_{2j} + if_{2j+1}$, $j = 0, 1, \ldots, N/2 - 1$. Taking the FFT of $\vec{h}$ (using the existing function `my_fft`) gives another vector $\vec{H} \in \mathbb{C}^{N/2}$, where $H_n = F_n^e + iF_n^o$, for $n = 0, 1, \ldots, N/2 - 1$. It can be shown that $H_n + H_{N/2-n} = 2F_n^e$ and $H_n - H_{N/2-n} = 2iF_n^o$. Now all that is left to do is find each F_n by using the Danielson-Lanczos lemma again:

$$\begin{aligned} F_n &= F_n^e + e^{-2\pi in/N} F_n^o \\ &= \mathrm{Re}(H_n) + e^{-2\pi in/N} \mathrm{Im}(H_n) \\ &= \frac{1}{2}(H_n + \overline{H_{N/2-n}}) - \frac{i}{2} e^{-2\pi in/N} (H_n - \overline{H_{N/2-n}}) \end{aligned}$$

for $n = 0, 1, \ldots, N/2$, using the fact that $H_0 = H_{N/2}$ [1, Chapter 12.3]. We can also use $F_{N-n} = \overline{F_n}$ (where $\overline{z}$ denotes the complex conjugate of $z \in \mathbb{C}$) to restore the second half of the array so that the function's output matches that of algorithm 2.

Algorithm 4 Fast Fourier Transform of real data

```
1:  procedure REALFFT(f⃗)
2:      N ← LENGTH(f⃗)
3:      h⃗ ← 0⃗ ∈ ℂ^{N/2}
4:      for j ← 0, 1, ..., N/2 − 1 do
5:          h_j ← f_{2j} + i f_{2j+1}
6:      end for
7:      H⃗ ← MY_FFT(h⃗)
8:      H⃗ ← (H⃗, H_0)
9:      θ ← −2πi/N
10:     F⃗ ← 0⃗ ∈ ℂ^{N/2+1}
11:     for n ← 0, 1, ..., N/2 do
12:         F_n ← ½(H_n + conj(H_{N/2−n})) − (i/2) e^{−2πin/N} (H_n − conj(H_{N/2−n}))
13:     end for
14:     F ← (F_0, F_1, ..., F_{N/2−1}, F_{N/2}, conj(F_{N/2−1}), ..., conj(F_2), conj(F_1))
15: return F⃗
16: end procedure
```

Now we need an algorithm to invert the process, i.e., to find the inverse DFT of an array whose inverse DFT is known to be real. The process is essentially just algorithm 4 in reverse. Suppose we begin with $F_0, F_1, \ldots, F_{N-1}$. For $n = 0, 1, \ldots, N/2 - 1$, let

$$F_n^e = \frac{1}{2}(F_n + \overline{F_{N/2-n}}) = \frac{1}{2}(F_n + F_{N/2+n}),$$

$$F_n^o = \frac{1}{2} e^{2\pi in/N} (F_n - \overline{F_{N/2-n}}) = \frac{1}{2} e^{2\pi in/N} (F_n - F_{N/2+n}).$$

Then construct $H_n = F_n^e + iF_n^o$, and take the inverse FFT of $\vec{H}$ with `my_ifft` to get $h_j = f_{2j} + if_{2j+1}$. Finally, extract the even and odd components of $\vec{f}$ from the real and imaginary parts of this array [1, Chapter 12.3].

Algorithm 5 Inverse Fast Fourier Transform of real data

procedure REALIFFT($\vec{F}$)
  $N \leftarrow$ LENGTH($\vec{F}$)
  $\vec{H} \leftarrow \vec{0} \in \mathbb{C}^{N/2}$
  for $n \leftarrow 0, 1, \ldots, N/2 - 1$ **do**
    $H_n \leftarrow \frac{1}{2}(F_n + F_{N/2+n}) + \frac{i}{2}e^{2\pi i n/N}(F_n - F_{N/2+n})$
  end for
  $\vec{h} \leftarrow$ MY_IFFT($\vec{H}$)
  $\vec{f} \leftarrow \vec{0} \in \mathbb{C}^N$
  for $j \leftarrow 0, 1, \ldots, N/2 - 1$ **do**
    $f_{2j} \leftarrow \mathrm{Re}(h_j)$
    $f_{2j+1} \leftarrow \mathrm{Im}(h_j)$
  end for
return $\vec{f}$
end procedure

