Kai Molander

DIGITAL SIGNAL PROCESSING THEORY FOR IN-PHASE AND QUADRATURE SIGNAL COMPONENTS RECORDED FROM WCDMA FREQUENCIES

Fourier-transform, filtering, and DSP theory.

Thesis
CENTRIA UNIVERSITY OF APPLIED
SCIENCES
Degree Programme in Telecommunications technology
March 2013

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Yksikkö Ylivieska	**Aika** Marraskuu, 2013	**Tekijä/tekijät** Kai Molander
Koulutusohjelma Tietoliikennetekniikka		
Työn nimi Digitaalisen signaalinkäsittelyn teoriaa I/Q signaali komponenteilla, jotka on nauhoitettua WCDMA – taajuuksilta. Fourier-muunnos, suodatus ja WCDMA signaalien käsittelyyn liittyvä teoria.		
Työn ohjaaja Joni Jämsä	**Sivumäärä** 50	
Työelämäohjaaja Ville Kukonlehto		

Dokumentin tarkoituksena on esitellä meidän ensimmäiset digitaaliseen signaalin näytteiden processointiin liittyvät askeleet. Tässä tapauksessa, nämä signaali näytteet on nauhoitettu WCDMA taajuuksilta ja ne tulee muuttaa taajuus-tasoon, sekä suodattaa, jotta (takaisin aikatasoon muuttamisen jälkeen) näytteiden processointia voidaan jatkaa WCDMA – spesifikaatioiden algoritmeillä.

Tavoite tässä teesissä on selkeästi kuvata diskreettiin Fourier-muunnokseen liittyvät teoriat, sekä miten näitä käytetään käytetään signaalinkäsittelyssä.

Asiasanat
Fourier muunnos, digitaalinen signaalinkäsittely, WCDMA (UMTS -verkoissa käytettävä radiorajapinta)

ABSTRACT

CENTRIA UNIVERSITY OF APPLIED SCIENCES	Date March, 2013	Author Kai Molander
Degree programme Telecommunications technology		
Name of thesis DIGITAL SIGNAL PROCESSING THEORY FOR IN-PHASE AND QUADRATURE SIGNAL COMPONENTS RECORDED FROM WCDMA FREQUENCIES Fourier-transform filtering, and DSP theory		
Instructor Joni Jämsä	**Pages** 50	
Supervisor Ville Kukonlehto		

The objective of this thesis is to describe our first steps of signal sample processing. In our case these samples are recorded from WCDMA frequencies and they need to be transformed into the frequency domain as well as filtered so that the signal samples (once transformed back to the time domain) can be processed by WCDMA - specification algorithms.

The objective of this thesis is to clearly explain the basics of the discrete Fourier-transform and how it is applied to signal processing.

Key words

Fourier-transform, digital signal processing, Wideband Code Division Multiple Access, WCDMA

DEFINITIONS AND ABBREVIATIONS

Function	In mathematics, a function (f) associates an argument or input (x) with one value or output: $f(x)$.
Transform	In mathematics an operation where the input is a function (f) and the output is another function: Transform of f or as it is usually expressed in its shortened form as just a capital F.
Discrete	In mathematics and signal processing, we study a signal or set of values in a certain range, (a finite or limited set of values) rather than a continuous or infinite range.
Discrete transform	In signal processing, mathematical transforms of signals between discrete domains / ranges, such as discrete time or discrete frequency.
Fourier transform	A mathematical operation that allows us to transform a mathematical function of time to its frequency representation.
Fourier series	In mathematics or signal processing allows us to describe a periodic function or signal as a set of sine and cosine functions. In signal processing specifically, the Fourier transform allows us to see the effect of each individual frequency (dependent on resolution). The study of Fourier series is a branch of Fourier analysis.
Fourier analysis	In modern mathematics, refers to both the operation of decomposing a function into simpler pieces as well as rebuilding it from those pieces (pieces referring to values).

DFT	Discrete Fourier Transform, a specific kind of discrete transform used in Fourier analysis.
FFT	Fast Fourier Transform, describes an efficient algorithm used to compute the discrete Fourier transform and its inverse.
DIT	When discussing the FFT, decimation refers to how we break down the FFT. In the case of separating the calculations to the "even" and "odd" indices (where these refer to the index for an arbitrary array that holds samples used in calculation of the FFT) we are discussing a *decimation-in-time* algorithm.
DIF	When discussing the FFT, if we break down the FFT by separating in a first-half/second-half (again, indices) approach it is referred to as the *decimation in frequency* algorithm.
WCDMA	Wideband Code Division Multiple Access, air interface (or radio) technology of UMTS.
UMTS	Universal Mobile Telecommunications System. UMTS is an umbrella term for the third generation radio technologies developed within the 3GPP.
3GPP	The 3rd Generation Partnership Project (3GPP) unites [Six] telecommunications standards bodies, known as “Organizational Partners” and provides their members with a stable environment to produce the highly successful Reports and Specifications that define 3GPP technologies.
DSP	Digital signal processing, is the processing of discrete signals which are represented in a digital format. While working with

	digitized data we gain several advantages in the processing stages.
SAMPLING	Sampling (in signal processing) refers to the reduction of a continuous signal to series of discrete samples.
SAMPLE	Refers to a value or set of values at a point in time and/or space.
SAMPLER	A system (ADC) or operation (mathematics) that extracts samples from a continuous signal.
ADC	Analog-to-Digital converter, also abbreviated as A/D , is a device that in DSP is used to convert continuous signals into a discrete digital representation (signal sampling) in the time domain.
I/Q signal	In-phase and Quadrature signal components can be represented as complex number format in mathematics as cosine (real) and sine components (imaginary). These discrete values are referred to as symbols. Symbols can be used visualized as points on the complex plane and mapped via the constellation diagram.
Complex number	Complex number is a number which can be represented in the form , where *r* is the real part and *i* is the imaginary part of the number. Also referred to as the imaginary unit . Complex numbers are used in signal processing to represent the *phase* information of a signal since they extend the idea of one dimensional numbers to two dimensional.
Imaginary unit (*i)*	Defined in mathematics as $i = \sqrt{-1}$.

Pi (π)	The ratio of a circle's circumference to its diameter. Approximation defined as 3.1415926535.
Euler's number (*e*)	Defined in mathematics as an approximation of 2.71828. Used to define the exponential function.
Prime number	Defined in mathematics as a natural number greater than 1 that has no positive divisors other than 1 and itself. For example 3.
Composite number	Defined in mathematics as a positive integer which has a positive divisor other than one or itself.
Natural number	Defined in mathematics as a set of positive integers 1, 2, 3, etc. or a set of nonnegative integers 0, 1, 2, 3 etc.
Radix	In the FFT and DFT divide-and-conquer algorithms, the radix refers to the decomposition of the FFT size N when it is separated into a calculation radices r , where N is a composite and r is a prime, $N = r_1 * r_2 * r_3 \dots r_n$.

Mathematical symbols

*	Multiplication sign.
/ , ÷	Division sign(s).
$\frac{x}{y}$ x / y	Division of x with y.
$\sum$	Summation sign.
$\int$	Integral sign.
$\sum_{lower\ limit}^{upper\ limit} x(n) * y(n)$	Operation of summation and multiplication with limits.

$X(k) = \sum_{n=0}^{upper\ limit} x(n) * y(k-n)$	Discrete convolution.
$x \in X$	Element-of sign (x is included in or is an element of X)

CONTENTS

1 INTRODUCTION

This document discusses the projects first signal processing steps, the transformation of time domain signal samples to the frequency domain via the Fast Fourier Transform. It does not go into detail about the shortcomings of the FFT in some signal processing applications, nor are there any proper comparisons made against direct convolution filters.

Firstly the theory related to our defined problem is overviewed and necessary definitions are established. In the following chapters we move on to look at the process of calculating a Discrete Fourier Transform and finally implement one of the simplest forms of the FFT algorithm, the Cooley-Tukey Fast Fourier-transform algorithm. A portion of the algorithm as programming code is presented in the annexes.

The evaluation chapter presents the next problems that need to be solved in the case of WCDMA signal processing.

2 THEORETICAL BASIS

A system records signal samples from the air and these signal samples need to be post-processed in a non – real-time software / hardware combination. This recording system design may vary immensely, in our case it has to take into account the requirements of the third generation (3G) air interface technology in WCDMA.

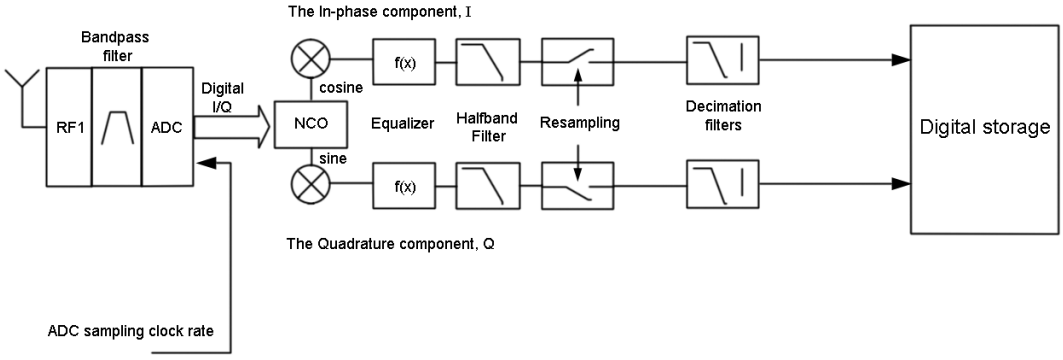


Figure 1: A pseudo block diagram for a signal sampling and recording device (paraphrased from R&S FSQ-B17 Digital Baseband Interface manual)

The above graph shows a possible receiver block diagram based heavily on the one described in the R&S FSQ-B17 Digital Baseband Interface manual. The bandpass filter may be used to limit the bandwidth currently recorded. The signal is then digitized by the sampler, an ADC, at the chosen sampling rate. These digital samples are carried to the two mixers that are fed by the numerical controlled oscillator. Here the I/Q signal is split to its In-Phase and Quadrature components. The I/Q split is the most important portion of the processing since this is where we will start our digital signal processing via software. Processing such as the FFT requires the signal the be split since it employs complex number operations to achieve signal processing.

2.1 In-Phase and Quadrature signals

The I/Q signal is represented as complex numbers and they carry along with them phase information. There are many different modulation techniques that map this phase information to bits (via demodulating the signal). The following graph shows a two dimensional representation of the I/Q signal.

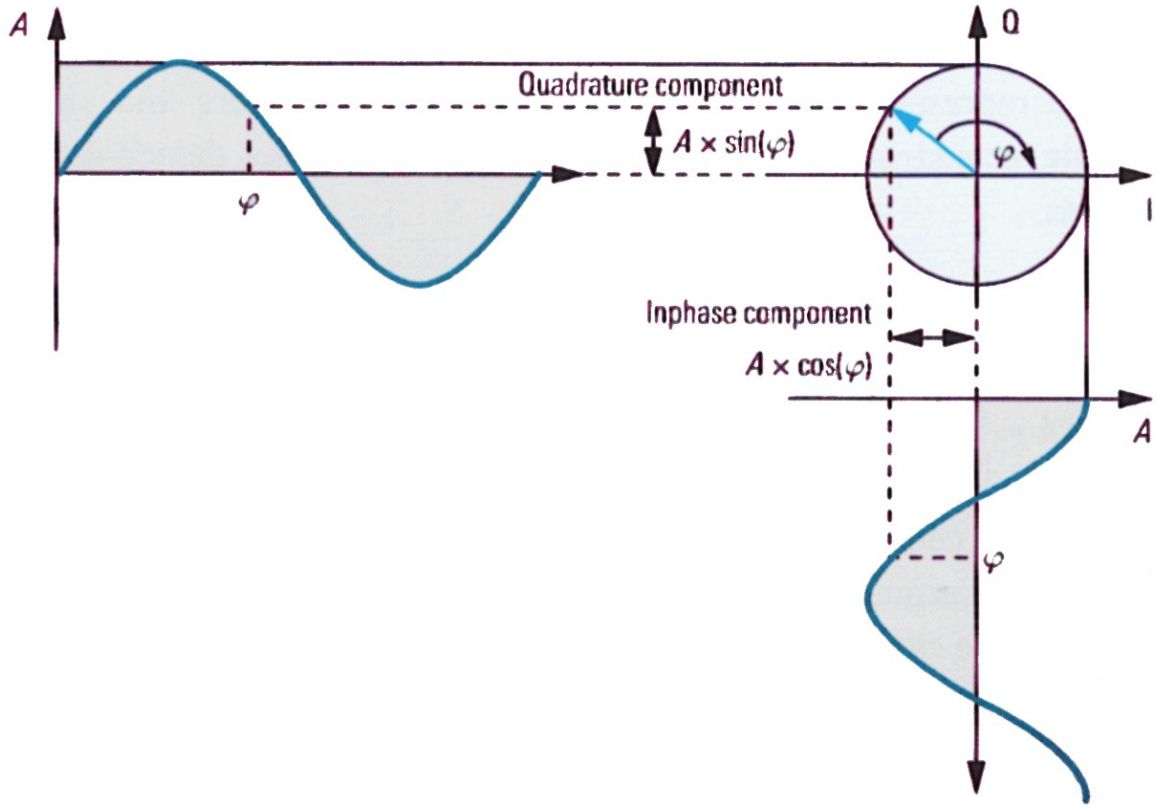


Figure 2: In-Phase and Quadrature signal 2D graphical representation (Hiebel 2007.)

