Error Free High Quality Audio Signal Processing

BY

Robin Ince

**A Dissertation submitted to the
School of Computing, Science and Engineering
in partial fulfillment of the requirements
for the Degree**

MSc Audio Acoustics

**University of Salford
Salford, UK**

August, 2005

Supervisor: Prof. J. Angus

ABSTRACT

Error Free High Quality Audio Signal Processing

BY

Robin Ince

MSc Audio Acoustics

University of Salford
Salford, UK

The area of number theoretic transforms over a finite field structure is introduced, with application to signal processing of high quality digital audio. Such transforms eliminate computational errors due to round-off and truncation that occur in conventional signal processing, presenting the possibility of truly lossless digital audio processing. The relevant signal processing theory is summarised and the mathematical background pertinent to the area of finite fields and number theoretic transforms is presented. A selection of possible transforms are introduced and assessed. A MATLAB implementation of a complex mersenne number transform is developed, and examples of processing 24 bit digital audio are given. The viability of the implementation is assessed, and avenues for future work are discussed.

Contents

Chapter 1
Introduction and Literature Review

The mathematical concept of a field (Section 2.2.1) was used implicitly by Niels Henrik Abel and Evariste Galois while studying the solvability of equations, but was first clearly defined in 1893 by Heinrich Weber. The theory of groups and fields was formalised by Emil Artin in the first half of the twentieth century. Since then these ideas have found many applications including coding theory, computer science, solution of equations, multiplication of very long integers and signal processing.

Finite fields (also called Galois fields) and transforms thereon were first considered in a signal processing context, relevant to this work, during the 1960's and 70's. At that time digital technologies were in early stages of development, and hardware limitations restricted their usefulness for signal processing. Many people were searching for ways to improve efficiency of signal processing algorithms, and finite field transforms were investigated as a possible way to reduce the costs of convolution.

The first use of finite fields in an engineering context was by Bartree and Schneider [1] in which they introduce Galois fields (GF's) and what would now be known as extension fields. They outline a hardware implementation for GF computation and detail an application for a tracking radar. Following this, Pollard [2] was the first to introduce the generalised number theoretic transform (or NTT) and list it's basic properties, including the cyclic convolution property. He also provides some applications, such as multiplication of polynomials and very large integers, as well as covering practical methods for performing the transform and a generalisation to k dimensions. Building on this work Rader [3] considers transforms on fields whose order is a Mersenne number (not necessarily prime), since these numbers allow efficient calculation in the digital domain. While he is concerned with using the transforms to perform convolutions, the goal is efficient rather than errorless computation. He uses 2 as a primitive element for the transform which introduces a restriction on the length of transforms, but allows computation with just additions and bit rotations of the word. He concludes that the transforms may be useful in special cases, for example when multiplication is very costly.

Perhaps the most important reference for this work is Reed and Truong [4]. They consider the number theoretic transform over $GF\left(q^2\right)$, an extension field of $GF(q)$ which is analogous to the field of complex numbers. Using q a Mersenne prime this extension yields highly composite transform lengths allowing efficient computation via FFT type algorithms. They highlight the fact that this transform is information-lossless, however the primitive element used in the transform means that multiplication cannot be avoided and the computation is more costly. They also develop an algorithm for finding primitive elements. In another paper in the same year [5], they discuss a method to increase the dynamic range of the transforms within a limited word length by using the Chinese Remainder Theorem (CRT) to split a larger field into a direct sum of smaller fields. They also give examples of using the technique to compute the usual discrete Fourier transform (DFT) without round-off error.

Another significant paper published in 1975 was Agarwal and Burrus [6]. They provide a comprehensive introduction to the field, covering the basics of convolution, modular arithmetic and number theoretic transforms. They concentrate on the Fermat Number Transform, with analysis of overflow and quantization considerations. They also mention Mersenne Number Transforms, and briefly cover Complex Number Transforms.

Nussbaumer [7] further develops the idea of transforms over $GF\left(q\right)$ with q a Mersenne prime or pseudo-Mersenne prime by inclusion of a complex root, calling them Complex Mersenne Number Transforms (CMNT) or Complex pseudo-Mersenne Number Transforms respectively. By using a complex primitive element the transform length can be increased (up to $8q$), but the possible lengths are still very restrictive and the direction of the work is very much towards reducing computation (choosing the primitive element to reduce multiplications) rather than develop a more flexible lossless system. He also [8] evaluates various NTT's for digital filtering. Computational efficiency is again the main criteria, and his conclusion is in favour of Fermat number transforms and some pseudo-CMNT's.

Other articles of interest include Murakami and Reed [9], who present a method for designing FIR filters in $GF\left(q^2\right)$, but with applications geared more towards radar and communications and Martens [10] who deals with image processing applications and develops new algorithms for longer convolution lengths. In [11] Murakami and Reed build on their earlier work, defining a complex number-theoretic z-transform (CNT z-transform) and using it to design recursive and non-recursive filters on a finite ring. Thomas *et. al.* [12] also develop more flexible transforms, but sacrifice the convolution property which makes them unsuitable for consideration here.

More recently Gudvangen [13] considers NTT's for audio processing. He details the problem of dynamic range, and concludes that while not competitive in software implementations, NTT's may have advantages is ASIC implementations. He has also produced a concise introduction to the area of number theoretic transforms [14]. Angus [15] is the work this project builds on primarily. He outlines the background of finite fields and signal processing as well as the different moduli that can be chosen, favouring complex mersenne transforms. Computational cost is not considered primarily, instead it is the possibility of errorless high quality audio signal processing that is being investigated.

As can be seen, in the early development of these techniques people were mainly looking for 'cheap' convolution and therefore concentrated on transforms that could be calculated without multiplication (using a simple primitive element). Nowadays, however, computation is much cheaper, and in this project more flexible transforms are considered. The goal now is errorless high quality (24 bit) audio signal processing, without worrying too much about the computational cost. A high quality lossless audio processing implementation is developed in MATLAB and implementations of the CMNT and overlap-add convolution are presented, based on this framework.

Chapter 2
Background Theory

In this chapter the background theory relevant to the area of number theoretic transforms is presented.

2.1 Conventional Signal Processing

A more thorough discussion of these topics can be found in ([16], [17],[18],[19]) or any good book on digital signal processing. They are summarised here for completeness.

2.1.1 The Fourier Transform and Convolution

There are two main types of convolution, linear and circular. The linear convolution of an N-point sequence $\{x_n\}$ and an M-point impulse response $\{h_m\}$ is denoted $y_n = x_n * h_n$ and is defined as

$$y_i = \sum_{j=0}^{M-1} h_j x_{i-j} = \sum_{j=0}^{N-1} x_j h_{i-j} \text{ for } 0 \leq i \leq N+M-1$$

with the output sequence having length $N+M-1$. The circular convolution of two M-point sequences $\{x_m\}$ and $\{h_m\}$ is denoted $y_n = x_n \circledast h_n$ and is defined as

$$y_i = \sum_{j=0}^{M-1} h_j x_{(i-j) \bmod M} = \sum_{j=0}^{M-1} x_j h_{(i-j) \bmod M} \text{ for } 0 \leq i \leq M-1$$

with the output sequence having length M. The difference is the modulo in the indices in circular convolution, which means the sequence 'wraps around' at it's end points; effectively it is as if the sequence were periodic.

The Fourier Transform, $\hat{x}_k$ of a discrete n-point signal x_j is defined as

$$\hat{x}_k = \sum_{j=0}^{n-1} x_j e^{\frac{2\pi ijk}{n}}$$

There are several important properties of the Fourier Transform (FT) that cause it to be an invaluable tool in almost all modern signal processing.

- *Linearity:*

  The FT is a linear operation.
- *Inverse:*

  The original signal can be recovered from it's FT as $x_j = \frac{1}{n}\sum_{k=0}^{n-1} \hat{x}_k e^{-\frac{2\pi ijk}{n}}$.
- *Convolution Theorem:*

  This relates circular convolution in the time domain to multiplication in the frequency domain (and vice-versa) by $F[f \circledast g] = F[f] \cdot F[g]$ where F is the FT operator and f, g are functions.
- *Frequency Decomposition:*

  The components of the FT give the values of the spectral components at discrete frequencies (up to the Nyquist limit).

- *Efficient Calculation:*

  The direct calculation of the FT from the equation above is an $O\left(N^2\right)$ operation, but there exists a family of algorithms know as Fast Fourier Transform (FFT) algorithms, which allows computation of the FT in $O(N \log N)$ operations.

  For the purposes of this work, the convolution theorem and FFT are the most important properties, since these allow the convolution procedure to be reduced from an $O(N^2)$ operation[1] to an $O(N \log N)$ operation.

2.1.1.1 Complex Transform of Real Signals

The Fourier transform as defined above is a complex transform (although a purely real Fourier transform does exist and is defined similarly, but using $\cos\frac{2\pi jk}{n}$ in place of $e^{\frac{2\pi ijk}{n}}$). The kernel $e^{\frac{2\pi ijk}{n}}$ is complex, and so complex arithmetic is required throughout the computation. Obviously the input signal x_j can also be complex. However, if the signals considered are purely real, there is a time-saving 'trick' that exploits the symmetries of the Fourier transform to transform two real signals with one complex transform. To see this consider two purely real sequences x_i and y_j and define a complex signal $z_k = x_k + iy_k$. Then the original signals can be recovered as

$$\begin{aligned} x_i &= \frac{z_i + \overline{z_i}}{2} \\ y_i &= \frac{-j\left(z_i - \overline{z_i}\right)}{2} \end{aligned}$$

Taking the Fourier transform (denoted $\mathcal{F}()_k$)

$$\begin{aligned} \mathcal{F}(x)_k &= \frac{\mathcal{F}(z_i)_k + \mathcal{F}\left(\overline{z_i}\right)_k}{2} \\ \mathcal{F}(y)_k &= \frac{-j\left(\mathcal{F}(z_i)_k - \mathcal{F}\left(\overline{z_i}\right)_k\right)}{2} \end{aligned}$$

and noting that $\mathcal{F}\left(\overline{z_i}\right)_k = \sum_{j=0}^{n-1}\overline{z_j}e^{\frac{2\pi ijk}{n}} = \overline{\sum_{j=0}^{n-1} z_j e^{\frac{-2\pi ij(n-k)}{n}}} = \overline{\mathcal{F}\left(z_i\right)_{n-k}}$ we obtain

$$\begin{aligned} \hat{x}_k &= \frac{\hat{z}_k + \overline{\hat{z}_{n-k}}}{2} \\ \hat{y}_k &= \frac{-j\left(\hat{z}_k - \overline{\hat{z}_{n-k}}\right)}{2} \end{aligned}$$

It is then possible to apply any filtering before taking the inverse transform and recovering the original signals in a similar fashion.

2.1.1.2 The Fast Fourier Transform

There are actually many algorithms that fall under the umbrella term of 'FFT', the first and perhaps best know of which is the Cooley-Tukey [20] algorithm. This is a recursive, divide and conquer algorithm, in which the transform of a composite length $N = n_1 n_2$ is reduced to two transforms of length n_1 and n_2, reducing the computation time to $O(N \log N)$ for highly-composite N. Since it simply reduces a DFT to smaller DFT's this can be combined with other FFT methods, for example if there is a large prime factor. The simplest example of the Cooley-Tukey algorithm is the 'radix-2 decimation in time (DIT)'

[1] $2N-1$ outputs, each calculated using N complex multiplications and $N-1$ complex adds $\Rightarrow (2N-1)(N+N-1) \approx O\left(N^2\right)$

case, which can be used if the transform length N is a power of 2.

$$\begin{aligned}
\hat{x}_k &= \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i k}{N}j} = \sum_{j=0}^{\frac{N}{2}-1} x_{2j} e^{-\frac{2\pi i k}{N}2j} + \sum_{j=0}^{\frac{N}{2}-1} x_{2j+1} e^{-\frac{2\pi i k}{N}(2j+1)} \\
&= \sum_{j=0}^{n'-1} x_j^e e^{-\frac{2\pi i k}{n'}j} + e^{-\frac{2\pi i}{N}k} \sum_{j=0}^{n'-1} x_j^o e^{-\frac{2\pi i k}{n'}j} \\
&= \left\{ \begin{array}{cc} \hat{x}_k^e + e^{-\frac{2\pi i}{N}k}\hat{x}_k^o & \text{if } k < n' \\ \hat{x}_{k-n'}^e - e^{-\frac{2\pi i}{N}(k-n')}\hat{x}_{k-n'}^o & \text{if } k \geq n' \end{array} \right\}
\end{aligned}$$

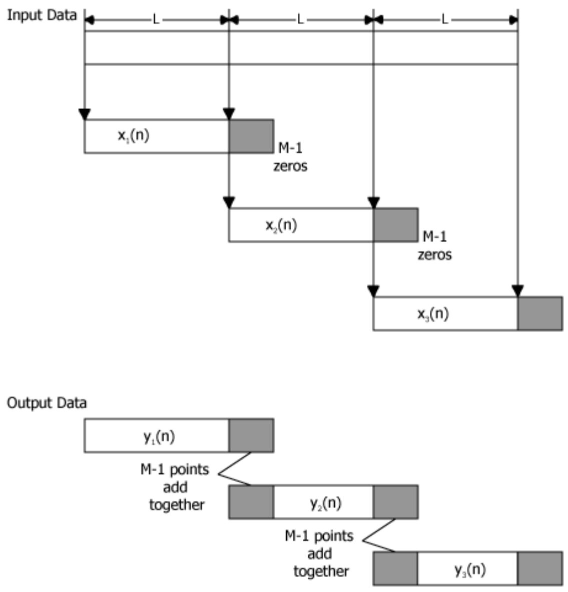
Other important FFT algorithms include the Rader's algorithm, the prime-factor algorithm and Winograd algorithm. Rader's algorithm is a method of computing prime length DFT's by re-expressing the DFT as a cyclic convolution (which is now of composite length $N-1$) which can be performed using the convolution theorem and other FFT algorithms. If necessary ($N-1$ has large prime factors) the algorithm can be applied recursively; alternatively the convolution can be zero-padded to a power of two length, and the radix-2 Cooley-Tukey algorithm discussed above can be applied. The prime-factor algorithm reduces a composite length transform $N = n_1 n_2$ into a two dimensional $n_1 \times n_2$ DFT, provided n_1 and n_2 are relatively prime. A more accurate indexing of the data is required than with the Cooley-Tukey algorithm, but there is a reduced multiplicative cost due to the absence of the 'twiddle factors'. The Winograd algorithm gives efficient equations, derived from considering the factorisation of $z^n - 1$ into cyclotomic polynomials, for small transform lengths. It can reduce the number of multiplications to $O(N)$, at the expense of an increased number of additions. This can be useful in situations where multiplication is very expensive. Another algorithm, the Rader-Brenner algorithm, also reduced multiplications while increasing additions, by modifying the Cooley-Tukey algorithm to use purely imaginary twiddle factors. This also reduces numerical stability.

As can be seen, there is a great variety of FFT algorithms, each with their own advantages and disadvantages. The most efficient solution depends on the specifics of a given problem, in terms of the data and hardware involved, often a combination of different algorithms could be most efficient.

2.1.1.3 FFT Convolution

As mentioned above, the convolution theorem, which reduces convolution in the time domain to multiplication in the frequency domain, combined with an $O(N \log N)$ FFT algorithm yields $O(N \log N)$ convolution. Care must be taken however, since multiplication in the frequency domain corresponds to circular convolution, whereas in most applications we are interested in the linear convolution. In a circular convolution, any output that would continue outside of the block in question, is wrapped around and added to the beginning of the block. This is impossible to undo. However a linear convolution can easily be performed by ensuring that the transform block is sufficiently large to contain all the required results. For example, if the two signals being convolved are of length N and M then the length of the resulting convolution, and hence the minimum length for the transform block, is $N + M - 1$. In practise, at least for audio applications, we are usually concerned with convolving a long input signal with a relatively short impulse response. In such cases it is often more practical to compute the convolution in blocks, rather than in one long transform. This is especially important in real-time applications, where it would be undesirable to have the signal effectively paused for the duration of a very long transform.

Two well known algorithms for achieving convolution in this way are the overlap-add and overlap-save algorithms. In overlap-add (OLA) a block of N samples of input data is taken and padded with zero's to a length of at least $N + M - 1$. After convolution in the frequency domain the first N samples contain the filtered data and the remaining $M - 1$ samples are the tail, or filter ringing, that would wrap around in cyclic convolution. These $N + M - 1$ samples are added to the output. The next block is then added to the output, starting where the input block started, so that it overlaps the tail of the previous block. [Fig 1][2]



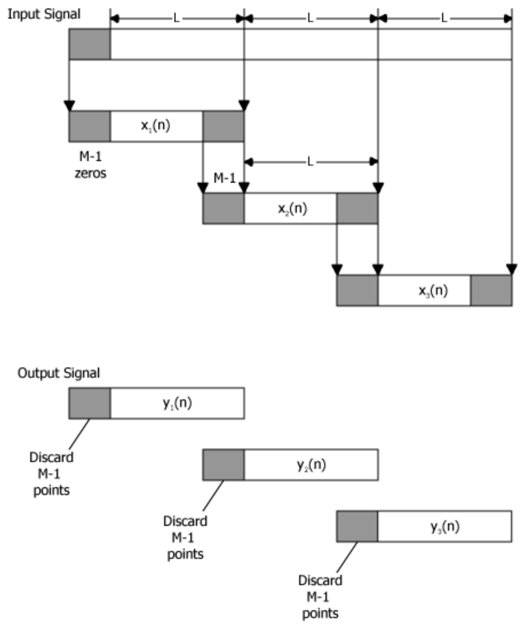
1.Overlap Add Method

In overlap-save (OLS) the signal is divided into blocks of N samples. A transform of length $N + M - 1$ is again used, but this time the transform data is filled out by reaching back to include $M - 1$ samples from the end of the previous block (for the first block zero's are used). The first $M - 1$ samples of the convolved signal are then discarded, since they contain the cyclic convolution aliasing, and the N samples corresponding to the original block are used as the output, since they already contain the tail from the previous block. [Fig 2]

2.1.2 FIR Filtering

Many important physical system are linear time-invarient or LTI systems. Such systems can be characterised by their impulse response (response to a single spike) since any input signal can be written as a linear superposition of single spikes. To simulate the action of such as system, a filter can be used. There are two main types of filter process, finite impulse response (FIR) and infinite impulse response (IIR). FIR filters consist of series of delays, after each of which the signal is multiplied by a filter coefficient and added to the output. The filter coefficients are the impulse response of the system in question, and the filtering operation is nothing less than convolution. These filters are simpler in terms of analysis, but can prove too costly to implement for very long impulse responses. IIR filters have a recursive structure, incorporating a feedback loop from the output back to the input. This means it is possible to simulate an infinite impulse response (hence the name) with a finite number of coefficients,

[2] OLA/OLS diagrams by Douglas Jones, reproduced from http://cnx.rice.edu/content/m12022/latest/ under creative commons license.



2.Overlap Save Method

but they are harder to design than FIR filters and can become unstable. They are also more sensitive to errors in the coefficients, since these are magnified by the recursive structure.

In audio, although IIR filters can be useful to implement effects such as reverberation very efficiently (due to their use of feedback), FIR filters are generally satisfactory, since any realistic impulse response is likely to be of finite length. This is why the FFT convolution methods described above are of interest as since the FIR filtering operation is a convolution, these methods allow faster calculation of the output of a FIR LTI system.

2.1.3 Errors

Despite the common misconception that digital audio is errorless, there are in fact a number of sources of inaccuracy that manifest themselves in the digital domain. Neglecting quantization effects, a thorough discussion of which is beyond the scope of this report, errors are still introduced within the digital signal path [21] [15]. In a computer each data point is stored in a finite word within the computer. This leads to small errors in almost all arithmetic, caused by rounding, truncation and overflow.

Rounding occurs when the result of a computation is too long to fit in the available word length. The result is either automatically rounded down by discarding the excess least significant bits (truncation), or rounded to the nearest number representable in the word length. This type of error also occurs when trying to represent some numbers. For example the base of the natural logarithm e is an irrational number, that can only be represented approximately within any finite word. This number is obviously very important in the FFT methods discussed above, and introduction of such errors is clearly undesirable.

Overflow errors are more serious, and occur when the result of a calculation is too big (numerically) to be represented in the word length and number representation (fixed point, floating point etc) being used. In practise, care must be taken in the design and implementation of filters to ensure that overflow does not occur.

Overflow and rounding are often the result of intermediate calculations in digital filters, despite the fact that the output is known to be representable in the available word size. One approach to solve this problem is to perform the computation in a mathematical structure known as a Finite Field. Within a finite field structure both addition and multiplication are *closed*, meaning there exists an exact result, so rounding errors are eliminated. This also means that overflows, in the traditional sense, are not a problem. Provided the output is representable in the number of bits available, the result will always be exact.

2.2 Number Theory of Finite Fields

This section covers the mathematical background required for developing the theory of number theoretic transforms. More information can be found in [22], [23], [24] or other good books on abstract algebra and number theory. [25] and [26] also provide a useful quick reference for these, and many other topics.

2.2.1 Definition of a Field

A *field* is a mathematical structure $(F, +, *)$ consisting of a set of objects, F, and two binary operations on members of that set. In order for such a structure to be a field the following axioms must hold.

- *Closure:*

  For all a, b belonging to F, both $a + b$ and $a * b$ belong to F.
- *Associativity:*

  For all a, b, c in F, $a + (b + c) = (a + b) + c$ and $a * (b * c) = (a * b) * c$.
- *Commutativity:*

  For all a, b belonging to F, $a + b = b + a$ and $a * b = b * a$.
- *Identity:*

  There exists an element 0 in F, such that for all a belonging to F, $a + 0 = a$.
  There exists an element 1 in F, different from 0, such that for all a belonging to F, $a * 1 = a$.
- *Inverses:*

  For every a belonging to F, there exists an element $-a$ in F, such that $a + (-a) = 0$.
  For every $a \neq 0$ belonging to F, there exists an element a^{-1} in F, such that $a * a^{-1} = 1$.
- *Distributivity (of $*$ over $+$):*

  For all a, b, c, belonging to F, $a * (b + c) = (a * b) + (a * c)$.

These conditions provide a structure in which addition, subtraction, multiplication and division can be performed on different objects, while maintaining our intuition about the operations from the ordinary numbers. Indeed most familiar number systems ($\mathbb{R}, \mathbb{C}, \mathbb{Q}$, respectively the real, complex and rational numbers) are themselves fields. If the set F has finite order (contains a finite number of elements) then the field is known as a *finite field* or *Galois field*[3].

There is an elementary theorem in field theory [22] which states that any finite field must be of prime order. Without this property, the existence of multiplicative inverses fails, resulting in a structure which

[3] After Évariste Galois (1811-1832), a French mathematician, who pioneered the topic of Group Theory while working on the solvability of polynomial equations.

is no-longer a field, but which is known as a *ring*. Another theorem [22] states that a finite field of p elements is isomorphic (a strictly defined mathematical concept meaning 'structurally equivalent to') to $\mathbb{Z}_p$ (the set of integers, with operations modulo p).