B.3 Multidimensional DFT and FFT

Given a 2-dimensional array (i.e. a matrix) with N_1 rows and N_2 columns, and entries $f_{j,k}$, we can define its two-dimensional discrete Fourier transform as an array of the same size, with entries

$$F_{m,n} = \sum_{k=0}^{N_2-1} \sum_{j=0}^{N_1-1} e^{-2\pi i k n/N_2} e^{-2\pi i j m/N_1} f_{j,k}.$$

Rearranging this a little, we find that

$$F_{m,n} = \sum_{k=0}^{N_2-1} e^{-2\pi i k n/N_2} \left(\sum_{j=0}^{N_1-1} e^{-2\pi i j m/N_1} f_{j,k} \right). \tag{5}$$

Equation 5 shows that the 2-dimensional DFT of a matrix can be found by taking the DFT of each row and then taking the DFT of each column [1, Chapter 12.5]. We can do this with the `my_fft` function, which imposes the restriction that both N_1 and N_2 must be powers of 2. This is not necessarily the quickest way to calculate a 2-dimensional DFT, but at least it gives a very simple algorithm 6.

Algorithm 6 2-dimensional Fast Fourier Transform

procedure MY_FFT2($\mathbf{X}$)
  $\mathbf{Y} \leftarrow \mathbf{X}$
  for all rows $\vec{y}$ of $\mathbf{Y}$ **do**
    $\vec{y} \leftarrow$ MY_FFT($\vec{y}$)
  end for
  $\mathbf{Y} \leftarrow \mathbf{Y}^\top$
  for all rows $\vec{y}$ of $\mathbf{Y}$ **do**
    $\vec{y} \leftarrow$ MY_FFT($\vec{y}$)
  end for
  $\mathbf{Y} \leftarrow \mathbf{Y}^\top$
return $\mathbf{Y}$
end procedure

The inverse 2-dimensional DFT is

$$f_{j,k} = \frac{1}{N_1 N_2} \sum_{k=0}^{N_2-1} \sum_{j=0}^{N_1-1} e^{2\pi i k n/N_2} e^{2\pi i j m/N_1} F_{m,n} = \frac{1}{N_2} \sum_{k=0}^{N_2-1} e^{2\pi i k n/N_2} \left(\frac{1}{N_1} \sum_{j=0}^{N_1-1} e^{2\pi i j m/N_1} F_{m,n} \right).$$

We can compute this in a similar way to algorithm 6: take the inverse DFT of each row, then take the inverse DFT of each column.

C The Discrete Sine and Cosine Transforms

C.1 Computing the Discrete Sine Transform

The discrete sine transform (abbreviated as DST) of $f_0 = 0, f_1, f_2, \ldots, f_{N-1} \in \mathbb{R}$ is defined as

$$F_n = \sum_{j=1}^{N-1} f_j \sin\left(\frac{\pi jn}{N}\right). \tag{6}$$

Note that we always take f_0 to be 0. This is because even if we were to include the $j = 0$ term in the sum in equation 6, we would get $f_0 \sin(0\pi n/N) = f_0 \sin 0 = 0$. Therefore the $j = 0$ term contributes nothing, so it makes no difference what f_0 is.

I have written a function `slow_dst` in MATLAB to compute the DST using a matrix, in a similar way to how algorithm 1 computes the DFT. But, as before, this is $\mathcal{O}(N^2)$ - is there a faster way to compute it making use of the FFT?

