# MATH 603 - Final Assignment

Aiden Taylor - 30092686

December 5, 2024

# Contents

# 1 Problems

1. Write a computer program to implement the Fast Fourier Transform (FFT).

2. Using the FFT, write a computer program to solve numerically the initial-value problem (IVP) for the heat equation,

$$\begin{cases} u_t = u_{xx} & (t,x) \in [0,T] \times [0,L] \\ u(0,x) = u_0(x) & x \in [0,L] \end{cases}.$$

## 2 Problem 1

To implement the FFT, we should first revist the Continuous Fourier Transform of some function $f(x)$,

$$F(\omega) = \hat{f}(\omega) = \mathcal{F}[f](\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(x)e^{-i\omega x}dx,$$

where the function can be recovered as

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega)e^{i\omega x}d\omega.$$

Now, consider discretizing both the original and frequency domains into $n$ equally spaced points, where

$$\begin{cases} \omega_m = 2\pi m/n, & m = 0, 1, \ldots, n-1, \\ x_k = x_0 + k\Delta x, & k = 0, 1, \ldots, n-1, \end{cases} \tag{1}$$

given that $x_0 = 0$ and $\Delta x = L/(n-1)$. Then, if we let $f_k = f(x_k)$ for $k = 0, 1, \ldots, n-1$, we can define the Discrete Fourier Transform (DFT) as

$$f_m^{\#} = \sum_{k=0}^{n-1} f_k e^{-i\omega_m k}, \quad m = 0, 1, \ldots, n-1,$$

where the discretization of the function can be recovered as

$$f_k = \frac{1}{n}\sum_{m=0}^{n-1} f_k^{\#} e^{i\omega_m k}, \quad k = 0, 1, \ldots, n-1,$$

which we call the Inverse DFT (IDFT). Letting $\xi = e^{i2\pi/n}$, we can instead represent the DFT and IDFT respectively as the following two matrix-vector multiplications,

$$\begin{bmatrix} f_0^{\#} \\ f_1^{\#} \\ f_2^{\#} \\ \vdots \\ f_{n-1}^{\#} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \xi^{-1} & \xi^{-2} & \ldots & \xi^{-(n-1)} \\ 1 & \xi^{-2} & \xi^{-4} & \ldots & \xi^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \xi^{-(n-1)} & \xi^{-2(n-1)} & \ldots & \xi^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{bmatrix},$$

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{bmatrix} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \xi^{1} & \xi^{2} & \ldots & \xi^{(n-1)} \\ 1 & \xi^{2} & \xi^{4} & \ldots & \xi^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \xi^{(n-1)} & \xi^{2(n-1)} & \ldots & \xi^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} f_0^{\#} \\ f_1^{\#} \\ f_2^{\#} \\ \vdots \\ f_{n-1}^{\#} \end{bmatrix}.$$

Since both the DFT and IDFT are just $n \times n$ systems, it follows that they both have a computational complexity of $\mathcal{O}(n^2)$. From here, the FFT is derived from noticing redundancies in the computation of the DFT, specifically, from noticing that $\xi$ is periodic and that certain powers of $\xi$ are equal. To illustrate this claim, consider the system of equations generated by the DFT when $n = 4$,

$$\begin{cases} f_0^{\#} &= f_0\xi^0 + f_1\xi^0 + f_2\xi^0 + f_3\xi^0 \\ f_1^{\#} &= f_0\xi^0 + f_1\xi^{-1} + f_2\xi^{-2} + f_3\xi^{-3} \\ f_2^{\#} &= f_0\xi^0 + f_1\xi^{-2} + f_2\xi^{-4} + f_3\xi^{-6} \\ f_3^{\#} &= f_0\xi^0 + f_1\xi^{-3} + f_2\xi^{-6} + f_3\xi^{-9}. \end{cases}$$

If we notice that $\xi^0 = \xi^{-4} = 1$, $\xi^{-2} = \xi^{-6} = -1$, $\xi^{-1} = \xi^{-9} = -i$, and $\xi^{-3} = i$, then we can simplify the above system of equations to,

$$\begin{cases} f_0^{\#} &= (f_0 + f_1) + \xi^0(f_2 + f_3) \\ f_1^{\#} &= (f_0 - f_1) + \xi^{-1}(f_2 - f_3) \\ f_2^{\#} &= (f_0 + f_1) + \xi^{-2}(f_2 + f_3) \\ f_3^{\#} &= (f_0 - f_1) + \xi^{-3}(f_2 - f_3), \end{cases}$$