2.2.2 Modulo Arithmetic

A direct consequence of the theorem stated above, is that when considering finite fields it is sufficient to study sets $\mathbb{Z}_p$, with p prime. It is therefore useful to better define this structure.

Two numbers a and b are said to be *congruent modulo p* if their difference is divisible by p. This is the same as saying they both have the same remainder on division by p. This defines an *equivalence relation* on the integers, since the relation 'a is congruent to b mod p' satisfies the properties of transitivity, symmetry and reflexivity. Formally, the set $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ is the set of all equivalence classes of the integers under this equivalence relation. However, this formal definition does not provide much intuition as to how the structure behaves. A simpler way of thinking about it is as 'clock arithmetic', where the results 'wrap around' from p to 0. For example if $p = 5$, $2 + 3 = 5 = 0 \bmod 5$ and $2 \times 3 = 6 = 1 \bmod 5$.

The structure is made clearer by considering the multiplication table.

mod 7 (x)	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Multiplication Table for p=7

It can clearly be seen that the property of closure is satisfied. Compare this with the following multiplication table for p=8.

mod 8 (x)	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Multiplication table for p=8

It can be seen that closure is still satisfied, however in this case it can be seen that multiplicative inverses do not exist for 2,4 and 6 [there is no 1 in the corresponding rows]. This shows the structure is a ring and not a field, as discussed above. Contrast this with a multiplication table for normal integer arithmetic. It can be seen that the majority of the results overflow.

*	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	4	6	8	10	12	14	16
3	0	3	6	9	12	15	18	21	24
4	0	4	8	12	16	20	24	28	32
5	0	5	10	15	20	25	30	35	40
6	0	6	12	18	24	30	36	42	48
7	0	7	14	21	28	35	42	49	56
8	0	8	16	24	32	40	48	56	64

Multiplcation Table for 3-bit Multiplication

Theorem 1 (Fermat's Little Theorem) *Let p be a prime number and let a be any number with $a \neq 0 \bmod p$. Then $a^{p-1} \equiv 1 \bmod p$.*

Rearranging the above result gives $a \cdot a^{p-2} \equiv 1 \bmod p$, which provides (by definition) the multiplicative inverse of a, $a^{-1} = a^{p-2}$. This allows division within the finite field, since division is nothing more than multiplication by an element's multiplicative inverse.

2.2.3 Quadratic Residues, the Legendre Symbol and Roots of Unity

A number a is called a *quadratic residue* modulo q if there exists an integer x such that

$$x^2 \equiv a \bmod q$$

Otherwise it is called a *quadratic nonresidue*. So if a number is a quadratic residue modulo q it has a square root (in modulo q arithmetic).

The Legendre symbol [23], [25] $\left(\frac{a}{b}\right)$ is defined as

$$\left(\frac{a}{q}\right) = \begin{cases} 0 & \text{if } q \text{ divides } a \\ 1 & \text{if } a \text{ is a quadratic residue } \bmod q \\ -1 & \text{if } a \text{ is a quadratic non-residue } \bmod q \end{cases}$$

This symbol has many interesting properties, one of which, originally proved by Euler[4], is

$$\left(\frac{-1}{q}\right) = (-1)^{\frac{p-1}{2}} = \begin{matrix} 1 & \text{if } q \equiv 1 \bmod 4 \\ -1 & \text{if } q \equiv 3 \bmod 4 \end{matrix}$$

In any number system an element r is an d^{th} *root of unity* if $r^d = 1$ and a *primitive* d^{th} *root of unity* if, in addition, d is the smallest integer for which this is true. The d^{th} roots of unity form a cyclic (generated by a single element) subgroup (under multiplication) of order d, and a generator of this group is a primitive d^{th} root of unity.

2.2.4 Field Extensions

An extension of a field F is a field E which contains F as a subfield. The extension field E can be thought of as a vector space over F, with the vector addition being the field addition on E and scalar multiplication being the restriction of vector multiplication. An alternative representation is the polynomial viewpoint. In this, the field of polynomials in a variable x, with coefficients in F (denoted $F[x]$) is considered. If $g(x)$ is a monic, irreducible polynomial with coefficients in F, then an algebraic extension field can be defined as the quotient ring of the field $F[x]$ over the maximal principal ideal generated by the polynomial $g(x)$, $E = F[x]/(g(x))$. Addition in the field is polynomial addition, and multiplication in the field is polynomial multiplication modulo $g(x)$.

In the finite case, which is of interest here, this results in fields of order q^n where n is the order of $g(x)$ (highest power of x appearing). The specific case of interest for the rest of this work is the field extension $GF(q^2)$. If $q \equiv 3 \bmod 4$ then $\left(\frac{-1}{q}\right) = -1$ from above (Section 2.2.3), and so -1 is a quadratic nonresidue of q. This means $x^2 \equiv -1 \bmod q$ has no solution and therefore $x^2 + 1$ has no roots. Since any factorization would result in linear factors, and by definition a linear factor must have a root, it is clear that $x^2 + 1$ is irreducible in $GF(q)$. It is therefore possible to define $GF(q^2) = GF(q)[x]/\left(x^2 + 1\right)$. Now $GF(q)[x]/\left(x^2 + 1\right) = \{a + bx | a, b \in GF(q)\}$ since any x^2 terms reduce to -1 modulo $x^2 + 1$, and similarly higher terms are reduced by the modulo. It can be seen that x is a root of the polynomial, since $x^2 + 1 = 0 \bmod (x^2 + 1)$, so the field can also be thought of in the more familiar form $GF(q^2) = \{a + \hat{\imath}b | a, b \in GF(q)\}$ with $\hat{\imath}$ a root of $x^2 + 1$.

This extension field has an interesting property, namely that multiplication is analogous to complex multiplication in our more familiar number systems. This can be seen to follow quite clearly from the

[4] Leonhard Euler (1707-1783), a Swiss mathematician and physicist, widely considered to be one of the greatest mathematicians who ever lived.

fact that $\hat{\imath}^2 = -1$ and so

$$\begin{aligned}(a + \hat{\imath}b) \pm (c + \hat{\imath}d) &= (a \pm c) + (b \pm d)\hat{\imath} \\ (a + \hat{\imath}b)(c + \hat{\imath}d) &= ac - bd + (bc + ad)\hat{\imath}\end{aligned}$$

Chapter 3
Number Theoretic Transforms

In this chapter the theory of Number Theoretic Transforms is presented. There are a wide variety of notations used by different groups. For example the transform presented here as the Complex Mersenne Number Transform (CMNT) is known to those working in the area of large integer multiplication for the purpose of computing digits of π, as the Fast Galois Transform (FGT). This is just one confusing example of many that the author has found. For consistency, here Number Theoretic Transform (NTT) shall refer to the generalised transform, independent of ring, field or particular root of unity. Some specific cases, over specific types of field are defined and considered below.

3.1 Generalised Number Theoretic Transform

The generalised number theoretic transform of a sequence $a_k \in F$ where F is a field of order n is

$$A_k = \sum_{j=0}^{d-1} a_j r^{jk}$$

where d divides $n-1$, and r is a primitive d^{th} root of unity. The inverse transform is given by

$$a_k = d^{-1} \sum_{j=0}^{d-1} A_j r^{-jk}$$

As can be seen, this is similar in appearance to the Fourier transform, in which the place of r is taken by $e^{\frac{2\pi i}{d}}$, a primitive d^{th} root of unity in the field of complex numbers. In addition, there is the extra constraint that the transform length must divide the field order. This would have no meaning in the case of the FT since the order of the field of complex numbers is infinite; this is why error always arises in a finite computer representation. In fact, the condition that F is a field is not strictly required. The transform can be defined over any ring, and the inverse is well defined provided d^{-1} exists in the ring. However, matters are simplified if fields are considered, since all transform lengths for which a suitable root of unity exists can be considered.

The condition that d divides $n-1$ is a direct consequence of r being a primitive d^{th} root of unity, rather than being an additional condition in its own right. This is because as mentioned above (Section 2.2.3) the roots of unity of order d form a cyclic subgroup of order d under multiplication. Now the multiplicative group of the field is $F^* = F/\{0\}$ (the non-zero elements of F, since the element 0 does not have a multiplicative inverse) which has order $n-1$. Lagrange's theorem, an important result in group theory [22], states that the order of a subgroup divides the order of the larger group, hence $d|n-1$.

The NTT shares many of the properties of the FFT considered previously (Section 2.1.1), namely linearity, the existence of an inverse, the convolution property and in some cases, the potential for efficient calculation (Appendix A). One property it does not share, however, is frequency decomposition. Whilst the FT gives useful frequency information (indeed the transformed signal is often referred to as in the *frequency domain*), the NTT conveys no useful information about the signal in the transform domain. This is because, while the FT can be thought of as a decomposition in terms of basis vectors of trigonometric functions of different frequencies, the basis vectors in the NTT case have no underlying structure and, while orthogonal, are apparently random.

Another significant difference between the NTT and the FFT is the limitation on transforms lengths. With the FFT there is a fixed relationship between the transform length and the root used in the transform; this relationship is very clear and the root is easy to calculate ($e^{\frac{2\pi i}{L}}$) for any transform length. With the NTT there is a similar relationship, but there is the extra variable of the modulus used, which limits the available transform lengths. There is also no longer a simple formula that can be written down for the root of a given length. For example, as discussed above (Section 1), historically NTT's have been of interest in signal processing to reduce the computational cost of computation. This is achieved by choosing a root which allows very easy computation of the transform (often solely by shifts and adds), however the price for this is limited transform lengths that are frequently too short to be useful. In the next section some examples are presented to give an idea of what is possible.

Obviously there are a great range of possible moduli to consider, even if we restrict ourselves to prime moduli (to ensure the number system over which we work is a field). Even this is not strictly necessary, the properties of the transform hold over any ring, with the condition that for an inverse to exist for a transform of length d, d^{-1} must exist within the ring. This presents a whole range of compound moduli with interesting properties, however the analysis becomes more involved. For the purposes of this work, only Fermat and Mersenne prime moduli will be considered, since historically these have been the most promising for the field of signal processing, since their form as a number one away from a power of 2, means modulo arithmetic is particularly easy to implement in a binary system such as a computer, or digital signal processing hardware.

3.2 Specific Examples

In general prime moduli are expensive to implement due to division required in the modulo operation (recall modulo gives the remainder on division). There are some prime moduli, however, that lend themselves very well to implementation on a binary computer system. For example numbers of the form $2^q \pm 1$ are easily implemented, as explained below.

3.2.1 Fermat Number Transform

A *Fermat number*[5] is a number of the form $F_n = 2^{2^n} + 1$ where n is a positive integer, with such a number being called a *Fermat prime* if it is, in addition, prime. Fermat postulated that all such numbers were prime, but since the advent of computers this has been proved wrong and to date the only known Fermat primes are $F_0, .., F_4$. The Fermat Number Transform (FNT) is the NTT where the field order is a Fermat prime.

Due to the structure of the Fermat prime, FNT's have several desirable properties. They are relatively easy to implement in binary computer arithmetic [27], being only slightly more expensive then standard two's complement addition (Section 4.1.1.1). Since $F_n - 1 = 2^{2^n}$ they yield highly composite transform lengths, that are amenable to a simple radix-2 Cooley-Tukey transform algorithm. There are also some choices of root that yield particularly efficient calculation. For $r = 2$ transforms of length 2^{n+1} are possible[6], and multiplication by powers of 2 corresponds to simple bit shifts in binary arithmetic, eliminating the need for full multiplication. Similarly for $r = \sqrt{2}$ transforms of length 2^{n+2} are possible. In the finite field $r = \sqrt{2}$ means $r^2 = 2 \bmod F_n$, which solves to give $r = 2^{2^{n-2}}\left(2^{2^{n-1}} - 1\right) = 2^{3\cdot 2^{n-2}} - 2^{2^{n-2}}$, so again explicit multiplication can be avoided, this time with two shifts and an addition.

[5] After Pierre de Fermat (1601-1665), a French lawyer and mathemetician, credited with, among other things, the development of modern calculus and widely considered the founder of modern number theory.

[6] 2 is a $(2^{n+1})^{th}$ root of unity, since $2^{2^{n+1}} = \left(2^{2^n}\right)^2 = (-1)^2 \bmod F_n = 1 \bmod F_n$. So its order must divide 2^{n+1}(and therefore be of the form 2^k). Now if $2^{2^k} = 1 \bmod F_n$ for some $k < n+1$ then $2^{2^{k+j}} = 1 \bmod F_n$ for all $j > 0$. But $2^{2^n} = -1 \bmod F_n$ and so we have a contradiction, the order of the element 2 is 2^{n+1} and hence it is a primitive root of unity.

As mentioned earlier (Section 1) most work in this area has been aimed at trying to minimise the number of multiplications required to perform a convolution, and hence FNT's have been considered very promising and studied extensively. Unfortunately there are also some major disadvantages of using this modulus. Firstly, a $2^n + 1$ bit word length is required. This results in a significant redundancy in the representation since the extra bit word length provides an addition $2^{2^n} - 1$ values which will not be used. Further redundancy results since available word lengths in existing computers and hardware are nearly always powers of two, hence in a practical implementation half the word (the most significant half and therefore the great majority of possible values) is redundant. More importantly the largest known Fermat prime, $F_4 = 2^{16} + 1 = 65537$ (approximately 16 bits) gives a dynamic range that is too small to carry even a single high quality audio signal, which usually requires a dynamic range of 24 bits, let alone the convolution of two such signals. Finally, of the known Fermat primes, only $F_0 \equiv 3 \bmod 4$ as required (Section 2.2.4) for the existence of an extension field analogous to the complex numbers.

3.2.2 Mersenne Number Transform

A *Mersenne number*[7] is a number of the form $M_n = 2^n - 1$ where n is a positive integer, with such a number being called a *Mersenne prime* if it is, in addition, prime. For all Mersenne primes, the power n is prime, but the converse, that all Mersenne numbers with n prime are prime, is not true. There are currently 42 known Mersenne primes, the search for which has been revolutionised by the advent of the digital computer and, more recently, the internet[8]. The Mersenne Number Transform (MNT) is the NTT where the field order is a Mersenne prime.

Arithmetic modulo a Mersenne prime is particularly convenient to implement, since arithmetic modulo $2^p - 1$ is one's complement arithmetic with an p bit word length (Section 4.1.1.1). The cases usually considered (Section 1), due to their efficiency in computation, are $r = 2$ and $r = -2$. In the first case it can be clearly seen that

$$2^p = (2^p - 1) + 1 = M_p + 1 \equiv 1 \bmod M_p$$

is a p^{th} root of unity. It can be seen from the above that the order of the element 2 (let $k = O(2)$) must divide p, but since p is prime we have that $k = p$ and so the root is also primitive, therefore yielding a transform of length p. Fermat's Little Theorem (Section 1), which states

$$a^p - a \equiv 0 \bmod p \text{ for } a \neq 0 \bmod p$$

confirms that p is a valid transform length since it divides $M_p - 1 = 2^p - 2$. In the second case, $r = -2$ is a primitive $(2p)^{th}$ root of unity and hence yields a transform of length $2p$. To see this first note that $(-2)^p = -(2^p) \equiv -1 \bmod M_p$, since p is prime and therefore odd (for $p > 2$) and $2^p = 1 \bmod M_p$ from above. Then

$$(-2)^{2p} = (-2)^p (-2)^p \equiv (-1)(-1) \bmod M_p \equiv 1 \bmod M_p$$

and as before, the order of element -2 (let $k = O(-2)$) must divide $2p$, but since p is prime this implies $k = 1, 2, p$ or $2p$. Clearly $(-2)^k \neq 1 \bmod M_p$ for all but $k = 2p$, and so -2 is a primitive $(2p)^{th}$ root of unity. It can be seen that while the modulo arithmetic in the case of the MNT is much more convenient to implement in a computer than in the FNT, the transform lengths are far more restrictive and do not lend themselves to efficient FFT-type algorithms. However, as before, there are special cases for which the transform can be calculated without multiplications, using only adds and bit rotations of the word (as well as negation in the case of $r = -2$). There is also the useful feature that for any Mersenne prime with $p > 2$,

$$M_p = 2^p - 1 = 4 \cdot 2^{p-2} - 1 \equiv -1 \bmod 4 \equiv 3 \bmod 4$$

which is the requirement (Section 2.2.4) for the existence of an extension field analogous to the complex

[7] After Marin Mersenne (1588-1648), a French theologian, philosopher, mathematician and music theorist.

[8] GIMPS - The Great Internet Mersenne Prime Search (http://www.mersenne.org/)

numbers.

3.2.3 Pseudo-Complex Mersenne

The pseudo-Complex Mersenne Number Transform (pCMNT), is so called because it is not a true complex transform (ie a transform over a complex field). Instead it is a the normal MNT, but with the root of unity taken to be a complex number. This is what Nussbaumer [7] called the Complex Mersenne Transform. Mathematically this is somewhat of an artificial construct, since the field in question is not complex, but is being extended to a complex field in an ad-hoc manner outside the structure of the transform. This 'trick' yields a maximum transform length of $8p$. This is not to be confused with the Complex pseudo-Mersenne Number Transform, which is the MNT over a field of order a Mersenne number that is not prime.

3.3 The Complex Mersenne Number Transform

Reed and Truong first proposed the Complex Mersenne Number Transform (CMNT) in [4]. In this they exploit the fact (Section 3.2.2) that for any Mersenne prime $q = M_p \equiv 3 \bmod 4$, and hence there exists an extension field $GF(q^2)$ analogous to the complex numbers with operations modulo q. It is therefore possible to define a transform over this complex field. Since the operations remain modulo q it shares the property of the normal Mersenne number transform that the arithmetic is very convenient to implement (as one's complement arithmetic on a p bit word, where $q = 2^p - 1$). However now the order of the group is q^2 rather then q and so the possible transform lengths are factors of $q^2 - 1 = (2^p - 1)^2 - 1 = \left(2.2^{2p} - 2.2^p + 1\right) - 1 = 2^{p+1}\left(2^p - 1\right)$ which is highly composite (contains a factor 2^{p+1}) and so amenable to radix-2 type transforms. The longer transform lengths provide greater flexibility, albeit at increased computational expense. Only transforms of length q or $2q$ (with roots $r = 2$, $r = -2$ respectively) can be computed without multiplications (using only bit shifts and negations).

3.3.1 Finding Primitive Roots of Unity

Given a desired transform length, it is a non-trivial task to find the primitive root of unity of that order. In [28] a method is developed for doing so and another algorithm is presented in [4]. The algorithm proved in [4] is as follows. A primitive $\left(2^{p+1}\right)^{th}$ root of unity α in the field $GF\left(q^2\right)$ with $q = 2^p - 1$ is given by $\alpha = a + ib$ where

$$\begin{aligned} a &= \pm 2^{2^{p-2}} \bmod \left(2^p - 1\right) \\ b &= \pm \left(-3\right)^{2^{p-2}} \bmod \left(2^p - 1\right) \end{aligned}$$

Given this element, it is clear that if d divides 2^{p+1} then $r = \alpha^{\frac{2^{p+1}}{d}}$ is a primitive d^{th} root of unity in $GF\left(q^2\right)$ since $r^d = \alpha^{2^{p+1}} \equiv 1 \bmod \left(2^p - 1\right)$ by definition of α, and there is no $k < d$ for which this is true since α is a primitive $\left(2^{p+1}\right)^{th}$ root. In this way roots for a range of transform lengths can be quickly found.

3.3.2 Fast Algorithm for Computing Powers of a Primitive Root

In [29] two theorems are proved giving congruences for primitive roots in $GF\left(q^2\right)$. These congruences can be used to reduce the number of computations needed to calculate the powers r^k as required for the transform. The first theorem provides a reduction in the number of calculations needed to obtain the powers r^k for $0 \leq k \leq d - 1$ to $\left(\frac{d}{4}\right) - 1$. The second theorem allows for a reduction to $\left(\frac{d}{8}\right) - 1$

computations, although the procedure is a little more complicated and relies on the use of some tabled values in the computation. The first of these theorems is presented below:

Theorem 2 *If $d = 2^k, 3 \leq k \leq p+1$ and r is a primitive d^{th} root of unity in GF $\left(q^2\right)$ with $q = 2^p - 1$ a Mersenne prime, then*

$$\begin{aligned} r^{n\cdot 2^{k-2}+m} &\equiv (-1)^n \cdot \overline{r^{n\cdot 2^{k-2}-m}} \bmod q \textit{ for } 3 \leq k \leq p \\ r^{n\cdot 2^{p-1}+m} &\equiv (-1)^{n+m} \cdot \overline{r^{n\cdot 2^{p-1}-m}} \bmod q \textit{ for } k = p+1 \end{aligned}$$

where $n = 1, 2, 3$, $m = 1, 2, ..., 2^{k-2}$.

3.4 Application to Audio Processing

This work is concerned with application of the transforms discussed above to audio signal processing. This area presents some specific processing challenges. In the following discussion let h_n be an impulse response, x_n an input signal and y_n the result of the convolution, $y_n = h_n * x_n$, and M the modulus of the field used for the transform.

3.4.1 Dynamic Range

In general terms, dynamic range describes the ratio of the smallest and largest possible values of a changeable quantity. In digital audio, this is a value that is a direct function of word length, with the dynamic range in dB equal to approximately six times the number of bits used to represent each sample of the signal.

In a finite field it is important to ensure that the result of the convolution of the two signals can be represented within the word length used for the transform. If this is not the case overflow will occur, which will result in information being lost, and the calculation being incorrect. For example if the result of an operation within the finite field is some number N which is greater than the modulus, then $N = kq + r$ where q is the modulus used and k is some positive integer. But within the finite field structure this is of course reduced to r, so the details of the number of times the modulus divided the number (k) is lost. This is irreversible and undetectable, since without knowing the right answer, there is no indication that this reduction has occurred. It is therefore imperative to ensure that the situation does not occur, by monitoring the magnitude of the input signals.

It is of course necessary for the input signals to be bounded; that is $|h_n|_{\max}$, $|x_n|_{\max}$ and $|y_n|_{\max} \leq \frac{M}{2}$, in order that they can even be represented within the finite field system. Note that this is assuming the input signals are signed, so the $\frac{M}{2}$ is required, since both the positive and negative values of the signal need to be mapped into the M values of the finite field. However to avoid the wrapping of y_n described above, stricter boundaries are required. When both sequences are unknown the following must be satisfied to ensure no overflow occurs

$$|y_n|_{\max} \leq N\,|x_n|_{\max}\,|h_n|_{\max} < \frac{M}{2} \qquad \text{(Condition 1)}$$

where N is the transform length. This covers the absolute worst case, where both input and impulse response are saturated, with the same sign, at their maximum values. Obviously in most real applications this is not the case, and so this boundary is usually too strict. If one of the signals, for example the impulse response, is known, the following criteria ensures no overflow

$$|y_n| \leq |x_n|_{\max} \sum_{k=0}^{L_h-1} |h_k| < \frac{M}{2} \qquad \text{(Condition 2)}$$

where L_h is the length of the impulse response. This gives the output of the maximum value of the input signal, which though linearity of convolution, is the maximum value of the output. This estimate is still quite conservative however, since it assumes that the input signal is saturated at the maximum values, with signs matching the signs of the impulse response.