Suppose we start with the real-valued data $f_1, f_2, \ldots, f_{N-1}$ (and $f_0 = 0$). Construct the auxiliary array $\vec{y}$ as follows:

$$\begin{aligned} y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}), \quad j = 1, 2, \ldots, N-1. \end{aligned}$$

Now take the FFT of $\vec{y}$. We can use `realfft` since all of the values in $\vec{y}$ are real. Due to the use of `realfft`, we need an input vector $\vec{f}$ whose length is a power of 2 (including $f_0 = 0$). The result is

$$\begin{aligned} Y_n &= \sum_{j=0}^{N-1} y_j e^{-2\pi inj/N} \\ &= \sum_{j=0}^{N-1} y_j \left(\cos\left(\frac{-2\pi nj}{N}\right) + i \sin\left(\frac{-2\pi nj}{N}\right)\right) \\ &= \sum_{j=0}^{N-1} y_j \left(\cos\left(\frac{2\pi nj}{N}\right) - i \sin\left(\frac{2\pi nj}{N}\right)\right) \\ &= \sum_{j=0}^{N-1} y_j \cos\left(\frac{2\pi nj}{N}\right) - i \sum_{j=0}^{N-1} y_j \sin\left(\frac{2\pi nj}{N}\right). \end{aligned}$$

Letting $x_j = \sin(j\pi/N)(f_j + f_{N-j})$, $z_j = \frac{1}{2}(f_j - f_{N-j})$, we can see that $x_{N-j} = x_j$ and $z_{N-j} = -z_j$. Then we can show that

$$x_{N-j} \cos\left(\frac{2\pi n(N-j)}{N}\right) = x_j \cos\left(2\pi n - \frac{2\pi nj}{N}\right) = x_j \cos\left(\frac{2\pi nj}{N}\right) \tag{7}$$

$$z_{N-j} \cos\left(\frac{2\pi n(N-j)}{N}\right) = -z_j \cos\left(\frac{2\pi nj}{N}\right) \tag{8}$$

$$x_{N-j} \sin\left(\frac{2\pi n(N-j)}{N}\right) = x_j \sin\left(2\pi n - \frac{2\pi nj}{N}\right) = -x_j \sin\left(\frac{2\pi nj}{N}\right) \tag{9}$$

$$z_{N-j} \sin\left(\frac{2\pi n(N-j)}{N}\right) = z_j \sin\left(\frac{2\pi nj}{N}\right) \tag{10}$$

Denote the real and imaginary parts of Y_n by R_n and I_n:

$$
\begin{aligned}
R_n &= \mathrm{Re}(Y_n) \\
&= \sum_{j=0}^{N-1} y_j \cos\left(\frac{2\pi nj}{N}\right) \\
&= \underbrace{\sum_{j=0}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos\left(\frac{2\pi nj}{N}\right)}_{\text{use (7)}} + \underbrace{\frac{1}{2} \sum_{j=0}^{N-1} (f_j - f_{N-j}) \cos\left(\frac{2\pi nj}{N}\right)}_{=0 \text{ due to (8)}} \\
&= \sum_{j=0}^{N-1} 2 f_j \sin(j\pi/N) \cos\left(\frac{2\pi nj}{N}\right) \\
&= \sum_{j=0}^{N-1} f_j \left(\sin\left(\frac{(2n+1)j\pi}{N}\right) - \sin\left(\frac{(2n-1)j\pi}{N}\right)\right) \\
&= F_{2n+1} - F_{2n-1} \\
I_n &= \mathrm{Im}(Y_n) \\
&= -\sum_{j=0}^{N-1} y_j \sin\left(\frac{2\pi nj}{N}\right) \\
&= \underbrace{-\frac{1}{2} \sum_{j=0}^{N-1} (f_j - f_{N-j}) \sin\left(\frac{2\pi nj}{N}\right)}_{\text{use (10)}} \underbrace{- \sum_{j=0}^{N-1} (f_j + f_{N-j}) \sin\left(j\pi/N\right) \sin\left(\frac{2\pi nj}{N}\right)}_{=0 \text{ due to (9)}} \\
&= -\sum_{j=0}^{N-1} f_j \sin\left(\frac{2\pi nj}{N}\right) \\
&= -F_{2n}.
\end{aligned}
$$

So the even terms of $\vec{F}$ are directly determined by $F_{2n} = -I_n$. The odd terms are given by the recurrence relation $F_{2n+1} = F_{2n-1} + R_n$, for $n = 0, 1, \ldots, {}^N\!/_2 - 1$. To initialize the recurrence start with $n = 0$: $F_1 = F_{-1} + R_0 = -F_1 + R_0 \implies F_1 = \frac{1}{2} R_0$.