The important thing to note for us considering the FFT is that we need to keep this phase information and deal with the complex number representation of these signals in order to gain a useful end product out of our calculations. If we only use the real data (the In-phase components) we will not be able to continue processing the signal in terms of WCDMA receiver algorithms.

2.2 WCDMA and the sampling theorem

The Nyquist–Shannon sampling theorem comes into play when the ADC takes discrete time domain samples from the air interface. In the graph below, a bandwidth of 40 MHz is represented, where (roughly) at the center a single 5 MHz WCDMA frequency band exists. Note that the x-axis displays the number of bins from 0 to 1024.

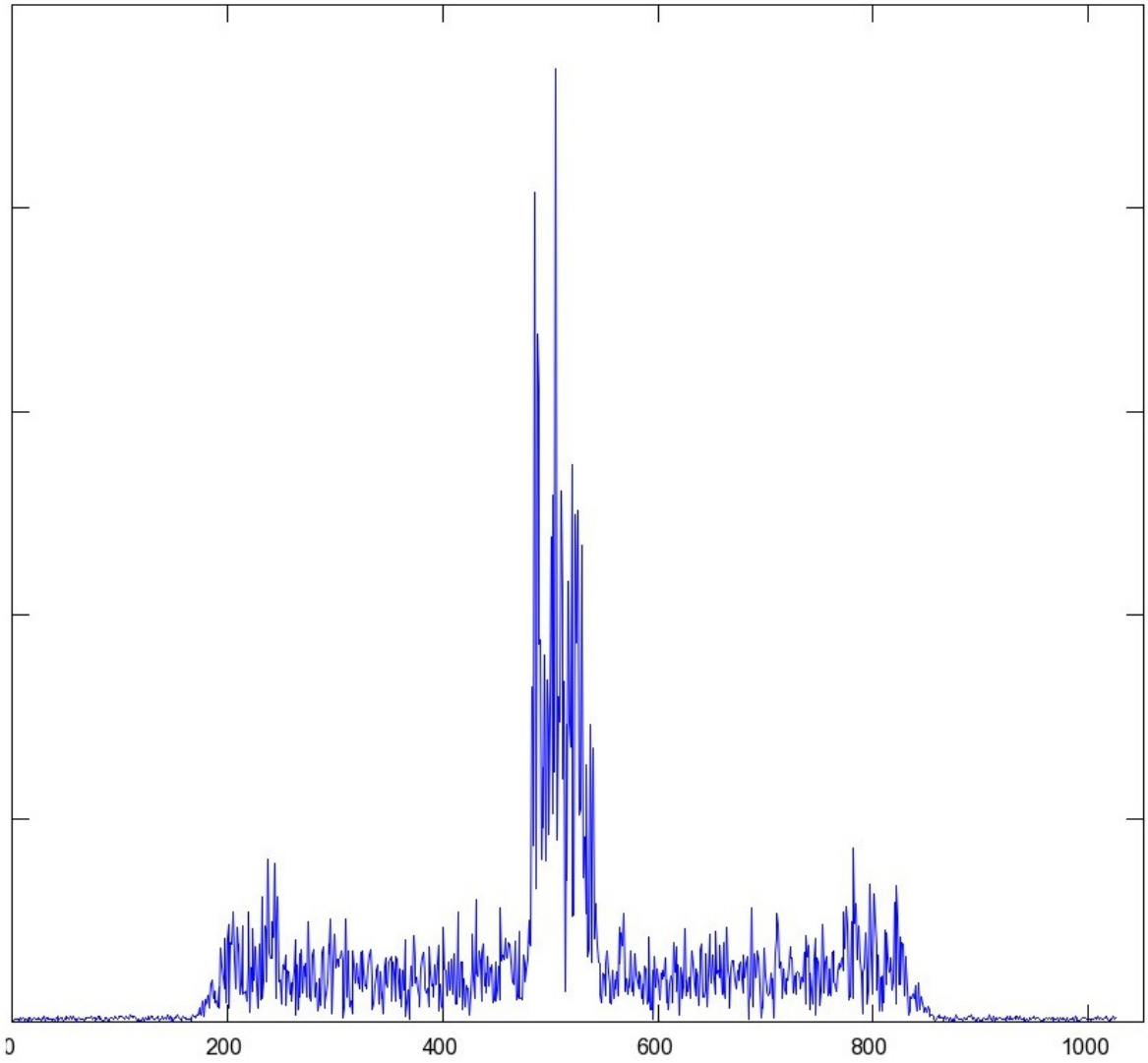


Figure 3: 40 MHz frequency band represented as 1024 point FFT in the frequency domain.

In the very essence of this sampling theorem states that; The sampling frequency is determined as range of signal frequencies. If these are kept below half of the sampling frequency, all of the noise frequencies remain above this limit and can be kept away from the receiver by a low-pass filter. The transmission of a signal may thus be completely distortionless, if the sampling frequency is twice the highest signal frequency. (Lüke 1999, 108).

The previous statement simplified in a mathematical form, where *Fmax* is the highest frequency of the original signal that can be sampled, where :

$$F_{max} = \frac{Sampling\ rate}{2}$$

In practice, the sampling theorem shows that a band-limited analog signal that has been sampled can be reconstructed from sequence of samples if the sampling rate is double the maximum frequency (*Fmax)*. This also means that from a bandwidth point of view, it is the maximum bandwidth, that we can sample with the chosen sampling rate, where no information should be lost.

So looking at figure 3, it means that each point (or frequency bin) in the FFT represents a frequency resolution of 39.0625 kilohertz. This can be calculated by dividing the bandwidth with the number of FFT bins or the sample rate with the size of the FFT.

$$Frequency\ resolution = \frac{Sample\ Rate}{FFT\ points} = \frac{Frequency\ range}{Number\ of\ bins}$$

Now, a single WCDMA band takes up 5 MHz of actual bandwidth. The number of FFT bins it should approximately occupy is equal to 128 and can be calculated by dividing the actual bandwidth with the frequency resolution:

$$Bins\ occupied\ by\ a\ single\ WCDMA\ band = \frac{5\ MHz}{39.0625\ kHz}$$

The most important point to realize from this is that if the recording system we use does not satisfy the sampling theorem we will be losing information. The second important point is that if the resolution of the FFT is not accurate enough (FFT size large enough), the frequency components, in case of multiple WCDMA bands, will start to leak into other bands the way you do not want them to. What the reader should realize now, is that a single bin actually consists of signal information from multiple frequencies. Specifically, if the starting frequency of bin 0 in figure 3 is 0 Hz the first frequency bin actually contains information from frequencies 0 Hz to 39.0625 kHz. The Fourier-transform enables us to separate these frequencies in the resolution we desire.

2.3 Quadrature sampling

When looking for information on the sampling theorem I discovered a few key notes about sampling systems. In essence, the sampling system devices today can be designed in a multitude of different ways and there are some interesting points to gain in terms of signal information. The following extracted from the book The Fast Fourier Transform and its Applications by E. Oran Brigham explains the idea behind the figures one and four where the receiver separates the signal into to two functions.

Applications of the FFT are sometimes limited by the sampling rates achieved by analog-to-digital converters. For these cases, it is possible to achieve a lower sampling rate by separating the signal into two waveforms, or channels, and sampling each channel. This concept is based on the principle that a signal can be expressed in terms of two waveforms called *quadrature functions.* Each of the two quadrature functions occupies only one-half the bandwidth of the original signal. Hence, it is possible to sample each quadrature function at one-half of the sample rate required to sample the original signal.
(Brigham 1988, 327)

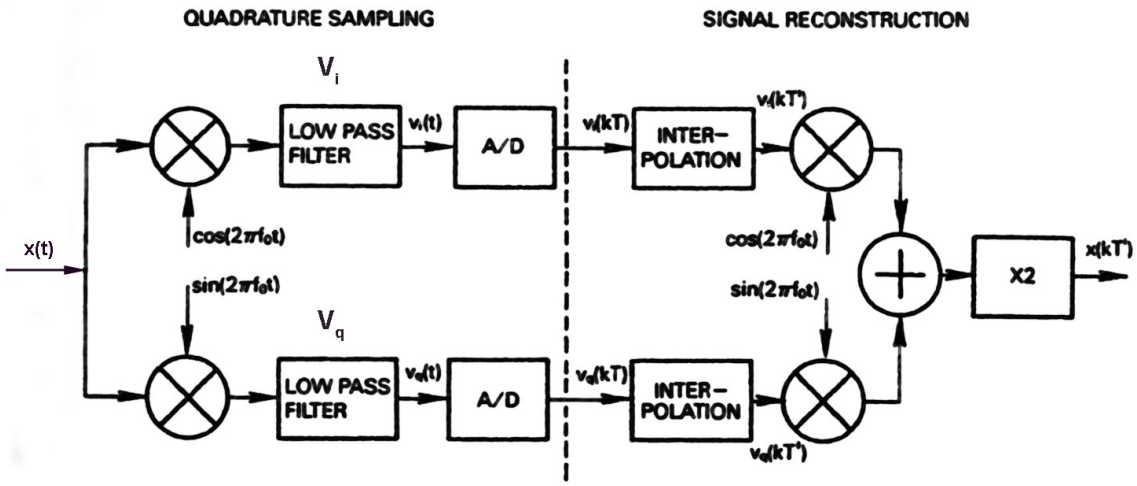


Figure 4: Block diagram of the quadrature-sampling and signal-recombination process. (Brigham 1988, 331)

The following chapter looks at the DFT from the point of view of DSP theory. It presents the examples that need to be understood when discussing the Fourier-transforms and what we actually gain from it.

2.4 What is the point of the DFT and the FFT

The Fourier-transform decomposes the specified signal (in our case, a series of time domain values) and splits the information up to the specified frequency resolution (size of the Fourier-transform) giving us a frequency domain representation of the signal. Some of the mathematical operations that we need to employ on our signal would be very costly and slow (from a computational point of view) in the time domain. However in the frequency domain they can be done more easily and filtering can even be done during the FFT calculation via convolution.

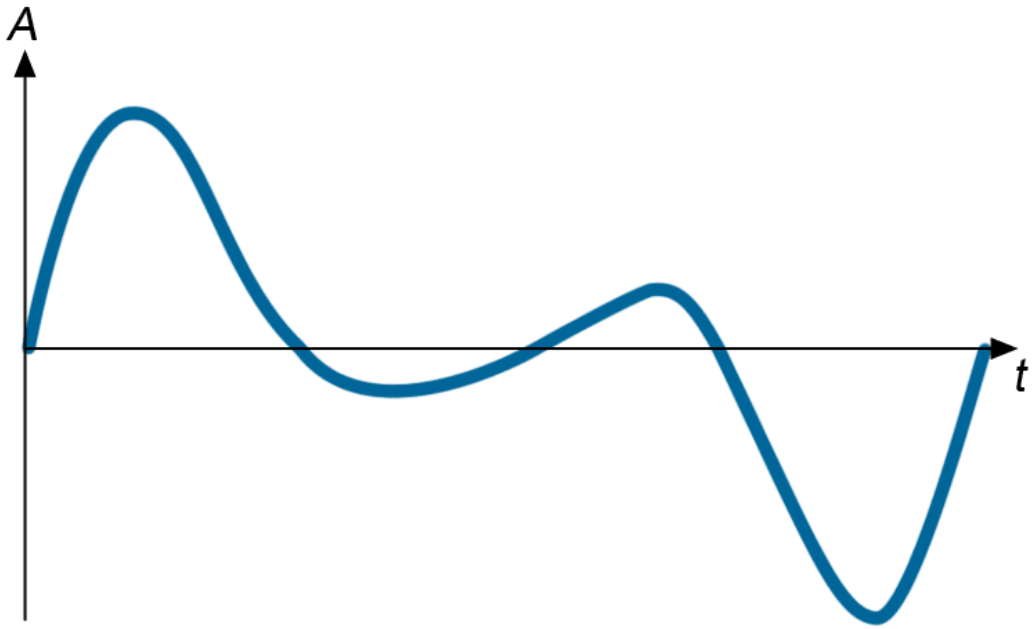


Figure 5: Example of a time-domain signal is represented with all of its frequency components summed.

Here we are presented with an example of signal with all of its different frequency components summed. This is often what we are looking at when we see the signal in a time domain representation. What the Fourier-transform allows us to do is separate these different frequency components from this signal. We are modifying the signal to gain another signal, or in this example multiple different signals.