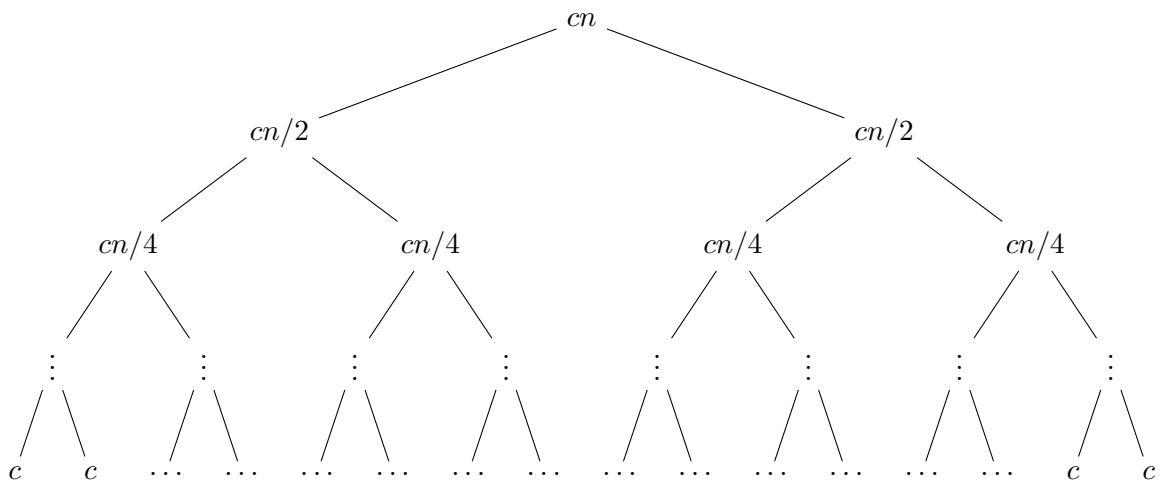
which has reduced the number of computations from 16 multiplications and 12 additions to 4 multiplications and 12 additions (without doing any caching or precomputing). This idea can be generalized when $n$ is some positive integer power of 2, i.e. $n = 2^\ell$ where $\ell \in \mathbb{Z}^+$, which instead allows us to represent the DFT as

$$f_m^{\#} = \sum_{k=0}^{n-1} f_k \xi^{-mk} = \sum_{k=0}^{\frac{n}{2}-1} f_{2k} \xi^{-m(2k)} + \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1} \xi^{-m(2k+1)}, \tag{2}$$

for $m = 0, 1 \ldots, n-1$, where we are essentially just breaking up the summation into its even and odd indexed summations. If we also notice that

$$\begin{aligned} f_m^{\#} &= \sum_{k=0}^{\frac{n}{2}-1} f_{2k} \xi^{-m(2k)} + \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1} \xi^{-m(2k+1)} \\ &= \sum_{k=0}^{\frac{n}{2}-1} f_{2k} \xi^{-2mk} + \xi^{-m} \sum_{k=0}^{\frac{n}{2}-1} f_{2k+1} \xi^{-2mk} \end{aligned} \tag{3}$$

for $m = 0, 1, \ldots, n-1$, then we can use the idea from (2) on each of the two individual summations in (3). Continuing to apply this process recursively until each of the individual summations has just one term, reduces the computational complexity of the DFT from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2 n)$, giving us the FFT. This reduction in complexity can be visualized by the corresponding binary tree generated by the recursive process,




where the value at each node in the tree represents an order of the number of floating point operations required to solve that nodes sub-problem, given that $c$ is some arbitrary

constant. The summation of the values of each node at any particular level of the binary tree should equal $cn$, so, to compute the computational complexity of this recursive process, we just need to compute the height of this binary tree. Now, at the $i^{th}$ level of the binary tree, the values of the individual nodes at said level represent the $cn/2^i$ floating point operations required to solve that nodes specific sub-problem, and at the bottom level we know that we should only need

$$cn/2^i = \mathcal{O}(1) = k \tag{4}$$

floating point operations for each node, where $k$ is also some arbitrary constant value. So, if we take the logarithm of both sides of (4), we get

$$\begin{aligned}
&\log_2 (cn/2^i) = \log_2 k \\
&\Rightarrow \log_2 cn - \log_2 2^i = \log_2 k \\
&\Rightarrow \log_2 n + \log_2 c - i \log_2 2 = \log_2 k \\
&\Rightarrow \log_2 n + \log_2 c - \log_2 k = i
\end{aligned}$$

which implies that the height of the binary tree is $\mathcal{O}(\log_2 n)$. Then finally, since we're doing $cn$ floating point operations $\mathcal{O}(\log_2 n)$ times, we get a computational complexity of $\mathcal{O}(n \log_2 n)$. The following is an algorithm for the FFT which implements the recursive process that reduces the computational complexity of the DFT:

