university of groningen

faculty of mathematics
and natural sciences

The Fourier Transform on Finite Abelian Groups

Bachelor Project Mathematics

July 2015

Student: D.R.G. Hulzebos

First supervisor: J. Top

Second supervisor: A. Sterk

Abstract

In this research the Fourier transform on finite abelian groups is studied. To define this transform, characters are introduced. After the Fourier transform has been defined, an algorithm that will improve the computation time needed to do the Fourier transform is recalled. After this two applications of the Fourier transform will be discussed. In these applications the Fourier transform will be used to multiply huge integers in a faster way and to put watermarks on music.

Contents

1 Introduction

A well known concept in mathematics and physics is the Fourier transform. This is mostly known as an integral or infinite series which rewrites a function in terms of sines and cosines. However, this is only one case of a Fourier transform. Besides this case, there are many other cases with different Fourier transforms. We will be looking at one of those other cases, which is the Fourier transform on finite abelian groups. We will be looking at this Fourier transform, since it has many applications, from which we will consider two.

As the name suggests, we are dealing with finite abelian groups. As one might remember, these are commutative groups with finitely many elements. To define the Fourier transform on finite abelian groups, we will first need to introduce characters. Once we know what characters are, we will look at a few properties of them, which we will use to define the Fourier transform on finite abelian groups.
Once we have found the Fourier transform on finite abelian groups, we will look at a method which allows us to do this Fourier transform faster. This method is known as the Fast Fourier Transform. When we have this method, we will take a look at two applications of this Fourier transform. The first application will be multiplying huge integers in a faster way. The second one will be putting watermarks on pieces of music. In the end we will have a conclusion where we will summarize everything that we have done.

2 The Fourier transform on finite abelian groups

In this section we will introduce the Fourier transform on finite abelian groups. Therefore we first need to introduce the so-called characters. This will be done in the same fashion as done in for example [Bab02], [Goo] and [Con]. After this has been done, we will define an inner product, which is also needed to define the Fourier transform. When we have these things, we will define the Fourier transform on finite abelian groups.

2.1 Characters

In this part we will introduce characters, and we will look at some useful properties of the characters. These will later on be used to define the Fourier transform on finite abelian groups.

From now on, we let G be a nontrivial finite abelian group of order n with addition as the group law. This will be the case throughout the whole text. Now we will define characters as follows.

Definition 2.1. A *character* of the group G is a homomorphism $\chi : G \to \mathbb{C}^*$, which means that

$$\chi(a+b) = \chi(a)\chi(b), \text{ for } a, b \in G. \tag{2.1}$$

The *trivial character* is defined as

$$\chi_0(a) = 1, \text{ for } a \in G. \tag{2.2}$$

Since G is a finite abelian group of order n, and χ is a homomorphism, we have that

$$\chi(a)^n = \chi(na) = \chi(0) = 1 \text{ for any } a \in G. \tag{2.3}$$

Thus the values of χ are n^{th} roots of unity. In particular we have that

$$\chi(-a) = \chi(a)^{-1} = \overline{\chi(a)}. \tag{2.4}$$

Here the bar denotes the complex conjugation.

Note that if we have found a character χ for which $\chi(a)^n = 1$, such that $\chi(a)^j \neq 1$ for $j = 1, \ldots, n-1$, then all the characters are given by χ^k for $k = 1, \ldots, n$. This is because we still have that $\chi^k(a+b) = (\chi(a+b))^k = (\chi(a)\chi(b))^k = \chi^k(a)\chi^k(b)$. There are not any more characters since these are n^{th} roots of unity, and therefore there are only n of them.

Definition 2.2. the *dual group* of G, denoted by $\widehat{G}$, is the set of all characters of G.

This dual group turns out to be an abelian group with a group operation defined by

$$\chi\psi : G \to \mathbb{C}^*, \qquad a \mapsto \chi(a)\psi(a) \qquad \text{for } \chi, \psi \in \widehat{G} \text{ and } a \in G.$$

To see that this is indeed a group, note that we can take the trivial character as the identity element, and that the inverse element of a character is its conjugate character. The group is abelian since multiplication in $\mathbb{C}^*$ is commutative.
Another thing is that this group operation defines a homomorphism. To see that, we take $a, b \in G$ and $\chi, \psi \in \widehat{G}$. Then

$$\begin{aligned}(\chi\psi)(a+b) &= \chi(a+b)\psi(a+b) \\ &= \chi(a)\chi(b)\psi(a)\psi(b) \\ &= (\chi\psi)(a)(\chi\psi)(b),\end{aligned}$$

and hence it is a homomorphism.

Now we have an idea of what characters are. Characters that are much used are the characters of the group $\mathbb{Z}/n\mathbb{Z}$. The reason for this is that for any nontrivial finite abelian group G it holds that $G \cong \mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$, with $n_i \in \mathbb{Z}, i = 1, 2, \ldots, k$ and $1 < n_1|n_2|\ldots|n_k$, as one might remember from group theory. Therefore we will have a special look at these characters in the next proposition.

Proposition 2.3. *Let ω be the primitive n^{th} root of unity, i.e. $\omega = e^{\frac{2\pi i}{n}}$. Then the map $\chi_j : \mathbb{Z}/n\mathbb{Z} \to \mathbb{C}^*$ defined by $\chi_j(a) := \omega^{ja}$ is a character of $\mathbb{Z}/n\mathbb{Z}$ for every $j \in \mathbb{Z}$. Furthermore*

(a) $\chi_j = \chi_k$ *if and only if* $j \equiv k \mod n$,

(b) $\chi_j = \chi_1^j$,

(c) $\widehat{\mathbb{Z}/n\mathbb{Z}} = \{\chi_0, \ldots, \chi_{n-1}\}$,

(d) $\widehat{\mathbb{Z}/n\mathbb{Z}} \cong \mathbb{Z}/n\mathbb{Z}$.

Proof. First of all we notice that $\chi_j(a+b) = \omega^{j(a+b)} = \omega^{ja}\omega^{jb}$, and thus this indeed defines a character of $\mathbb{Z}/n\mathbb{Z}$ for every $j \in \mathbb{Z}$.
Now let $\chi_j = \chi_k$. Then $\omega^{ja} = \omega^{ka}$ for all $a \in \mathbb{Z}/n\mathbb{Z}$. This is possible if and only if $\omega^j = \omega^k$, and this holds if and only if $j \equiv k \mod n$, since ω is a primitive n^{th} root of unity. This proves part (a).
We can see that $\chi_j(a) = \omega^{ja} = \left(\omega^{1\cdot a}\right)^j = \chi_1^j(a)$, which proves part (b).

For part (c) we notice that $\widehat{\mathbb{Z}/n\mathbb{Z}} \supset \{\chi_0, \ldots, \chi_{n-1}\}$. Now let χ be an arbitrary character of $\mathbb{Z}/n\mathbb{Z}$. Then $\chi(1) = \omega^j$, for some $j, 0 \leq j \leq n-1$, since it is an nth root of unity. It follows that $\chi = \chi_j$, and thus $\widehat{\mathbb{Z}/n\mathbb{Z}} \subset \{\chi_0, \ldots, \chi_{n-1}\}$. Therefore $\widehat{\mathbb{Z}/n\mathbb{Z}} = \{\chi_0, \ldots, \chi_{n-1}\}$. Hence $\widehat{\mathbb{Z}/n\mathbb{Z}} \cong \mathbb{Z}/n\mathbb{Z}$. □

Now we know what the characters of the group $\mathbb{Z}/n\mathbb{Z}$ are, and we know that G can be written as a combination of these groups. Now we will try to say something about the dual group of this group G. Therefore we state and prove the following lemma and theorem.

Lemma 2.4. *If A and B are finite abelian groups, then there exists an isomorphism $\widehat{A \times B} \cong \widehat{A} \times \widehat{B}$.*

Proof. Let $\chi \in \widehat{A \times B}$. Let χ_A and χ_B be the restrictions to A and B respectively by setting $\chi_A(a) = \chi(a, 1)$ and $\chi_B(b) = \chi(1, b)$. Then

$$\chi(a, b) = \chi((a, 1)(1, b)) = \chi(a, 1)\chi(1, b) = \chi_A(a)\chi_B(b).$$

Now define the map

$$f : \widehat{A \times B} \to \widehat{A} \times \widehat{B}$$

by sending χ to (χ_A, χ_B). First we see that this is a homomorphism, by noticing that

$$f(\chi\psi) = ((\chi\psi)_A, (\chi\psi_B)) = (\chi_A\psi_A, \chi_B\psi_B) = (\chi_A, \chi_B)(\psi_A, \psi_B) = f(\chi)f(\psi).$$

The kernel is given by all characters $\chi \in \widehat{A \times B}$ for which $(\chi_A, \chi_B) = (1, 1)$. Since $\chi(a, b) = \chi_A(a)\chi_B(b)$, it follows that the kernel is only the trivial character $\chi = 1$. Therefore the map f is injective.
For surjectivity, let $(\chi_A, \chi_B) \in \widehat{A} \times \widehat{B}$. Now let $\chi(a, b) = \chi_A(a)\chi_B(b) \in \widehat{A \times B}$. Then $f(\chi) = (\chi_A, \chi_B)$, thus the map f is surjective, and hence we have an isomorphism $\widehat{A \times B} \cong \widehat{A} \times \widehat{B}$. □

Note that the result from lemma 2.4 can be extended in such a way that the dual group of $A_1 \times \ldots \times A_k$ is isomorphic to $\widehat{A_1} \times \ldots \times \widehat{A_k}$. This will be used to prove the following theorem.

Theorem 2.5. $G \cong \widehat{G}$.

Proof. Recall that we can write any finite abelian group G as
$G \cong \mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$, with $n_i \in \mathbb{Z}, i = 1, 2, \ldots, k$ and $1 < n_1|n_2|\ldots|n_k$. Using Lemma 2.4 and Proposition 2.3(d) we find that $\widehat{G} \cong \widehat{\mathbb{Z}/n_1\mathbb{Z}} \times \ldots \times \widehat{\mathbb{Z}/n_k\mathbb{Z}} \cong \mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z} = G$. □

Now we know that G is isomorphic to its dual group. But to define the Fourier transform, we need to know a bit more properties of the characters. Therefore we state the following propositions.