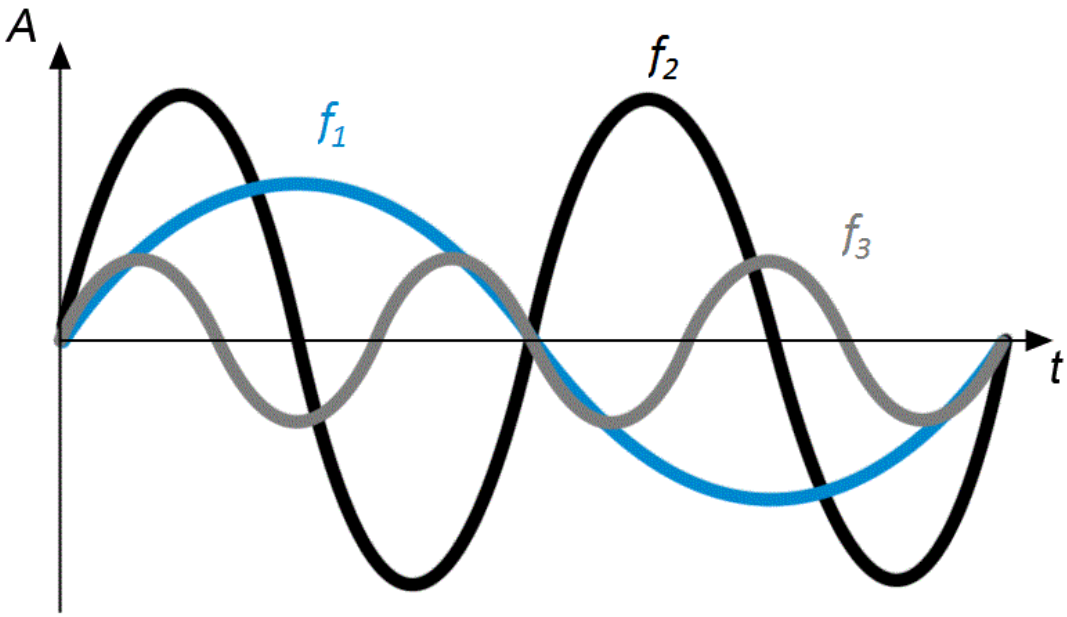


Figure 6: Example signal is represented with all of its frequency components separated in the time-domain.

Here we have the same signal previously from figure 5, but with all the different frequencies (in this case only three for easier display in this 2D domain) displayed separately. When we run our signal or signal samples through the forward Fourier-transform operation, the algorithms and mathematical operations are designed to give us a frequency representation of the signal.

If we try to represent a signal with a 5 MHz bandwidth in the previous way (separating the frequencies into to signals relative to time) not only would the drawing get quite complicated but we would not be able to tell much from it. Next let's look at the frequency domain representation of our example signal.

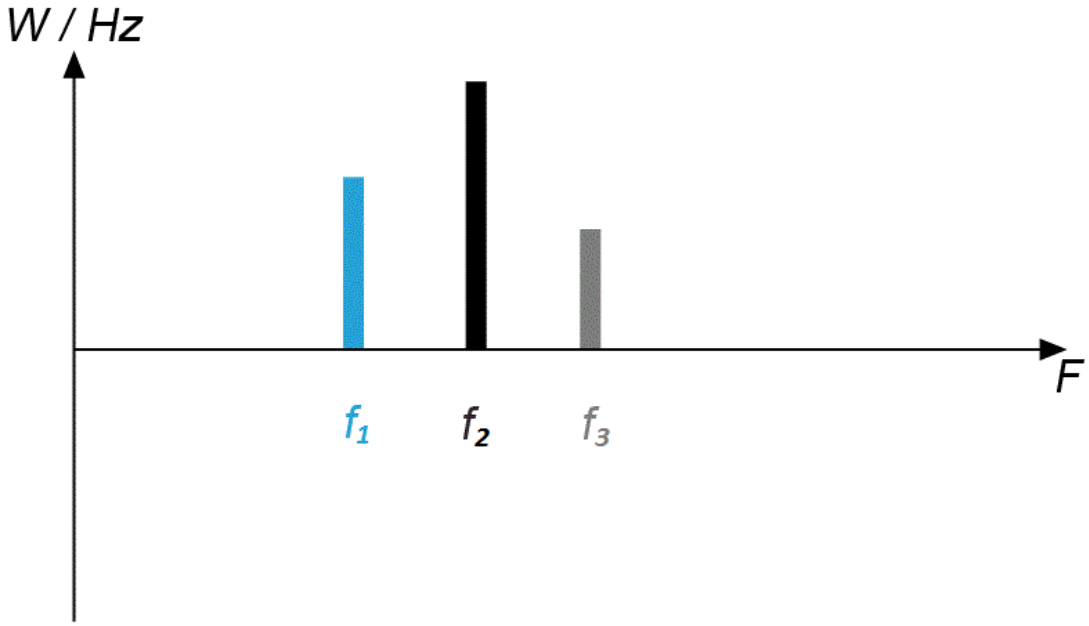


Figure 7: Example signal is represented in the frequency domain.

Here the results of the Fourier-transform are plotted in the frequency domain. From this it is much easier to distinguish, in the case of wide-band signals, the range of frequencies in our signal.

In the very core the Fourier-transform is nothing but multiplications and additions with sine and cosine waves. Moreover when these waves are expressed as a series of discrete values, the mathematics of it suddenly becomes more familiar. As a side note, although the mathematics of it can be very simple, some of the problems that it is used to solve are not.

The only problem with mathematics such as the Fourier-transform is that, although the calculations are just additions and multiplications, the explanations behind the formulas are almost always lacking in detail. This is true especially from the point of view of people that have marginal experience with mathematics. We will now look at some of the mathematics that need to be understood when looking at the Fourier-transform and FFT algorithms.

2.4.1 Euler's formula

Euler introduced the use of exponential functions and logarithms in analytic proofs. Most times in the FFT formulas the sine and cosine components are represented in exponential function notation. It comes down to being another polar format representation of the complex number format.

Euler's number *(e*) helps us define an exponential format for complex numbers, as well as provides a relation to the trigonometric functions. This is the exponential function representation often used with the DFT and FFT formulae.

$$e^{i\phi} = \cos(\phi) + i * \sin(\phi)$$

Where *i* is the imaginary component and φ (phi) is any real number. Euler's formula allows us to move between the polar and Cartesian representation. A complex number can be represented on the complex-plane as Cartesian coordinates.

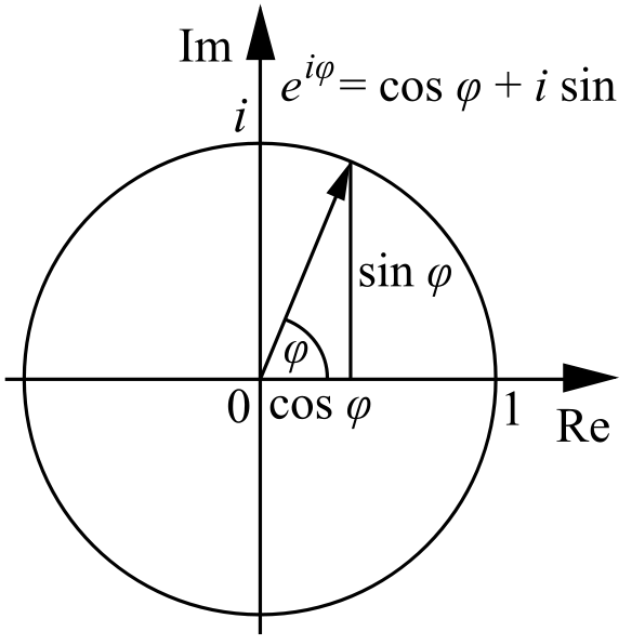


Figure 8: A geometric interpretation of Euler's formula (Wikipedia, 2009)

This is the same complex-plane representation as the I/Q signal example in figure two. It is displayed here once more to demonstrate the link between the mathematics and signal processing.

2.5 Discrete Fourier Transform (DFT)

If you need and explanation on the basics of the equation notation, please see Annex C.

Before discussing the discrete transform lets take a look at the actual Fourier-transform integral it is based on. This example is taken from the Stanford University website:

$$X(\omega) = \int_{-inf}^{inf} x(n) * e^{i*\omega*n} \; dt, \qquad \omega \in (-inf, inf)$$

This is the Fourier transform notation for infinite signals The DFT replaces this previous integral with a finite sum, where N is the size of the DFT:

$$X(\omega_k) = \sum_{n=0}^{N-1} x(n) * e^{-i*\omega_k*n}$$

In summary, the DFT is *simpler mathematically*, and *more relevant computationally* than the Fourier transform. At the same time, the basic concepts are the same. (Smith, J.O. 2007.)

The DFT gives us access to the signals frequency domain, however it is basically implemented as a direct convolution operation. This means that the calculation still requires a lot of arithmetic operations. This is why the algorithm known as the Fast Fourier-transform was developed. The most important point to note at this time, is that the FFT is not an approximation of the DFT. It is the DFT with a reduced number of arithmetic operations. Next, before moving on to the FFT formula and the explanation behind the Cooley-Tukey decimation-in-time FFT algorithm, we will take a look at the divide-and-conquer algorithm approach and how these FFT algorithms relate to it.

2.6 Divide and conquer algorithms for computation of the DFT

This next example is based on the book Digital Signal Processing: Principles, Algorithms and Applications by Proakis and Manolakis, 1996. The books chapter 6.1.2 "Divide-&-Conquer Approach to Computation of the DFT gives a detailed explanation to how the family of FFT algorithms is formed.

The development of computationally efficient algorithms for the DFT is made possible if we adopt a divide-and-conquer approach. This approach is based on the decomposition of an *N*-point DFT into successively smaller DFTs. This basic approach leads to a family of computationally efficient algorithms known collectively as FFT algorithms. (Proakis and Manolakis, 1996).

The divide-and-conquer method solves this problem by breaking it into subproblems that are smaller instances of the same type of problem. It them solves these problems and appropriately combines the results

For the Cooley-Tukey radix-2 FFT algorithm, where N is the size of the DFT, the computational sizes will be factored as a product of two integers: $N = L * M$. the values were chose as $M = N/2$ and $L = 2$. This comes down to breaking the DFT into two smaller DFTs where the summation runs to half of the original size of the DFT. Additionally these two DFTs are designated to compute the even and odd-numbered signal samples.

This "index reversing" is discussed in chapter 2.8 with an example displayed in figure 10. Where the butterfly diagrams (discussed in chapter 2.8.1) use complete bit-reversing of the signal sample indices to produce the frequency-domain results, the Cooley-Tukey algorithm only computes the even- and odd-index split stage.

In the next chapter we will look at the actual FFT formula and the additional rules that provide the algorithm its speed, mainly due to the periodic properties of the coefficients.

2.7 Fast Fourier Transform (FFT)

The Cooley-Tukey decimation-in-time (DIT) algorithm separates the calculations into two parts (hence the name radix-2). The basic formula operations consist of signal samples being multiplied by coefficients and then summed together. A more common notation for the FFT is as follows:

$$FFT_N[k, f] = \sum_{n=0}^{N-1} f(n) * e^{-i*2*\pi*k*n/N}$$

Where we take the FFT of function f where $f(n)$ defines the values of the function being multiplied by our coefficients and then summed together. We assign the calculated values to be stored by index k .

The indices k and n run from 0 to N – 1, where N is the size of the FFT. What this means is that the the calculated frequency domain value at index $k=0$ consists of all values in our function (up to the size of the FFT) multiplied with a portion of our coefficients. The coefficients actually run much longer than the size of the FFT because of both indices existing inside the exponential function.

To transform this notation into the DIT radix-2 algorithm, we need to split the DFT into the even and odd portion (decimation by factor of two) calculations as follows:

$$FFT_N[k, f] =$$

$$\sum_{n=0}^{N/2-1} f(\mathbf{2*n}) * e^{-i*2*\pi*k*\mathbf{2*n}/N} +$$

$$\sum_{n=0}^{N/2-1} f(\mathbf{2*n+1}) * e^{-i*2*\pi*k*(\mathbf{2*n+1})/N}$$

This means that the even (2*n = 0, 2, 4 etc.) indexed samples and the odd (2*n+1 = 1, 3, 5 etc.) indexed samples are handled separately. This is where the name 'decimation-in-time' comes form. Index n now runs from 0 to $N/2$ due to being split into two portions.

