FAST FOURIER TRANSFORMS

C. Sidney Burrus
Rice University

LibreTexts™

Rice University

Fast Fourier Transforms

C. Sidney Burrus

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide and accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 07/13/2025

TABLE OF CONTENTS

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Fast Fourier Transforms

Topic hierarchy

1.1: Introduction

The development of fast algorithms usually consists of using special properties of the algorithm of interest to remove redundant or unnecessary operations of a direct implementation. Because of the periodicity, symmetries, and orthogonality of the basis functions and the special relationship with convolution, the discrete Fourier transform (DFT) has enormous capacity for improvement of its arithmetic efficiency.

There are four main approaches to formulating efficient DFT algorithms. The first two break a DFT into multiple shorter ones. This is done in Multidimensional Index Mapping by using an index map and in Polynomial Description of Signals by polynomial reduction. The third is Factoring the Signal Processing Operators which factors the DFT operator (matrix) into sparse factors. The DFT as Convolution or Filtering develops a method which converts a prime-length DFT into cyclic convolution. Still another approach is interesting where, for certain cases, the evaluation of the DFT can be posed recursively as evaluating a DFT in terms of two half-length DFTs which are each in turn evaluated by a quarter-length DFT and so on.

The very important computational complexity theorems of Winograd are stated and briefly discussed in Winograd's Short DFT Algorithms. The specific details and evaluations of the Cooley-Tukey FFT and Split-Radix FFT are given in The Cooley-Tukey Fast Fourier Transform Algorithm, and PFA and WFTA are covered in The Prime Factor and Winograd Fourier Transform Algorithms. A short discussion of high speed convolution is given in Convolution Algorithms, both for its own importance, and its theoretical connection to the DFT. We also present the chirp, Goertzel, QFT, NTT, SR-FFT, Approx FFT, Autogen, and programs to implement some of these.

Ivan Selesnick gives a short introduction in Winograd's Short DFT Algorithms to using Winograd's techniques to give a highly structured development of short prime length FFTs and describes a program that will automaticlly write these programs. Markus Pueschel presents his " role="presentation" style="position:relative;" tabindex="0"> Algebraic Signal Processing" in DFT and FFT - An Algebraic View on describing the various FFT algorithms. And Steven Johnson descril

The organization of the book represents the various approaches to understanding the FFT and to obtaining efficient computer programs. It also shows the intimate relationship between theory and implen

A fairly long list of references is given but it is impossible to be truly complete. I have referenced the work that I have used and that I am aware of. The collection of computer programs is also somewha

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

2: Multidimensional Index Mapping

A powerful approach to the development of efficient algorithms is to break a large problem into multiple small ones. One method for doing this with both the DFT and convolution uses a linear change of index variables to map the original one-dimensional problem into a multi-dimensional problem. This approach provides a unified derivation of the Cooley-Tukey FFT, the prime factor algorithm (PFA) FFT, and the Winograd Fourier transform algorithm (WFTA) FFT. It can also be applied directly to convolution to break it down into multiple short convolutions that can be executed faster than a direct implementation. It is often easy to translate an algorithm using index mapping into an efficient program.

Topic hierarchy

2.1: Introduction

Learning Objectives

- A change of index variable or an index mapping is used to uncouple the calculations of the discrete Fourier transform (DFT).
- This can result is a significant reduction in the required arithmetic and the resulting algorithm is called the fast Fourier transform (FFT).

A powerful approach to the development of efficient algorithms is to break a large problem into multiple small ones. One method for doing this with both the DFT and convolution uses a linear change of index variables to map the original one-dimensional problem into a multi-dimensional problem. This approach provides a unified derivation of the Cooley-Tukey FFT, the prime factor algorithm (PFA) FFT, and the Winograd Fourier transform algorithm (WFTA) FFT. It can also be applied directly to convolution to break it down into multiple short convolutions that can be executed faster than a direct implementation. It is often easy to translate an algorithm using index mapping into an efficient program.

Definition

The basic definition of the discrete Fourier transform (DFT) is

$$C(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

where nn" role="presentation" style="position:relative;" tabindex="0">n, k and N are integers,

$$j = \sqrt{-1}$$

the basis functions are the NN" role="presentation" style="position:relative;" tabindex="0">N roots of unity,

NN" role="presentation" style="position:relative;" tabindex="0">

$$W_N = e^{\frac{-j2\pi}{N}}$$

and k=0,1,2,...,N-1

If the NNN" role="presentation" style="position:relative;" tabindex="0"> values of the transform are calculated from the N values of the data, $x(n)$, it is easily seen that N^2 complex multiplications an

N2N2" role="presentation" style="position:relative;" tabindex="0">Contributor

- ContribEEBurrus

2.2: The Index Map

For a length-N sequence, the time index takes on the values $n = 0, 1, 2, \ldots, N-1$.

When the length of the DFT is not prime, N can be factored as $N = N_1 N_2$ and two new independent variables can be defined over the ranges $n_1 = 0, 1, 2, \ldots, N_1 - 1$ and $n_2 = 0, 1, 2, \ldots, N_2 - 1$.

A linear change of variables is defined which maps n_1 and n_2 to n and is expressed by

$$n = ((K_1 n_1 + K_2 n_2))_N$$

where K_i are integers and the notation $((x))_N$ denotes the integer residue of x modulo N. This map defines a relation between all possible combinations ofn1n1" role="presentation" style="position:relative;" tabindex="0"> n_1 and n_2 as shown in the equations below. The question as to whether all of the n are represented, i.e., whether the map is one-to-c

Case 1

N_1 and N_2 are relatively prime, i.e., the greatest common divisor $(N_1, N_2) = 1$

The integer map of the equation is one-to-one if and only if:

$$(K_1 = aN_2) \;\; and/or \;\; (K_2 = bN_1) \;\; and \;\; (K_1, N_1) = (K_2, N_2) = 1$$

where a and b are integers.

Case 2

N_1 and N_2 are not relatively prime, i.e., $(N_1, N_2) > 1$

The integer map of the above equation is one-to-one if and only if:

$$(K_1 = aN_2) \;\; and \;\; (K_2 \neq bN_1) \;\; and \;\; (a, N_1) = (K_2, N_2) = 1$$

or

$$(K_1 \neq aN_2) \;\; and \;\; (K_2 = bN_1) \;\; and \;\; (K_1, N_1) = (b, N_2) = 1$$

Two classes of index maps are defined from these conditions.

Type-One Index Map

The map of the above equation is called a type-one map when when integers a and b exist such that

$$K_1 = aN_2 \;\; and \;\; K_2 = bN_1,$$

Type-Two Index Map

The map of the above equation is called a type-two map when when integers a and b exist such that

$$K_1 = aN_2 \;\; or \;\; K_2 = bN_1, \;\; but\ not\ both.$$

The type-one can be used **only** if the factors of N are relatively prime, but the type-two can be used whether they are relatively prime or not. Good, Thomas and Winograd all used the type-one map in tl

The frequency index is defined by a map similar to the equation as

$$k = ((K_3 k_1 + K_4 k_2))_N$$

where the same conditions are used for determining the uniqueness of this map in terms of the integers K_3 and K_4K3K3" role="presentation" style="position:relative;" tabindex="0">.

Two-dimensional arrays for the input data and its DFT are defined using these index maps to give

$$\widehat{x}(n_1, n_2) = x((K_1 n_1 + K_2 n_2))_N$$

$$\widehat{X}(k_1, k_2) = X((K_3 k_1 + K_4 k_2))_N$$

In some of the following equations, the residue reduction notation will be omitted for clarity. These changes of variables applied to the definition of the DFT given in the equation in section 2.1 give:

$$C(k) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(n) W_N^{K_1 K_3 n_1 k_1} W_N^{K_1 K_4 n_1 k_2} W_N^{K_2 K_3 n_2 k_1} W_N^{K_2 K_4 n_2 k_2}$$

where all of the exponents are evaluated modulo NNN" role="presentation" style="position:relative;" tabindex="0">.

The amount of arithmetic required to calculate the above equation is the same as in the direct calculation of that equation. However, because of the special nature of the DFT, the integer constants K_i car

$$((K_1 K_4))_N = 0 \;\; and/or \;\; ((K_2 K_3))_N = 0$$

When this condition and those for uniqueness in the equation are applied, it is found that the K_i may **always** be chosen such that one of the terms in the equation is zero. If the N_i are relatively prime, it

An example of the Cooley-Tukey radix-4 FFT for a length-16 DFT uses the type-two map with

$$K_1 = 4, K_2 = 1, K_3 = 1, K_4 = 4$$

giving

$$n = 4n_1 + n_2$$

$$k = k_1 + 4k_2$$

The residue reduction in the equation is not needed here since n does not exceed N as n_1 and n_2 take on their values. Since, in this example, the factors of N have a common factor, only one of the conc

$$\widehat{C}(k_1, k_2) = C(k) = \sum_{n_2=0}^{3} \sum_{n_1=0}^{3} x(n) W_4^{n_1 k_1} W_{16}^{n_2 k_1} W_4^{n_2 k_2}$$

Note the definition of W_N in the equation allows the simple form of

W16K1K3=W4W16K1K3=W4" role="presentation" style="position:relative;" tabindex="0">

$$W_{16}^{K_1 K_3} = W_4$$

This has the form of a two-dimensional DFT with an extra term W_{16}, called a “twiddle factor". The inner sum over n_1 represents four length-4 DFTs, the W_{16} term represents 16 complex multiplication

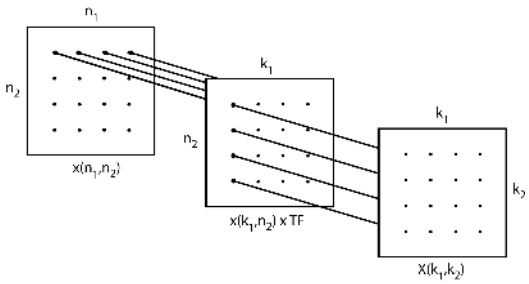


Fig. 2.2.1 Uncoupling of the Row and Column Calculations (Rectangles are Data Arrays)

The left 4-by-4 array is the mapped input data, the center array has the rows transformed, and the right array is the DFT array. The row DFTs and the column DFTs are independent of each other. The twi

This uncoupling feature reduces the amount of arithmetic required and allows the results of each row DFT to be written back over the input data locations, since that input row will not be needed again. T

An example of the type-two map used when the factors of N are relatively prime is given for $N = 15$ as

$$n = 5n_1 + n_2$$

$$k = k_1 + 3k_2$$

The residue reduction is again not explicitly needed. Although the factors 3 and 5 are relatively prime, use of the type-two map sets only one of the terms in Equation to zero. The DFT in the equation be

$$X = \sum_{n_2=0}^{4} \sum_{n_1=0}^{2} x W_3^{n_1 k_1} W_{15}^{n_2 k_1} W_5^{n_2 k_2}$$

which has the same form as the equation, including the existence of the twiddle factors (TF). Here the inner sum is five length-3 DFTs, one for each value of k_1. This is illustrated in the equation where t

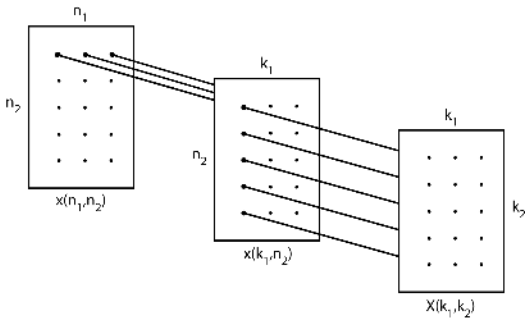


Fig. 2.2.2 Uncoupling of the Row and Column Calculations (Rectangles are Data Arrays)

The type-one map is illustrated next on the same length-15 example. This time the situation of the equation with the “and" condition is used in the equation using an index map of

$$n = 5n_1 + 3n_2$$

and

$$k = 10k_1 + 6k_2$$

The residue reduction is now necessary. Since the factors of N are relatively prime and the type-one map is being used, both terms in the equation are zero, and the equation becomes

$$\widehat{X} = \sum_{n_2=0}^{4} \sum_{n_1=0}^{2} \widehat{x} W_3^{n_1 k_1} W_5^{n_2 k_2}$$

which is similar to the equation, except that now the type-one map gives a pure two-dimensional DFT calculation with no TFs, and the sums can be done in either order. the above figures also describe th

The purpose of index mapping is to improve the arithmetic efficiency. For example a direct calculation of a length-16 DFT requires 16^2 or 256 real multiplications (recall, one complex multiplication req

Algorithms of practical interest use short DFT's that require fewer than N^2 multiplications. For example, length-4 DFTs require no multiplications and, therefore, for the length-16 DFT, only the TFs mu

The concept of using an index map can also be applied to convolution to convert a length

$$N = N_1 N_2$$

one-dimensional cyclic convolution into a N_1 by N_2 two-dimensional cyclic convolution. There is no savings of arithmetic from the mapping alone as there is with the DFT, but savings can be obtained b

Contributor

- ContribEEBurrus

2.3: In-Place Calculation of the DFT and Scrambling

Because use of both the type-one and two index maps uncouples the calculations of the rows and columns of the data array, the results of each short length N_i DFT can be written back over the data as it will not be needed again after that particular row or column is transformed. This is easily seen from Figures in section 2.2 where the DFT of the first row of $x(n_1, n_2)$ can be put back over the data rather written into a new array. After all the calculations are finished, the total DFT is in the array of the original data. This gives a significant memory savings over using a separate array for the output.

Unfortunately, the use of in-place calculations results in the order of the DFT values being permuted or scrambled. This is because the data is indexed according to the input map equation and the results are put into the same locations rather than the locations dictated by the output map equation. For example with a length-8 radix-2 FFT, the input index map is

$$n = 4n_1 + 2n_2 + n_3$$

which to satisfy the equation requires an output map of

$$k = k_1 + 2k_2 + 4k_3$$

The in-place calculations will place the DFT results in the locations of the input map and these should be reordered or unscrambled into the locations given by the output map. Examination of these two maps shows the scrambled output to be in a “bit reversed" order.

For certain applications, this scrambled output order is not important, but for many applications, the order must be unscrambled before the DFT can be considered complete. Because the radix of the radix-2 FFT is the same as the base of the binary number representation, the correct address for any term is found by reversing the binary bits of the address. The part of most FFT programs that does this reordering is called a bit-reversed counter. Examples of various unscramblers are found in the appendices.

The development here uses the input map and the resulting algorithm is called “decimation-in-frequency". If the output rather than the input map is used to derive the FFT algorithm so the correct output order is obtained, the input order must be scrambled so that its values are in locations specified by the output map rather than the input map. This algorithm is called “decimation-in-time". The scrambling is the same bit-reverse counting as before, but it precedes the FFT algorithm in this case. The same process of a post-unscrambler or pre-scrambler occurs for the in-place calculations with the type-one maps. It is possible to do the unscrambling while calculating the FFT and to avoid a separate unscrambler. This is done for the Cooley-Tukey FFT and for the PFA.

If a radix-2 FFT is used, the unscrambler is a bit-reversed counter. If a radix-4 FFT is used, the unscrambler is a base-4 reversed counter, and similarly for radix-8 and others. However, if for the radix-4 FFT, the short length-4 DFTs (butterflies) have their outputs in bit-revered order, the output of the total radix-4 FFT will be in bit-reversed order, not base-4 reversed order. This means any radix- 2^n FFT can use the same radix-2 bit-reversed counter as an unscrambler if the proper butterflies are used.

Contributor

- ContribEEBurrus

2.4: Efficiencies Resulting from Index Mapping with the DFT

In this section the reductions in arithmetic in the DFT that result from the index mapping alone will be examined. In practical algorithms several methods are always combined, but it is helpful in understanding the effects of a particular method to study it alone.

The most general form of an uncoupled two-dimensional DFT is given by

$$X(k_1,k_2)=\sum_{n_2=0}^{N_2-1}\left\{\sum_{n_1=0}^{N_1-1}x(n_1,n_2)f_1(n_1,n_2,k_1)\right\}f_2(n_2,k_1,k_2)$$

where the inner sum calculates N_2 length-N_1 DFT's and, if for a type-two map, the effects of the TFs. If the number of arithmetic operations for a length-N DFT is denoted by $F(N)$, the number of operations for this inner sum is

F=N2F(N1)F=N2F(N1)" role="presentation" style="position:relative;" tabindex="0">

$$F=N_2F(N_1)$$

The outer sum which gives N_1 length-N_2 DFT's requires $N_1F(N_2)$ operations. The total number of arithmetic operations is then

$$F=N_2F(N_1)+N_1F(N_2)$$

The first question to be considered is for a fixed length N, what is the optimal relation of N_1 and N_2 in the sense of minimizing the required amount of arithmetic. To answer this question, N_1 and N_2 a

$$F(N_i)=N_i^2$$

Thus **Efficiencies Resulting from Index Mapping with the DFT** becomes

$$F=N_2N_1^2+N_1N_2^2=N(N_1+N_2)=N(N_1+NN_1^{-1})$$

To find the minimum of F over N_1, the derivative of F with respect to N_1 is set to zero (temporarily assuming the variables to be continuous) and the result requires $N_1=N_2$

$$\frac{\mathrm{d}F}{\mathrm{d}N_1}=0\quad\Rightarrow N_1=N_2$$

This result is also easily seen from the symmetry of N_1, and in N_2, in $N=N_1N_2$. If a more general model of the arithmetic complexity of the short DFT's is used, the same result is obtained, but a clos

N1=N2N1=N2" role="presentation" style="position:relative;" tabindex="0">If only the effects of the index mapping are to be considered, then the $F(N)=N^2$ model is used and the equation states tha

F(N)=N2F(N)=N2" role="presentation" style="position:relative;" tabindex="0">

$$N=R^M$$

there are now M length-R DFT's and, since the factors are all equal, the index map must be type two. This means there must be twiddle factors.

In order to simplify the analysis, only the number of multiplications will be considered. If the number of multiplications for a length-R DFT is $F(R)$, then the formula for operation counts in the equatio

$$F=N\sum_{i=1}^{M}\frac{F(N_i)}{N_i}=NM\frac{F(R)}{R}$$

for $N_i=R$

$$F=N\ln R(N)\frac{F(R)}{R}=(N\ln N)\frac{F(R)}{R\ln R}$$

This is a very important formula which was derived by Cooley and Tukey in their famous paper on the FFT. It states that for a given R which is called the radix, the number of multiplications (and additi

In order to get some idea of the “best" radix, the number of multiplications to compute a length-R DFT is assumed to be

$$F(R)=R^x$$

If this is used with the equation, the optimal R can be found.

$$\frac{\mathrm{d}F}{\mathrm{d}R}=0\quad\Rightarrow R=e^{\frac{1}{(x-1)}}$$

For $x=2$ this gives $R=e$, with the closest integer being three.

The result of this analysis states that if no other arithmetic saving methods other than index mapping are used, and if the length-R DFT's plus TFs requireF=R2F=R2" role="presentation" style="position

While this is an interesting result from the analysis of the effects of index mapping alone, in practice, index mapping is almost always used in conjunction with special algorithms for the short length-N_i

$$F(N_i)=KN_i$$

and the operation count F in "Efficiencies Resulting from Index Mapping with the DFT" is independent of N_i. Therefore, the derivative of F is zero for all N_i. Obviously, these particular cases must be

Contributor

- ContribEEBurrus

2.5: The FFT as a Recursive Evaluation of the DFT

It is possible to formulate the DFT so a length-N DFT can be calculated in terms of two length-$(N/2)$ DFTs.

And, if $N = 2^M$ each of those length-$(N/2)$ DFTs can be found in terms of length-$(N/4)$ DFTs. This allows the DFT to be calculated by a recursive algorithm with M recursions, giving the familiar order $N \log N$ arithmetic complexity.

Calculate the even indexed DFT values from the equation by:

$$C(2k) = \sum_{n=0}^{N-1} x(n) W_N^{2nk} = \sum_{n=0}^{N-1} x(n) W_{N/2}^{nk}$$

$$C(2k) = \sum_{n=0}^{N/2-1} x(n) W_N^{2nk} + \sum_{n=N/2}^{N-1} x(n) W_{N/2}^{nk}$$

$$C(2k) = \sum_{n=0}^{N/2-1} \{x(n) + x(n+N/2)\} W_{N/2}^{nk}$$

and a similar argument gives the odd indexed values as:

$$C(2k+1) = \sum_{n=0}^{N/2-1} \{x(n) - x(n+N/2)\} W_N^n W_{N/2}^{nk}$$

Together, these are recursive DFT formulas expressing the length-N DFT of $x(n)$ in terms of length-N/2 DFTs:

$$C(2k) = DFT_{N/2}\{x(n) + x(n+N/2)\}$$

$$C(2k+1) = DFT_{N/2}\{[x(n) - x(n+N/2)]W_N^n\}$$

This is a “decimation-in-frequency" (DIF) version since it gives samples of the frequency domain representation in terms of blocks of the time domain signal.

A recursive Matlab program which implements this is given by:

```
function c = dftr2(x)
% Recursive Decimation-in-Frequency FFT algorithm, csb 8/21/07
L = length(x);
if L > 1
     L2 = L/2;
     TF = exp(-j*2*pi/L).^[0:L2-1];
     c1 = dftr2( x(1:L2) + x(L2+1:L));
     c2 = dftr2((x(1:L2) - x(L2+1:L)).*TF);
     cc = [c1';c2'];
     c = cc(:);
else
     c  = x;
end
```

NOT_CONVERTED_YET: caption

DIF Recursive FFT for $N = 2^M$

A DIT version can be derived in the form:

$$C(k) = DFT_{N/2}\{x(2n)\} + W_N^k DFT_{N/2}\{x(2n+1)\}$$

$$C(k+N/2) = DFT_{N/2}\{x(2n)\} - W_N^k DFT_{N/2}\{x(2n+1)\}$$

which gives blocks of the frequency domain from samples of the signal.

A recursive Matlab program which implements this is given by:

```
function c = dftr(x)
% Recursive Decimation-in-Time FFT algorithm, csb
L = length(x);
if L > 1
     L2 = L/2;
     ce = dftr(x(1:2:L-1));
     co = dftr(x(2:2:L));
     TF = exp(-j*2*pi/L).^[0:L2-1];
     c1 = TF.*co;
     c  = [(ce+c1), (ce-c1)];
else
     c  = x;
end
```

NOT_CONVERTED_YET: caption

DIT Recursive FFT for $N = 2^M$

Similar recursive expressions can be developed for other radices and and algorithms. Most recursive programs do not execute as efficiently as looped or straight code, but some can be very efficient, e.g. parts of the FFTW.

Note a length- 2^M sequence will require M recursions, each of which will require $N/2$ multiplications. This give the $N \log N$ formula that the other approaches also derive.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

3: Polynomial Description of Signals

Polynomials are important in digital signal processing because calculating the DFT can be viewed as a polynomial evaluation problem and convolution can be viewed as polynomial multiplication This is indeed the basis for the important results of Winograd discussed in Winograd's Short DFT Algorithms.

Topic hierarchy

3.1: Introduction

Learning Objectives

- Polynomials are important in digital signal processing because calculating the DFT can be viewed as a polynomial evaluation problem and convolution can be viewed as polynomial multiplication

This is indeed the basis for the important results of Winograd discussed in Winograd's Short DFT Algorithms. A length-N signal $x(n)$ will be represented by an $N-1$ degree polynomial $X(s)$ defined by:

$$X(s) = \sum_{n=0}^{N-1} x(n)s^n$$

This polynomial $X(s)$ is a single entity with the coefficients being the values of $x(n)$. It is somewhat similar to the use of matrix or vector notation to efficiently represent signals which allows use of new mathematical tools.

The convolution of two finite length sequences, $x(n)$ and $h(n)$, gives an output sequence defined by

$$y(n) = \sum_{k=0}^{N-1} x(k)h(n-k)$$

$$n = 0, 1, 2, \ldots, 2N-1 \quad where \quad h(k) = 0 \quad for \quad k < 0$$

This is exactly the same operation as calculating the coefficients when multiplying two polynomials. The equation is the same as

$$Y(s) = X(s)H(s)$$

In fact, convolution of number sequences, multiplication of polynomials, and the multiplication of integers (except for the carry operation) are all the same operations. To obtain cyclic convolution, where the indices in the equation are all evaluated modulo N, the polynomial multiplication in the equation is done modulo the polynomial:

$$P(s) = s^N - 1$$

This is seen by noting that $N = 0 \, mod \, N$ therefore, $s^N = 1$ and the polynomial modulus is $s^N = 1$.

Contributor

- ContribEEBurrus

3.2: Polynomial Reduction and the Chinese Remainder Theorem

Residue reduction of one polynomial modulo another is defined similarly to residue reduction for integers. A polynomial $F(s)$ has a residue polynomial $R(s)$ modulo $P(s)$ if, for a given $F(s)$ and $P(s)$, a $Q(s)$ and $R(s)$ exist such that:

$$F(s) = Q(s)P(s) + R(s)$$

with

$$degree R(s) < degree P(s)$$

The notation that will be used is:

$$R(s) = ((F(s)))_{P(s)}$$

For example,

$$(s+1) = ((s^4 + s^3 - s - 1))_{(s^2-1)}$$

The concepts of factoring a polynomial and of primeness are an extension of these ideas for integers. For a given allowed set of coefficients (values of $x(n)$), any polynomial has a unique factored representation:

$$F(s) = \prod_{i=1}^{M} F_i(s)^{k_i}$$

where the $F_i(s)$ are relatively prime. This is analogous to the fundamental theorem of arithmetic.

There is a very useful operation that is an extension of the integer Chinese Remainder Theorem (CRT) which says that if the modulus polynomial can be factored into relatively prime factors:

$$P(s) = P_1(s)P_2(s)$$

then there exist two polynomials, $K_1(s)$ and $K_2(s)$, such that any polynomial $F(s)$ can be recovered from its residues by:

$$F(s) = K_1(s)F_1(s) + K_2(s)F_2(s) \; mod \; P(s)$$

where F_1 and F_2 are the residues given by:

$$F_1(s) = ((F(s)))_{P_1(s)}$$

and

$$F_2(s) = ((F(s)))_{P_2(s)}$$

if the order of $F(s)$ is less than $P(s)$. This generalizes to any number of relatively prime factors of $P(s)$ and can be viewed as a means of representing $F(s)$ by several lower degree polynomials, $F_i(s)$.

This decomposition of $F(s)$ into lower degree polynomials is the process used to break a DFT or convolution into several simple problems which are solved and then recombined using the CRT of the above equation. This is another form of the “divide and conquer" or “organize and share" approach similar to the index mappings in Multidimensional Index Mapping.

One useful property of the CRT is for convolution. If cyclic convolution of $x(n)$ and $x(n)$ is expressed in terms of polynomials by:

$$Y(s) = H(s)X(s) \; mod \; P(s)$$

where

$$P(s) = s^N - 1$$

and if $P(s)$ is factored into two relatively prime factors

P=P1P2P=P1P2" role="presentation" style="position:relative;" tabindex="0">

$$P = P_1P_2$$

using residue reduction of $H(s)$ and $X(s)$ modulo P_1 and P_2, the lower degree residue polynomials can be multiplied and the results recombined with the CRT. This is done by:

$$Y(s) = ((K_1H_1X_1 + K_2H_2X_2))_P$$

where

$$H_1 = ((H))_{P_1}, \quad X_1 = ((X))_{P_1}, \quad H_2 = ((H))_{P_2}, \quad X_2 = ((X))_{P_2},$$

and K_1 and K_2 are the CRT coefficient polynomials from the above equation. This allows two shorter convolutions to replace one longer one.

Another property of residue reduction that is useful in DFT calculation is polynomial evaluation. To evaluate $F(s)$ at $s = x$, $F(s)$ is reduced modulo $s = x$.

$$F(x) = ((F(s)))_{s-x}$$

This is easily seen from the definition in the equation.

$$F(S) = Q(s)(s - x) + R(s)$$

Evaluating $s = x$, gives $R(s) = F(x)$ which is a constant. For the DFT this becomes:

$$C(k) = ((X(s)))_{s-W^k}$$

Contributor

- ContribEEBurrus

3.3: The DFT as a Polynomial Evaluation

The Z-transform of a number sequence $x(n)$ is defined as:

$$X(z) = \sum_{n=0}^{\infty} x(n) z^{-n}$$

which is the same as the polynomial description in Equation 3.1.1 but with a negative exponent. For a finite length-N sequence the above equation becomes

$$X(z) = \sum_{n=0}^{\infty} x(n) z^{-n}$$

$$X(z) = x(0) + x(1) z^{-1} + x(2) z^{-2} + \ldots + x(N-1) z^{-N+1}$$

This $N-1$ order polynomial takes on the values of the DFT of $x(n)$ when evaluated at

$$z = e^{j2\pi k/N}$$

which gives

$$C(k) = X(z) \mid_{z=e^{j2\pi k/N}} = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N}$$

In terms of the positive exponent polynomial from the above equation, the DFT is:

$$C(k) = X(s)|_{s=W^k}$$

where

$$W = e^{-j2\pi/N}$$

is an N^{th} root of unity (raising W to the N^{th} power gives one). The N values of the DFT are found from $X(s)$ evaluated at the N N^{th} roots of unity which are equally spaced around the unit circle in the complex s plane.

One method of evaluating $X(z)$ is the so-called Horner's rule or nested evaluation. When expressed as a recursive calculation, Horner's rule becomes the Goertzel algorithm which has some computational advantages especially when only a few values of the DFT are needed.

Another method for evaluating $X(s)$ is the residue reduction modulo $(s - W^k)$ as shown in Equation 3.3.1 above. Each evaluation requires N multiplications and therefore, N^2 multiplications for the N values of $C(k)$.

$$C(k) = ((X(s)))_{(s-W^k)}$$

A considerable reduction in required arithmetic can be achieved if some operations can be shared between the reductions for different values of k. This is done by carrying out the residue reduction in stages that can be shared rather than done in one step for each k in the above equation.

The N values of the DFT are values of $X(s)$ evaluated at s equal to the N roots of the polynomial

$$P(s) = s^N - 1$$

which are W^k. First, assuming N is even, factor $P(s)$ as:

$$P(s) = (s^N - 1) = P_1(s) P_{2(s)} = (s^{N/2} - 1)(s^{N/2} + 1)$$

$X(s)$ is reduced modulo these two factors to give two residue polynomials, $X_1(s)$ and $X_2(s)$. This process is repeated by factoring P_1 and further reducing X_1 then factoring P_2 and reducing X_2. This is continued until the factors are of first degree which gives the desired DFT values as in Equation. This is illustrated for a length-8 DFT. The polynomial whose roots are W^k, factors as

$$P(s) = s^8 - 1$$

$$P(s) = [s^4 - 1][s^4 + 1]$$

$$P(s) = [(s^2 - 1)(s^2 + 1)][(s^2 - j)(s^2 + j)]$$

$$P(s) = [(s-1)(s+1)(s-j)(s+j)][s-a)(s+a)(s-ja)(s+ja)]$$

where

$$a^2 = j$$

Reducing $X(s)$ by the first factoring gives two third degree polynomials:

$$X(s) = x_0 + x_1 s + x_2 s^2 + \ldots + x_7 s^7$$

gives the residue polynomials

$$X_1(s) = ((X(S)))_{(s^4-1)} = (x_0 + x_4) + (x_1 + x_5)s + (x_2 + x_6)s^2 + (x_3 + x_7)s^3$$

$$X_2(s) = ((X(S)))_{(s^4+1)} = (x_0 - x_4) + (x_1 - x_5)s + (x_2 - x_6)s^2 + (x_3 - x_7)s^3$$

Two more levels of reduction are carried out to finally give the DFT. Close examination shows the resulting algorithm to be the decimation-in-frequency radix-2 Cooley-Tukey FFT. Martens has used this approach to derive an efficient DFT algorithm.

Other algorithms and types of FFT can be developed using polynomial representations and some are presented in the generalization in DFT and FFT - An Algebraic View.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

4: The DFT as Convolution or Filtering

A major application of the FFT is fast convolution or fast filtering where the DFT of the signal is multiplied term-by-term by the DFT of the impulse (helps to be doing finite impulse response (FIR) filtering) and the time-domain output is obtained by taking the inverse DFT of that product. What is less well-known is the DFT can be calculated by convolution. There are several different approaches to this, each with different application.

Topic hierarchy

CHAPTER OVERVIEW

Front Matter

Rice University

4: The DFT as Convolution or Filtering

C. Sidney Burrus

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide and accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 07/13/2025

4.1: Introduction

Learning Objectives

- Calculating DFT by convolution

A major application of the FFT is fast convolution or fast filtering where the DFT of the signal is multiplied term-by-term by the DFT of the impulse (helps to be doing finite impulse response (FIR) filtering) and the time-domain output is obtained by taking the inverse DFT of that product. What is less well-known is the DFT can be calculated by convolution. There are several different approaches to this, each with different application.

Contributor

- ContribEEBurrus

4.2: Rader's Conversion of the DFT into Convolution

In this section a method quite different from the index mapping or polynomial evaluation is developed. Rather than dealing with the DFT directly, it is converted into a cyclic convolution which must then be carried out by some efficient means. Those means will be covered later, but here the conversion will be explained. This method requires use of some number theory.