Converting to word length, by taking the logarithm base 2 of the above conditions yields the following

$$\log_2 N + \log_2 |h_n|_{\max} + \log_2 |x_n|_{\max} < \log_2 M - 1 \qquad \text{(Condition 1b)}$$

$$\log_2 \left(\sum_{k=0}^{L_h - 1} |h_k| \right) + \log_2 |x_n|_{\max} < \log_2 M - 1 \qquad \text{(Condition 2b)}$$

Assuming 24 bit signals, the following is obtained

$$49 + \log_2 N < \log_2 M \qquad \text{(Condition 1c)}$$

$$25 + \log_2 \left(\sum_{k=0}^{L_h - 1} |h_k| \right) < \log_2 M \qquad \text{(Condition 2c)}$$

Note that where possible, the second family of conditions should be used (2a, 2b and 2c) since there will generally be quite a large difference between these and the first. The first set of conditions are very strict, to the point of being almost impractical, since the 'worst case' they assume will never occur in practice in an audio signal. Indeed most real impulse responses, for example that of a room, decay rapidly resulting in a tail of very low values.

Even when the possibility of overflow within the finite field had been eliminated, it is necessary to map the signal back from the finite field into 24 bit integers. Even if the above conditions are met, it is certainly possible that the output of the convolution is too large to be represented within the 24 bits available for the output signal. There are several approaches to this problem. Conditions such as those above can be developed, to ensure that the output of the convolution fits within the stricter limit of a 24 bit output word. Alternatively the output in the finite field can be scaled to fit into the 24 bit word, or the output will be clipped when converted into the 24 bit signal.

3.4.2 Choice of Modulus

Modern high quality digital audio tends to be a 24 bit digital signal at sampling frequencies up to 96kHz. Clearly this requirement already precludes use of the FNT, since the largest known Fermat prime is $F_4 = 65537$ (just over 16 bits) which is too small to hold even single signal, let alone compute the convolution of two such signals.

The Complex Mersenne Number Transform was chosen as the most promising transform for high quality audio processing. Since this work is concerned less with the elimination of multiplications and more with flexibility of the transform for audio applications, it was felt the selection of transform lengths available with the CMNT was more promising than that of the other Mersenne prime based transforms. This is because longer, but highly composite transform lengths are available, which allow for implementation of FFT type algorithms. The computational convenience of the Mersenne modulo was also seen as a major advantage. Finally, since it is a complex transform, some of the increased computational expense is offset against the fact that two real input signals can be processed at the same time, in the same way that a complex Fourier transform can compute the transform of two real sequences (Section 2.1.1.1). This would allow the simultaneous computations of two blocks from a block convolution algorithm, or alternatively the simultaneous computation of two different audio channels.

This leaves only the choice of which Mersenne prime to use. The first 10 Mersenne primes are $M_2, M_3, M_5, M_7, M_{13}, M_{17}, M_{19}, M_{31}, M_{61}$ and M_{89}. The output of a lossless multiplication of two

24 bit words is $24 + 24 = 48$ bits long, and so the smallest prime we can can consider is M_{61}. Using condition $1c$ from the previous section we obtain $\log_2 N < 61 - 49 = 12$ and so transform lengths up to $2^{12} = 4096$ are possible. As described above this is quite a conservative 'worst case' and so it is likely that in practice much longer transform lengths may be possible. This seems likely to be sufficient for a practical audio application, and M_{61} has the added advantage that it is conveniently close to 64 bit word length, which is commonly implemented in computer hardware and software. Modular arithmetic modulo M_{61} can therefore be implemented within a 64 bit word, with little overhead in terms of unused bits.

Chapter 4
MATLAB Implementation

As discussed above, (Section 3.4.2), it was felt that the complex mersenne number transform provided the most promising candidate for processing high quality audio losslessly. An implementation was therefore developed, with the goal of creating a framework in which calculations in this field could easily be performed, to test the feasibility of such a system. MATLAB was chosen as a platform, since it is ideal for rapidly developing and testing new algorithms, and it has a great deal of existing signal processing functionality, for example, it has built in audio file access libraries. This chapter describes this implementation. The first section deals with the core finite field framework that was coded in C to overload the standard MATLAB operations for the `uint64` datatype. The second section describes MATLAB code that was written using this framework, to investigate the feasibility of the NTT for audio processing.

4.1 Finite Field Framework

4.1.1 Number Representations

There are two major number representations in use today [25],[30]. The first, and simplest, is 'fixed point' arithmetic. In this scheme, a number is represented in it's binary form, with the decimal point *fixed* in a certain location. This provides representation to a fixed accuracy of numbers within a certain range, and is particularly convenient when working with integers. The other is 'floating point arithmetic', which is described below.

4.1.1.1 Signed Integer Arithmetic

There are several methods of representing signed integers in a binary word. The simplest, known as sign and magnitude, simply devotes one bit of the word to be the sign bit (ie 1 in this bit signifies a negative number, 0 a positive), and the remainder is the normal magnitude of the number. With this system the range $-2^{p-1} - 1$ to $2^{p-1} - 1$ can be represented in a p bit word. This gives $2^p - 1$ values, which can also be seen from the fact that a p bit word contains 2^p different values, and in this system the number 0 is represented by two different values. However arithmetic is difficult to implement. There are two further methods of interest, known as ones' complement and two's complement respectively. The table below illustrates the difference between these methods with the example of an 8-bit binary word.

decimal	**binary**	**sign magnitude**	**ones' complement**	**two's complement**
0	00000000	0	0	0
1	00000001	1	1	1
⋮	⋮	⋮	⋮	⋮
126	01111110	126	126	126
127	01111111	127	127	127
128	10000000	0	-127	-128
129	10000001	-1	-126	-127
⋮	⋮	⋮	⋮	⋮
254	11111110	-126	-1	-2
255	11111111	-127	0	-1

In ones'[9] complement arithmetic the negation of a number is represented as the *binary complement* (or bitwise NOT) of the number. This is obtained by switching all the bits of the word (ie $1 \rightarrow 0, 0 \rightarrow 1$). As can be seen from the table this yields the same range and number of values ($2^p - 1$) as sign and magnitude, and again there are two representations of the number 0. However arithmetic is much easier to implement, simply add the two numbers in normal binary fashion, then add any carry over from the most significant bit back to the least significant bit. However it is worth noting that this is nothing else than addition modulo $2^p - 1$. This is because the end-around carry operation is in fact modulo reduction. To see this, consider reduction modulo 256 on a 16 bit word. In this case the lower 8 bits would be the remainder, r, and the upper 8 bits would be the quotient, q. That is the number $n = q \times 2^8 + r$. The result of modulo reduction would then be simply the unmodified lower 8 bits, r. Now for modulo 255, consider that in the previous division (which is a process of subtracting 256 repeatedly until the remainder is less than 256); each time 256 was subtracted in place of the desired 255. To adjust the remainder, 1 must be added for each time 256 was subtracted. The number of times this occurred is stored in q. So in this case modulo 255 reduction is achieved by adding the carry, q, to the remainder. In a general modulo reduction scenario, this result could itself be greater than 255 and so the process would need to be repeated, but in the case of addition or two 8 bit words, it is known the overflow can only be one bit and so one carry is certainly enough.

In two's complement negation is achieved by taking the binary complement as in ones' complement, but then adding 1 to the result. Although negation is now more complicated this system has several advantages. There is now only one representation for zero; the range -2^{p-1} to $2^{p-1} - 1$ can be represented in a p bit word (2^p values). Arithmetic is also greatly simplified, normally binary addition is all that is required, without any consideration of the carry. It is for this reason particularly that two's complement is perhaps the most widely used signed integer convention in modern computing. By considering the previous argument for ones' complement, it can be seen that this is arithmetic modulo 2^p, since it is normal addition, keeping only the p least significant bits. This is precisely the remainder upon division by 2^p.

4.1.1.2 Floating Point Arithmetic

In this representation the word used to represent the number is split into a fixed-point *mantissa* and a base-2 *exponent*. The actual number represented is then given by $mantissa \times 2^{\textit{exponent}}$. The advantage of this method is that it allows a much wider range of values to be represented within a given word length than the fixed point representation. However it clearly cannot represent as many digits accurately, given the same word length, since some of the word is devoted to storing the exponent rather than actual digits of the number.

4.1.1.3 Representation in MATLAB

MATLAB represents numbers internally using the IEEE standard[31] for double precision floating point numbers. This defines a 64 bit word where 1 bit defines the sign of the number, 11 bits are used for the exponent of the number and the remaining 52 bits are used for the mantissa. Obviously this can only represent integer (fixed point) values up to 52 bits, whereas for this implementation 61 bit integers are required. MATLAB does include both signed and unsigned 64bit integer data types, although no operations are implemented for these types. It was decided to use the unsigned 64 bit integer data type (`uint64`) data type as a base and overload the standard operations (+, -, * etc.), that were in fact undefined, to implement the modulo arithmetic. Convolution algorithms and transforms could then be

[9] "Detail-oriented readers should notice the position of the apostrophe in terms like "two's complement" and "ones' complement": A two's complement number is complemented with respect to a single power of 2; while a ones' complement is complemented with respect to a long sequence of 1s."[30, Chapter 4.1]

written in the normal, familiar, MATLAB code, but using 64 bit integer data type would ensure all arithmetic was in the finite field.

Since MATLAB cannot support the integer precision required internally, it was necessary to code these overloaded operators externally. MATLAB natively supports external calls in FORTRAN and C, but in this case C was chosen as the authors preference. The unsigned 64 bit data type (`uint64`) was chosen, since sign considerations are dealt with in the external implementation. The ones' complement sign convention was used, although this is only really relevant when converting data into and out of the finite field structure. There is, of course, no need for a notion of sign within the finite field structure; all elements of the finite field are on equal footing due to the circular nature of the modulo operation. The conversion works as follows. If an incoming integer is positive, it simply keeps it's same value in the finite field. If the incoming integer is negative, it has M_{61} added to it. This is exactly the ones' complement convention described earlier (Section 4.1.1.1). Consider the example given there in which $-1 + 255 = 254$, the actual value of the signed representation. Conversion out of the finite field is similar, if the sign bit is set, M_{61} is first subtracted from the number to obtain the corresponding negative, otherwise the number is presented as is. This allows numbers in the range from $-(2^{60} - 1)$ to $(2^{60} - 1)$ to be represented within the finite field, which is more than enough for our 24 bit audio signals, although the additional overhead is required to ensure the convolution can be help in the finite field without overflowing (Section 3.4.2).

The operations were performed as normal 64 bit integer operations, with modulo reduction applied to the result. In [32], an alternative implementation is proposed, by which the prime length word required for the finite field is *sign extended* to a more convenient word length. This means the sign bit (in this case bit 61, representing the value 2^{60}) would be propagated to the high end of the word. This would ensure if the sign bit was 1, all higher order bits would also be 1, and visa-versa. This would then allow standard ones' complement arithmetic on the longer, power of two word length, to be used. This method was not used here, since it was felt that it would complicate the implementation. Since there is no ones' complement arithmetic in the ANSI C specification, (two's complement is used) the end-around carry aspect, or modulo reduction must be included in the code. It was felt it would be much harder to implement this if the sign-extension method was used, since the result of the addition of two 64 bit words is another 64 bit word and it would be hard to obtain the carry. It seemed quite convenient, from the point of view of this implementation, that there were three 'spare' bits in the word, that could be used to hold the overflow from addition in order to perform the modulo reduction of the result, and of course 6 'spare' bits in the case of multiplication where the result can be twice the word length.

4.1.2 The MEX Interface

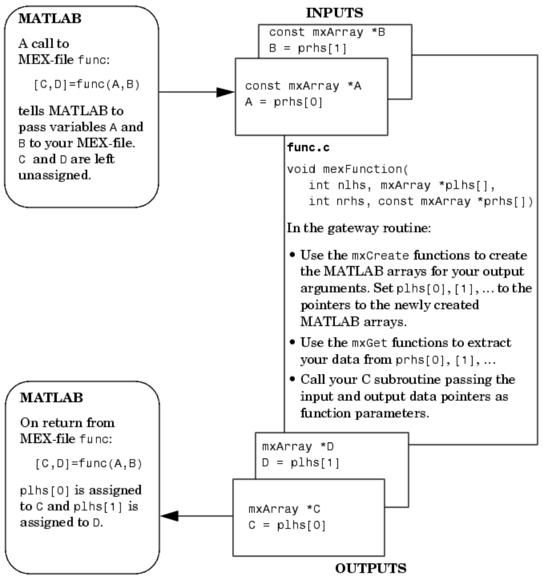
A MEX file [33] is a MATLAB callable C program; a dynamically linked subroutine that the MATLAB interpreter can automatically load and execute. Rather than provide a lengthy explanation of the interface, a brief overview will be given. For more information the reader is referred to the excellent documentation provided with MATLAB.

4.1.2.1 Outline of the Interface

To create a MEX file, a standard C file is created. However, unlike a standard C program, this file contains no main() function. Instead the program execution begins at, and is controlled by the mexFunction() function. This is what is called by MATLAB when the MEX files command is invoked, and it's definition in the file looks like

```
#include "i64.h"
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
                             int nrhs, const mxArray *prhs[])
{
}
```

The arguments are, in order, the number of left hand side (output) arguments when the MATLAB function is called, an array of pointers to the output arguments mxArrays, the number of right hand side (input) arguments and an array of pointers to the input arguments mxArrays. An mxArray is MATLAB's generic structure for holding data, which can be complex or real, scalar or array and of any MATLAB data type. All argument checking for expected numbers and types of arguments must be taken care of by the function, as must allocation of memory for the output. The function then processes the data as required, either in line or by calling a further function, before returning it, by pointing the output pointers (in the `plhs` array) to the required output data. The MEX file process is better illustrated by the following diagram [10]


MATLAB
A call to MEX-file func:
[C,D]=func(A,B)
tells MATLAB to pass variables A and B to your MEX-file. C and D are left unassigned.
INPUTS
const mxArray *B
B = prhs[1]
const mxArray *A
A = prhs[0]
func.c
void mexFunction(
int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[])
In the gateway routine:
• Use the mxCreate functions to create the MATLAB arrays for your output arguments. Set plhs[0], [1], ... to the pointers to the newly created MATLAB arrays.
• Use the mxGet functions to extract your data from prhs[0], [1], ...
• Call your C subroutine passing the input and output data pointers as function parameters.
MATLAB
On return from MEX-file func:
[C,D]=func(A,B)
plhs[0] is assigned to C and plhs[1] is assigned to D.
mxArray *D
D = plhs[1]
mxArray *C
C = plhs[0]
OUTPUTS

3.MEX Cycle

However note that in this application, since many of computations were not so complicated, they were often performed within the mexFunction gateway function, rather than in a separate function.

The MATLAB API provides a full set of routines that handle the various data types supported by MATLAB. These are broadly divided into routines beginning `mx` which are routines for creating and manipulating MATLAB data mxArray's, and routines beginning `mex` which perform operations in the calling MATLAB environment.

4.1.2.2 Using the Interface

Given a C file for a function written with the MEX interface as described above, it is compiled either from within MATLAB or from the operating system prompt using the `mex` command. The compiler used can be setup using `mex -setup`. This will scan the system for available compilers and allow

[10] Reproduced from MATLAB documentation, ©1994-2005 by The MathWorks, Inc.

the user to make a choice. There is a configuration file, on Windows it is `C:\Documents and Settings\Robin Ince\Application Data\MathWorks\MATLAB\R13\mexopts.bat`, in which various options such as the include or library paths might need to be specified to ensure error free compilation. Once compiled, the resulting file (`.dll` on Windows) can be run from the MATLAB prompt, or in scripts, provided it is in the current working directory.

4.1.3 Overloaded Operators

A useful feature of MATLAB is the ability to overload operators. This is provided to facilitate object orientated programming, by allowing users to define standard operations on user defined data types, or structures. Whenever a built-in function or operator is applied to a data type, the function applied is first looked for in the `@datatype` subdirectory within the current working directory. By placing suitable named functions, either m-files or mex files, within that directory, the standard operations, such as +, - etc. can be *overloaded* to provide different functionality. Similarly, built in MATLAB functions, such as $conj()$ can also be overloaded for different data types. In this case, it is the unsigned 64 bit integer data type (MATLAB data type uint64) that is of interest (Section 4.1.1.3), so the MEX-files to overload the operators must be placed in a directory named `@uint64`. The core operations that will be required to implement the transform and convolution algorithms in the finite field framework are

Operation	MATLAB operation	MATLAB function name
addition	$+$	plus
subtraction	$-$	minus
negation (unary minus)	$-$	uminus
scalar multiplication	$*$	times
scalar exponentiation	.^	power
complex conjugation	$conj()$	conj
real / imaginary part	real / $imag$	real / image

4.1.4 MATLAB Integration

There were some additional noteworthy points related to ensuring the framework fits smoothly in with MATLAB. The first concerns the MATLAB help system. In an M-file, MATLAB considers the first group of consecutive commented lines (lines that begin with %) immediately following the function definition to be the help section for the function. This can be obtained from the command line by `help <function name>`. For functions that overload standard operations, the help can be obtained as `help <class>/<function name>`. In the case of MEX files, if an M-file of the same name is created, containing a commented help text, this will be displayed in the same way.

It is important to note that by default, MATLAB treats any numbers entered as the double datatype. For example, if `x=[3 2];` is entered, it will create a one by two array of doubles. Therefore to enter numbers in the finite field, a conversion function is required to convert from the standard `double` datatype to the `uint64` type. There are conversion functions built in to MATLAB, but of course the standard `uint64()` conversion function does not handle sign in the required way. Unfortunately it is not possible to overload these conversion functions in the normal way, so it was necessary to give the conversion function a different name, in this case `ui64`.

It is also worth noting that when a function is called with no arguments given, it is assumed that the function is acting on a double and so in addition to the current working directory, the `@double` directory will also be searched for a matching function. This is why the getq function described below, which takes no arguments, should be placed in the `@double` directory.

Finally, the framework is made more transparent to the user by controlling the way the data type is displayed. The MATLAB display function can be overloaded like the arithmetic operations outlined earlier, but this function controls the way MATLAB displays an element of that datatype on the screen. The standard display for `uint64` shows its actual value. This is fine, but becomes confusing when the finite field is representing negative integers, since all small negative integers display as "$2.3058e + 018$" as a small negative number is very close to M_{61}. The display function was therefore overloaded to perform the sign conversion (Section 4.1.1.3) so the finite field data can be viewed in a more intuitive and familiar way.

4.1.5 Implementation

4.1.5.1 File Listing

Below is a listing of the files in the `@uint64` directory, which form the core of the installation, with a brief description of their function. Where there is a .c file and a .m file with the same name, the .c file contains the code and the .m file contains the help text. (Section 4.1.4)

i64.h This is an include file used in all the other code files. It includes definitions for important constants, as well as `typedef`'s to ease compatibility with different compilers (different compilers tend to have different notations for data types).

conj.c, conj.m Performs conjugation. Returns input argument with imaginary part negated. It can accept arrays of any dimension, since the output array is duplicated from the input array.

convert.c Converts a number in the finite field representation to a signed double. This is used for converting from the finite field domain into the 24bit signed integer signal domain. Returns a double array of same dimension (and complex or real) as the input array. The calculation is achieved by returning the finite field value unchanged, if the most significant (61^{st}) or sign bit is not set, and returns the finite field value minus M_{61} if the sign bit is set. This is the same as ones' complement arithmetic (Section 4.1.1.1).

getq.c This function simply returns the value $M_{61} = 2^{61} - 1$. (Section 4.1.4).

imag.c, imag.m Returns the imaginary part of the input, or zero if it does not have one.

plus.c, plus.m Pointwise addition of two real or complex arrays of the same dimensions. Modular reduction is performed simply with the line `(temp & M61) + (temp >> 61)`. There are four cases in the main control loop depending on the whether the input and output are real or complex.

power.c, power.m This function does pointwise exponentiation (.^ operation in MATLAB). This is implemented with right-to-left binary exponentiation [30].

times.c, times.m This is perhaps the most complicated function in the framework and implements pointwise multiplication of two arrays of the same dimensions, or of an array by a scalar. Inputs can be real or complex. The actual 64 bit multiplication is implemented as four 32 bit multiplications, with modulo reduction applied to 128 bit result (represented in an array of two 64 bit values) (Section 4.1.5.2).

uminus.c, uminus.m This function implements simple unary minus, or negation of a real or complex input array. For complex arrays both real and imaginary parts are negated.

sum.c This implements an simple sum function. There is no treatment of arrays, all the elements of whatever dimension argument is provided are summed together producing a single output. This was included for optimisation of the Rader-Brenner algorithm (Section 4.2.4).

r8.c, r8conj.c These files implement multiplication by the primitive eighth root of unity and, respectively, it's conjugate. Since this root is a power of two (Section 4.2.1.2) multiplication can be performed using only shifts and adds, avoiding explicit multiplication. Since these operations cannot be performed within MATLAB, these external functions were necessary.

The files that follow implement core functionality, but without recourse to the MEX interface, since they utilise the functions described above.

abs.m Returns the absolute value of a complex number.

display.m Controls how MATLAB should display a number on the screen. Converts to a signed number first.

dot.m Implements the vector dot product. Taken from the MATLAB dot.m.

rdivide.m Pointwise division of elements in one array by corresponding elements in a second array of equal dimensions. Originally only real divisors were supported, since this function is only needed for calculating the inverse of the transform length when implementing the inverse transform. However for the convolution testing, further division was required for calculating inverse filters, and so full complex division was implemented.

mexall.m This was a basic build script to compile all of the component mex files with one command.

The final part of the implementation is the files that should be placed in an `@double` directory.

i64.h Header file required for compilation.

ui64.c This function converts the standard MATLAB data type (double) into the finite field, taking account of sign (Section 4.1.1.3).

getq.c This function simply returns the value $M_{61} = 2^{61} - 1$. This is required in this directory since the getq function requires no arguments. (Section 4.1.4).