Proposition 2.6. *For any nonprincipal character $\chi \in \widehat{G}$ it holds that*

$$\sum_{a \in G} \chi(a) = 0. \tag{2.5}$$

Proof. Let $b \in G$ such that $\chi(b) \neq 1$ and let $S = \sum_{a \in G} \chi(a)$. Then

$$\chi(b)S = \sum_{a \in G} \chi(b)\chi(a) = \sum_{a \in G} \chi(b + a) = S.$$

Because we find that $\chi(b)S = S$, and since $\chi(b) \neq 1$, we have that $S = 0$. □

Proposition 2.7. *Let χ and ψ be two characters of G. Then*

$$\sum_{a\in G} \overline{\chi(a)}\psi(a) = \begin{cases} n & \text{if } \chi = \psi \\ 0 & \text{if } \chi \neq \psi. \end{cases} \tag{2.6}$$

Proof. The first case, where $\chi = \psi$, follows from equation 2.4 and the fact that G is a group of order n:

$$\sum_{a\in G} \overline{\chi(a)}\psi(a) = \sum_{a\in G} \overline{\psi(a)}\psi(a) = \sum_{a\in G} \psi(a)^{-1}\psi(a) = \sum_{a\in G} 1 = n.$$

The second case, where $\chi \neq \psi$, follows from Proposition 2.6 and by using the fact that $\overline{\chi}\psi$ is a nontrivial character:

$$\sum_{a\in G} \overline{\chi(a)}\psi(a) = \sum_{a\in G} (\overline{\chi}\psi)(a) = 0.$$

□

2.2 The Fourier transform on finite abelian groups

Now we know what characters are, and now we know some properties of them, we are almost ready to define the Fourier transform on finite abelian groups. However, to introduce this Fourier transform, we need to define one more thing, which is an inner product. This is done in the following way:

Definition 2.8. Let $\mathbb{C}^G$ denote the space of all functions $f : G \to \mathbb{C}$. Then $\mathbb{C}^G$ is an n-dimensional linear space over $\mathbb{C}$. Over this this space an inner product is defined by:

$$(f, g) = \frac{1}{n} \sum_{a\in G} \overline{f(a)}g(a) \qquad \text{for } f, g \in \mathbb{C}^G. \tag{2.7}$$

Theorem 2.9. *$\widehat{G}$ forms an orthonormal basis in $\mathbb{C}^G$.*

Proof. Using Proposition 2.7 we find that

$$(\chi, \psi) = \frac{1}{n} \sum_{a\in G} \overline{\chi(a)}\psi(a) = \begin{cases} 1 & \text{if } \chi = \psi \\ 0 & \text{if } \chi \neq \psi, \end{cases}$$

where $\chi, \psi \in \widehat{G}$. Since we have proven in Theorem 2.5 that $G \cong \widehat{G}$, it follows that $|\widehat{G}| = n = \dim(\mathbb{C}^G)$, which shows that we have completeness. □

This is a very useful theorem, and it allows us to state and proof the following corollary.

Corollary 2.10. *Any function $f \in \mathbb{C}^G$ can be written as a linear combination of characters:*

$$f = \sum_{\chi\in\widehat{G}} c_\chi \chi. \tag{2.8}$$

This linear combination is often called a trigonometric sum *because f is expressed as a combination of nth roots of unity. The coefficients c_χ are called the* Fourier coefficients *and are given by*

$$c_\chi = (\chi, f). \tag{2.9}$$

Proof. Since we have shown in Theorem 2.9 that $\widehat{G}$ forms an orthonormal basis in $\mathbb{C}^G$, it follows that Equation 2.8 holds. Using the orthonormality, we find that

$$(\psi, f) = (\psi, \sum_{\chi \in \widehat{G}} c_\chi \chi) = \sum_{\chi \in \widehat{G}} c_\chi (\psi, \chi) = c_\psi,$$

and thus Equation 2.9 indeed gives the Fourier coefficients. □

Now we have all the ingredients we need to define the Fourier transform.

Definition 2.11. The *Fourier transform* of a function $f : G \to C$ is a function $\widehat{f} : \widehat{G} \to \mathbb{C}$ given by

$$\widehat{f}(\chi) = nc_{\overline{\chi}} = \sum_{a \in G} \chi(a) f(a), \qquad \text{where } \chi \in \widehat{G}. \tag{2.10}$$

To find the inverse of the map $f \mapsto \widehat{f}$, we note the following. Making use of Equations 2.8 and 2.10 one observes:

$$f = \sum_{\chi \in \widehat{G}} c_\chi \chi = \sum_{\chi \in \widehat{G}} \frac{1}{n} \widehat{f}(\overline{\chi}) \chi = \sum_{\chi \in \widehat{G}} \frac{1}{n} \widehat{f}(\chi) \overline{\chi}.$$

This tells us that we can define the inverse Fourier in the following way.

Definition 2.12. The *inverse Fourier transform* of the map $f \mapsto \widehat{f}$ is given by

$$f(a) = \frac{1}{n} \sum_{\chi \in \widehat{G}} \widehat{f}(\chi) \chi(-a) \qquad \text{for } a \in G. \tag{2.11}$$

We will now describe what the Fourier transform does to a basis of $\mathbb{C}^G$. The basis of $\mathbb{C}^G$ we will use is given by $\left\{ \mathbb{1}_{\{b\}} : b \in G \right\}$. If we apply the Fourier transform to this we find that

$$\widehat{\mathbb{1}_{\{b\}}}(\chi) = \sum_{a \in G} \chi(a) \mathbb{1}_{\{b\}}(a) = \chi(b).$$

From this we see that $\widehat{\mathbb{1}_{\{b\}}}$ equals the character of $\widehat{G}$ given by $\chi \mapsto \chi(b)$. So we conclude that our basis is transformed to the basis given by all the characters: $\widehat{G} \to \mathbb{C}^*$.

To make clear how this Fourier transform works, we will look at two examples. In the first example we will consider the Fourier transform of the integers modulo n, where $n \in \mathbb{Z}$. In the second example we will look at an example where we compute the Fourier transform of a vector.

Example 2.13. In this example we will look at the case where $G = \mathbb{Z}/n\mathbb{Z}$. As we have seen in Proposition 2.3, the characters of this group are given by ω^j, where $j \in \mathbb{Z}/n\mathbb{Z}$ and $\omega = e^{\frac{2\pi i}{n}}$. Now let $f \in \mathbb{C}^{\mathbb{Z}/n\mathbb{Z}}$. Then the Fourier transform of f is given by

$$\widehat{f}(\chi^j) = \sum_{a \in \mathbb{Z}/n\mathbb{Z}} f(a) \omega^{ja}. \tag{2.12}$$

We have seen in Theorem 2.5 that $\widehat{G} \cong G$. Therefore this Fourier transform is often written as

$$\widehat{f}(j) = \sum_{a \in \mathbb{Z}/n\mathbb{Z}} f(a) \omega^{ja}. \tag{2.13}$$

So instead of putting values of $\widehat{G}$ in the Fourier transform, there are values of G entered. This does not change anything to the Fourier transform itself.
The inverse Fourier transform of f is given by

$$f(a) = \frac{1}{n} \sum_{j \in \mathbb{Z}/n\mathbb{Z}} \widehat{f}(j)\omega^{-ja}. \tag{2.14}$$

We have seen how we can transform functions using the Fourier transform. It also turns out to be possible to transform vectors using the Fourier transform. This is done by seeing the vector as a function. How this works is illustrated in the following example.

Example 2.14. In this example we consider the vector $v = (0, 9, 8, 3)$. We can look at this as a function $f : \mathbb{Z}/4\mathbb{Z} \to \mathbb{C}$, which sends $a \in \mathbb{Z}/4\mathbb{Z}$ to the ath element of the vector v. Then we can use formula 2.13 from example 2.13 to determine the Fourier transform. In this case we have that

$$\omega = e^{\frac{2\pi i}{4}} = i.$$

Then we can compute the Fourier transform as follows:

$$\begin{aligned}
\widehat{f}(0) &= \sum_{a \in \mathbb{Z}/4\mathbb{Z}} f(a)\omega^{0 \cdot a} = 0 \cdot 1 && +9 \cdot 1 && +8 \cdot 1 && +3 \cdot 1 && = 20, \\
\widehat{f}(1) &= \sum_{a \in \mathbb{Z}/4\mathbb{Z}} f(a)\omega^{1 \cdot a} = 0 \cdot 1 && +9 \cdot i && +8 \cdot (-1) && +3 \cdot (-i) && = -8 + 6i, \\
\widehat{f}(2) &= \sum_{a \in \mathbb{Z}/4\mathbb{Z}} f(a)\omega^{1 \cdot a} = 0 \cdot 1 && +9 \cdot (-1) && +8 \cdot 1 && +3 \cdot (-1) && = -4, \\
\widehat{f}(3) &= \sum_{a \in \mathbb{Z}/4\mathbb{Z}} f(a)\omega^{3 \cdot a} = 0 \cdot 1 && +9 \cdot (-i) && +8 \cdot (-1) && +3 \cdot i && = -8 - 6i.
\end{aligned}$$

This can also be written in a vector notation. If we do that, then we find that the Fourier transform of the vector v is given by the vector $\widehat{v} = (20, -8 + 6i, -4, -8 - 6i)$.

Now we know how the Fourier transform works, we will try to improve this method to make it work faster. This will be done in the next section, where we will also look at an application of the Fourier transform.

3 Fast multiplication

In this section we will look at the first application of the Fourier transform. This application uses the Fourier transform to be able to do multiplications faster, in terms of the operations needed for the computation. This idea is based on the algorithm for faster computation of the Fourier transform explained by Cooley and Tukey [CT65]. First we will take a look at the so-called Fast Fourier Transform, which allows us to perform the Fourier transform a lot quicker. Once we have this, we will take a look at the method for faster multiplication.

3.1 The Fast Fourier Transform

In this part we will look at the general method to increase the speed for computing the Fourier transform. This method has been described for example by Ando Emerencia [Eme]. We will do this in terms of polynomials with variable x. Therefore we define $p(x)$ as a polynomial of degree $n-1$:

$$p(x) = p_0 + p_1 x + \cdots + p_{n-1} x^{n-1}.$$

The reason why we look at a polynomial, is that the Fourier transform can be seen as computing values of a polynomial. We therefore let the values $p_0, \ldots, p_{n-1}$ be given by the values of the function f of which we want to compute the Fourier transform. Then we have to calculate all the values of this polynomial at the roots of unity, which will give us the Fourier transform. The method that we will describe now will allow us to compute all of these values in a faster way, by reducing the number of computations that are needed.

We start of by defining the polynomial $p(x)$ as described above. Then the first thing to do is that we note that if n is even, then we can split up the polynomial into two polynomials of degree $(\frac{n}{2}-1)$:

$$\begin{aligned} p_{even}(x) &= p_0 + p_2 x + p_4 x^2 + \cdots + p_{n-2} x^{\frac{n}{2}-1}, \\ p_{odd}(x) &= p_1 + p_3 x + p_5 x^2 + \cdots + p_{n-1} x^{\frac{n}{2}-1}. \end{aligned}$$

We can combine these two polynomials to describe the polynomial $p(x)$:

$$p(x) = p_{even}(x^2) + x p_{odd}(x^2). \tag{3.1}$$

If n is odd, we can add an extra term $p_n x^n$ to $p(x)$ with $p_n = 0$. Then we can do the same as above. Since this is possible, we will extend the polynomial $p(x)$ until it is a polynomial of degree n, where n is a power of 2, i.e. $n = 2^m$ for the smallest $m \in \mathbb{N}$ possible. This will be useful later on.