Definition

The DFT and cyclic convolution are defined by

$$C(k) = \sum_{n=0}^{N-1} x(n) W^{nk}$$

$$y(k) = \sum_{n=0}^{N-1} x(n) h(k-n)$$

For both, the indices are evaluated modulo N. In order to convert the DFT in Equation 4.2.1 into the cyclic convolution of Equation 4.2.2, the nk product must be changed to the $k-n$ difference. With real numbers, this can be done with logarithms, but it is more complicated when working in a finite set of integers modulo N. From number theory it can be shown that if the modulus is a prime number, a base (called a primitive root) exists such that a form of integer logarithm can be defined. This is stated in the following way. If N is a prime number, a number r called a primitive roots exists such that the integer equation

$$n = ((r^m))_N$$

creates a unique, one-to-one map of the $N-1$ member set $m = 0, \ldots, N-2$ and the $N-1$ member set $n = 1, \ldots, N-1$. This is because the multiplicative group of integers modulo a prime, p, is isomorphic to the additive group of integers modulo $(p-1)$ and is illustrated for $N = 5$ below.

r	m=	0	1	2	3	4	5	6
1		1	1	1	1	1	1	1
2		1	2	4	3	1	2	4
3		1	3	4	2	1	3	4
4		1	4	1	4	1	4	1
5		*	0	0	0	*	0	0
6		1	1	1	1	1	1	1

Table of Integers $n = ((r^m))$ modulo 5, [* not defined]

n={1,...,N-1}n={1,...,N-1}" role="presentation" style="position:relative;" tabindex="0">

n={1,...,N-1}n={1,...,N-1}" role="presentation" style="position:relative;" tabindex="0">The above Table is an array of values of r^m modulo N and it is easy to see that there are two primitive roots, 2 a

$$n = r^{-m}$$

and

$$k = r^s$$

where the term with the negative exponent (the inverse) is defined as the integer that satisfies

$$((r^{-m} r^m))_N = 1$$

If N is a prime number, r^{-m} always exists. For example

$$((2^{-1}))_5 = 3$$

The Equation 4.2.1 now becomes

$$C(r^s) = \sum_{m=0}^{N-2} x(r^{-m}) W^{r^{-m} r^s} + x(0)$$

for $s = 0, 1, \ldots, N-2$ and

$$C(0) = \sum_{n=0}^{N-1} x(n)$$

New functions are defined, which are simply a permutation in the order of the original functions, as

$$x'(m) = x(r^{-m}), \quad C'(s) = C(r^s), \quad W'(n) = W(r^n)$$

The above equation then becomes

$$C'(s) = \sum_{m=0}^{N-2} x'(m) W'(s-m) + x(0)$$

which is cyclic convolution of length $N-1$ (plus $x(0)$ and is denoted as

$$C'(k) = x'(k) * W'(k) + x(0)$$

Applying this change of variables (use of logarithms) to the DFT can best be illustrated from the matrix formulation of the DFT. The equation is written for a length-5 DFT as:

$$\begin{bmatrix} C(0) \\ C(1) \\ C(2) \\ C(3) \\ C(4) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \\ 0 & 3 & 1 & 4 & 2 \\ 0 & 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix}$$

where the square matrix should contain the terms of W^{nk} but for clarity, only the exponents nknknk" role="presentation" style="position:relative;" tabindex="0"> are shown. Separating the $x(0)$ term, a

r=2r=2" role="presentation" style="position:relative;" tabindex="0">

$$\begin{bmatrix} C(1) \\ C(2) \\ C(3) \\ C(4) \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 3 \\ 3 & 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix} + \begin{bmatrix} x(0) \\ x(0) \\ x(0) \\ x(0) \end{bmatrix}$$

and

$$C(0) = x(0) + x(1) + x(2) + x(3) + x(4)$$

which can be seen to be a reordering of the structure in the above equation. This is in the form of cyclic convolution as indicated in the above equation. Rader first showed this in 1968, stating that a prim

Until 1976, this conversion approach received little attention since it seemed to offer few advantages. It has specialized applications in calculating the DFT if the cyclic convolution is done by distributed

Contributor

- ContribEEBurrus

4.3: The Chirp Z-Transform or Bluestein's Algorithm

The DFT of $x(n)$ evaluates the Z-transform of $x(n)$ on N equally spaced points on the unit circle in the z plane. Using a nonlinear change of variables, one can create a structure which is equivalent to modulation and filtering $x(n)$ by a “chirp" signal.

The mathematical identity

$$(k-n)^2 = k^2 - 2kn + n^2$$

gives

$$nk = \frac{\left(n^2 - (k-n)^2 + k^2\right)}{2}$$

which substituted into the definition of the DFT in Multidimensional Index Mapping Equation gives

$$C(k) = \left\{ \sum_{n=0}^{N-1} [x(n) W^{n^2/2}] W^{-(k-n)^2/2} \right\} W^{k^2/2}$$

This equation can be interpreted as first multiplying (modulating) the data $x(n)$ by a chirp sequence $W^{n^2/2}$ then convolving (filtering) it, then finally multiplying the filter output by the chirp sequence to give the DFT.

Define the chirp sequence or signal as h(n)=Wn2/2h(n)=Wn2/2" role="presentation" style="position:relative;" tabindex="0">

$$h(n) = W^{n^2/2}$$

which is called a chirp because the squared exponent gives a sinusoid with changing frequency. Using this definition, we have:

$$C(n) = \left\{ [x(n)h(n)] * h^{-1} \right\} h(n)$$

We know that convolution can be carried out by multiplying the DFTs of the signals, here we see that evaluation of the DFT can be carried out by convolution. Indeed, the convolution represented by **'

As developed here, the chirp z-transform evaluates the z-transform at equally spaced points on the unit circle. A slight modification allows evaluation on a spiral and in segments and allows savings with

Two Matlab programs to calculate an arbitrary length DFT using the chirp z-transform is shown in Pre.

```
function y = chirpc(x);
% function y = chirpc(x)
% computes an arbitrary-length DFT with the
% chirp z-transform algorithm.  csb  6/12/91
%
N  = length(x);  n = 0:N-1;     %sequence length
W  = exp(-j*pi*n.*n/N);         %chirp signal
xw = x.*W;                      %modulate with chirp
ww = [conj(W(N:-1:2)),conj(W)]; %construct filter
y  = conv(ww,xw);               %convolve w filter
y  = y(N:2*N-1).*W;             %demodulate w chirp


function y = chirp(x);
% function y = chirp(x)
% computes an arbitrary-length discrete Fourier transform (DFT)
% with the chirp z transform algorithm. The linear convolution
% then required is done with FFTs.
% ioan: L. Arevalo; 11.86.91 K. Schwarz, LNT Erlangen; 6/12/91 csb.
%
N  = length(x);                 %sequence length
L  = 2^ceil(log((2*N-1))/log(2)); %FFT length
n  = 0:N-1;
W  = exp(-j*pi*n.*n/N);         %chirp signal
FW = fft([conj(W), zeros(1,L-2*N+1), conj(W(N:-1:2))],L);
y  = ifft(FW.*fft(x.',L),L);    %Convolve using FFT
y  = y(1:N).*W;                 %Demodulate
```

Contributor

- ContribEEBurrus

4.4: Goertzel's Algorithm or A Better DFT Algorithm

Goertzel's algorithm is another methods that calculates the DFT by converting it into a digital filtering problem. The method looks at the calculation of the DFT as the evaluation of a polynomial on the unit circle in the complex plane. This evaluation is done by Horner's method which is implemented recursively by an IIR filter.

The First-Order Goertzel Algorithm

The polynomial whose values on the unit circle are the DFT is a slightly modified z-transform of $x(n)$ given by

$$X(z) = \sum_{n=0}^{N-1} x(n) z^{-n}$$

which for clarity in this development uses a positive exponent . This is illustrated for a length-4 sequence as a third-order polynomial by

$$X(z) = x(3)z^3 + x(2)z^2 + x(1)z + x(0)$$

The DFT is found by evaluating the equation at $z = W^k$ which can be written as

$$C(k) = X(z)\,|_{z=W^k} = DFT\,\{x(n)\} \quad where \quad W = e^{-j2\pi/N}$$

The most efficient way of evaluating a general polynomial without any pre-processing is by “Horner's rule" which is a nested evaluation. This is illustrated for the polynomial in the equation below:

$$X(z) = \{[x(3)z + x(2)]z + x(1)z\} + x(0)$$

This nested sequence of operations can be written as a linear difference equation in the form of

$$y(m) = zy(m-1) + x(N-m)$$

with initial condition $y(0) = 0$, and the desired result being the solution at m=N. The value of the polynomial is given by

$$X(z) = y(N)$$

This equation can be viewed as a first-order IIR filter with the input being the data sequence in reverse order and the value of the polynomial at z being the filter output sampled at $m = N$. Applying this to the DFT gives the Goertzel algorithm:

$$y(m) = W^k y(m-1) + x(N-m)$$

with

$$y(0) = 0 \quad and \quad C(k) = y(n)$$

where

$$C(k) = \sum_{n=0}^{N-1} x(n) W^{nk}$$

When comparing this program with the direct calculation of equation, it is seen that the number of floating-point multiplications and additions are the same. In fact, the structures of the two algorithms look similar, but close examination shows that the way the sines and cosines enter the calculations is different. In the equation, new sine and cosine values are calculated for each frequency and for each data value, while for the Goertzel algorithm in the equation, they are calculated only for each frequency in the outer loop. Because of the recursive or feedback nature of the algorithm, the sine and cosine values are “updated" each loop rather than recalculated. This results in $2N$ trigonometric evaluations rather than $2N^2$. It also results in an increase in accumulated quantization error.

It is possible to modify this algorithm to allow entering the data in forward order rather than reverse order. The difference in the equation becomes

$$y(m) = z^{-1} y(m-1) + x(m-1)$$

if the equation becomes

$$C(k) = z^{N-1} y(N)$$

for $y(0) = 0$. This is the algorithm programmed later.

The Second-Order Goertzel Algorithm

One of the reasons the first-order Goertzel algorithm does not improve efficiency is that the constant in the feedback or recursive path is complex and, therefore, requires four real multiplications and two real additions. A modification of the scheme to make it second-order removes the complex multiplications and reduces the number of required multiplications by two.

Define the variable $q(m)$ so that

$$y(m) = q(m) - z^{-1} q(m-1)$$

This substituted into the right-hand side of the above equation gives

$$y(m) = zq(m-1) - q(m-2) + x(N-m)$$

Combining the two equations gives the second order difference equation

$$q(m) = (z + z^{-1}) q(m-1) - q(m-2) + x(N-m)$$

which together with the output equation, comprise the second-order Goertzel algorithm where

$$X(z) = Y(n)$$

for initial conditions

$$q(0) = q(-1) = 0$$

A similar development starting with the equation gives a second-order algorithm with forward ordered input as

$$q(m) = (z + z^{-1}) q(m-1) - q(m-2) + x(m-1)$$

$$y(m) = q(m) - zq(-1)$$

with

$$X(z) = z^{N-1} y(N) \quad for \quad q(0) = q(-1) = 0$$

Note that both the equations are not changed if z is replaced with z^{-1}, only the output Equation and Equation are different. This means that the polynomial $X(z)$ may be evaluated at a particular z and its inverse z^{-1} from one solution of the two equations using the output equations

$$X(z) = q(N) - z^{-1} q(N-1)$$

and

$$X(1/z) = z^{N-1} \left(q(N) - zq(N-1) \right)$$

Clearly, this allows the DFT of a sequence to be calculated with half the arithmetic since the outputs are calculated two at a time. The second-order DE actually produces a solution $q(m)$ that contains two first-order components. The output equations are, in effect, zeros that cancel one or the other pole of the second-order solution to give the desired first-order solution. In addition to allowing the calculating of two outputs at a time, the second-order DE requires half the number of real multiplications as the first-order form. This is because the coefficient of the $q(m-2)$ is unity and the coefficient of the $q(m-1)$ is real if z and z^{-1} are complex conjugates of each other which is true for the DFT.

Analysis of Arithmetic Complexity and Timings

Analysis of the various forms of the Goertzel algorithm from their programs gives the following operation count for real multiplications and real additions assuming real data.

Algorithm	Real Mults.	Real Adds	Trig Eval.
Direct DFT	$4N^2$	$4N^2$	$2N^2$
First-Order	$4N^2$	$4N^2$ - 2N	2N
Second-Order	$2N^2$ + 2N	$4N^2$	2N
Second-Order 2	N^2 + N	$2N^2$ + N	N

Timings of the algorithms on a PC in milliseconds are given in the following table.

Algorithm	N=125	N=257
Direct DFT	4.90	19.83
First-Order	4.01	16.70
Second-Order	2.64	11.04
Second-Order 2	1.32	5.55

These timings track the floating point operation counts fairly well.

Conclusions

Goertzel's algorithm in its first-order form is not particularly interesting, but the two-at-a-time second-order form is significantly faster than a direct DFT. It can also be used for any polynomial evaluation or for the DTFT at unequally spaced values or for evaluating a few DFT terms. A very interesting observation is that the inner-most loop of the Glassman-Ferguson FFT is a first-order Goertzel algorithm even though that FFT is developed in a very different framework.

In addition to floating-point arithmetic counts, the number of trigonometric function evaluations that must be made or the size of a table to store precomputed values should be considered. Since the value of the W^{nk} terms in the equation are iteratively calculate in the IIR filter structure, there is round-off error accumulation that should be analyzed in any application.

It may be possible to further improve the efficiency of the second-order Goertzel algorithm for calculating all of the DFT of a number sequence. Perhaps a fourth order DE could calculate four output values at a time and they could be separated by a numerator that would cancel three of the zeros. Perhaps the algorithm could be arranged in stages to give an $N \log N$ operation count. The current algorithm does not take into account any of the symmetries of the input index.

Contributor

- ContribEEBurrus

4.5: The Quick Fourier Transform (QFT)

One stage of the QFT can use the symmetries of the sines and cosines to calculate a DFT more efficiently than directly implementing the equation in Multidimensional Index Mapping. Similar to the Goertzel algorithm, the one-stage QFT is a better N^2 DFT algorithm for arbitrary lengths. See The Cooley-Tukey Fast Fourier Transform Algorithm .

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

Back Matter

Index

CHAPTER OVERVIEW

5: Factoring the Signal Processing Operators

Topic hierarchy

5.1: Introduction

Learning Objectives

- To introduce a third approach to removing redundancy in an algorithm to express the algorithm as an operator and then factor that operator into sparse factors

A third approach to removing redundancy in an algorithm is to express the algorithm as an operator and then factor that operator into sparse factors. This approach is used by Tolimieri, Egner, Selesnick, Elliott and others. It is presented in a more general form in DFT and FFT: An Algebraic View The operators may be in the form of a matrix or a tensor operator.

The following sections briefly describe this approach.

Contributor

- ContribEEBurrus

5.2: The FFT from Factoring the DFT Operator

The definition of the DFT in Multidimensional Index Mapping can written as a matrix-vector operation by $C = WX$ where $N = 8$

$$\begin{bmatrix} C(0) \\ C(1) \\ C(2) \\ C(3) \\ C(4) \\ C(5) \\ C(6) \\ C(7) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^8 & W^{10} & W^{12} & W^{14} \\ W^0 & W^3 & W^6 & W^9 & W^{12} & W^{15} & W^{18} & W^{21} \\ W^0 & W^4 & W^8 & W^{12} & W^{16} & W^{20} & W^{24} & W^{28} \\ W^0 & W^5 & W^{10} & W^{15} & W^{20} & W^{25} & W^{30} & W^{35} \\ W^0 & W^6 & W^{12} & W^{18} & W^{24} & W^{30} & W^{36} & W^{42} \\ W^0 & W^7 & W^{14} & W^{21} & W^{28} & W^{35} & W^{42} & W^{49} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}$$

which clearly requires $N^2 = 64$ complex multiplications and $N(N-1)$ additions. A factorization of the DFT operator, W, gives $W = F_1F_2F_3$ *and* $C = F_1F_2F_3X$.

Expanding on that gives

$$\begin{bmatrix} C(0) \\ C(4) \\ C(2) \\ C(6) \\ C(1) \\ C(5) \\ C(3) \\ C(7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ W^0 & 0 & -W^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & W^0 & 0 & -W^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & W^0 & 0 & -W^0 & 0 \\ 0 & 0 & 0 & 0 & 0 & W^2 & 0 & -W^2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ W_0 & 0 & 0 & 0 & -W_0 & 0 & 0 & 0 \\ 0 & W_1 & 0 & 0 & 0 & -W_1 & 0 & 0 \\ 0 & 0 & W_2 & 0 & 0 & 0 & -W_2 & 0 \\ 0 & 0 & 0 & W_3 & 0 & 0 & 0 & -W_3 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}$$

where the F_i matrices are sparse. Note that each has 16 (or $2N$) non-zero terms and F_2 and F_3 have 8 (or N) non-unity terms. If $N = 2^M$, then the number of factors is $\log(N) = M$. In another form with the twiddle factors separated so as to count the complex multiplications we have

$$\begin{bmatrix} C(0) \\ C(4) \\ C(2) \\ C(6) \\ C(1) \\ C(5) \\ C(3) \\ C(7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & W^0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & W^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & W^0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & W^0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & W^1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & W^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}$$

which is in the form

C=A1M1A2M2A3XC=A1M1A2M2A3X" role="presentation" style="position:relative;" tabindex="0">

$$C = A_1M_1A_2M_2X$$

described by the index map. A_1, A_2, and A_3 each represents 8 additions, or, in general, N additions. M_1 and M_2 each represent 4 (or $N/2$) multiplications.

This is a very interesting result showing that implementing the DFT using the factored form requires considerably less arithmetic than the single factor definition. Indeed, the form of the formula that Co

Much of the theory of the FFT can be developed using operator factoring and it has some advantages for implementation of parallel and vector computer architectures. The eigenspace approach is somev

Contributor

- ContribEEBurrus

5.3: Algebraic Theory of Signal Processing Algorithms

A very general structure for all kinds of algorithms can be generalized from the approach of operators and operator decomposition. This is developed as “Algebraic Theory of Signal Processing" discussed in the module DFT and FFT - An Algebraic View by Püschel and others.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

6: Winograd's Short DFT Algorithms

In 1976, S. Winograd presented a new DFT algorithm which had significantly fewer multiplications than the Cooley-Tukey FFT which had been published eleven years earlier. This new Winograd Fourier Transform Algorithm (WFTA) is based on the type- one index map from Multidimensional Index Mapping with each of the relatively prime length short DFT's calculated by very efficient special algorithms. It is these short algorithms that this section will develop. They use the index permutation of Rader described in the another module to convert the prime length short DFT's into cyclic convolutions. Winograd developed a method for calculating digital convolution with the minimum number of multiplications. These optimal algorithms are based on the polynomial residue reduction techniques of Polynomial Description of Signals to break the convolution into multiple small ones.

Topic hierarchy

6.1: Introduction

Learning Objectives

- To study the Winograd Fourier Transform Algorithm (WFTA)

In 1976, S. Winograd presented a new DFT algorithm which had significantly fewer multiplications than the Cooley-Tukey FFT which had been published eleven years earlier. This new Winograd Fourier Transform Algorithm (WFTA) is based on the type- one index map from Multidimensional Index Mapping with each of the relatively prime length short DFT's calculated by very efficient special algorithms. It is these short algorithms that this section will develop. They use the index permutation of Rader described in the another module to convert the prime length short DFT's into cyclic convolutions. Winograd developed a method for calculating digital convolution with the minimum number of multiplications. These optimal algorithms are based on the polynomial residue reduction techniques of Polynomial Description of Signals to break the convolution into multiple small ones.

Contributor

- ContribEEBurrus

6.2: Winograd Fourier Transform Algorithm (WFTA)

The operation of discrete convolution defined by

$$y(n) = \sum_k h(n-k)x(k)$$

is called a **bilinear** operation because, for a fixed $h(n)$, $y(n)$ is a linear function of $x(n)$ and for a fixed $x(n)$ it is a linear function of $h(n)$. The operation of cyclic convolution is the same but with all indices evaluated modulo N.

Recall from Polynomial Description of Signals that length-N cyclic convolution of $x(n)$ and $h(n)$ can be represented by polynomial multiplication

$$Y(s) = X(s)H(s) \ mod \ (s^N - 1)$$

This bilinear operation can also be expressed in terms of linear matrix operators and a simpler bilinear operator denoted by o which may be only a simple element-by-element multiplication of the two vectors. This matrix formulation is

$$Y = C\left[AX_0 o BH\right]$$

where X, H and Y are length-N vectors with elements of $x(n)$, $h(n)$, and $y(n)$ respectively. The matrices A and B have dimension $M \times N$, and C is \[N\times M\) with $M \geq N$.

The elements of A, B, and C are constrained to be simple; typically small integers or rational numbers. It will be these matrix operators that do the equivalent of the residue reduction on the polynomials in the equation.

In order to derive a useful algorithm consider the polynomial formulation equation again. To use the residue reduction scheme, the modulus is factored into relatively prime factors. Fortunately the factoring of this particular polynomial, $s^N - 1$, has been extensively studied and it has considerable structure. When factored over the rationals, which means that the only coefficients allowed are rational numbers, the factors are called **cyclotomic polynomials**. The most interesting property for our purposes is that most of the coefficients of cyclotomic polynomials are zero and the others are plus or minus unity for degrees up to over one hundred. This means the residue reduction will generally require no multiplications.

The operations of reducing $X(s)$ and $H(s)$ in the equation are carried out by the matrices A and B in the equation. The convolution of the residue polynomials is carried out by the o operator and the recombination by the CRT is done by the C matrix. The important fact is that the A and B matrices usually contain only zero and plus or minus unity entries and the C matrix only contains rational numbers. The only general multiplications are those represented by o. Indeed, in the theoretical results from computational complexity theory, these real or complex multiplications are usually the only ones counted. In practical algorithms, the rational multiplications represented by C could be a limiting factor.

The $h(n)$ terms are fixed for a digital filter, or they represent the W terms from Multidimensional Index Mapping if the convolution is being used to calculate a DFT. Because of this, $d = BH$ in the equation can be precalculated and only the A and C operators represent the mathematics done at execution of the algorithm. In order to exploit this feature, it was shown that the properties of the equation allow the exchange of the more complicated operator C with the simpler operator B. Specifically this is given by

$$Y = C\left[AX_0 o BH\right]$$

$$Y' = B^T\left[AX_0 o C^T H'\right]$$

where H' has the same elements as H, but in a permuted order, and likewise Y' and Y. This very important property allows precomputing the more complicated $C^T H'$ in the equation rather than BH as in the above equation.

Because BH or $C^T H'$ can be precomputed, the bilinear form of the above equations can be written as a linear form. If an $M \times M$ diagonal matrix D is formed from $d = C^T H'$, or in the case of the equation, $d = BH$, assuming a commutative property for o, the equations become

$$Y' = B^T DAX$$

$$Y = CDAX$$

In most cases there is no reason not to use the same reduction operations on X and H, therefore, B can be the same as A and the equation then becomes

$$Y' = A^T DAX$$

In order to illustrate how the residue reduction is carried out and how the A matrix is obtained, the length-5 DFT algorithm started in The DFT as Convolution or Filtering will be continued. The DFT is first converted to a length-4 cyclic convolution by the index permutation from The DFT as Convolution or Filtering to give the cyclic convolution in The DFT as Convolution or Filtering. To avoid confusion from the permuted order of the data $x(n)$ in The DFT as Convolution or Filtering, the cyclic convolution will first be developed without the permutation, using the polynomial $U(s)$

$$U(s) = x(1) + x(3)s + x(4)s^2 + x(2)s^3$$

$$U(s) = u(0) + u(1)s + u(2)s^2 + u(3)s^3$$

and then the results will be converted back to the permuted $x(n)$. The length-4 cyclic convolution in terms of polynomials is

$$Y(s) = U(s)H(s) \ mod \ (s^4 - 1)$$

and the modulus factors into three cyclotomic polynomials

$$s^4 - 1 = (s^2 - 1)(s^2 + 1)$$

$$s^4 - 1 = (s - 1)(s + 1)(s^2 + 1)$$

$$s^4 - 1 = P_1 P_2 P_3$$

Both $U(s)$ and $H(s)$ are reduced modulo these three polynomials. The reduction modulo P_1 and P_2 is done in two stages. First it is done modulo $(s^2 - 1)$, then that residue is further reduced modulo $(s - 1)$ and $(s + 1)$.

$$U(s) = u_0 + u_1 s + u_2 s^2 + u_3 s^3$$

$$U'(s) = ((U(s)))_{(s^2-1)} = (u_0 + u_2) + (u_1 + u_3)s$$

$$U1(s) = ((U'(s)))_{P_1} = (u_0 + u_1 + u_2 + u_3)$$

$$U2(s) = ((U'(s)))_{P_2} = (u_0 - u_1 + u_2 - u_3)$$

$$U3(s) = ((U'(s)))_{P_3} = (u_0 - u_2) + (u_1 - u_3)s$$

The reduction in the equation of the data polynomial equation can be denoted by a matrix operation on a vector which has the data as entries.

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \end{bmatrix}$$

and the reduction in the equation is

$$\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u_0 - u_2 \\ u_1 - u_3 \end{bmatrix}$$

Combining the two equations gives one operator

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \\ u_0 - u_2 \\ u_1 - u_3 \end{bmatrix} = \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \\ u_0 - u_2 \\ u_1 - u_3 \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ v_0 \\ v_1 \end{bmatrix}$$

Further reduction of $v_0 + v_1 s$ is not possible because $P_3 = s^2 + 1$ cannot be factored over the rationals. However $s^2 - 1$ can be factored into $P_1 P_2 = (s-1)(s+1)$ and, therefore, $w_0 + w_1 s$ can be further reduced as was done in the above equations by

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = w_0 + w_1 = u_0 + u_2 + u_1 + u_3$$

Combining all the equations gives

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ v_0 \\ v_1 \end{bmatrix}$$

The same reduction is done to $H(s)$ and then the convolution of the equation is done by multiplying each residue polynomial of $X(s)$ and $H(s)$ modulo each corresponding cyclotomic factor of $P(s)$ and finally a recombination using the polynomial **Chinese Remainder Theorem** (CRT) as in Polynomial Description of Signals.

$$Y(s) = K_1(s)U_1(s)H_1(s) + K_2(s)U_2(s)H_2(s) + K_3(s)U_3(s)H_3(s) \ mod \ (s^4 - 1)$$

where $U_1(s) = r_1$ and $U_1(s) = r_1$ are constants and $U_3(s) = v_0 + v_1(s)$ is a first degree polynomial. U_1 times H_1 and U_2 times H_2 are easy, but multiplying U_3 times H_3 modulo $(s^2 + 1)$ is more difficult.

The multiplication of times $U_3(s)$ times $H_3(s)$ can be done by the Toom-Cook algorithm which can be viewed as Lagrange interpolation or polynomial multiplication modulo a special polynomial with three arbitrary coefficients. To simplify the arithmetic, the constants are chosen to be plus and minus one and zero.

For this example it can be verified that

$$((v0 + v1s)(h0 + h1s))_{s^2+1} = (v_0 h_0 - v_1 h_1) + (v_0 h_1 - v_1 h_0)s$$

which by the Toom-Cook algorithm or inspection is

$$\begin{bmatrix} 1 & -1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} o \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

where o signifies point-by-point multiplication. The total A matrix in is a combination of

$$AX = A_1 A_2 A_3 X$$

$$AX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ v_0 \\ v_1 \end{bmatrix}$$

where the matrix A_3 gives the residue reduction $(s^2 - 1)$ and $(s^2 + 1)$, the upper left-hand part of A_2 gives the reduction modulo $s - 1$ and $s + 1$, and the lower right-hand part of A_1 carries out the Toom-Cook algorithm modulo $s^2 + 1$ with the multiplication in the Y equation. Notice that by calculating the equation in the three stages, seven additions are required. Also notice that A_1 is not square. It is this "expansion" that causes more than N multiplications to be required in o or D in the Y equation. This staged reduction will derive the A operator.

The method described above is very straight-forward for the shorter DFT lengths. For Y, both of the residue polynomials are constants and the multiplication given by o in the equation is trivial. For $N = 5$, which is the example used here, there is one first degree polynomial multiplication required but the Toom-Cook algorithm uses simple constants and, therefore, works well as indicated in the equation. For $N = 7$, there are two first degree residue polynomials which can each be multiplied by the same techniques used in the $N = 5$ example. Unfortunately, for any longer lengths, the residue polynomials have an order of three or greater which causes the Toom-Cook algorithm to require constants of plus and minus two and worse. For that reason, the Toom-Cook method is not used, and other techniques such as index mapping are used that require more than the minimum number of multiplications, but do not require an excessive number of additions. The resulting algorithms still have the structure of the equation. Blahut and Nussbaumer have a good collection of algorithms for polynomial multiplication that can be used with the techniques discussed here to construct a wide variety of DFT algorithms.

The constants in the diagonal matrix D can be found from the CRT matrix C in the equation using $d = C^T H'$ for the diagonal terms in D. As mentioned above, for the smaller prime lengths of 3, 5, and 7 this works well but for longer lengths the CRT becomes very complicated. An alternate method for finding D uses the fact that since the linear form of the equations calculates the DFT, it is possible to calculate a known DFT of a given $x(n)$ from the definition of the DFT in Multidimensional Index Mapping and, given the A matrix in the equation, solve for D by solving a set of simultaneous equations.

A modification of this approach also works for a length which is an odd prime raised to some power: $N = P^M$. This is a bit more complicated but has been done for lengths of 9, and 25. For longer lengths, the conventional Cooley-Tukey type- two index map algorithm seems to be more efficient. For powers of two, there is no primitive root, and therefore, no simple conversion of the DFT into convolution. It is possible to use two generators to make the conversion and there exists a set of length 4, 8, and 16 DFT algorithms of the form in the equation.

In Table 6.2.1 below, an operation count of several short DFT algorithms is presented. These are practical algorithms that can be used alone or in conjunction with the index mapping to give longer DFT's as shown in The Prime Factor and Winograd Fourier Transform Algorithms. Most are optimized in having either the theoretical minimum number of multiplications or the minimum number of multiplications without requiring a very large number of additions. Some allow other reasonable trade-offs between numbers of multiplications and additions. There are two lists of the number of multiplications. The first is the number of actual floating point multiplications that must be done for that length DFT. Some of these (one or two in most cases) will be by rational constants and the others will be by irrational constants. The second list is the total number of multiplications given in the diagonal matrix D in the equation. At least one of these will be unity (the one associated with $X(0)$) and in some cases several will be unity (for $N = 2^M$). The second list is important in programming the WFTA in The Prime Factor and Winograd Fourier Transform Algorithms.

Length N	Mult Non-one	Mult Total	Adds
2	0	4	4
3	4	6	12
4	0	8	16

5	10	12	34
7	16	18	72
8	4	16	52
9	20	22	84
11	40	42	168
13	40	42	188
16	20	36	148
17	70	72	314
19	76	78	372
25	132	134	420
32	68	-	388

Table 6.2.1 Number of Real Multiplications and Additions for a Length-N DFT of Complex Data

Because of the structure of the short DFTs, the number of real multiplications required for the DFT of real data is exactly half that required for complex data. The number of real additions required is slightly less than half that required for complex data because $(N-1)$ of the additions needed when N is prime add a real to an imaginary, and that is not actually performed. When $N=2m$, there are $(N-2)$ of these pseudo additions.

The structure of these algorithms are in the form of X'=CDAXX'=CDAX" role="presentation" style="position:relative;" tabindex="0">

$$X' = CDAX \;\; or \;\; B^T DAX \;\; or \;\; A^T DAX$$

The A and B matrices are generally M by N with $M \geq N$ and have elements that are integers, generally 0 or ± 1. A pictorial description is given in the figures below:

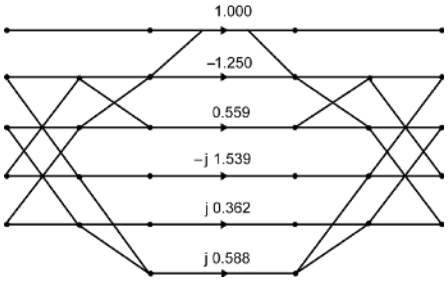


Fig. 6.2.1 Flow Graph for the Length-5 DFT

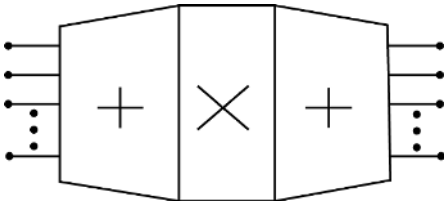


Fig. 6.2.2 Block Diagram of a Winograd Short DFT

The flow graph in Fig. 6.2.1 should be compared with the matrix description of the above equations, and with the programs and the appendices. The shape in Fig. 6.2.2 illustrates the expansion of the dat

An important characteristic of the D operator in the calculation of the DFT is its entries are either purely real or imaginary. The reduction of the W vector by

$$(s^{(N-1)/2}-1) \;\; and \;\; (s^{(N-1)/2}+1)$$

separates the real and the imaginary constants. The number of multiplications for complex data is only twice those necessary for real data, not four times.

Although this discussion has been on the calculation of the DFT, very similar results are true for the calculation of convolution and correlation, and these will be further developed in Algorithms for Data

Contributor

- ContribEEBurrus

6.3: The Bilinear Structure

The bilinear form introduced in earlier and the related linear form in are very powerful descriptions of both the DFT and convolution.

$$Bilinear: \quad Y = C\,[AXoBH]$$

$$Linear: \quad Y = CDAX$$

Since the equation is a bilinear operation defined in terms of a second bilinear operator o, this formulation can be nested. For example if o is itself defined in terms of a second bilinear operator @, , by

$$XoH = C'\,[A'X@B'H]$$

The equation then becomes

$$Y = CC'\,[A'AX@B'BH]$$

For convolution, if A, represents the polynomial residue reduction modulo the cyclotomic polynomials, then A, is square (e.g. the equation and o, represents multiplication of the residue polynomials modulo the cyclotomic polynomials. If A, represents the reduction modulo the cyclotomic polynomials plus the Toom-Cook reduction as was the case in the example of the equation, then A, is $N \times M$ and o, is term-by- term simple scalar multiplication. In this case AX, can be thought of as a transform of X and C is the inverse transform. This is called a rectangular transform because A is rectangular. The transform requires only additions and convolution is done with M multiplications. The other extreme is when A represents reduction over the N complex roots of $s^N - 1$. In this case A is the DFT itself, as in the example of (43), and o is point by point complex multiplication and C is the inverse DFT. A trivial case is where A, B and C are identity operators and o is the cyclic convolution.