Algorithm 7 Discrete sine transform using the FFT

```
1:  procedure MY_DST(f⃗)
2:      N ← LENGTH(f⃗)+1
3:      y⃗ ← 0⃗ ∈ ℝ^N
4:      for j ← 1, 2, ..., N − 1 do
5:          y_j ← sin(jπ/N)(f_j + f_{N−j}) + ½(f_j − f_{N−j})
6:      end for
7:      Y⃗ ← REALFFT(y⃗)
8:      R⃗ ← Re(Y⃗)
9:      I⃗ ← Im(Y⃗)
10:     F⃗ ← 0⃗ ∈ ℝ^N
11:     for n ← 0, 1, ..., N/2 − 1 do
12:         F_{2n} ← −I_n
13:         if n = 0 then
14:             F_1 ← ½R_0
15:         else
16:             F_{2n+1} ← F_{2n−1} + R_n
17:         end if
18:     end for
19: return F⃗
20: end procedure
```

Algorithm 7 calculates the discrete sine transform. The DST is actually its own inverse, up to a factor of $N/2$, so to calculate the inverse DST, simply find the DST and then multiply by $2/N$ [1, Chapter 12.4].

C.2 Computing the Discrete Cosine Transform

There are several versions of the discrete cosine transform, but the one I will focus on is known as the DCT-II:

$$F_n = \sum_{j=0}^{N-1} f_j \cos\left(\frac{\pi n(j+1/2)}{N}\right). \tag{11}$$

Its inverse (times $N/2$) is the DCT-III:

$$f_j = \frac{1}{2}F_0 + \sum_{n=1}^{N-1} F_n \cos\left(\frac{\pi n(j+1/2)}{N}\right). \tag{12}$$

So if we start with a vector $\vec{f}$, take the DCT-II, then take the DCT-III of the result, and finally multiply by $2/N$, we will get back the original vector $\vec{f}$.

Suppose we want to compute the DCT-II of a vector $\vec{f} \in \mathbb{R}^N$. As we did with the sine transform, define an auxiliary array $\vec{y}$ as follows:

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) + \sin\left(\frac{\pi(j+1/2)}{N}\right)(f_j - f_{N-j-1}), \quad j = 0, 1, \ldots, N-1. \tag{13}$$

Then, as before, take the FFT using `realfft`, to get $\vec{Y} = \vec{R} + i\vec{I}$. We find $\vec{F}$ via

$$F_{2n} = \cos(n\pi/N)R_n + \sin(n\pi/N)I_n, \quad F_{2n-1} = \sin(n\pi/N)R_n - \cos(n\pi/N)I_n + F_{2n+1}.$$

This time we have to iterate backwards to find the odd-indexed elements. Initialize the recurrence with $n = N/2$, to find that $F_{N-1} = \sin\frac{\pi}{2}R_{N/2} - \cos\frac{\pi}{2}I_{N/2} + F_{N+1} = R_{N/2} - F_{N-1} \implies F_{N-1} = \frac{1}{2}R_{N/2}$. Now just perform the recurrence for $n = N/2 - 1, N/2 - 2, \ldots, 1, 0$ to find the rest of the elements, and here we have $\vec{F}$ [1, Chapter 12.4].

Algorithm 8 Discrete cosine transform using the FFT

```
1:  procedure MY_DCT(f⃗)
2:      N ← LENGTH(f⃗)
3:      y⃗ ← 0⃗ ∈ ℝ^N
4:      y_0 ← f_0
5:      for j ← 1, 2, ..., N − 1 do
6:          y_j ← ½(f_j + f_{N−j−1}) + sin(π(j + 1/2)/N)(f_j − f_{N−j−1})
7:      end for
8:      Y⃗ ← REALFFT(y⃗)
9:      R⃗ ← Re(Y⃗)
10:     I⃗ ← Im(Y⃗)
11:     F⃗ ← 0⃗ ∈ ℝ^N
12:     F_{N−1} ← ½R_{N/2}
13:     for n ← N/2 − 1, N/2 − 2, ..., 1, 0 do
14:         c ← cos(nπ/N)
15:         s ← sin(nπ/N)
16:         F_{2n} ← cR_n + sI_n
17:         if n ≠ 0 then
18:             F_{2n−1} ← sR_n − cI_n + F_{2n+1}
19:         end if
20:     end for
21: return F⃗
22: end procedure
```