4.1.5.2 Discussion

Obviously there were some technical obstructions to be overcome in order to implement the framework as described above. The first was how to ensure the functions were general enough to handle inputs of arbitrary size and dimension. As a first attempt, the C code was written only for scalars. This functionality was then extended with m-files to handle the arrays. The assumption was made that only arrays of dimension two would be used, and after finding the size of the inputs, hardcoded loops iterated through the indexes for each dimension. After some further investigation however, it was found that it is in fact easier to implement this array handling within the external C function. This is because of the way the mxArray data structure works. After obtaining the pointer to the start of the array data set, it is possible to simply increment this pointer (provided it is of the correct data type) and all the array elements will be reached. Since the imaginary part of complex data is stored in a different memory area, with another pointer, the same process can be applied there. The only information required is the total number of elements in the array, which is returned by the mxGetNumberOfElements() function. Of course this only works for pointwise operations, any operation which depends on an elements position in the array (such as matrix multiplication or even the dot product) must be implemented differently.

The getq function was added after it was realised that it was difficult to input large numbers, close to the modulo used, as was necessary for testing many of the functions. This is of course because any number entered as digits into MATLAB is automatically a double, and so has too few significant figures to represent the integer in question. getq is therefore a simple function that simply returns a `uint64` element containing the modulus.

It was initially desired to keep the framework flexible enough that a simple change in the header file followed by a recompile, would allow the framework to implement a different modulus. However it was found that this would have added a significant amount to the development time, and was somewhat beyond the scope of this project. There are some places in the code where the choice of modulus affects the actual code, for example the bit shifting in the modulo reduction (see below). It might be possible to use precompiler directives to make these easier to change, provided the modulo was still a Mersenne prime, but as it stands some changes to the code are required should a different modulus be required.

Other points to note are that mtimes (MATLAB $*$) was not implemented, since matrix-matrix multiplication was not required for the algorithms considered. It would be a simple matter to code it as an m-file however, or even add the addition MEX function. There is also some redundancy in the code, for example the rmultmod() function which implements real scalar modulo multiplication internally is repeated in both the times.c and power.c files. This again could be overcome by placing it in a separate file, but due to the small number of such functions and the size of the project it was not felt this was necessary.

As the most complicated, and also the most important function in the implementation, times.c deserves some additional discussion, which will also better illustrate the framework as a whole. The times.c file contains four functions, addmod, which performs modulo addition, submod which performs modulo subtraction, rmultmod which performs real scalar modular multiplication and mexFunction which is the MEX entry function and contains the runtime logic of the function. Addition is performed using the normal 64 bit integer addition in C, the result is the reduced modulo M_{61} by the return command `(temp & M61) + (temp >> 61)`. This first term in this sum is the temp variable with a bitmask (equal to M_{61}) applied, which keeps only the lower 61 bits of the word. The second term performs the bitshifting, or 'carry', which completes the modulo operation, as described previously (Section 4.1.1.1). Addition followed by this reduction is precisely ones' complement addition, or addition with end-around carry, as noted earlier. Subtraction is exactly the same, except this time the second term is negated before addition. This is achieved by the `((~y) & M61)` operation, which takes the bitwise NOT of the number (ones' complement), and again masks it with the modulo so that only the digits of interest are preserved. In these operations overflow is not a concern, since the result of adding two 61 bit words is at most a 62 bit word, which still fits comfortably within the 64 bit variables used. This masking of the complement ensures that the three 'spare' high order bits are set to 0, leaving space for this overflow. Similarly, only one shift and add modulo reduction operation is required, since the overflow after the addition can be at most one bit, which is not enough to cause a second overflow. The scalar multiplication presented a problem since there is no 64 bit multiplication, with a 128 bit result implemented in the any of the C compilers tested. It was therefore necessary to code it as four 32 bit multiplications, using the technique of 'schoolboy multiplication' [30]. Unfortunately there is a lot of overhead at this stage, both in terms of the increased number of operations required to perform the technique and the number of temporary variables required. The first step of the modulo reduction works on the 128 bit result, that is split into two 64 bit variables. This result can be held in the 64 bit variable since, the addition of two 61 bit and two 3 bit words must fit into 64 bits. Two further shift and add operations are required, to ensure that all overflow is accounted for and the final reduction is obtained.

The main runtime logic takes place inside the mexFunction gateway function. Initially the arguments are checked to ensure there are the right number. There must be two input arguments and not more than one output. There does not have to be exactly one output, the function can be called from the MATLAB prompt outside of a variable assignment (so no outputs) and the result would be displayed. An if statement checks if either of the two inputs is a scalar (in MATLAB a scalar is a 1×1 array) and if so, the output arguments are created accordingly (to the size of the other element, that may or may not be a scalar). An if...else structure then takes care of the four possible cases, combinations of either argument being real or complex. If neither is scalar, the dimensions are checked to ensure they are the same and then a similar if...else structure deals with the permutations of the arguments being complex or real, creating the output array accordingly.

4.1.5.3 Testing

The framework was tested as it was developed, with each function being tested first with small positive integers, to ensure the behaviour matches that which was expected, and then with small negative integers, which of course are mapped in the finite field to very large unsigned integers. Some arbitrary larger integers were also tested, with an arbitrary precision computer algebra package on hand for confirmation (MapleSoft's Maple v. 9 was used). Different combinations of real and complex inputs were tried, to ensure the code for handling different cases was all tested. In this way any problems were quickly uncovered and the relevant function updated.

4.2 MATLAB Code

The code described above provides a basic implementation of a finite field number system, with a broad set of operations conveniently defined for arithmetic modulo M_{61} using normal MATLAB notation. This basic framework was then extended with more conventional MATLAB code to implement the actual transform and convolution algorithms. This section describes this code, and supplements the previous one as documentation by example for the finite field framework. The results of the various algorithms discussed here are presented in the next chapter.

4.2.1 Radix-2 CMNT

4.2.1.1 cmnt.m

A basic CMNT was written in MATLAB in a function `cmnt.m`. The function takes as its argument a vector with length a power of two, and implements a simple recursive radix-2 Cooley-Tukey algorithm. First the primitive root of order 2^{p+1} is set (Section 3.3.1) and from that, the primitive root of the correct order for the length of the transform is calculated. The table of twiddle factors is generated, and then the transform proceeds by way of a recursive function call.

By using the MATLAB profiler, it was shown that most of the computation time was spent on the initial calculation of the table of twiddle factors, for later use in the recursive algorithm. It was found that the speed of this stage could be improved simply by ensuring the vector of twiddle factors was defined in advanced. Initially the MATLAB soft typing had been exploited, as below

```
for i=1:d
 twiddle(i) = r.^uint64(i-1);
end
```

Simply by adding `twiddle=uint64(zeros(1,d));` before this block, decreased the execution time by a factor of 10. This must be due to the overhead caused by MATLAB having to dynamically extend the memory allocated to the `twiddle` array. A further performance improvement was obtained by reducing the computation to one multiplication per loop operation, instead of one exponentiation, as below

```
twiddle(1) = 1;
for i=2:d
 twiddle(i) = r .* twiddle(i-1);
end
```

The results of the MATLAB Profiler, illustrating these improvements, can be found in Section C.1.1.

Another point to note is that the output of this function is always a $1 \times N$ array, regardless of whether the input is $1 \times N$ or $N \times 1$. This is a result of the reconstruction of the result in the recursive algorithm, but of course could be easily rectified. For the purposes of this demonstration though it wasn't felt that it was necessary. Once the `cmnt` function described above was shown to be working (Section 5.2), further optimisations were applied to improve the performance of the transform.

To provide an indication of the overhead involved with the finite field operations, an additional MATLAB function, `fft1.m` was implemented. The code for this was exactly the same as for the `cmnt.m` function described above, apart from the fact that the primitive root is changed to $e^{-\frac{2\pi i}{d}}$ and the standard MATLAB operations are used, instead of the `uint64` overloaded ones.

4.2.1.2 cmnt2.m

In a preliminary investigation into possible optimum transform lengths, the primitive root r for a range of lengths was calculated in MATLAB.

```
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),
TABui64(-3).^(uint64(2).^uint64(59)) );
for i=1:62
TABd = 2^i;
TABr(i) = alpha .^((uint64(2).^uint64(62))./uint64(d));
end
```

From manual observation of these results it was noted that `r(2)`, that is the primitive root of order $2^2 = 4$, was of a particularly simple form, namely $r = -i$. By manual calculating the result of this transform in the general case, it was possible to obtain an optimised algorithm for transforms of length $n = 4$, requiring only 8 complex additions. This derivation is given below. Recall the definition of the transform is $A_k = \sum_{j=0}^{d-1} a_j r^{jk}$. Expanding this and substituting $r = -i$ gives

$$\begin{aligned}
A_0 &= a_0 + a_1 + a_2 + a_3 \\
A_1 &= a_0 + a_1(-i) + a_2(-i)^2 + a_3(-i)^3 = a_0 - a_1 i - a_2 + a_3 i \\
A_2 &= a_0 + a_1(-i)^2 + a_2\left(-i\right)^4 + a_3\left(-i\right)^6 = a_0 - a_1 + a_2 - a_3 \\
A_3 &= a_0 + a_1\left(-i\right)^3 + a_2\left(-i\right)^6 + a_3\left(-i\right)^9 = a_0 + a_1 i - a_2 - a_3 i
\end{aligned}$$

so defining $A = a_0 + a_2$, $B = a_1 + a_3$, $C = a_0 - a_2$, $D = a_3 - a_1$, and $E = Di = (a_3 - a_1)i$ the transform can be computed as below, with eight additions and a negation. Note that the indexes are different to the above, since MATLAB array indices start at 1 instead of 0.

```
A = x(1) + x(3);
B = x(2) + x(4);
C = x(1) - x(3);
D = x(4) - x(2);
E = complex(-imag(D),real(D));
y = [ (A+B) (C+E) (A-B) (C-E) ];
```

This is a significant improvement over the 6 multiplications and 12 additions required for the standard radix-2 algorithm to compute a transform of length $n = 4$.

Another 'special case' primitive root is that of order 8. It is shown in [9] that primitive eighth roots of unity in the field $GF(q^2)$ are powers of 2 when q is a Mersenne prime. In this case, some simple manual testing in MATLAB showed that the 8^{th} primitive root of unity when $q = M_{61}$ is $r = 2^{30} - 2^{30}i$. It is therefore possible to perform multiplications by this number as simple bitshifts, avoiding the need for explicit multiplications. The powers of the primitive eighth root of unity are presented below

power of r	0	1	2	3	4	5	6	7
value	1	r	$-i$	$-\bar{r}$	-1	$-r$	i	$\bar{r}$

Using this information an algorithm was derived 'long hand' in a similar manner to that above, yielding the following result for a length $n = 8$ transform.

```
C = x(1) - x(5);
```

```
D = x(2) - x(6);
Er = x(3) - x(7);
E = complex(-imag(Er),real(Er));
F = x(8) - x(4);
G = x(1) + x(5);
H = x(3) + x(7);
I = x(4) + x(8);
J = x(2) + x(6);
A = G + H;
B = I + J;
K = G - H;
Lr = I - J;
L = complex(-imag(Lr),real(Lr));

y = [ (A+B) (C + r8(D) - E + r8conj(F)) (K + L) (C - r8(F) + E - r8conj(D)) ...
(A-B) (C - r8(D) - E - r8conj(F)) (K - L) (C + r8(F) + E + r8conj(D))];
```

This implements the length 8 transform in 28 additions and four shifts, with no actual multiplications. Of course there is no benefit to having both optimisations included, since the $n = 4$ optimisation will never be used, as the length 8 transform is computed directly and not split into two length 4 transforms. It was found (Section 5.2) that the length 8 optimisation provided the greatest performance improvement, so this is the one that was implemented.

This change was implemented in `cmnt2.m` and the resulting performance increases can be seen in Section 5.2.

4.2.1.3 cmnt3.m and cmnt4.m

Since most audio signals are real, the implementation of the transform was modified to compute the transform of a length N real signal as a length $\frac{N}{2}$ complex transform, as described earlier (Section 2.1.1.1). This was implemented in `cmnt3.m`. However in this case in order to reconstruct the length N transform result from the two half length transforms, the same principles as used in the recursive radix-2 Cooley-Tukey algorithm were employed. The real signal was split into a complex signal, with even elements forming the real part, and odd elements the complex part. This complex signal is then transformed, yielding the transforms of the two subsequences, which must then be reconstructed by multiplication with twiddle factors as in the normal recursive algorithm. For this reason, the twiddle factors for the full length N transform must be computed, and since the manual reconstruction is of the same computational complexity as the extra recursive step that has been eliminated, little performance improvement is seen.

However in `cmnt4.m` the same principle is put to use in a slightly more productive way. In this case, the transform of two real signals is performed simultaneously, by making them real and complex parts respectively of a complex signal. The two output transforms are reconstructed as before, and the output of the function is the concatenation of the two resulting transforms.

4.2.2 Overlap Add Convolution

The CMNT function, `cmnt.m`, described above was then used to implement a convolution function, `cmntconv1.m`. This was a simple overlap-add algorithm, with no checking of dynamic range. The impulse response must have length a power of two, due to the requirements of the transform. Since this was intended to be a simple proof of concept, there is no checking of input variable dimensions. Inputs must be row vectors. This was used to check the finite field structure was working, and that convolution was possible. It was then used as a starting point for development of a more optimized block convolution function.

Various optimizations were applied to this base function. In `cmntconv2.m` the function is expanded to include the code for the transform. This means that the twiddle factors can be computed once, at the start of the convolution operation, instead of every time a block is transformed as in the previous version. In `cmntconv3.m` a the special case optimisation developed in `cmnt2.m` for transforms of length $n = 8$ (Section 4.2.1.2) is included. Finally in `cmntconv4.m`, which is valid only for real input, blocks are transformed in pairs as a single complex transform. This uses the same technique as `cmnt3.m` and `cmnt4.m` above, but this time it is more beneficial since no explicit reconstruction is required. This is due to the linearity of the transform. The 'complexified' data can be multiplied directly by the transfer function (transform of impulse response) and then inverse transformed as a whole. Linearity then ensures that the real part of the result is the convolution of the real part of the input and likewise for the imaginary part. The individual transforms of the two input data sets are never explicitly calculated.

4.2.3 CMNTFilt

Another function, `cmntfilt.m`, was written to provide equivalent functionality to the MATLAB Signal Processing Toolbox function `fftfilt`. The main difference between this and the `conv` functions described above, is that the `conv` functions give the full output of the convolution including the tail. The output from the function is therefore $L_x + L_h - 1$ where L_x, L_h are the lengths of the signal and the impulse response respectively. The MATLAB `fftfilt` function returns a vector of size L_x, truncating the tail of the result. In addition the MATLAB `fftfilt` function calculates the optimum block length, based on hard coded data regarding the *FLOPS* (Floating Point Operations Per Second) needed to compute FFT's of various lengths. The fftfilt function is written in standard (albeit slightly obfuscated) MATLAB code, and can be found in the `toolbox/signal/signal` directory under the main MATLAB installation. This code was then modified to work with the CMNT, with the twiddle factors precomputed as above, and the recursive function including the optimisation for the case $n = 8$.

Unfortunately, since the MATLAB adoption of the LAPACK floating point library[11], there is no way to count FLOPS for user written code, and so it was not possible to replace these values with the corresponding data for the CMNT. Instead, the FFT data was left in place. It was assumed that the relative FLOPS values for different transform lengths should be similar for both transforms, since it is likely that the same basic algorithm is being used. Also, this way the `cmntfilt` function would always be using the same block lengths as the `fftfilt` function, which provides for fairer comparisons of their performance.

While the original `fftfilt` function can work on arrays, transforming the columns independently, for simplicity the `cmntfilt` function is restricted to vectors. Both row and column vectors are acceptable for the input, and the output will be the same type of vector.

4.2.4 Rader-Brenner Algorithm

The Rader-Brenner algorithm [34] is a Cooley-Tukey-like split radix algorithm. However it uses a different construction of subsequences to split the input data, which results in the twiddle factors having purely imaginary parts. This reduces the total number of multiplications required, since a complex multiplication is usually three (or four) real multiplications, whereas multiplication by a purely imaginary constant requires only two real multiplications. It is not often used for the FFT, since it has poor numerical properties, as in the reconstruction of the result, division by small numbers is required.

[11] In MATLAB Version 6

This amplifies any error and quantisation effects in a normal number system. In [35] the algorithm is presented for the CMNT. In this case, the finite field arithmetic means the result is exactly equal to the direct convolution, so disadvantage of poor noise characteristics does not appear. It was considered especially promising in this implementation since it was felt that the C code for the 64 bit multiplications on a 32 bit processor(Section 4.1.5.1) was far from optimal and so reducing the multiplication count would yield performance improvements.

4.2.4.1 The Algorithm

The derivation of the algorithm in a finite field proceeds exactly as that in the complex number field [34] resulting in the following algorithm. Given an input sequence $\{a_n\}$ the following subsequences are defined

$$\begin{aligned} b_n &= a_{2n} \\ c_n &= a_{2n+1} - a_{2n-1} \end{aligned}$$

where all operations are modulo q, and the indices are calculated modulo N where N is the length of the transform. The transform of a_n, A_k is then given by

$$\begin{aligned} A_k &= B_k - C_k \left(r^k - r^{-k}\right)^{-1} \text{ for } k \neq 0, \frac{N}{2} \\ A_0 &= B_0 + Q \\ A_{\frac{N}{2}} &= B_0 - Q \end{aligned}$$

where B_k, C_k are the transforms of b_n, c_n respectively and $Q = \sum_{i=0}^{N-1} a_{2n+1}$ (sum of odd terms). The algorithm can then be applied recursively, as in the case of the Cooley-Tukey algorithm. The algorithm can be derived form consideration of the Cooley-Tukey radix-2 case. Let $d_n = a_{2n+1}$ with D_k it's transform. Then the Cooley-Tukey algorithm (Section 2.1.1.2) gives $A_k = B_k + r^k D_k$. However from the circular shifting theorem for the FFT, which also holds for a NTT, we see $C_k = D_k \left(1 - r^{2k}\right)$ and so $r^k D_k = -C_k \left(r^k - r^{-k}\right)^{-1}$, yielding the algorithm above. The multiplication savings come from the fact [35, Theorem 1], that for $N = 2^p$, $p < 2^{q+1}$, $\left(r^k - r^{-k}\right)^{-1}$ is purely imaginary, due to r^k and r^{-k} being complex conjugates of each other.

4.2.4.2 Implementation

The Rader-Brenner algorithm was implemented in a function `cmntrb.m`. The structure of the function is similar to that used previously. The twiddle factors are calculated and the transform proceeds through a recursive function call. The earlier optimization for length $n = 8$ is also included (Section 4.2.1.2).

When profiling the initial implementation with the MATLAB profiler, it was found that the most expensive section of the code was the calculation of the sum of odd elements Q, which has to occur during every recursive function call. It was felt this was in a large part due to the fact that the `sum` function was originally implemented in MATLAB code. A simple `sum` function was therefore coded in C to speed up this section.

Unfortunately despite this effort, the expected performance improvements were not seen. It seems likely that the extra complexity required in the less efficient MATLAB code is not offset by the multiplication savings in the faster C code, but it is felt that the algorithm may have promise in a hardware implementation (Sections 5.2, 6.1.1).

Chapter 5
Results

This chapter describes some of the results of the MATLAB code described in the previous chapter are presented.

5.1 Standard Procedures

The procedure for importing audio signals into the MATLAB environment was as follows. First, the command `[data, Fs, bits] = wavread('filename');` is used to obtain the data. This returns numbers in the range $[-1, 1]$. Obviously it is impossible to convert these non-integer values into the finite field structure in a meaningful way, so it is necessary to scale the data to it's actual integer values by multiplying by 2^{bits-1} where $bits$ is the word length of the data (usually 24). The one is subtracted in the exponent, since a b bit signed word can represent numbers in the range $[-2^{b-1}, 2^{b-1}-1]$ in standard two's complement arithmetic. The data is now in the form of signed integers of an appropriate magnitude, and so can be converted into the finite field by using the function `ui64()`. Note that `ui64` must be used where values could have negative sign; in cases where the values are guaranteed to be positive the built in `uint64` could also be used. To obtain the results below 24 bit, 96 kHz WAV files were used, as this is the standard for high quality digital audio.

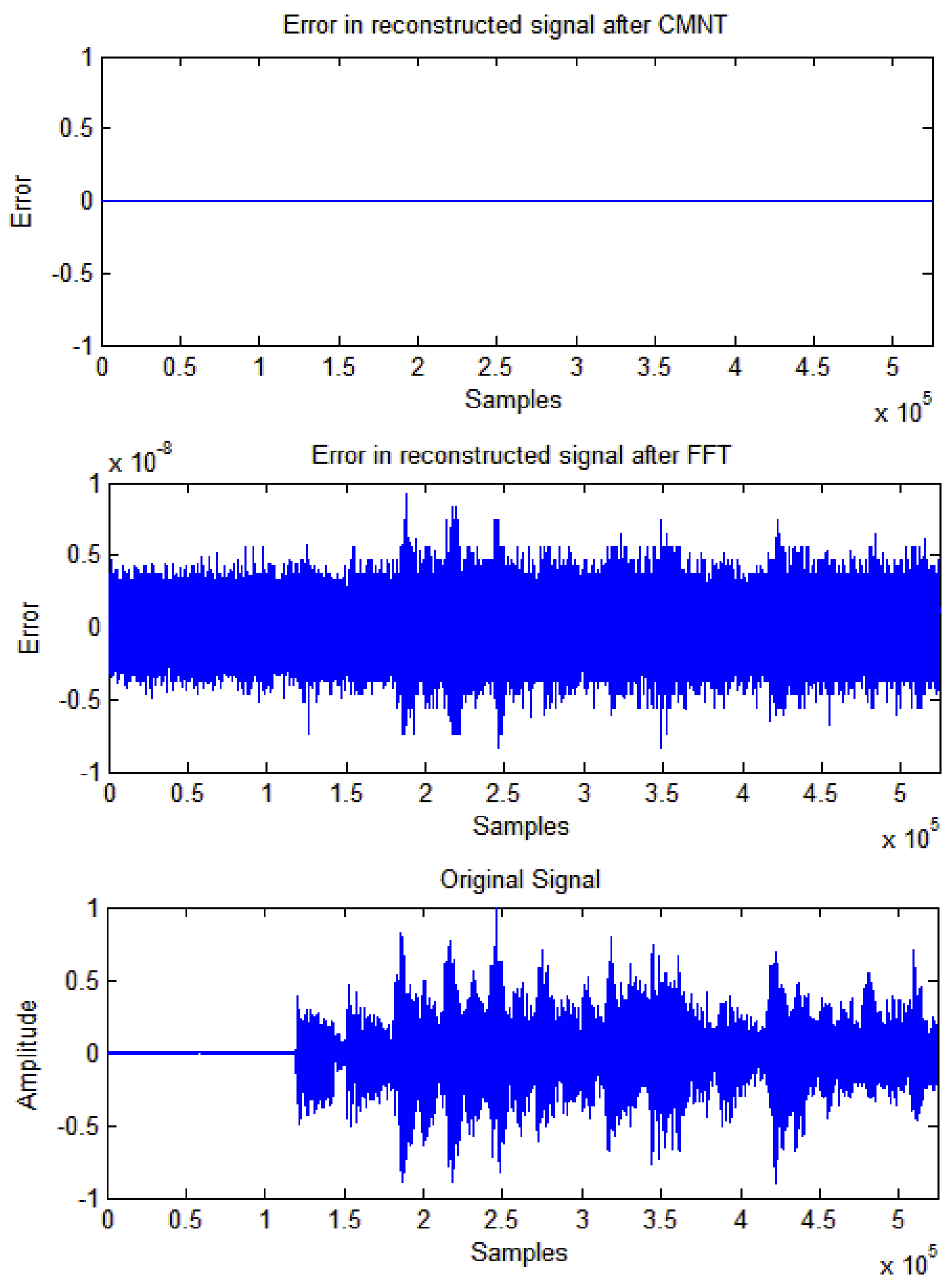
Where times for operations are presented, these have been obtained either through the MATLAB profiler, which gives a break down of computation time for each command in an m-file, or more commonly by using the **`cputime`** function. This returns the total CPU time (in seconds) used by MATLAB from the time it was started. Therefore a sequence of commands such as `t=cputime; command; t=cputime-t` will return the CPU time used to execute `command` in the variable `t`. All the example timings are taken from a Toshiba Tecra M2 laptop, with a $1.8Ghz$ Pentium-M CPU and $512MB$ of RAM.