This very general and flexible bilinear formulation coupled with the idea of nesting in the equation gives a description of most forms of convolution.

Contributor

- ContribEEBurrus

6.4: Winograd's Complexity Theorems

Because Winograd's work has been the foundation of the modern results in efficient convolution and DFT algorithms, it is worthwhile to look at his theoretical conclusions on optimal algorithms. Most of his results are stated in terms of polynomial multiplication as Polynomial Description of Signals. The measure of computational complexity is usually the number of multiplications, and only certain multiplications are counted. This must be understood in order not to misinterpret the results.

This section will simply give a statement of the pertinent results and will not attempt to derive or prove anything. A short interpretation of each theorem will be given to relate the result to the algorithms developed in this chapter. The indicated references should be consulted for background and detail.

Theorem 1

Given two polynomials, $x(s)$ and $h(s)$, of degree N and M respectively, each with indeterminate coefficients that are elements of a field H, $N+M+1$ multiplications are necessary to compute the coefficients of the product polynomial $x(s)h(s)$. Multiplication by elements of the field G(the field of constants), which is contained in H, are not counted and G contains at least $N+M$ distinct elements.

The upper bound in this theorem can be realized by choosing an arbitrary modulus polynomial $P(s)$ of degree $N+M+1$ composed of $N+M+1$ distinct linear polynomial factors with coefficients in G which, since its degree is greater than the product $x(s)h(s)$, has no effect on the product, and by reducing $x(s)$ and $h(s)$ to $N+M+1$ residues modulo the $N+M+1$ factors of $P(s)$. These residues are multiplied by each other, requiring $N+M+1$ multiplications, and the results recombined using the Chinese remainder theorem (CRT). The operations required in the reduction and recombination are not counted, while the residue multiplications are. Since the modulus $P(s)$ is arbitrary, its factors are chosen to be simple so as to make the reduction and CRT simple. Factors of zero, plus and minus unity, and infinity are the simplest. Plus and minus two and other factors complicate the actual calculations considerably, but the theorem does not take that into account. This algorithm is a form of the Toom-Cook algorithm and of Lagrange interpolation. For our applications, H is the field of reals and G the field of rationals.

Theorem 2

If an algorithm exists which computes $x(s)h(s)$ in $N+M+1$ multiplications, all but one of its multiplication steps must necessarily be of the form

$$mk = (gk' + xg(k))(gk'' + hg(k)) \quad for \quad k = 0, 1, \ldots, N+M$$

where g_k are distinct elements of G; and g'_k and g''_k are arbitrary elements of G.

This theorem states that the structure of an optimal algorithm is essentially unique although the factors of $P(s)$ may be chosen arbitrarily.

Theorem 3

Let $P(s)$ be a polynomial of degree N and be of the form $P(s) = Q(s)k$, where $Q(s)$ is an irreducible polynomial with coefficients in G and k is a positive integer. Let $x(s)$ and $h(s)$ be two polynomials of degree at least $N-1$ with coefficients from H, then $2N-1$ multiplications are required to compute the product $x(s)h(s)$ modulo $P(s)$.

This theorem is similar to Theorem 1 with the operations of the reduction of the product modulo $P(s)$ not being counted.

Theorem 4

Any algorithm that computes the product $x(s)h(s)$ modulo $P(s)$ according to the conditions stated in Theorem 3 and requires $2N-1$ multiplications will necessarily be of one of three structures, each of which has the form of Theorem 2 internally.

As in Theorem 2, this theorem states that only a limited number of possible structures exist for optimal algorithms.

Theorem 5

If the modulus polynomial $P(s)$ has degree N and is not irreducible, it can be written in a unique factored form

$$P(s) = P_1^{m_1}(s)P_2^{m_2}(s)\ldots P_k^{m_k}(s)$$

where each of the $P_i(s)$ are irreducible over the allowed coefficient field G. $2N-k$ multiplications are necessary to compute the product $x(s)h(s)$ modulo $P(s)$ where $x(s)$ and $h(s)$ have coefficients in H and are of degree at least $N-1$. All algorithms that calculate this product in $2N-k$ multiplications must be of a form where each of the k residue polynomials of $x(s)$ and $h(s)$ are separately multiplied modulo the factors of $P(s)$ via the CRT.

Corollary

If the modulus polynomial is $P(s)=s^N-1$ then $2N-t(N)$ multiplications are necessary to compute $x(s)h(s)$ modulo $P(s)$, where $t(N)$ is the number of positive divisors of N.

Theorem 5 is very general since it allows a general modulus polynomial. The proof of the upper bound involves reducing $x(s)$ and $h(s)$ modulo the k factors of $P(s)$. Each of the k irreducible residue polynomials is then multiplied using the method of Theorem 4 requiring $2Ni-1$ multiplies and the products are combined using the CRT. The total number of multiplies from the k parts is $2n-k$. The theorem also states the structure of these optimal algorithms is essentially unique. The special case of $P(s)=s^N-1$ is interesting since it corresponds to cyclic convolution and, as stated in the corollary, k is easily determined. The factors of s^N-1 are called cyclotomic polynomials and have interesting properties.

Theorem 6

Consider calculating the DFT of a prime length real-valued number sequence. If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length-P DFT is

$$u(DFT(N))=2P-3-t(P-1) \quad where \quad t(P-1) \; is \; the \; number \; of \; divisors \; of \; P-1$$

This theorem not only gives a lower limit on any practical prime length DFT algorithm, it also gives practical algorithms for $N=3,5,$ and 7. Consider the operation counts inthe Table to understand this theorem. In addition to the real multiplications counted by complexity theory, each optimal prime-length algorithm will have one multiplication by a rational constant. That constant corresponds to the residue modulo $(s-1)$ which always exists for the modulus $P(s)=s^N-1$. In a practical algorithm, this multiplication must be carried out, and that accounts for the difference in the prediction of Theorem 6 and count in the Table. In addition, there is another operation that for certain applications must be counted as a multiplication. That is the calculation of the zero frequency term $X(0)$ in the first row of the example in The DFT as Convolution or Filtering. For applications to the WFTA discussed in The Prime Factor and Winograd Fourier Transform Algorithms, that operation must be counted as a multiply. For lengths longer than 7, optimal algorithms require too many additions, so compromise structures are used.

Theorem 7

If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length-N DFT where N is a prime number raised to an integer power: $N=Pm$, is given by

$$u(DFT(N))=2N-((m2+m)/2)t(P-1)-m-1$$

where $t(P-1)$ is the number of divisors of $(P-1)$.

This result seems to be practically achievable only for $N=9$, or perhaps 25. In the case of $N=9$, there are two rational multiplies that must be carried out and are counted in the Table but are not predicted by Theorem 7. Experience indicates that even for $N=25$, an algorithm based on a Cooley-Tukey FFT using a type 2 index map gives an over-all more balanced result.

Theorem 8

If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length-N DFT where $N=2m$ is given by

$$u(DFT(N))=2N-m2-m-2$$

This result is not practically useful because the number of additions necessary to realize this minimum of multiplications becomes very large for lengths greater than 16. Nevertheless, it proves the minimum number of multiplications required of an optimal algorithm is a linear function of N rather than of $N\log N$ which is that required of practical algorithms. The best practical power-of-two algorithm seems to the Split-Radix FFT discussed in The Cooley-Tukey Fast Fourier Transform Algorithm.

All of these theorems use ideas based on residue reduction, multiplication of the residues, and then combination by the CRT. It is remarkable that this approach finds the minimum number of required multiplications by a constructive proof which generates an

algorithm that achieves this minimum; and the structure of the optimal algorithm is, within certain variations, unique. For shorter lengths, the optimal algorithms give practical programs. For longer lengths the uncounted operations involved with the multiplication of the higher degree residue polynomials become very large and impractical. In those cases, efficient suboptimal algorithms can be generated by using the same residue reduction as for the optimal case, but by using methods other than the Toom-Cook algorithm of Theorem 1 to multiply the residue polynomials.

Practical long DFT algorithms are produced by combining short prime length optimal DFT's with the Type 1 index map from Multidimensional Index Mapping to give the Prime Factor Algorithm (PFA) and the Winograd Fourier Transform Algorithm (WFTA) discussed in The Prime Factor and Winograd Fourier Transform Algorithms. It is interesting to note that the index mapping technique is useful inside the short DFT algorithms to replace the Toom-Cook algorithm and outside to combine the short DFT's to calculate long DFT's.

Contributor

- ContribEEBurrus

6.5: The Automatic Generation of Winograd's Short DFTs

Introduction

Efficient prime length DFTs are important for two reasons. A particular application may require a prime length DFT and secondly, the maximum length and the variety of lengths of a PFA or WFTA algorithm depend upon the availability of prime length modules.

This discusses automation of the process Winograd used for constructing prime length FFTs for $N < 7$ and that Johnson and Burrus extended to $N < 19$. It also describes a program that will design any prime length FFT in principle, and will also automatically generate the algorithm as a C program and draw the corresponding flow graph.

Winograd's approach uses Rader's method to convert a prime length DFT into a $P-1$ length cyclic convolution, polynomial residue reduction to decompose the problem into smaller convolutions, and the Toom-Cook algorithm. The Chinese Remainder Theorem (CRT) for polynomials is then used to recombine the shorter convolutions. Unfortunately, the design procedure derived directly from Winograd's theory becomes cumbersome for longer length DFTs, and this has often prevented the design of DFT programs for lengths greater than 19.

Here we use three methods to facilitate the construction of prime length FFT modules. First, the matrix exchange property is used so that the transpose of the reduction operator can be used rather than the more complicated CRT reconstruction operator. This is then combined with the numerical method for obtaining the multiplication coefficients rather than the direct use of the CRT. We also deviate from the Toom-Cook algorithm, because it requires too many additions for the lengths in which we are interested. Instead we use an iterated polynomial multiplication algorithm. We have incorporated these three ideas into a single structural procedure that automates the design of prime length FFTs.

Matrix Description

It is important that each step in the Winograd FFT can be described using matrices. By expressing cyclic convolution as a bilinear form, a compact form of prime length DFTs can be obtained.

If y is the cyclic convolution of h and x, then y can be expressed as

$$y = C\left[Ax. * Bh\right]$$

where, using the Matlab convention, .* represents point by point multiplication. When A, B and C are allowed to be complex, A and B are seen to be the DFT operator and C, the inverse DFT. When only real numbers are allowed, A, B and C will be rectangular. Using the matrix exchange property this form can be written as

$$y = RB^T\left[C^T Rh. * Ax\right]$$

where R is the permutation matrix that reverses order.

When h is fixed, as it is when considering prime length DFTs, the term $C^T Rh$ can be precomputed and a diagonal matrix D formed by

$$D = diagC^T Rh$$

This is advantageous because in general, C is more complicated than B, so the ability to "hide" C saves computation. Now y=RBTDAxy=RBTDAx" role="presentation" style="position:relative;" tabindex="0">

$$y = RB^T DAx \;\; or \;\; y = RA^T DAx$$

since A and B can be the same; they implement a polynomial reduction. The form $y = R^T DAxT$ can also be used for the prime length DFTs, it is only necessary to permute the entries of x and to ensu

$$DFT\left\{x\right\} = RA^T DAx$$

Johnson observes that by permuting the elements on the diagonal of D, the output can be permuted, so that the R matrix can be hidden in D, and

DFT{x}=ATDAxDFT{x}=ATDAx" role="presentation" style="position:relative;" tabindex="0">

$$DFT\left\{x\right\} = A^T DAx$$

From the knowledge of this form, once A is found, D can be found numerically.

Programming the Design Procedure

Because each of the above steps can be described by matrices, the development of a prime length FFTs is made convenient with the use of a matrix oriented programming language such as Matlab. After

Each matrix is a section of one stage of the flow graph that corresponds to the DFT program. The four stages are:

1. Permutation Stage: Permutes input and output sequence.
2. Reduction Stage: Reduces the cyclic convolution to smaller polynomial products.
3. Polynomial Product Stage: Performs the polynomial multiplications.
4. Multiplication Stage: Implements the point-by-point multiplication in the bilinear form.

Each of the stages can be clearly seen in the flow graphs for the DFTs. Fig. 6.5.1 shows the flow graph for a length 17 DFT algorithm that was automatically drawn by the program.

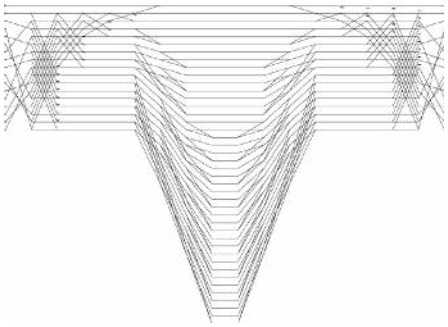


Fig. 6.5.1 Flowgraph of length-17 DFT

The programs that accomplish this process are written in Matlab and C. Those that compute the appropriate matrices are written in Matlab. These matrices are then stored as two ASCII files, with the dir

The Reduction Stage

The reduction of an N^{th} degree polynomial, $X(s)$, modulo the cyclotomic polynomial factors of $(s^N - 1)$ requires only additions for many N, however, the actual number of additions depends upon th

$$If \;\; N = 4 \;\; and \;\; \left((X(s))\right)_{s-1}, \;\; \left((X(s))\right)_{s+1} \;\; and \;\; \left((X(s))\right)_{s^2+1},$$

where the double parenthesis denote polynomial reduction modulo (s-1)(s-1)" role="presentation" style="position:relative;" tabindex="0">

$$(s-1),\ \ (s+1),\ \ and\ \ (s^2+1)$$

then in the first step

((X(s)))s2-1((X(s)))s2-1" role="presentation" style="position:relative;" tabindex="0">

$$((X(s)))_{s^2-1}\ \ and\ \ ((X(s)))_{s^2+1}$$

should be computed.

In the second step,

((Xs)))s-1((Xs)))s-1" role="presentation" style="position:relative;" tabindex="0">

$$((X(s)))_{s-1}\ \ and\ \ ((X(s)))_{s+1}$$

can be found by reducing ((X(s)))s2-1((X(s)))s2-1" role="presentation" style="position:relative;" tabindex="0">

$$((X(s)))_{s^2-1}$$

This process is described by the diagram in Fig. 6.5.2 below.

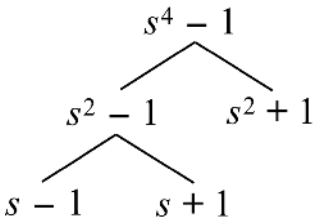


Fig. 6.5.2 Factorization of s^4-1 in steps

When N is even, the appropriate first factorization is

(SN/2-1)(sN/2+1)(SN/2-1)(sN/2+1)" role="presentation" style="position:relative;" tabindex="0">

$$(S^{N/2}-1)(S^{N/2}+1)$$

However, the next appropriate factorization is frequently less obvious. The following procedure has been found to generate a factorization in steps that coincides with the factorization that minimizes the

$$s^N-1=\prod_{n|N}C_n(s)$$

where $C_n(s)$ is the n^{th} cyclotomic polynomial.

We first introduce the following two functions defined on the positive integers,

$$\psi(N)=the\ smallest\ prime\ factor\ of\ N\ for\ N>1\ and\ \psi(1)=1$$

Suppose $P(s)$ is equal to either (s^N-1) or an intermediate noncyclotomic polynomial appearing in the factorization process, for example, (a^2-1), above. Write $P(s)$ in terms of its cyclotomic factors

(sN-1)(sN-1)" role="presentation" style="position:relative;" tabindex="0">

$$P(s)=C_{k_1}(s)C_{k_2}(s)\ldots C_{k_L}(s)$$

define the two sets, G and G, by

$$G=\{k_1,\ldots,k_L\}\ \ and\ G=\{k/gcd(G):k\epsilon G\}$$

and define the two integers, t and T, by

$$t=min\,\{\psi(k):k\,\epsilon\,G,\ k>1\}\ \ \ and\ \ T=max\ nu(k,t):k\,\epsilon\,G$$

Then form two new sets,

$$A=\{k\,\epsilon\,G:T\mid k\}\ \ \ and\ \ B=\{k\,\epsilon\,G:T\mid k\}$$

The factorization of $P(s)$,

$$P(s)=\left(\prod_{k\,\epsilon\,A}C_k(s)\right)\left(\prod_{k\,\epsilon\,B}C_k(s)\right)$$

has been found useful in the procedure for factoring (s^N-1). This is best illustrated with an example.

Example

$N=36$

Step 1:

Let

$$P(s)=s^{36}-1$$

Since

$$P=C_1C_2C_3C_4C_6C_9C_{12}C_{18}C_{36}$$

$$G=G=\{1,2,3,4,6,9,12,18,36\}$$

$$t=min\ \{2,3\}=2$$

$$A=\{k\,\epsilon\,G:4\mid k\}=\{1,2,3,6,9,18\}$$

$$B=\{k\,\epsilon\,G:4\mid k\}=\{4,12,36\}$$

Hence the factorization of $s^{36}-1$ into two intermediate polynomials is as expected,

$$\prod_{k\,\epsilon\,A} C_k(s) = s^{18} - 1, \quad \prod_{k\,\epsilon\,B} C_k(s) = s^{18} + 1$$

If a 36^{th} degree polynomial, $X(s)$, is represented by a vector of coefficients,

$$X = (x_{35}, \ldots x_0)' \; then \; ((X(s)))_{s^{18}-1} \; represented \; by \; X' \; and \; ((X(s)))_{s^{18}+1} \; represented \; by \; X''$$

is given by **test** which entails 36 additions.

Step 2:

This procedure is repeated with P(s)=s18-1P(s)=s18-1" role="presentation" style="position:relative;" tabindex="0">

$$P(s) = s^{18} - 1 \; and \; P(s) = s^{18} + 1$$

We will just show it for the later. Let P(s)=s18+1P(s)=s18+1" role="presentation" style="position:relative;" tabindex="0">

$$P(s) = s^{18} + 1$$

Since

$$P = C_4 C_{12} C_{36}$$

$$G = \{4, 12, 36\} \; and \; G' = \{1, 3, 9\}$$

$$t = min\, 3 = 3$$

$$T = max\, \nu(k, 3) : k\,\epsilon\, G = max\, 1, 3, 9 = 9$$

$$A = \{k\,\epsilon\, G : 9 \mid k\} = \{4, 12\}$$

$$B = \{k\,\epsilon\, G : 9 \mid k\} = \{36\}$$

This yields the two intermediate polynomials

$$s^6 + 1 \;\; and \;\; s^{12} - s^6 + 1$$

In the notation used above,

$$[\, X' \quad X'' \,] = \begin{bmatrix} I_6 & -I_6 & I_6 \\ I_6 & I_6 & \\ -I_6 & & I_6 \end{bmatrix} X$$

entailing 24 additions. Continuing this process results in a factorization in steps.

In order to see the number of additions this scheme uses for numbers of the form $N = P - 1$ (which is relevant to prime length FFT algorithms).

The Polynomial Product Stage

The iterated convolution algorithm can be used to construct an N point linear convolution algorithm from shorter linear convolution algorithms. Suppose the linear convolution y, of the n point vectors x

$$y = E_n^T D E_n x$$

where E_n is an “expansion" matrix the elements of which are $\pm$1's and 0's and D is an appropriate diagonal matrix. Because the only multiplications in this expression are by the elements of D, the num

Given a matrix E_{n1} and a matrix E_{n2}, the iterated algorithm gives a method for combining E_{n1} and E_{n2} to construct a valid expansion matrix, E_n, for

$$N \leq n_1 n_2$$

Specifically,

$$E_{n_1,n_2} = (I_{m(n_2)} \otimes E_{n_1})(E_{n_2} \times I_{n_1})$$

The product $n_1 n_2$ may be greater than N, for zeros can be (conceptually) appended to x. The operation count associated with $E_{n1} E_{n2}$ is

$$A(n_1, n_2) = n! A(n_2) + A(n_1) M(n_2)$$

$$M(n_1, n_2) = M(n_1) M(n_2)$$

Although they are both valid expansion matrices,

$$E_{n_1,n_2} \neq E_{n_2,n_1} \;\; and \;\; A_{n_1,n_2} \neq A_{n_2,n_1}$$

Because Mn1,n2≠Mn2,n1Mn1,n2≠Mn2,n1" role="presentation" style="position:relative;" tabindex="0">

$$M_{n_1,n_2} \neq M_{n_2,n_1}$$

it is desirable to chose an ordering of factors to minimize the additions incurred by the expansion matrix.

Multiple Factors

Note that a valid expansion matrix, E_n, can be constructed from E_{n_1,n_2} and E_{n3}, for

N≤n1n2n3N≤n1n2n3" role="presentation" style="position:relative;" tabindex="0">

$$N \leq n_1 n_2 n_3$$

In general, any number of factors can be used to create larger expansion matrices. The operation count associated with is E_{n_1,n_2,n_3} is

$$A(n_1, n_2, n_3) = n_1 n_2 A(n_3) + n_1 A(n_2) M(n_3) + A(n_1) M(n_2) M(n_3)$$

$$M(n_1, n_2, n_3) = M(n_1) M(n_2) M(n_3)$$

These equations generalize in the predicted way when more factors are considered. Because the ordering of the factors is relevant in the equation for $A(\cdot)$ but not for $M(\cdot)$, it is again desirable to order t

Reservation of Optimal Ordering

Suppose

$$A(n_1, n_2, n_3) \leq \min\{A(n_{k_1}, n_{k_2}, n_{k_3}) \; k_1, k_2, k_3 \in \{1, 2, 3\} \text{ and distinct}\}, \text{ then}$$

1.	$A(n_1, n_2) \leq A(n_2, n_1)$

2.	$A(n_2, n_3) \leq A(n_3, n_2)$
3.	$A(n_1, n_3) \leq A(n_3, n_1)$

The generalization of this property to more than two factos revelas that an optimal ordering of $\{n_1, \ldots, n_{L-i}\}$ is preserved in an optimal ordering of $\{n_1, \ldots, n_L\}$. Therefore, if $(n_1, \ldots, n_{L-i})$ is an op

$$\frac{A(n_k)}{M(n_k) - n_k} \leq \frac{A(n_{k+1})}{M(n_{k+1}) - n_{k+1}}$$

for all $k = 1, 2, \ldots, L-1$.

This immediately suggests that an optimal ordering of $\{n_1, \ldots, n_L\}$, to minimize the number of additions incurred by $E_{n_1, \ldots, n_L}$ simply involves computing the appropriate ratios.

Discussion and Conclusion

We have designed prime length FFTs up to length 53 that are as good as the previous designs that only went up to 19. Table 1 gives the operation counts for the new and previously designed modules, as

It is interesting to note that the operation counts depend on the factorability of $P-1$. The primes 11, 23, and 47 are all of the form $1+2P_1$ making the design of efficient FFTs for these lengths more d

Further deviations from the original Winograd approach than we have made could prove useful for longer lengths. We investigated, for example, the use of twiddle factors at appropriate points in the dec

N	Mult	Adds
7	16	72
11	40	168
13	40	188
17	82	274
19	88	360
23	174	672
29	190	766
31	160	984
37	220	920
41	282	1140
43	304	1416
47	640	2088
53	556	2038

Operation counts for prime length DFTs

The approach in writing a program that writes another program is a valuable one for several reasons. Programming the design process for the design of prime length FFTs has the advantages of being pra

More details on the generation of programs for prime length FFTs can be found in the 1993 Technical Report.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

7: DFT and FFT - An Algebraic View

Topic hierarchy

7.1: Introduction

Learning Objectives

- To study the algebraic description of the DFT and on the algebraic derivation of the general-radix Cooley-Tukey FFT from Factoring the Signal Processing Operators.

by Markus Pueschel, Carnegie Mellon University

In infinite, or non-periodic, discrete-time signal processing, there is a strong connection between the z-transform, Laurent series, convolution, and the discrete-time Fourier transform (DTFT). As one may expect, a similar connection exists for the DFT but bears surprises. Namely, it turns out that the proper framework for the DFT requires modulo operations of polynomials, which means working with so-called polynomial algebras. Associated with polynomial algebras is the Chinese remainder theorem, which describes the DFT algebraically and can be used as a tool to concisely derive various FFTs as well as convolution algorithms (see also Winograd's Short DFT Algorithms). The polynomial algebra framework was fully developed for signal processing as part of the algebraic signal processing theory (ASP). ASP identifies the structure underlying many transforms used in signal processing, provides deep insight into their properties, and enables the derivation of their fast algorithms. Here we focus on the algebraic description of the DFT and on the algebraic derivation of the general-radix Cooley-Tukey FFT from Factoring the Signal Processing Operators. The derivation will make use of and extend the Polynomial Description of Signals. We start with motivating the appearance of modulo operations.

The z-transform associates with infinite discrete signals $X + (\ldots, x(-1), x(0), x(1), \ldots)$ a Laurent series:

$$X \mapsto X(s) = \sum_{n \in \mathbb{Z}} x(n) s^n$$

Here we used $s = z^{-1}$ to simplify the notation in the following. The DTFT of X is the evaluation of $X(s)$ on the unit circle

$$X(e^{-jw}), -\pi < \omega \leq \pi$$

Finally, filtering or (linear) convolution is simply the multiplication of Laurent series,

$$X \mapsto X(s) = \sum_{n \in \mathbb{Z}}^{N-1} x(n) s^n$$

and that the DFT is an evaluation of these polynomials. Indeed, the definition of the DFT in Winograd's Short DFT Algorithms shows that

$$C(k) = X(W_N^k) = X\left(e^{-j\frac{2\pi k}{N}}\right), \ 0 \leq k < N$$

i.e., the DFT computes the evaluations of the polynomial $X(s)$ at the n^{th} roots of unity.
The problem arises with the equivalent of Equation, since the multiplication $H(s)X(s)$ of two polynomials of degree $N-1$ yields one of degree $2N-2$. Also, it does not coincide with the circular convolution known to be associated with the DFT. The solution to both problems is to reduce the product modulo $s^n - 1$:

$$H *_{\text{circ}} X \leftrightarrow H(s)X(s) \bmod (s^n - 1)$$

Concept	Infinite Time	Finite Time
Signal	$X(s) = \sum_{n \in z} x(n) s^n$	$\sum_{n=0}^{N-1} x(n) s^n$
Filter	$H(s) = \sum_{n \in z} h(n) s^n$	$\sum_{n=0}^{N-1} h(n) s^n$
Convolution	$H(s)X(s)$	$H(s)X(s) \bmod (s^n - 1)$
Fourier transform	DTFT: $X(e^{-j\omega}), -\pi < \omega \leq \pi$	DFT: $X\left(e^{-j\frac{2\pi k}{n}}\right), 0 \leq k < n$

The resulting polynomial then has again degree $N-1$ and this form of convolution becomes equivalent to circular convolution of the polynomial coefficients. We also observe that the evaluation points in Equation are precisely the roots of s^n-1. This connection will become clear in this chapter.

The discussion is summarized in Table.

The proper framework to describe the multiplication of polynomials modulo a fixed polynomial are polynomial algebras. Together with the Chinese remainder theorem, they provide the theoretical underpinning for the DFT and the Cooley-Tukey FFT.

In this chapter, the DFT will naturally arise as a linear mapping with respect to chosen bases, i.e., as a matrix. Indeed, the definition shows that if all input and outputs are collected into vectors $X=(X(0),\ldots,X(N-1))$ and $C=(C(0),\ldots C(N-1))$, then Winograd's Short DFT Algorithms is equivalent to

$$C=DFT_N\,X,$$

where

$$DFT_N=\left[W_N^{kn}\right]_{0\le k,n<N}.$$

The matrix point of view is adopted in the FFT books.

Contributor

- ContribEEBurrus

7.2: Polynomial Algebras and the DFT

In this section we introduce polynomial algebras and explain how they are associated to transforms. Then we identify this connection for the DFT. Later we use polynomial algebras to derive the Cooley-Tukey FFT.

Polynomial Algebra

An algebra $\mathbb{A}$ is a vector space that also provides a multiplication of its elements such that the distributivity law holds (see link for a complete definition). Exampels include the sets off complex or real numbers $\mathbb{C}$ or $\mathbb{R}$, and the sets of complex or real polynomials in the variable s : $\mathbb{C}[s]$ or $\mathbb{R}[s]$.

The key player in this chapter is the **polynomial algebra**. Given a fixed polynomial $P(s)$ of degree $\deg(P)=N$, we define a polynomial algebra as the set

$$\mathbb{C}[s]/P(s)=\{X(s)\,|\,\deg(X)<\deg(P)\}$$

of polynomials of degree smaller than N with addition and multiplication modulo P. Viewed as a vector space, $\mathbb{C}[s]/P(s)$ hence has dimension N.

Every polynomial $X(s)\in\mathbb{C}[s]$ is reduced to a unique polynomial $R(s)$ modulo $P(s)$ of degree smaller than N. $R(s)$ is computed using division with rest, namely

$$X(s)=Q(s)P(s)+R(s),\ \deg(R)<\deg(P)$$

Regarding this equation modulo P, $P(s)$ becomes zero, and we get

$$X(s)\equiv R(s) \bmod P(s)$$

We read this equation as "$X(s)$ is congruent (or equal) $R(s)$ modulo $P(s)$." We will also write $X(s) \bmod P(s)$ to denote that $X(s)$ is reduced modulo $P(s)$. Obviously,

$$P(s)\equiv 0 \bmod P(s)$$

As a simple example we consider A=C[s]/(s2-1)A=C[s]/(s2-1)" role="presentation" style="position:relative;" tabindex="0">

$$\mathbb{A}=\mathbb{C}[s]/(s^2-1)$$

which has dimension 2. A possible basis is b=(1,s)b=(1,s)" role="presentation" style="position:relative;" tabindex="0">$b=(1,s)$. In $\mathbb{A}$, for example,

$$s\cdot(s+1)=s^2+s\equiv s+1 \bmod (s^2-1)$$

obtained through division with rest

$$s^2+s=1\cdot(s^2-1)+(s+1)$$

or simply by replacing s^2 with 1 (since $(s^2-1)=0$ implies $s^2=1$).

Chinese Remainder Theorem (CRT)

Assume $P(S)=Q(s)R(s)$ factors into two coprime (no common factors) polynomials Q and R. Then the Chinese remainder theorem (CRT) for polynomials is the linear mapping, more precisely, isor

$$\Delta:\mathbb{C}[s]/P(s)\to\mathbb{C}[s]/Q(s)\oplus\mathbb{C}[s]/R(s)$$

$$X(s)\mapsto(X(s) \bmod Q(s), X(s) \bmod R(s))$$

Here, $\oplus$ is the Cartesian product of vector spaces with elementwise operation (also called outer direct sum). In words, the CRT asserts that computing (addition, multiplication, scalar multiplication) in $\mathbb{C}$

If we choose bases b, c, d in the three polynomial algebras, then Δ can be expressed as a matrix. As usual with linear mapping, this matrix is obtained by mapping every element of b with Δ, expressing

As an example, we consider again the polynomial P(s)=s2-1=(s-1)(s+1)P(s)=s2-1=(s-1)(s+1)" role="presentation" style="position:relative;" tabindex="0">

$$P(s)=s^2-1=(s-1)(s+1)$$

and the CRT decomposition

$$\Delta:\mathbb{C}[s]/(s^2-1)\to\mathbb{C}[s]/(s-1)\oplus\mathbb{C}[s]/(s+1)$$

As bases, we choose $b=(1,x), c=(1), d=(1)$. $\Delta(1)=(1,1)$ with the same coordinate vector in $c\cup d=(1,1)$. Further, because of $x\equiv 1 \bmod (x-1)$ and $x\equiv -1 \bmod (x+1)$, $\Delta(x)=(x,x)\equiv$

$$DFT_2=\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

Polynomial Transforms

Assume $P(s)\in\mathbb{C}[s]$ has pairwise distinct zeros α=(α0,⋯,αN-1)α=(α0,⋯,αN-1)" role="presentation" style="position:relative;" tabindex="0">$\alpha=\alpha_0,\ldots,\alpha_N-1$. The

$$\Delta:\mathbb{C}[s]/P(s)\to\mathbb{C}[s]/(s-\alpha_0)\oplus\ldots\oplus\mathbb{C}[s]/(s-\alpha_{N-1})$$

$$X(s)\mapsto(X(s) \bmod (s-\alpha_0),\ldots,X(s) \bmod (s-\alpha_{N-1}))=(s(\alpha_0),\ldots,s(\alpha_{N-1}))$$

If we choose a basis b=(P0(s),⋯,PN-1(s))b=(P0(s),⋯,PN-1(s))" role="presentation" style="position:relative;" tabindex="0">$b=(P_0(s),\ldots,P_{N-1}(s))$ in C[s]/P(s)C[s]/P(s)" role="presentatior

$$\mathfrak{P}_{b,\alpha}=[P_n(\alpha_k)]_{0\le k,n<N}$$

and is called the **polynomial transform** for A=C[s]/P(s)A=C[s]/P(s)" role="presentation" style="position:relative;" tabindex="0">$\mathfrak{A}=\mathbb{C}[s]/P(s)$ with basis b

If, in general, we choose bi=(βi)bi=(βi)" role="presentation" style="position:relative;" tabindex="0">$b_i=(\beta_i)$ as spectral basis, then the matrix corresponding to the decomposition equation is t

$$\mathrm{diag}_{0\le k<N}(1/\beta_n)\mathfrak{P}_{b,\alpha}$$

where $\mathrm{diag}_{0\le k<N}(\gamma_n)$ denotes a diagonal matrix with diagonal entries γnγn" role="presentation" style="position:relative;" tabindex="0">γ_n.