To invert the process and compute the DCT-III, we can perform the above steps in reverse. Suppose we already have $F_0, F_1, \ldots, F_{N-1}$. For the recurrence relation step, first let

$$R_0 = F_0, \quad R_{N/2} = 2F_{N-1}, \quad I_0 = 0, \quad I_{N/2} = 0.$$

Then for $n = 1, 2, \ldots, {}^N\!/_2 - 1$, let

$$R_n = R_{N-n} = \sin(n\pi/N)(F_{2n-1} - F_{2n+1}) + \cos(n\pi/N)F_{2n},$$

$$I_n = -I_{N-n} = -\cos(n\pi/N)(F_{2n-1} - F_{2n+1}) + \sin(n\pi/N)F_{2n}.$$

Now let $\vec{Y} = \vec{R} + i\vec{I}$, then take the inverse FFT of $\vec{Y}$ using `realifft` to get $\vec{y}$. We can use (13) and some algebra to show that, for $j = 0, 1, \ldots, {}^N\!/_2 - 1$,

$$\begin{aligned} y_j + y_{N-j-1} &= f_j + f_{N-j-1} \\ y_j - y_{N-j-1} &= 2\sin\left(\frac{\pi(j + {}^1\!/_2)}{N}\right)(f_j - f_{N-j-1}) \end{aligned}$$

or equivalently,

$$\begin{aligned} f_j &= \frac{y_j - y_{N-j-1}}{4\sin\left(\frac{\pi(j+1/2)}{N}\right)} + \frac{y_j + y_{N-j-1}}{2} = \frac{y_j - y_{N-j-1}}{4}\csc\left(\frac{\pi(j + {}^1\!/_2)}{N}\right) + \frac{y_j + y_{N-j-1}}{2} \\ f_{N-j-1} &= -\frac{y_j - y_{N-j-1}}{4\sin\left(\frac{\pi(j+1/2)}{N}\right)} + \frac{y_j + y_{N-j-1}}{2} = -\frac{y_j - y_{N-j-1}}{4}\csc\left(\frac{\pi(j + {}^1\!/_2)}{N}\right) + \frac{y_j + y_{N-j-1}}{2}. \end{aligned}$$

Note that it is not necessary to multiply $\vec{f}$ by $\frac{2}{N}$, because we obtained the inverse by reversing the steps of algorithm 8, rather than by manipulating definition 12.

Algorithm 9 Inverse discrete cosine transform using the FFT

procedure MY_IDCT($\vec{F}$)
  $N \leftarrow$ LENGTH($\vec{F}$)
  $\vec{R} \leftarrow \vec{0} \in \mathbb{R}^N$
  $\vec{I} \leftarrow \vec{0} \in \mathbb{R}^N$
  $R_0 \leftarrow F_0$
  $R_{N/2} \leftarrow 2F_{N-1}$
  for $n \leftarrow 1, 2, \ldots, {}^N\!/_2 - 1$ **do**
    $c \leftarrow \cos(n\pi/N)$
    $s \leftarrow \sin(n\pi/N)$
    $R_n \leftarrow s(F_{2n-1} - F_{2n+1}) + cF_{2n}$
    $R_{N-n} \leftarrow R_n$
    $I_n \leftarrow -c(F_{2n-1} - F_{2n+1}) + sF_{2n}$
    $I_{N-n} \leftarrow -I_n$
  end for
  $\vec{Y} \leftarrow \vec{R} + i\vec{I}$
  $\vec{y} \leftarrow$ REALIFFT($\vec{Y}$)
  for $j \leftarrow 0, 1, \ldots, {}^N\!/_2 - 1$ **do**
    $\alpha \leftarrow \frac{1}{4}(y_j - y_{N-j-1})\csc\left(\frac{\pi(j+1/2)}{N}\right)$
    $\beta \leftarrow \frac{y_j + y_{N-j-1}}{2}$
    $f_j \leftarrow \alpha + \beta$
    $f_{N-j-1} \leftarrow -\alpha + \beta$
  end for
return $\vec{f}$
end procedure

C.3 Multidimensional DST and DCT

Just like with the discrete Fourier transform in equation 5, we can define the discrete sine and cosine transforms of 2-dimensional data stored in a matrix. We can compute them in a similar way to algorithm 6, i.e. by taking the DST/DCT of each row, then of each column. As in the 1-dimensional case, the 2-dimensional DCT-II and DCT-III are inverses of each other.