**Algorithm 1** Fast Fourier Transform

**Procedure** FFT($f$, $f^{\#}$, $n$, $\xi$)
**if** $n \leftarrow 1$ **then**
  $f^{\#}[0] \leftarrow f[0]$
**else**
  $f_e[k] \leftarrow$ empty array of size $\frac{n}{2}$
  $f_o[k] \leftarrow$ empty array of size $\frac{n}{2}$
  **for** $k$ from 0 to $\frac{n}{2} - 1$ **do**
    $f_e[k] \leftarrow f[2k]$
    $f_o[k] \leftarrow f[2k+1]$
  **end for**
  $f_e^{\#}[k] \leftarrow$ empty array of size $\frac{n}{2}$
  $f_o^{\#}[k] \leftarrow$ empty array of size $\frac{n}{2}$
  FFT($f_e$, $f_e^{\#}$, $\frac{n}{2}$, $\xi^2$)
  FFT($f_o$, $f_o^{\#}$, $\frac{n}{2}$, $\xi^2$)
  **for** $k$ from 0 to $n-1$ **do**
    $f[k] \leftarrow f_e^{\#}[k \mod \frac{n}{2}] + \xi^k f_o^{\#}[k \mod \frac{n}{2}]$
  **end for**
**end if**
**End Procedure**

## 3 Problem 2

Writing out the problem again, our goal is to solve numerically the IVP,

$$\begin{cases} u_t = u_{xx} & (t,x) \in [0,T] \times [0,L] \\ u(0,x) = u_0(x) & x \in [0,L] \end{cases}$$

using the FFT. However, before we discretize any of the domains, consider taking the Fourier Transform of the unknown function, $u$, in the spatial domain,

$$u(t,x) \overset{\mathcal{F}}{\Longrightarrow} \hat{u}(t,\omega),$$

and the Fourier Transforms of $u_t$ and $u_{xx}$ in the spatial domain as well,

$$u_t \overset{\mathcal{F}}{\Longrightarrow} \frac{d}{dt}\hat{u}(t,\omega) \quad \text{and} \quad u_{xx} \overset{\mathcal{F}}{\Longrightarrow} -\omega^2 \hat{u}(t,\omega). \tag{5}$$

Then rewriting the heat equation in terms of the identities in (5) results in

$$\frac{d}{dt}\hat{u}(t,\omega) = -\omega^2 \hat{u}(t,\omega),$$

which is just an ordinary differential equation (ODE), specifically a decay equation, which has the well known solution

$$\hat{u}(t,\omega) = \hat{u_0} e^{-\omega^2 t},$$

where $\hat{u_0}$ is the Fourier Transform of the intital-value function $u_0$. Now, to solve this IVP numerically, we use the discretization of the frequency and spatial domains described in (1), which gives us a system of ODEs,

$$\begin{bmatrix} \frac{d}{dt}\hat{u}(t,\omega_0) \\ \vdots \\ \frac{d}{dt}\hat{u}(t,\omega_{n-1}) \end{bmatrix} = \begin{bmatrix} -\omega_0^2 \hat{u}(t,\omega_0) \\ \vdots \\ -\omega_{n-1}^2 \hat{u}(t,\omega_{n-1}) \end{bmatrix},$$

which in turn has a system of solutions,

$$\begin{bmatrix} \hat{u}(t,\omega_0) \\ \vdots \\ \hat{u}(t,\omega_{n-1}) \end{bmatrix} = \begin{bmatrix} \hat{u_0}(t,\omega_0) e^{-\omega_0^2 t} \\ \vdots \\ \hat{u_0}(t,\omega_{n-1}) e^{-\omega_{n-1}^2 t} \end{bmatrix},$$

for each step in time, $t$. This gives us a vector of Fourier coefficients at each time step, $t$, and performing the Inverse FFT (IFFT) on each of these Fourier coefficient vectors recovers the approximation of the unknown function, $u$, at each time step, $t$, as desired. To illustrate this numerical process I wrote a Matlab program that implements the following IVP,

$$\begin{cases} u_t = u_{xx} & (t,x) \in [0,100] \times [0,1] \\ u(0,x) = u_0(x) & x \in [0,1] \end{cases}$$

where

$$u_0 = \begin{cases} 1 & x \in [0.3725, 0.6235] \\ 0 & \text{otherwise} \end{cases}.$$