We jointly refer to polynomial transforms, scaled or not, as Fourier transforms.

DFT as a Polynomial Transform

We show that the DFTNDFTN" role="presentation" style="position:relative;" tabindex="0">DFT_N is a polynomial transform for A=C[s]/(sN-1)A=C[s]/(sN-1)" role="presentation" style="position:rela

$$s^{N-1}=\prod_{0\le k<N}(x-W_N^k)$$

which means that Δ takes the form

$$\Delta:\mathbb{C}[s]/(s^N-1)\to\mathbb{C}[s]/(s-W_N^0)\oplus\ldots\oplus\mathbb{C}[s]/(s-W_N^{N-1})$$

$$X(s)\mapsto(X(s) \bmod (s-W_N^0),\ldots,X(s) \bmod (s-W_N^{N-1}))=(X(W_N^0),\ldots,X(W_N^{N-1}))$$

The associated polynomial transform hence becomes

$$\mathfrak{P}_{b,\alpha} = \left[W_N^{kn}\right]_{0\leq k,n<N} = \mathrm{DFT}_N$$

This interpretation of the DFT has been known for a long time and clarifies the connection between the evaluation points in and the circular convolution in the equations.

DFTs of types $1-4$ are defined, with type 1 being the standard DFT. In the algebraic framework, type 3 is obtained by choosing A=C[s]/(sN+1)A=C[s]/(sN+1)" role="presentation" style="position:rela

$$\mathfrak{P}_{b,\alpha} = \left[W_N^{(k+1/2)n}\right]_{0\leq k,n<N} = \mathrm{DFT}-3_N$$

The DFTs of type 2 and 4 are scaled polynomial transforms.

Contributor

- ContribEEBurrus

7.3: Algebraic Derivation of the Cooley-Tukey FFT

Knowing the polynomial algebra underlying the DFT enables us to derive the Cooley-Tukey FFT **algebraically**. This means that instead of manipulating the DFT definition, we manipulate the polynomial algebra C[s]/(sN-1)C[s]/(sN-1)" role="presentation" style="position:relative;" tabindex="0">$\mathbb{C}[s]/(s^N-1)$. The basic idea is intuitive. We showed that the DFT is the matrix representation of the complete dec

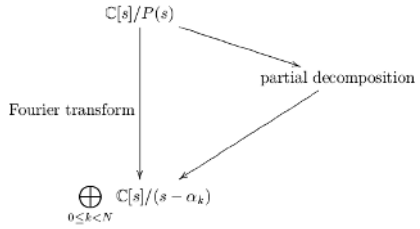


Fig. 7.3.1 Basic idea behind the algebraic derivation of Cooley-Tukey type algorithms

This stepwise decomposition can be formulated generically for polynomial transforms. Here, we consider only the DFT. We first introduce the matrix notation we will use and in particular the Kroneckei

Matrix Notation

We denote the N×NN×N" role="presentation" style="position:relative;" tabindex="0">$N \times N$ identity matrix with ININ" role="presentation" style="position:relative;" tabindex="0">I_N, and dia

$$\mathrm{diag}_{0\le k<N}(\gamma_k) = \begin{bmatrix} \gamma_0 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \gamma_{N-1} \end{bmatrix}$$

The N×NN×N" role="presentation" style="position:relative;" tabindex="0">$N \times N$ **stride permutation** matrix is defined for N=KMN=KM" role="presentation" style="position:relative;" tabind

$$L^N_M = iK + j \mapsto jM + i$$

for $0 \le i < K,\ 0 \le j < M$. This definition shows that LMNLMN" role="presentation" style="position:relative;" tabindex="0">L^N_M transposes a $K \times M$ matrix stored in row-major order. Alternativel

$$L^N_M : i \mapsto \mathrm{iM} \bmod N-1,\ \text{for } 0 \le i < N-1,\ N-1 \mapsto N-1$$

For example (·means 0),

$$L^6_2 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

$L^N_{N/2}$ is sometimes called the perfect shuffle.

Further, we use matrix operators; namely the direct sum

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

and the Kronecker or tensor product

$$A \otimes B = [a_{k,l}B]_{k,l},\ \text{for } A = [a_{k,l}]$$

In particular,

$$I_n \otimes A = A \oplus \ldots \oplus A = \begin{bmatrix} A & & & \\ & \ddots & & \\ & & \ddots & \\ & & & A \end{bmatrix}$$

is block-diagonal.

We may also construct a larger matrix as a matrix of matrices, e.g.,

$$\begin{bmatrix} A & B \\ B & A \end{bmatrix}$$

If an algorithm for a transform is given as a product of sparse matrices built from the constructs above, then an algorithm for the transpose or inverse of the transform can be readily derived using mather

$$(AB)^T = B^T A^T,\quad (AB)^{-1} = B^{-1}A^{-1}$$
$$(A \oplus B)^T = A^T \oplus B^T,\quad (A \oplus B)^{-1} = A^{-1} \oplus B^{-1}$$
$$(A \otimes B)^T = A^T \otimes B^T,\quad (A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$$

Permutation matrices are orthogonal, i.e., $P^T = P^{-1}$. The transposition or inversion of diagonal matrices is obvious.

Radix-2 FFT

The DFT decomposes $\mathfrak{A} = \mathbb{C}[s]/(s^N-1)$ with basis b=(1,s,⋯,sN-1)b=(1,s,⋯,sN-1)" role="presentation" style="position:relative;" tabindex="0">$b = 1, s, \ldots, s^{N-1}$ as shown in the equation.

$$s^{2M} - 1 = (s^M - 1)(s^M + 1)$$

factors and we can apply the CRT in the following steps:

$$\mathbb{C}[s]/(s^N-1)$$
$$\to \mathbb{C}[s]/(s^M-1) \oplus \mathbb{C}[s]/(s^M+1)$$
$$\to \bigoplus_{0\le i<M} \mathbb{C}[s]/(x-W_N^{2i}) \oplus \bigoplus_{0\le i<M} \mathbb{C}[s]/(x-W_M^{2i+1})$$
$$\to \bigoplus_{0\le i<N} \mathbb{C}[s]/(x-W_N^{i})$$

As bases in the smaller algebras $\mathbb{C}[s]/(s^M-1)$ and $\mathbb{C}[s]/(s^M+1)$, we choose $c=d=1,s,\ldots,s^{M-1}$. The derivation of an algorithm for DFTNDFTN" role="presentation" style="position:relative;"
First, we derive the base change matrix BB" role="presentation" style="position:relative;" tabindex="0">B corresponding to the equation. To do so, we have to express the base elements sn∈bs

$$B=\begin{bmatrix} I_M & * \\ I_M & * \end{bmatrix}$$

where the entries $*$ are determined next. For the base elements $s^{M+n},\ 0\le n<M$, we have

$$s^{M+n} = s^n \bmod (s^M-1)$$
$$s^{M+n} = -s^n \bmod (s^M+1)$$

which yields the final result

$$B=\begin{bmatrix} I_M & I_M \\ I_M & -I_M \end{bmatrix} = \mathrm{DFT}_2 \otimes I_M$$

Next, we consider step equation. $\mathbb{C}[s]/(s^M-1)$ is decomposed by DFTMDFTM" role="presentation" style="position:relative;" tabindex="0">DFT_M and $\mathbb{C}[s]/(s^M+1)$ by $\mathrm{DFT}-3_M$ in the equatic
Finally, the permutation in step in the equation is the perfect shuffle LMNLMN" role="presentation" style="position:relative;" tabindex="0">L_M^N, which interleaves the even and odd spectral component
The final algorithm obtained is

$$\mathrm{DFT}_{2M} = L_M^N(\mathrm{DFT}_M \oplus \mathrm{DFT}-3_M)(\mathrm{DFT}_2 \otimes I_M)$$
$$\mathrm{DFT}_{2M} = L_M^N(I_2 \otimes \mathrm{DFT}_M)(I_M \oplus D_M)(\mathrm{DFT}_2 \otimes I_M)$$

The last expression is the radix-2 decimation-in-frequency Cooley-Tukey FFT. The corresponding decimation-in-time version is obtained by transposition using the equation and the symmetry of the DF

$$\mathrm{DFT}_{2M} = (\mathrm{DFT}_2 \otimes I_M))(I_M \oplus D_M)(I_2 \otimes \mathrm{DFT}_M)L_2^N$$

The entries of the diagonal matrix IM⊕DMIM⊕DM" role="presentation" style="position:relative;" tabindex="0">$I_M \oplus D_M$ are commonly called **twiddle factors**.

General-radix FFT

To algebraically derive the general-radix FFT, we use the **decomposition property** of sN-1sN-1" role="presentation" style="position:relative;" tabindex="0">s^N-1 . Namely, if N=KMN=KM" role="p

$$s^N-1 = (s^M)^K-1$$

Decomposition means that the polynomial is written as the composition of two polynomials: here, s^M is inserted into s^K-1 . Note that this is a special property: most polynomials do not decompose.
Based on this polynomial decomposition, we obtain the following stepwise decomposition of $\mathbb{C}[s]/(s^N-1)$, which is more general than the previous one in the equations. The basic idea is to first deco

$$\mathbb{C}[s]/(s^N-1) = \mathbb{C}[x]/\left((s^M)^K-1\right)$$
$$\to \bigoplus_{0\le i<K} \mathbb{C}[s]/(s^M-W_K^i)$$
$$\to \bigoplus_{0\le i<K} \bigoplus_{0\le j<M} \mathbb{C}[s]/(x-W_N^{jK+i})$$
$$\to \bigoplus_{0\le i<N} \mathbb{C}[s]/(x-W_N^{i})$$

As bases in the smaller algebras $\mathbb{C}[s]/(s^M-W_K^i)$ we choose ci=(1,s,⋯,sM-1)ci=(1,s,⋯,sM-1)" role="presentation" style="position:relative;" tabindex="0">$c_i=1,s,\ldots,s^{M-1}$. As before, th
The first decomposition step requires us to compute $s^n \bmod \left(s^M-W_K^i\right),\ 0\le n<N$. To do so, we decompose the index n as $n=lM+m$ and compute

$$s^n = s^{lM+m} = (s^M)^l s^m \equiv W_k^{lm} s^m \bmod (s^M-W_K^i)$$

This shows that the matrix for is given by: $\mathrm{DFT}_K \otimes I_M$
In step equation, each $\mathbb{C}[s]/(s^M-W_K^i)$ is completely decomposed by its polynomial transform

$$\mathrm{DFT}_M(i,K) = \mathrm{DFT}_M.\,\mathrm{diag}_{0\le i<M}(W_N^{ij})$$

At this point, $\mathbb{C}[s]/(s^N-1)$ is completely decomposed, but the spectrum is ordered according to jK+ijK+i" role="presentation" style="position:relative;" tabindex="0">$jK+i,\ 0\le i<M,\ 0\le j<K$
Thus, in step equation, we need to apply the permutation $jK+i \mapsto iM+j$, which is exactly the stride permutation LMNLMN" role="presentation" style="position:relative;" tabindex="0">L_M^N in the
In summary, we obtain the Cooley-Tukey decimation-in-frequency FFT with arbitrary radix:

$$L_M^N\left(\bigoplus_{0\le i<K} \mathrm{DFT}_M.\,\mathrm{diag}_{0\le i<M}\left(W_N^{ij}\right)\right)(\mathrm{DFT}_k \otimes I_M)$$
$$= L_M^N(I_K \otimes \mathrm{DFT}_M)T_M^N(\mathrm{DFT}_k \otimes I_M)$$

The matrix TMNTMN" role="presentation" style="position:relative;" tabindex="0">T_M^N is diagonal and usually called the **twiddle matrix**. Transposition using the equation yields the corresponding dec

$$(\mathrm{DFT}_k \otimes I_M)T_M^N(I_K \otimes \mathrm{DFT}_M)L_K^N$$

Contributor

- ContribEEBurrus

7.4: Discussion and Further Reading

This chapter only scratches the surface of the connection between algebra and the DFT or signal processing in general. We provide a few references for further reading.

Algebraic Derivation of Transform Algorithms

As mentioned before, the use of polynomial algebras and the CRT underlies much of the early work on FFTs and convolution algorithms. For example, Winograd's work on FFTs minimizes the number of non-rational multiplications. This and his work on complexity theory in general makes heavy use of polynomial algebras (see Chapter Winograd's Short DFT Algorithms for more information and references).

Since $\mathbb{C}[x]/(s^N-1) = \mathbb{C}[C_N]$ can be viewed a group algebra for the cyclic group, the methods shown in this chapter can be translated into the context of group representation theory. However, Fourier transforms for groups have found only sporadic applications. Along a related line of work, using group theory it is possible that to discover and generate certain algorithms for trigonometric transforms, such as discrete cosine transforms (DCTs), automatically using a computer program.

More recently, the polynomial algebra framework was extended to include most trigonometric transforms used in signal processing, besides the DFT, the discrete cosine and sine transforms and various real DFTs including the discrete Hartley transform. It turns out that the same techniques shown in this chapter can then be applied to derive, explain, and classify most of the known algorithms for these transforms and even obtain a large class of new algorithms including general-radix algorithms for the discrete cosine and sine transforms (DCTs/DSTs).

This latter line of work is part of the algebraic signal processing theory briefly discussed next.

Algebraic Signal Processing Theory

The algebraic properties of transforms used in the above work on algorithm derivation hints at a connection between algebra and (linear) signal processing itself. This is indeed the case and was fully developed in a recent body of work called algebraic signal processing theory (ASP).

ASP first identifies the algebraic structure of (linear) signal processing: the common assumptions on available operations for filters and signals make the set of filters an **algebra**AA" role="presentation" style="position:relative;" tabindex="0"> $\mathfrak{A}$ and the set of signals an associated AA" role="presentation" style="position:relative;" tabindex="0">$\mathfrak{A}$-module AA" role="p

Signal model	Infinite time	Finite time
AA" role="presentation" style="position:relative;" tabindex="0"> $\mathfrak{A}$	$\left\{ \sum_{n\in\mathbb{Z}} H(n)s^n \mid (\dots, H(-1), H(0), H(1), \dots) \in l^1(\mathbb{Z}) \right\}$	$\mathbb{C}[x]/(s^n-1)$
$\mathfrak{M}$	$\left\{ \sum_{n\in\mathbb{Z}} X(n)s^n \mid (\dots, X(-1), X(0), X(1), \dots) \in l^2(\mathbb{Z}) \right\}$	$\mathbb{C}[s]/(s^n-1)$
Φ	$\Phi : l^2(\mathbb{Z}) \to \mathfrak{M}$	$\Phi : \mathbb{C}^n \to \mathfrak{M}$

Table 7.4.1 Infinite and finite time models as defined in ASP

ASP shows that many signal models are in principle possible, each with its own notion of filtering and Fourier transform. Those that support shift-invariance have commutative algebras. Since finite-dim

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

8: The Cooley-Tukey Fast Fourier Transform Algorithm

The publication by Cooley and Tukey in 1965 of an efficient algorithm for the calculation of the DFT was a major turning point in the development of digital signal processing. During the five or so years that followed, various extensions and modifications were made to the original algorithm. By the early 1970's the practical programs were basically in the form used today. The standard development shows how the DFT of a length-N sequence can be simply calculated from the two length-N/2 DFT's of the even index terms and the odd index terms. This is then applied to the two half-length DFT's to give four quarter-length DFT's, and repeated until N scalars are left which are the DFT values. Because of alternately taking the even and odd index terms, two forms of the resulting programs are called decimation-in-time and decimation-in-frequency. For a length of 2^M, the dividing process is repeated $M = \log_2 N$ times and requires N multiplications each time. This gives the famous formula for the computational complexity of the FFT of $N \log_2 N$ which was derived in Multidimensional Index Mapping.

Topic hierarchy

CHAPTER OVERVIEW

Front Matter

Rice University

8: The Cooley-Tukey Fast Fourier Transform Algorithm

C. Sidney Burrus

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide and accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.

LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 07/13/2025

8.1: Introduction

Learning Objectives

- To develop the Cooley-Tukey FFT using index map from Multidimensional Index Mapping

The publication by Cooley and Tukey in 1965 of an efficient algorithm for the calculation of the DFT was a major turning point in the development of digital signal processing. During the five or so years that followed, various extensions and modifications were made to the original algorithm. By the early 1970's the practical programs were basically in the form used today. The standard development shows how the DFT of a length-N sequence can be simply calculated from the two length-N/2 DFT's of the even index terms and the odd index terms. This is then applied to the two half-length DFT's to give four quarter-length DFT's, and repeated until N scalars are left which are the DFT values. Because of alternately taking the even and odd index terms, two forms of the resulting programs are called decimation-in-time and decimation-in-frequency. For a length of 2^M, the dividing process is repeated $M = \log_2 N$ times and requires N multiplications each time. This gives the famous formula for the computational complexity of the FFT of $N \log_2 N$ which was derived in Multidimensional Index Mapping.

Although the decimation methods are straightforward and easy to understand, they do not generalize well. For that reason it will be assumed that the reader is familiar with that description and this chapter will develop the FFT using the index map from Multidimensional Index Mapping.

Contributor

- ContribEEBurrus

8.2: Basic Cooley-Tukey FFT

The Cooley-Tukey FFT always uses the Type 2 index map from Multidimensional Index Mapping. This is necessary for the most popular forms that have $N = R^M$, but is also used even when the factors are relatively prime and a Type 1 map could be used. The time and frequency maps from Multidimensional Index Mapping are

$$n = ((K_1 n_1 + K_2 n_2))_N$$
$$k = ((K_3 k_1 + K_4 k_2))_N$$

Type-2 conditions in the 2.2: The Index Map become

$$K_1 = aN_2 \;\; or \;\; K_2 = bN_1 \;\; but\ not\ both$$

and

$$K_3 = cN_2 \;\; or \;\; K_4 = dN_1 \;\; but\ not\ both$$

The row and column calculations in 2.2: The Index Map are uncoupled by Type-two index map which for this case are

$$((K_1 K_4))_N = 0 \;\; or \;\; ((K_2 K_3))_N = 0 \;\; but\ not\ both$$

To make each short sum a DFT, the KiKi" role="presentation" style="position:relative;" tabindex="0">K_i must satisfy

$$((K_1 K_3))_N = N_2 \;\; and \;\; ((K_2 K_4))_N = N_1$$

In order to have the smallest values for KiKi" role="presentation" style="position:relative;" tabindex="0">K_i the constants in the equation are chosen to be

$$a = d = K_2 = K_3 = 1$$

which makes the index maps of the equations to become

$$n = N_2 n_1 + n_2$$
$$k = k_1 + N_1 k_2$$

These index maps are all evaluated modulo N, but in the equation, explicit reduction is not necessary since n never exceeds N. The reduction notation will be omitted for clarity. From Multidimensiona

$$X = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2}$$

This map of the equation and the form of the DFT in the equation are the fundamentals of the Cooley-Tukey FFT.

The order of the summations using the Type 2 map in the above equation cannot be reversed as it can with the Type-1 map. This is because of the W_N terms, the twiddle factors.

Turning the equation into an efficient program requires some care. From Efficiencies Resulting from Index Mapping with the DFT we know that all the factors should be equal. If $N = R^M$, with R calle

$$TF : W_8^{n_2 k_1} = \begin{bmatrix} W^0 & W^0 \\ W^0 & W^1 \\ W^0 & W^2 \\ W^0 & W^3 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & W \\ 1 & -j \\ 1 & -jW \end{bmatrix}$$

The twiddle factor array will always have unity in the first row and first column.

To complete the equation at this point, after the row DFT's are multiplied by the TF array, the N_1 length-N_2 DFT's of the columns are calculated. However, since the columns DFT's are of length R^{M-1},

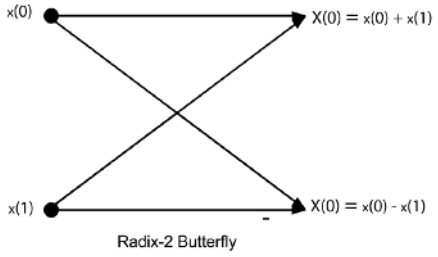


Fig. 8.2.1 A Radix-2 Butterfly

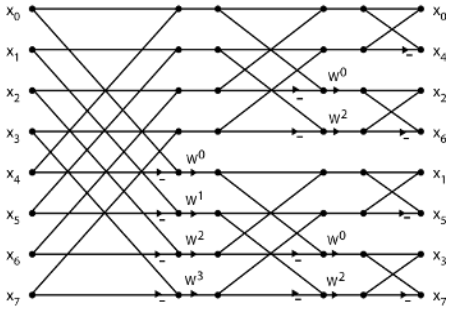


Fig. 8.2.2 Length-8 Radix-2 FFT Flow Graph

This flow-graph, the twiddle factor map of the above equation, and the basic equation should be completely understood before going further.

A very efficient indexing scheme has evolved over the years that results in a compact and efficient computer program. A FORTRAN program is given below that implements the radix-2 FFT. It should b

```
N2 = N
      DO 10 K = 1, M
         N1 = N2
         N2 = N2/2
         E  = 6.28318/N1
         A  = 0
         DO 20 J = 1, N2
            C = COS (A)
            S =-SIN (A)
            A = J*E
            DO 30 I = J, N, N1
               L = I + N2
               XT   = X(I) - X(L)
               X(I) = X(I) + X(L)
               YT   = Y(I) - Y(L)
               Y(I) = Y(I) + Y(L)
               X(L) = XT*C - YT*S
               Y(L) = XT*S + YT*C
  30        CONTINUE
  20     CONTINUE
  10  CONTINUE
```

%NOT_CONVERTED_YET: caption

A Radix-2 Cooley-Tukey FFT Program

This discussion, the flow graph of Winograd's Short DFT Algorithms and the program of Pre are all based on the input index map of The Index Map and the calculations are performed in-place. Accordi

Contributor

- ContribEEBurrus

8.3: Modifications to the Basic Cooley-Tukey FFT

Soon after the paper by Cooley and Tukey, there were improvements and extensions made. One very important discovery was the improvement in efficiency by using a larger radix of 4, 8 or even 16. For example, just as for the radix-2 butterfly, there are no multiplications required for a length-4 DFT, and therefore, a radix-4 FFT would have only twiddle factor multiplications. Because there are half as many stages in a radix-4 FFT, there would be half as many multiplications as in a radix-2 FFT. In practice, because some of the multiplications are by unity, the improvement is not by a factor of two, but it is significant. A radix-4 FFT is easily developed from the basic radix-2 structure by replacing the length-2 butterfly by a length-4 butterfly and making a few other modifications. Programs can be found in and operation counts will be given in Evaluation of the Cooley-Tukey FFT Algorithms.

Increasing the radix to 8 gives some improvement but not as much as from 2 to 4. Increasing it to 16 is theoretically promising but the small decrease in multiplications is somewhat offset by an increase in additions and the program becomes rather long. Other radices are not attractive because they generally require a substantial number of multiplications and additions in the butterflies.

The second method of reducing arithmetic is to remove the unnecessary TF multiplications by plus or minus unity or by plus or minus the square root of minus one. This occurs when the exponent of W_N is zero or a multiple of $N/4$. A reduction of additions as well as multiplications is achieved by removing these extraneous complex multiplications since a complex multiplication requires at least two real additions. In a program, this reduction is usually achieved by having special butterflies for the cases where the TF is one or j. As many as four special butterflies may be necessary to remove all unnecessary arithmetic, but in many cases there will be no practical improvement above two or three.

In addition to removing multiplications by one or j, there can be a reduction in multiplications by using a special butterfly for TFs with $W_{N/8}$, which have equal real and imaginary parts. Also, for computers or hardware with multiplication considerably slower than addition, it is desirable to use an algorithm for complex multiplication that requires three multiplications and three additions rather than the conventional four multiplications and two additions. Note that this gives no reduction in the total number of arithmetic operations, but does give a trade of multiplications for additions. This is one reason not to use complex data types in programs but to explicitly program complex arithmetic.

A time-consuming and unnecessary part of the execution of a FFT program is the calculation of the sine and cosine terms which are the real and imaginary parts of the TFs. There are basically three approaches to obtaining the sine and cosine values. They can be calculated as needed which is what is done in the sample program above. One value per stage can be calculated and the others recursively calculated from those. That method is fast but suffers from accumulated round-off errors. The fastest method is to fetch precalculated values from a stored table. This has the disadvantage of requiring considerable memory space.

If all the N DFT values are not needed, special forms of the FFT can be developed using a process called pruning which removes the operations concerned with the unneeded outputs.

Special algorithms are possible for cases with real data or with symmetric data. The decimation-in-time algorithm can be easily modified to transform real data and save half the arithmetic required for complex data. There are numerous other modifications to deal with special hardware considerations such as an array processor or a special microprocessor such as the Texas Instruments TMS320.

Contributor

- ContribEEBurrus

8.4: The Split-Radix FFT Algorithm

Recently several papers have been published on algorithms to calculate a length-2^M DFT more efficiently than a Cooley-Tukey FFT of any radix. They all have the same computational complexity and are optimal for lengths up through 16 and until recently was thought to give the best total add-multiply count possible for any power-of-two length. Yavne published an algorithm with the same computational complexity in 1968, but it went largely unnoticed. Johnson and Frigo have recently reported the first improvement in almost 40 years. The reduction in total operations is only a few percent, but it is a reduction.

The basic idea behind the split-radix FFT (SRFFT) as derived by Duhamel and Hollmann is the application of a radix-2 index map to the even-indexed terms and a radix-4 map to the odd- indexed terms. The basic definition of the DFT is:

$$C_k = \sum_{n=0}^{N-1} x_n W^{nk}$$

with W=e-j2π/NW=e-j2π/N" role="presentation" style="position:relative;" tabindex="0">$W = e^{-j2\pi/N}$ gives

$$C_{2k} = \sum_{n=0}^{N/2-1} \left[x_n + x_{n+N/2}\right] W^{2nk}$$

for the even index terms, and

$$C_{4k+1} = \sum_{n=0}^{N/4-1} \left[(x_n - x_{n+N/2}) - j(x_{n+N/4} - x_{n+3N/4})\right] W^n W^{4nk}$$

and

$$C_{4k+3} = \sum_{n=0}^{N/4-1} \left[(x_n - x_{n+N/2}) - j(x_{n+N/4} - x_{n+3N/4})\right] W^{3n} W^{4nk}$$

for the odd index terms. This results in an L-shaped “butterfly" shown in Fig. 8.4.1 which relates a length-N DFT to one length-N/2 DFT and two length-N/4 DFT's with twiddle factors. Repeating this p

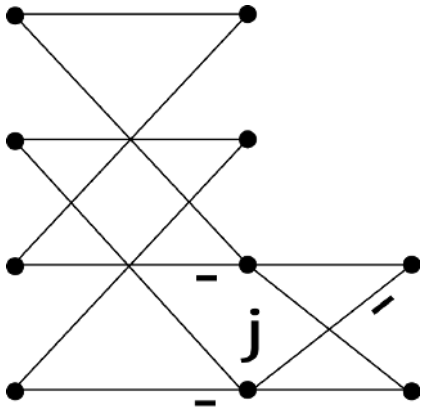


Fig. 8.4.1 SRFFT Butterfly

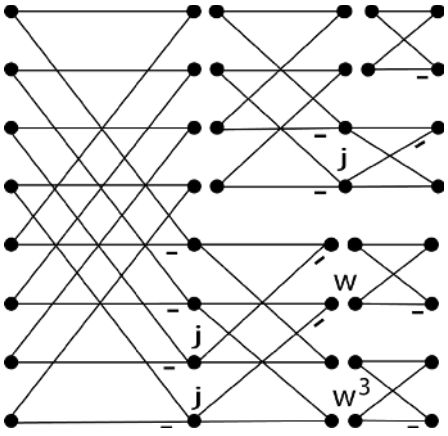


Fig. 8.4.2 Length-8 SRFFT

Unlike the fixed radix, mixed radix or variable radix Cooley-Tukey FFT or even the prime factor algorithm or Winograd Fourier transform algorithm , the Split-Radix FFT does not progress completely

A FORTRAN program is given below which implements the basic decimation-in-frequency split-radix FFT algorithm. The indexing scheme of this program gives a structure very similar to the Cooley-

FORTRAN Program implementing split-radix FFT algorithm

```
SUBROUTINE FFT(X,Y,N,M)
N2 = 2*N
DO 10 K = 1, M-1
N2 = N2/2
N4 = N2/4
```

```
E = 6.283185307179586/N2
A = 0
DO 20 J = 1, N4
A3 = 3*A
CC1 = COS(A)
SS1 = SIN(A)
CC3 = COS(A3)
SS3 = SIN(A3)
A = J*E
IS = J
ID = 2*N2
40 DO 30 I0 = IS, N-1, ID
I1 = I0 + N4
I2 = I1 + N4
I3 = I2 + N4
R1 = X(I0) - X(I2)
X(I0) = X(I0) + X(I2)
R2 = X(I1) - X(I3)
X(I1) = X(I1) + X(I3)
S1 = Y(I0) - Y(I2)
Y(I0) = Y(I0) + Y(I2)
S2 = Y(I1) - Y(I3)
Y(I1) = Y(I1) + Y(I3)
S3 = R1 - S2
R1 = R1 + S2
S2 = R2 - S1
R2 = R2 + S1
X(I2) = R1*CC1 - S2*SS1
Y(I2) =-S2*CC1 - R1*SS1
X(I3) = S3*CC3 + R2*SS3
Y(I3) = R2*CC3 - S3*SS3
30 CONTINUE
IS = 2*ID - N2 + J
ID = 4*ID
IF (IS.LT.N) GOTO 40
20 CONTINUE
10 CONTINUE
IS = 1
ID = 4
50 DO 60 I0 = IS, N, ID
I1 = I0 + 1
R1 = X(I0)
X(I0) = R1 + X(I1)
X(I1) = R1 - X(I1)
R1 = Y(I0)
Y(I0) = R1 + Y(I1)
60 Y(I1) = R1 - Y(I1)
IS = 2*ID - 1
ID = 4*ID
IF (IS.LT.N) GOTO 50
```

NOT_CONVERTED_YET: caption

Split-Radix FFT FORTRAN Subroutine

As was done for the other decimation-in-frequency algorithms, the input index map is used and the calculations are done in place resulting in the output being in bit-reversed order. It is the three statemer

An improvement in operation count has been reported by Johnson and Frigo which involves a scaling of multiplying factors. The improvement is small but until this result, it was generally thought the S

Contributor

- ContribEEBurrus

8.5: Evaluation of the Cooley-Tukey FFT Algorithms

The evaluation of any FFT algorithm starts with a count of the real (or floating point) arithmetic. The Table 8.5.1 below gives the number of real multiplications and additions required to calculate a length-N FFT of complex data. Results of programs with one, two, three and five butterflies are given to show the improvement that can be expected from removing unnecessary multiplications and additions. Results of radices two, four, eight and sixteen for the Cooley-Tukey FFT as well as of the split-radix FFT are given to show the relative merits of the various structures. Comparisons of these data should be made with the table of counts for the PFA and WFTA programs in The Prime Factor and Winograd Fourier Transform Algorithms . All programs use the four-multiply-two-add complex multiply algorithm. A similar table can be developed for the three-multiply-three-add algorithm, but the relative results are the same.

From the table it is seen that a greater improvement is obtained going from radix-2 to 4 than from 4 to 8 or 16. This is partly because length 2 and 4 butterflies have no multiplications while length 8, 16 and higher do. It is also seen that going from one to two butterflies gives more improvement than going from two to higher values. From an operation count point of view and from practical experience, a three butterfly radix-4 or a two butterfly radix-8 FFT is a good compromise. The radix-8 and 16 programs become long, especially with multiple butterflies, and they give a limited choice of transform length unless combined with some length 2 and 4 butterflies.

N	M1	M2	M3	M5	A1	A2	A3	A5
2	4	0	0	0	6	4	4	4
4	16	4	0	0	24	18	16	16
8	48	20	8	4	72	58	52	52
16	128	68	40	28	192	162	148	148
32	320	196	136	108	480	418	388	388
64	768	516	392	332	1152	1026	964	964
128	1792	1284	1032	908	2688	2434	2308	2308
256	4096	3076	2568	2316	6144	5634	5380	5380
512	9216	7172	6152	5644	13824	12802	12292	12292
1024	20480	16388	14344	13324	30720	28674	27652	27652
2048	45056	36868	32776	30732	67584	63490	61444	61444
4096	98304	81924	73736	69644	147456	139266	135172	135172
4	12	0	0	0	22	16	16	16
16	96	36	28	24	176	146	144	144
64	576	324	284	264	1056	930	920	920
256	3072	2052	1884	1800	5632	5122	5080	5080
1024	15360	11268	10588	10248	28160	26114	25944	25944
4096	73728	57348	54620	53256	135168	126978	126296	126296
8	32	4	4	4	66	52	52	52
64	512	260	252	248	1056	930	928	928
512	6144	4100	4028	3992	12672	11650	11632	11632
4096	65536	49156	48572	48280	135168	126978	126832	126832
16	80	20	20	20	178	148	148	148

256	2560	1540	1532	1528	5696	5186	5184	5184
4096	61440	45060	44924	44856	136704	128514	128480	128480
2	0	0	0	0	4	4	4	4
4	8	0	0	0	20	16	16	16
8	24	8	4	4	60	52	52	52
16	72	32	28	24	164	144	144	144
32	184	104	92	84	412	372	372	372
64	456	288	268	248	996	912	912	912
128	1080	744	700	660	2332	2164	2164	2164
256	2504	1824	1740	1656	5348	5008	5008	5008
512	5688	4328	4156	3988	12060	11380	11380	11380
1024	12744	10016	9676	9336	26852	25488	25488	25488
2048	28216	22760	22076	21396	59164	56436	56436	56436
4096	61896	50976	49612	48248	129252	123792	123792	123792

Table 8.5.1: Number of Real Multiplications and Additions for Complex Single Radix FFTs

In Table 8.2.1 Mi and Ai refer to the number of real multiplications and real additions used by an FFT with i separately written butterflies. The first block has the counts for Radix-2, the second for Radix-4, the third for Radix-8, the fourth for Radix-16, and the last for the Split-Radix FFT. For the split-radix FFT, M3 and A3 refer to the two- butterfly-plus program and M5 and A5 refer to the three-butterfly program.