As said before, if we would like to calculate the Fourier transform, then we would need to evaluate $p(x)$ at all n powers of ω, where ω denotes the principal n^{th} root of unity. Since ω is the principal n^{th} root of unity, it follows that ω^2 is the principal $\left(\frac{n}{2}\right)^{th}$ root of unity. Furthermore we can note, using that $\omega = e^{\frac{2\pi i}{n}}$, that

$$\omega^{k+\frac{n}{2}} = e^{(k+\frac{n}{2})\frac{2\pi i}{n}} = e^{\frac{2n\pi i}{2n}} e^{k\frac{2\pi i}{n}} = -e^{k\frac{2\pi i}{n}} = -\omega^k.$$

This is called the *reflective property*. If we square both sides, then we find that

$$\left(\omega^{k+\frac{n}{2}}\right)^2 = \omega^{2k}.$$

Using this, together with the reflective property and Equation 3.1, we find that

$$\begin{aligned} p(\omega^k) &= p_{even}(\omega^{2k}) + \omega^k p_{odd}(\omega^{2k}), \\ p(\omega^{k+\frac{n}{2}}) &= p_{even}(\omega^{2k}) - \omega^k p_{odd}(\omega^{2k}), \end{aligned} \tag{3.2}$$

for $k = 0, 1, \ldots, \frac{n}{2} - 1$.

Now we have to evaluate p_{even} and p_{odd} at all powers of ω^2. But this can be computed in the same manner as done for the original polynomial. Continuing recursive, we can eventually find all the values of $p(x)$ evaluated at a power of ω, which will give us the Fourier transform. This is possible since we have chosen n to be a power of 2. This method is called the *Fast Fourier Transform*, or FFT for short.

An important thing to note is that if we extend the polynomial in such a way that $n = 2^m$ for the smallest m possible, then we might not get the same Fourier transform. The reason for this is that the principal root of unity ω also changes. However, for some applications this does not matter, because it will be corrected when using the inverse Fourier transform. In for example the first application that we will look at, where we will multiply integer numbers using the Fourier transform, this does not matter.
Also one can choose to ignore the new length of the polynomial, and just compute the principal root of unity by using the old length. This will give the right result for the Fourier transform, since the added term are equal to zero.

Usually calculating the Fourier transform would take $O(n^2)$ operations. However with this algorithm it only takes $O(n \log n)$ operations. Thus this method is indeed faster than the usual Fourier transform. We won't go into detail why this is true. If you are interested in this, you can find the reason in the paper of Ando Emerencia [Eme].

Example 3.1. To make things a bit more clear, we will now do an example. Therefore we take the number 3890. The first thing to do now is to write the number as a vector in base 10. If we do that, we obtain the vector $v = (0, 9, 8, 3)$. As you might remember, this is the same vector as in example 2.14. Now we want to compute its Fourier transform by using the Fast Fourier Transform. Therefore we write the vector as a polynomial, and we evaluate this polynomial in all powers of the root of unity $\omega_n = e^{\frac{2\pi i}{n}}$. This will be the Fourier transform of the vector v.

If we want to do this, then the first thing to do is to make sure that this vector has a length that is a power of 2. In this example that is already the case, since we have a vector of length 4. Therefore we do not have to add zeros at the end of the vector.
The next step is to write the vector v as a polynomial $p(x)$. This is done as follows:

$$p(x) = v_0 + v_1 x + v_2 x^2 + v_3 x^3 = 0 + 9x + 8x^2 + 3x^3.$$

Now we have to split up this polynomial $p(x)$ into p_{even} and p_{odd}. This is done by

$$\begin{aligned} p_{even}(x) &= v_0 + v_2 x = 0 + 8x, \\ p_{odd}(x) &= v_1 + v_3 x = 9 + 3x. \end{aligned}$$

Now we can calculate the values of $p(x)$ evaluated at the powers of ω_n by

$$\begin{aligned} p(\omega_n^k) &= p_{even}(\omega_n^{2k}) + \omega_n^k p_{odd}(\omega_n^{2k}), \\ p(\omega_n^{k+\frac{n}{2}}) &= p_{even}(\omega_n^{2k}) - \omega_n^k p_{odd}(\omega_n^{2k}), \end{aligned}$$

for $k = 0, 1$ and $n = 4$.

But to evaluate p_{odd} and p_{even} at these powers of ω_n, we can do the same thing again. In other words, we will split up both p_{even} and p_{odd} into their even and odd parts. If we do that we end up with

$$\begin{aligned} p_{even,even}(x) &= v_0 = 0, \\ p_{even,odd}(x) &= v_2 = 8, \\ p_{odd,even}(x) &= v_1 = 9, \\ p_{odd,odd}(x) &= v_3 = 3. \end{aligned}$$

Now we can calculate the values of $p(x)$ evaluated at the powers of ω_n by

$$\begin{aligned} p_{even}(\omega_n^j) &= p_{even,even}(\omega_n^{2j}) + \omega_n^j p_{even,odd}(\omega_n^{2j}), \\ p_{even}(\omega_n^{j+\frac{n}{2}}) &= p_{even,even}(\omega_n^{2j}) - \omega_n^j p_{even,odd}(\omega_n^{2j}), \\ p_{odd}(\omega_n^j) &= p_{odd,even}(\omega_n^{2j}) + \omega_n^j p_{odd,odd}(\omega_n^{2j}), \\ p_{odd}(\omega_n^{j+\frac{n}{2}}) &= p_{odd,even}(\omega_n^{2j}) - \omega_n^j p_{odd,odd}(\omega_n^{2j}), \end{aligned}$$

for $j = 0$ and $n = 2$.

Since $n = 2$ and $j = 0$, we have that $\omega_n^j = e^{\frac{2\pi i}{2}\cdot 0} = 1$ and $\omega_n^{j+\frac{n}{2}} = e^{\frac{2\pi i}{2}\cdot(0+\frac{2}{2})} = -1$. If we plug this into the previous equations, then we find that

$$\begin{aligned} p_{even}(1) &= 0 + 1 \cdot 8 = 8, \\ p_{even}(-1) &= 0 - 1 \cdot 8 = -8, \\ p_{odd}(1) &= 9 + 1 \cdot 3 = 12, \\ p_{odd}(-1) &= 9 - 1 \cdot 3 = 6. \end{aligned}$$

Now that we have found the values for p_{even} and p_{odd},, we can use these to compute the values for $p(x)$. Since we have here that $n = 4$, we have that $\omega_n^k = e^{\frac{2\pi i k}{4}} = i^k$. We also have that $k = 0$ or $k = 1$. For $k = 0$ we find that

$$\begin{aligned} p(\omega_4^0) &= p_{even}(i^0) + i^0 p_{odd}(i^0) = 8 + 1 \cdot 12 = 20, \\ p(\omega_4^2) &= p_{even}(i^0) - i^0 p_{odd}(i^0) = 8 - 1 \cdot 12 = -4, \end{aligned}$$

and for $k = 1$ we find that

$$\begin{aligned} p(\omega_4^1) &= p_{even}(i^2) + i^1 p_{odd}(i^2) = -8 + i \cdot 6, \\ p(\omega_4^3) &= p_{even}(i^2) - i^1 p_{odd}(i^2) = -8 - i \cdot 6. \end{aligned}$$

Hence we find that $\widehat{v} = \big(p(\omega_4^0), p(\omega_4^1), p(\omega_4^2), p(\omega_4^3)\big) = (20, -8 + 6i, -4, -8 - 6i)$, which is the Fourier transform of the vector v. As can be seen, we indeed get the same result as in example 2.14, just as we expected.

To show that this method really is faster, the method has been implemented in MATLAB. The MATLAB codes that were used can be found in Appendix A.1. Then the time it takes to do the calculation has been measured and plotted. The same has been done for the usual Fourier transform. The result can be seen on the next page.

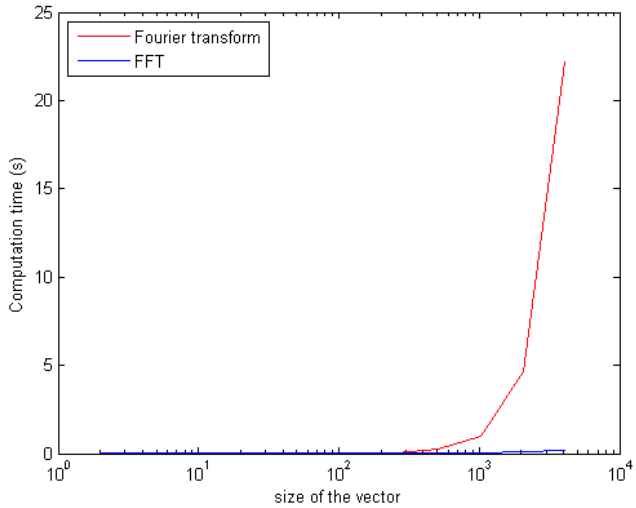


Figure 3.1: Computation times of the Fourier transform and the FFT.

As can be seen from figure 3.1, the FFT algorithm is indeed faster than the usual method. Especially when trying to compute the Fourier transform of large vectors, it is better to use the FFT.

3.2 Fast multiplication using the Fast Fourier Transform

Using the Fast Fourier Transform described in the previous part, we can now make an algorithm which allows us to multiplicate two integers in a faster way. This method has been described for example by Ando Emerencia [Eme] and by Rich Schwartz [Sch12]. But before we can actually state the algorithm, there is one more definition we need. This will be the definition of the convolution. Therefore we have to make use of a certain group ring. This is done in the following way.

We look at a cyclic group of order n denoted by $C_n = \{w_0, \ldots, w_{n-1}\}$. Then we can create the group ring $\mathbb{C}[C_n]$. This group ring has elements

$$\sum_{i=0}^{n-1} x_i w_i = x_0 w_0 + \ldots + x_{n-1} w_{n-1}, \qquad x_i \in \mathbb{C}.$$

Now we can identify such an element with the vector in the basis $\{w_0, \ldots, w_{n-1}\}$, which is the vector

$$x = (x_0, \ldots, x_{n-1}), \qquad x_i \in \mathbb{C} \text{ for } i = 0, \ldots, n-1.$$

Using this, we can define the convolution of two vectors X and Y, denoted by $X * Y$, as the product of $\sum_{i=0}^{n-1} x_i w_i$ and $\sum_{i=0}^{n-1} y_i w_i$ in $\mathbb{C}[C_n]$. If we do that, we end up with:

$$\left(\sum_{i=0}^{n-1} x_i w_i\right)\left(\sum_{j=0}^{n-1} y_j w_j\right) = \sum_{k=0}^{n-1}\left(\sum_{i+j\equiv k \mod n} x_i y_j\right) w_k$$

If we identify this again with the vector in the basis $\{w_0, \ldots, w_{n-1}\}$, then we get the convolution. This gives us the following definition of the convolution.

Definition 3.2. Let $X = (x_0, \ldots, x_{n-1})$ and $Y = (y_0, \ldots, y_{n-1})$ be vectors in $\mathbb{C}^n$. Then the *convolution* $Z = (z_0, \ldots, z_{n-1})$ of X and Y, $Z \in \mathbb{C}_n$ is denoted by $Z = X * Y$ and it is defined by

$$z_k = \sum_{a+b \equiv k \mod n} x_a y_b \qquad \text{for } k = 0, \ldots, n-1. \tag{3.3}$$

One useful property of the convolution is stated in the next proposition.