5.2 Transform

Figure **??** illustrates the lossless nature of the CMNT. $2^{19} = 524288$ samples of high quality audio, corresponding to just under $5.5s$ at $96kHz$ was transformed using both the FFT and the various implementations of the CMNT (Section 4.2.1.1). The results were then inverse transformed, and the results of that compared with the original signal. Note that although the magnitude is very low, there is a clear amount of error in the FFT case, while in the case of the CMNT, the error is precisely zero. It can be seen that the error in the FFT is not a constant noise floor, but is in some way dependant on the signal; peaks in the FFT error can clearly be seen to coincide with peaks in the original signal. This is very much undesired behaviour in an audio system as it is more likely to result in perceptible audio artifacts. The time taken for the relevant transforms was also recorded. Note that the results from the different versions of the CMNT function were all exactly equal.

Transform	Time for Transform (s)	Time for Inverse (s)
`cmnt.m`	42.5	43.5
`fft1.m`	33.6	35.4
`cmnt2.m`	21.9	22.9
`cmnt3.m`	18.3	
`cmnt4.m`	32.1 (16.05)	
`cmntrb.m`	35.0	36.6
FFT	0.2	0.27

The MATLAB code used to generate this plot can be found in Section C.1.2. Note that no inverse time is given for `cmnt3.m` and `cmnt4.m` since these functions require real input and so can not be used for simple inversion (where the input is the complex result of an earlier transform). Also in the case of `cmnt4.m` two sequences were transformed, so the time for one can be considered half the total, which is presented in brackets. When the special optimisation for the case $n = 4$ was tested it was found the operation above took $25.5s$. Therefore the length 8 optimisation was more effective, and this is the one that was used.



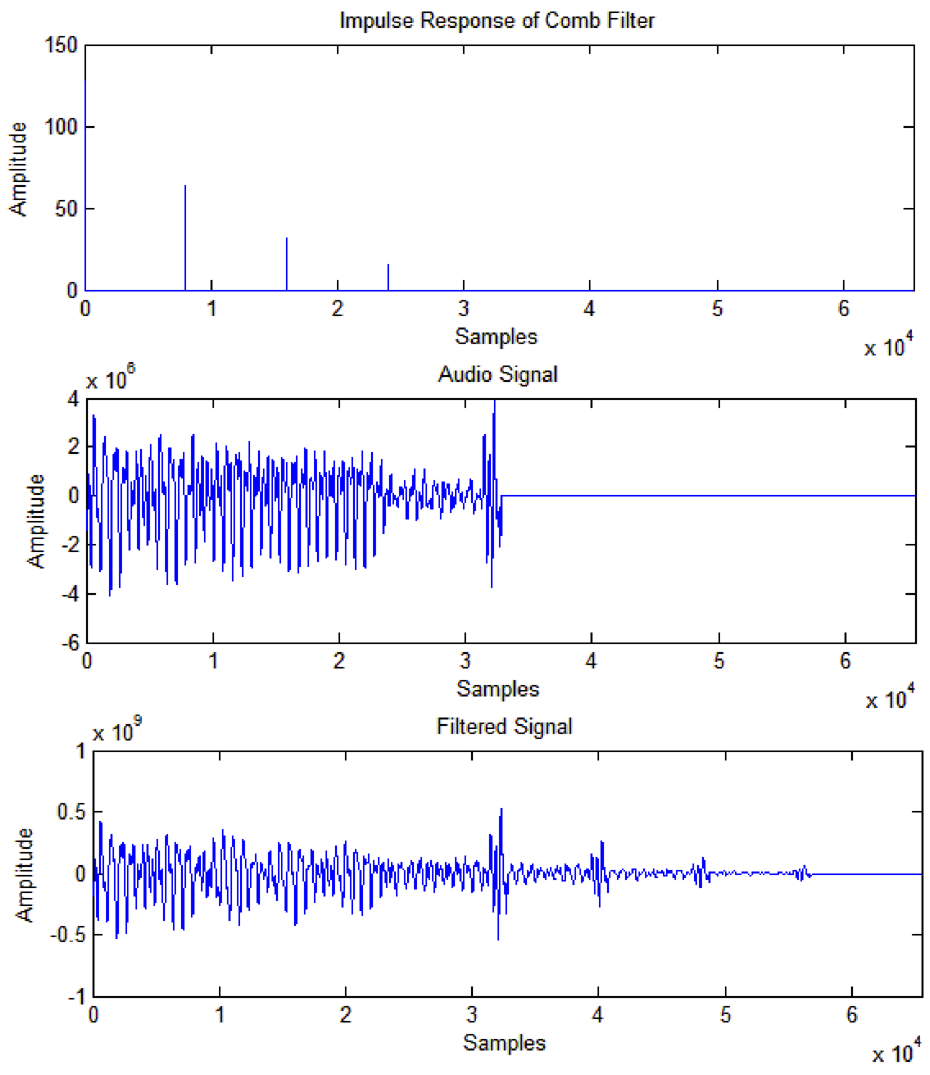
CMNT and FFT Error

optimisations made: cmnt2 3 4

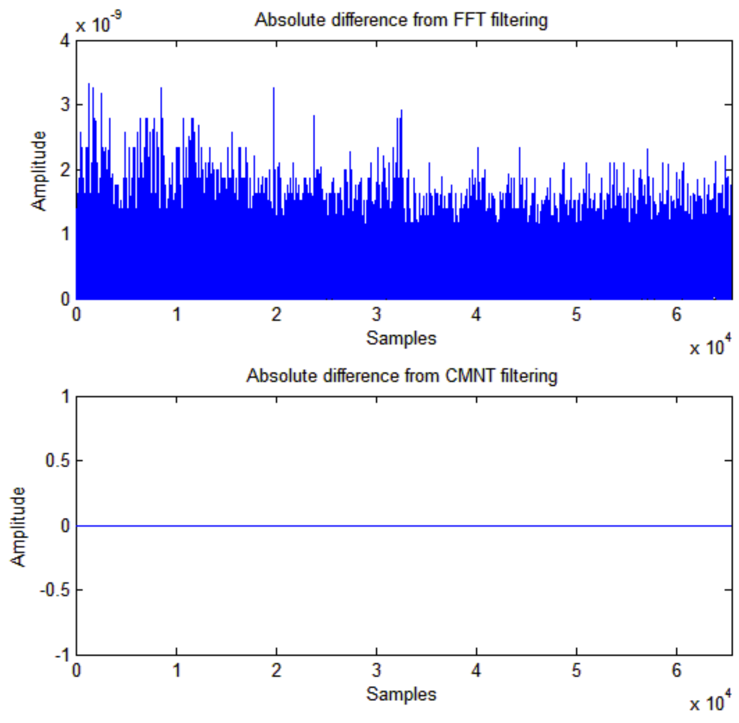
5.3 Convolution

Here a simple comb filter is demonstrated. This is similar to the demonstration in [15, Section 8], but using larger wordlengths. A 24 bit audio signal was filtered with the filter having a maximum value of

128. This was to ensure the result would fit comfortably within the finite field structure. The code used to generate these plots can be found in Section C.2.1.



CMNT Based Convolution



CMNT and FFT Inverse Filtering Error

It can be clearly seen that in the case of the CMNT the result of the inverse filtering operation is a bit for bit copy of the original signal, whereas in the case of the FFT, there is error introduced.

The times taken for different implementations of the convolution were then recorded.

Convolution Function	Time (signal 2^{15}, IR 2^{12}) (s)	Time (signal 2^{19}, IR 2^{13}) (s)
`conv` (MATLAB builtin)	0.86	25.7
`cmntconv1.m`	11.10	169.1
`cmntconv2.m`	9.97	152.2
`cmntconv3.m`	4.47	68.6
`cmntconv4.m`	2.45	35.5
`fftfilt` (MATLAB builtin)	0.04	0.64
`cmntfilt.m`	3.80	46.5

Chapter 6
Discussions

In this chapter, the results presented previously are discussed. Applications, optimisations and possible avenues for future work are also considered.

6.1 Interpretation of Results

6.1.1 Transform

As expected, the diagrams show that the transform is perfectly invertible. That is the result of applying the transform, and then the inverse transform, is a bit for bit copy of the original signal. This is in stark contrast to the standard FFT, which, as discussed above, results in a noise signal that is clearly dependant on the input signal.

Clearly the CMNT takes significantly longer than the FFT. The fairest comparison is between `cmnt.m` and `fft1.m`, since those functions employ the same algorithm, the only difference being the use of the new finite field arithmetic rather than the standard MATLAB arithmetic. This shows directly the overhead due to the use of the finite field framework. It can be seen that the use of the finite field operations results in a performance degradation of about 30%, on a data set of this size. However this does not seem that bad, considering the implementation of 64 bit arithmetic on a 32 bit processor. While the MATLAB double also uses a 64 bit floating point data set, this uses a highly optimised machine coded floating point library, whereas the finite field operation implements a crude 64 bit multiplication in C.

The optimisations developed can also be seen to have been quite effective. The most pronounced improvement is between `cmnt.m` and `cmnt2.m` where the specific optimisations for direct calculation in the case $n = 8$ was included, causing an improvement of almost 50%. The other optimisations which applied on to real signals caused further, albeit less significant, increases. Disappointingly the Rader-Brenner algorithm did not perform so well. Bearing in mind the `cmntrb.m` function includes the length 8 optimisation, it must be compared to `cmnt2.m` which is the equivalent standard Cooley-Tukey implementation. Clearly the saving from the reduced complexity of the multiplication is offset by the cost of the extra manipulations required in the MATLAB implementation of the algorithm. Even with the mex-file `sum.c` function, the sum of odd terms still provides a significant overhead. It is possible in a native or hardware implementation, where there is less overhead for the subsequence creation and the computation of the twiddle factors it might be more competitive. However it seems likely that a native optimised standard Cooley-Tukey algorithm is likely to be both simpler and more efficient.

It is clear that none of the CMNT functions tested are very competitive compared with the MATLAB builtin FFT function. However, given that this is very much a rough and ready test implementation of the CMNT, and that the MATLAB FFT uses the highly optimised and widely renowned *fftw* library[12], it is perhaps not a fair comparison. The inclusion of the `fft1.m` function, provides a fairer comparison as noted above. It is clear from this, that the majority of the extra computation time for the CMNT functions, versus the MATLAB FFT, is due to the overhead of using MATLAB for implementing the algorithm. There is no reason that the same performance increase seen between the basic `fft1.m` function and the MATLAB FFT couldn't be seen with the CMNT, if it was implemented more

[12] The "Fastest Fourier Transform in the West", *http://www.fftw.org/*, winner of the 1999 J. H. Wilkinson Prize for Numerical Software.

efficiently, and natively instead of in MATLAB code. For example most optimised FFT routines do not use a recursive function call and many compute the transform 'in place' which reduces the overhead associated with moving data between different areas in memory. The only performance hit would be the extra modulo reductions required, which have been shown to add a penalty of approximately 30%, although it it felt that this could also be improved (Section 6.3). This seems a small price to pay for completely error free signal processing.

6.1.2 Convolution

Again the results show that convolution in a finite field is indeed lossless, and that in the case of a simple comb filter, convolved in one block, the operation can be perfectly inverted. The CMNT performs worse than the MATLAB builtin functions, likely for similar reasons as those discussed above. The MATLAB `conv` function implements convolution as multiplication in the frequency domain by means of the FFT, but only one block is used and the inputs are zero padded to the correct size. The MATLAB `fftfilt.m` function implements overlap-add block convolution, as discussed above (Section 4.2.3). Again the large difference seen between the finite field functions and the MATLAB built-in's is due to the highly optimised, native FFT implementation.

The optimisations developed provide significant performance improvements. As before the most significant comes from the inclusion of the $n = 8$ optimisation (`cmntconv3.m`) which nearly halves the computation time. It is also noteworthy that the optimisation for real signals (computing two real blocks as one complex block) performs much better, achieving close to the expected 50% reduction in computation time. The reason this improvement is seen here, but was not seen when dealing with the individual transform, is that in this case there is no additional manipulation of the data necessary to recover the output. Due to the linearity of the transform the convolution can be applied to both blocks simultaneously (Section 4.2.2).

Unfortunately none of the CMNT functions get close to the performance that would be required for real-time processing (recall that a signal of 2^{19} samples corresponds to about $5.5s$ of audio at $96kHz$). However there is no reason this could not be improved significantly given a more efficient native transform implementation, especially on 64 bit hardware (Section 6.3).

6.2 Filter Design

Given the transform and convolution algorithms described above, the question remains as to how to perform useful signal processing in the unfamiliar finite field. Fortunately this is an issue that has been addressed in the literature, most notably in the work of Murakami and Reed ([11],[9]). While they deal primarily with a recursive implementation of FIR filters, in [11] they also present a method for designing filters with a non-recursive implementation as considered in this project.

In [11, Eqn 11a] a complex number theoretic (CNT) z-transform is introduced

$$H(z) = \sum_{n=0}^{d-1} h_n z^n$$

It is shown to share many many of the properties of the usual z-transform, most notably the linearity, shifting, convolution and differentiation properties. This transform can be used to aid in filter design, as in conventional techniques.

If the impulse response of the required filter is specified, it is easy to scale this appropriately (usually by multiplication by a large integer, possibly with quantization to ensure all integer values in the impulse response) and implement it through the transform methods discussed above. However if the frequency response of the filter is specified, a little more manipulation is required to obtain a form suitable for the finite field. An expression for the number theoretic transform of a filter with a given frequency response is [11, Eqn 26]

$$\hat{H}(k) = \sum_{n=0}^{d-1} \left(\sum_{l=0}^{d-1} G_l \left[\rho \omega^{nl} \right] \right) r^{kn}$$

where ρ is a scaling constant, [] denotes quantization (integer part) and G_l is the frequency sample at frequency ω^{-l}, $G_l = G(z)|_{z=\omega^{-l}}$. A bound for the error caused by the quantization in this step is obtained, and is found to be $\epsilon \leq \frac{2}{\rho} \left(\frac{1}{d} \sum_{l=}^{d-1} |G_l| \right)$.

Using these expressions, existing filter design techniques can be employed in the finite field structure. The main requirement is that any resulting impulse response's be scaled and quantized appropriately.

6.3 Optimisations

As already mentioned, the most significant performance improvement would come from implementing the transform and associated algorithms in a lower-level language such as C, rather than as MATLAB code. It is also felt that using a 64 bit processor, with a compiler that natively supports 64 bit multiplication (with a 128 bit result) would also make a big difference, since it would eliminate the messy 'schoolboy multiplication' implementation. However even with the current system there are still some improvements that could be made.

6.3.1 Current Implementation

In some areas it might be possible to reduce the number of modulo reductions. At the moment the reduction takes place in the C code after every operation. However it might be possible to implement functions where operations are applied repeatedly, such as multiplication in `power.c`, more efficiently. The 'spare' three bits in the 64 bit word provide a bit of leeway. For example the reduction procedure only needs to occur every three additions. If a tally of the additions can be kept as in `sum.c`, then the total number of reductions can be reduced. In `power.c`, the second reduction in the `rmultmod()` function might not need to be applied every time.

Another optimisation that could be applied to the system as it stands would be a trick for performing complex multiplication as three real multiplications [30]. In this if $a+ib$ and $c+id$ are the two numbers to be multiplied, three multiplications are performed, $A = ac$, $B = bd$, and $C = (a+c)(b+d)$. Then the result is given by:

$$(ac - bd) + (ad + bc)\, i = (A - B) + (C - A - B)\, i$$

A similar technique of reducing number of multiplications can be used in the schoolboy multiplication algorithm [30]. Unfortunately there were some problems implementing this in the C code, which led to erroneous results, and so the standard schoolboy method was used, although as noted above, the use of a 64 bit platform would eliminate the need for this section of the code.

It would also be possible to significantly reduce the time taken to calculate the twiddle factors, using the techniques for fast computation of powers of a primitive root described in Section 3.3.2. This would have a greater impact on the standalone transform functions than on the convolution functions implemented, since in those the twiddle factors are only computed once. With a transform length of 2^{19} (as in Section 5.2), calculation of the twiddle factors takes approximately 6 seconds, so this could provide a significant saving.

6.3.2 Other Implementations

6.3.2.1 Software

As discussed above implementing the transform algorithm in a lower level language would yield immediate improvements. It is also likely that the algorithm could be optimised. In practise, recursive algorithms are seldom used for implementing the FFT [36]. Instead bit reversal (simply reversing the order of the binary representation) is used on the indices to re-order the data. The result of this rearrangement is that the transform can be computed in a much simpler fashion. Pairs of points are combined to form the length 2 transforms, then adjacent length 2 transforms are combined to form length four transform, and so on until the whole vector is transformed. An algorithm of this type consists of two stages, the data re-ordering stage, followed by a loop that calculates, in turn, the transforms of length $2, 4, 6$ and so on. Within this loop there is a pair of nested loops that range over the subtransforms already computed, and the elements of each subtransform. Within these loops the twiddle factors are applied to create the larger transform. This is much more efficient than the recursive structure implemented here, since the overhead of the recursive function call, and moving the data around in memory is removed.

A large amount of research has gone into optimising FFT algorithms, and fortunately there is no reason why any algorithm designed for the FFT can not be applied to the CMNT, simply with different twiddle factors (which are now powers of a primitive root) and modulo arithmetic. Indeed, the very algorithm used for power of two length transforms by the MATLAB built in FFT function could be used for the CMNT, with only a small overhead for the modulo arithmetic.

6.3.2.2 Hardware

A competitive implementation could also be developed in hardware. The main requirements are a sufficient word length (probably 64 bit) as well as, ideally, a native ones' complement (addition with end-around carry) operation. If this is present, then even if the word length does not match the prime length required, it is easy to sign extend the word, as discussed earlier (Section 4.1.1.3, [32]). Modulo reduction of the result of a multiplication can be performed by shifting and adding as before.

In a hardware filter it is likely that the block size would be fixed, allowing the twiddle factors to be precomputed and perhaps stored in a look-up table. This would provide further savings.

6.4 Future Directions

The approach presented here is only one of many that can be taken. There is a wide range of literature covering different approaches to computation in the finite field (Section 1). Some possible avenues for future work are now considered.

6.4.1 Improvements to Implementation

As well as the optimisations listed above, there are various other areas in which the implementation developed here could be improved. As mentioned previously any existing FFT algorithm can be applied to the NTT, with very little change except for the inclusion of the modulo reduction in the underlying arithmetic. This opens the possibility of efficient algorithms for calculation of transform lengths not a power of two. Mixed radix algorithms are particularly promising, as are a combination of Winograd and prime factor algorithms (Section 2.1.1.2, [37]). Indeed there seems no reason why a high performance adaptable transform library, similar to the FFTW could not be developed in the finite field case. The FFTW library is adaptable in the sense that it consists of a variety of composable solvers, representing different FFT algorithms and implementation strategies, whose combination into a particular plan for a given size can be determined at runtime according to the characteristics of hardware in use. This peculiar software architecture allows FFTW to adapt itself to almost any machine and there seems no reason why a similar system could not be applied in the case of the number-theoretic transforms considered here. It might also be possible to derive other small transforms, as was done for lengths 4 and 8 (Section 4.2.1.2).

Another area that deserves further consideration is that of dynamic range and amplitude checking to ensure the result of the convolution can be held within the finite field structure (Section 3.4.1). The conditions derived to prevent overflow could be checked in the code of the transform. For example, the impulse response values could be summed and substituted into condition $2c$ from Section 3.4.1, to ensure that the transform length does not exceed a safe level, assuming a saturated input signal. As discussed this is particularly important to ensure validity of results in a finite field computation. As mentioned, the conditions derived earlier are very strict, assuming a 'worst case' in regards to a maximum amplitude input signal with same sign as the impulse response. It is possible that further analysis could yield weaker conditions, given further assumptions about the input and impulse response. For example it is often the case that 'real world' impulse responses exhibit an exponential decay. It might be possible to use such an assumption to obtain a less restrictive condition on transform length. An alternative approach to the problem is presented in [38], where an 'overflow' signal is defined, which provides a runtime indication if any overflow should occur. The author suggests such a signal could be used for automatic scaling of the input sequences. Unfortunately the results presented are for a different implementation than the one developed here, and therefore can not be applied directly. They are developed for a transform where the computation internally is performed module a Mersenne prime, but the final result is evaluate modulo $M = \frac{(2^{q^2}-1)}{2^q-1}$, where $2^q - 1$ is the Mersenne prime used. It is possible that with further investigation a similar scheme could be developed for the transform implemented here.

6.4.2 Alternative Approaches

There are various alternative approaches to dealing with the challenges posed by computation in a finite field, such as the relationships between the order of the field and the transform length and the CMNT proposed here is just one possible solution. There are many others that are worthy of consideration by anyone interested in the field.

6.4.2.1 Direct Sums using the Chinese Remainder Theorem

To avoid the requirement of long word lengths, it is possible to implement the transform over a field formed as the direct sum of several smaller fields. In [5] the authors propose just such a scheme. The field used is a direct sum of fields of the form $GF(q^2)$ where q is a Mersenne prime, for example $F = GF(q_1^2) \oplus GF(q_2^2) \oplus GF(q_3^2)$. Then the input signal is converted into the subfields of F by the relevant modulo reduction. Provided the transform length used is valid in all the subfields, ($d|2^{p_i+1}$ for each i, where $q_i = 2^{p_i} - 1$), the transform can be performed independently in each field, and then the correct result constructed by use of the Chinese Remainder Theorem (Appendix B). This approach provides increased dynamic range, without the requirement of excessively long word lengths. It requires some overhead, due to the addition step of applying the Chinese Remainder Theorem to convert out of the direct sum structure, and of course, a long word length is still required to hold the final result.