The first evaluations of FFT algorithms were in terms of the number of real multiplications required as that was the slowest operation on the computer and, therefore, controlled the execution speed. Later with hardware arithmetic both the number of multiplications and additions became important. Modern systems have arithmetic speeds such that indexing and data transfer times become important factors. Morris has looked at some of these problems and has developed a procedure called autogen to write partially straight-line program code to significantly reduce overhead and speed up FFT run times. Some hardware, such as the TMS320 signal processing chip, has the multiply and add operations combined. Some machines have vector instructions or have parallel processors. Because the execution speed of an FFT depends not only on the algorithm, but also on the hardware architecture and compiler, experiments must be run on the system to be used.

In many cases the unscrambler or bit-reverse-counter requires 10% of the execution time, therefore, if possible, it should be eliminated. In high-speed convolution where the convolution is done by multiplication of DFT's, a decimation-in-frequency FFT can be combined with a decimation-in-time inverse FFT to require no unscrambler. It is also possible for a radix-2 FFT to do the unscrambling inside the FFT but the structure is not very regular.

Although there can be significant differences in the efficiencies of the various Cooley-Tukey and Split-Radix FFTs, the number of multiplications and additions for all of them is on the order of $N \log N$. That is fundamental to the class of algorithms.

Contributor

- ContribEEBurrus

8.6: The Quick Fourier Transform - An FFT based on Symmetries

The development of fast algorithms usually consists of using special properties of the algorithm of interest to remove redundant or unnecessary operations of a direct implementation. The discrete Fourier transform (DFT) defined by

$$C(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

where

$$W_N = e^{-j2\pi/N}$$

has enormous capacity for improvement of its arithmetic efficiency. Most fast algorithms use the periodic and symmetric properties of its basis functions. The classical Cooley-Tukey FFT and prime factor FFT exploit the periodic properties of the cosine and sine functions. Their use of the periodicities to share and, therefore, reduce arithmetic operations depends on the factorability of the length of the data to be transformed. For highly composite lengths, the number of floating-point operation is of order $N \log(N)$ and for prime lengths it is of order N^2.

This section will look at an approach using the symmetric properties to remove redundancies. This possibility has long been recognized but has not been developed in any systematic way in the open literature. We will develop an algorithm, called the quick Fourier transform (QFT), that will reduce the number of floating point operations necessary to compute the DFT by a factor of two to four over direct methods or Goertzel's method for prime lengths. Indeed, it seems the best general algorithm available for prime length DFTs. One can always do better by using Winograd type algorithms but they must be individually designed for each length. The Chirp Z-transform can be used for longer lengths.

Input and Output Symmetries

We use the fact that the cosine is an even function and the sine is an odd function. The kernel of the DFT or the basis functions of the expansion is given by

$$W_N^{nk} = e^{-j2\pi nk/N} = \cos(2\pi nk/N) + j\sin(2\pi nk/N)$$

which has an even real part and odd imaginary part. If the data $x(n)$ are decomposed into their real and imaginary parts and those into their even and odd parts, we have

$$x(n) = u(n) + jv(n) = [u_e(n) + u_o(n)] + j[v_e(n) + v_o(n)]$$

where the even part of the real part of $x(n)$ is given by

$$u_e(n) = (u(n) + u(-n))/2$$

and the odd part of the real part is

$$u_o(n) = (u(n) - u(-n))/2$$

with corresponding definitions of $v_e(n)$ and $v_o(n)$. Using Convolution Algorithms with a simpler notation, the DFT of Convolution Algorithms becomes

$$C(k) = \sum_{n=0}^{N-1} (u + jv)(\cos - j\sin)$$

The sum over an integral number of periods of an odd function is zero and the sum of an even function over half of the period is one half the sum over the whole period. This causes the equations to become

$$C(k) = \sum_{n=0}^{N/2-1} [u_e \cos + v_o \sin] + j[v_e \cos - v_o \sin]$$

for $k = 0, 1, 2, \ldots, N-1$

The evaluation of the DFT using the convolution algorithm equation requires half as many real multiplication and half as many real additions as evaluating it using the other equations. We have exploited the symmetries of the sine and cosine as functions of the

time index n. This is independent of whether the length is composite or not. Another view of this formulation is that we have used the property of associatively of multiplication and addition. In other words, rather than multiply two data points by the same value of a sine or cosine then add the results, one should add the data points first then multiply the sum by the sine or cosine which requires one rather than two multiplications.

Next we take advantage of the symmetries of the sine and cosine as functions of the frequency index k. . Using these symmetries on the equation gives

$$C(k) = \sum_{n=0}^{N/2-1} [u_e \cos + v_o \sin] + j[v_e \cos - v_o \sin]$$

$$C(N-k) = \sum_{n=0}^{N/2-1} [u_e \cos - v_o \sin] + j[v_e \cos + v_o \sin]$$

for $k = 0, 1, 2, \ldots, N/2-1$. This again reduces the number of operations by a factor of two, this time because it calculates two output values at a time. The first reduction by a factor of two is always available. The second is possible only if both DFT values are needed. It is not available if you are calculating only one DFT value. The above development has not dealt with the details that arise with the difference between an even and an odd length. That is straightforward.

Further Reductions if the Length is Even

If the length of the sequence to be transformed is even, there are further symmetries that can be exploited. There will be four data values that are all multiplied by plus or minus the same sine or cosine value. This means a more complicated pre-addition process which is a generalization of the simple calculation of the even and odd parts in the equations will reduce the size of the order N^2 part of the algorithm by still another factor of two or four. It the length is divisible by 4, the process can be repeated. Indeed, it the length is a power of 2, one can show this process is equivalent to calculating the DFT in terms of discrete cosine and sine transforms with a resulting arithmetic complexity of order $N \log(N)$ and with a structure that is well suited to real data calculations and pruning.

If the flow-graph of the Cooley-Tukey FFT is compared to the flow-graph of the QFT, one notices both similarities and differences. Both progress in stages as the length is continually divided by two. The Cooley-Tukey algorithm uses the periodic properties of the sine and cosine to give the familiar horizontal tree of butterflies. The parallel diagonal lines in this graph represent the parallel stepping through the data in synchronism with the periodic basis functions. The QFT has diagonal lines that connect the first data point with the last, then the second with the next to last, and so on to give a “star" like picture. This is interesting in that one can look at the flow graph of an algorithm developed by some completely different strategy and often find section with the parallel structures and other parts with the star structure. These must be using some underlying periodic and symmetric properties of the basis functions.

Arithmetic Complexity and Timings

A careful analysis of the QFT shows that $2N$ additions are necessary to compute the even and odd parts of the input data. This is followed by the length $N/2$ inner product that requires $4(N/2)^2 = N^2$ real multiplications and an equal number of additions. This is followed by the calculations necessary for the simultaneous calculations of the first half and last half of $C(k)$ which requires $4(N/2) = 2N$ real additions. This means the total QFT algorithm requires M^2 real multiplications and $N^2 + 4N$ real additions. These numbers along with those for the Goertzel algorithm and the direct calculation of the DFT are included in the following table. Of the various order-N^2 DFT algorithms, the QFT seems to be the most efficient general method for an arbitrary length N.

Algorithm	Real Mults.	Real Adds	Trig Eval.
Direct DFT	$4N^2$	$4N^2$	$2N^2$
Mod. 2nd Order Goertzel	$N^2 + N$	$2N^2 + N$	N
QFT	N^2	$N^2 + 4N$	$2N$

Timings of the algorithms on a PC in milliseconds are given in the following table.

Algorithm	$N = 125$	$N = 256$
Direct DFT	4.90	19.83
Mod. 2O. Goertzel	1.32	5.55
QFT	1.09	4.50
Chirp + FFT	1.70	3.52

These timings track the floating point operation counts fairly well.

Conclusions

The QFT is a straight-forward DFT algorithm that uses all of the possible symmetries of the DFT basis function with no requirements on the length being composite. These ideas have been proposed before, but have not been published or clearly developed. It seems that the basic QFT is practical and useful as a general algorithm for lengths up to a hundred or so. Above that, the chirp z-transform or other filter based methods will be superior. For special cases and shorter lengths, methods based on Winograd's theories will always be superior. Nevertheless, the QFT has a definite place in the array of DFT algorithms and is not well known. A Fortran program is included in the appendix.

It is possible, but unlikely, that further arithmetic reduction could be achieved using the fact that W_N has unity magnitude as was done in second-order Goertzel algorithm. It is also possible that some way of combining the Goertzel and QFT algorithm would have some advantages. A development of a complete QFT decomposition of a DFT of length-2^M shows interesting structure and arithmetic complexity comparable to average Cooley-Tukey FFTs. It does seem better suited to real data calculations with pruning.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

Back Matter

Index

CHAPTER OVERVIEW

9: The Prime Factor and Winograd Fourier Transform Algorithms

Topic hierarchy

9.1: Introduction

Learning Objectives

- To study the Prime Factor Algorithm and the Winograd Fourier Transform Algorithm

The prime factor algorithm (PFA) and the Winograd Fourier transform algorithm (WFTA) are methods for efficiently calculating the DFT which use, and in fact, depend on the Type-1 index map from Multidimensional Index Mapping. The use of this index map preceded Cooley and Tukey's paper but its full potential was not realized until it was combined with Winograd's short DFT algorithms.

The number theoretic basis for the indexing in these algorithms may, at first, seem more complicated than in the Cooley-Tukey FFT; however, if approached from the general index mapping point of view of Multidimensional Index Mapping, it is straightforward, and part of a common approach to breaking large problems into smaller ones. The development in this section will parallel that in The Cooley-Tukey Fast Fourier Transform Algorithm.

The general index maps of Multidimensional Index Mapping must satisfy the Type-1 conditions which are

$$K_1 = aN_2 \text{ and } K_2 = bN_1 \text{ with } (K_1 N_1) = (K_2 N_2) = 1$$

$$K_3 = cN_2 \text{ and } K_4 = dN_1 \text{ with } (K_3 N_1) = (K_4 N_2) = 1$$

The row and column calculations in The Index Map are uncoupled by which for this case are

$$((K_1 K_4))_N = ((K_2 K_3))_N = 0$$

In addition, to make each short sum a DFT, the KiKi" role="presentation" style="position:relative;" tabindex="0">K_i must also satisfy

$$((K_1 K_3))_N = N_2 \text{ and } ((K_2 K_4))_N = N_1$$

In order to have the smallest values for KiKi" role="presentation" style="position:relative;" tabindex="0">K_i, the constants in the equation are chosen to be

KiKi" role="presentation" style="position:relative;" tabindex="0">

$$a = b = 1,\ c = ((N_2^{-1}))_N,\ d = ((N_1^{-1}))_N$$

which gives for the index maps in the equation

$$n = ((N_2 n_1 + N_1 n_2))_N$$

$$k = ((K_3 k_1 + K_4 k_2))_N$$

The frequency index map is a form of the Chinese remainder theorem. Using these index maps, the DFT in Multidimensional Index Mapping becomes

$$X = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2}$$

which is a pure two-dimensional DFT with no twiddle factors and the summations can be done in either order. Choices other than the equation could be used. For example $a = b = c = d = 1$ will cause

An important feature of the short Winograd DFT's described in Winograd's Short DFT Algorithms that is useful for both the PFA and WFTA is the fact that the multiplier constants in Winograd's Short D

DD" role="presentation" style="position:relative;" tabindex="0">Contributor

- ContribEEBurrus

9.2: The Prime Factor Algorithm

If the DFT is calculated directly using the equation in 9.1: Introduction, the algorithm is called a prime factor algorithm and was discussed in Winograd's Short DFT Algorithms. When the short DFT's are calculated by the very efficient algorithms of Winograd discussed in Factoring the Signal Processing Operators, the PFA becomes a very powerful method that is as fast or faster than the best Cooley-Tukey FFT's.

A flow graph is not as helpful with the PFA as it was with the Cooley-Tukey FFT, however, the following representation in Fig. 9.2.1 below, which combines the figures in The Index Map and Winograd Fourier Transform Algorithm (WFTA) gives a good picture of the algorithm with the example of Multidimensional Index Mapping.

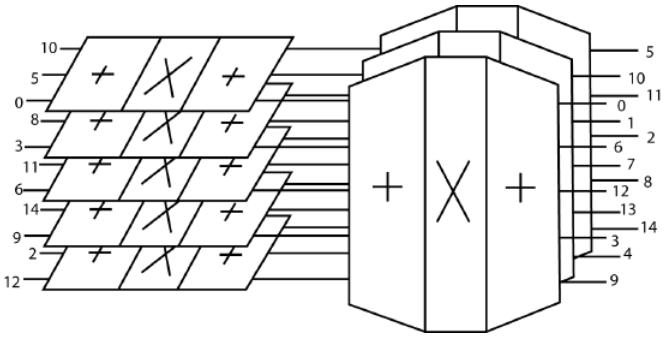


Fig. 9.2.1 A Prime Factor FFT for N = 15

If N is factored into three factors, the DFT of the equation would have three nested summations and would be a three-dimensional DFT. This principle extends to any number of factors; however, recall that the Type-1 map requires that all the factors be relatively prime. A very simple three-loop indexing scheme has been developed which gives a compact, efficient PFA program for any number of factors. The basic program structure is illustrated below with the short DFT's being omitted for clarity. Complete programs are given in the appendices.

```
C---------------PFA INDEXING LOOPS--------------
               DO 10 K = 1, M
                  N1 = NI(K)
                  N2 = N/N1
                  I(1) = 1
                  DO 20 J = 1, N2
                     DO 30 L=2, N1
                        I(L) = I(L-1) + N2
                        IF (I(L .GT.N)  I(L) = I(L) - N
         30           CONTINUE
                     GOTO (20,102,103,104,105), N1
                     I(1) = I(1) + N1
         20        CONTINUE
         10    CONTINUE
               RETURN
       C---------------MODULE FOR N=2-----------------
         102   R1       = X(I(1))
               X(I(1))  = R1 + X(I(2))
               X(I(2))  = R1 - X(I(2))
               R1       = Y(I(1))
               Y(I(1))  = R1 + Y(I(2))
               Y(I(2))  = R1 - Y(I(2))
               GOTO 20
       C---------------OTHER MODULES-----------------
         103   Length-3 DFT
         104   Length-4 DFT
         105   Length-5 DFT
               etc.
```

:NOT_CONVERTED_YET: caption

Part of a FORTRAN PFA Program

As in the Cooley-Tukey program, the DO 10 loop steps through the M stages (factors of N) and the DO 20 loop calculates the N/N1 length-N1 DFT's. The input index map of the equation is implemented in the DO 30 loop and the statement just before label 20. In the PFA, each stage or factor requires a separately programmed module or butterfly. This lengthens the PFA program but an efficient Cooley-Tukey program will also require three or more butterflies.

Because the PFA is calculated in-place using the input index map, the output is scrambled. There are five approaches to dealing with this scrambled output. First, there are some applications where the output does not have to be unscrambled as in the case of high-speed convolution. Second, an unscrambler can be added after the PFA to give the output in correct order just as the bit-reversed-counter is used for the Cooley-Tukey FFT. The third method does the unscrambling in the modules while they are being calculated. This is probably the fastest method but the program must be written for a specific length. A fourth method is similar and achieves the unscrambling by choosing the multiplier constants in the modules properly. The fifth method uses a separate indexing method for the input and output of each module.

Contributor

- ContribEEBurrus

9.3: The Winograd Fourier Transform Algorithm

The Winograd Fourier transform algorithm (WFTA) uses a very powerful property of the Type-1 index map and the DFT to give a further reduction of the number of multiplications in the PFA. Using an operator notation where F_1 represents taking row DFT's and F_2 represents column DFT's, the two-factor PFA of the equation is represented by

$$X = F_2 F_1 x$$

It has been shown that if each operator represents identical operations on each row or column, they commute. Since F_1 and F_2 represent length N_1 and N_2 DFT's, they commute and the equation can also be written

N1N1" role="presentation" style="position:relative;" tabindex="0">F1F1" role="presentation" style="position:relative;" tabindex="0">

$$X = F_1 F_2 x$$

If each short DFT in FFF" role="presentation" style="position:relative;" tabindex="0"> is expressed by three operators as in Winograd's Short DFT Algorithms FFF" role="presentation" style="positio

FF" role="presentation" style="position:relative;" tabindex="0">

$$F = A^T D A$$

where AFF" role="presentation" style="position:relative;" tabindex="0"> represents the set of additions done on each row or column that performs the residue reduction as Winograd's Short DFT Algori

$$X = A_2^T D_2 A_2 A_1^T D_1 A_1 x$$

This is the PFA of the equation and Fig. 9.2.1 where $A_1 D_1 A_1$ represents the row DFT's on the array formed from x. Because these operators commute, the equation can also be written as

$$X = A_2^T A_1^T D_2 D_1 A_2 A_1 x$$

or

$$X = A_1^T A_2^T D_2 D_1 A_2 A_1 x$$

but the two adjacent multiplication operators can be premultiplied and the result represented by one operator $D = D_2 D_1$ which is no longer the same for each row or column. The equation then become

$$X = A_1^T A_2^T D A_2 A_1 x$$

This is the basic idea of the Winograd Fourier transform algorithm. The commuting of the multiplication operators together in the center of the algorithm is called nesting and it results in a significant de

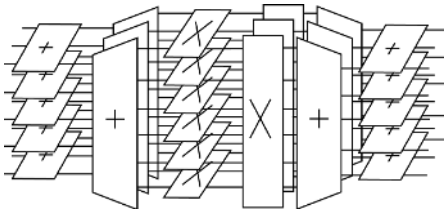


Fig. 9.3.1 A Length-15 WFTA with Nested Multiplications

The rectangular structure of the preweave addition operators causes an expansion of the data in the center of the algorithm. The 15 data points in Fig. 9.3.1 become 18 intermediate values. This expansio

From Fig. 9.3.1 and the idea of premultiplying the individual multiplication operators, it can be seen why the multiplications by unity had to be considered in Winograd Fourier Transform Algorithm (W

The number of additions depends on the order of the pre- and postweave operators. For example in the length-15 WFTA in Fig. 9.3.1, if the length-5 had been done first and last, there would have been s

A program for the WFTA is not as simple as for the FFT or PFA because of the very characteristic that reduces the number of multiplications, the nesting. The same lengths are possible with the PFA and

Contributor

- ContribEEBurrus

9.4: Modifications of the PFA and WFTA Type Algorithms

In the previous section it was seen how using the permutation property of the elementary operators in the PFA allowed the nesting of the multiplications to reduce their number. It was also seen that a proper ordering of the operators could minimize the number of additions. These ideas have been extended in formulating a more general algorithm optimizing problem. If the DFT operator F FF" role="presentation" style="position:relative;" tabindex="0"> in the equation is expressed in a still more factored form obtained from Winograd's Short DFT Algorithms, a greater variety of ordering

FF" role="presentation" style="position:relative;" tabindex="0">

$$F_1 = A_1^T A_1'^T D_1 A_1' A_1$$

The DFT in the equation becomes

$$X = A_2^T A_2'^T D_2 A_2' A_2 A_1^T A_1'^T D_1 A_1' A_1 x$$

The operator notation is very helpful in understanding the central ideas, but may hide some important facts. It has been shown that operators in different F_iFiFi" role="presentation" style="position:relat

FiFi" role="presentation" style="position:relative;" tabindex="0">This formulation allows a very large set of possible orderings, in fact, the number is so large that some automatic technique must be use

FiFi" role="presentation" style="position:relative;" tabindex="0">Results obtained applying the dynamic programming method to the design of fairly long DFT algorithms gave algorithms that had fewe

FiFi" role="presentation" style="position:relative;" tabindex="0">There are other modifications of the basic structure of the Type-1 index map DFT algorithm. One is to use the same index structure and

FiFi" role="presentation" style="position:relative;" tabindex="0">Contributor

- FiFi" role="presentation" style="position:relative;" tabindex="0">FiFi" role="presentation" style="position:relative;" tabindex="0">ContribEEBurrus

9.5: Evaluation of the PFA and WFTA

As for the Cooley-Tukey FFT's, the first evaluation of these algorithms will be on the number of multiplications and additions required. The number of multiplications to compute the PFA in the equation is given by Multidimensional Index Mapping. Using the notation that $T(N)$ is the number of multiplications or additions necessary to calculate a length-N DFT, the total number for a four-factor PFA of length-N, where N=N1N2N3N4N=N1N2N3N4" role="presentation" style="position:relative;" tabindex="0">$N = N_1 N_2 N_3 N_4$ is

$$T(N) = N_1 N_2 N_3 T(N_4) + N_2 N_3 N_4 T(N_1) + N_3 N_4 N_1 T(N_2) + N_4 N_1 N_2 T(N_3)$$

The count of multiplies and adds in the Table 9.5.1 below are calculated from (105) with the counts of the factors taken from Winograd Fourier Transform Algorithm (WFTA) Table 6.2.1. The list of len

The number of multiplications necessary for the WFTA is simply the product of those necessary for the required modules, including multiplications by unity. The total number may contain some unity m

Calculating the number of additions for the WFTA is more complicated than for the PFA because of the expansion of the data moving through the algorithm. For example the number of additions, TA, fc

$$TA(N) = N_2 TA(N_1) + TM_1 TA(N_2)$$

where $N_1 = 3,\ N_2 = 5,\ TM_1 =$ the number of multiplies for the length-3 module and hence the expansion factor. As mentioned earlier there is an optimum ordering to minimize additions. The orderin

Table 9.5.1: Number of Real Multiplications and Additions for Complex PFA and WFTA FFTs

Length	PFA	PFA	WFTA	WFTA	WFTA
N	Mults	Adds	Mults	RMults	Adds
10	20	88	24	20	88
12	16	96	24	16	96
14	32	172	36	32	172
15	50	162	36	34	162
18	40	204	44	40	208
20	40	216	48	40	216
21	76	300	54	52	300
24	44	252	48	36	252
28	64	400	72	64	400
30	100	384	72	68	384
35	150	598	108	106	666
36	80	480	88	80	488
40	100	532	96	84	532
42	152	684	108	104	684
45	190	726	132	130	804
48	124	636	108	92	660
56	156	940	144	132	940
60	200	888	144	136	888
63	284	1236	198	196	1394
70	300	1336	216	212	1472
72	196	1140	176	164	1156
80	260	1284	216	200	1352
84	304	1536	216	208	1536
90	380	1632	264	260	1788
105	590	2214	324	322	2418
112	396	2188	324	308	2332
120	460	2076	288	276	2076
126	568	2724	396	392	3040
140	600	2952	432	424	3224
144	500	2676	396	380	2880
168	692	3492	432	420	3492
180	760	3624	528	520	3936
210	1180	4848	648	644	5256
240	1100	4812	648	632	5136
252	1136	5952	792	784	6584
280	1340	6604	864	852	7148
315	2050	8322	1188	1186	10336
336	1636	7908	972	956	8508
360	1700	8148	1056	1044	8772
420	2360	10536	1296	1288	11352
504	2524	13164	1584	1572	14428

Length	PFA	PFA	WFTA	WFTA	WFTA
560	3100	14748	1944	1928	17168
630	4100	17904	2376	2372	21932
720	3940	18276	2376	2360	21132
840	5140	23172	2592	2580	24804
1008	5804	29100	3564	3548	34416
1260	8200	38328	4752	4744	46384
1680	11540	50964	5832	5816	59064
2520	17660	82956	9504	9492	99068
5040	39100	179772	21384	21368	232668

From the Table 9.5.1 we see that compared to the PFA or any of the Cooley-Tukey FFT's, the WFTA has significantly fewer multiplications. For the shorter lengths, the WFTA and the PFA have approxi

The size of the Cooley-Tukey program is the smallest, the PFA next and the WFTA largest. The PFA requires the smallest number of stored constants, the Cooley-Tukey or split-radix FFT next, and the '

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

10: Implementing FFTs in Practice

Topic hierarchy

10.1: Introduction

Learning Objectives

- Discussion of the considerations involved in high-performance FFT implementations, which center largely on memory access and other non-arithmetic concerns, as illustrated by a case study of the FFTW library.

by Steven G. Johnson (Department of Mathematics, Massachusetts Institute of Technology) and Matteo Frigo (Cilk Arts, Inc.)

Although there are a wide range of fast Fourier transform (FFT) algorithms, involving a wealth of mathematics from number theory to polynomial algebras, the vast majority of FFT implementations in practice employ some variation on the Cooley-Tukey algorithm. The Cooley-Tukey algorithm can be derived in two or three lines of elementary algebra. It can be implemented almost as easily, especially if only power-of-two sizes are desired; numerous popular textbooks list short FFT subroutines for power-of-two sizes, written in the language du jour. The implementation of the Cooley-Tukey algorithm, at least, would therefore seem to be a long-solved problem. In this chapter, however, we will argue that matters are not as straightforward as they might appear.

For many years, the primary route to improving upon the Cooley-Tukey FFT seemed to be reductions in the count of arithmetic operations, which often dominated the execution time prior to the ubiquity of fast floating-point hardware (at least on non-embedded processors). Therefore, great effort was expended towards finding new algorithms with reduced arithmetic counts, from Winograd's method to achieve $\Theta(n)$ multiplications[1] (at the cost of many more additions) to the split-radix variant on Cooley-Tukey that long achieved the lowest known total count of additions and multiplications for power-of-two sizes (but was recently improved upon). The question of the minimum possible arithmetic count continues to be of fundamental theoretical interest—it is not even known whether better than $\Theta(n \log n)$ complexity is possible, since Ω(nlogn)Ω(nlogn)" role="presentation" style="position:relative;" tabindex="0">$\Omega(n \log n)$ lower bounds on the count of additions have only been proven subject to restrictive assumptions abou

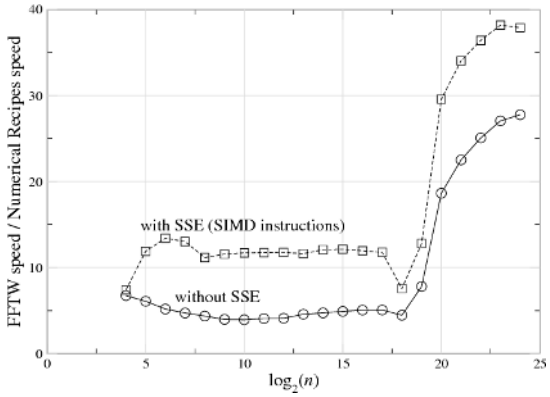


Fig. 10.1.1 The ratio of speed (1/time) between a highly optimized FFT (FFTW 3.1.2) and a typical textbook radix-2 implementation (Numerical Recipes in C on a 3 GHz Intel Core Duo with the Intel C

And yet there is a vast gap between this basic mathematical theory and the actual practice—highly optimized FFT packages are often an order of magnitude faster than the textbook subroutines, and the i

In particular, in this chapter we will discuss some of the lessons learned and the strategies adopted in the FFTW library. FFTW is a widely used free-software library that computes the discrete Fourier tra

This chapter is structured as follows. First Review of the Cooley-Tukey FFT we briefly review the basic ideas behind the Cooley-Tukey algorithm and define some common terminology, especially focu

Footnotes

[1] We employ the standard asymptotic notation of O for asymptotic upper bounds, Θ for asymptotic tight bounds, and Ω for asymptotic lower bounds

[2] We won't address the question of parallelization on multi-processor machines, which adds even greater difficulty to FFT implementation—although multi-processors are increasingly important, achievi

Contributor

- ContribEEBurrus

10.2: Review of the Cooley-Tukey FFT

The (forward, one-dimensional) discrete Fourier transform (DFT) of an array XX" role="presentation" style="position:relative;" tabindex="0">$\mathbf{X}$ n complex numbers is the array XX" role="presentation" style="position:relative;" tabindex="0">$\mathbf{Y}$ given by

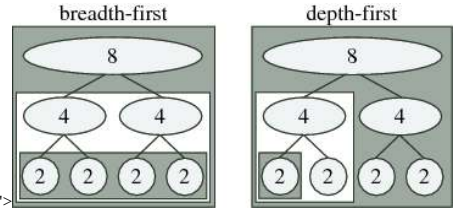
YY" role="presentation" style="position:relative;" tabindex="0">

$$\mathbf{Y}[k] = \sum_{l=0}^{n-1} \mathbf{X}[l]\omega_n^{lk}$$

where $0 \leq k < n$ and $\omega_n = \exp(-2\pi i/n)$ is a primitive root of unity. Implemented directly, the equation would require Θ(n2)Θ(n2)" role="presentation" style="position:relative;" tabindex="0">
The Cooley-Tukey algorithm can be derived as follows. If n1n1" role="presentation" style="position:relative;" tabindex="0">n can be factored into n=n1n2n=n1n2" role="presentation" style="position:

$$\mathbf{Y}\left[k_1 + k_2 n_1\right] = \sum_{l_2=0}^{n_2-1} \left[\left(\sum_{l_1=0}^{n_1-1} \mathbf{X}[l_1 n_2 + l_2] \omega_{n_1}^{l_1 k_1} \right) \omega_n^{l_2 k_1} \right] \omega_{n_2}^{l_2 k_2}$$

where $k_{1,2} = 0, \ldots n_{1,2} - 1$. Thus, the algorithm computes n1n1" role="presentation" style="position:relative;" tabindex="0">n_2 DFTs of size n1n1" role="presentation" style="position:relative;" tabin
Many well-known variations are distinguished by the radix alone. A **decimation in time** (**DIT**) algorithm uses n1n1" role="presentation" style="position:relative;" tabindex="0">n1n1" role="presentatic
A key difficulty in implementing the Cooley-Tukey FFT is that the n1n1" role="presentation" style="position:relative;" tabindex="0">n_1 dimension corresponds to discontiguous inputs ℓ1ℓ1" r



n2n2" role="presentation" style="position:relative;" tabindex="0">

Fig. 10.2.1 Schematic of traditional breadth-first (left) vs. recursive depth-first (right) ordering for radix-2 FFT of size 8: the computations for each n

One ordering distinction is between recursion and iteration. As expressed above, the Cooley-Tukey algorithm could be thought of as defining a tree of smaller and smaller DFTs, as depicted in Fig. 10.2.

```
C--------------PFA INDEXING LOOPS--------------
               DO 10 K = 1, M
                  N1 = NI(K)
                  N2 = N/N1
                  I(1) = 1
                  DO 20 J = 1, N2
                     DO 30 L=2, N1
                        I(L) = I(L-1) + N2
                        IF (I(L .GT.N)  I(L) = I(L) - N
            30        CONTINUE
                     GOTO (20,102,103,104,105), N1
                     I(1) = I(1) + N1
            20     CONTINUE
            10  CONTINUE
                RETURN
          C---------------MODULE FOR N=2----------------
            102   R1       = X(I(1))
                  X(I(1))  = R1 + X(I(2))
                  X(I(2))  = R1 - X(I(2))
                  R1       = Y(I(1))
                  Y(I(1))  = R1 + Y(I(2))
                  Y(I(2))  = R1 - Y(I(2))
                  GOTO 20
          C---------------OTHER MODULES----------------
            103   Length-3 DFT
            104   Length-4 DFT
            105   Length-5 DFT
                  etc.
```

NOT_CONVERTED_YET: caption
Part of a FORTRAN PFA Program

A second ordering distinction lies in how the digit-reversal is performed. The classic approach is a single, separate digit-reversal pass following or preceding the arithmetic computations; this approach is

Finally, we should mention that there are many FFTs entirely distinct from Cooley-Tukey. Three notable such algorithms are the **prime-factor algorithm** for $\gcd(n_1, n_2) = 1$ along with Rader's and Blu

nn" role="presentation" style="position:relative;" tabindex="0">Contributor

- ContribEEBurrus

10.3: Goals and Background of the FFTW Project

The FFTW project, begun in 1997 as a side project of the authors Frigo and Johnson as graduate students at MIT, has gone through several major revisions, and as of 2008 consists of more than 40,000 lines of code. It is difficult to measure the popularity of a free-software package, but (as of 2008) FFTW has been cited in over 500 academic papers, is used in hundreds of shipping free and proprietary software packages, and the authors have received over 10,000 emails from users of the software. Most of this chapter focuses on performance of FFT implementations, but FFTW would probably not be where it is today if that were the only consideration in its design. One of the key factors in FFTW's success seems to have been its flexibility in addition to its performance. In fact, FFTW is probably the most flexible DFT library available:

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFTs in O(nlogn)O(nlogn)" role="presentation" style="position:relative;" tabindex="0">$O(n \log n)$ time for any length nn" role="presentation" style="position:relative;" tabindex="0">n. (Most other DFT i
- FFTW imposes no restrictions on the rank (dimensionality) of multi-dimensional transforms. (Most other implementations are limited to one-dimensional, or at most two- and three-dimensional data.
- FFTW supports multiple and/or strided DFTs; for example, to transform a 3-component vector field or a portion of a multi-dimensional array. (Most implementations support only a single DFT of co
- FFTW supports DFTs of real data, as well as of real symmetric/anti-symmetric data (also called discrete cosine/sine transforms).