Proposition 3.3. *Let X and Y be as in the previous definition, and let $\widehat{X}$ and $\widehat{Y}$ denote the Fourier transforms of X and Y respectively. Then*

$$\widehat{X}\widehat{Y} = \widehat{X * Y}, \tag{3.4}$$

where $\widehat{X * Y}$ denotes the Fourier transform of the convolution of X and Y, and where the multiplication is point-wise multiplication.

Proof. Note that

$$\widehat{X} = (X_0, \ldots, X_{n-1}),$$

where

$$X_j = \sum_{a=0}^{n-1} x_a \omega^{ja}, \qquad \text{for } j = 0, \ldots n-1.$$

An analogue expression exists for the Fourier transform of Y. Then

$$\widehat{X}\widehat{Y} = (V_0, \ldots, V_{n-1}),$$

where

$$V_j = \left(\sum_{a=0}^{n-1} x_a \omega^{ja}\right)\left(\sum_{b=0}^{n-1} y_b \omega^{jb}\right) = \sum_{c=0}^{n-1}\left(\sum_{a+b \equiv c \mod n} x_a y_b\right)\omega^{jc}, \qquad \text{for } j = 0, \ldots, n-1.$$

As stated before, we have that

$$X * Y = Z = (z_0, \ldots, z_{n-1}),$$

where

$$z_k = \sum_{a+b \equiv k \mod n} x_a y_b, \qquad \text{for } k = 0, \ldots, n-1.$$

Then the Fourier transform of this convolution is given by

$$\widehat{X * Y} = (Z_0, \ldots, Z_{n-1}),$$

where

$$Z_j = \sum_{c=0}^{n-1} z_c \omega^{jc} = \sum_{c=0}^{n-1}\left(\sum_{a+b \equiv c \mod n} x_a y_b\right)\omega^{jc}, \qquad \text{for } j = 0, \ldots, n-1.$$

As can be seen, the expressions for $\widehat{X}\widehat{Y}$ and $\widehat{X * Y}$ are equal, and thus we have proven the proposition. □

Now we have all the information we need to define the faster multiplication of integers. We will do this in terms of polynomials with variable x. This time these polynomials won't represent the Fourier transform, but they will represent our integer numbers that we will multiply. The variable x will denote the base of the vectors that the numbers will be rewritten in.

To define the multiplication, take two polynomials

$$p(x) = \sum_{i=0}^{n-1} p_i x^i, \qquad q(x) = \sum_{j=0}^{n-1} q_j x^j,$$

which represent our integers. For both $p(x)$ and $q(x)$ we sum up to $n-1$, however we do allow the possibility that some of the last coefficients are equal to 0. The product of these two polynomials is given by

$$p(x)q(x) = \left(\sum_{i=0}^{n-1} p_i x^i\right)\left(\sum_{j=0}^{n-1} q_j x^j\right) = \sum_{k=1}^{2n-2}\left(\sum_{i+j=k} p_i q_j\right) x^k. \tag{3.5}$$

As we can see, the inner sum looks a bit like the expression for the convolution. The only difference is that in the equation for the convolution we have that $i + j = k \mod n$. To make sure that we can replace our inner sum by the convolution, we take some value $N > 2n-2$, and we create the vectors

$$P = (p_0, \ldots, p_{n-1}, 0, \ldots, 0) \in \mathbb{C}^N \text{ and } Q = (q_0, \ldots, q_{n-1}, 0, \ldots, 0) \in \mathbb{C}^N.$$

The elements of these vectors will be the coefficients for our new polynomials $p(x)$ and $q(x)$ that we will multiplicate, which will still denote the same integer number.
If we now have that $i + j = k \geq N$, then $i + j > 2n - 2$, and therefore either $i > n - 1$ or $j > n - 1$. Hence either $p_i = 0$ or $q_j = 0$. It then follows that the equation for the inner sum and the convolution give the same answer, and we can thus replace the inner sum by the convolution. Therefore we let $C = P * Q = (c_0, \ldots, c_N)$ denote the convolution. Then

$$p(x)q(x) = \sum_{h=0}^{N} c_h x^h. \tag{3.6}$$

Using Proposition 3.3, we find that

$$C = \mathcal{F}(\widehat{P}\widehat{Q}), \tag{3.7}$$

where $\mathcal{F}$ denotes the inverse Fourier transform.

When we replace the x by a coefficient which represents the base we are working with, we need to be a bit careful. The reason for this is that after we have calculated the values for C in Equation 3.7 and put this in Equation 3.6, it is possible that the coefficients of x^h is larger than $x - 1$ for some h. In that case we need to perform "carrying". For example, if we have chosen $x = 10$, and if it happens that the coefficient of 10^h is equal to 19, then we replace this coefficient by 9, and we add 1 to the coefficient of 10^{h+1}. In this way we get the true base-10 expansion.

To illustrate how this multiplication algorithm works exactly, we will do an example.

Example 3.4. In this example we will try to multiplicate the numbers 983 and 25. If we apply the naive method of multiplication, then we would do:

$$\begin{aligned} 983 \cdot 25 &= 900 \cdot 20 + 900 \cdot 5 + 80 \cdot 20 + 80 \cdot 5 + 3 \cdot 20 + 3 \cdot 5 \\ &= 18000 + 4500 + 1600 + 400 + 60 + 15 \\ &= 24575. \end{aligned}$$

Now we will try to do the same computation by using the Fourier transform. First we need to write the numbers as vectors with base 10. Then we find the two vectors

$$v = (3, 8, 9) \qquad \text{and} \qquad w = (5, 2).$$

The next thing to do is to make sure that these vectors have the same length, and that this length is a power of 2 to be able to use the Fast Fourier Transform. This power of 2 has to be as small as possible, but bigger than 2 times the length of the vectors to prevent terms from wrapping around. We do that by adding zeros at the end of the vector. If we do this, we get the vectors

$$v = (3, 8, 9, 0, 0, 0, 0, 0) \qquad \text{and} \qquad w = (5, 2, 0, 0, 0, 0, 0, 0).$$

Now we can apply equation 3.7 to find the convolution of the two numbers. Therefore we first need the Fourier transforms of the vectors v and w. After applying the Fast Fourier Transform, we find that these are given by

$$\begin{aligned} \widehat{v} = \Big(&20,\ 3 + 4\sqrt{2} + (9 + 4\sqrt{2})i,\ -6 + 8i,\ 3 - 4\sqrt{2} + (-9 + 4\sqrt{2})i, \\ &4,\ 3 - 4\sqrt{2} - (-9 + 4\sqrt{2})i,\ -6 - 8i,\ 3 + 4\sqrt{2} - (9 + 4\sqrt{2})i\Big), \\ \widehat{w} = \Big(&7,\ 5 + \sqrt{2} + \sqrt{2}i,\ 5 + 2i,\ 5 - \sqrt{2} + \sqrt{2}i,\ 3,\ 5 - \sqrt{2} - \sqrt{2}i,\ 5 - 2i,\ 5 + \sqrt{2} - \sqrt{2}i\Big). \end{aligned}$$

Now we have to point-wise multiply these vectors, which gives us

$$\begin{aligned} \widehat{v} \cdot \widehat{w} = \Big(&140,\ 15 + 14\sqrt{2} + (61 + 32\sqrt{2})i,\ -46 + 28i,\ 15 - 14\sqrt{2} + (-61 + 32\sqrt{2})i,\ 12, \\ &15 - 14\sqrt{2} - (-61 + 32\sqrt{2})i,\ -46 - 28i,\ 15 + 14\sqrt{2} - (61 + 32\sqrt{2})i\Big). \end{aligned}$$

All that is left is to take the inverse Fourier transform of this vector, which can be done in a similar way to the method for the Fast Fourier Transform to improve the computation time. This gives us that

$$v * w = (15, 46, 61, 18, 0, 0, 0, 0).$$

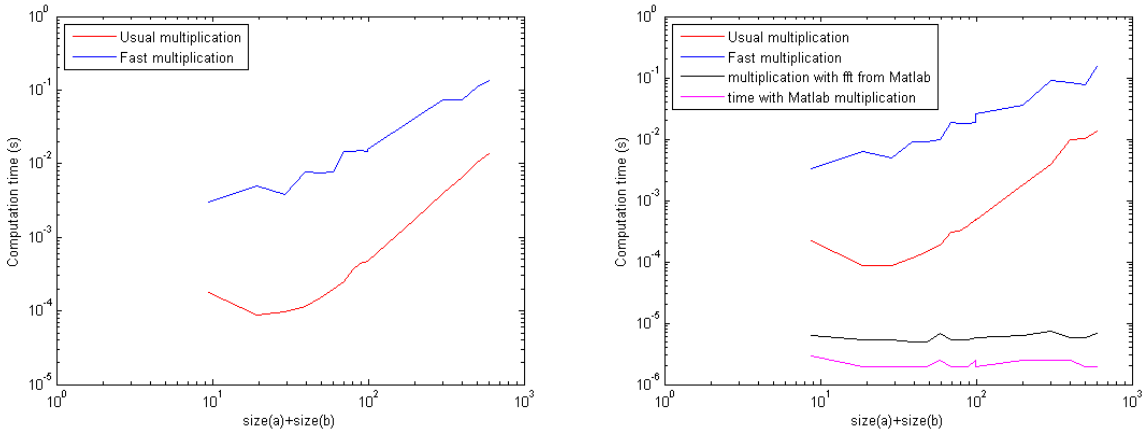
Since the elements of this vector need to be between 0 and 9, we have to perform carrying. This is done as follows:

$$\begin{aligned} (15, 46, , 61, 18, 0, 0, 0, 0) &\Rightarrow (5, 47, 61, 18, 0, 0, 0, 0) \\ &\Rightarrow (5, 7, 65, 18, 0, 0, 0, 0) \\ &\Rightarrow (5, 7, 5, 24, 0, 0, 0, 0) \\ &\Rightarrow (5, 7, 5, 4, 2, 0, 0, 0). \end{aligned}$$

If we rewrite this vector in base 10 back to a number, we find that $983 \cdot 25 = 24575$. This is indeed the same as we found with the usual multiplication.

Note that for these two vectors it was easier to compute the convolution by using the definition. We have chosen to do it by using Equation 3.7 to show that it really works. But this method is mainly focused on improving the computation time for larger integers.

To show that this method really is faster, the method has been implemented in MATLAB. The MATLAB codes can be found in Appendix A.2. Then the time it takes to do the calculation has been measured and plotted. The same has been done for the naive method of multiplication, and the corresponding build-in MATLAB functions. The result can be seen below.



(a) Mult and FastMult

(b) Mult, FastMult and build-in functions

Figure 3.2: Computation times of multiplication using the usual method and the FFT. In the second plot also the computation times of the usual multiplication in MATLAB and the method using the fft from MATLAB have been plotted.