Now to produce the computational savings that we are after, the FFT needs to be modified further by extracting the twiddle factor element from our coefficients. In the following equations well only look at the exponential functions, our coefficients of the odd part:

$$\ldots \; e^{-i*2*\pi*k*(\mathbf{2*n+1})/N}$$

From this, we want to extract the twiddle factor or phase factor by separating the summation inside the exponential function to produce two exponential functions:

$$\ldots \; e^{-i*2*\pi*k*\mathbf{2*n}/N} * e^{-i*2*\pi*k/N}$$

This leaves the odd part coefficient with the same expression as the even part coefficients. And now due to the operations inside the transform only consisting of multiplications, we can move this twiddle factor to the front of our summation clause to produce:

$$FFT_N[k,f] =$$

$$\sum_{n=0}^{N/2-1} f(\mathbf{2*n}) * e^{-i*2*\pi*k*2*n/N} +$$

$$e^{-i*2*\pi*k/N} * \sum_{n=0}^{N/2-1} f(\mathbf{2*n+1}) * e^{-i*2*\pi*k*2*n/N}$$

Now as stated, the even and odd parts are multiplied with the same coefficients. The other coefficients that are moved to the front of the odd summation part of the algorithm are also referred to as 'twiddle factors' or 'phase factors'.

Lets define a more compact representation for the summation and multiplication parts as follows, where f_E denotes the DFT for the even and f_O for the odd index part:

$$FFT_{N/2}[k,f_E] = \sum_{n=0}^{N/2-1} f(\mathbf{2*n}) * e^{-i*2*\pi*k*2*n/N}$$

$$FFT_{N/2}[k,f_O] = \sum_{n=0}^{N/2-1} f(\mathbf{2*n+1}) * e^{-i*2*\pi*k*2*n/N}$$

This change of expressing the DFT parts hopefully makes the next rule we introduce a little more clear. These are the periodic properties of the radix-2 FFT. Note that the index n is now hidden inside compact representation of the summation equation that we defined:

$$FFT_N[k, f] = \begin{cases} FFT_{N/2}[k, f_E] + e^{-i*2*\pi*k/N} * FFT_{N/2}[k, f_O] \ , \\ \quad when\ k < N/2 \\ FFT_{N/2}[\boldsymbol{k-\frac{N}{2}}, f_E] - \mathbf{e}^{\boldsymbol{-i*2*\pi*k/N}} * FFT_{N/2}[\boldsymbol{k-\frac{N}{2}}, f_O] \ , \\ \quad when\ k \geq N/2 \end{cases}$$

The most important thing to note here is the index k and the sign flip on the twiddle factor during the calculations. The algorithm recycles the calculation results of the even and odd DFT parts to compute the final output of the FFT.

The Cooley-Tukey algorithm is limited FFT sizes N of power of two or 2^p . This main limitation for the algorithm comes from the radix-2 size. For the radix-2 decomposition the FFT size must be able to be divided to sizes of two prime numbers r , where we split the single DFT into these two DFT's, according to the divide and conquer principle, for which the sizes of the DFT's are primes as long as the rule for power of two size for the FFT is followed.

In the following chapter we will look at the periodic properties of the twiddle factors in much more detail to hopefully gain an understanding how the final summation of the FFT algorithm works.

2.7.1 Twiddle factors and their redundancy properties

In the mathematical notation the coefficients and the twiddle factors (also known as phase factors) are presented in the exponential function notation.

The Cooley-Tukey DIT algorithm splits up the coefficients into the standard multipliers and the twiddle factors but their basic formats are the same. The following equation shows only the twiddle factors expressed in exponential function format that will be converted with Euler's formula:

$$X_k = \cdots \mathrm{e}^{\frac{-2*\pi*i}{N}*k} \; * \; FFT_{N/2}[\boldsymbol{k}-\frac{\boldsymbol{N}}{\boldsymbol{2}}, f_O]$$

Where X_k is the final calculated sample at the index k, which is an integer ranging form 0 to N – 1, N is the size of the FFT (for this example lets say it is 1024), and $FFT_{N/2}[\boldsymbol{k}-\frac{\boldsymbol{N}}{\boldsymbol{2}}, f_O]$ denotes the DFT of the odd-indexed values. The division to even and odd comes from separating the DFT into the radix-2 representation.

As stated, exponential format can be modified into a geometric representation via Euler's formula. In this case it results in the following:

$$\mathrm{e}^{\frac{-2*\pi*i}{N}*k} = \cos(\frac{-2*\pi}{N}*k) + i*\sin(\frac{-2*\pi}{N}*k)$$

The first point here is to provide a simpler representation (other than the exponential format) which is often used when expressing the FFT algorithm formulae. The second important point is that when the FFT is calculated, the results of it can then be manipulated with standard vector mathematics to extract information from them.

Now, this all comes back to the phase difference of the sine and cosine waves to separate the real and imaginary parts of the signal.

The coefficients are often written in the form W_N^k , where it denotes the twiddle factors for FFT size N and index k .

$$W_N^k = \mathrm{e}^{-i*2*\pi*k\,/\,N}$$

Since the twiddle factors rotate on the complex plane it actually creates a form of redundancy which makes the FFT possible. Lets look at the DFT formula again. The important thing to note here is how integers *k* effects the twiddle factor definition. We will need this information shortly.

$$X_k = \cdots W_N^k \ * \ FFT_{Odd}$$

Some authors like to explain this arithmetic reduction by the redundancies inherent in the twiddle factors. They illustrate this with the *starburst* pattern in figure 10 showing the equivalencies of some of the twiddle factors in an 8-point DFT. (Lyons, 2004).

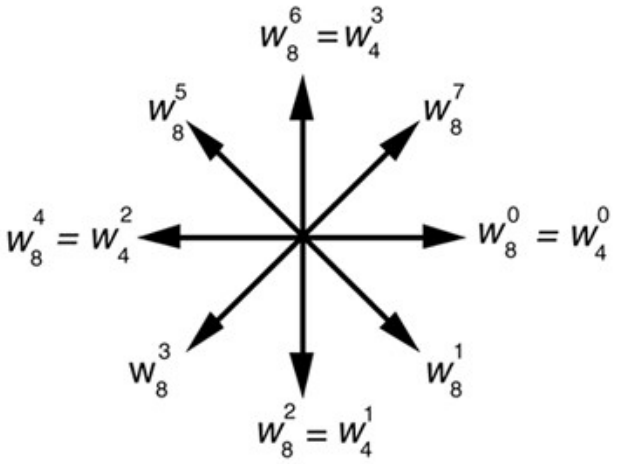


Figure 9: Cyclic redundancies in the twiddle factors (coefficients) of an 8-point FFT (Lyons, 2004).

The above figure as explained by Lyons describes the redundancy and symmetry of the twiddle factors . This redundancy is part of what makes the FFT possible.

Notice, in figure 8, how some of the coefficients W_N are actually the same. For example, how $W_8^6 = W_4^3$. What the notations mean is that for FFT size 8 and index six and FFT size 4 with index three are actually the same (remembering that *N* is the size of the DFT or FFT). Next lets look at how the FFT operates slight more closely.

2.8 The FFT operations

The Fast Fourier transform works by decomposing a signal of N point time domain samples. The key differences between the algorithms are called decimation-in-time and decimation-in-frequency. The Cooley-Tukey version, discussed in detail later, is a DIT algorithm that decomposes the signal samples by separating them into two between the even and odd signal samples.

This example and figure taken from Steven W. Smiths DSP book illustrates the point very well.

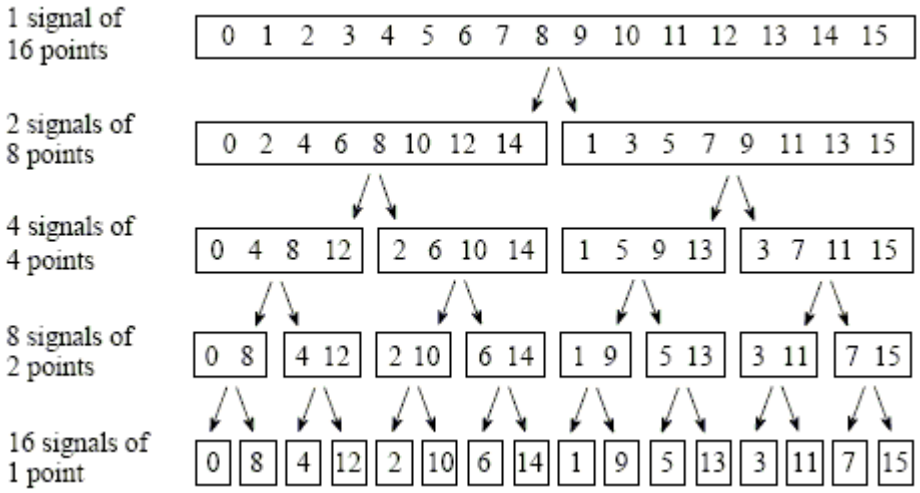


Figure 10: The FFT decomposition. An N point signal is decomposed into N signals, each containing a single point. Each stage uses and interlace decomposition, separating the even and odd numbered samples (DIT) (Smith, 2002).

Figure shows an example of the time domain decomposition used in the FFT. In this example, a 16 point signal is decomposed through four. separate stages. The first stage break the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signal of 4 points. This pattern continues until there are N signals composed of a single point. (Smith, 2002).

The butterfly diagram operations (the actual calculation operations of the FFT) are what is missing from between the lines of Figure 9. The next chapter looks at the twiddle factors as well as the butterfly diagrams and why they are so important to the FFT.

2.8.1 FFT butterfly diagrams

Now lets take a look at the simplest form of FFT butterfly diagram, for more comprehensive images please see Annexes A and B. The next example is taken from Richard G. Lyon's book "Understanding Digital Signal Processing". Where N is again, the size of the DFT or FFT.

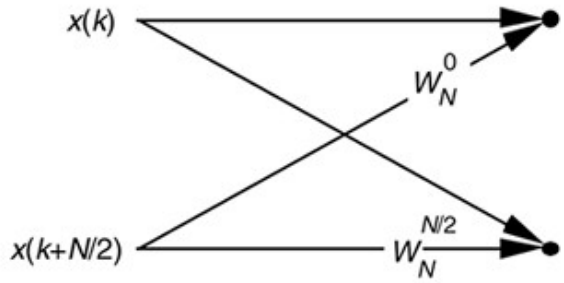


Figure 11: A single 2-point DFT butterfly (Lyons, 2004).

Here, $x(k)$ represents the signal samples at index k and W_N represents the coefficients that are defined as $W_N = e^{-i*2*pi/N}$. This is the very basic operation that takes place when the FFT is calculated, coefficients are multiplied with signal samples to produce the outputs from the FFT. In figure 9 at the coefficients W_N , a multiplication operation takes place and at the end dots a summation operation takes place.

The next important bit of information is the redundancy of the coefficients also known as twiddle factors as discussed in chapter 2.7.1. For larger size butterflies, some of the calculations can be skipped due to the twiddle factor redundancy properties.

A larger butterfly is presented in Annex A, where some of these skips are marked. Derivation of these butterfly images can get quite complicated, however the formulae and explanations behind them already exist so most of the mathematical work has been done for us.

The next chapter looks at the inverse of the Fourier transform and what tricks we can use for calculating it.

2.7 The Inverse Fast Fourier Transform

The reason we calculate the FFT from, in this case, our signal is to do operations such as filtering at lower computational cost. The reason we need the inverse is to transform that frequency representation back to continue processing a WCDMA signal with our next processing step we will need to signal samples back in the time domain.

$$X(k) = * \sum_{n=0}^{N-1} x(n) * e^{-i*\phi / N}$$

Forward DFT

$$x(n) = \frac{1}{N} * \sum_{k=0}^{N-1} X(k) * e^{i*\phi / N}$$

Inverse DFT

The inverse DFT is basically defined as swapping the calculated samples $X(k)$ with the signal samples $x(n)$, as well as the sign flip for the twiddle factors $e^{-i*\phi}$ where $\phi = 2 * pi * m * n$ and N is the size of the DFT.

Continuing on, the next example is again taken from the book by Richard G. Lyons. One of the techniques for calculating the inverse Fourier-transform using the forward transform is shown next. The data is separated into real and imaginary parts.

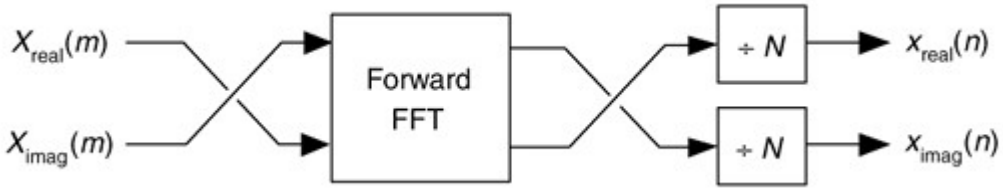


Figure 12: Processing for the inverse FFT calculation method (Lyons, 2004).

In this clever inverse FFT scheme we don't bother with conjugation. Instead, we merely swap the real and imaginary parts of sequences of complex data. (Lyons, 2004).

Now lets start implementing one of the FFT algorithms in the next chapter.

3 PRACTICAL IMPLEMENTATION

The algorithm introduced by J.W. Cooley and J.W. Tukey can be expressed in one of the most simple and understandable forms when it comes to FFT algorithms. This is why it is a good starting point and we will now go over it.