Our design philosophy has been to first define the most general reasonable functionality, and then to obtain the highest possible performance without sacrificing this generality. In this section, we offer a

FFTW's generality is partly a consequence of the fact the FFTW project was started in response to the needs of a real application for one of the authors (a spectral solver for Maxwell's equations, which f

Even for one-dimensional DFTs, there is a common misperception that one should always choose power-of-two sizes if one cares about efficiency. Thanks to FFTW's code generator (described in Gener

One initially missing feature was efficient support for large prime sizes; the conventional wisdom was that large-prime algorithms were mainly of academic interest, since in real applications (including

Another form of flexibility that deserves comment has to do with a purely technical aspect of computer software. FFTW's implementation involves some unusual language choices internally (the FFT-ke

Ultimately, very few scientific-computing applications should have performance as their top priority. Flexibility is often far more important, because one wants to be limited only by one's imagination, ra

Contributor

- ContribEEBurrus

10.4: FFTs and the Memory Hierarchy

There are many complexities of computer architectures that impact the optimization of FFT implementations, but one of the most pervasive is the memory hierarchy. On any modern general-purpose computer, memory is arranged into a hierarchy of storage devices with increasing size and decreasing speed: the fastest and smallest memory being the CPU registers, then two or three levels of cache, then the main-memory RAM, then external storage such as hard disks.[3] Most of these levels are managed automatically by the hardware to hold the most-recently-used data from the next level in the hierarchy.[4] There are many complications, however, such as limited cache associativity (which means that certain locations in memory cannot be cached simultaneously) and cache lines (which optimize the cache for contiguous memory access), which are reviewed in numerous textbooks on computer architectures. In this section, we focus on the simplest abstract principles of memory hierarchies in order to grasp their fundamental impact on FFTs.

Because access to memory is in many cases the slowest part of the computer, especially compared to arithmetic, one wishes to load as much data as possible in to the faster levels of the hierarchy, and then perform as much computation as possible before going back to the slower memory devices. This is called **temporal locality**: if a given datum is used more than once, we arrange the computation so that these usages occur as close together as possible in time.

Understanding FFTs with an ideal cache

To understand temporal-locality strategies at a basic level, in this section we will employ an idealized model of a cache in a two-level memory hierarchy. This **ideal cache** stores $\mathbf{Z}$ data items from main memory (e.g. complex numbers for our purposes): when the processor loads a datum from memory, the access is quick if the datum is already in the cache (a **cache hit**) and slow otherwise (a **cache miss**, which requires the datum to be fetched into the cache). When a datum is loaded into the cache,[5] it must replace some other datum, and the ideal-cache model assumes that the optimal replacement strategy is used: the new datum replaces the datum that will not be needed for the longest time in the future; in practice, this can be simulated to within a factor of two by replacing the least-recently used datum, but ideal replacement is much simpler to analyze. Armed with this ideal-cache model, we can now understand some basic features of FFT implementations that remain essentially true even on real cache architectures. In particular, we want to know the **cache complexity**, the number Q(n;Z)Q(n;Z)" role="presentation" style="position:relative;" tabindex="0">$Q(n; \mathbf{Z})$ of cache misses for an FFT of size nn" role="presentation" style="position:relative;" tabindex="0">n with an ideal c

ZZ" role="presentation" style="position:relative;" tabindex="0">First, consider a textbook radix-2 algorithm, which divides nn" role="presentation" style="position:relative;" tabindex="0">n by 2 at eac

One traditional solution to this problem is **blocking**: the computation is divided into maximal blocks that fit into the cache, and the computations for each block are completed before moving on to the ne

$$Q_b(n; \mathbf{Z}) = \Theta(\log_{\mathbf{Z}} n)$$

In fact, this complexity is rigorously **optimal** for Cooley-Tukey FFT algorithms, and immediately points us towards **large radices** (not radix 2!) to exploit caches effectively in FFTs.

However, there is one shortcoming of any blocked FFT algorithm: it is **cache aware**, meaning that the implementation depends explicitly on the cache size $\mathbf{Z}$. The implementation must be modified (e.g

The goal of cache-obliviousness is to structure the algorithm so that it exploits the cache without having the cache size as a parameter: the same code achieves the same asymptotic cache complexity rega

For instance, Fig 10.2.1 (right) and the algorithm of Pre shows a way to obliviously exploit the cache with a radix-2 Cooley-Tukey algorithm, by ordering the computation depth-first rather than breadth-

$$Q_2(n; \mathbf{Z}) = \begin{cases} n & n \le \mathbf{Z} \\ 2Q_2(n/2; \mathbf{Z}) + \Theta(n) & \text{otherwise} \end{cases}$$

The key property is this: once the recursion reaches a size n≤Zn≤Z" role="presentation" style="position:relative;" tabindex="0">$n \le \mathbf{Z}$, the subtransform fits into the cache and no further misse

$$Q_2(n; \mathbf{Z}) = \Theta(n \log\lfloor n/\mathbf{Z} \rfloor)$$

This is worse than the theoretical optimum $Q_b(n; \mathbf{Z})$ from the equation, but it is cache-oblivious ($\mathbf{Z}$ never entered the algorithm) and exploits at least **some** temporal locality.[7] On the other hand, when it

There exists a different recursive FFT that is **optimal** cache-oblivious, however, and that is the radix-$\sqrt{n}$ "four-step" Cooley-Tukey algorithm (again executed recursively, depth-first). The cache comple

$$Q_o(n; \mathbf{Z}) = \begin{cases} n & n \le \mathbf{Z} \\ 2\sqrt{n}Q_o(\sqrt{n}; \mathbf{Z}) + \Theta(n) & \text{otherwise} \end{cases}$$

That is, at each stage one performs $\sqrt{n}$ DFTs of size $\sqrt{n}$ (recursively), then multiplies by the Θ(n)Θ(n)" role="presentation" style="position:relative;" tabindex="0">$\Theta(n)$ twiddle factors (and do

nn" role="presentation" style="position:relative;" tabindex="0">

$$Q_o(n; \mathbf{Z}) = \Theta(n \log_{\mathbf{Z}} n)$$

the same as the optimal cache complexity equation!

These algorithms illustrate the basic features of most optimal cache-oblivious algorithms: they employ a recursive divide-and-conquer strategy to subdivide the problem until it fits into cache, at which p

Cache-obliviousness in practice

Even though the radix-$\sqrt{n}$ algorithm is optimal cache-oblivious, it does not follow that FFT implementation is a solved problem. The optimality is only in an asymptotic sense, ignoring constant factors,

Perhaps most importantly, one needs to perform an optimization that has almost nothing to do with the caches: the recursion must be "coarsened" to amortize the function-call overhead and to enable cor

One might get the impression that there is a strict dichotomy that divides cache-aware and cache-oblivious algorithms, but the two are not mutually exclusive in practice. Given an implementation of a ca

Memory strategies in FFTW

The recursive cache-oblivious strategies described above form a useful starting point, but FFTW supplements them with a number of additional tricks, and also exploits cache-obliviousness in less-obvio

We currently find that the general radix-$\sqrt{n}$ algorithm is beneficial only when n becomes very large, on the order of 220≈106220≈106" role="presentation" style="position:relative;" tabindex="

Thus, for more moderate n, FFTW uses depth-first recursion with a bounded radix, similar in spirit to the algorithm of Pre but with much larger radices (radix 32 is common) and base cases (size 32 or 6

For small n (including the radix butterflies and the base cases of the recursion), hard-coded FFTs (FFTW's **codelets**) are employed. However, this gives rise to an interesting problem: a codelet for (e.g.)

(When implementing hard-coded base cases, there is another choice because a loop of small transforms is always required. Is it better to implement a hard-coded FFT of size 64, for example, or an unrol

In addition, there are many other techniques that FFTW employs to supplement the basic recursive strategy, mainly to address the fact that cache implementations strongly favor accessing consecutive da

Footnotes

[3] A hard disk is utilized by "out-of-core" FFT algorithms for very large nn" role="presentation" style="position:relative;" tabindex="0">n, but these algorithms appear to have been largely superseded in

[4] This includes the registers: on current "x86" processors, the user-visible instruction set (with a small number of floating-point registers) is internally translated at runtime to RISC-like "μ-ops" with a m

[5] More generally, one can assume that a **cache line** of LL" role="presentation" style="position:relative;" tabindex="0"> consecutive data items are loaded into the cache at once, in order to exploit spat

[6] Of course, $O(n)$ additional storage may be required for twiddle factors, the output data (if the FFT is not in-place), and so on, but these only affect the nn" role="presentation" style="position:relative;"

[7] This advantage of depth-first recursive implementation of the radix-2 FFT was pointed out many years ago by Singleton (where the "cache" was core memory)

[8] In principle, it might be possible for a compiler to automatically coarsen the recursion, similar to how compilers can partially unroll loops. We are currently unaware of any general-purpose compiler th

[9] One practical difficulty is that some "optimizing" compilers will tend to greatly re-order the code, destroying FFTW's optimal schedule. With GNU gcc, we circumvent this problem by using compiler

Contributor

- ContribEEBurrus

10.5: Adaptive Composition of FFT Algorithms

As alluded to several times already, FFTW implements a wide variety of FFT algorithms (mostly rearrangements of Cooley-Tukey) and selects the “best” algorithm for a given n automatically. In this section, we describe how such self-optimization is implemented, and especially how FFTW's algorithms are structured as a composition of algorithmic fragments. These techniques in FFTW are described in greater detail elsewhere, so here we will focus only on the essential ideas and the motivations behind them.

An FFT algorithm in FFTW is a composition of algorithmic steps called a **plan**. The algorithmic steps each solve a certain class of **problems** (either solving the problem directly or recursively breaking it into sub-problems of the same type). The choice of plan for a given problem is determined by a **planner** that selects a composition of steps, either by runtime measurements to pick the fastest algorithm, or by heuristics, or by loading a pre-computed plan. These three pieces: problems, algorithmic steps, and the planner, are discussed in the following subsections.

The problem to be solved

In early versions of FFTW, the only choice made by the planner was the sequence of radices, and so each step of the plan took a DFT of a given size n, possibly with discontiguous input/output, and reduced it (via a radix rr" role="presentation" style="position:relative;" tabindex="0">r) to DFTs of size n/rn/r" role="presentation" style="position:relative;" tabindex="0">n/r, which were solved recursively. That is, each st

The difficulty with our initial (n,I,ι,O,o)(n,I,ι,O,o)" role="presentation" style="position:relative;" tabindex="0">$(n, \mathbf{I}, \imath, \mathbf{O}, o)$ problem definition was that it forced each algorithmic step to addre

DFT problems in FFTW are expressed in terms of structures called I/O tensors,[10] which in turn are described in terms of ancillary structures called I/O dimensions. An **I/O dimension** d is a triple d=(n,

OO" role="presentation" style="position:relative;" tabindex="0">For simplicity, let us consider only one-dimensional DFTs, so that $\mathbf{N} = \{(n, \imath, o)\}$ implies a DFT of length n on input data with stride $\imath$

$$\mathrm{dft}\,(\mathbf{N}, \{(n, \imath, o)\} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$$

is recursively defined as a “loop” of n problems: for all 0≤k<n0≤k<n" role="presentation" style="position:relative;" tabindex="0">$0 \le k < n$, do all computations in

$$\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I} + k.\ \imath, \mathbf{O} + k.\ o)$$

The case of multi-dimensional DFTs is defined more precisely elsewhere, but essentially each I/O dimension in $mathbf{N}$ gives one dimension of the transform.

We call $mathbf{N}$ the **size** of the problem. The **rank** of a problem is defined to be the rank of its size (i.e., the dimensionality of the DFT). Similarly, we call $mathbf{V}$ the **vector size** of the problem, a

DFT problem examples

A more detailed discussion of the space of problems in FFTW can be found, but a simple understanding can be gained by examining a few examples demonstrating that the I/O tensor representation is su

As a more complicated example, suppose we have an n1×n2n1×n2" role="presentation" style="position:relative;" tabindex="0">$n_1 \times n_2$ matrix $\mathbf{X}$ stored as n1n1" role="presentation" style="pos

A size-1 DFT is simply a copy Y[0]=X[0]Y[0]=X[0]" role="presentation" style="position:relative;" tabindex="0">$\mathbf{Y}[0] = \mathbf{X}[0]$, and here this can also be denoted by N=N=" role="presentation" style="

The space of plans in FFTW

Here, we describe a subset of the possible plans considered by FFTW; while not exhaustive, this subset is enough to illustrate the basic structure of FFTW and the necessity of including the vector loop(s

Roughly speaking, to solve a general DFT problem, one must perform three tasks. First, one must reduce a problem of arbitrary vector rank to a set of loops nested around a problem of vector rank 0, i.e.

Rank-0 plans

The rank-0 problem dft(,V,I,O)dft(,V,I,O)" role="presentation" style="position:relative;" tabindex="0">$\mathrm{dft}(\{\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ denotes a permutation of the input array into the output array. FFTW does not s

- When $|\mathbf{V}| = 1$ and $\mathbf{I} \neq 0$, FFTW produces a plan that copies the input array into the output array. Depending on the strides, the plan consists of a loop or, possibly, of a call to the ANSI C function
- When $|\mathbf{V}| = 2$ and $\mathbf{I} = 0$, and the strides denote a matrix-transposition problem, FFTW creates a plan that transposes the array in-place. FFTW implements the square transposition $\mathrm{dft}(\{\}, \{(n, \imath, o)$

Rank-1 plans

Rank-1 DFT problems denote ordinary one-dimensional Fourier transforms. FFTW deals with most rank-1 problems as follows.

Direct plans

When the DFT rank-1 problem is “small enough” (usually, $n \le 64$, FFTW produces a **direct plan** that solves the problem directly. These plans operate by calling a fragment of C code (a **codelet**) specia

Cooley-Tukey plans

For problems of the form $\mathrm{dft}(\{(n, \imath, o)\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $n = rm$, FFTW generates a plan that implements a radix-r Cooley-Tukey algorithm Review of the Cooley-Tukey FFT . Both decimation-in-tir

The most common case is a **decimation in time** (**DIT**) plan, corresponding to a **radix** r=n2r=n2" role="presentation" style="position:relative;" tabindex="0">$r = n_2$ (and thus m=n1m=n1" role="presen

Plans for higher vector ranks

These plans extract a vector loop to reduce a DFT problem to a problem of lower vector rank, which is then solved recursively. Any of the vector loops of VV" role="presentation" style="position:relativ

Formally, to solve $\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, where $\mathbf{V} = \{(n, \imath, o)\} \cup \mathbf{V}_1$, FFTW generates a loop that, for all k such that 0≤k<n0≤k<n" role="presentation" style="position:relative;" tabindex="0

Indirect plans

Indirect plans transform a DFT problem that requires some data shuffling (or discontiguous operation) into a problem that requires no shuffling plus a rank-0 problem that performs the shuffling.

Formally, to solve dft(N,V,I,O)dft(N,V,I,O)" role="presentation" style="position:relative;" tabindex="0">$\mathrm{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $|\mathbf{N}| > 0$, FFTW generates a plan that first solves \[\text{dft}(\left \{ \rig

Plans for prime sizes

As discussed in Goals and Background of the FFTW Project it turns out to be surprisingly useful to be able to handle large prime n (or large prime factors). **Rader plans** implement the algorithm from a

Discussion

Although it may not be immediately apparent, the combination of the recursive rules in **The space of plans in FFTW** above, can produce a number of useful algorithms. To illustrate these compositions

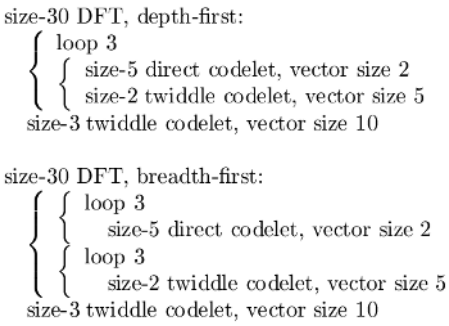


Fig. 10.5.1 Two possible decompositions for a size-30 DFT, both for the arbitrary choice of DIT radices 3 then 2 then 5, and prime-size codelets. Items grouped by a " " result from the plan for a single s

As discussed previously in sections Review of the Cooley-Tukey FFT and Understanding FFTs with an ideal cache, the same Cooley-Tukey decomposition can be executed in either traditional breadth-f

Another example of the effect of loop reordering is a style of plan that we sometimes call **vector recursion** (unrelated to “vector-radix” FFTs). The basic idea is that, if one has a loop (vector-rank 1) of t

In-place 1d transforms (with no separate bit reversal pass) can be obtained as follows by a combination DIT and DIF plans **Cooley-Tukey plans** with transposes "Rank-0 plans". First, the transform is de

The FFTW planner

Given a problem and a set of possible plans, the basic principle behind the FFTW planner is straightforward: construct a plan for each applicable algorithmic step, time the execution of these plans, and s

A direct implementation of this approach, however, faces an exponential explosion of the number of possible plans, and hence of the planning time, as mm" role="presentation" style="position:relative;"

Alternatively, there is an **estimate mode** that performs no timing measurements whatsoever, but instead minimizes a heuristic cost function. This can reduce the planner time by several orders of magnitu

Footnote

[10] I/O tensors are unrelated to the tensor-product notation used by some other authors to describe FFT algorithms.

nn" role="presentation" style="position:relative;" tabindex="0">Contributor

- ContribEEBurrus

10.6: Generating Small FFT Kernels

The base cases of FFTW's recursive plans are its **codelets**, and these form a critical component of FFTW's performance. They consist of long blocks of highly optimized, straight-line code, implementing many special cases of the DFT that give the planner a large space of plans in which to optimize. Not only was it impractical to write numerous codelets by hand, but we also needed to rewrite them many times in order to explore different algorithms and optimizations. Thus, we designed a special-purpose "FFT compiler" called **genfft** that produces the codelets automatically from an abstract description.

A typical codelet in FFTW computes a DFT of a small, fixed size n (usually, $n \leq 64$ possibly with the input or output multiplied by twiddle factors Cooley-Tukey plans. Several other kinds of codelets can be produced by genfft , but we will focus here on this common case.

In principle, all codelets implement some combination of the Cooley-Tukey algorithm from the equation and/or some other DFT algorithm expressed by a similarly compact formula. However, a high-performance implementation of the DFT must address many more concerns than the equation alone suggests. For example, the equation contains multiplications by 1 that are more efficient to omit. The equation entails a run-time factorization of n which can be precomputed if n is known in advance. The equation operates on complex numbers, but breaking the complex-number abstraction into real and imaginary components turns out to expose certain non-obvious optimizations. Additionally, to exploit the long pipelines in current processors, the recursion implicit in the equation should be unrolled and re-ordered to a significant degree. Many further optimizations are possible if the complex input is known in advance to be purely real (or imaginary). Our design goal for genfft was to keep the expression of the DFT algorithm independent of such concerns. This separation allowed us to experiment with various DFT algorithms and implementation strategies independently and without (much) tedious rewriting.

Genfft is structured as a compiler whose input consists of the kind and size of the desired codelet, and whose output is C code. genfft operates in four phases: creation, simplification, scheduling, and unparsing.

In the **creation** phase, genfft produces a representation of the codelet in the form of a directed acyclic graph (**dag**). The dag is produced according to well-known DFT algorithms: Cooley-Tukey equation, prime-factor, split-radix and Rader. Each algorithm is expressed in a straightforward math-like notation, using complex numbers, with no attempt at optimization. Unlike a normal FFT implementation, however, the algorithms here are evaluated symbolically and the resulting symbolic expression is represented as a dag, and in particular it can be viewed as a **linear network** (in which the edges represent multiplication by constants and the vertices represent additions of the incoming edges).

In the **simplification** phase, genfft applies local rewriting rules to each node of the dag in order to simplify it. This phase performs algebraic transformations (such as eliminating multiplications by 1) and common-subexpression elimination. Although such transformations can be performed by a conventional compiler to some degree, they can be carried out here to a greater extent because genfft can exploit the specific problem domain. For example, two equivalent subexpressions can always be detected, even if the subexpressions are written in algebraically different forms, because all subexpressions compute linear functions. Also, genfft can exploit the property that **network transposition** (reversing the direction of every edge) computes the transposed linear operation, in order to transpose the network, simplify, and then transpose back—this turns out to expose additional common subexpressions. In total, these simplifications are sufficiently powerful to derive DFT algorithms specialized for real and/or symmetric data automatically from the complex algorithms. For example, it is known that when the input of a DFT is real (and the output is hence conjugate-symmetric), one can save a little over a factor of two in arithmetic cost by specializing FFT algorithms for this case—with genfft , this specialization can be done entirely automatically, pruning the redundant operations from the dag, to match the lowest known operation count for a real-input FFT starting only from the complex-data algorithm. We take advantage of this property to help us implement real-data DFTs, to exploit machine-specific "SIMD" instructions SIMD instructions , and to generate codelets for the discrete cosine (DCT) and sine (DST) transforms. Furthermore, by experimentation we have discovered additional simplifications that improve the speed of the generated code. One interesting example is the elimination of negative constants: multiplicative constants in FFT algorithms often come in positive/negative pairs, but every C compiler we are aware of will generate separate load instructions for positive and negative versions of the same constants.[11] We thus obtained a 10–15% speedup by making all constants positive, which involves propagating minus signs to change additions into subtractions or vice versa elsewhere in the dag (a daunting task if it had to be done manually for tens of thousands of lines of code).

In the **scheduling** phase, genfft produces a topological sort of the dag (a **schedule**). The goal of this phase is to find a schedule such that a C compiler can subsequently perform a good register allocation. The scheduling algorithm used by genfft offers certain theoretical guarantees because it has its foundations in the theory of cache-oblivious algorithms (here, the registers are viewed as a

form of cache), as described in Memory strategies in FFTW . As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by SPIRAL.

In the stock genfft implementation, the schedule is finally unparsed to C. A variation from this implements the rest of a compiler back end and outputs assembly code.

Footnote

[11] Floating-point constants must be stored explicitly in memory; they cannot be embedded directly into the CPU instructions like integer “immediate” constants.

Contributor

- ContribEEBurrus

10.7: SIMD instructions

Unfortunately, it is impossible to attain nearly peak performance on current popular processors while using only portable C code. Instead, a significant portion of the available computing power can only be accessed by using specialized SIMD (single-instruction multiple data) instructions, which perform the same operation in parallel on a data vector. For example, all modern “x86” processors can execute arithmetic instructions on “vectors” of four single-precision values (SSE instructions) or two double-precision values (SSE2 instructions) at a time, assuming that the operands are arranged consecutively in memory and satisfy a 16-byte alignment constraint. Fortunately, because nearly all of FFTW's low-level code is produced by genfft , machine-specific instructions could be exploited by modifying the generator—the improvements are then automatically propagated to all of FFTW's codelets, and in particular are not limited to a small set of sizes such as powers of two.

SIMD instructions are superficially similar to “vector processors”, which are designed to perform the same operation in parallel on an all elements of a data array (a “vector”). The performance of “traditional” vector processors was best for long vectors that are stored in contiguous memory locations, and special algorithms were developed to implement the DFT efficiently on this kind of hardware. Unlike in vector processors, however, the SIMD vector length is small and fixed (usually 2 or 4). Because microprocessors depend on caches for performance, one cannot naively use SIMD instructions to simulate a long-vector algorithm: while on vector machines long vectors generally yield better performance, the performance of a microprocessor drops as soon as the data vectors exceed the capacity of the cache. Consequently, SIMD instructions are better seen as a restricted form of instruction-level parallelism than as a degenerate flavor of vector parallelism, and different DFT algorithms are required.

The technique used to exploit SIMD instructions in genfft is most easily understood for vectors of length two (e.g., SSE2). In this case, we view a **complex** DFT as a pair of **real** DFTs:

$$\mathrm{DFT}(A+i.B) = \mathrm{DFT}(A)+i.\mathrm{DFT}(B)$$

where A and B are two real arrays. Our algorithm computes the two real DFTs in parallel using SIMD instructions, and then it combines the two outputs according to the equation. This SIMD algorithm has two important properties. First, if the data is stored as an array of complex numbers, as opposed to two separate real and imaginary arrays, the SIMD loads and stores always operate on correctly-aligned contiguous locations, even if the the complex numbers themselves have a non-unit stride. Second, because the algorithm finds two-way parallelism in the real and imaginary parts of a single DFT (as opposed to performing two DFTs in parallel), we can completely parallelize DFTs of any size, not just even sizes or powers of 2.

Contributor

- ContribEEBurrus

10.8: Numerical Accuracy in FFTs

An important consideration in the implementation of any practical numerical algorithm is numerical accuracy: how quickly do floating-point roundoff errors accumulate in the course of the computation? Fortunately, FFT algorithms for the most part have remarkably good accuracy characteristics. In particular, for a DFT of length n computed by a Cooley-Tukey algorithm with finite-precision floating-point arithmetic, the **worst-case** error growth is O(logn)O(logn)" role="presentation" style="position:relative;" tabindex="0">$O(\log n)$ and the mean error growth for random inputs is only $O(\sqrt{\log n})$. This is so good that, in practical applications, a

The amazingly small roundoff errors of FFT algorithms are sometimes explained incorrectly as simply a consequence of the reduced number of operations: since there are fewer operations compared to a

However, these encouraging error-growth rates **only** apply if the trigonometric "twiddle" factors in the FFT algorithm are computed very accurately. Many FFT implementations, including FFTW and cc

O(n)O(n)" role="presentation" style="position:relative;" tabindex="0">There are, in fact, trigonometric recurrences with the same logarithmic error growth as the FFT, but these seem more difficult to in

There are a few non-Cooley-Tukey algorithms that are known to have worse error characteristics, such as the "real-factor" algorithm but these are rarely used in practice (and are not used at all in FFTW

Footnote

[12] In an FFT, the twiddle factors are powers of ωnωn" role="presentation" style="position:relative;" tabindex="0">ω_n, so θθ" role="presentation" style="position:relative;" tabindex="0">

Contributor

- ContribEEBurrus

10.9: Concluding Remarks

It is unlikely that many readers of this chapter will ever have to implement their own fast Fourier transform software, except as a learning exercise. The computation of the DFT, much like basic linear algebra or integration of ordinary differential equations, is so central to numerical computing and so well-established that robust, flexible, highly optimized libraries are widely available, for the most part as free/open-source software. And yet there are many other problems for which the algorithms are not so finalized, or for which algorithms are published but the implementations are unavailable or of poor quality. Whatever new problems one comes across, there is a good chance that the chasm between theory and efficient implementation will be just as large as it is for FFTs, unless computers become much simpler in the future. For readers who encounter such a problem, we hope that these lessons from FFTW will be useful:

- Generality and portability should almost always come first.
- The number of operations, up to a constant factor, is less important than the order of operations.
- Recursive algorithms with large base cases make optimization easier.
- Optimization, like any tedious task, is best automated.
- Code generation reconciles high-level programming with low-level performance.

We should also mention one final lesson that we haven't discussed in this chapter: you can't optimize in a vacuum, or you end up congratulating yourself for making a slow program slightly faster. We started the FFTW project after downloading a dozen FFT implementations, benchmarking them on a few machines, and noting how the winners varied between machines and between transform sizes. Throughout FFTW's development, we continued to benefit from repeated benchmarks against the dozens of high-quality FFT programs available online, without which we would have thought FFTW was “complete” long ago.

Acknowledgements

SGJ was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334; MF was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. We are also grateful to Sidney Burrus for the opportunity to contribute this chapter, and for his continual encouragement—dating back to his first kind words in 1997 for the initial FFT efforts of two graduate students venturing outside their fields.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

11: Algorithms for Data with Restrictions

Topic hierarchy

11.1: Introduction

Learning Objectives

- Study various approaches to developing special algorithms or to modifying complex algorithms for real data

Many applications involve processing real data. It is inefficient to simply use a complex FFT on real data because arithmetic would be performed on the zero imaginary parts of the input, and, because of symmetries, output values would be calculated that are redundant. There are several approaches to developing special algorithms or to modifying complex algorithms for real data.

In the next section we will take a look at all such approaches.

Contributor

- ContribEEBurrus

11.2: Various Approaches to Developing Special Methods

There are two methods which use a complex FFT in a special way to increase efficiency. The first method uses a length-N complex FFT to compute two length-N real FFTs by putting the two real data sequences into the real and the imaginary parts of the input to a complex FFT. Because transforms of real data have even real parts and odd imaginary parts, it is possible to separate the transforms of the two inputs with 2N-4 extra additions. This method requires, however, that two inputs be available at the same time.

The second method uses the fact that the last stage of a decimation-in-time radix-2 FFT combines two independent transforms of length N/2 to compute a length-N transform. If the data are real, the two half length transforms are calculated by the method described above and the last stage is carried out to calculate the total length-N FFT of the real data. It should be noted that the half-length FFT does not have to be calculated by a radix-2 FFT. In fact, it should be calculated by the most efficient complex-data algorithm possible, such as the SRFFT or the PFA. The separation of the two half-length transforms and the computation of the last stage requires $N-6$ real multiplications and (5/2)N-6(5/2)N-6" role="presentation" style="position:relative;" tabindex="0">$(5/2)N-6$ real additions.

It is possible to derive more efficient real-data algorithms directly rather than using a complex FFT. The basic idea is from Bergland and Sande which, at each stage, uses the symmetries of a constant rac

Special versions of both the PFA and WFTA can also be developed for real data. Because the operations in the stages of the PFA can be commuted, it is possible to move the combination of the transforn

Contributor

- ContribEEBurrus

11.3: Special Algorithms for input Data that is mostly Zero

In some cases, most of the data to be transformed are zero. It is clearly wasteful to do arithmetic on that zero data. Another special case is when only a few DFT values are needed. It is likewise wasteful to calculate outputs that are not needed. We use a process called “pruning" to remove the unneeded operations.

In other cases, the data are non-uniform sampling of a continuous time signal.

There are certain applications where approximations to the DFT are all that is needed.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

12: Convolution Algorithms

Topic hierarchy

12.1: Introduction

Learning Objectives

- Learn methods to do convolution by FFT more efficiently.

One of the main applications of the FFT is to do convolution more efficiently than the direct calculation from the definition which is:

$$y(n) = \sum h(m)x(n-m)$$

which, with a change of variables, can also be written as:

$$y(n) = \sum x(m)h(n-m)$$

This is often used to filter a signal x(n)x(n)" role="presentation" style="position:relative;" tabindex="0">$x(n)$ with a filter whose impulse response is x(n)x(n)" role="presentation" style="position:relative;" tabindex="0">$h(n)$h(n)h(n)" r
Because the DFT converts convolution to multiplication:

$$\mathrm{DFT}y(n) = \mathrm{DFT}\left\{h(n)\mathrm{DFT}x(n)\right\}$$

can be calculated with the FFT and bring the order of arithmetic operations down to Nlog(N)Nlog(N)" role="presentation" style="position:relative;" tabindex="0">$N\log(N)$ which can be significant fo
NN" role="presentation" style="position:relative;" tabindex="0">This approach, which is called “fast convolutions", is a form of block processing since a whole block or segment of x(n)x(n)" role="pres
For filtering and some other applications, one wants “on going" convolution where the filter response x(n)x(n)" role="presentation" style="position:relative;" tabindex="0">$h(n)$ may be finite in length

h(n)h(n)" role="presentation" style="position:relative;" tabindex="0">Contributor

- ContribEEBurrus

12.2: Fast Convolution by Overlap-Add and Overlap-Save

Fast Convolution by Overlap-Add

In order to use the FFT to convolve (or filter) a long input sequence x(n)x(n)" role="presentation" style="position:relative;" tabindex="0">$x(n)$ with a finite length-M impulse response, x(n)x(n)" role="presentation" style="position:relative;" tabindex="0">$h(n)$, we part

The reason this procedure is not totally straightforward, is the length of the output of convolving a length-L block with a length-M filter is of length $L+M-1$. This means the output blocks cannot sin

The second issue that must be taken into account is the fact that the overlap-add steps need non-cyclic convolution and convolution by the FFT is cyclic. This is easily handled by appending L-1L-1" role

The savings in arithmetic can be considerable when implementing convolution or performing FIR digital filtering. However, there are two penalties. The use of blocks introduces a delay of one block len

The efficiency in terms of number of arithmetic operations per output point increases for large blocks because of the $M\log(M)$ requirements of the FFT. However, the blocks become very large ($L>>$

LL" role="presentation" style="position:relative;" tabindex="0">Usually, the block convolutions are done by the FFT, but they could be done by any efficient, finite length method. One could use “recta

LL" role="presentation" style="position:relative;" tabindex="0">Fast Convolution by Overlap-SaveLL" role="presentation" style="position:relative;" tabindex="

An alternative approach to the Overlap-Add can be developed by starting with segmenting the output rather than the input. If one considers the calculation of a block of output, it is seen that not only the

Contributor

- ContribEEBurrus

12.3: Block Processing - a Generalization of Overlap Methods

Convolution is intimately related to the DFT. It was shown in The DFT as Convolution or Filtering that a prime length DFT could be converted to cyclic convolution. It has been long known that convolution can be calculated by multiplying the DFTs of signals.