As can be seen from figure 3.2a, the fast multiplication does not seem to be faster than the usual method. However, it seems like the lines will eventually intersect. From that point the fast multiplication will indeed be faster. This will only be for very large integers. This could not be checked, since MATLAB cannot handle such large integers.
The build-in codes in MATLAB do not seem to eventually intersect, as can be seen in figure 3.2b. The reason for this can be that the multiplication in MATLAB does not use the naive method of multiplication, but it has been improved such that it is as fast as possible. Therefore the Fast Fourier Transform might even already be a part of the MATLAB function for multiplication. Therefore it is most likely that these lines will never intersect.
The interesting thing is that the line of the naive method is always above the line which uses the fft function from MATLAB to do the fast multiplication, and it seems that it will stay there for even larger numbers. That shows that the idea of the fast multiplication is correct, and that it can be faster than the naive multiplication. The reason why this is not the case for the implemented fast multiplication is that this has been implemented in a very naive way. This code can be improved a lot to make it quicker. If that is done, then it is likely that we can see that the fast multiplication is faster than the naive method for smaller numbers than it is expected to be now.

4 Watermarking music

For now we only have been looking at one application of the Fourier transform. However, there are many more. Right now, we will look at one of those other applications. In this application we will try to change music a bit by using the Fourier transform. If it is possible for us to do so, then we will try to change the music in such a small way such that we won't hear the change anymore, but that it is noticeable for a computer that something changed. If this is possible, then it is possible to hide secret information within a sound, and we can watermark our music.

The first question that arises for this problem, is the question how we can use the Fourier transform on a sound. Therefore we need to know that it is possible to write a piece of music as a vector of numbers. We will look at the case where these numbers are sequences of 8 bit integers. How this is done will not be explained here in detail. In the C code in Appendix A.3 can be seen how this has been done.

After we have this string, we still have to find a way to use the Fourier transform. The idea to do this is as follows: We would like to use the Fourier transform on the data. Since the elements in the vector are 8 bit integers, these are elements of $\mathbb{F}_{256}$. To do a Fourier transform we need a function f that sends the string of the 8 bit integers to $\mathbb{F}_{256}$. Let G denote the sequence of 8 bit integers. Then we thus have a function $f : G \to \mathbb{F}_{256}$. But for the Fourier transform we need a function that goes to $\mathbb{C}^*$ instead of $\mathbb{F}_{256}$. It turns out that this is not a big problem. We can still define the Fourier transform in the same way. Only a few details will change as we will see in a moment. Thus we will the Fourier transform in the following way.

$$\widehat{f}(\chi) = \sum_{h \in G} f(h)\chi(h), \qquad \chi \in \widehat{G}. \tag{4.1}$$

Obviously, $f(h) \in \mathbb{F}_{256}$. But for the Fourier transform to make any sense, we must have that $\chi(h) \in \mathbb{F}_{256}$ as well. And if we want to be able to find an inverse, we must even have that $\chi(h) \in \mathbb{F}_{256}^*$. Thus χ must be a character such that $\chi : G \to \mathbb{F}_{256}^*$.
Because χ also must satisfy other properties, like being a homomorphism, there are only a few possibilities left for the group G. One choice of G could be $\mathbb{Z}/255\mathbb{Z}$. If we do that, then we can take for the principal character χ the function that sends an element $h \in \mathbb{Z}/255\mathbb{Z}$ to α^h, where α is a primitive root of $\mathbb{F}_{256}^*$. All the characters are then given by the powers of χ since χ is a primitive root of $\mathbb{F}_{256}^*$, where we use that $\chi^n(h) = \chi(hn) = \alpha^{hn}$. Then the Fourier transform becomes as in the following definition.

Definition 4.1. The Fourier transform of a function $f : \mathbb{Z}/255\mathbb{Z} \to \mathbb{F}_{256}$ is given by

$$\widehat{f}(\alpha^n) = \sum_{h \in G} f(h)\alpha^{hn}, \qquad n \in \mathbb{Z}/255\mathbb{Z}. \tag{4.2}$$

Here α is a primitive root of $\mathbb{F}_{256}^*$.

Corollary 4.2. *The Fourier transform of a function $f : G \to \mathbb{F}_{256}$ has order 4.*

Proof. To show that this Fourier transform has order 4, we will actually perform the Fourier transform 4 times to the basis of a group G. We have already seen that the basis of G will be transformed to the basis given by all characters χ^b. To see what happens if we apply the Fourier transform two times, we have to compute the Fourier transform of χ^b. This will give us

$$\widehat{\chi^b}(j) = \sum_{a \in G} \chi(aj)\chi^b(a) = \sum_{a \in G} \chi(aj + ab) = \sum_{a \in G} \chi^{j+b}(a) = \begin{cases} 1 & \text{if } j + b = 0 \\ 0 & \text{otherwise} \end{cases}.$$

Hence we find that $\widehat{\chi^b} = \mathbb{1}_{\{-b\}}$. The proof for the fact that the sum equals one or zero is analogue to the proof of Proposition 2.7. Here we only have to use the order of the groups to get one instead of the order of the group.
Since $-b \in G$, we have that $\mathbb{1}_{\{-b\}}$ is an element of the basis for G. Therefore we find that the Fourier transform of this will give χ^{-b}. Using the Fourier transform one last time we get $\mathbb{1}_{\{b\}}$. This was the basis that we started with, and thus this Fourier transform has order 4. □

As we have seen before we showed that the order of the Fourier transform is equal to 4, we must have that $G = \mathbb{Z}/255\mathbb{Z}$. However, the sequence of 8 bit integers might be a lot longer. Therefore we need to chop the sequence into parts of length 255. If at the end this is not possible anymore, and there are still a few numbers left, we can just throw them away. The reason for this is that these numbers only represent a really short part of the sound, which is only a small fragment of a second. If this is left out, then this won't be heard by the human ear.

So we know how we can apply the Fourier transform to a sound. Now we can ask ourselves the question whether we can use this knowledge to do something useful with this. What we could try for example is if we can use the Fourier transform to put a watermark onto a piece of music. This could be useful in real life. For example, imagine that you have created a piece of music, but you don't want others to distribute it. Therefore you want to put a watermark on the pieces of music that you give to others. In case they do distribute that piece of music, then you can proof that it was your piece of music by showing the watermark.
But how can we use the Fourier transform to put a watermark on the piece of sound? One method is to just take some kind of key, which is a string of numbers. The we add this string of numbers to the values in the Fourier spectrum of the music. Then we use the Fourier transform to get something that can be played as a sound again. This will change the music a bit. However, we don't want to change the music too much, since we preferably have that we hear the same music, but we can still detect the watermark. So we have a few requirements, which are based on the requirements given by Cox et al. [CKLS97]. These requirements are as follows.

The watermark should be:

1. **(Unobtrusive)** The watermark cannot be heard when added to the music file.
2. **(Robust)** The watermark is difficult or impossible to remove. If someone tries to remove it, then the music file should be damaged in such a way that it won't sound the same. Preferably it cannot be recognized at all when somebody has tried to remove the watermark.
3. **(Error correcting)** In case a few bits of the places where the watermark was placed have been changed, then the watermark should still be retrievable.

To see if we can create a watermark such that it meets the requirements, a few different possibilities have been tried. The piece of music that has been used is the following:

Original sound:[1]

Play/Pause

We have a sound of some people talking gibberish. In this way we can easily hear if the sound is changed, because if we can't understand anymore what they are saying, then the file has been altered. Also there are parts with silence, which allow us to easily tell if something has been changed there. Because if we can hear something there, then something was altered there.

[1]The sound might not be played correctly. This depends on your PDF reader. If you cannot play the sound, then you could try to use a different PDF reader.

The key that was used for the watermark was the sentence "This is a secret message that we will use to put a watermark on our music file", which was rewritten as bytes. This gave us a vector $(B_1, B_2, \ldots, B_m)$, where each element represents one byte. But each byte consists of 8 bits. Thus we can write $B_i = b_{8(i-1)+1} b_{8(i-1)+2} \cdots b_{8(i-1)+7} b_{8i}$. This gives us a new vector $(b_1, b_2, \ldots, b_{8m})$.
To create the watermark, we can try for example to add this vector to the values in the Fourier spectrum of the music file. However, there are not enough numbers to add. Therefore the same thing was added over and over again, until there was something added to every number in the Fourier spectrum.
After this had been done, the new string of numbers had been converted back by using the inverse Fourier transform. The result of this was a lot of noise. The original sound could not be heard or even recognized. What we can conclude from this is that our watermark is too much notable. Therefore we can try to make it less notable by not putting it everywhere. For example, we can choose to put in in every 10th chunk of 255 numbers. That is also what has been tried next. This resulted in the following sound:

Sound with watermark on every 10th chunk of length 255:

Play/Pause

As can be heard, the watermark can still easily be heard, but we can also recognize the original sound. So this method works better. But to make it in such a way that the watermark cannot be heard, we have to split it up in such a way that it is divided in small pieces that are not close to each other. So for example, we could put the watermark on every 1000th bit or on even less bits. Because the parts with the watermark are so small, it cannot be heard by the human ear. However, if we do this, then it won't be very safe. The reason for this is that it is known to everyone how the watermark has been put on. The only thing unknown to everyone is what sentence is used for the watermark. But in this case they do know where the watermark is placed. And if we place the watermark in such a way that is cannot be heard by the human ear because it is too short, then they can just remove those bits from the music. This will also not be heard by the human ear, so the music will sound the same. But for us it is impossible to prove that the piece of music was originally ours. Therefore this method is not very safe to use, and we have to come up with a different method.

So now will try to come up with a method that is robust. So we need to find something that is unknown to everyone, but what is known to us. For example, that is the original values of the music file. So what we could do is to only change a few of the elements that have the same value. When they are changed, people don't know which values have been changed, because they don't know what they have been changed to. And especially if we change different values every time we put a watermark on a sound, then it will be even harder to figure out where the watermark has been placed exactly for people who want to remove the watermark.
Therefore the next method we will try is to only add the watermark to the elements in the Fourier spectrum that are above some value. Then only a small part will be changed, and maybe that cannot be heard by the human ear. To make sure that a watermark is placed, one could for example calculate the maximum value that occurs, and place the watermark on the highest five percent of that maximum. However, since we have a large file of numbers, calculating this maximum takes a lot of times. Therefore it has been tried to change only the numbers

above 230. The result of this was complete noise, and the original piece of music could not be recognized. The conclusion that we can draw from this is that there are a lot of numbers that are above 230. Therefore we can try to take an even higher number. So the next thing that has been tried was to only put the watermark on the numbers that were above 253, which are only 254 and 255. This resulted in the following sound.

Sound with watermark on numbers above 253:

Play/Pause

As can be heard, we can recognize the original sound, but we can also clearly hear the watermark. So this also is not working. But it might be the case that there are a lot of high numbers, and fewer low numbers. Therefore it has been tried to change only small numbers. This however had the same result as with the high numbers. This also makes sense. To see why this makes sense we look at the numbers in the Fourier spectrum as random integers between 0 and 255. Since we have a lot of these numbers in one music file, we would expect that one in every 256 numbers is for example a 255. But then there are a lot of numbers that are equal to 255, and therefore it can be heard when they are changed.
The positive thing is that we can still recognize the sound. Therefore we can try to make a slight change to this method. What we will try now is to put the watermark on only one number. In addition to that, we will now not put it on all of these numbers, but only once in every 20 times this number appears. This resulted in the following sound:

Sound with watermark on every $20^{\mathbf{th}}$ of one number:

Play/Pause

As can be heard, this is a lot better then before, but we can still hear the watermark clearly in some places. Since we can still clearly hear the watermark, we did not try to continue with this method. Therefore another approach has been considered.