We've already taken a look at portions of this algorithm in the previous chapters, here is to full decimation-in-time DFT formula:

$$X(k) = \sum_{n=0}^{N-1} x_n * \mathrm{e}^{\frac{-2*\pi*i}{N}*n*k}$$

Where N is the size of the DFT, n and k are integers ranging from 0 to $N-1$. The symbols X and x are functions where the integers n and k are the inputs where each input correspond to one output. They can also be viewed as arrays where the integers are used as index indicators to access a value stored in that index. The twiddle factors $\mathrm{e}^{\phi*i}$, though not an immediately representable as an array by itself, can be pre-calculated and stored in memory.

The radix-2 DIT algorithm (Cooley-Tukey, 1965) divides the DFT into two parts: a sum over the even-numbered indices and a sum over the odd-numbered indices. Although the original papers notation is quite complex, it can be expressed much more simply as follows:

$$X(k) = \sum_{m=0}^{N/2-1} x_{(2*m)} * \mathrm{e}^{\frac{-2*\pi*i}{N/2}*(2*m)*k} + \sum_{m=0}^{N/2-1} x_{(2*m+1)} * \mathrm{e}^{\frac{-2*\pi*i}{N/2}*(2*m+1)*k}$$

Note the change to indicator n . Here index indicator n is calculated as $2*m$ for the even index parts and as $2*m+1$ for the odd index part. When $m=0$ the index for even parts is $2*0=0$ and $(2*0)+1=1$ and so on.

Before heading into the actual coding of the algorithm (the snap of source code can be found in Annex D), we need to look at some more of fundamental theory.

3.1 FFT windowing and window functions

Following chapter discusses the window functions in general and introduces the window function to be used for WCDMA data. The next example is taken from Richard G. Lyon's book.

Windowing reduces DFT leakage by minimizing the magnitude of a functions side-lobes. We do this by forcing the amplitude of the input time sequence at both the beginning and the end of the sample interval to go smoothly toward a single common amplitude value. (Lyons, 2004).

In other words, windowing is a technique used to shape the time portion of your measurement data, to minimize edge effects that result in spectral leakage in the FFT spectrum. By using Window Functions correctly, the spectral resolution of your frequency-domain result will increase. (National Instruments, 2012).

To apply a window function to our signal samples, we can modify our FFT formula and include the application of the window function during its calculation. This means, that our input sample sequence $x_{(m)}$ is multiplied by the generated window function coefficients $y_{(m)}$. The modified formula would become:

where $\omega = \frac{-2*\pi*i}{N/2}$, N is the size of the FFT and $k = 0,1,2 \ldots N-1$.

$$X(k) = \sum_{m=0}^{N/2-1} y_{(2*m)} * x_{(2*m)} * e^{\omega*(2*m)*k} \quad \cdots$$

$$+ \sum_{m=0}^{N/2-1} y_{(2*m+1)} * x_{(2*m+1)} * e^{\omega*(2*m+1)*k}$$

Here, our time domain samples would be multiplied by the window function coefficients before the FFT is performed. Next lets look at some of the common window functions and their coefficient generation.

Firstly looking at the simplest form of window, the rectangular window. The rectangular window is defined as: $y(m)=1$ for all values of m i.e when $m = 0,1,2 \ldots N-1$.

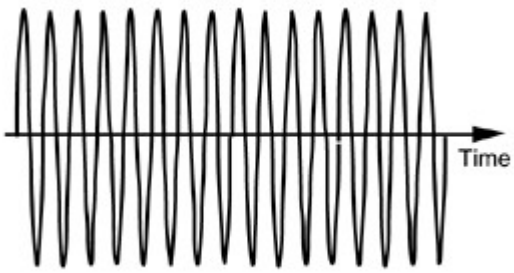


Figure 13: Input signal, time domain (Lyons, 2004)

Figure 13 shows an input sinusoid with constant frequency and amplitude. Next, lets look the windowing functions and what effect they have when applied to this signal.

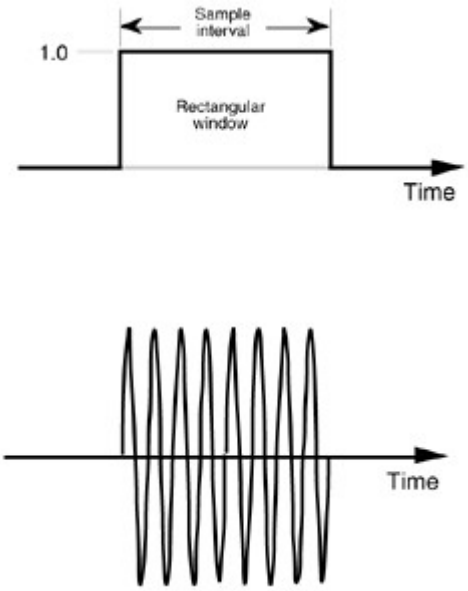


Figure 14: Rectangular window (Lyon, 2004)

In figure 14, the lower portion of the image is the sample output after the windowing function has been applied to our original function, or input signal. Notice how the window function only effects the length of our data, by defining its own sample interval. Only data from this sample interval is used within the window function. Because the rectangular window defines all of its values to one, not much else has changed.

Next, lets look at the Hanning window, also known as the raised cosine window. Where, $m = 0,1,2 \ldots N-1$ and N is the size of the FFT, the coefficients $y(m)$ are:

$$y(m) = 0.5 - 0.5 * \cos\left(\frac{2 * \pi * m}{N-1}\right)$$

When the Hanning window function is applied to our original signal (previous page, figure 14) the results (lower portion of figure 15) are vastly different, since the coefficients are generated using a cosine function.

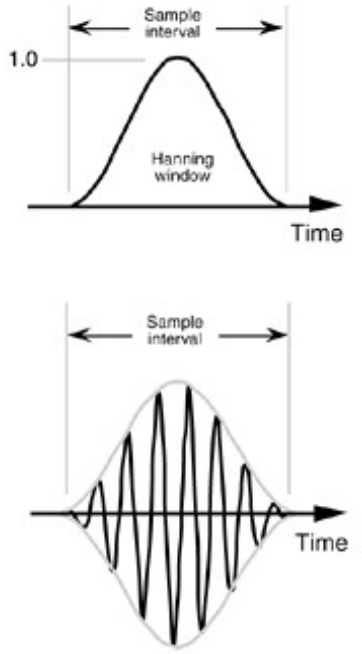


Figure 15: Hanning window
(Lyons, 2004)

The Hanning window is the more interesting example, since it is closer to the shape of the Root-raised cosine filter used in WCDMA. The RRC filter definition can be found in Annex E. The RRC window coefficients are the coefficients of choice when actually working with WCDMA data.

Now lets look at the closely related topic to the window-functions which is the overlap that needs to be taken into account during signal processing.

3.2 Signal sample overlap

Because of windowing (discussed in the previous chapter) our output signal can be greatly reduced around the edges. This of course assumes that we are using a window-function that focuses the signal amplitude around the center of the window-function, and such is the case in WCDMA.

In DSP terms, this means we need to apply what is called the overlap-add method to our calculations. This simply means that the processed overlapping data parts need to be summed together to produce a viable representation of our signal.

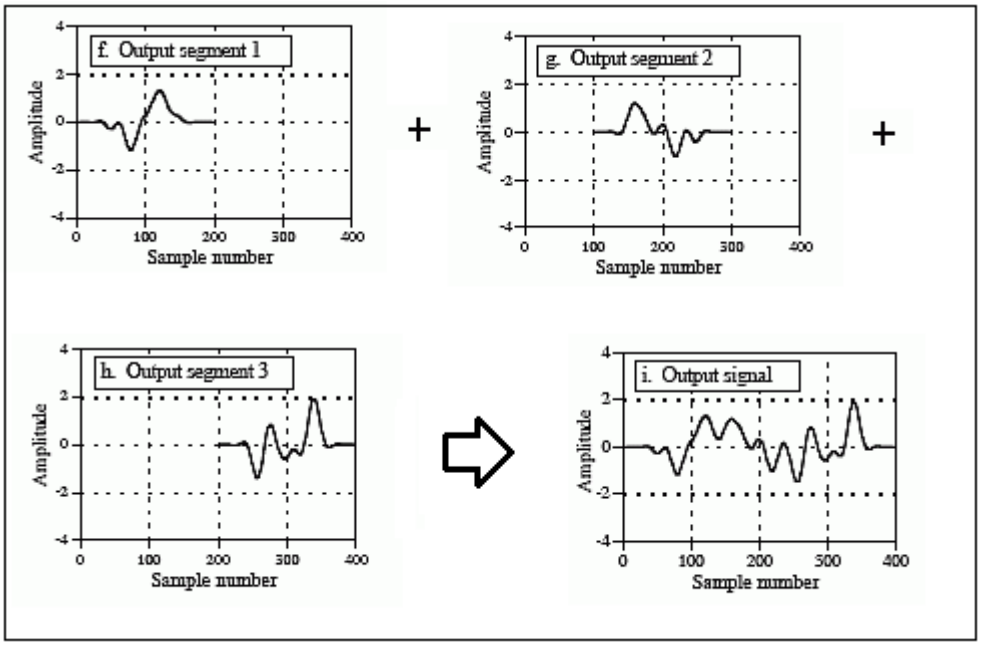


Figure 16: The Overlap-Add method (Smith, 2002)

Take note of the sample numbers in figure 16. As a window-function is applied to our processing segments, these output segments (1 to 3) are produced. The additional step to produce our output signal is to sum the overlapping sample parts. For example, output segments one and two have overlapping sample numbers ranging from 100 to 200. In the output signal graphic, you can see the shape that summing together those two segments produces with the two rises quite clearly visible.

Finally, in the next chapter lets look at an implementation of the FFT in detail.

3.3 Implementation in C-based languages

To view the full source code, please see Annex D. Note that the following piece of complex multiplication code has been implemented by Microsoft Visual Studio compiler *intrinsic*.

An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsic functions are inherently more efficient than called functions because no calling linkage is required.

Intrinsics make the use of processor-specific enhancements easier because they provide a C/C++ language interface to assembly instructions. In doing so, the compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data. (Microsoft, 2011).

Please note that the implementation code relies on the compiler to generate an efficient starting point for modifying the generated assembly code. The memory stack and heap management are left without much of a mention. This efficient complex multiplication is taken from the Intel SSE instructions document by Mostafa Hagog of Intel's Microprocessor Technology Labs.

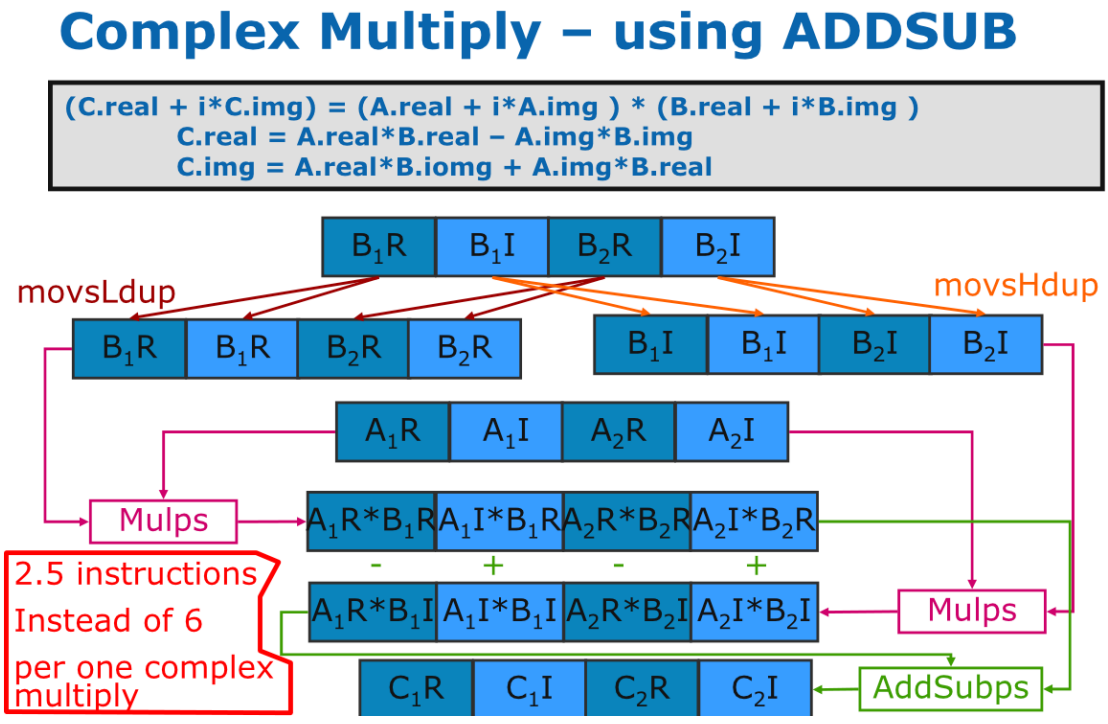


Figure 17: Complex multiplication using SSE intrinsic (Hagog, 2007)