Now, discretizing our frequency and spatial domains into $n = 256$ equally spaced points will result in the following three plots generated by the Matlab program, which concludes my final assignment:

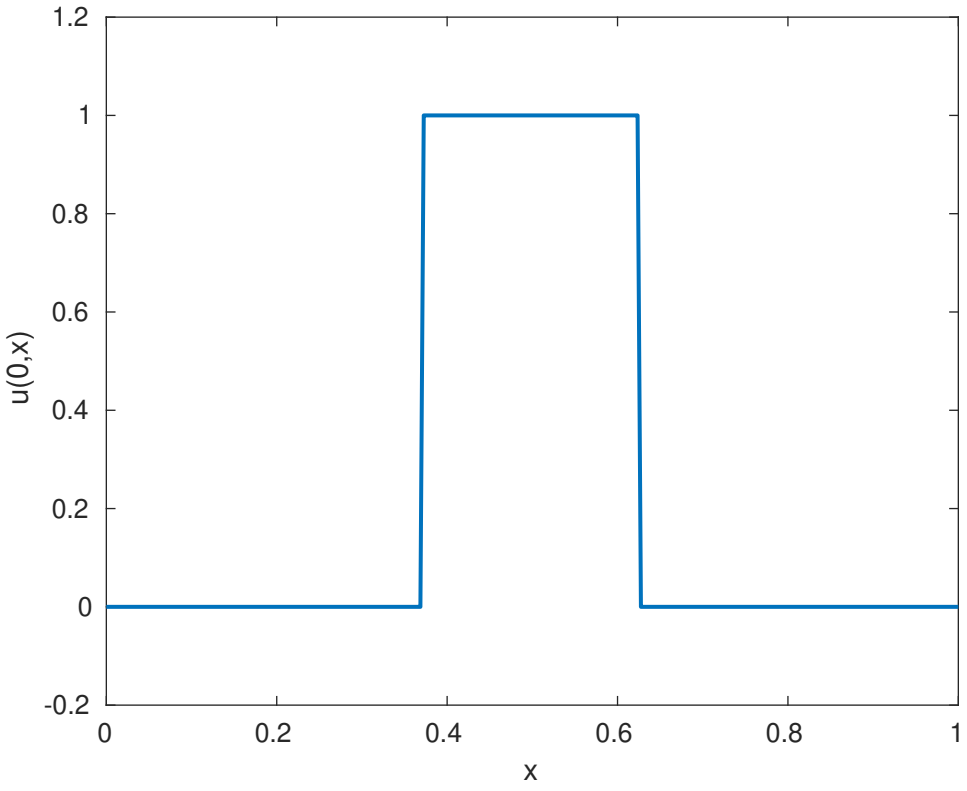



Figure 1: Plot of $u_0(x)$.