6.4.2.2 Multiplication-Free Fast Convolution

Another alternative approach is to trade flexibility in terms of transform lengths for speed of computation. This is the strategy taken during most of previous work in the field. Since a major application will be block convolution, having a relatively inflexible fixed transform length might not be such a serious constraint. If the finite field and transform length can be restricted so that the primitive root is a power of two, then the multiplication by the root (for example, the twiddle factors in a Cooley-Tukey type algorithm) can be performed using only shifts and add's, eliminating the need for explicit multiplication and greatly reducing computational cost. Unfortunately in the implementation developed here, the only roots available as simple powers of two are 2 and -2, with transform lengths of p and $2p$ respectively. Of course p is prime and therefore does not yield a convenient algorithm, although it is possible that use of alternative algorithms, combined with the fact no multiplication is required, could yield an efficient solution. It is possible the other roots exist, as linear combinations of powers of two, which could similarly be implemented without recourse to multiplication by use of shifts and adds. It might also be possible to develop a multiplication free transform using the direct sum techniques outlined above (Section 6.4.2.1), although the fact that the transform length used must be valid in all subfields, combined with the fact that the moduli of the subfields must be relatively prime (so cannot have the same field twice) might prove problematic.

However the idea of lossless signal processing that is computationally cheaper than the existing FFT based techniques is of course very attractive. It is therefore felt that this is a very promising area for further research.

6.4.2.3 Computation on Finite Rings

A further area for investigation is opened by considering that the requirement that the computation be performed over fields is not a strict one. The NTT is well defined over any ring, so by dropping the requirement that it the structure is a field (all inverses exist), and therefore that the modulus q be a prime, many more possibilities are available. The only additional requirement is that for the inverse transform of length d to exist, the multiplicative inverse d^{-1} must exist (this is not guaranteed in a ring). This allows a wide range of composite moduli with interesting properties. It is possible that some of these would prove promising candidates for audio processing, and again, such an approach could be combined with the direct sum or multiplication-free techniques mentioned above.

6.4.2.4 Low Delay Block Convolution

Another direction for future investigations, is that of low-delay block convolution. This is important for real-time signal processing, since block convolutions such as those presented here, while providing reduced computational cost, also cause a degree of latency in the signal path. This is because no output is generated until the calculation of the whole first block is complete. In many applications, this is unacceptable. The most promising solution to this problem is a technique know as *partitioned convolution*. This extends the idea of block convolution, where the signal is split into separate blocks that are processed separately, by in addition splitting the impulse response into blocks. This alone can reduce computation time, since the block size used can be reduced to whatever is required. It can also be extended to a scheme whereby the blocks that the impulse response is partitioned into are of non-uniform size. In such a scheme the first few outputs of the convolution can be computed directly, with very low latency, followed by blocks of increasing size that are more efficient to compute.

Unfortunately such techniques are protected by patents, for example, in the US patent no. 5,502,747, granted March 1996 to Lake DSP, and patent no 6,625,629, granted September 2003 to Microsoft Corporation. The Lake patent covers non-uniform partitioned convolution, while the Microsoft patent appears to cover both uniform partitioned convolution and some special cases of non-uniformed partitioned convolution. However, these patent's are controversial, and many question whether they would stand up in court. It is indeed difficult to see how the second patent, granted to Microsoft, does not infringe on the earlier patent award to Lake DSP. It is felt that this is a promising area for application of the number theoretic transform, since these patents explicitly describe the use of the FFT, and there is therefore the possibility that they would not apply to an NTT implementation. With the Microsoft patent this certainly seems to be the case. Indeed, in the abstract they mention the terms DFT and FFT a total of eight times. The Lake patent is slightly more difficult to untangle, although statements such as “wherein said fast convolution filters are built using the Discrete Fourier Transform or the Fast Fourier Transform” from Claim 1, suggest that it might not apply in this case. This provides another compelling advantage to the finite field representation, since these valuable techniques are not currently available for common use via conventional techniques.

6.4.2.5 Recursive Implementation

As already mentioned (Section 6.2) another method for lossless filtering in a finite field is that of recursive implementation of FIR filters ([11], [9]). To achieve this, the z-transform of the FIR filter is manipulated to reveal an implementation as the cascade of simple FIR system with a recursive IIR system [11, Fig. 1]. This technique can be employed in the usual complex field, although the error’s resulting can affect the stability of the IIR stage, so it is often impractical to implement. Again, the exact nature of computation in the finite field overcomes this problem, and so this method becomes a viable alternative to the non-recursive transform techniques presented here. In [11] the authors note that in some cases, for example a frequency selective filter with only a few passbands, many of the multiplicative constants turn out to be zero, and this method can be more efficient than the non-recursive transform techniques. It therefore warrants further investigation to reduce the computational cost for certain types of filter, while maintaining the errorless output of the finite field computation.

6.5 Conclusion

In conclusion, it has been shown that the number theoretic transform in a finite field allows for signal processing without the round-off and truncation error that results with conventional fixed or floating point arithmetic. This errorless processing can take advantage of much of the work done on developing and optimising algorithms for the fast fourier transform, with the major difference being the use of modulo arithmetic.

Of course, the major factor is the computational cost. It has been shown here that there is no reason a properly designed method cannot be reasonably competitive with the conventional method, although the implementation developed here is certainly more expensive than optimised conventional methods. The question is, of course, is the gain from errorless processing worth the additional cost and the answer depends on the application requirements. It can be argued that the conventional methods, while not errorless, can achieve an arbitrary level of accuracy, given a sufficiently large word length. There will inevitably be a cross-over point, where, for a required level of accuracy, the methods presented here will be cheaper than conventional methods with a long word length. Further work on optimising the implementation of finite field arithmetic will serve to raise this crossover point. The existing high-end audio market, however, is evidence of the fact that, for many, no compromise is acceptable in the reproduction of musical recordings and it is felt that the market could certainly support the extra expense for a system that was completely errorless.

As illustrated in the previous section there is a wide range of promising avenues to explore in this area. It is hoped that in the future, many of these will be developed further, and the original promise of the digital domain, for lossless storage, transmission, processing and reproduction of audio will be realised.

Given the techniques presented here, it is easy to envisage a pure, errorless digital signal path for high quality audio, where the only noise and artifacts introduced into the signal are those that occur at the quantisation and sampling stage. After this, all processing of the audio, for example mastering and other effects, can be done in a lossless manner. The audio can be stored digitally, and presented to the listener through digital loudspeakers with lossless digital crossovers. This would achieve the ultimate goal of digital audio, a truly noiseless signal path from A/D converter to loudspeaker cone.

References

[1] T. C. Bartree and D. I. Schneider, “Computation with finite fields,” *Information and Control*, vol. 6, pp. 79–83, 1963.

[2] J. M. Pollard, “The fast fourier transform in a finite field,” *Mathematics of Computation*, vol. 25, no. 114, pp. 365–374, April 1971.

[3] C. M. Rader, “Discrete convolutions via mersenne transforms,” *Transactions on Computers*, vol. C-21, no. 12, pp. 1269–1273, December 1972.

[4] I. S. Reed and T. K. Truong, “The use of finite fields to compute convolutions,” *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 208–213, March 1975.

[5] ——, “Complex integer convolutions over a direct sum of galois fields,” *IEEE Transactions on Information Theory*, vol. IT-21, no. 6, pp. 657–661, November 1975.

[6] R. C. Agarwal and C. S. Burrus, “Number theoretic transforms to implement fast digital convolution,” *Proceedings of the IEEE*, vol. 63, no. 4, pp. 550–560, April 1975.

[7] H. J. Nussbaumer, “Digital filtering using complex mersenne transforms,” *IBM Journal of Research and Development*, pp. 498–504, 1976.

[8] ——, “Relative evaluation of various number theoretic transforms for digital filtering applications,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-26, no. 1, pp. 88–93, February 1978.

[9] H. Murakami and I. S. Reed, “Recursive realization of finite impulse filters using finite field arithmetic,” *IEEE Transactions on Information Theory*, vol. IT-23, no. 2, pp. 232–242, March 1977.

[10] J.-B. Martens, “Number theoretic transforms for the calculation of convolutions,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-31, no. 4, pp. 969–978, August 1983.

[11] H. Murakami, I. S. Reed, and A. Arcese, “Recursive FIR digital filter design using a z-transform on a finite ring,” *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-31, no. 5, pp. 1155–1164, October 1983.

[12] J. J. Thomas, J. M. Keller, and G. N. Larsen, “The calculation of multiplicative inverses over GF(p) efficiently where p is a mersenne prime,” *IEEE Transactions on Computers*, vol. C-35, no. 5, pp. 478–482, May 1986.

[13] S. Gudvangen, “Number theoretic transforms in audio processing,” *DAFx99 Workshop Submission*, 1999.

[14] ——, “Number theoretic transforms: A tutorial.”

[15] J. A. S. Angus and T. Jackson, “Lossless signal processing with complex mersenne transforms,” *AES 115th Convention*, October 2003.

[16] A. V. Oppenheim and R. W. Shafer, *Digital Signal Processing*. Prentice-Hall, 1975.

[17] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.

[18] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, 1982.

[19] S. W. Smith, *The Scientist and Engineer’s Guide to Digital Signal Processing*. http://www.dspguide.com/, 1999.

[20] J. W. Cooley and J. W. Tukey, “An algorithm for machine calculation of complex fourier series,” *Mathematical Computation*, vol. 19, pp. 297–301, April 1965.

[21] J. Goodman, “The principles of scientific computing: Sources of error,” *http://www.math.nyu.edu/faculty/goodman/teaching/Scientific_Computing/scientific_computing.html*, 2003.

[22] P. M. Cohn, *Classic Algebra*. John Wiley and Sons Ltd, 2000.

[23] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. Oxford University Press, 1980.

[24] R. E. Blahut, *Algebraic Methods for Signal Processing and Communications Coding*. Springer-Verlag, 1992.

[25] "Wikipedia: The free encyclopedia," *http://www.wikipedia.org/*.

[26] "Mathworld," *http://mathworld.wolfram.com/*.

[27] L. Liebowitz, "A simplified binary arithmetic for the fermat number transform," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 24, no. 5, pp. 356–359, 1976.

[28] R. L. Miller, I. S. Reed, and T. K. Truong, "A theorem for computing primitive elements in the field of complex integers of a characteristic mersenne prime," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-29, no. 1, pp. 119–120, February 1981.

[29] T. Nakamura, "Fast algorithms for computing the powers of a primitive element in GF(q^2)," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-26, no. 4, pp. 376–378, August 1978.

[30] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2.

[31] "IEEE standard for binary floating-point arithmetic (ANSI/IEEE std 754-1985)."

[32] J. A. S. Angus, "Finite field transforms for lossless audio signal processing," AES Preprint 4837, September 1998, presented at the 105th Convention.

[33] MathWorks, "MATLAB version 6.5 documentation."

[34] C. M. Rader and N. M. Brenner, "A new principle for fast fourier transforms," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-24, no. 3, pp. 264–266, June 1976.

[35] R. L. Nevin, "Application of the rader-brenner FFT algorithm to number theoretic transforms," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-25, no. 2, pp. 196–198, April 1977.

[36] *Numerical Recipes in C: The Art of Scientific Computing*. http://www.numerical-recipes.com/: Cambridge University Press.

[37] D. Bailey, "Winograd's algorithm applied to number-theoretic transforms," *Electronics Letters*, vol. 13, no. 18, pp. 548–549, September 1977.

[38] H. J. Nussbaumer, "Overflow detection in the computation of convolutions by some number theoretic transforms," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-26, no. 1, pp. 108–109, February 1978.

AppendixA
Proof of Properties of NTT

Lemma 3 *Let F be a field, $z \in F$ a d^{th} root of unity. Then*

$$\sum_{i=0}^{d-1} z^i = \begin{cases} d & \text{if } z = 1 \\ 0 & \text{otherwise} \end{cases}$$

Proof. $z\left(\sum_{i=0}^{d-1} z^i\right) + 1 = \sum_{i=0}^{d} z^i$, so

$$z\left(\sum_{i=0}^{d-1} z^i\right) = \sum_{i=0}^{d} z^i - 1 = \sum_{i=0}^{d-1} z^i + z^d - 1 = \sum_{i=0}^{d-1} z^i$$

since $z^d = 1$. If $\sum_{i=0}^{d-1} z^i \neq 0$ then can divide both sides above by $\sum_{i=0}^{d-1} z^i$ giving $z = 1$.

Now if $z = 1$, $\sum_{i=0}^{d-1} z^i = \sum_{i=0}^{d-1} 1 = d$. If $z \neq 1$ then must have $\sum_{i=0}^{d-1} z^i = 0$ otherwise the above division yielding $z = 1$ would produce a contradiction. ■

Lemma 4 *Let r be a primitive d^{th} root of unity. Then $\forall k \in \mathbb{Z}$,*

$$\sum_{i=0}^{d-1} r^{ik} = \begin{cases} d & \text{if } k \equiv 0 (\operatorname{mod} d) \\ 0 & \text{otherwise} \end{cases}$$

Proof. Let $r^k = z$. Then result follows trivially from previous lemma. ■

Theorem 5 (Inverse of NTT) *Given the general NTT $A_k = \sum_{j=0}^{d-1} a_j r^{jk}$, the inverse of the transform is given by $a_k = d^{-1} \sum_{j=0}^{d-1} A_j r^{-jk}$.*

Proof.

$$\begin{aligned} d^{-1}\sum_{j=0}^{d-1} A_j r^{-jk} &= d^{-1}\sum_{j=0}^{d-1}\left(\sum_{l=0}^{d-1} a_l r^{lj}\right) r^{-jk} \\ &= d^{-1}\sum_{l=0}^{d-1} a_l \sum_{j=0}^{d-1} r^{j(l-k)} \\ &= d^{-1} a_k d = a_k \end{aligned}$$

since from lemma, $\sum_{j=0}^{d-1} r^{j(l-k)} = d$ if $k = l$ and 0 otherwise. ■

Theorem 6 (Convolution Theorem) *Let $\{y_i\}, \{x_i\}, \{h_i\}$ be d-point sequences and let $\{Y_i\}, \{X_i\}$ and $\{H_i\}$ be their respective NTT's. Then*

$$y = x \circledast h \Leftrightarrow Y = X \cdot H$$

Proof.

$$\begin{aligned}
Y_i &= \sum_{l=0}^{d-1} y_l r^{li} \\
&= \sum_{l=0}^{d-1} \left(\sum_{j=0}^{d-1} h_j x_{(l-j) \bmod d} \right) r^{li} \\
&= \sum_{j=0}^{d-1} h_j \sum_{l=0}^{d-1} x_{(l-j) \bmod d} r^{li} \\
&= \sum_{j=0}^{d-1} h_j \sum_{k=0}^{d-1} x_k r^{(k+j)i} \\
&= \sum_{j=0}^{d-1} h_j r^{ji} \sum_{k=0}^{d-1} x_k r^{ki} \\
&= H_i \cdot X_i
\end{aligned}$$

where in the fourth line the substitution $k = (l - j) \bmod d$ is made. Note that this does not change the limits, since because of the modulo, for any value of j, k still runs through all values $0..d-1$. ■

AppendixB
Chinese Remainder Theorem

Theorem 7 *If $m_1, m_2, ...m_k$ are relatively prime, then the system of congruences, $x \equiv c_i \bmod m_i$ for $i = 1, 2, ...k$ has a unique solution x given by*

$$x = \sum_{i=1}^{k} c_i M_i M_i^{-1}$$

where $m = m_1 m_2 ... m_k = m_1 M_1 = m_2 M_2 = ... = m_k M_k$ and M_i^{-1} uniquely satisfies $(\bmod m_i)$ the congruence

$$M_i M_i^{-1} \equiv 1 \bmod m_i$$

for $i = 1, 2, ...k$

AppendixC
MATLAB Code

C.1 CMNT

C.1.1 Output of Profiler: Computations of Twiddle Factors

Below is the output of the MATLAB profiler, to illustrate the different performance of various methods of calculating the twiddle factors.

```
 time calls acc line
                   1 % Fast Complex Mersenne Number Transform
                   2 %
                   3 % Calculation of twiddle factors
                   4 % Profiling Test
                   5
                   6 function y = cmnt(x)
                   7 % Power of two length sequences only
0.000     1  .     8 d=length(x);
0.110     1  x     9 if( d ~= 2.^nextpow2(d) )
                   10    error('Length of input vector must be a power of 2'
                   11 end
                   12
                   13 % Generate alpha's
                   14 % Primitive root of order 2^(p+1), p=61
0.621     1  x     15 alpha = complex( (uint64(2).^(uint64(2).^uint64(59))) ,
                             ui64(-3).^(uint64(2).^uint64(59)) );
                   16
                   17 %generate r - primtive root of order d
0.171     1  x     18 r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));
                   19
                   20 % Original Method
          1  x     21 for i=1:d
15.252 32768  x    22  twiddle1(i)=r.^uint64(i);
 0.120 32768  .    23 end
                   24
                   25 % Recursive Method
          1  x     26 twiddle2(i)=r;
          1  x     27 for i=2:d
61.288 32767  x    28  twiddle2(i)=r .* twiddle2(i-1);
 0.110 32767  .    29 end
                   30
                   31 % Original Method with Explicit Declaration
          1  x     32 twiddle3=uint64(zeros(1,d));
          1  x     33 for i=1:d
 1.442 32768  x    34     twiddle3(i) = r.^uint64(i);
```

```
        32768  .     35 end
                     36
                     37 % Recursive Method with Explicit Declaration
  0.010      1  x    38 twiddle4=uint64(zeros(1,d));
             1  x    39 twiddle4(1)=r;
             1  x    40 for i=2:d
  0.761  32767  x    41  twiddle4(i) = r .* twiddle4(i-1);
  0.050  32767  .    42 end
```

C.1.2 Comparision of CMNT and FFT Error

The following code was used to generate the plot in Figure **??**.

```
% fig_transform_error.m
% Generation of Plot
% Comparision of CMNT and FFT Error

% Load Data
[in,fs,bits] = wavread('audio/test1_hi_norm_24.wav',[110001 634288]);
% Scale
x=2^(bits-1) .* in';
x64=ui64(x);
d=length(x64);
% FFT
t = cputime;    x_fft = fft(x);                 fft_time   = cputime - t
t = cputime;    x_fft_out = ifft(x_fft);        ifft_time  = cputime - t
% CMNT
t = cputime;    x_cmnt = cmnt(x64);             cmnt_time  = cputime - t
t = cputime;    x_cmnt_out = icmnt(x_cmnt);     icmnt_time = cputime - t
% CMNT2
t = cputime;    temp1 = cmnt2(x64);             cmnt2_time  = cputime - t
t = cputime;    temp2 = (uint64(1)./uint64(d)).*conj(cmnt2(conj(temp1)));       ic
% CMNT3
t = cputime;    temp1 = cmnt3(x64);             cmnt3_time  = cputime - t
% CMNT4
t = cputime;    temp1 = cmnt4(x64,x64);         cmnt4_time  = cputime - t
% CMNTRB
t = cputime;    temp1 = cmntrb(x64);            cmntrb_time  = cputime - t
t = cputime;    temp2 = (uint64(1)./uint64(d)).*conj(cmntrb(conj(temp1)));      ic
% FFT1
t = cputime;    temp1 = fft1(x);                fft1_time  = cputime - t
t = cputime;    temp2 = (1./d).*conj(fft1(conj(temp1)));        ifft1_time = cputi

subplot(3,1,1);
plot( convert(x_cmnt_out) - x );
title('Error in reconstructed signal after CMNT');
xlabel('Samples');
ylabel('Error');
xlim([0 524288]);
```

```
subplot(3,1,2)
plot( real(x_fft_out) - x );
title('Error in reconstructed signal after FFT')
xlabel('Samples');
ylabel('Error');
xlim([0 524288]);

subplot(3,1,3)
plot( in );
title('Original Signal');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 524288]);
```

C.1.3 CMNT Transform Code

C.1.3.1 cmnt.m

```
function y = cmnt(x)
% Fast Complex Mersenne Number Transform
%
% Simple Radix-2 Transform

% Power of two length sequences only
d=length(x);
if( d ~= 2.^nextpow2(d) )
    error('Length of input vector must be a power of 2');
end

% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uir

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = 1;
for i=2:d
    twiddle(i) = r .* twiddle(i-1);
end

% Output
y = cmntrecur(x, twiddle, d);

% -----------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% ----------------------------------------------------

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.1.3.2 cmnt2.m

```
function y = cmnt(x)
% Fast Complex Mersenne Number Transform
%
% Simple Radix-2 Transform
% including n=4 optimization

% Power of two length sequences only
d=length(x);
if( d ~= 2.^nextpow2(d) )
    error('Length of input vector must be a power of 2');
end

% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uint6

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = 1;
for i=2:d
    twiddle(i) = r .* twiddle(i-1);
end
% Output
y = cmntrecur(x, twiddle, d);

% ----------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% ----------------------------------------------------
```

```
function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
          (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.1.3.3 cmnt3.m

```
function y = cmnt(x)
% Fast Complex Mersenne Number Transform
%
% Simple Radix-2 Transform
% including n=4 optimization
% computes real transform as half length complex transform

% Power of two length sequences only
n=length(x);
if( n ~= 2.^nextpow2(n) )
    error('Length of input vector must be a power of 2');
```

```
end
d=n/2;

% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uint64

% Generate r - primtive root of order n
r = alpha .^ ((uint64(2).^uint64(62))./uint64(n));

% Split first and second half into complex and real part
x_comp = complex( x(1:2:n) , x(2:2:n) );

% Table of twiddle factors
twiddle=uint64(zeros(1,n));
twiddle(1) = 1;
for i=2:n
    twiddle(i) = r .* twiddle(i-1);
end

% Transform
y_comp = cmntrecur(x_comp, twiddle(1:2:n), d);

% Reconstruct Real Output
% Declare arrays (most important; dynamic allocation kills performance)
yT = uint64(zeros(1,d));
yB = yT;

i2 = uint64(1) ./ uint64(2); % Precompute 1/2 (multiplication cheaper than divisio
% Want real and imaginary part of y_comp(1) here
% done this way to get multiplication by i2 out of the loop
% Quicker to do vector .* scalar than to have scalar mult in the loop.
yT(1) = y_comp(1) + conj(y_comp(1));
yB(1) = conj(y_comp(1)) - y_comp(1);

for m = 2:d
        a=y_comp(m);
        b=conj(y_comp((d+2)-m));
        yT(m) = (a + b);
        yB(m) = (b - a);
end

yT = yT .* i2;
yB = yB .* complex(uint64(0),i2);

fiddle = 1;
z = twiddle(1:fiddle:fiddle*d).*yB;
y = [(yT+z) (yT-z)];

% -------------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% -------------------------------------------------------

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
          (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.1.3.4 cmnt4.m