This new method distributes the secret message equally over the music file exactly one time. Then the places where the watermark is placed are far away from each other such that it is likely that it won't be heard. This resulted in the following sound.

Sound with watermark equally distributed only once:

Play/Pause

As we can hear, it indeed almost does not change the sound. So this method seems to work well.

Now one might say that we said before that this approach is not robust. Because before we tried to distribute the watermark equally over the sound by for example putting it on every 1000th bit. However there is one important difference with this approach. With this new method we distribute the watermark over the sound only once. Therefore the places where it has been added depend on the length of the watermark. But this length is unknown to someone who tries to remove the watermark. In the old method we put it on every 1000th bit, so the places were known to the person who tried to remove the watermark. But if the places are unknown, then it won't be easy for someone to find these places and remove them. Therefore this method is a lot safer. Thus the second requirement we had, which was robustness, is satisfied. However, it can possibly be improved. This might be done by making slight changes to this method or one can try a completely different method to improve the robustness.

If you listen really carefully, you can still hear a tiny difference between the original sound and the sound with the watermark. However, if you don't have the original sound, then it is nearly impossible to notice the difference. So the first requirement, which was unobtrusiveness, is satisfied. However, it can be improved such that no change can be heard at all. The ways of improving are the same as for the robustness.

So the first two requirements, which were unobtrusiveness and robustness, are satisfied. The third requirements, which was the ability of correcting errors, leads us towards coding theory, where error correcting codes appear. These codes allow some changes to happen to the data. If this is the case, then the code will notice this, and it can find the original values of the places that had been changed. This works up to a certain number of changes.
For our code, this might also be arranged. However, we won't go into that, since that has nothing to do with the Fourier transform anymore. If one would like to try to improve this method, then he or she could try to use the theory from error correcting codes and apply it on this application. This might result in a watermark that is safer than it is now. This might then also increase the robustness of the watermark. It is not certain whether it will really improve the safety of the method, and this still has to be tested.

So let's quickly repeat all the methods that we have tried, and what the positives and the negatives of each method were.
We can split our method up into three groups. The first group is the group where we had predetermined the places where the watermark would be placed, which was independent of the music file, and independent of our secret message. If we would choose to spread the watermark enough, then this method will be unobtrusive. However, it is not robust, since it is known to everyone where the watermark has been placed, and thus others can remove it easily.
The second group is the group where we put the watermark only (partially) on one or more numbers. This method was not completely unobtrusive, since we could still hear the watermark in some places. This method is robust, since people don't know exactly which bits have been changed. The reason for this is that this depends on the values of the original music file, but they don't have this, so they don't know which bits have been changed. Also one can choose the number that will be changed differently every time he or she places a watermark on a piece of sound, so it will be even harder to find the places of the watermark.
The third group consists only of the method where we distributed the watermark equally and only once over the music file. This method turned out to be almost completely unobtrusive, and it is pretty robust as well.
For all these methods it holds that the safety can possibly be improved by applying the theory of error correcting codes. This however is not certain, and has still to be tested.

Here everything has been done with 8 bit music. However, this music is often not very complicated. More complicated music is given by 16 bit sounds. We have not tried to change these kinds of music. The reason for this is that the method to do the Fourier transform for this kind of music is exactly analogue to the method described for 8 bit music, but it takes a lot more computations, and therefore a lot more time.

5 Conclusion

We started with defining the Fourier transform on finite abelian groups. To do this, we needed to define characters. After these characters had been defined, we have looked at a few properties of them. Using these properties we were able to define the Fourier transform on finite abelian groups.
Once we had found this Fourier transform, we described a method called the Fast Fourier Transform which allows us to calculate the Fourier transform much faster. Using this Fast Fourier Transform, we then looked at an algorithm which allows us to multiply integers much faster. However, this method is only likely to be faster than the naive method for very large integers.
Last but not least we looked at a method that uses the Fourier transform to put a watermark onto a piece of music. Various approaches of this have been tried to meet the three requirements we had, which were unobtrusiveness, robustness, and the ability to correct errors. We eventually came up with a method which equally distributes the secret message once over the file. This method met the first two requirements, and the third one can possibly be met by using the knowledge of coding theory about error correcting codes.
Apart from improving the method by using error correcting codes, there are also other parts of this method that can be improved. For example, if you listen very carefully, then you can still hear a difference between the original piece of music, and the one with the watermark. Also one might try to make the system even more robust. To do so, one can try to continue with this method of putting the watermark on and make slight changes to it, or one can try to distribute the watermark in a completely different way in the Fourier spectrum to meet the three requirements.
We only have looked at putting a watermark on 8-bit sounds. However, there also exist other types of sounds, for example 16-bit sounds. Since 16-bit sounds contains larger numbers when written as a vector, it will take more computation time to put a watermark on these sounds, and therefore we have not looked at that. The method for putting a watermark on these types of sound is exactly the same as the method for 8-bit sounds.

6 References

[Bab02] László Babai. The fourier transform and equations over finite abelian groups. Technical report, University of Chicago, June 2002.

[CKLS97] Ingemar J. Cox, Joe Killian, Tom Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Trans*, 6(12):1673–1687, 1997. `http://www.geocities.ws/radicalnair/Research/cox95secure.pdf`.

[Con] Keith Conrad. *Characters of Finite Abelian Groups*. `http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/charthy.pdf`.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computations*, 19(90):297–301, April 1965. `https://web.stanford.edu/class/cme324/classics/cooley-tukey.pdf`.

[Eme] Ando Emerencia. *Multiplying Huge Integers Using Fourier Transform*. `http://www.cs.rug.nl/~ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_paper.pdf`.

[Goo] Andrew Goodall. *Fourier analysis on finite Abelian groups: some graphical applications*. `http://kam.mff.cuni.cz/~andrew/dwajgApr06.pdf`.

[Sch12] Rich Schwartz. *Multiplication and the Fast Fourier Transform*, October 22, 2012. `http://www.math.brown.edu/~res/MathNotes/FFT.pdf`.

A Appendix

A.1 Matlab codes for the FFT

Fourier transform:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the Fourier Transform.

% Starting the function with input v.
% v is a vector of length n.
% Output is the vector f_hat, which is the Fourier transform of v.
function f_hat = FourTrans(v)
n = length(v); %Computing the length of the vector a.
omega = exp(2*pi*1i/n); % Computing the n'th root of unity omega.
% Creating the vector f_hat.
f_hat = zeros(1,n);
% Creating a vector w with zeros, which will be filled later.
w = zeros(1,n);
% Computing the Fourier transform.
for j=0:n-1
    for k=0:n-1
        w(k+1) = v(k+1)*omega^((n-j)*k); % Filling w.
    end
    f_hat(j+1) = sum(w); % Computing the j'th element of the Fourier transform.
end
end
```

Fast Fourier Transform (FFT):

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the Fast Fourier Transform.

% Starting the function with input a.
% a is a vector of arbitrary length.
% Output is the vector y, which is the Fourier transform of a extended to a
% vector of length 2^m with zeros, where m is the smallest integer possible such that
% 2^m is greater than or equal to n.
function[y] = FFT(a)
n = length(a); %Computing the length of the vector a.
if n==1 %If a is already a scalar, then the Fourier transform is the same as a.
    y = a;
end
% Extending the vector to a vector of length 2^m.
if n~=1
```

```
    x = 2^(nextpow2(n));
    if n~=x
        a(x)=0;
    end
    % Creating the vectors a_even and a_odd.
    a_even = zeros(1,0.5*x);
    a_odd = zeros(1,0.5*x);
    % Filling in the right values for a_even and a_odd.
    for i=1:0.5*x
        a_even(i) = a(2*i-1);
        a_odd(i) = a(2*i);
    end
    % Recursive calls of the FFT.
    y_even = FFT(a_even);
    y_odd = FFT(a_odd);
    % Computing the x'th root of unity omega, and computing its powers.
    omega = exp(2*pi*1i/x);
    Omega = zeros(1,x);
    for k=1:(0.5*x+1)
        Omega(k) = omega^(k-1);
    end
    % Computing the Fast Fourier Transform.
    for j=1:0.5*x
        y(j) = y_even(j) + Omega(j)*y_odd(j);
        y(j+0.5*x) = y_even(j) - Omega(j)*y_odd(j);
    end
end
end
```

Computation times for the Fourier transform and the FFT:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the times needed to compute the Fourier transform
% and the Fast Fourier Transform.

% Creating empty vectors for the computed times.
time_FT = zeros(1,12);
time_FFT = zeros(1,12);

% Computing the time it takes to do the Fourier transform and the FFT for
% vectors of different lengths.
for j = 1:12
    a = randi(10,1,2^j); %Creating a random vector of length 2^j.
    omega = exp(2*pi*1i/j); % Creating the n^th root of unity omega.
    tic; %Starting to measure the time for the Fourier transform.
    [f_hat] = FourTrans(a); %Computing the Fourier transform.
```

```
    time_FT(j) =  toc; %computing the time it took to compute the Fourier transform.
    tic; %Starting to measure the time for the FFT.
    [y] = FFT(a); %Computing the FFT.
    time_FFT(j) =  toc; %computing the time it took to compute the FFT.
end
x = 2.^(1:12); % Creating a vector for the x-axis with the lengths.
figure
semilogx(x,time_FT,'r',x,time_FFT,'b') % Making a plot of the different times.
ylabel('Computation time (s)')
xlabel('size of the vector')
legend('Fourier transform','FFT','Location','northwest')
```

A.2 Faster multiplication using the FFT:

The inverse Fourier transform:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the Inverse of the Fast Fourier Transform.

% Starting the function with input b.
% b is a vector of arbitrary length.
% Output is the vector f, which is the inverse Fast Fourier transform of b extended to a
% vector of length 2^m with zeros, where m is the smallest integer possible such that
% 2^m is greater than or equal to n.
function [f] = InvFFT(b)
n = length(b); %Computing the length of the vector b.
f = 1/n*nInvFFT(b); % Computing the inverse Fast Fourier Transform of b.
% Taking the real part and rounding it to get rid of some rounding errors.
f = round(real(f(1:n)));
end
```

In this code we use the function nInvFFT, which is given by:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the Inverse of the Fast Fourier Transform.

% Note that this function only computes the inverse Fast Fourier
% transform of a vector times its length. The function InvFFT calculates the real
% inverse Fast Fourier Transform using this function.