As we are dealing with complex valued signal samples, this will provide us with a more efficient implementation in general. Complex multiplication is defined as:

$$z * w = (a + ib) * (c + id)$$

$$\rightarrow \; a * c + ib * c + a * id - b * d$$

$$\rightarrow (a * c - b * d) + i(a * d + b * c)$$

With the SSE vector, we will end up loading four signal sample values as 32-bit floating point values at a time to our processor registers. Let us define them as:

$$VECTOR_s = [\, a_r \;,\; a_i \;,\; b_r \;,\; b_i \,]$$

Where the subscript denotes either the real part x_r or the imaginary part x_i of the complex value. Let us also define the coefficients, that our signal samples will be multiplied with, in a similar manner as:

$$VECTOR_c = [\, c_r \;,\; c_i \;,\; d_r \;,\; d_i \,]$$

Now the important thing to note here, is that the way the SSE load operation works, is that it takes the next four sequentially stored 32-bit floating point values from the memory location we specify (memory alignment restrictions apply here). So in order to pair the right coefficients with the correct signal samples our complex multiplication output should look as follows:

$$VECTOR_{0,1} = (a_r * c_r - a_i * c_i) + (a_r * c_i + a_i * c_r)$$

$$VECTOR_{2,3} = (b_r * d_r - b_i * d_i) + (b_r * d_i + b_i * d_r)$$

To produce this result with SSE programming our first step uses the SSE intrinsics *movsLdups* and *movsHdups* on our coefficients to produce two new arrays of values:

$$\begin{array}{cc} [\, c_r \;,\; c_i \;,\; d_r \;,\; d_i \,] & [\, c_r \;,\; c_i \;,\; d_r \;,\; d_i \,] \\ \downarrow & \downarrow \\ movsLdup & movsHdup \\ \downarrow & \downarrow \\ [\, c_r \;,\; \boldsymbol{c_r} \;,\; d_r \;,\; \boldsymbol{d_r} \,] & [\, \boldsymbol{c_i} \;,\; c_i \;,\; \boldsymbol{d_i} \;,\; d_i \,] \end{array}$$

The *movsLdup* intrinsic duplicates the first and third elements and places the first element in the positions of the **first and second** elements, and the third element into the **third and fourth** elements of the return value. The *movsHdup* intrinsic in turn duplicates the second and fourth elements and placing them in a similar manner. One of the arrays now consists only of real values and the other one only of imaginary values.

Next we will multiply the new coefficient arrays with out signal sample arrays with the *mulps* intrinsic.

$$\begin{array}{cc} [\ c_r\ ,\ \boldsymbol{c_r}\ ,\ d_r\ ,\ \boldsymbol{d_r}\ ] & [\ \boldsymbol{c_i}\ ,\ c_i\ ,\ \boldsymbol{d_i}\ ,\ d_i\ ] \\ *\quad *\quad *\quad * & *\quad *\quad *\quad * \\ [\ a_r\ ,\ a_i\ ,\ b_r\ ,\ b_i\ ] & [\ a_r\ ,\ a_i\ ,\ b_r\ ,\ b_i\ ] \\ \downarrow & \downarrow \\ [\ c_r{\cdot}a_r\ ,\ c_r{\cdot}a_i\ ,\ d_r{\cdot}b_r\ ,\ d_r{\cdot}b_i\ ] & [\ c_i{\cdot}a_r\ ,\ c_i{\cdot}a_i\ ,\ d_i{\cdot}b_r\ ,\ d_i{\cdot}b_i\ ] \end{array}$$

The split coefficient arrays are multiplied with the signal sample to produce our next intermediate results. Following the complex multiplication defined in Figure 17, our next step is to use the intrinsic *addsubps* for the final summation of the complex multiplication.

$$\begin{array}{rl} Multiplied_1 = [\ c_r{\cdot}a_r & ,\ c_r{\cdot}a_i\ ,\ d_r{\cdot}b_r\ ,\ d_r{\cdot}b_i\ ] \\ \quad\quad\quad\quad\quad\quad - & \quad\quad + \quad\quad - \quad\quad + \\ Multiplied_2 = [\ c_i{\cdot}a_r & ,\ c_i{\cdot}a_i\ ,\ d_i{\cdot}b_r\ ,\ d_i{\cdot}b_i\ ] \end{array}$$

Notice how the intrinsic alternates with the subtraction and addition of corresponding elements in the vectors. However, here we note that the results would not match the required steps for complex multiplication as defined earlier with $VECTOR_{0,1}$ and $VECTOR_{2,3}$.

$$RESULT_{0,1} = (c_r * a_r - \underbrace{c_i * a_r}_{\text{incorrect position}}) + (c_r * a_i + \underbrace{c_i * a_i}_{\text{incorrect position}})$$

We will need to shuffle one of our intermediate result arrays to produce the correct output as follows:

$$CORRECT_{0,1} = (c_r * a_r - \underbrace{c_i * a_i}_{\text{correct}}) + (c_r * a_i + \underbrace{d_i * b_r}_{\text{correct}})$$

These modifications group the real parts of the calculation together while respectively doing the same with the imaginary parts of the calculation. Note that $i * i = -1$.

The exact same result and solution can be noted for the other complex multiplication:

$$RESULT_{2,3} = (d_r * b_r - \underbrace{d_i * b_r}_{incorrect\ position}) + (d_r * b_i + \underbrace{d_i * b_i}_{incorrect\ position})$$

$$CORRECT_{2,3} = (d_r * b_r - \underbrace{d_i * b_i}_{correct}) + (d_r * b_i + \underbrace{d_i * b_r}_{correct})$$

To achieve this result in programming we will use the SSE macro *SHUFPS* to switch the position of our required values in combination with another macro *_MM_SHUFFLE* for the creation of our shuffle mask.

We want to switch the position of the first and second elements and the third and fourth elements in the $Multiplied_2$ vector to produce the correct output, so we define our bit mask as:

$$MASK = \{2, 3, 0, 1\}$$

Note that the mask macros indices are actually the opposite of the vector indices so the index zero in the SSE vector actually corresponds to index three in the mask macro. This is due to the way the mask is calculated. Now, this means that the SSE vectors indices zero and one will switch places to produce the correct output as follows:

$$SHUFPS\{[\ c_i \cdot a_r\ ,\ c_i \cdot a_i\ ,\ d_i \cdot b_r\ ,\ d_i \cdot b_i\ ], MASK\}$$
$$\downarrow$$
$$RESULT_{shuffle} = [\ \underbrace{c_i \cdot a_i\ ,\ c_i \cdot a_r}_{switched}\ ,\ \underbrace{d_i \cdot b_i\ ,\ d_i \cdot b_r}_{switched}\ ]$$

After the shuffle operation we can proceed with the calculation of the complex multiplication as defined in Figure 17:

$$Multiplied_1 = [\ c_r \cdot a_r\ ,\ c_r \cdot a_i\ ,\ d_r \cdot b_r\ ,\ d_r \cdot b_i\ ]$$
$$-\qquad +\qquad -\qquad +$$
$$Multiplied_{2,shuffle} = [\ \underbrace{c_i \cdot a_i\ ,\ c_i \cdot a_r}_{switched}\ ,\ \underbrace{d_i \cdot b_i\ ,\ d_i \cdot b_r}_{switched}\ ]$$

Which will finally produce the correct output to our SSE vector elements as:

$$FINAL\begin{Bmatrix}0\\1\\2\\3\end{Bmatrix} = \begin{Bmatrix}c_r \cdot a_r - c_i \cdot a_i\\ c_r \cdot a_i + c_i \cdot a_r\\ d_r \cdot b_r - d_i \cdot b_i\\ d_r \cdot b_i + d_i \cdot b_r\end{Bmatrix} = \begin{Bmatrix}[0]\,real\ part\\ [0]\,imaginary\ part\\ [1]\,real\ part\\ [1]\,imaginary\ part\end{Bmatrix}$$

The real and imaginary parts of samples[0, 1] and coefficients[0, 1] are multiplied with each other to produce the correct results of our complex multiplication.

3.4 Results and their interpretation

The FFT results shifted roughly to the center of the spectrum are presented in the figure below.

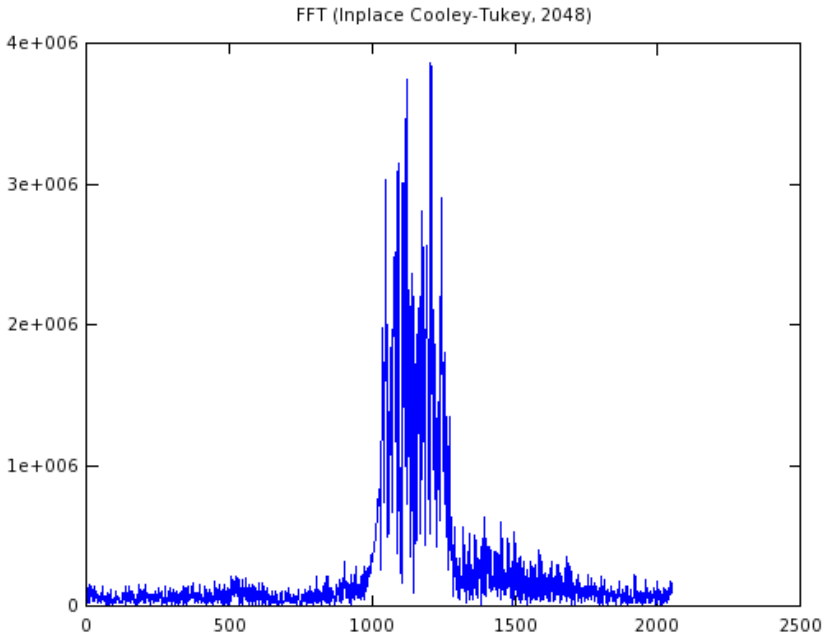


Figure 18: Cooley-Tukey algorithm results, no window-function.

When taking into account our 40 MHz recording bandwidth, our 5 MHz WCDMA band should take roughly 256 FFT bins with size 2048 point FFT. As we can see from the figure it is roughly equivalent to this even without any window-functions applied. A more detailed result verification is done via additional processing, but this is not included in the scope of this thesis.

4 THE SOLUTION AND ITS EVALUATION

An important thing to note of our test data, is that the WCDMA band is almost centered. Due to this, minimal frequency error is present and hence no frequency shifting is needed when just drawing the spectrum. In a real world application however, multiple bands can exist near each other in efficient recording systems. Additional theories and especially implementations to be considered, because of this, would include spectrum frequency shifting being implemented.

The next signal processing might be decimation, or lowering the sampling rate (down-sampling) to the WCDMA defined chip rate of 3.84 MHz. Depending on the recording system however, in some cases it may be necessary to actually raise the sampling rate of our signal, or interpolate (up-sample). This is done since decimation basically requires the the higher sampling rate to be a multiple of our sampling we want to reduce it to. This is because at its simplest, a decimator simply lets through every $n:th$ sample, where n is the integer factor for decimation. More complex decimation schemes however, do exist. One such is described as being part of a polyphase filter, where four filter banks are used to control the signal flow from the ADC (Richardson, 2005).

The solutions covered in this document however are valid theories and provide us with a solid base of knowledge for our implementation of the first part of signal processing when working with our data.

REFERENCES

Brigham, Oran E. 1988. The Fast Fourier Transform and its Applications. New Jersey: Prentice-Hall, Inc.

Cooley, James W. and Tukey, John W. 1965. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation. 19, 297-301. Available at http://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/home.html.

Hiebel, M. 2007. Fundamentals of Vector Network Analysis. Rohde & Schwarz.

Hagog, Mostafa. 2007. Intel: Looking for 4x speedups? SSE to the rescue! Www-document. Available at: http://software.intel.com/file/1000.

Lüke, Hans D. 1999. The Origins of the Sampling Theorem. IEEE Communications Magazine, April. Www-document. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.163.2887&rep=rep1&type=pdf. Read on 20.9.2011.

Lyons, Richard G. 2004. Understanding Digital Signal Processing, Second Edition. Prentice Hall Professional Technical Reference.

Microsoft. 2011. MMX, SSE, and SSE2 Intrinsics. Www-document. Available at http://msdn.microsoft.com/en-us/library/y0dh78ez%28v=vs.80%29.asp. Read on 12.9.2011

National Instruments (NI). 2012. Windowing: Optimizing FFT's Using Window Functions. Www-document. Available at http://zone.ni.com/devzone/cda/tut/p/id/4844. Read on 4.3.2012

Proakis, John G and Manolakis, Dimitris G. 1996. Digital Signal Processing: Principles, Algorithms and Applications, Third Edition. New Jersey: Prentice-Hall Inc.