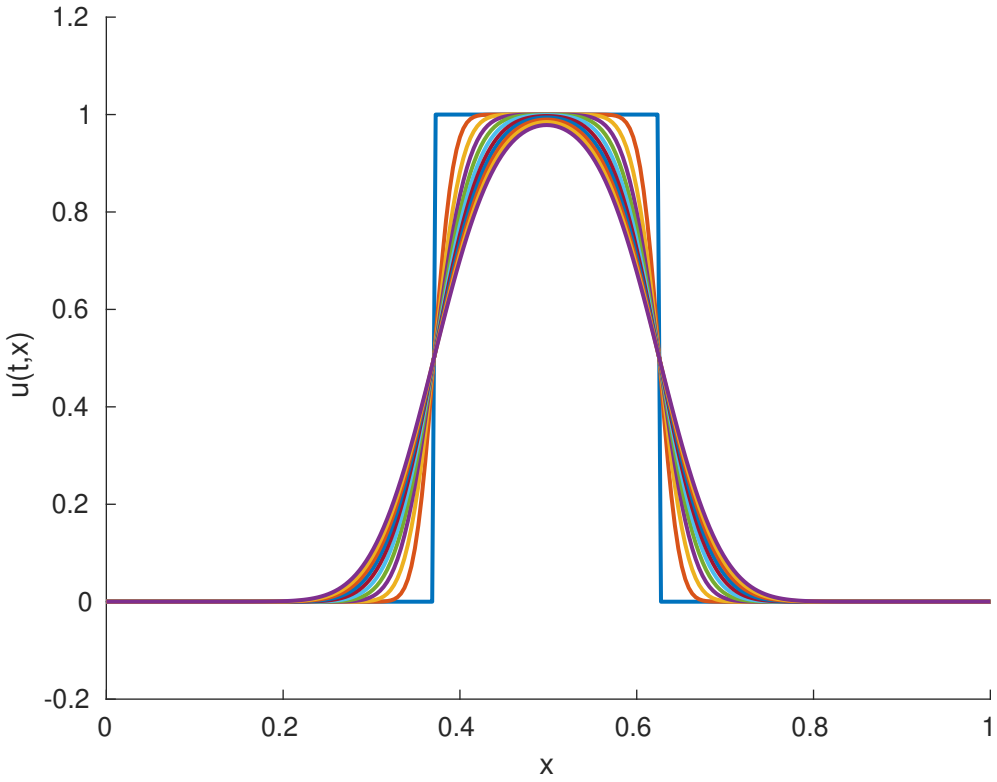



Figure 2: 2-dimensional plot of the numerical solution to the IVP.

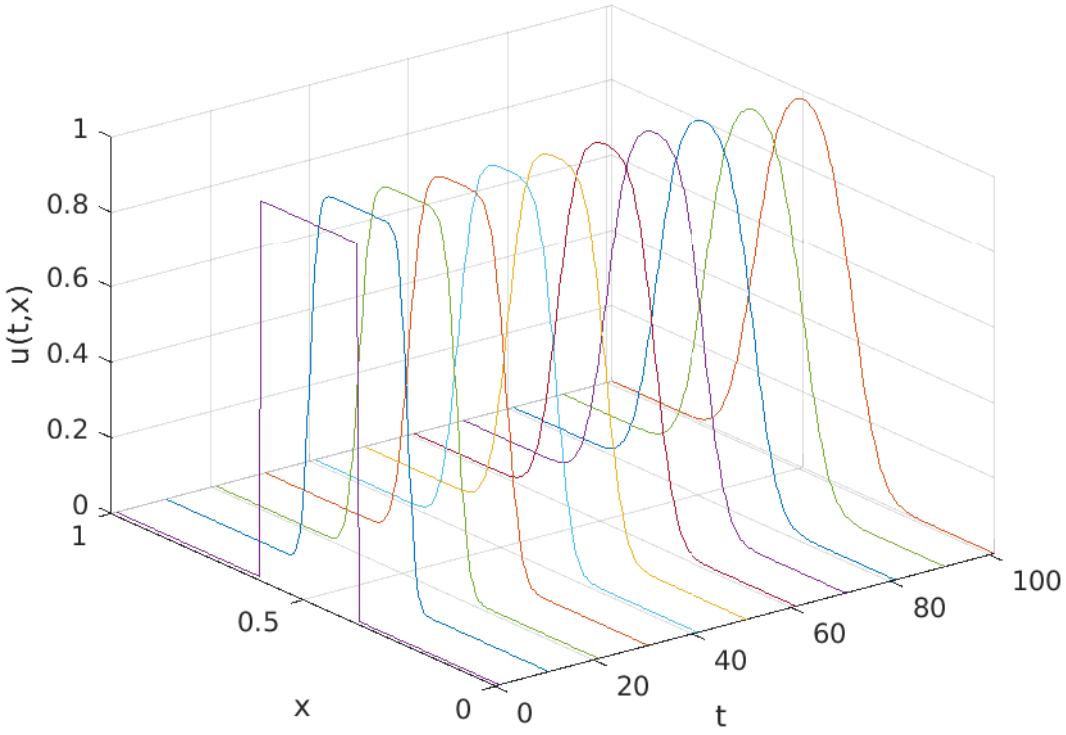



Figure 3: 3-dimensional plot of the numerical solution to the IVP.