D Applications

D.1 Solving the heat equation

The heat equation in one dimension is

$$\frac{\partial}{\partial t}u(x,t) = \alpha^2 \frac{\partial^2}{\partial x^2}u(x,t), \quad x \in (0,L), \quad t > 0. \tag{14}$$

The PDE (14) describes the distribution of heat in a uniform rod. The constant α^2 is known as the thermal diffusivity, and depends on the material from which the rod is made [6, Chapter 10.5].

In order to solve (14), we also require an initial condition, $u(x,0) = f(x)$, for all $x \in [0,L]$. We can also assume the boundary conditions $u(0,t) = u(L,t) = 0$.

The solution can be found analytically to be $u(x,t) = \sum_{n=1}^{\infty} c_n e^{-n^2\pi^2\alpha^2 t/L^2} \sin\frac{n\pi x}{L}$, where the constants c_n are given by $c_n = \frac{2}{L}\int_0^L f(x)\sin\frac{n\pi x}{L}\,\mathrm{d}x$ [6, Chapter 10.5]. For example, if $f(x) = \sin x$, $L = 2\pi$, and $\alpha^2 = 1$, then the solution is $u(x,t) = e^{-t}\sin x$.

Here is a way to solve the problem numerically using the Fast Fourier Transform. To simplify the problem slightly, suppose $\alpha^2 = 1$ and $L = 2\pi$, and use the initial condition $u(x,0) = \sin x$. Take the Fourier transform (with respect to x) of both sides of the equation:

$$\begin{aligned} \partial_t u(x,t) &= \partial_{xx} u(x,t) \\ \mathcal{F}_x\{\partial_t u(x,t)\} &= \mathcal{F}_x\{\partial_{xx} u(x,t)\} \\ \partial_t \hat{u}(k,t) &= (2\pi i k)^2 \hat{u}(k,t) = -4\pi^2 k^2 \hat{u}(k,t). \end{aligned}$$

Now for each independent value of k we have a first order ODE in t, which we can solve with Euler's method [6, Chapter 8.1] (or some other appropriate method).

First discretize the time interval: pick a step size h and the number of samples N. Then $t_j = jh$, $j = 0, 1, \ldots, N-1$. Then discretize the points in space: pick the number of samples M (since we will be performing a FFT, M should be a power of 2), then the space between points is $\Delta = 2\pi/M$. Then $x_j = j\Delta$, $j = 0, 1, \ldots, M-1$. The values of k are then $k_n = n/M\Delta$, $n = -N/2, \ldots, N/2 - 1$.

Approximate the initial values $\hat{u}(k,0) = \mathcal{F}_x\{u(x,0)\}(k) = \mathcal{F}_x\{\sin x\}(k)$ by taking the FFT of a vector of samples of $\sin x$ at the points x_j. The Fourier transform of $\sin x$ is actually given in terms of delta functions as $\mathcal{F}\{\sin x\}(k) = i\pi(\delta(k + \frac{1}{2\pi}) - \delta(k - \frac{1}{2\pi}))$. The FFT will approximate it as a function which is close to zero almost everywhere, apart from two "spikes" at $\pm\frac{1}{2\pi}$, as shown in figure 4.