% Starting the function with input b.
% b is a vector of length n, with n a power of 2.
% Output is the vector f, which is the inverse Fourier transform of b' times its length.
% Here b' denotes the vector b which is extended to a vector of length 2^m with zeros,
% where m is the smallest integer possible such that 2^m is greater than or equal to n.
function[f] = nInvFFT(b)
n = length(b); %Computing the length of the vector b.
% If b is already a scalar, then the inverse Fourier transform is the same.
if n==1
    f = b;
end
% Extending the vector to a vector of length 2^m.
if n~=1
    x = 2^(nextpow2(n));
    if n~=x
        b(x)=0;
    end
```

```
    % Creating the vectors b_even and b_odd.
    b_even = zeros(1,0.5*x);
    b_odd = zeros(1,0.5*x);
    % Filling in the right values for b_even and b_odd.
    for i=1:0.5*x
        b_even(i) = b(2*i-1);
        b_odd(i) = b(2*i);
    end
    % Recursive calls of the FFT.
    f_even = nInvFFT(b_even);
    f_odd = nInvFFT(b_odd);
    % Computing the x'th root of unity omega, and computing its powers.
    omega = exp(2*pi*1i/x);
    Omega = zeros(1,x);
    for k=1:(0.5*x+1)
        Omega(k) = omega^(x-k+1);
    end
    % Computing the inverse of the Fast Fourier Transform (times n).
    for j=1:0.5*x
        f(j) = (f_even(j) + Omega(j)*f_odd(j));
        f(j+0.5*x) = (f_even(j) - Omega(j)*f_odd(j));
    end
end
end
```

Faster multiplication using the FFT:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Faster multiplication using the Fast Fourier Transform.

% Starting the function with input a and b.
% a and b are numbers in base 10.
% Output is the vector c, which is the result of multiplying a and b.
function [c] = FastMult(a,b)
% First we write a and b as vectors, instead of numbers.
% This gives the base 10 vector.
% The vectors are flipped to get the value of 10^i on the i'th place in the
% vector.
a = fliplr(sprintf('%.0f', a) - '0');
b = fliplr(sprintf('%.0f', b) - '0');
n_a = length(a); %Computing the length of the vector a.
n_b = length(b); %Computing the length of the vector b.
% Now we put zeros at the end of the vectors to get vectors of length 2^n,
% for the smallest n possible.
if n_a > n_b
    x = 2^(nextpow2(2*n_a));
```

```
    if n_a~=x
        a(x)=0;
    end
    b(x)=0;
end
if n_b > n_a
    x = 2^(nextpow2(2*n_b));
    if n_b~=x
        b(x)=0;
    end
    a(x)=0;
end
if n_a==n_b
    x = 2^(nextpow2(2*n_b));
    if n_b~=x
        b(x)=0;
        a(x)=0;
    end
end
% Calculating the Fourier Transforms of a and b, and multiply (element-wise)
%these with each other.
Fa = FFT(a);
Fb = FFT(b);
ab = Fa.*Fb;
% Calculating the inverse of this product, and taking the real part and rounding it
%to get rid of rounding errors.
c = round(fliplr(Carrying(real(InvFFT(ab)),10)));
% Deleting the possible zeros that appear at the beginning of the vector c.
c(1:find(c,1,'first')-1) = [];
% Rewrite the vector c as a number.
c = str2double(sprintf('%1d',c));
end
```

In this code we used the code called 'Carrying', which is given by:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Performing carrying on a vector of base x.

% Starting the function with input a and x.
% a is a vector.
% x is the base we will be working with.
% Output is the vector b, which is the vector a after carrying.
function [a] = Carrying(a,x)
% Calculating the length of the vector a.
n = length(a);
% Performing the carrying.
```

```
for j=1:n-1
   if a(j) > x-1
      % Computing how many times x fits into a(j).
      h = floor(a(j)/x);
      % Replace a(j) by a(j) mod x.
      a(j) = a(j) - h*x;
      % Adding the number of times x fits into a(j) to a(j+1).
      a(j+1) = a(j+1) + h;
   end
end
% If a(n) becomes larger than x-1, we add an extra place to a such that we
% can continue to perform carrying.
while a(n) > x-1
    h = floor(a(n)/x);
    a(n) = a(n) - h*x;
    a=[a,0];
    a(n+1) = h;
    n=n+1;
end
end
```

Usual multiplication:

```
% Bachelor project
% Dani Hulzebos, S2379678

% The usual multiplication of two integers.

% Starting the function with input a and b.
% a and b are integers in base 10.
% Output is the vector c, which is the result of multiplying a and b.
function [c] = Mult(a,b)
% First we write a and b as vectors, instead of numbers.
% This gives the base 10 vector.
A = sprintf('%.0f', a) - '0';
B = sprintf('%.0f', b) - '0';
% Creating the value c, which will be computed later on.
c = 0;
% Calculating the lengths of the vectors A and B.
n_A = length(A);
n_B = length(B);
% Computing the usual multiplication of a and b.
for i = 1:n_A
    for j=1:n_B
        c = c + A(i)*B(j)*10^(n_A+n_B-i-j);
    end
end
end
```

Computation times for the usual multiplication and the multiplication which uses the FFT:

```
% Bachelor project
% Dani Hulzebos, S2379678

% Computing the times needed to do a multiplication in the usual
% way and with the algorithm that uses the Fast Fourier Transform.