```
function y = cmnt(x,y)
% Fast Complex Mersenne Number Transform
%
% Simple Radix-2 Transform
% including n=4 optimization
% computes two real transforms as complex function

% Power of two length sequences only
```

```
nx=length(x);
ny=length(y);

if( nx ~= ny)
    error('Vectors must be same length');
end
n=nx;

if( n ~= 2.^nextpow2(n) )
    error('Length of input vectors must be a power of 2');
end

% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uint6

% Generate r - primtive root of order n
r = alpha .^ ((uint64(2).^uint64(62))./uint64(n));

% Split first and second half into complex and real part
x_comp = complex( x , y );

% Table of twiddle factors
twiddle=uint64(zeros(1,n));
twiddle(1) = 1;
for i=2:n
    twiddle(i) = r .* twiddle(i-1);
end

% Transform
y_comp = cmntrecur(x_comp, twiddle, n);

% Reconstruct Real Output
% Declare arrays (most important; dynamic allocation kills performance)
X = uint64(zeros(1,n));
Y = X;

i2 = uint64(1) ./ uint64(2); % Precompute 1/2 (multiplication cheaper than divisi
% Want real and imaginary part of y_comp(1) here
% done this way to get multiplication by i2 out of the loop
% Quicker to do vector .* scalar than to have scalar mult in the loop.
X(1) = y_comp(1) + conj(y_comp(1));
Y(1) = conj(y_comp(1)) - y_comp(1);

for m = 2:n
        a=y_comp(m);
        b=conj(y_comp((n+2)-m));
        X(m) = (a + b);
        Y(m) = (b - a);
end
```

```
y = [(X.*i2) (Y.*complex(uint64(0),i2))];

% ------------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% ------------------------------------------------------

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
          (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.1.3.5 cmntrb.m

```
% Fast Complex Mersenne Number Transform
%
% Rader-Brenner Algorithm
% always returns 1 x n, regardless of input
```

```
function y = cmntrb(x)

% Power of two length sequences only
d=length(x);
if( d ~= 2.^nextpow2(d) )
    error('Length of input vector must be a power of 2');
end

% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uint64

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = uint64(1);
for k=2:d
    twiddle(k) = r .* twiddle(k-1);
end

twiddle = imag( twiddle ./ ((twiddle.*twiddle) - uint64(ones(1,d)) ));

% Output
y = cmntrecur(x, twiddle, d);

% ---------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% ---------------------------------------------------

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
```

```
        Lr = I - J;
        L = complex(-imag(Lr),real(Lr));
        M = r8conj(F);
        N = r8(D);
        O = r8conj(D);
        P = r8(F);

        y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
              (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
    else
            fiddle=d/n;
            m = n/2;                  % length of next level transform
            t = uint64(zeros(1,m)); % setup arrays (dynamic allocation slow)

            Q = sum(x(2:2:n));        % constant term

            % Setup first subsequence
            t = [ x(1) x(3:2:n) ];

            % Transform first sub-sequences
            B = cmntrecur(t, table, d);

            % Setup + transform second subsequence
            t = [ (x(2) - x(n)) (x(4:2:n) - x(2:2:n-2)) ]; %c
            C = cmntrecur(t, table, d);

            t = table(1:fiddle:d/2);

            % Recreate Output
            D = complex(uminus(t .* imag(C)), t.*real(C));
            y = [(B-D) (B+D)];
            y(1) = B(1) + Q;
            y(m+1) = B(1) - Q;
    end
```

C.2 Convolution

C.2.1 Simple Comb Filter Demonstration

The following code was used to generate the plots in Figures **??**,**??**.

```
    % fig_convolution1.m
    % Generation of Plot
    % Comparision of CMNT and FFT Error

    % Load Signal
    [in,fs,bits] = wavread('audio/test1_hi_norm_24.wav',[120000 644288]);
```

```
% Scale
x=2^(bits-1) .* in';
x=x(1:32768);
% Create Comb Filter
h=zeros(1,65536);
h(1)=128;
h(8000)=64;
h(16000)=32;
h(24000)=16;
% Zero Pad Signal
x=[x zeros(1,32768)];

% CMNT Calculations
X_64 = cmnt(ui64(x));
H_64 = cmnt(ui64(h));
Y_64 = X_64 .* H_64;
y_cmnt = convert(real(icmnt(Y_64)));

% FFT Calculations
X_fft = fft(x);
H_fft = fft(h);
Y_fft = X_fft .* H_fft;
y_fft = real(ifft(Y_fft));

figure(1)
subplot(3,1,1)
plot(h)
title('Impulse Response of Comb Filter');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 65536]);

subplot(3,1,2)
plot(x)
title('Audio Signal');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 65536]);

subplot(3,1,3)
plot(y_cmnt)
title('Filtered Signal');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 65536]);

% CMNT
Y_64 = cmnt(ui64(y_cmnt));
X_64_2 = Y_64 ./ H_64;
```

```
x_cmnt = convert(real(icmnt(X_64_2)));
% FFT
Y_fft = fft(y_fft);
X_fft = Y_fft ./ H_fft;
x_fft = real(ifft(X_fft));

% Plot Errors
figure(2)
subplot(2,1,1)
plot(abs(x_fft - x))
title('Absolute difference from FFT filtering');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 65536]);

subplot(2,1,2)
plot(abs(x_cmnt - x))
title('Absolute difference from CMNT filtering');
xlabel('Samples');
ylabel('Amplitude');
xlim([0 65536]);
```

C.2.2 Convolution Times

The following code was used to calculate the times for the different convolution methods.

```
% fig_convolution1.m
% Generation of Plot
% Comparision of CMNT and FFT Error

% Load Signal
[in,fs,bits] = wavread('audio/test1_hi_norm_24.wav',[120001 644288]);
% Scale
x=2^(bits-1) .* in';
x=x(1:32768);

% Load Impulse Response
[in,fs,bits] = wavread('audio/flutter_echo_ir_stereo_24.wav');
h=2^(bits-1) .* in(1:4096,1)';

% fft convolution
t = cputime;    temp = conv(x,h);                        conv_time = cputime - t
% conv1
t = cputime;    temp = cmntconv1(x,h);                   conv1_time = cputime -
% conv2
t = cputime;    temp = cmntconv2(x,h);                   conv2_time = cputime -
% conv3
t = cputime;    temp = cmntconv3(x,h);                   conv3_time = cputime -
% conv4
```

```
t = cputime;    temp = cmntconv4(x,h);                        conv4_time = cputime - t
% fftfilt
t = cputime;    temp = fftfilt(h,x);                          fftfilt_time = cputime - t
% cmntfilt
t = cputime;    temp = cmntfilt(h,x);                         cmntfilt_time = cputime -
```

C.2.3 Convolution Code

C.2.3.1 cmntconv1.m

```
function y = conv1(signal,filter)
%
% CONV1(x,h)
%       Simple Overlap Add Block Convolution Algorithm Using CMNT.
%       Filter Length must be a factor of two, due to requirements of CMNT.
%       Version 1: uses cmnt function

%rows/columns
Lx = length(signal); %signal length
Lh = length(filter); %filter length
Lt = Lx+Lh-1;        %output length

if( Lh~= 2.^nextpow2(Lh) )
    error('Length of impulse response must be a power of 2');
end
padding = zeros(1,Lh); %zero padding equal to filter length
h=ui64([filter padding]); %pad filter with zeros, same length as filter
H = cmnt(h); %Transform filter - only need do this once!!

num = floor(Lx/Lh); %number of complete times to filter
leftover = mod(Lx,Lh); % partial remainder left to filter

y = uint64(zeros(1,Lt)); %create empty output length of signal + length of filter-

for n = 1:num
    x = ui64([signal(1+(n-1)*Lh:Lh*n) padding]); %extract section of signal and pa
    X = cmnt(x); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = icmnt(XH); % inverse transform back to time domain
    y(1+(n-1)*Lh:((n-1)*Lh)+((2*Lh)-1)) = y(1+(n-1)*Lh:((n-1)*Lh)+((2*Lh)-1))+xh(1
end

if leftover ~= 0
    start=1+(n*Lh);
    x = ui64([signal(start:Lx) zeros(1,(Lh-leftover)) padding]); %extract section
    X = cmnt(x); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = icmnt(XH); % inverse transform back to time domain
    y(start:Lt) = y(start:Lt)+xh(1:(Lt-start)+1); % add new section into output
end

y = convert(y);
```

C.2.3.2 cmntconv2.m

```
function y = conv2(signal,filter)
%
% CONV2(x,h)
%       Simple Overlap Add Block Convolution Algorithm Using CMNT.
%       Filter Length must be a factor of two, due to requirements of CMNT.
%       Version 2: implements transform internally to reuse table of twiddle fa


Lx = length(signal); %signal length
Lh = length(filter); %filter length
Lt = Lx+Lh-1;        %output length

if( Lh ~= 2.^nextpow2(Lh) )
    error('Length of impulse response must be a power of 2');
end

padding = zeros(1,Lh); %zero padding equal to filter length
h=ui64([filter padding]); %pad filter with zeros, same length as filter
d=Lh*2; %length of transform

% Precompute Twiddle Factors
% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uin

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = 1;
for i=2:d
    twiddle(i) = r .* twiddle(i-1);
end

H = cmntrecur(h,twiddle,d); %Transform filter - only need do this once!!

num = floor(Lx/Lh); %number of complete times to filter
leftover = mod(Lx,Lh); % partial remainder left to filter

y = uint64(zeros(1,Lt)); %create empty output length of signal + length of filt
```

```
for n = 1:num
    x = ui64([signal(1+(n-1)*Lh:Lh*n) padding]); %extract section of signal and pa
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse tr
    y(1+(n-1)*Lh:((n-1)*Lh)+(d-1)) = y(1+(n-1)*Lh:((n-1)*Lh)+(d-1))+xh(1:d-1); % a
    %num-n
end

if leftover ~= 0
    start=1+(n*Lh);
    x = ui64([signal(start:Lx) zeros(1,(Lh-leftover)) padding]); %extract section
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse tr
    y(start:Lt) = y(start:Lt)+xh(1:(Lt-start)+1); % add new section into output, a
end

y = convert(y);

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.2.3.3 cmntconv3.m

```
function y = conv3(signal,filter)
%
% CONV3(x,h)
%       Simple Overlap Add Block Convolution Algorithm Using CMNT.
%       Filter Length must be a factor of two, due to requirements of CMNT.
%       Version 3:      implements transform internally to reuse table of twiddle
%                       includes n=4 optimisation


Lx = length(signal); %signal length
Lh = length(filter); %filter length
Lt = Lx+Lh-1;        %output length
```

```
if( Lh ~= 2.^nextpow2(Lh) )
    error('Length of impulse response must be a power of 2');
end

padding = zeros(1,Lh); %zero padding equal to filter length
h=ui64([filter padding]); %pad filter with zeros, same length as filter
d=Lh*2; %length of transform

% Precompute Twiddle Factors
% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uin

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = 1;
for i=2:d
    twiddle(i) = r .* twiddle(i-1);
end

H = cmntrecur(h,twiddle,d); %Transform filter - only need do this once!!

num = floor(Lx/Lh); %number of complete times to filter
leftover = mod(Lx,Lh); % partial remainder left to filter

y = uint64(zeros(1,Lt)); %create empty output length of signal + length of filt

for n = 1:num
    x = ui64([signal(1+(n-1)*Lh:Lh*n) padding]); %extract section of signal anc
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse
    y(1+(n-1)*Lh:((n-1)*Lh)+(d-1)) = y(1+(n-1)*Lh:((n-1)*Lh)+(d-1))+xh(1:d-1);
end

if leftover ~= 0
    start=1+(n*Lh);
    x = ui64([signal(start:Lx) zeros(1,(Lh-leftover)) padding]); %extract secti
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse
    y(start:Lt) = y(start:Lt)+xh(1:(Lt-start)+1); % add new section into output
end

y = convert(y);
```

```
function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
          (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.2.4 cmntconv4.m

```
function y = conv4(signal,filter)
%
% CONV4(x,h)
%       Simple Overlap Add Block Convolution Algorithm Using CMNT.
%       Filter Length must be a factor of two, due to requirements of CMNT.
%       Version 3:      implements transform internally to reuse table of twiddle
%                       includes n=4 optimisation
%                       computes two real blocks as one complex transform
```

```
Lx = length(signal); %signal length
Lh = length(filter); %filter length
Lt = Lx+Lh-1;        %output length

if( Lh ~= 2.^nextpow2(Lh) )
    error('Length of impulse response must be a power of 2');
end

padding = zeros(1,Lh); %zero padding equal to filter length
h=ui64([filter padding]); %pad filter with zeros, same length as filter
d=Lh*2; %length of transform

% Precompute Twiddle Factors
% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uin

% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(d));

% Table of twiddle factors
twiddle=uint64(zeros(1,d));
twiddle(1) = 1;
for i=2:d
    twiddle(i) = r .* twiddle(i-1);
end

H = cmntrecur(h,twiddle,d); %Transform filter - only need do this once!!

num = floor(Lx/Lh); % number of complete times to filter
num2 = floor(num/2); % number of pairs of blocks
leftover = mod(Lx,Lh); % partial remainder left to filter

y = uint64(zeros(1,Lt)); %create empty output length of signal + length of filt

for n = 1:2:num2*2      % calculate for pairs of blocks
    x1 = ui64([signal(1+(n-1)*Lh:Lh*n) padding]); % extract 2 blocks
    x2 = ui64([signal(1+(n)*Lh:Lh*(n+1)) padding]);
    X = cmntrecur(complex(x1,x2),twiddle,d); % transform as complex signal
    XH = X.*H; % convolution
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d));
    %xh1 = real((uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d))); %
    %xh2 = imag((uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d))); %
    y(1+(n-1)*Lh:((n-1)*Lh)+(d-1)) = y(1+(n-1)*Lh:((n-1)*Lh)+(d-1))+real(xh(1:d
    y(1+n*Lh:(n*Lh)+(d-1)) = y(1+n*Lh:(n*Lh)+(d-1))+imag(xh(1:d-1)); % add bloc
end

if (num/2) ~= num2      % one more complete block
    n=num;
    x = ui64([signal(1+(n-1)*Lh:Lh*n) padding]); %extract section of signal and
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse tr
    y(1+(n-1)*Lh:((n-1)*Lh)+(d-1)) = y(1+(n-1)*Lh:((n-1)*Lh)+(d-1))+xh(1:d-1); % a
end

if leftover ~= 0
    n=num;
    start=1+(n*Lh);
    x = ui64([signal(start:Lx) zeros(1,(Lh-leftover)) padding]); %extract section
    X = cmntrecur(x,twiddle,d); %transform section of signal
    XH = X.*H; %multiply i.e. convolution in time domain = mult in freq domain
    xh = (uint64(1)./uint64(d)).*conj(cmntrecur(conj(XH),twiddle,d)); % inverse tr
    y(start:Lt) = y(start:Lt)+xh(1:(Lt-start)+1); % add new section into output, a
end

y = convert(y);

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B) (C + N - E + M) (K + L) (C - P + E - O) ...
          (A-B) (C - N - E - M) (K - L) (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z) (yT-z)];
end
```

C.2.4.1 cmntfilt.m

```
function y = cmntfilt(b_in,x_in,ncmnt)

error(nargchk(2,3,nargin));

% Check inputs are vectors, and ensure column
[mx,n] = size(x_in);
if mx == 1
    x_in = x_in(:);     % turn row into a column
elseif n ~=1
    error('Function only accepts vector inputs')
end
nx = size(x_in,1);

[mb,n] = size(b_in);
if mb == 1
    b_in = b_in(:);     % turn row into a column
elseif n ~=1
    error('Function only accepts vector inputs')
end
nb = size(b_in,1);

if nargin < 3
% figure out which ncmnt and L to use - same as fftfilt
    if nb >= nx     % take a single FFT in this case
        ncmnt = 2^nextpow2(nb+nx-1);
        L = nx;
    else
        fftflops= [ 18 59 138 303 660 1441 3150 6875 14952 32373 69762 ...
       149647 319644 680105 1441974 3047619 6422736 13500637 28311786 59244791
        n = 2.^(1:20);
        validset = find(n>(nb-1));   % must have ncmnt > (nb-1)
        n = n(validset);
        fftflops = fftflops(validset);
        % minimize (number of blocks) * (number of flops per cmnt)
        L = n - (nb - 1);
        [dum,ind] = min( ceil(nx./L) .* fftflops );
        ncmnt = n(ind);
        L = L(ind);
    end

else  % ncmnt is given
    if ncmnt < nb
```

```
        ncmnt = nb;
    end
    ncmnt = 2.^(ceil(log(ncmnt)/log(2))); % force this to a power of 2 for speed
    L = ncmnt - nb + 1;
end

% Convert to Finite Field
x = ui64(x_in);
b = ui64([b_in; zeros(ncmnt-nb,1)]); %convert and pad

% Precompute twiddle factors for transform
% Generate alpha - primitive root of order 2^(p+1), (p=61)
alpha = complex( (uint64(2).^(uint64(2).^uint64(59))),ui64(-3).^(uint64(2).^uint64
% Generate r - primtive root of order d
r = alpha .^ ((uint64(2).^uint64(62))./uint64(ncmnt));
% Table of twiddle factors
twiddle=uint64(zeros(ncmnt,1));
twiddle(1) = 1;
for i=2:ncmnt
    twiddle(i) = r .* twiddle(i-1);
end

B = cmntrecur(b, twiddle, ncmnt);

y = uint64(zeros(size(x)));

istart = 1;
while istart <= nx
    iend = min(istart+L-1,nx);
    if (iend - istart) == 0
        X = x(istart(ones(ncmnt,1)),:);  % need to fft a scalar
    else
%       block = [x(istart:iend,:); zeros(ncmnt-(iend-istart),1)];
        X = cmntrecur([x(istart:iend,:); zeros(ncmnt-(iend-istart+1),1)], twiddle,
    end
    Y=(uint64(1)./uint64(ncmnt)).*conj(cmntrecur(conj(X.*B), twiddle, ncmnt));
    yend = min(nx,istart+ncmnt-1);
    y(istart:yend,:) = y(istart:yend,:) + Y(1:(yend-istart+1),:);
    istart = istart + L;
end

if ~any(imag(b_in)) & ~any(imag(x_in))
        y = real(y);
end

y=convert(y);

if (mx == 1)&(size(y,2) == 1)
    y = y(:).';    % turn column back into a row
```

```
end


% ------------------------------------------------------
% cmntrecur - Recursive Subroutine to Implement radix-2
% ------------------------------------------------------

function y = cmntrecur(x, table, d)

n=length(x);
if (n==1)
    y=x;
elseif (n==8)
    C = x(1) - x(5);
    D = x(2) - x(6);
    Er = x(3) - x(7);
    E = complex(-imag(Er),real(Er));
    F = x(8) - x(4);
    G = x(1) + x(5);
    H = x(3) + x(7);
    I = x(4) + x(8);
    J = x(2) + x(6);
    A = G + H;
    B = I + J;
    K = G - H;
    Lr = I - J;
    L = complex(-imag(Lr),real(Lr));
    M = r8conj(F);
    N = r8(D);
    O = r8conj(D);
    P = r8(F);

    y = [ (A+B); (C + N - E + M); (K + L); (C - P + E - O); ...
          (A-B); (C - N - E - M); (K - L); (C + P + E + O)];
else
    m = n/2;
    yT=cmntrecur(x(1:2:n),table, d);
    yB=cmntrecur(x(2:2:n),table, d);
    fiddle = d/n;
    z = table(1:fiddle:fiddle*m).*yB;
    y = [(yT+z); (yT-z)];
end
```

AppendixD
C Code

The following C code, which resides in the `@uint64` directory, forms the core of the finite field framework.