An important question is what is the fastest method for calculating digital convolution. There are several methods that each have some advantage. The earliest method for fast convolution was the use of sectioning with overlap-add or overlap-save and the FFT. In most cases the convolution is of real data and, therefore, real-data FFTs should be used. That approach is still probably the fastest method for longer convolution on a general purpose computer or microprocessor. The shorter convolutions should simply be calculated directly.

Introduction

The partitioning of long or infinite strings of data into shorter sections or blocks has been used to allow application of the FFT to realize on-going or continuous convolution. This section develops the idea of block processing and shows that it is a generalization of the overlap-add and overlap-save methods. They further generalize the idea to a multidimensional formulation of convolution. Moving in the opposite direction, it is shown that, rather than partitioning a string of scalars into blocks and then into blocks of blocks, one can partition a scalar number into blocks of bits and then include the operation of multiplication in the signal processing formulation. This is called distributed arithmetic and since it describes operations at the bit level, is completely general. These notes try to present a coherent development of these ideas.

Block Signal Processing

In this section the usual convolution and recursion that implements FIR and IIR discrete-time filters are reformulated in terms of vectors and matrices. Because the same data is partitioned and grouped in a variety of ways, it is important to have a consistent notation in order to be clear. The nthnth" role="presentation" style="position:relative;" tabindex="0">n^{th} element of a data sequence is expressed h(n)h(n)" role="presentation" style="position:relative;" tabindex="0">$h(n)$ or, in some c

Block Convolution

The operation of a finite impulse response (FIR) filter is described by a finite convolution as

$$y(n) = \sum_{k=0}^{L-1} h(k)x(n-k)$$

where $x(n)$ is causal, $h(n)$ is causal and of length LL" role="presentation" style="position:relative;" tabindex="0">L, and the time index LL" role="presentation" style="position:relative;" tabindex="0'

$$y(n) = \sum h(n-k)x(k)$$

which can be expressed as a matrix operation by

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} h_0 & 0 & 0 & \cdots & 0 \\ h_1 & h_0 & 0 & & \\ h_2 & h_1 & h_0 & & \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

The H matrix of impulse response values is partitioned into N by N square sub matrices and the X and Y vectors are partitioned into length-N blocks or sections. This is illustrated for $N = 3$ by

$$H_0 = \begin{bmatrix} h_0 & 0 & 0 \\ h_1 & h_0 & 0 \\ h_2 & h_1 & h_0 \end{bmatrix} \quad H_1 = \begin{bmatrix} h_3 & h_2 & h_1 \\ h_4 & h_3 & h_2 \\ h_5 & h_4 & h_3 \end{bmatrix} \quad \text{etc.}$$

$$\underline{x}_0 = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \quad \underline{x}_1 = \begin{bmatrix} x_3 \\ x_4 \\ x_5 \end{bmatrix} \quad \underline{y}_0 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} \quad \text{etc.}$$

Substituting these definitions into the equation gives

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} H_0 & 0 & 0 & \cdots & 0 \\ H_1 & H_0 & 0 & & \\ H_2 & H_1 & H_0 & & \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

The general expression for the n^{th} output block is

$$\underline{y}_n = \sum_{k=0}^{n} H_{n-k}\underline{x}_k$$

which is a vector or block convolution. Since the matrix-vector multiplication within the block convolution is itself a convolution, the equation is a sort of convolution of convolutions and the finite leng

The equation for one output block can be written as the product

$$\underline{y}_2 = \begin{bmatrix} H_2 & H_1 & H_0 \end{bmatrix} \begin{bmatrix} \underline{x}_0 \\ \underline{x}_1 \\ \underline{x}_2 \end{bmatrix}$$

and the effects of one input block can be written

$$\begin{bmatrix} H_0 \\ H_1 \\ H_2 \end{bmatrix} \underline{x}_1 = \begin{bmatrix} \underline{y}_0 \\ \underline{y}_1 \\ \underline{y}_2 \end{bmatrix}$$

These are generalize statements of overlap save and overlap ad. The block length can be longer, shorter, or equal to the filter length.

Block Recursion

Although less well-known, IIR filters can also be implemented with block processing. The block form of an IIR filter is developed in much the same way as for the block convolution implementation of

$$y(n) = \sum_{l=1}^{N-1} a_l y_{n-l} + \sum_{k=0}^{M-1} b_k x_{n-k}$$

using both functional notation and subscripts, depending on which is easier and clearer. The impulse response $h(n)$ is

$$y(n) = \sum_{l=1}^{N-1} a_l h(n-l) + \sum_{k=0}^{M-1} b_k \delta(n-k)$$

which can be written in matrix operator form

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_1 & 1 & 0 & & \\ a_2 & a_1 & 1 & & \\ a_3 & a_2 & a_1 & & \\ 0 & a_3 & a_2 & & \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ 0 \\ \vdots \end{bmatrix}$$

In terms of *N* by *N* submatrices and length-*N* blocks, this becomes

$$\begin{bmatrix} A_0 & 0 & 0 & \cdots & 0 \\ A_1 & A_0 & 0 & & \\ 0 & A_1 & A_0 & & \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} \underline{y}_0 \\ \underline{y}_1 \\ \underline{y}_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} B_0 & 0 & 0 & \cdots & 0 \\ B_1 & B_0 & 0 & & \\ 0 & B_1 & B_0 & & \\ \vdots & & & & \vdots \end{bmatrix} \begin{bmatrix} \underline{x}_0 \\ \underline{x}_1 \\ \underline{x}_2 \\ \vdots \end{bmatrix}$$

From the partitioned rows of the equation, one can write the block recursive relation

$$A_0 \underline{y}_{n+1} + A_1 \underline{y}_n = B_0 \underline{x}_{n+1} + B_1 \underline{x}_n$$

Solving for $\underline{y}_{n+1}$ gives

$$\underline{y}_{n+1} = -A_0^{-1} A_1 \underline{y}_n + A_0^{-1} B_0 \underline{x}_{n+1} + A_0^{-1} B_1 \underline{x}_n$$

$$\underline{y}_{n+1} = K \underline{y}_n + H_0 \underline{x}_{n+1} + \tilde{H}_1 \underline{x}_n$$

which is a first order vector difference equation. This is the fundamental block recursive algorithm that implements the original scalar difference equation in the equation. It has several important charact

- The block recursive formulation is similar to a state variable equation but the states are blocks or sections of the output.
- The eigenvalues of *K* are the poles of the original scalar problem raised to the *N* power plus others that are zero. The longer the block length, the “more stable" the filter is, i.e. the further the poles are
- If the block length were shorter than the denominator, the vector difference equation would be higher than first order. There would be a non zero A_2 . If the block length were shorter than the numerat
- The actual arithmetic that goes into the calculation of the output is partly recursive and partly convolution. The longer the block, the more the output is calculated by convolution and, the more arithm
- It is possible to remove the zero eigenvalues in *K* by making *K* rectangular or square and N by N This results in a form even more similar to a state variable formulation. This is briefly discussed belo
- There are several ways of using the FFT in the calculation of the various matrix products in the equations. Each has some arithmetic advantage for various forms and orders of the original equation. I
- By choosing the block length equal to the period, a periodically time varying filter can be made block time invariant. In other words, all the time varying characteristics are moved to the finite matrix

Block State Formulation

It is possible to reduce the size of the matrix operators in the block recursive description equation to give a form even more like a state variable equation. If *K* in the equation has several zero eigenvalues

$$\underline{z}_n = K_1 \underline{z}_{n-1} + K_2 \underline{x}_n$$

$$\underline{y}_n = H_1 \underline{z}_{n-1} + H_0 \underline{x}_n$$

where H_0 is the same *N* by *N* convolution matrix, N_1 is a rectangular *L* by *N* partition of the convolution matrix *H* , K_1 is a square *N* by *N* matrix of full rank, and K_2 is a rectangular *N* by *L* matrix.

This is now a minimal state equation whose input and output are blocks of the original input and output. Some of the matrix multiplications can be carried out using the FFT or other techniques.

Block Implementations of Digital Filters

The advantage of the block convolution and recursion implementations is a possible improvement in arithmetic efficiency by using the FFT or other fast convolution methods for some of the multiplicati

These methods could also be used in the various filtering methods for evaluating the DFT. This the chirp z-transform, Rader's method, and Goertzel's algorithm.

Multidimensional Formulation

This process of partitioning the data vectors and the operator matrices can be continued by partitioning the equations and creating blocks of blocks to give a higher dimensional structure. One should use

Periodically Time-Varying Discrete-Time Systems

Most time-varying systems are periodically time-varying and this allows special results to be obtained. If the block length is set equal to the period of the time variations, the resulting block equations are

Multirate Filters, Filter Banks, and Wavelets

Another area that is related to periodically time varying systems and to block processing is filter banks. Recently the area of perfect reconstruction filter banks has been further developed and shown to b

Parks has noted that design of multirate filters has some elements in common with complex approximation and of 2-D filter design and is looking at using Tang's method for these designs.

Distributed Arithmetic

Rather than grouping the individual scalar data values in a discrete-time signal into blocks, the scalar values can be partitioned into groups of bits. Because multiplication of integers, multiplication of po

Contributor

- ContribEEBurrus

12.4: Direct Fast Convolution and Rectangular Transforms

A relatively new approach uses index mapping directly to convert a one dimensional convolution into a multidimensional convolution. This can be done by either a type-1 or type-2 map. The short convolutions along each dimension are then done by Winograd's optimal algorithms. Unlike for the case of the DFT, there is no savings of arithmetic from the index mapping alone. All the savings comes from efficient short algorithms. In the case of index mapping with convolution, the multiplications must be nested together in the center of the algorithm in the same way as for the WFTA. There is no equivalent to the PFA structure for convolution. The multidimensional convolution can not be calculated by row and column convolutions as the DFT was by row and column DFTs.

It would first seem that applying the index mapping and optimal short algorithms directly to convolution would be more efficient than using DFTs and converting them to convolution to be calculated by the same optimal algorithms. In practical algorithms, however, the DFT method seems to be more efficient.

A method that is attractive for special purpose hardware uses distributed arithmetic. This approach uses a table look up of precomputed partial products to produce a system that does convolution without requiring multiplications.

Another method that requires special hardware uses number theoretic transforms to calculate convolution. These transforms are defined over finite fields or rings with arithmetic performed modulo special numbers. These transforms have rather limited flexibility, but when they can be used, they are very efficient.

Contributor

- ContribEEBurrus

12.5: Number Theoretic Transforms for Convolution

Here we look at the conditions placed on a general linear transform in order for it to support cyclic convolution. The form of a linear transformation of a length-N sequence of number is given by

$$X(k) = \sum_{n=0}^{N-1} t(n,k)x(n)$$

for $k = 0, 1, \ldots, (N-1)$. The definition of cyclic convolution of two sequences is given by

$$y(n) = \sum_{m=0}^{N-1} x(m)h(n-m)$$

for $n = 0, 1, \ldots, (N-1)$ and all indices evaluated modulo N. We would like to find the properties of the transformation such that it will support the cyclic convolution. This means that if $X(k),\ H(k),\ Y(k)$ are the transforms of $x(n),\ h(n),\ y(n)$ respectively,

$$Y(k) = X(k)H(k)$$

The conditions are derived by taking the transform defined in the above equations of both sides of the equation which gives

$$Y(k) = \sum_{n=0}^{N-1} t(n,k) \sum_{m=0}^{N-1} x(m)h(n-m)$$

$$Y(k) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m)h(n-m)t(n,k)$$

Making the change of index variables, $l = n - m$ gives

$$Y(k) = \sum_{m=0}^{N-1} \sum_{l=0}^{N-1} x(m)h(l)t(l+m,k)$$

But from the equation, this must be

$$Y(k) = \sum_{n=0}^{N-1} x(n)t(n,k) \sum_{m=0}^{N-1} x(m)t(m,k)$$

$$Y(k) = \sum_{m=0}^{N-1} \sum_{l=0}^{N-1} x(m)h(l)t(n,k)t(l,k)$$

This must be true for all \(x(n),\ h(n) and k, therefore from the above equations we have

$$t(m+l,k) = t(m,k)t(l,k)$$

For $l = 0$ we have

$$t(m,k) = t(m,k)t(0,k)$$

Therefore, $t(0,k) = 1$. For $l = m$ we have

$$t(2m,k) = t(m,k)t(m,k) = t^2(m,k)$$

For $l = pm$ we likewise have

$$t(pm,k) = t^p(m,k)$$

Therefore,

$$t^N(m,k) = t(Nm,k) = t(0,k) = 1$$

But

$$t(m,k) = t^m(1,k) = t^k(m,1)$$

Therefore,

$$t(m,k) = t^{mk}(1,1)$$

Defining $t(1,1) = \alpha$ gives the form for our general linear transform equation as

$$X(k) = \sum_{n=0}^{N-1} \alpha^{nk} x(n)$$

where α is a root of order NNN" role="presentation" style="position:relative;" tabindex="0">, which means that NNN" role="presentation" style="position:relative;" tabindex="0"> is the smallest integer such that αN=1

Theorem 1 The transform equation supports cyclic convolution if and only if ααα" role="presentation" style="position:relative;" tabindex="0"> is a root of order N and N^{-1} is defined.

Theorem 2 The transform equation supports cyclic convolution if and only if

$$N \mid O(M)$$

where

$$O(M) = \gcd\ \{p_1 - 1, p_2 - 1, \ldots, p_l - 1\}$$

and

$$M = p_1^{r_1} p_2^{r_2} \ldots p_l^{r_l}$$

This theorem is a more useful form of Theorem 1. Notice that Nmax=O(M)Nmax=O(M)" role="presentation" style="position:relative;" tabindex="0">$N_{max} = O(M)$.

One needs to find appropriate N, M and αNN" role="presentation" style="position:relative;" tabindex="0"> such that

- N should be appropriate for a fast algorithm and handle the desired sequence lengths.
- M should allow the desired dynamic range of the signals and should allow simple modular arithmetic.
- α should allow a simple multiplication for αnkx(n)αnkx(n)" role="presentation" style="position:relative;" tabindex="0">$\alpha^{nk} x(n)$.

We see that if MM" role="presentation" style="position:relative;" tabindex="0">M is even, it has a factor of 2 and, therefore $O(M) = N_{max} = 1$ which implies MM" role="presentation" style="positio
2p2p" role="presentation" style="position:relative;" tabindex="0">For $M = 2^k + 1$ and k odd, 3 divides $2^k + 1$ and the maximum possible transform length is 2. Thus we consider only even k. Let $k =$
Since Fermat numbers up to F4F4" role="presentation" style="position:relative;" tabindex="0">F_4 are prime, $O(F_t) = 2^b$ where $b = 2^t$ and we can have a Fermat number transform for any length $N =$
The following table gives possible parameters for various Fermat number moduli.

t	b	$M = F_t$	N_2	$N_{\sqrt{2}}$	N_{max}	α for N_{max}
3	8	$2^8 + 1$	16	32	256	3
4	16	$2^{16} + 1$	32	64	65536	3
5	32	$2^{32} + 1$	64	128	128	$\sqrt{2}$
6	64	$2^{64} + 1$	128	256	256	$\sqrt{2}$

Table 12.5.1 Fermat number moduli

This table gives values of NN" role="presentation" style="position:relative;" tabindex="0">N for the two most important values of αα" role="presentation" style="position:relative;" tabindex="(

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

13: Comments and Conclusion

Topic hierarchy

13.1: Comments

This section comes from a note describing results on efficient algorithms to calculate the discrete Fourier transform (DFT) that were collected over years. Perhaps the most interesting is the discovery that the Cooley-Tukey FFT was described by Gauss in 1805. That gives some indication of the age of research on the topic, and the fact that a 1995 compiled bibliography on efficient algorithms contains over 3400 entries indicates its volume. Three IEEE Press reprint books contain papers on the FFT. An excellent general purpose FFT program has been described in and is used in Matlab and available over the internet.

In addition to this book there are several others that give a good modern theoretical background for the FFT, one book that gives the basic theory plus both FORTRAN and TMS 320 assembly language programs, and other books that contain chapters on advanced FFT topics. There is a good up-to-date, on-line reference with both theory and programming techniques. The history of the FFT and excellent survey articles can also be found. The foundation of much of the modern work on efficient algorithms was done by S. Winograd.

Efficient FFT algorithms for length-2^M were described by Gauss and discovered in modern times by Cooley and Tukey. These have been highly developed and good examples of FORTRAN programs can be found. Several new algorithms have been published that require the least known amount of total arithmetic. Of these, the split-radix FFT seems to have the best structure for programming, and an efficient program has been written to implement it. A mixture of decimation-in-time and decimation-in-frequency with very good efficiency is given in and one called the Sine-Cosine FT. Recently a modification to the split-radix algorithm has been described that has a slightly better total arithmetic count. Theoretical bounds on the number of multiplications required for the FFT based on Winograd's theories and schemes for calculating an in-place, in-order radix-2 FFT can be found. Also found are various forms of unscramblers. A discussion of the relation of the computer architecture, algorithm and compiler can be found. A modification would be to allow lengths of $N = q2^m$ for q.

The “other” FFT is the prime factor algorithm (PFA) which uses an index map originally developed by Thomas and by Good. The theory of the PFA was derived in and further developed and an efficient in-order and in-place program. A method has been developed to use dynamic programming to design optimal FFT programs that minimize the number of additions and data transfers as well as multiplications. This new approach designs custom algorithms for a particular computer architecture. An efficient and practical development of Winograd's ideas has given a design method that does not require the rather difficult Chinese remainder theorem for short prime length FFT's. These ideas have been used to design modules of length 11, 13, 17, 19, and 25. Other methods for designing short DFT's can be found. A program that implements the nested Winograd Fourier transform algorithm (WFTA) is also found but it has not proven as fast or as versatile as the PFA. An interesting use of the PFA was announced in searching for large prime numbers.

These efficient algorithms can not only be used on DFT's but on other transforms with a similar structure. They have been applied to the discrete Hartley transform and the discrete cosine transform.

The fast Hartley transform has been proposed as a superior method for real data analysis but that has been shown not to be the case. A well-designed real-data FFT is always as good as or better than a well-designed Hartley transform. The Bruun algorithm also looks promising for real data applications as does the Rader-Brenner algorithm.

Apart from the general length algorithms, for lengths that are not highly composite or prime, the chirp z-transform in a good candidate for longer lengths and an efficient order-N^2 algorithm called the QFT for shorter lengths. A method which automatically generates near-optimal prime length Winograd based programs is available. This gives the same efficiency for shorter lengths (i.e. N≤19N≤19" role="presentation" style="position:relative;" tabindex="0">$N \leq 19$ and new algorithms for much longer lengths and with well-structured algorithms. Special methods are availabl

The use of the FFT to calculate discrete convolution was one of its earliest uses. Although the more direct rectangular transform would seem to be more efficient, use of the FFT or PFA is still probably t

Various approaches to calculating approximate DFTs have been based on cordic methods, short word lengths, or some form of pruning. A new method that uses the characteristics of the signals being tra

The study of efficient algorithms not only has a long history and large bibliography, it is still an exciting research field where new results are used in practical applications.

Contributor

- ContribEEBurrus

13.2: Conclusion

This book has developed a class of efficient algorithms based on index mapping and polynomial algebra. This provides a framework from which the Cooley-Tukey FFT, the split-radix FFT, the PFA, and WFTA can be derived. Even the programs implementing these algorithms can have a similar structure. Winograd's theorems were presented and shown to be very powerful in both deriving algorithms and in evaluating them. The simple radix-2 FFT provides a compact, elegant means for efficiently calculating the DFT. If some elaboration is allowed, significant improvement can be had from the split-radix FFT, the radix-4 FFT or the PFA. If multiplications are expensive, the WFTA requires the least of all.

Several method for transforming real data were described that are more efficient than directly using a complex FFT. A complex FFT can be used for real data by artificially creating a complex input from two sections of real input. An alternative and slightly more efficient method is to construct a special FFT that utilizes the symmetries at each stage.

As computers move to multiprocessors and multicore, writing and maintaining efficient programs becomes more and more difficult. The highly structured form of FFTs allows automatic generation of very efficient programs that are tailored specifically to a particular DSP or computer architecture.

For high-speed convolution, the traditional use of the FFT or PFA with blocking is probably the fastest method although rectangular transforms, distributed arithmetic, or number theoretic transforms may have a future with special VLSI hardware.

The ideas presented in these notes can also be applied to the calculation of the discrete Hartley transform, the discrete cosine transform, and to number theoretic transforms.

There are many areas for future research. The relationship of hardware to algorithms, the proper use of multiple processors, the proper design and use of array processors and vector processors are all open. There are still many unanswered questions in multi-dimensional algorithms where a simple extension of one-dimensional methods will not suffice.

Contributor

- ContribEEBurrus

CHAPTER OVERVIEW

14: Appendix

Topic hierarchy

14.1: Appendix 1 - FFT Flowgraphs

The following four figures are flow graphs for Radix-2 Cooley-Tukey FFTs. The first is a length-16, decimation-in-frequency Radix-2 FFT with the input data in order and output data scrambled. The first stage has 8 length-2 "butterflies" (which overlap in the figure) followed by 8 multiplications by powers of W which are called "twiddle factors". The second stage has 2 length-8 FFTs which are each calculated by 4 butterflies followed by 4 multiplies. The third stage has 4 length-4 FFTs, each calculated by 2 butterflies followed by 2 multiplies and the last stage is simply 8 butterflies followed by trivial multiplies by one. This flow graph should be compared with the index map in Polynomial Description of Signals , the polynomial decomposition in The DFT as Convolution or Filtering , and the program in Appendix 3. In the program, the butterflies and twiddle factor multiplications are done together in the inner most loop. The outer most loop indexes through the stages. If the length of the FFT is a power of two, the number of stages is that power (log N).

The second figure below is a length-16, decimation-in-time FFT with the input data scrambled and output data in order. The first stage has 8 length-2 "butterflies" followed by 8 twiddle factors multiplications. The second stage has 4 length-4 FFTs which are each calculated by 2 butterflies followed by 2 multiplies. The third stage has 2 length-8 FFTs, each calculated by 4 butterflies followed by 8 multiplies and the last stage is simply 8 length-2 butterflies. This flow graph should be compared with the index map in Polynomial Description of Signals, the polynomial decomposition in The DFT as Convolution or Filtering, and the program in Appendix 3. Here, the FFT must be preceded by a scrambler.

The third and fourth figures below are a length-16 decimation-in-frequency and a decimation-in-time but, in contrast to the figures above, the DIF has the output in order which requires a scrambled input and the DIT has the input in order which requires the output be unscrambled. Compare with the first two figures. Note the order of the twiddle factors. The number of additions and multiplications in all four flow graphs is the same and the structure of the three-loop program which executes the flow graph is the same.

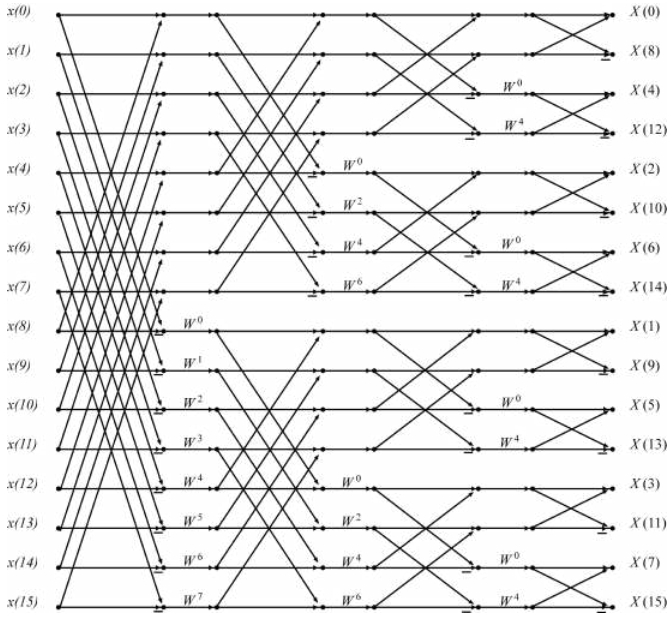


Fig. 14.1.1 Length-16, Decimation-in-Frequency, In-order input, Radix-2 FFT

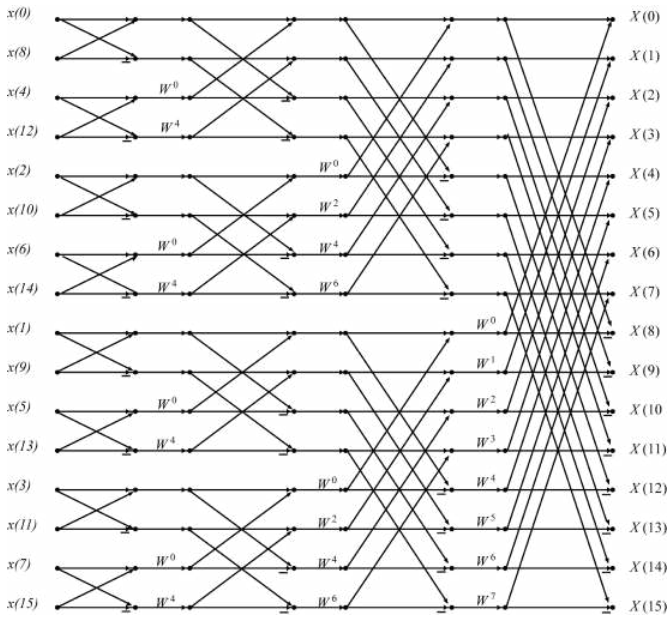


Fig. 14.1.2 Length-16, Decimation-in-Frequency, In-order output, Radix-2 FFT

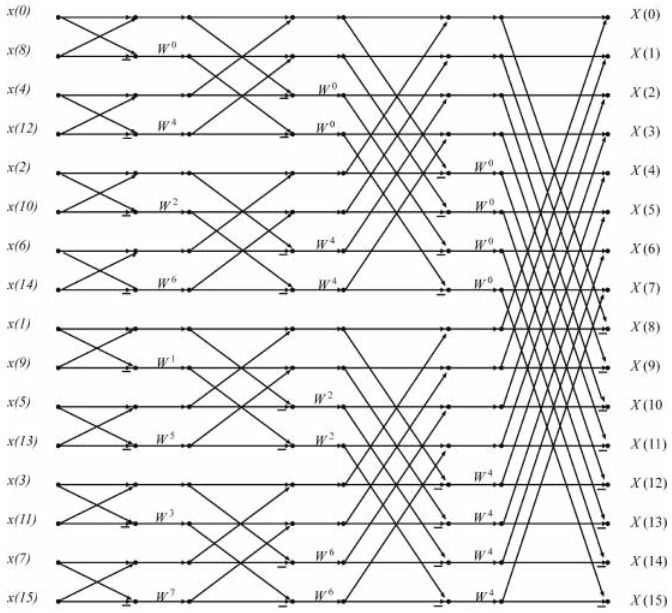


Fig. 14.1.3 Length-16, Decimation-in-Frequency, In-order output, Radix-2 FFT

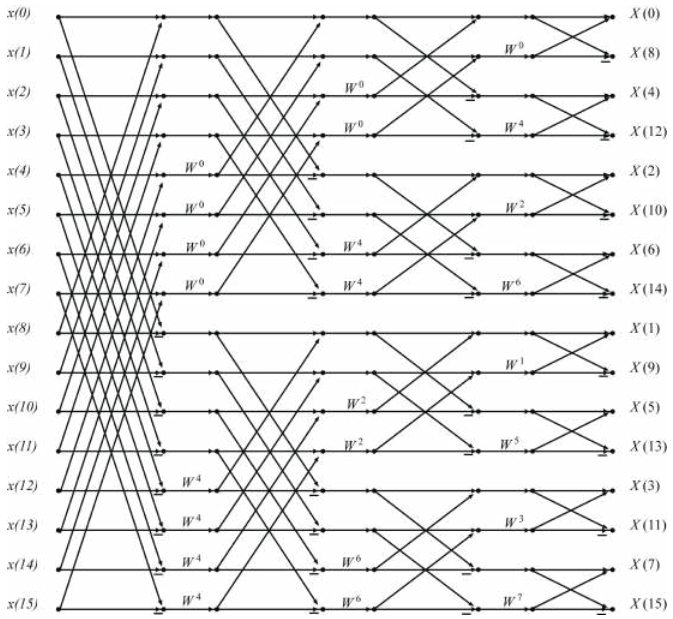


Fig. 14.1.4 Length-16, Decimation-in-Frequency, In-order output, Radix-2 FFT

The following is a length-16, decimation-in-frequency Radix-4 FFT with the input data in order and output data scrambled. There are two stages with the first stage having 4 length-4 "butterflies" followed by 12 multiplications by powers of W which are called "twiddle factors. The second stage has 4 length-4 FFTs which are each calculated by 4 butterflies followed by 4 multiplies. Note, each stage here looks like two stages but it is one and there is only one place where twiddle factor multiplications appear. This flow graph should be compared with the index map in Polynomial Description of Signals, the polynomial decomposition in The DFT as Convolution or Filtering, and the program in Appendix 3 - FFT Computer Programs. Log to the base 4 of 16 is 2. The total number of twiddle factor multiplication here is 12 compared to 24 for the radix-2. The unscrambler is a base-four reverse order counter rather than a bit reverse counter, however, a modification of the radix four butterflies will allow a bit reverse counter to be used with the radix-4 FFT as with the radix-2.

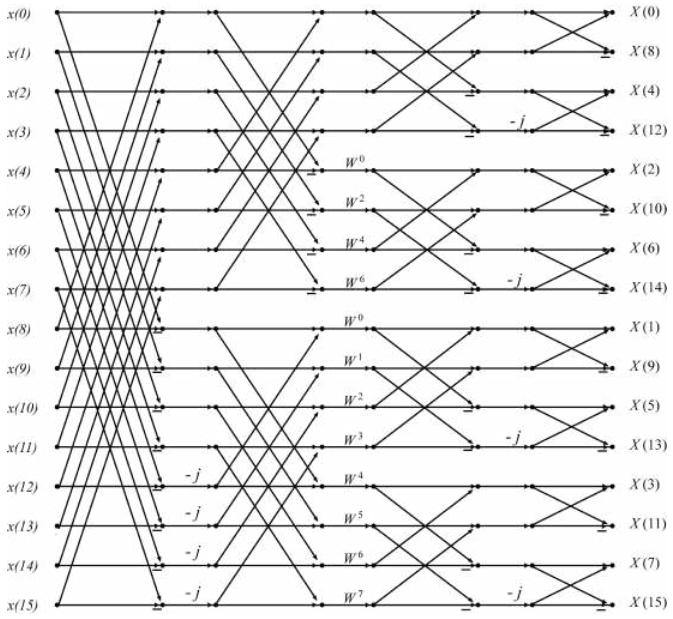


Fig. 14.1.5 Length-16, Decimation-in-Frequency, In-order output, Radix-4 FFT

The following two flowgraphs are length-16, decimation-in-frequency Split Radix FFTs with the input data in order and output data scrambled. Because the "butterflies" are L shaped, the stages do not progress uniformly like the Radix-2 or 4. These two figures are the same with the first drawn in a way to compare with the Radix-2 and 4, and the second to illustrate the L shaped butterflies.

These flow graphs should be compared with the index map in Polynomial Description of Signals and the program in Appendix 3 - FFT Computer Programs. Because of the non-uniform stages, the program indexing is more complicated. Although the number of twiddle factor multiplications is 12 as was the radix-4 case, for longer lengths, the split-radix has slightly fewer multiplications than the radix-4.

Because the structures of the radix-2, radix-4, and split-radix FFTs are the same, the number of data additions is same for all of them. However, each complex twiddle factor multiplication requires two real additions (and four real multiplications) the number of additions will be fewer for the structures with fewer multiplications.

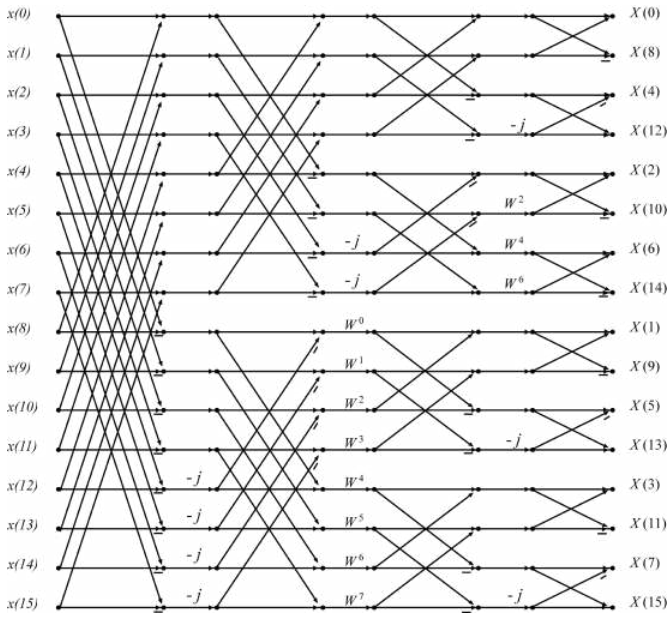


Fig. 14.1.6 Length-16, Decimation-in-Frequency, In-order output, Split-Radix FFT

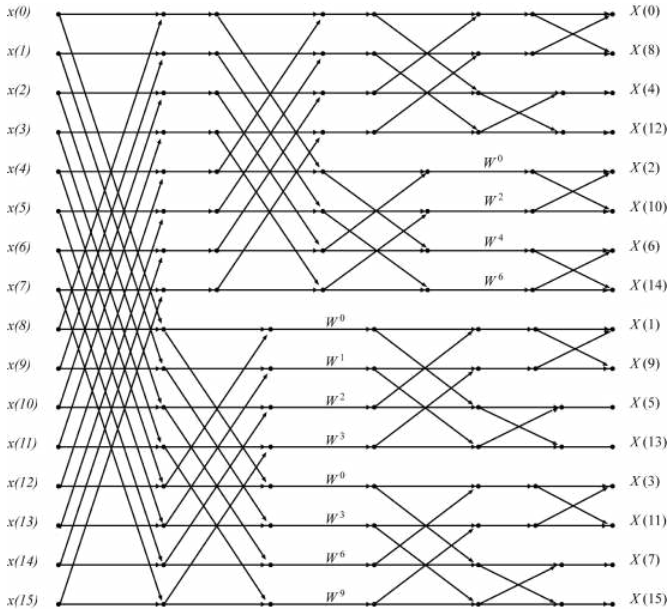


Fig. 14.1.5 Length-16, Decimation-in-Frequency,Split-Radix with special BFs FFT

Contributor

- ContribEEBurrus

14.2: Appendix 2 - Operation Counts for General Length FFT

The Glassman-Ferguson FFT is a compact implementation of a mixed-radix Cooley-Tukey FFT with the short DFTs for each factor being calculated by a Goertzel-like algorithm. This means there are twiddle factor multiplications even when the factors are relatively prime, however, the indexing is simple and compact. It will calculate the DFT of a sequence of any length but is efficient only if the length is highly composite. The figures contain plots of the number of floating point multiplications plus additions vs. the length of the FFT. The numbers on the vertical axis have relative meaning but no absolute meaning.