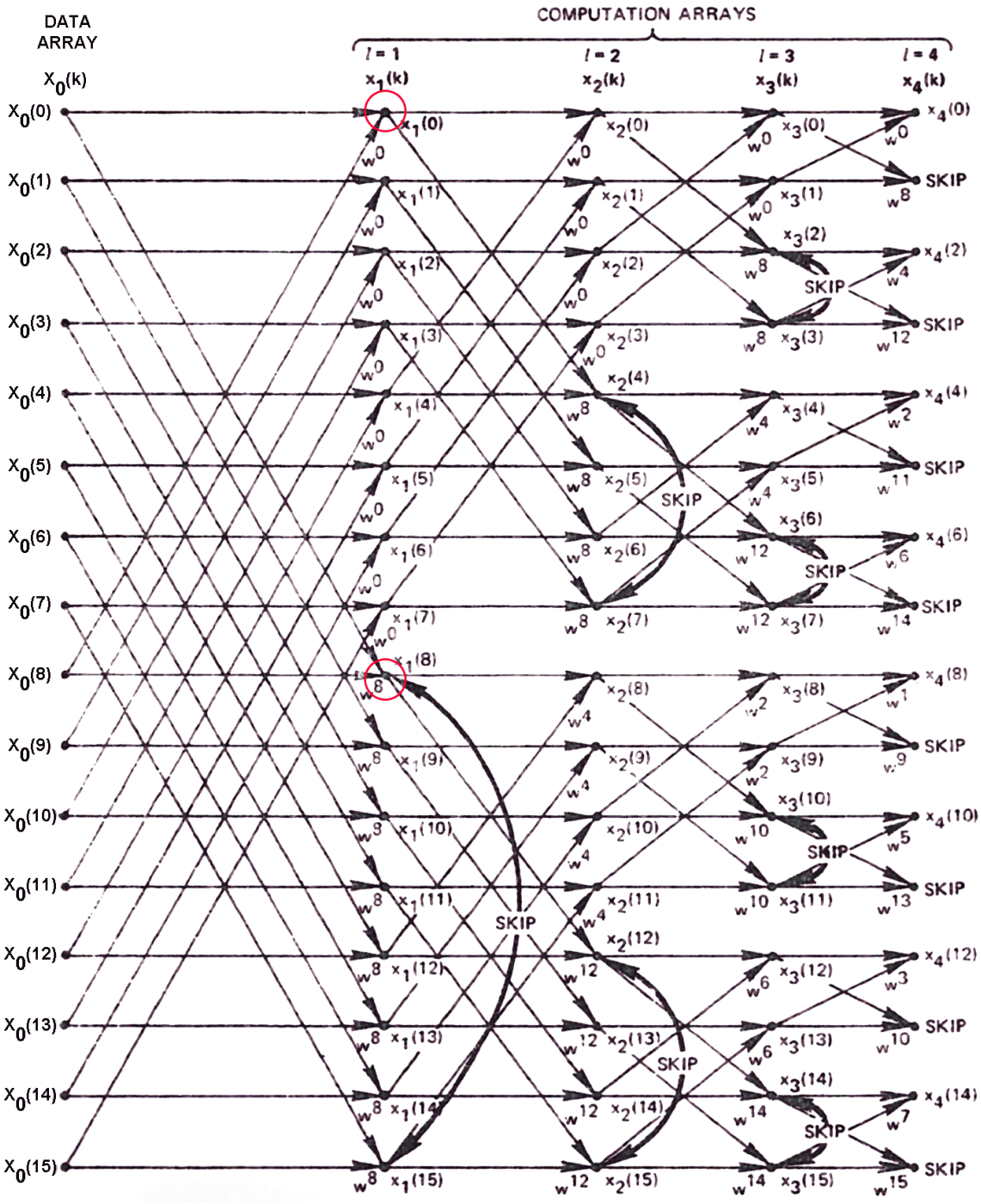
Richardson, Andrew. 2005. WCDMA Design Handbook. New York: Cambridge University Press.

Rohde & Schwarz. 2007. Operating Manual for the R&S®FSQ-B17 Digital Baseband Interface. Www-document. Available at http://www2.rohde-schwarz.com/file/FSQ-B17e.pdf. Read on 31.8.2011.

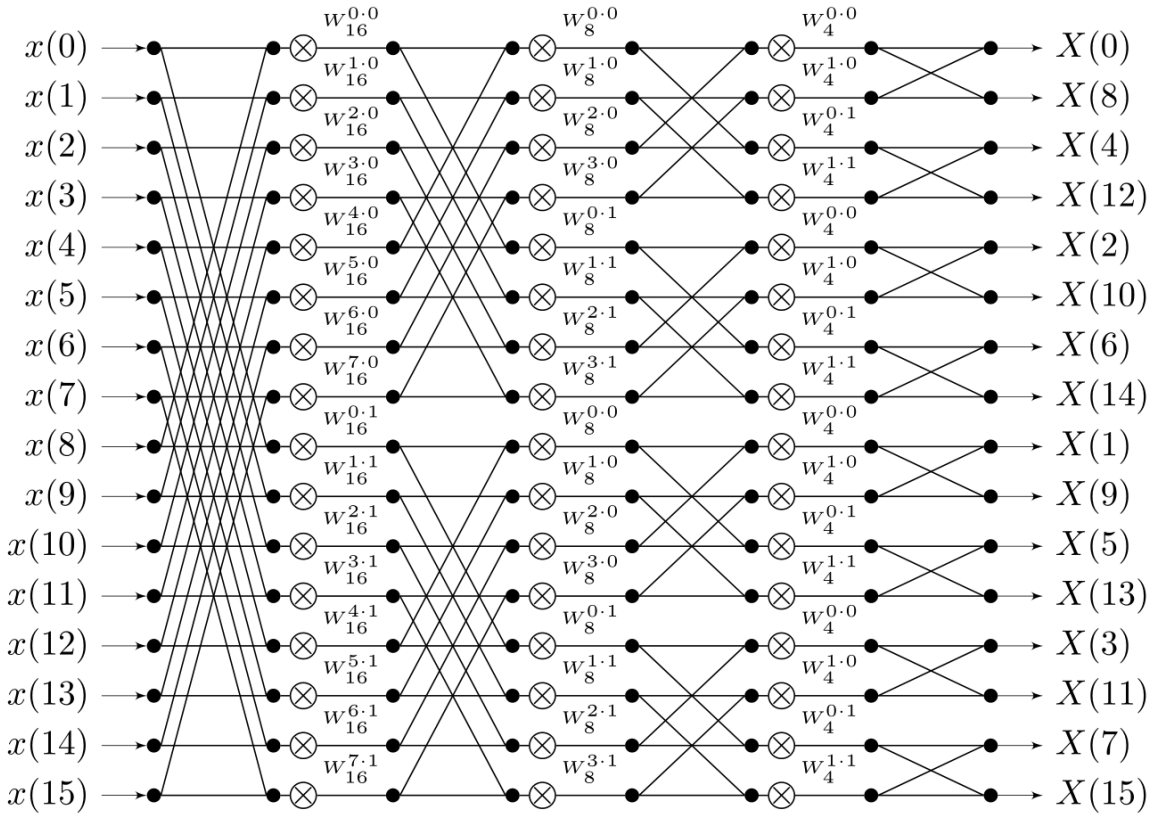
Smith, Steven W. 2002. The Scientist and Engineer's Guide to Digital Signal Processing. Www-document. Available at http://www.dspguide.com/pdfbook.htm. Read on 11.10.2011.

Smith, J.O. 2007. Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications, Second Edition. Www-document. Available at http://ccrma.stanford.edu/~jos/mdft/. Read on 12.8.2011.

ANNEX A



ANNEX B


$x(0)$
$x(1)$
$x(2)$
$x(3)$
$x(4)$
$x(5)$
$x(6)$
$x(7)$
$x(8)$
$x(9)$
$x(10)$
$x(11)$
$x(12)$
$x(13)$
$x(14)$
$x(15)$
$W_{16}^{0\cdot0}$ $W_{16}^{1\cdot0}$ $W_{16}^{2\cdot0}$ $W_{16}^{3\cdot0}$ $W_{16}^{4\cdot0}$ $W_{16}^{5\cdot0}$ $W_{16}^{6\cdot0}$ $W_{16}^{7\cdot0}$ $W_{16}^{0\cdot1}$ $W_{16}^{1\cdot1}$ $W_{16}^{2\cdot1}$ $W_{16}^{3\cdot1}$ $W_{16}^{4\cdot1}$ $W_{16}^{5\cdot1}$ $W_{16}^{6\cdot1}$ $W_{16}^{7\cdot1}$
$W_{8}^{0\cdot0}$ $W_{8}^{1\cdot0}$ $W_{8}^{2\cdot0}$ $W_{8}^{3\cdot0}$ $W_{8}^{0\cdot1}$ $W_{8}^{1\cdot1}$ $W_{8}^{2\cdot1}$ $W_{8}^{3\cdot1}$ $W_{8}^{0\cdot0}$ $W_{8}^{1\cdot0}$ $W_{8}^{2\cdot0}$ $W_{8}^{3\cdot0}$ $W_{8}^{0\cdot1}$ $W_{8}^{1\cdot1}$ $W_{8}^{2\cdot1}$ $W_{8}^{3\cdot1}$
$W_{4}^{0\cdot0}$ $W_{4}^{1\cdot0}$ $W_{4}^{0\cdot1}$ $W_{4}^{1\cdot1}$ $W_{4}^{0\cdot0}$ $W_{4}^{1\cdot0}$ $W_{4}^{0\cdot1}$ $W_{4}^{1\cdot1}$ $W_{4}^{0\cdot0}$ $W_{4}^{1\cdot0}$ $W_{4}^{0\cdot1}$ $W_{4}^{1\cdot1}$ $W_{4}^{0\cdot0}$ $W_{4}^{1\cdot0}$ $W_{4}^{0\cdot1}$ $W_{4}^{1\cdot1}$
$X(0)$
$X(8)$
$X(4)$
$X(12)$
$X(2)$
$X(10)$
$X(6)$
$X(14)$
$X(1)$
$X(9)$
$X(5)$
$X(13)$
$X(3)$
$X(11)$
$X(7)$
$X(15)$

ANNEX C

In the last chapter (2.4.1 Euler's formula and the 'Twiddle factors') we took a look at a cut-off Cooley-Tukey twiddle factor portion of the formula. Now we will look at the basic mathematical operations and expressions and hopefully realize how we can use them to calculate the DFT and the FFT.

Even the most simplest forms of the FFT and DFT formulas use the summation operator symbol $\sum x$. Lets look at a modified one from the previous chapter, with similar index notation:

$$X_k = \sum_{n=k}^{k+3} y_n$$

Because the summation operation is a key definition and is used extensive lets look at this example in detail and see what is actually happening.

The most basic thing to realize is that X and y represent some arbitrary array or storage space and their indices n and k determine the accessed value (which ever value resides in that index) for this simple algorithm.

If our index indicator k begins at zero:

when k = 0, index n goes from 0 to 3, so $X_0 = y_0 + y_1 + y_2 + y_3$
when k = 1, index n goes from 1 to 4, so $X_1 = y_1 + y_2 + y_3 + y_4$
and so on.

If you are familiar with programming this is basically a simple for-loop with index increase of one and upper limit of $k+3$. Here, we just did not define an end value to *k*.

It is very important to understand the summation notation before moving on to the DFT and FFT, as it is one of the simplest representations used.

ANNEX D

Definitions used in the FFT source code:

```
#if defined(AVX)
    // AVX would hold and process 256-bits
    #define FVECTOR_SIZE sizeof(float)*2
    #define FVECTOR_INPLACE_MULTIPLIER 4
    #define FVECTOR_INPLACE_MULTIPLIER_POSITION sizeof(float)*2
    #define FVECTOR_FFT_DIVISOR sizeof(float)*2
                .
                .
                .
#endif


#if defined(SSE)

    // SSE would hold and process 128-bits
    #define FVECTOR_SIZE sizeof(float)
    #define FVECTOR_INPLACE_MULTIPLIER 2
    #define FVECTOR_INPLACE_MULTIPLIER_POSITION sizeof(float)
    #define FVECTOR_FFT_DIVISOR sizeof(float)

    // SSE, process two samples at a time so we divide by two.
    #define FVECTOR_DIVISOR 2

    // Create the mask for shuffle single precision.
    // Note the index differences -> Vector index: [0, 1, 2, 3]
    // Macro index:  [3, 2, 1, 0]
    #define FVECTOR_SHUFFLE(x,y,z,w)   (z<<6)|(y<<4)|(x<<2)|w

    #include <xmmintrin.h>
    #include <intrin.h>

    // Floating point (single precision):
    typedef __m128                              FVECTOR;
    #define FVECTOR_SHUFPS                      _mm_shuffle_ps
    #define FVECTOR_SETZERO                     _mm_setzero_ps()
    #define FVECTOR_LOAD                        _mm_load_ps
    #define FVECTOR_STOREPS                     _mm_store_ps
    #define FVECTOR_MULPS                       _mm_mul_ps
    #define FVECTOR_ADDPS                       _mm_add_ps
    #define FVECTOR_SUBPS                       _mm_sub_ps
    #define FVECTOR_STORELOWPS                  _mm_storel_pi
    #define FVECTOR_SQRTPS                      _mm_sqrt_ps
    #define FVECTOR_MOVSLDUPS                   _mm_moveldup_ps
    #define FVECTOR_MOVSHDUPS                   _mm_movehdup_ps
    #define FVECTOR_ADDSUBPS                    _mm_addsub_ps   // A0-B0, A1+B1, A2-B2, A3+B3
#endif
```

The Cooley-Tukey FFT algorithm requires the samples to be processed while being split into the even and odd index parts. This is equivalent to the bit-reversing operation seen in chapter 2.8, however instead of reversing each sample separately we stop at the even and odd split.