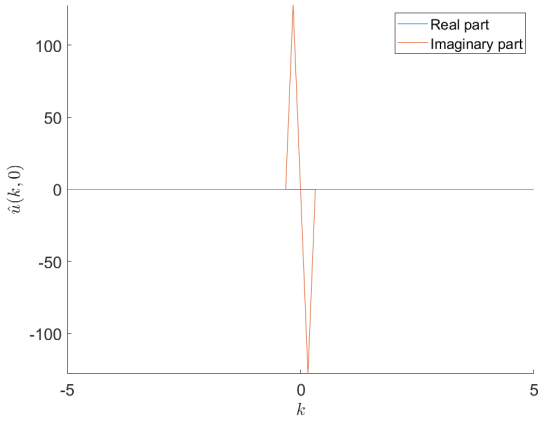


Figure 4: How the FFT approximates the Fourier transform of $\sin x$

Then Euler's method gives

$$\begin{aligned}\hat{u}(k, t_{j+1}) &\approx \hat{u}(k, t_j) + h\partial_t \hat{u}(k, t_j) \\ &= \hat{u}(k, t_j) - 4h\pi^2 k^2 \hat{u}(k, t_j) \\ &= (1 - 4h\pi^2 k^2)\hat{u}(k, t_j).\end{aligned}$$

Then, for each time t_j, take the inverse FFT: $u(x, t_j) = \mathcal{F}_x^{-1}\{\hat{u}(k, t_j)\}$.

Figure 5: Solution to the heat equation with initial condition $u(x, 0) = \sin x$ for $x \in (0, 2\pi)$, $t \in (0, 1)$

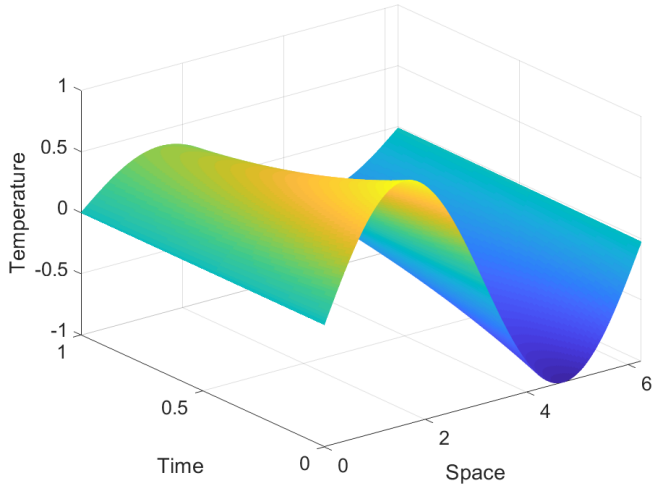
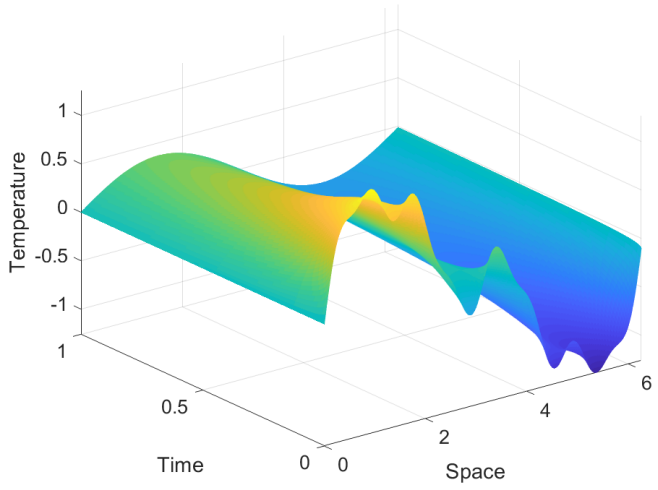


Figure 5 shows a surface plot of the solution obtained by following the steps above. The same process can be repeated for other initial conditions, such as sums of sine and cosine functions. Figure 6 shows the solution for one example.

Figure 6: Solution to the heat equation with initial condition $u(x, 0) = \sin x + \frac{1}{2}\sin 2x + \frac{1}{4}\sin 4x + \frac{1}{8}\sin 8x$ for $x \in (0, 2\pi)$, $t \in (0, 1)$



The idea of taking the Fourier transform and solving an ODE for each k with Euler's method could be applied to other PDEs.

D.2 JPEG compression

The JPEG image file format is one of the most commonly used file formats for storing photographs. The compression process involves the discrete cosine transform (11), and works by distorting some of the highest frequencies in the image which the human eye is less sensitive to.