% Creating empty vectors for the computed times and for the x-axis.
time_Mult = zeros(1,16);
time_FastMult = zeros(1,16);
time_fft = zeros(1,16);
time_mult = zeros(1,16);
x = zeros(1,16);
% Creating the vectors a and b with the integers that will be multiplied.
a = zeros(1,16);
b = zeros(1,16);
a(1:10) = round(rand*10.^(5*(1:10)));
b(1:10) = round(rand*10.^(5*(1:10)));
a(11:16) = round(rand*10.^(50*(1:6)));
b(11:16) = round(rand*10.^(50*(1:6)));
% Computing the time it takes to do the usual multiplication and the FFT multiplication
% for integers of different sizes.
for j = 1:16
    tic; %Starting to measure the time for the usual multiplication.
    [c] = Mult(a(j),b(j)); %Computing the usual multiplication.
    time_Mult(j) =  toc; %computing the time it took to do the usual multiplication.
    tic; %Starting to measure the time for the FFT multiplication.
    [d] = FastMult(a(j),b(j)); %Computing the FFT multiplication.
    time_FastMult(j) =  toc; %computing the time it took to do the FFT multiplication.
    tic; %Starting to measure the time for the FFT multiplication.
    [e] = ifft(fft(a(j))*fft(b(j))); %Computing the FFT multiplication.
    time_fft(j) =  toc; %computing the time it took to do the FFT multiplication.
    tic; %Starting to measure the time for the FFT multiplication.
    [f] = a(j)*b(j); %Computing the FFT multiplication.
    time_mult(j) =  toc; %computing the time it took to do the FFT multiplication.
    x(j) = log10(a(j))+log10(b(j));
end
% Making a plot of the different times.
loglog(x,time_Mult,'r',x,time_FastMult,'b',x,time_fft,'g',x,time_mult,'y')
ylabel('Computation time (s)')
xlabel('size(a)+size(b)')
legend('Usual multiplication','Fast multiplication','multiplication with
fft from Matlab','time with Matlab multiplication','
```

A.3 C code for sounds

This code has been created by Eduardo Ruiz Duarte. This code has been changed a bit in some places, but still it is mainly written by him. I would like to thank him for all the work he did to make this code.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include "galois.h"

/* Order of x, it will be the order of the block too */
#define ORDER 255

extern int prim_poly[];
typedef struct wav_s
{
  char RIFF_marker[4];
  uint32_t file_size;
  char filetype_header[4];
  char format_marker[4];
  uint32_t data_header_length;
  uint16_t format_type;
  uint16_t number_of_channels;
  uint32_t sample_rate;
  uint32_t bytes_per_second;
  uint16_t bytes_per_frame;
  uint16_t bits_per_sample;
  char data_marker[4];
  uint32_t datasize;
} wav_t;

/* Return an array opulses8_mono[sample] */
uint32_t
parse_one_chan_8bit (int file, uint8_t * opulses8_mono)
{
  int i = 0, r;
  uint8_t ipulses8_mono;
  /* one channel 8 bits */
  while ((r = read (file, &ipulses8_mono, sizeof (ipulses8_mono))) > 0)
    {
      opulses8_mono[i] = ipulses8_mono;
      i++;
    }
  return i;
```

```
}

void
print_poly (unsigned int f)
{
  int i, v, c = 0;
  for (i = 31; i >= 0; i--)
    {
      v = (f >> i) & 1;
      if (v != 0)
{
  if (c > 0)
    fprintf (stderr, " + ");
  if (i != 0)
    fprintf (stderr, "x^%d", i);
  else
    fprintf (stderr, "1");
  c++;
}
    }
  fprintf (stderr, "\n");
}

int
galois_elem_order (int a, int siz)
{
  int i, t = 1;
  for (i = 1; i < (int) pow (2.0, (double) siz); i++)
    {
      t = galois_single_multiply (a, t, siz);
      if (t == 1)
return i;
    }
  return 0;
}

/* character table for character(-n) = x^-n, such that x^-1 =0x8e */
uint8_t character8_invx[] = {
  0x01, 0x8e, 0x47, 0xad, 0xd8, 0x6c, 0x36, 0x1b, 0x83, 0xcf, 0xe9, 0xfa,
  0x7d, 0xb0, 0x58, 0x2c,
  0x16, 0x0b, 0x8b, 0xcb, 0xeb, 0xfb, 0xf3, 0xf7, 0xf5, 0xf4, 0x7a, 0x3d,
  0x90, 0x48, 0x24, 0x12,
  0x09, 0x8a, 0x45, 0xac, 0x56, 0x2b, 0x9b, 0xc3, 0xef, 0xf9, 0xf2, 0x79,
  0xb2, 0x59, 0xa2, 0x51,
  0xa6, 0x53, 0xa7, 0xdd, 0xe0, 0x70, 0x38, 0x1c, 0x0e, 0x07, 0x8d, 0xc8,
  0x64, 0x32, 0x19, 0x82,
  0x41, 0xae, 0x57, 0xa5, 0xdc, 0x6e, 0x37, 0x95, 0xc4, 0x62, 0x31, 0x96,
  0x4b, 0xab, 0xdb, 0xe3,
  0xff, 0xf1, 0xf6, 0x7b, 0xb3, 0xd7, 0xe5, 0xfc, 0x7e, 0x3f, 0x91, 0xc6,
```

```
  0x63, 0xbf, 0xd1, 0xe6,
  0x73, 0xb7, 0xd5, 0xe4, 0x72, 0x39, 0x92, 0x49, 0xaa, 0x55, 0xa4, 0x52,
  0x29, 0x9a, 0x4d, 0xa8,
  0x54, 0x2a, 0x15, 0x84, 0x42, 0x21, 0x9e, 0x4f, 0xa9, 0xda, 0x6d, 0xb8,
  0x5c, 0x2e, 0x17, 0x85,
  0xcc, 0x66, 0x33, 0x97, 0xc5, 0xec, 0x76, 0x3b, 0x93, 0xc7, 0xed, 0xf8,
  0x7c, 0x3e, 0x1f, 0x81,
  0xce, 0x67, 0xbd, 0xd0, 0x68, 0x34, 0x1a, 0x0d, 0x88, 0x44, 0x22, 0x11,
  0x86, 0x43, 0xaf, 0xd9,
  0xe2, 0x71, 0xb6, 0x5b, 0xa3, 0xdf, 0xe1, 0xfe, 0x7f, 0xb1, 0xd6, 0x6b,
  0xbb, 0xd3, 0xe7, 0xfd,
  0xf0, 0x78, 0x3c, 0x1e, 0x0f, 0x89, 0xca, 0x65, 0xbc, 0x5e, 0x2f, 0x99,
  0xc2, 0x61, 0xbe, 0x5f,
  0xa1, 0xde, 0x6f, 0xb9, 0xd2, 0x69, 0xba, 0x5d, 0xa0, 0x50, 0x28, 0x14,
  0x0a, 0x05, 0x8c, 0x46,
  0x23, 0x9f, 0xc1, 0xee, 0x77, 0xb5, 0xd4, 0x6a, 0x35, 0x94, 0x4a, 0x25,
  0x9c, 0x4e, 0x27, 0x9d,
  0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x8f, 0xc9, 0xea, 0x75, 0xb4,
  0x5a, 0x2d, 0x98, 0x4c,
  0x26, 0x13, 0x87, 0xcd, 0xe8, 0x74, 0x3a, 0x1d, 0x80, 0x40, 0x20, 0x10,
  0x08, 0x04, 0x02, 0x01,
};

/* character table for character(n) = x^n such that x = 0x02*/
uint8_t character8_x[] = {
  0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1d, 0x3a, 0x74, 0xe8,
  0xcd, 0x87, 0x13, 0x26,
  0x4c, 0x98, 0x2d, 0x5a, 0xb4, 0x75, 0xea, 0xc9, 0x8f, 0x03, 0x06, 0x0c,
  0x18, 0x30, 0x60, 0xc0,
  0x9d, 0x27, 0x4e, 0x9c, 0x25, 0x4a, 0x94, 0x35, 0x6a, 0xd4, 0xb5, 0x77,
  0xee, 0xc1, 0x9f, 0x23,
  0x46, 0x8c, 0x05, 0x0a, 0x14, 0x28, 0x50, 0xa0, 0x5d, 0xba, 0x69, 0xd2,
  0xb9, 0x6f, 0xde, 0xa1,
  0x5f, 0xbe, 0x61, 0xc2, 0x99, 0x2f, 0x5e, 0xbc, 0x65, 0xca, 0x89, 0x0f,
  0x1e, 0x3c, 0x78, 0xf0,
  0xfd, 0xe7, 0xd3, 0xbb, 0x6b, 0xd6, 0xb1, 0x7f, 0xfe, 0xe1, 0xdf, 0xa3,
  0x5b, 0xb6, 0x71, 0xe2,
  0xd9, 0xaf, 0x43, 0x86, 0x11, 0x22, 0x44, 0x88, 0x0d, 0x1a, 0x34, 0x68,
  0xd0, 0xbd, 0x67, 0xce,
  0x81, 0x1f, 0x3e, 0x7c, 0xf8, 0xed, 0xc7, 0x93, 0x3b, 0x76, 0xec, 0xc5,
  0x97, 0x33, 0x66, 0xcc,
  0x85, 0x17, 0x2e, 0x5c, 0xb8, 0x6d, 0xda, 0xa9, 0x4f, 0x9e, 0x21, 0x42,
  0x84, 0x15, 0x2a, 0x54,
  0xa8, 0x4d, 0x9a, 0x29, 0x52, 0xa4, 0x55, 0xaa, 0x49, 0x92, 0x39, 0x72,
  0xe4, 0xd5, 0xb7, 0x73,
  0xe6, 0xd1, 0xbf, 0x63, 0xc6, 0x91, 0x3f, 0x7e, 0xfc, 0xe5, 0xd7, 0xb3,
  0x7b, 0xf6, 0xf1, 0xff,
  0xe3, 0xdb, 0xab, 0x4b, 0x96, 0x31, 0x62, 0xc4, 0x95, 0x37, 0x6e, 0xdc,
  0xa5, 0x57, 0xae, 0x41,
```

```
  0x82, 0x19, 0x32, 0x64, 0xc8, 0x8d, 0x07, 0x0e, 0x1c, 0x38, 0x70, 0xe0,
  0xdd, 0xa7, 0x53, 0xa6,
  0x51, 0xa2, 0x59, 0xb2, 0x79, 0xf2, 0xf9, 0xef, 0xc3, 0x9b, 0x2b, 0x56,
  0xac, 0x45, 0x8a, 0x09,
  0x12, 0x24, 0x48, 0x90, 0x3d, 0x7a, 0xf4, 0xf5, 0xf7, 0xf3, 0xfb, 0xeb,
  0xcb, 0x8b, 0x0b, 0x16,
  0x2c, 0x58, 0xb0, 0x7d, 0xfa, 0xe9, 0xcf, 0x83, 0x1b, 0x36, 0x6c, 0xd8,
  0xad, 0x47, 0x8e, 0x01
};

int
galois_pow (int a, int n, int siz)
{
  int i, t = 1;
  for (i = 1; i <= n; i++)
    t = galois_single_multiply (a, t, siz);
  return t;
}

uint8_t **
chop8 (uint8_t * data, int size)
{
  uint8_t **new = calloc ((size / ORDER) + 1, sizeof (uint8_t *));
  int i, j = 0;
  for (i = 0; i <= (size / ORDER); i++)
    new[i] = calloc (ORDER, sizeof (uint8_t));

  for (i = 0; i < size; i++)
    {
      if ((i > 0) && ((i % ORDER) == 0))
j++;
      new[j][i % ORDER] = data[i];
      //if ((i > 0) && ((i % ORDER) == 0))
      //j++;
    }
  return new;
}

uint8_t
character8 (uint32_t n)
{
  return character8_x[n];
}

uint8_t
character8_inv (uint32_t n)
{
  return character8_invx[n];
}
```

```
uint8_t *
fourier8 (uint8_t * f)
{
  uint8_t *F;
  int n, m;
  F = calloc (sizeof (uint8_t), ORDER);
  for (m = 0; m < ORDER; m++)
    for (n = 0; n < ORDER; n++)
      F[m] ^=
galois_single_multiply (f[n], character8 ((n * m) % (ORDER)), 8);
  return F;
}

uint8_t *
ifourier8 (uint8_t * f)
{
  uint8_t *F;
  int n, m;
  F = calloc (sizeof (uint8_t), ORDER);
  for (m = 0; m < ORDER; m++)
    {
      for (n = 0; n < ORDER; n++)
{
  F[m] ^=
    galois_single_multiply (f[n], character8_inv ((n * m) % ORDER),
    8);
}
    }
  return F;
}

int
chopbits (char *wm_bits, int fd)
{

  char byte;
  int i, j = 0;
  while (read (fd, &byte, 1) > 0)
    {
      for (i = 0; i < 8; i++)
{
  wm_bits[j] = (byte >> i) & 1;
  j++;
}
    }
  return 1;
}
```

```
int
main (int argc, char **argv)
{
  int file, ofile;
  wav_t header;
  uint8_t ipulses8_mono, *opulses8_mono;
  uint8_t **chunks;
  uint8_t **Fchunks;
  int k, flag = 0, watermark = 0;
  int n, j;
  uint8_t t = 0;
  int secretfile;
  char *wm_bits;

/* Check, root of unity, Orthogonality condition, character tables  */

  if (strcmp (argv[1], "-c") == 0)
    {
      printf ("Check orthogonality of character\n");
      for (k = 1; k < ORDER; k++)
{
  for (j = 0; j < ORDER; j++)
    t ^= character8 ((j * k) % ORDER);
  printf ("k=%03d, t=%02x\n", k, t);
  t = 0;
}
      printf ("Root of unity ? x^%d = %02x\n", ORDER,
      galois_pow (0x02, ORDER, 8));
      printf ("Check character tables \n");
      for (k = 0; k < ORDER; k++)
{
  printf ("c^%03d * c^-%03d = %02x*%02x = %02x\n", k, k,
  character8 (k), character8_inv (k),
  galois_single_multiply (character8 (k), character8_inv (k),
  8));
}
      exit (1);
    }

  if (argc < 4)
    {
      fprintf (stderr,
       "arg1 fourier or inverse = {-f or -i}\narg2 Input.wav\narg2 Output.wav\n");
      fprintf (stderr,
       "Example inverse fourier:\n$ %s -i Input.wav Output.wav\n",
       argv[0]);
      exit (-1);
    }
```

```
  file = open (argv[2], O_RDONLY);
  ofile = open (argv[3], O_RDWR | O_CREAT, 0777);
  chmod (argv[3], 0755);
  int s;
  read (file, &header, sizeof (struct wav_s));
  fprintf (stderr, "Head size=%ld\n", sizeof (struct wav_s));
  fprintf (stderr, "Channels = %d\n", header.number_of_channels);
  fprintf (stderr, "Sample_rate=%d\n", header.sample_rate);
  fprintf (stderr, "Bits per sample=%d\n", header.bits_per_sample);
  if (strcmp (argv[1], "-f") == 0)
    {
      fprintf (stderr, "Calculating Fourier\n");
      flag = 1;
      watermark = 1;
    }
  else
    fprintf (stderr, "Calculating inverse Fourier\n");

  /* Write WAV header to screen output */
  write (ofile, &header, sizeof (struct wav_s));

  /* one channel 8 bits */
  if ((header.number_of_channels == 1) && (header.bits_per_sample == 8))
    {
      opulses8_mono = calloc (header.datasize, header.bits_per_sample / 8);
      /* s is the number of pulses of elements in F256 */
      s = parse_one_chan_8bit (file, opulses8_mono);

      /* rename to data8 the pulses */
      uint8_t *data8;
      struct stat finfo;
      int c = 0;
      data8 = opulses8_mono;

      Fchunks = calloc (sizeof (void *), (s / ORDER) + 1);
      chunks = chop8 (data8, s);

      /* Calculate Fourier in each block and output the block to the screen */

      if (watermark)
{
  secretfile = open (argv[4], O_RDONLY);
  fstat (secretfile, &finfo);
}

      wm_bits = calloc (finfo.st_size, 8);
      chopbits (wm_bits, secretfile);

      int num_chunks = s / ORDER;
```

```
      int secret_bits_per_chunk = finfo.st_size * 8 / num_chunks;
      if(secret_bits_per_chunk == 0)
      secret_bits_per_chunk = 1;

      printf ("Number of chunks = %d\n", num_chunks);
      printf ("secret bits per chunk = %d\n", secret_bits_per_chunk);

       s = 0;
      int i;
      for (k = 0; k < num_chunks; k++)
{
  if (flag)
    {
      Fchunks[k] = fourier8 (chunks[k]);

      if(s < finfo.st_size*8)
      for (i = 0; i < secret_bits_per_chunk; i++)
{
  Fchunks[k][i * (ORDER / secret_bits_per_chunk)] ^=
    wm_bits[s];
  s++;
}
    }
  else
    Fchunks[k] = ifourier8 (chunks[k]);
  /* 1 means screenoutput */


  write (ofile, Fchunks[k], ORDER * sizeof (uint8_t));
}
      printf("Written %d bits of secret\n",s);
      return 0;
    }
}
```

The part at the end, starting at if(flag) up to else, has been changed to obtain the sounds for different methods of placing the watermark. All the different methods have not been included here. The method shown here is the last method used, which puts the watermark on the sound only once by distributing it equally.