D.0.4.2 conj.c

```
/*
 * ==============================================================
 Overloading the conj() function for uint64 elements of GF(q^2)


 Robin Ince
 * ==============================================================
 */


#include "i64.h"
#include "mex.h"

/* The gateway routine */
void
mexFunction( int nlhs, mxArray *plhs[],  int nrhs, const mxArray *prhs[])
{
    int i, n;
    mod_t *pr, *pi;

    /* Check for proper number of input and output arguments */
    if (nrhs != 1) {
        mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument  */
    if (!(mxIsUint64(prhs[0]))){
        mexErrMsgTxt("Input argument must be of type uint64.");
    }

        /* Create output array */
    plhs[0] = mxDuplicateArray(prhs[0]);

    pr = (mod_t *) mxGetData(plhs[0]);
    pi = (mod_t *) mxGetImagData(plhs[0]);
    n = mxGetNumberOfElements(plhs[0]);

    /* If there is an imaginary part of the input, negate it (bitwise NOT) */
    if(pi != NULL) {
        for(i=0; i < n; i++) {
            pi[i] = ((~ pi[i]) & M61);
            }
        }

}
```

D.0.4.3 convert.c

```
/*
 * ================================================================
 Convert usigned 64 bit representation of GF(q^2) element to signed representat
 (double)

 Robin Ince
 * ================================================================
 */


#include "i64.h"
#include "mex.h"

double gf2double(mod_t x) {

        if( (x & SIGN_BIT) !=0 )
                return( (double) (((int64) x) - M61 ));
        else
                return( (double) (int64) x);
}

/* The gateway routine */
void
mexFunction( int nlhs, mxArray *plhs[],  int nrhs, const mxArray *prhs[])
{
    int i, n, ndim, *dims;
    mod_t *pr, *pi;
        double *zr, *zi;

    /* Check for proper number of input and output arguments */
    if (nrhs != 1) {
        mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument  */
    if (!(mxIsUint64(prhs[0]))){
        mexErrMsgTxt("Input argument must be of type uint64.");
    }

    pr = (mod_t *) mxGetData(prhs[0]);
    pi = (mod_t *) mxGetImagData(prhs[0]);
    n = mxGetNumberOfElements(prhs[0]);
    ndim=mxGetNumberOfDimensions(prhs[0]);
    dims=mxGetDimensions(prhs[0]);

        if(pi==NULL) {
            plhs[0] = mxCreateNumericArray(ndim, dims, mxDOUBLE_CLASS, mxREAL);
        }
        else {
                plhs[0] = mxCreateNumericArray(ndim, dims, mxDOUBLE_CLASS, mxCOMPI
        }
        zr=mxGetPr(plhs[0]);
        zi=mxGetPi(plhs[0]);

    /* Perform task... */
    for(i=0; i < n; i++) {
                zr[i]=gf2double(pr[i]);
    }

        if(pi!=NULL) {
    for(i=0; i < n; i++) {
                zi[i]=gf2double(pi[i]);
    }
        }
}
```

D.0.4.4 getq.c

```
/*
 * ============================================================
 Return modulus - Q = 0x1FFFFFFFFFFFFFFF


 Robin Ince
 * ============================================================
 */


#include "i64.h"
#include "mex.h"

/* The gateway routine */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  mod_t *z;
  int ndim=2;
  int dims[2];
  dims[0]=1;
  dims[1]=1;

  if (nlhs > 1)
        mexErrMsgTxt("Too many outputs");


  /* Set the output pointer to the output matrix. */
  plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mxREAL);

  /* Create a C pointer to a copy of the output matrix. */
  z = (mod_t *) (mxGetData(plhs[0]));

  // output q
  *z = M61;
}
```

D.0.4.5 imag.c

```
/*
 * ================================================================
 Overloading the imag() function for uint64 elements of GF(q^2)


 Robin Ince
 * ================================================================
 */


#include "i64.h"
#include "mex.h"

/* The gateway routine */
void
mexFunction( int nlhs, mxArray *plhs[],  int nrhs, const mxArray *prhs[])
{
    int i, n, ndim;
        int *dims;
    mod_t *pi, *z;

    /* Check for proper number of input and output arguments */
    if (nrhs != 1) {
        mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument   */
    if (!(mxIsUint64(prhs[0]))){
        mexErrMsgTxt("Input argument must be of type uint64.");
    }

        ndim=mxGetNumberOfDimensions(prhs[0]);
        dims=mxGetDimensions(prhs[0]);
        pi=(mod_t*) mxGetImagData(prhs[0]);

    plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mxREAL);

    z=(mod_t*) mxGetData(plhs[0]);
    n = mxGetNumberOfElements(plhs[0]);

    /* If there is an imaginary part of the input */
    if(pi != NULL) {
        for(i=0; i < n; i++) {
            z[i] = pi[i];
            }
        }
        else {
        for(i=0; i < n; i++) {
            z[i] = 0;
            }
        }

}
```

D.0.4.6 minus.c

```
/*
 * =============================================================
 Overloading the - operator for uint64 elements of GF(q^2)

 Robin Ince
 * =============================================================
 */



#include "i64.h"
#include "mex.h"

mod_t submod(mod_t x, mod_t y)
{
  // add x + -y
  mod_t temp = x + ((~ y) & M61);
  // modulo reduction
  return( (temp & M61) + (temp >> 61));
}

/* The gateway routine */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  mod_t *xr, *xi, *yr, *yi, *zr, *zi;
  int ndimx, ndimy, i, n;
  int *dimsx, *dimsy;

  /*  Check for proper number of arguments. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
     get to the else statement if mexErrMsgTxt is executed.
     (mexErrMsgTxt breaks you out of the MEX-file.)
  */
  if (nrhs != 2)
    mexErrMsgTxt("Two inputs required.");
  if (nlhs > 1)
    mexErrMsgTxt("Too many output arguments");

  if (!mxIsUint64(prhs[0]) || !mxIsUint64(prhs[1])) {
    mexErrMsgTxt("Inputs must be of type uint64");
  }

  /* Check to make sure the input arguments have same dimensions */
  ndimx=mxGetNumberOfDimensions(prhs[0]);
  ndimy=mxGetNumberOfDimensions(prhs[1]);
  dimsx=mxGetDimensions(prhs[0]);
  dimsy=mxGetDimensions(prhs[1]);

  if ( ndimx != ndimy )
         mexErrMsgTxt("Number of array dimensions must match");

  /* */
  for (i=0; i<ndimx; i++) {
          if (dimsx[i] != dimsy[i])
                  mexErrMsgTxt("Matrix dimensions must agree.");
  }

  xr = (mod_t*) mxGetData(prhs[0]);
  xi = (mod_t*) mxGetImagData(prhs[0]);
  yr = (mod_t*) mxGetData(prhs[1]);
  yi = (mod_t*) mxGetImagData(prhs[1]);

  if( (xi==NULL) && (yi==NULL) ) {
          plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxREAL);
```

```
            zr= (mod_t*) mxGetData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++)
                    zr[i]=submod(xr[i],yr[i]);
    }
    else if( (xi==NULL) ) {
            plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);
            zr= (mod_t*) mxGetData(plhs[0]);
            zi= (mod_t*) mxGetImagData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++) {
                    zr[i]=submod(xr[i],yr[i]);
                    zi[i]=((~ yi[i]) & M61);
            }
    }
    else if( (yi==NULL) ) {
            plhs[0] = mxDuplicateArray(prhs[0]);
            zr= (mod_t*) mxGetData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++)
                    zr[i]=submod(xr[i],yr[i]);
    }
    else {
            plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);
            zr= (mod_t*) mxGetData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            zi= (mod_t*) mxGetImagData(plhs[0]);
            for(i=0;i<n;i++) {
                    zr[i]=submod(xr[i],yr[i]);
                    zi[i]=submod(xi[i],yi[i]);
            }
    }

}
```

D.0.4.7 plus.c

```
/*
 * ==============================================================
 Overloading the + operator for uint64 elements of GF(q^2)

 Robin Ince
 * ==============================================================
 */



#include "i64.h"
#include "mex.h"
```

```
mod_t addmod(mod_t x, mod_t y)
{
        mod_t temp=x+y;
        return((temp & M61) + (temp >> 61));
}

/* The gateway routine */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  mod_t *xr, *xi, *yr, *yi, *zr, *zi;
  int ndimx, ndimy, i, n;
  int *dimsx, *dimsy;

  /*  Check for proper number of arguments. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
     get to the else statement if mexErrMsgTxt is executed.
     (mexErrMsgTxt breaks you out of the MEX-file.)
  */
  if (nrhs != 2)
    mexErrMsgTxt("Two inputs required.");
  if (nlhs > 1)
    mexErrMsgTxt("Too many output arguments");

  if (!mxIsUint64(prhs[0]) || !mxIsUint64(prhs[1])) {
    mexErrMsgTxt("Inputs must be of type uint64");
  }

  /* Check to make sure the input arguments have same dimensions */
  ndimx=mxGetNumberOfDimensions(prhs[0]);
  ndimy=mxGetNumberOfDimensions(prhs[1]);
  dimsx=mxGetDimensions(prhs[0]);
  dimsy=mxGetDimensions(prhs[1]);

  if ( ndimx != ndimy )
        mexErrMsgTxt("Number of array dimensions must match");

  /* */
  for (i=0; i<ndimx; i++) {
        if (dimsx[i] != dimsy[i])
                mexErrMsgTxt("Matrix dimensions must agree.");
  }

  xr = (mod_t*) mxGetData(prhs[0]);
  xi = (mod_t*) mxGetImagData(prhs[0]);
  yr = (mod_t*) mxGetData(prhs[1]);
  yi = (mod_t*) mxGetImagData(prhs[1]);
```

```
    zi=NULL;

    if( (xi==NULL) && (yi==NULL) ) {
            plhs[0] = mxDuplicateArray(prhs[0]);
            zr= (mod_t*) mxGetData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++)
                    zr[i]=addmod(xr[i],yr[i]);
    }
    else if( (xi==NULL) ) {
            plhs[0] = mxDuplicateArray(prhs[1]);
            zr= (mod_t*) mxGetData(plhs[0]);
            zi= (mod_t*) mxGetImagData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++)
            {
                    zr[i]=addmod(xr[i],yr[i]);
                    zi[i]=yi[i];
            }
    }
    else if( (yi==NULL) ) {
            plhs[0] = mxDuplicateArray(prhs[0]);
            zr= (mod_t*) mxGetData(plhs[0]);
            zi= (mod_t*) mxGetImagData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            for(i=0;i<n;i++)
            {
                    zr[i]=addmod(xr[i],yr[i]);
                    zi[i]=xi[i];
            }
    }
    else {
            plhs[0] = mxDuplicateArray(prhs[0]);
            zr= (mod_t*) mxGetData(plhs[0]);
            n = mxGetNumberOfElements(plhs[0]);
            zi= (mod_t*) mxGetImagData(plhs[0]);
            for(i=0;i<n;i++) {
                    zr[i]=addmod(xr[i],yr[i]);
                    zi[i]=addmod(xi[i],yi[i]);
            }
    }

}
```

D.0.4.8 r8.c

```
/*
 * ================================================================
 Special function for multiplication by r_8 (2^30)
 Robin Ince
 * ============================================================
 */


#include "i64.h"
#include "mex.h"



mod_t addmod(mod_t x, mod_t y)
{
  // addition
  mod_t temp = x + y;
  // modulo reduction
  return( (temp & M61) + (temp >> 61));
}

mod_t submod(mod_t x, mod_t y)
{
  // add x + -y
  mod_t temp = x + ((~ y) & M61);
  // modulo reduction
  return( (temp & M61) + (temp >> 61));
}

mod_t timesr(mod_t x)
{
  /* multiply by 2^30 and reduce */
  mod_t temp = ((x << 30) & M61);
  temp += (x >> 31);
  /*return( (temp & M61) + (temp >> 61));*/
  return(temp);
}

/* The gateway routine */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  mod_t *xr, *xi, *yr, *yi, *zr, *zi;
  mod_t temp1, temp2;
  int ndimx, ndimy, i, n, ndim;
  int *dimsx, *dimsy, *dims;
  mxArray *temp;

  /*  Check for proper number of arguments. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
```

```
        get to the else statement if mexErrMsgTxt is executed.
        (mexErrMsgTxt breaks you out of the MEX-file.)
     */
     if (nrhs != 1)
       mexErrMsgTxt("One input required.");
     if (nlhs > 1)
       mexErrMsgTxt("Too many output arguments");

     if (!mxIsUint64(prhs[0])) {
       mexErrMsgTxt("Inputs must be of type uint64");
     }

     xr = (mod_t*) mxGetData(prhs[0]);
     xi = (mod_t*) mxGetImagData(prhs[0]);

     ndimx=mxGetNumberOfDimensions(prhs[0]);
     dimsx=mxGetDimensions(prhs[0]);

     /* create output array */
     plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);

     if( (xi==NULL) ) {    /* real input */
             zr= (mod_t*) mxGetData(plhs[0]);
             zi= (mod_t*) mxGetImagData(plhs[0]);
             n = mxGetNumberOfElements(plhs[0]);
             for(i=0;i<n;i++) {
                     zr[i] = timesr(xr[i]);
                     zi[i] = ((~zr[i]) & M61);
             }
     }
     else {                /* complex input */
             zr= (mod_t*) mxGetData(plhs[0]);
             zi= (mod_t*) mxGetImagData(plhs[0]);
             n = mxGetNumberOfElements(plhs[0]);
             for(i=0;i<n;i++) {
                     temp1 = timesr(xr[i]);
                     temp2 = timesr(xi[i]);
                     zr[i] = addmod( temp1,temp2 );
                     zi[i] = submod( temp2,temp1 );
             }
     }

   }
```

D.0.4.9 sum.c

```
   /*
    * ============================================================
    Sum - just sum's all the elements in the argument (should be a vector)
```

```
 (61 bit ones' compliment)

 Robin Ince
 * ===============================================================
 */


#include "i64.h"
#include "mex.h"

/* The gateway routine */
void
mexFunction( int nlhs, mxArray *plhs[],  int nrhs, const mxArray *prhs[])
{
    int i, n, ndimx, dims[2], k;
    int *dimsx;
    mod_t *xr, *xi, *zr, *zi;
    int ndim=2;
    dims[0]=1;
    dims[1]=1;

    /* Check for proper number of input and output arguments */
    if (nrhs != 1) {
        mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument  */
    if (!(mxIsUint64(prhs[0]))){
        mexErrMsgTxt("Input argument must be of type uint64.");
    }

    xr = (mod_t *) mxGetData(prhs[0]);
    xi = (mod_t *) mxGetImagData(prhs[0]);
    n = mxGetNumberOfElements(prhs[0]);

        /* Real Input... */
    if(xi == NULL) {
        plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mxREAL);
        zr=(mod_t*) mxGetData(plhs[0]);
        zr[0] = 0;
        for(i=0; i < n; i++) {
            zr[0] += xr[i];
            k++;
            /* only need to reduce every 3 additions */
            if(k > 2) {
                    zr[0] = (zr[0] & M61) + (zr[0] >> 61);
                    k=0;
            }
            }
        if(k != 0)      /* reduction outstanding */
            zr[0] = (zr[0] & M61) + (zr[0] >> 61);
        }

    else {      /* Complex Input... */
        plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mxCOMPLEX);
        zr=(mod_t*) mxGetData(plhs[0]);
        zi=(mod_t*) mxGetImagData(plhs[0]);
        zr[0] = 0;
        zi[0] = 0;
        for(i=0; i < n; i++) {
            zr[0] += xr[i];
            zi[0] += xi[i];
            k++;
            /* only need to reduce every 3 additions */
            if(k > 2) {
                    zr[0] = (zr[0] & M61) + (zr[0] >> 61);
                    zi[0] = (zi[0] & M61) + (zi[0] >> 61);
                    k=0;
            }
            }
        if(k != 0)      /* reduction outstanding */
            zr[0] = (zr[0] & M61) + (zr[0] >> 61);
            zi[0] = (zi[0] & M61) + (zi[0] >> 61);
        }
}
```

D.0.4.10 times.c

```
/*
 * ===============================================================
 Overloading the .* operator for uint64 elements of GF(q^2)

 Robin Ince
 * ===============================================================
 */


#include "i64.h"
#include "mex.h"



mod_t addmod(mod_t x, mod_t y)
{
  // addition
  mod_t temp = x + y;
  // modulo reduction
  return( (temp & M61) + (temp >> 61));
}


mod_t submod(mod_t x, mod_t y)
{
  // add x + -y
  mod_t temp = x + ((~ y) & M61);
  // modulo reduction
  return( (temp & M61) + (temp >> 61));
}


mod_t rmultmod(mod_t x, mod_t y)
{
        // uint68 * uint64 -> uint128 multiplication
        uint32 a[2],b[2];
        uint64 temp1, temp2, temp3, temp4, sum1, sum2;
        mod_t out2[2];

        a[0] = x;
        a[1] = (x >> 32);
        b[0] = y;
        b[1] = (y >> 32);

        // schoolboy, no knuth trick
        temp1 = ((uint64) a[0]) * b[0];

        temp2 = ((uint64) a[1]) * b[0];

        temp3 = ((uint64) a[0]) * b[1];

        temp4 = ((uint64) a[1]) * b[1];

        sum1 = (temp1 >> 32) + (temp2 & M32) + (temp3 & M32);
        sum2 = (sum1 >> 32) + (temp2 >> 32) + (temp3 >> 32);

        out2[0] = (temp1 & M32) + (sum1 << 32);
        out2[1] = sum2 + temp4;

        // reduce 128 bit result modulo M61
        temp1= (out2[0] & M61) + (out2[0] >> 61) + ((out2[1] << 3) & M61) + (ou
        temp1= (temp1 & M61) + (temp1 >> 61);
        temp1= (temp1 & M61) + (temp1 >> 61);

        return(temp1);
```

```c
}

/* The gateway routine */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  mod_t *xr, *xi, *yr, *yi, *zr, *zi;
  int ndimx, ndimy, i, n, ndim;
  int *dimsx, *dimsy, *dims;
  mxArray *temp;

  /*  Check for proper number of arguments. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
     get to the else statement if mexErrMsgTxt is executed.
     (mexErrMsgTxt breaks you out of the MEX-file.)
  */
  if (nrhs != 2)
    mexErrMsgTxt("Two inputs required.");
  if (nlhs > 1)
    mexErrMsgTxt("Too many output arguments");

  if (!mxIsUint64(prhs[0]) || !mxIsUint64(prhs[1])) {
    mexErrMsgTxt("Inputs must be of type uint64");
  }

  /* Check to make sure the input arguments have same dimensions */
  ndimx=mxGetNumberOfDimensions(prhs[0]);
  ndimy=mxGetNumberOfDimensions(prhs[1]);
  dimsx=mxGetDimensions(prhs[0]);
  dimsy=mxGetDimensions(prhs[1]);

  /* Special case if either of the inputs is a scalar */
  if( ((ndimy==2) && (dimsy[0]*dimsy[1]==1)) || ((ndimx==2) && (dimsx[0]*dimsx[1]=
  {
          if( (ndimy==2) && (dimsy[0]*dimsy[1]==1) )    /* y a scalar */
          {
                  xr = (mod_t*) mxGetData(prhs[1]);
                  xi = (mod_t*) mxGetImagData(prhs[1]);
                  yr = (mod_t*) mxGetData(prhs[0]);
                  yi = (mod_t*) mxGetImagData(prhs[0]);
                  ndim=ndimx;
                  dims=dimsx;
          }
          else
          {
                  xr = (mod_t*) mxGetData(prhs[0]);
                  xi = (mod_t*) mxGetImagData(prhs[0]);
                  yr = (mod_t*) mxGetData(prhs[1]);
```

```
                yi = (mod_t*) mxGetImagData(prhs[1]);
                ndim=ndimy;
                dims=dimsy;
        }

        if( (xi==NULL) && (yi==NULL) ) {
                plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mx
                zr= (mod_t*) mxGetData(plhs[0]);
                n = mxGetNumberOfElements(plhs[0]);
                for(i=0;i<n;i++)
                        zr[i]=rmultmod(xr[0],yr[i]);
        }
        else if( (xi==NULL) ) {
                plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mx
                zr= (mod_t*) mxGetData(plhs[0]);
                zi= (mod_t*) mxGetImagData(plhs[0]);
                n = mxGetNumberOfElements(plhs[0]);
                for(i=0;i<n;i++) {
                        zr[i]=rmultmod(xr[0],yr[i]);
                        zi[i]=rmultmod(xr[0],yi[i]);
                }
        }
        else if( (yi==NULL) ) {
                plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mx
                zr= (mod_t*) mxGetData(plhs[0]);
                zi= (mod_t*) mxGetImagData(plhs[0]);
                n = mxGetNumberOfElements(plhs[0]);
                for(i=0;i<n;i++)      {
                        zr[i]=rmultmod(xr[0],yr[i]);
                        zi[i]=rmultmod(xi[0],yr[i]);
                }
        }
        else {
                plhs[0] = mxCreateNumericArray(ndim, dims, mxUINT64_CLASS, mx
                zr= (mod_t*) mxGetData(plhs[0]);
                n = mxGetNumberOfElements(plhs[0]);
                zi= (mod_t*) mxGetImagData(plhs[0]);
                for(i=0;i<n;i++) {
                        zr[i] = submod( rmultmod(xr[0],yr[i]),rmultmod(xi[0],
                        zi[i] = addmod( rmultmod(xr[0],yi[i]),rmultmod(xi[0],
                }
        }
}

else {                      /* pointwise multiplication... vectors must be

if ( ndimx != ndimy )
       mexErrMsgTxt("Number of array dimensions must match");
```

```
/* */
for (i=0; i<ndimx; i++) {
        if (dimsx[i] != dimsy[i])
                mexErrMsgTxt("Matrix dimensions must agree.");
}

xr = (mod_t*) mxGetData(prhs[0]);
xi = (mod_t*) mxGetImagData(prhs[0]);
yr = (mod_t*) mxGetData(prhs[1]);
yi = (mod_t*) mxGetImagData(prhs[1]);

if( (xi==NULL) && (yi==NULL) ) {
        plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxREAL);
        zr= (mod_t*) mxGetData(plhs[0]);
        n = mxGetNumberOfElements(plhs[0]);
        for(i=0;i<n;i++)
                zr[i]=rmultmod(xr[i],yr[i]);
}
else if( (xi==NULL) ) {
        plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);
        zr= (mod_t*) mxGetData(plhs[0]);
        zi= (mod_t*) mxGetImagData(plhs[0]);
        n = mxGetNumberOfElements(plhs[0]);
        for(i=0;i<n;i++) {
                zr[i]=rmultmod(xr[i],yr[i]);
                zi[i]=rmultmod(xr[i],yi[i]);
        }
}
else if( (yi==NULL) ) {
        plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);
        zr= (mod_t*) mxGetData(plhs[0]);
        zi= (mod_t*) mxGetImagData(plhs[0]);
        n = mxGetNumberOfElements(plhs[0]);
        for(i=0;i<n;i++)      {
                zr[i]=rmultmod(xr[i],yr[i]);
                zi[i]=rmultmod(xi[i],yr[i]);
        }
}
else {
        plhs[0] = mxCreateNumericArray(ndimx, dimsx, mxUINT64_CLASS, mxCOMPLEX);
        zr= (mod_t*) mxGetData(plhs[0]);
        n = mxGetNumberOfElements(plhs[0]);
        zi= (mod_t*) mxGetImagData(plhs[0]);
        for(i=0;i<n;i++) {
                zr[i] = submod( rmultmod(xr[i],yr[i]),rmultmod(xi[i],yi[i]) );
                zi[i] = addmod( rmultmod(xr[i],yi[i]),rmultmod(xi[i],yr[i]) );
        }
}
```

```
    }
}
```

D.0.4.11 uminus.c

```
/*
 * ============================================================
 Overloading the unary minus operator for uint64 elements of GF(q^2)

 (61 bit ones' compliment)

 Robin Ince
 * ============================================================
 */


#include "i64.h"
#include "mex.h"

/* The gateway routine */
void
mexFunction( int nlhs, mxArray *plhs[],  int nrhs, const mxArray *prhs[])
{
    int i, n;
    mod_t *pr, *pi;

    /* Check for proper number of input and output arguments */
    if (nrhs != 1) {
        mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument  */
    if (!(mxIsUint64(prhs[0]))){
        mexErrMsgTxt("Input argument must be of type uint64.");
    }

        /* Create output array */
    plhs[0] = mxDuplicateArray(prhs[0]);

    pr = (mod_t *) mxGetData(plhs[0]);
    pi = (mod_t *) mxGetImagData(plhs[0]);
    n = mxGetNumberOfElements(plhs[0]);

    /* Perform task... */
    for(i=0; i < n; i++) {
        pr[i] = ((~ pr[i]) & M61);
    }

    /* If there is an imaginary part of the input... */
    if(pi != NULL) {
        for(i=0; i < n; i++) {
            pi[i] = ((~ pi[i]) & M61);
            }
        }

}
```

D.0.4.12 rdivide.m

```
function z=rdivide(x,y)
% Division in GF(M61^2) (multiplication by inverse)
% Real only
% Only needed to renormalise inverse transform & calculate primitve roots
q=getq;

if      ( isreal(x) & isreal(y) )       % Both Real
        z = x .* (y.^(q-uint64(2)));
elseif  ( isreal(y) )                   % X Complex, Y Real
        z = x .* (y.^(q-uint64(2)));
elseif  ( isreal(x) )                   % X Real, Y Complex
        z = complex( x.*real(y),uminus(x.*imag(y)) );
        z = z .* ((real(y).*real(y) + imag(y).*imag(y)) ).^(q-uint64(2));
else                                    % Both Complex
        z = complex( (real(x).*real(y) + imag(x).*imag(y)), (imag(x).*real(y) - re
        z = times(z, (real(y).*real(y) + imag(y).*imag(y)).^(q-uint64(2)));
end
```