Suppose we start with a grayscale image, which is represented as a matrix where each entry is an integer between 0 (black) and 255 (white). The encoding process is as follows:

1. Split up the image into blocks of 8×8 pixels.
2. Subtract 128 from each entry, so that the entries are now integers between -128 and 127.
3. Take the 2-dimensional discrete cosine transform of the block.
4. Divide the block elementwise by a constant matrix $\mathbf{Q}$ known as a quantization matrix, and then round each entry to the nearest integer.
5. The resulting matrix contains a lot of zeroes, the non-zero entries being concentrated in the upper-left corner. The reason why JPEGs take up so little space is that the long sequences of zeroes are easily compressed.

The quantization matrix used for JPEG is

$$\mathbf{Q} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

The decoding process (to reconstruct the image from the data stored) is essentially the same, but in reverse:

1. Reconstruct each quantized 8×8 block.
2. Multiply elementwise by $\mathbf{Q}$.
3. Take the 2-dimensional inverse DCT.
4. Round to the nearest integer, and add 128 so that the entries are between 0 and 255.

Let's demonstrate this on one 8×8 block, shown in figure 7. As a matrix this is

$$\begin{bmatrix} 201 & 198 & 196 & 195 & 184 & 183 & 185 & 180 \\ 206 & 205 & 204 & 203 & 199 & 197 & 197 & 195 \\ 206 & 207 & 205 & 204 & 204 & 203 & 204 & 204 \\ 209 & 208 & 193 & 201 & 202 & 202 & 203 & 203 \\ 212 & 213 & 207 & 210 & 201 & 185 & 185 & 180 \\ 224 & 227 & 226 & 224 & 220 & 217 & 213 & 200 \\ 230 & 232 & 230 & 230 & 229 & 229 & 229 & 232 \\ 230 & 230 & 230 & 229 & 218 & 225 & 229 & 229 \end{bmatrix}.$$

Subtract 128 from each entry, take the 2D DCT-II, divide each element by the corresponding entry of $\mathbf{Q}$, and finally round to the nearest integer:

$$\begin{bmatrix} 325 & 17 & 0 & 0 & 0 & 1 & -1 & 0 \\ -45 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & -3 & 1 & -1 & 0 & 0 & 0 & 0 \\ -8 & 6 & -2 & 0 & 0 & 0 & 0 & 0 \\ -11 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Figure 7: An 8×8 block of an image before compression

Figure 8: The same 8×8 block after compression

This matrix contains mostly zeroes. Reversing the process gives the matrix

$$\begin{bmatrix} 201 & 200 & 195 & 193 & 185 & 181 & 185 & 182 \\ 204 & 206 & 206 & 208 & 203 & 196 & 196 & 189 \\ 205 & 204 & 201 & 204 & 204 & 204 & 209 & 205 \\ 213 & 208 & 201 & 200 & 199 & 200 & 206 & 203 \\ 213 & 211 & 206 & 206 & 199 & 190 & 186 & 176 \\ 226 & 227 & 226 & 228 & 222 & 214 & 211 & 202 \\ 229 & 229 & 228 & 230 & 228 & 227 & 234 & 232 \\ 230 & 230 & 227 & 228 & 223 & 223 & 230 & 229 \end{bmatrix}$$

which is similar to what we started with, as shown in figure 8.

References

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 2007 (3rd edition).

[2] Fourier Transform – from Wolfram MathWorld `http://mathworld.wolfram.com/FourierTransform.html` Accessed August 6, 2019.

[3] Delta Function – from Wolfram MathWorld `http://mathworld.wolfram.com/DeltaFunction.html` Accessed August 6, 2019.

[4] Implementing FFTs in Practice `https://cnx.org/contents/ulXtQbN7@15/Implementing-FFTs-in-Practice` Accessed August 6, 2019.

[5] Free small FFT in multiple languages `https://www.nayuki.io/page/free-small-fft-in-multiple-languages` Accessed August 6, 2019.

[6] William E. Boyce and Richard C. DiPrima. *Elementary Differential Equations and Boundary Value Problems.* Wiley, 2012 (10th edition).

[7] Image Compression and the Discrete Cosine Transform `https://www.math.cuhk.edu.hk/ lmlui/dct.pdf` Accessed August 6, 2019.