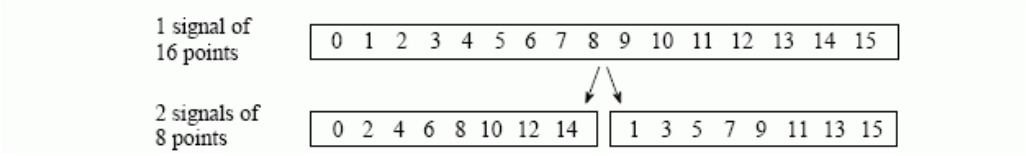


Figure 19: Cooley-Tukey FFT even and odd split

```
void CooleyTukey_SplitEvenOdd_float(   float** evenPtr, float** oddPtr, float** samplePtr,
                                       const uint32_t maxNumOfSamples)
    {
        float* sample = (*samplePtr);
        float* even = (*evenPtr);
        float* odd = (*oddPtr);

        float* addressLimit = ((*samplePtr) + (maxNumOfSamples));

        while(sample < addressLimit)
        {
            // Even real:
            (*even) = (*sample);
            ++even;
            ++sample;

            // Even imaginary:
            (*even) = (*sample);
            ++even;
            ++sample;

            // Odd real:
            (*odd) = (*sample);
            ++odd;
            ++sample;

            // Odd imaginary:
            (*odd) = (*sample);
            ++odd;
            ++sample;
        }
    }
```

The Cooley-Tukey FFT algorithm for floating point precision:

```
void Inplace_CooleyTukey_FFT(     float** coefPtr, float** twiddlePtr, float** evenPtr, float** oddPtr,
                                  float** resultsLow, float** resultsHigh, const uint32_t fftSize)
    {
        // Generate twiddle factors:
        Inplace_Cooley_GenTwiddle(twiddlePtr, 0, fftSize);
        InPlace_Cooley_GenTwiddleMultipliers(twiddlePtr, 2, fftSize);

        // Load twiddle factor:
        FVECTOR twiddle = FVECTOR_LOAD((*twiddlePtr));

        // Load the multipliers used to generate the next twiddle factor
        //(stored after the twiddle factor by default):
        FVECTOR multiplier = FVECTOR_LOAD( ((*twiddlePtr) + FVECTOR_SIZE) );

        uint32_t rotation = 0;

        // FFT loop.
        for(uint32_t i = 0; i < (fftSize / FVECTOR_FFT_DIVISOR); ++i)
        {
            // Reset pointers to the start of the data before DFT:
            float* even = (*evenPtr);
            float* odd = (*oddPtr);

            FVECTOR evenSum = FVECTOR_SETZERO;
            FVECTOR oddSum  = FVECTOR_SETZERO;

            // Generate coefficients:
            Inplace_CooleyTukey_GenCoefficients(coefPtr, 0, rotation, fftSize);

            Inplace_CooleyTukey_GenCoefMultipliers(coefPtr, FVECTOR_INPLACE_MULTIPLIER, rotation, fftSize);

            Inplace_CooleyTukey_DFT(&even, &odd, coefPtr, &evenSum, &oddSum, (fftSize / 2));

            // Twiddle factor multiplication for odd summation:
            oddSum = Inplace_Cooley_Twiddle_float(&twiddle, &multiplier, &oddSum);

           FVECTOR resultLow = FVECTOR_ADDPS(evenSum, oddSum);
           FVECTOR resultHigh = FVECTOR_SUBPS(evenSum, oddSum);     // Sign flip.

            FVECTOR_STOREPS((*resultsLow), resultLow);
            FVECTOR_STOREPS((*resultsHigh), resultHigh);

            (*resultsLow)  = (*resultsLow) + FVECTOR_SIZE;
            (*resultsHigh) = (*resultsHigh) + FVECTOR_SIZE;

            // Increase rotation by two, because two samples are processed at a time
            // (in the FFT formulae, rotation is the index 'k').
            rotation = rotation + 2;
        }

        // Move pointers to store the next data in the correct position.
        (*resultsLow)  = (*resultsLow) + fftSize;
        (*resultsHigh) = (*resultsHigh) + fftSize;

        // Move pointers to process next data.
        (*evenPtr) = (*evenPtr) + fftSize;
        (*oddPtr)  = (*oddPtr) + fftSize;
    }
```

The Cooley-Tukey algorithm DFT portion:

```
void Inplace_Cooley_DFT(   float** evenPtr, float** oddPtr, float** coefPtr, FVECTOR*
                           evenSumPtr, FVECTOR* oddSumPtr, const uint32_t dftSize)
    {
        // Same coefficient is used in the multiplication for odd and even parts.
        // Load the first coefficients:
        FVECTOR coefficient = FVECTOR_LOAD((*coefPtr));

        // Load the multipliers used to generate the next coefficients:
        FVECTOR multiplier = FVECTOR_LOAD( ((*coefPtr) + FVECTOR_SIZE) );

        // DFT loop.
        for(uint32_t i = 0; i < (dftSize / FVECTOR_DIVISOR); ++i)
        {
           // Split the coefficient to real and imaginary parts:
           FVECTOR coefReal = FVECTOR_MOVSLDUPS(coefficient);
           FVECTOR coefImag  = FVECTOR_MOVSHDUPS(coefficient);

           // Even DFT:
           FVECTOR sampleEven = FVECTOR_LOAD((*evenPtr));
           (*evenPtr) = (*evenPtr) + FVECTOR_SIZE;;
           *evenSumPtr = FVECTOR_ADDPS(*evenSumPtr,
                 VECTOR_InplaceComplexMultiply_float(&sampleEven, &coefReal, &coefImag));

           // Odd DFT:
           FVECTOR sampleOdd = FVECTOR_LOAD((*oddPtr));
           (*oddPtr) = (*oddPtr) + FVECTOR_SIZE;
           *oddSumPtr = FVECTOR_ADDPS(*oddSumPtr,
                VECTOR_InplaceComplexMultiply(&sampleOdd, &coefReal, &coefImag));

            // Generate next coefficient(s) through multiplication with the correct coefficient:
            coefficient = VECTOR_InplaceComplexMultiply_float(&multiplier, &coefReal, &coefImag);
        }
    }
```

The Cooley-Tukey algorithm twiddle-factor multiplication for the odd part:

```
FVECTOR Inplace_Cooley_Twiddle_float(   FVECTOR* twiddleFactorPtr, FVECTOR* multiplierPtr,
                                            FVECTOR* oddSumPtr)
    {
        FVECTOR twiddleReal = FVECTOR_MOVSLDUPS(*twiddleFactorPtr);
        FVECTOR twiddleImg  = FVECTOR_MOVSHDUPS(*twiddleFactorPtr);

        // Generate next twiddle factor(s) through multiplication with the correct multiplier:
        *twiddleFactorPtr = VECTOR_InplaceComplexMultiply_float(multiplierPtr, &twiddleReal, &twiddleImg);

        FVECTOR multiply1 = FVECTOR_MULPS(*oddSumPtr, twiddleReal);
        FVECTOR multiply2 = FVECTOR_MULPS(*oddSumPtr, twiddleImg);

        FVECTOR shuffle = FVECTOR_SHUFPS(multiply2, multiply2, FVECTOR_SHUFFLE(2, 3, 0, 1));

        return FVECTOR_ADDSUBPS(multiply1, shuffle);
    }
```

Complex multiplication using vector intrinsics (requires at least SSE3 support):

```
FVECTOR VECTOR_InplaceComplexMultiply_float(FVECTOR* signalPtr, FVECTOR* coefReal, FVECTOR* coefImag)
    {
        // Multiply samples with the coefficients:
        FVECTOR multiply1 = FVECTOR_MULPS(*signalPtr, *coefReal);
        FVECTOR multiply2 = FVECTOR_MULPS(*signalPtr, *coefImag);

        // Shuffle multiply2 result, so that the final part of the complex multiplication is calculated correctly:
        FVECTOR shuffle = FVECTOR_SHUFPS(multiply2, multiply2, FSSE_SHUFFLE(2, 3, 0, 1));

        // Final summation of complex multiplication:
        return FVECTOR_ADDSUBPS(multiply1, shuffle);
    }
```

Coefficient and multiplier generators can be combined into one if one wants to do so. Multipliers make use of the cyclic redundancy and generate the next coefficient by multiplying it with the appropriate multiplier.

```
void Inplace_CooleyTukey_GenCoefficients(   float** coefPtr, int32_t multiplier, const int32_t rotation,
                                        const uint32_t fftSize)
    {
        // Multipliers for generating next DFT coefficients:
        // SSE, stored in multiplier in index 0 to 3.
        // AVX, stored in multiplier 0 to 7.
        float* coef = (*coefPtr);

        //  Coefficients: exp(2*pi*m*k / N / 2)
        //  Rotation = k, multiplier = m.
        const float component = (float)( (M_PI / fftSize) * rotation);

        for(int32_t i = 0; i < FVECTOR_SIZE; i = i + 2)
        {
            (*coef) = cosf(component * multiplier);     // real
            ++coef;

            (*coef) = -sinf(component * multiplier);    // imag
            ++coef;

            multiplier = multiplier + 1;
        }
    }

    void Inplace_CooleyTukey_GenTwiddle(float** twiddlePtr, int32_t multiplier, const uint32_t fftSize)
    {
        // Multipliers for generating next DFT coefficients:
        // SSE, stored in multiplier in index 0 to 3.
        // AVX, stored in multiplier 0 to 7.
        float* twiddle = (*twiddlePtr);

        //  Coefficients: exp(2*pi*m / N)
        //  multiplier = m.
        const float component = (float)M_PI * 2 / fftSize;

        for(int32_t i = 0; i < FVECTOR_SIZE; i = i + 2)
        {
            (*twiddle) = cosf(component * multiplier);     // real
            ++twiddle;

            (*twiddle) = -sinf(component * multiplier);    // imag
            ++twiddle;

            multiplier = multiplier + 1;
        }
    }
```

```
void Inplace_CooleyTukey_GenCoefMultipliers(   float** coefPtr, const int32_t multiplier,
                                        const int32_t rotation, const uint32_t fftSize)
    {
        // Multipliers for generating next DFT coefficients:
        // SSE, stored in coefPtr, in index 4 to 7.
        // AVX, stored in index 8 to 15.
        float* coefMultiplier = (*coefPtr) + FVECTOR_INPLACE_MULTIPLIER_POSITION;

        //  Generator coefficients: exp(2*pi*m*k / N / 2)
        //  Rotation = k, multiplier = m.
        const float component = (float)( (M_PI / fftSize) * rotation);

        for(int32_t i = 0; i < FVECTOR_SIZE; i = i + 2)
        {
            (*coefMultiplier) = cosf(component * multiplier);     // real
            ++coefMultiplier;

            (*coefMultiplier) = -sinf(component * multiplier);    // imag
            ++coefMultiplier;
        }
    }

void InPlace_CooleyTukey_GenTwiddleMultipliers(float** twiddlePtr, const int32_t rotation, const uint32_t fftSize)
    {
        // Multipliers for generating next twiddle coefficients:
        // SSE, stored in twiddlePtr in index 4 to 7.
        // AVX, stored in index 8 to 15.
        float* twiddleMultiplier = (*twiddlePtr) + FVECTOR_INPLACE_MULTIPLIER_POSITION;

        //  Twiddle factors: exp(2*pi*k / N)
        //  rotation = k.
        const float component = (float)M_PI * 2 / fftSize;

        for(int32_t i = 0; i < FVECTOR_SIZE; i = i + 2)
        {
            (*twiddleMultiplier) = cosf(component * rotation);    // real
            ++twiddleMultiplier;

            (*twiddleMultiplier) = -sinf(component *rotation);    // imag
            ++twiddleMultiplier;
        }
    }
```

ANNEX E

3GPP TS 25.101

3rd Generation Partnership Project;
Technical Specification Group Radio Access Network;
User Equipment (UE) radio transmission and reception (FDD)

6.8.1 Transmit pulse shape filter

The transmit pulse shaping filter is a root-raised cosine (RRC) with roll-off α = 0.22 in the frequency domain. The impulse response of the chip impulse filter RC0(t) is:

$$RC_0(t) = \frac{\sin\left(\pi \frac{t}{T_C}(1-\alpha)\right) + 4\alpha \frac{t}{T_C}\cos\left(\pi \frac{t}{T_C}(1+\alpha)\right)}{\pi \frac{t}{T_C}\left(1-\left(4\alpha \frac{t}{T_C}\right)^2\right)}$$

Where the roll-off factor α = 0.22 and the chip duration is

$$T = \frac{1}{chiprate} \approx 0.26042\mu s$$