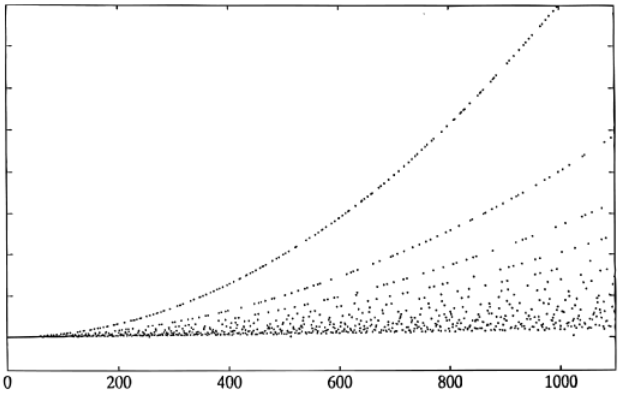


Fig. 14.2.1 Flop-Count vs Length for the Glassman-Ferguson FFT

Note the parabolic shape of the curve for certain values. The upper curve is for prime lengths, the next one is for lengths that are two times a prime, and the next one is for lengths that are for three times a prime, etc. The shape of the lower boundary is roughly $N \log N$. The program that generated these two figures used a Cooley-Tukey FFT if the length is two to a power which accounts for the points that are below the major lower boundary.

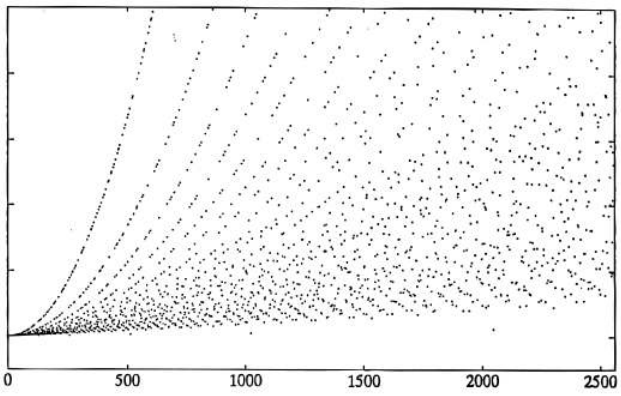


Fig. 14.2.1 Flop-Count vs Length for the Glassman-Ferguson FFT

Contributor

- ContribEEBurrus

14.3: Appendix 3 - FFT Computer Programs

Goertzel Algorithm

A FORTRAN implementation of the first-order Goertzel algorithm with in-order input is given below.

```
C-----------------------------------------------
C GOERTZEL'S DFT ALGORITHM
C First order, input inorder
C C. S. BURRUS, SEPT 1983
C-----------------------------------------------
SUBROUTINE DFT(X,Y,A,B,N)
REAL X(260), Y(260), A(260), B(260)
Q = 6.283185307179586/N
DO 20 J=1, N
C = COS(Q*(J-1))
S = SIN(Q*(J-1))
AT = X(1)
BT = Y(1)
DO 30 I = 2, N
T = C*AT - S*BT + X(I)
BT = C*BT + S*AT + Y(I)
AT = T
30 CONTINUE
A(J) = C*AT - S*BT
B(J) = C*BT + S*AT
20 CONTINUE
RETURN
END
NOT_CONVERTED_YET: caption
```

First Order Goertzel Algorithm

Second Order Goertzel Algorithm

Below is the program for a second order Goertzel algorithm.

```
C-----------------------------------------------
C GOERTZEL'S DFT ALGORITHM
C Second order, input inorder
C C. S. BURRUS, SEPT 1983
C-----------------------------------------------
SUBROUTINE DFT(X,Y,A,B,N)
REAL X(260), Y(260), A(260), B(260)
C
Q = 6.283185307179586/N
DO 20 J = 1, N
C = COS(Q*(J-1))
S = SIN(Q*(J-1))
CC = 2*C
A2 = 0
B2 = 0
A1 = X(1)
B1 = Y(1)
DO 30 I = 2, N
```

```
T = A1
A1 = CC*A1 - A2 + X(I)
A2 = T
T = B1
B1 = CC*B1 - B2 + Y(I)
B2 = T
30 CONTINUE
A(J) = C*A1 - A2 - S*B1
B(J) = C*B1 - B2 + S*A1
20 CONTINUE
C
RETURN
END
```

NOT_CONVERTED_YET: caption

Second Order Goertzel Algorithm

Second Order Goertzel Algorithm 2

Second order Goertzel algorithm that calculates two outputs at a time.

```
C-------------------------------------------------------
C GOERTZEL'S DFT ALGORITHM, Second order
C Input inorder, output by twos; C.S. Burrus, SEPT 1991
C-------------------------------------------------------
SUBROUTINE DFT(X,Y,A,B,N)
REAL X(260), Y(260), A(260), B(260)
Q = 6.283185307179586/N
DO 20 J = 1, N/2 + 1
C = COS(Q*(J-1))
S = SIN(Q*(J-1))
CC = 2*C
A2 = 0
B2 = 0
A1 = X(1)
B1 = Y(1)
DO 30 I = 2, N
T = A1
A1 = CC*A1 - A2 + X(I)
A2 = T
T = B1
B1 = CC*B1 - B2 + Y(I)
B2 = T
30 CONTINUE
A2 = C*A1 - A2
T = S*B1
A(J) = A2 - T
A(N-J+2) = A2 + T
B2 = C*B1 - B2
T = S*A1
B(J) = B2 + T
B(N-J+2) = B2 - T
20 CONTINUE
RETURN
```

```
END
```

Figure. Second Order Goertzel Calculating Two Outputs at a Time

Basic QFT Algorithm

A FORTRAN implementation of the basic QFT algorithm is given below to show how the theory is implemented. The program is written for clarity, not to minimize the number of floating point operations.

```
C
SUBROUTINE QDFT(X,Y,XX,YY,NN)
REAL X(0:260),Y(0:260),XX(0:260),YY(0:260)
C
N1 = NN - 1
N2 = N1/2
N21 = NN/2
Q = 6.283185308/NN
DO 2 K = 0, N21
SSX = X(0)
SSY = Y(0)
SDX = 0
SDY = 0
IF (MOD(NN,2).EQ.0) THEN
SSX = SSX + COS(3.1426*K)*X(N21)
SSY = SSY + COS(3.1426*K)*Y(N21)
ENDIF
DO 3 N = 1, N2
SSX = SSX + (X(N) + X(NN-N))*COS(Q*N*K)
SSY = SSY + (Y(N) + Y(NN-N))*COS(Q*N*K)
SDX = SDX + (X(N) - X(NN-N))*SIN(Q*N*K)
SDY = SDY + (Y(N) - Y(NN-N))*SIN(Q*N*K)
3 CONTINUE
XX(K) = SSX + SDY
YY(K) = SSY - SDX
XX(NN-K) = SSX - SDY
YY(NN-K) = SSY + SDX
2 CONTINUE
RETURN
END
```

NOT_CONVERTED_YET: caption

Simple QFT Fortran Program

Basic Radix-2 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Radix-2, one butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler.

```
C
C A COOLEY-TUKEY RADIX-2, DIF FFT PROGRAM
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-----------------------------------------------------------
SUBROUTINE FFT (X,Y,N,M)
REAL X(1), Y(1)
```

```
C--------------MAIN FFT LOOPS-----------------------------
C
N2 = N
DO 10 K = 1, M
N1 = N2
N2 = N2/2
E = 6.283185307179586/N1
A = 0
DO 20 J = 1, N2
C = COS (A)
S = SIN (A)
A = J*E
DO 30 I = J, N, N1
L = I + N2
XT = X(I) - X(L)
X(I) = X(I) + X(L)
YT = Y(I) - Y(L)
Y(I) = Y(I) + Y(L)
X(L) = C*XT + S*YT
Y(L) = C*YT - S*XT
30 CONTINUE
20 CONTINUE
10 CONTINUE
C
C------------DIGIT REVERSE COUNTER-----------------
100 J = 1
N1 = N - 1
DO 104 I=1, N1
IF (I.GE.J) GOXTO 101
XT = X(J)
X(J) = X(I)
X(I) = XT
XT = Y(J)
Y(J) = Y(I)
Y(I) = XT
101 K = N/2
102 IF (K.GE.J) GOTO 103
J = J - K
K = K/2
GOTO 102
103 J = J + K
104 CONTINUE
RETURN
END
```

Figure: Radix-2, DIF, One Butterfly Cooley-Tukey FFT

Basic DIT Radix-2 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Time, Radix-2, one butterfly Cooley-Tukey FFT preceeded by a bit-reversing scrambler.

```
C
C A COOLEY-TUKEY RADIX-2, DIT FFT PROGRAM
```

```
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C C. S. BURRUS, RICE UNIVERSITY, SEPT 1985
C
C-------------------------------------------------------
SUBROUTINE FFT (X,Y,N,M)
REAL X(1), Y(1)
C------------DIGIT REVERSE COUNTER-----------------
C
100 J = 1
N1 = N - 1
DO 104 I=1, N1
IF (I.GE.J) GOTO 101
XT = X(J)
X(J) = X(I)
X(I) = XT
XT = Y(J)
Y(J) = Y(I)
Y(I) = XT
101 K = N/2
102 IF (K.GE.J) GOTO 103
J = J - K
K = K/2
GOTO 102
103 J = J + K
104 CONTINUE
C--------------MAIN FFT LOOPS-----------------------------
C
N2 = 1
DO 10 K = 1, M
E = 6.283185307179586/(2*N2)
A = 0
DO 20 J = 1, N2
C = COS (A)
S = SIN (A)
A = J*E
DO 30 I = J, N, 2*N2
L = I + N2
XT = C*X(L) + S*Y(L)
YT = C*Y(L) - S*X(L)
X(L) = X(I) - XT
X(I) = X(I) + XT
Y(L) = Y(I) - YT
Y(I) = Y(I) + YT
30 CONTINUE
20 CONTINUE
N2 = N2+N2
10 CONTINUE
C
RETURN
END
```

DIF Radix-2 FFT Algorithm

Below is the Fortran code for a Decimation-in-Frequency, Radix-2, three butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler.

```
C A COOLEY-TUKEY RADIX 2, DIF FFT PROGRAM
C THREE-BF, MULT BY 1 AND J ARE REMOVED
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C TABLE LOOK-UP OF W VALUES
C C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-----------------------------------------------------
SUBROUTINE FFT (X,Y,N,M,WR,WI)
REAL X(1), Y(1), WR(1), WI(1)
C--------------MAIN FFT LOOPS-----------------------------
C
N2 = N
DO 10 K = 1, M
N1 = N2
N2 = N2/2
JT = N2/2 + 1
DO 1 I = 1, N, N1
L = I + N2
T = X(I) - X(L)
X(I) = X(I) + X(L)
X(L) = T
T = Y(I) - Y(L)
Y(I) = Y(I) + Y(L)
Y(L) = T
1 CONTINUE
IF (K.EQ.M) GOTO 10
IE = N/N1
IA = 1
DO 20 J = 2, N2
IA = IA + IE
IF (J.EQ.JT) GOTO 50
C = WR(IA)
S = WI(IA)
DO 30 I = J, N, N1
L = I + N2
T = X(I) - X(L)
X(I) = X(I) + X(L)
TY = Y(I) - Y(L)
Y(I) = Y(I) + Y(L)
X(L) = C*T + S*TY
Y(L) = C*TY - S*T
30 CONTINUE
GOTO 25
50 DO 40 I = J, N, N1
L = I + N2
T = X(I) - X(L)
X(I) = X(I) + X(L)
TY = Y(I) - Y(L)
Y(I) = Y(I) + Y(L)
```

```
X(L) = TY
Y(L) =-T
40 CONTINUE
25 A = J*E
20 CONTINUE
10 CONTINUE
C------------DIGIT REVERSE COUNTER Goes here----------
RETURN
END
```

Basic DIF Radix-4 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Radix-4, one butterfly Cooley-Tukey FFT to be followed by an unscrambler.

```
C A COOLEY-TUKEY RADIX-4 DIF FFT PROGRAM
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C LENGTH IS N = 4 ** M
C C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-------------------------------------------------------
SUBROUTINE FFT4 (X,Y,N,M)
REAL X(1), Y(1)
C--------------MAIN FFT LOOPS-----------------------------
N2 = N
DO 10 K = 1, M
N1 = N2
N2 = N2/4
E = 6.283185307179586/N1
A = 0
C--------------------MAIN BUTTERFLIES-------------------
DO 20 J=1, N2
B = A + A
C = A + B
CO1 = COS(A)
CO2 = COS(B)
CO3 = COS(C)
SI1 = SIN(A)
SI2 = SIN(B)
SI3 = SIN(C)
A = J*E
C----------------BUTTERFLIES WITH SAME W---------------
DO 30 I=J, N, N1
I1 = I + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)
X(I) = R1 + R2
```

```
R2 = R1 - R2
R1 = R3 - S4
R3 = R3 + S4
Y(I) = S1 + S2
S2 = S1 - S2
S1 = S3 + R4
S3 = S3 - R4
X(I1) = CO1*R3 + SI1*S3
Y(I1) = CO1*S3 - SI1*R3
X(I2) = CO2*R2 + SI2*S2
Y(I2) = CO2*S2 - SI2*R2
X(I3) = CO3*R1 + SI3*S1
Y(I3) = CO3*S1 - SI3*R1
30 CONTINUE
20 CONTINUE
10 CONTINUE
C-----------DIGIT REVERSE COUNTER goes here-----
RETURN
END
```

Basic DIF Radix-4 FFT Algorithm

Below is the Fortran code for a Decimation-in-Frequency, Radix-4, three butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler. Twiddle factors are precalculated and stored in arrays WR and WI.

```
C
C A COOLEY-TUKEY RADIX-4 DIF FFT PROGRAM
C THREE BF, MULTIPLICATIONS BY 1, J, ETC. ARE REMOVED
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C LENGTH IS N = 4 ** M
C TABLE LOOKUP OF W VALUES
C
C C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C
C-------------------------------------------------------
C
SUBROUTINE FFT4 (X,Y,N,M,WR,WI)
REAL X(1), Y(1), WR(1), WI(1)
DATA C21 / 0.707106778 /
C
C--------------MAIN FFT LOOPS-----------------------------
C
N2 = N
DO 10 K = 1, M
N1 = N2
N2 = N2/4
JT = N2/2 + 1
C---------------SPECIAL BUTTERFLY FOR W = 1---------------
DO 1 I = 1, N, N1
I1 = I + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
```

```
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)
C
X(I) = R1 + R2
X(I2)= R1 - R2
X(I3)= R3 - S4
X(I1)= R3 + S4
C
Y(I) = S1 + S2
Y(I2)= S1 - S2
Y(I3)= S3 + R4
Y(I1)= S3 - R4
C
1 CONTINUE
IF (K.EQ.M) GOTO 10
IE = N/N1
IA1 = 1

C---------------GENERAL BUTTERFLY-----------------
DO 20 J = 2, N2
IA1 = IA1 + IE
IF (J.EQ.JT) GOTO 50
IA2 = IA1 + IA1 - 1
IA3 = IA2 + IA1 - 1
CO1 = WR(IA1)
CO2 = WR(IA2)
CO3 = WR(IA3)
SI1 = WI(IA1)
SI2 = WI(IA2)
SI3 = WI(IA3)
C-----------------BUTTERFLIES WITH SAME W----------------
DO 30 I = J, N, N1
I1 = I + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)
C
X(I) = R1 + R2
R2 = R1 - R2
R1 = R3 - S4
```

```
R3 = R3 + S4
C
Y(I) = S1 + S2
S2 = S1 - S2
S1 = S3 + R4
S3 = S3 - R4
C
X(I1) = CO1*R3 + SI1*S3
Y(I1) = CO1*S3 - SI1*R3
X(I2) = CO2*R2 + SI2*S2
Y(I2) = CO2*S2 - SI2*R2
X(I3) = CO3*R1 + SI3*S1
Y(I3) = CO3*S1 - SI3*R1
30 CONTINUE
GOTO 20
C------------------SPECIAL BUTTERFLY FOR W = J-----------
50 DO 40 I = J, N, N1
I1 = I + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)
C
X(I) = R1 + R2
Y(I2)=-R1 + R2
R1 = R3 - S4
R3 = R3 + S4
C
Y(I) = S1 + S2
X(I2)= S1 - S2
S1 = S3 + R4
S3 = S3 - R4
C
X(I1) = (S3 + R3)*C21
Y(I1) = (S3 - R3)*C21
X(I3) = (S1 - R1)*C21
Y(I3) =-(S1 + R1)*C21
40 CONTINUE
20 CONTINUE
10 CONTINUE
C-----------DIGIT REVERSE COUNTER----------
100 J = 1
N1 = N - 1
DO 104 I = 1, N1
IF (I.GE.J) GOTO 101
R1 = X(J)
```

```
X(J) = X(I)
X(I) = R1
R1 = Y(J)
Y(J) = Y(I)
Y(I) = R1
101 K = N/4
102 IF (K*3.GE.J) GOTO 103
J = J - K*3
K = K/4
GOTO 102
103 J = J + K
104 CONTINUE
RETURN
END
```

Basic DIF Split Radix FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Split-Radix, one butterfly FFT to be followed by a bit-reversing unscrambler.

```
C A DUHAMEL-HOLLMANN SPLIT RADIX FFT PROGRAM
C FROM: ELECTRONICS LETTERS, JAN. 5, 1984
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C LENGTH IS N = 2 ** M
C C. S. BURRUS, RICE UNIVERSITY, MARCH 1984
C
C-------------------------------------------------------
SUBROUTINE FFT (X,Y,N,M)
REAL X(1), Y(1)
C--------------MAIN FFT LOOPS-----------------------------
C
N1 = N
N2 = N/2
IP = 0
IS = 1
A = 6.283185307179586/N
DO 10 K = 1, M-1
JD = N1 + N2
N1 = N2
N2 = N2/2
J0 = N1*IP + 1
IP = 1 - IP
DO 20 J = J0, N, JD
JS = 0
JT = J + N2 - 1
DO 30 I = J, JT
JSS= JS*IS
JS = JS + 1
C1 = COS(A*JSS)
C3 = COS(3*A*JSS)
S1 = -SIN(A*JSS)
S3 = -SIN(3*A*JSS)
I1 = I + N2
I2 = I1 + N2
```

```
I3 = I2 + N2
R1 = X(I ) + X(I2)
R2 = X(I ) - X(I2)
R3 = X(I1) - X(I3)
X(I2) = X(I1) + X(I3)
X(I1) = R1
C
R1 = Y(I ) + Y(I2)
R4 = Y(I ) - Y(I2)
R5 = Y(I1) - Y(I3)
Y(I2) = Y(I1) + Y(I3)
Y(I1) = R1
C
R1 = R2 - R5
R2 = R2 + R5
R5 = R4 + R3
R4 = R4 - R3
C
X(I) = C1*R1 + S1*R5
Y(I) = C1*R5 - S1*R1
X(I3) = C3*R2 + S3*R4
Y(I3) = C3*R4 - S3*R2
30 CONTINUE
20 CONTINUE
IS = IS + IS
10 CONTINUE
IP = 1 - IP
J0 = 2 - IP
DO 5 I = J0, N-1, 3
I1 = I + 1
R1 = X(I) + X(I1)
X(I1) = X(I) - X(I1)
X(I) = R1
R1 = Y(I) + Y(I1)
Y(I1) = Y(I) - Y(I1)
Y(I) = R1
5 CONTINUE
RETURN
END
```

DIF Split Radix FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Split-Radix, two butterfly FFT to be followed by a bit-reversing unscrambler. Twiddle factors are precalculated and stored in arrays WR and WI.

```
C---------------------------------------------------------------C
C A DUHAMEL-HOLLMAN SPLIT RADIX FFT C
C REF: ELECTRONICS LETTERS, JAN. 5, 1984 C
C COMPLEX INPUT AND OUTPUT DATA IN ARRAYS X AND Y C
C LENGTH IS N = 2 ** M, OUTPUT IN BIT-REVERSED ORDER C
C TWO BUTTERFLIES TO REMOVE MULTS BY UNITY C
C SPECIAL LAST TWO STAGES C
C TABLE LOOK-UP OF SINE AND COSINE VALUES C
C C.S. BURRUS, RICE UNIV. APRIL 1985 C
```

```
C-------------------------------------------------------------C
C
SUBROUTINE FFT(X,Y,N,M,WR,WI)
REAL X(1),Y(1),WR(1),WI(1)
C81= 0.707106778
N2 = 2*N
DO 10 K = 1, M-3
IS = 1
ID = N2
N2 = N2/2
N4 = N2/4
2 DO 1 I0 = IS, N-1, ID
I1 = I0 + N4
I2 = I1 + N4
I3 = I2 + N4
R1 = X(I0) - X(I2)
X(I0) = X(I0) + X(I2)
R2 = Y(I1) - Y(I3)
Y(I1) = Y(I1) + Y(I3)
X(I2) = R1 + R2
R2 = R1 - R2
R1 = X(I1) - X(I3)
X(I1) = X(I1) + X(I3)
X(I3) = R2
R2 = Y(I0) - Y(I2)
Y(I0) = Y(I0) + Y(I2)
Y(I2) =-R1 + R2
Y(I3) = R1 + R2
1 CONTINUE
IS = 2*ID - N2 + 1
ID = 4*ID
IF (IS.LT.N) GOTO 2
IE = N/N2
IA1 = 1
DO 20 J = 2, N4
IA1 = IA1 + IE
IA3 = 3*IA1 - 2
CC1 = WR(IA1)
SS1 = WI(IA1)
CC3 = WR(IA3)
SS3 = WI(IA3)
IS = J
ID = 2*N2
40 DO 30 I0 = IS, N-1, ID
I1 = I0 + N4
I2 = I1 + N4
I3 = I2 + N4
C
R1 = X(I0) - X(I2)
X(I0) = X(I0) + X(I2)
R2 = X(I1) - X(I3)
X(I1) = X(I1) + X(I3)
```

```
S1 = Y(I0) - Y(I2)
Y(I0) = Y(I0) + Y(I2)
S2 = Y(I1) - Y(I3)
Y(I1) = Y(I1) + Y(I3)
C
S3 = R1 - S2
R1 = R1 + S2
S2 = R2 - S1
R2 = R2 + S1
X(I2) = R1*CC1 - S2*SS1
Y(I2) =-S2*CC1 - R1*SS1
X(I3) = S3*CC3 + R2*SS3
Y(I3) = R2*CC3 - S3*SS3
30 CONTINUE
IS = 2*ID - N2 + J
ID = 4*ID
IF (IS.LT.N) GOTO 40
20 CONTINUE
10 CONTINUE
C
IS = 1
ID = 32
50 DO 60 I = IS, N, ID
I0 = I + 8
DO 15 J = 1, 2
R1 = X(I0) + X(I0+2)
R3 = X(I0) - X(I0+2)
R2 = X(I0+1) + X(I0+3)
R4 = X(I0+1) - X(I0+3)
X(I0) = R1 + R2
X(I0+1) = R1 - R2
R1 = Y(I0) + Y(I0+2)
S3 = Y(I0) - Y(I0+2)
R2 = Y(I0+1) + Y(I0+3)
S4 = Y(I0+1) - Y(I0+3)
Y(I0) = R1 + R2
Y(I0+1) = R1 - R2
Y(I0+2) = S3 - R4
Y(I0+3) = S3 + R4
X(I0+2) = R3 + S4
X(I0+3) = R3 - S4
I0 = I0 + 4
15 CONTINUE
60 CONTINUE
IS = 2*ID - 15
ID = 4*ID
IF (IS.LT.N) GOTO 50
C
IS = 1
ID = 16
55 DO 65 I0 = IS, N, ID
R1 = X(I0) + X(I0+4)
```

```
R5 = X(I0) - X(I0+4)
R2 = X(I0+1) + X(I0+5)
R6 = X(I0+1) - X(I0+5)
R3 = X(I0+2) + X(I0+6)
R7 = X(I0+2) - X(I0+6)
R4 = X(I0+3) + X(I0+7)
R8 = X(I0+3) - X(I0+7)
T1 = R1 - R3
R1 = R1 + R3
R3 = R2 - R4
R2 = R2 + R4
X(I0) = R1 + R2
X(I0+1) = R1 - R2
C
R1 = Y(I0) + Y(I0+4)
S5 = Y(I0) - Y(I0+4)
R2 = Y(I0+1) + Y(I0+5)
S6 = Y(I0+1) - Y(I0+5)
S3 = Y(I0+2) + Y(I0+6)
S7 = Y(I0+2) - Y(I0+6)
R4 = Y(I0+3) + Y(I0+7)
S8 = Y(I0+3) - Y(I0+7)
T2 = R1 - S3
R1 = R1 + S3
S3 = R2 - R4
R2 = R2 + R4
Y(I0) = R1 + R2
Y(I0+1) = R1 - R2
X(I0+2) = T1 + S3
X(I0+3) = T1 - S3
Y(I0+2) = T2 - R3
Y(I0+3) = T2 + R3
C
R1 = (R6 - R8)*C81
R6 = (R6 + R8)*C81
R2 = (S6 - S8)*C81
S6 = (S6 + S8)*C81
C
T1 = R5 - R1
R5 = R5 + R1
R8 = R7 - R6
R7 = R7 + R6
T2 = S5 - R2
S5 = S5 + R2
S8 = S7 - S6
S7 = S7 + S6
X(I0+4) = R5 + S7
X(I0+7) = R5 - S7
X(I0+5) = T1 + S8
X(I0+6) = T1 - S8
Y(I0+4) = S5 - R7
Y(I0+7) = S5 + R7
```

```
Y(I0+5) = T2 - R8
Y(I0+6) = T2 + R8
65 CONTINUE
IS = 2*ID - 7
ID = 4*ID
IF (IS.LT.N) GOTO 55
C
C------------BIT REVERSE COUNTER-----------------
C
100 J = 1
N1 = N - 1
DO 104 I=1, N1
IF (I.GE.J) GOTO 101
XT = X(J)
X(J) = X(I)
X(I) = XT
XT = Y(J)
Y(J) = Y(I)
Y(I) = XT
101 K = N/2
102 IF (K.GE.J) GOTO 103
J = J - K
K = K/2
GOTO 102
103 J = J + K
104 CONTINUE
RETURN
END
```

Prime Factor FFT Algorithm

Below is the Fortran code for a Prime-Factor Algorithm (PFA) FFT allowing factors of the length of 2, 3, 4, 5, and 7. It is followed by an unscrambler.

```
C-------------------------------------------------
C
C A PRIME FACTOR FFT PROGRAM WITH GENERAL MODULES
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C COMPLEX OUTPUT IN A AND B
C LENGTH N WITH M FACTORS IN ARRAY NI
C N = NI(1)*NI(2)* ... *NI(M)
C UNSCRAMBLING CONSTANT UNSC
C UNSC = N/NI(1) + N/NI(2) +...+ N/NI(M), MOD N
C C. S. BURRUS, RICE UNIVERSITY, JAN 1987
C
C-------------------------------------------------
C
SUBROUTINE PFA(X,Y,N,M,NI,A,B,UNSC)
C
INTEGER NI(4), I(16), UNSC
REAL X(1), Y(1), A(1), B(1)
C
DATA C31, C32 / -0.86602540,-1.50000000 /
DATA C51, C52 / 0.95105652,-1.53884180 /
```

```
DATA C53, C54 / -0.36327126, 0.55901699 /
DATA C55 / -1.25 /
DATA C71, C72 / -1.16666667,-0.79015647 /
DATA C73, C74 / 0.055854267, 0.7343022 /
DATA C75, C76 / 0.44095855,-0.34087293 /
DATA C77, C78 / 0.53396936, 0.87484229 /
C
C-----------------NESTED LOOPS----------------------
C
DO 10 K=1, M
N1 = NI(K)
N2 = N/N1
DO 15 J=1, N, N1
IT = J
DO 30 L=1, N1
I(L) = IT
A(L) = X(IT)
B(L) = Y(IT)
IT = IT + N2
IF (IT.GT.N) IT = IT - N
30 CONTINUE
GOTO (20,102,103,104,105,20,107), N1
C
C----------------WFTA N=2--------------------------------
C
102 R1 = A(1)
A(1) = R1 + A(2)
A(2) = R1 - A(2)
C
R1 = B(1)
B(1) = R1 + B(2)
B(2) = R1 - B(2)
C
GOTO 20
C----------------WFTA N=3--------------------------------
C
103 R2 = (A(2) - A(3)) * C31
R1 = A(2) + A(3)
A(1)= A(1) + R1
R1 = A(1) + R1 * C32
C
S2 = (B(2) - B(3)) * C31
S1 = B(2) + B(3)
B(1)= B(1) + S1
S1 = B(1) + S1 * C32
C
A(2) = R1 - S2
A(3) = R1 + S2
B(2) = S1 + R2
B(3) = S1 - R2
C
GOTO 20
```

```
C
C----------------WFTA N=4---------------------------------
C
104 R1 = A(1) + A(3)
T1 = A(1) - A(3)
R2 = A(2) + A(4)
A(1) = R1 + R2
A(3) = R1 - R2
C
R1 = B(1) + B(3)
T2 = B(1) - B(3)
R2 = B(2) + B(4)
B(1) = R1 + R2
B(3) = R1 - R2
C
R1 = A(2) - A(4)
R2 = B(2) - B(4)
C
A(2) = T1 + R2
A(4) = T1 - R2
B(2) = T2 - R1
B(4) = T2 + R1
C
GOTO 20
C
C----------------WFTA N=5---------------------------------
C
105 R1 = A(2) + A(5)
R4 = A(2) - A(5)
R3 = A(3) + A(4)
R2 = A(3) - A(4)
C
T = (R1 - R3) * C54
R1 = R1 + R3
A(1) = A(1) + R1
R1 = A(1) + R1 * C55
C
R3 = R1 - T
R1 = R1 + T
C
T = (R4 + R2) * C51
R4 = T + R4 * C52
R2 = T + R2 * C53
C
S1 = B(2) + B(5)
S4 = B(2) - B(5)
S3 = B(3) + B(4)
S2 = B(3) - B(4)
C
T = (S1 - S3) * C54
S1 = S1 + S3
B(1) = B(1) + S1
```

```
S1 = B(1) + S1 * C55
C
S3 = S1 - T
S1 = S1 + T
C
T = (S4 + S2) * C51
S4 = T + S4 * C52
S2 = T + S2 * C53
C
A(2) = R1 + S2
A(5) = R1 - S2
A(3) = R3 - S4
A(4) = R3 + S4
C
B(2) = S1 - R2
B(5) = S1 + R2
B(3) = S3 + R4
B(4) = S3 - R4
C
GOTO 20
C-----------------WFTA N=7--------------------------
C
107 R1 = A(2) + A(7)
R6 = A(2) - A(7)
S1 = B(2) + B(7)
S6 = B(2) - B(7)
R2 = A(3) + A(6)
R5 = A(3) - A(6)
S2 = B(3) + B(6)
S5 = B(3) - B(6)
R3 = A(4) + A(5)
R4 = A(4) - A(5)
S3 = B(4) + B(5)
S4 = B(4) - B(5)
C
T3 = (R1 - R2) * C74
T = (R1 - R3) * C72
R1 = R1 + R2 + R3
A(1) = A(1) + R1
R1 = A(1) + R1 * C71
R2 =(R3 - R2) * C73
R3 = R1 - T + R2
R2 = R1 - R2 - T3
R1 = R1 + T + T3
T = (R6 - R5) * C78
T3 =(R6 + R4) * C76
R6 =(R6 + R5 - R4) * C75
R5 =(R5 + R4) * C77
R4 = R6 - T3 + R5
R5 = R6 - R5 - T
R6 = R6 + T3 + T
C
```

```
T3 = (S1 - S2) * C74
T = (S1 - S3) * C72
S1 = S1 + S2 + S3
B(1) = B(1) + S1
S1 = B(1) + S1 * C71
S2 =(S3 - S2) * C73
S3 = S1 - T + S2
S2 = S1 - S2 - T3
S1 = S1 + T + T3
T = (S6 - S5) * C78
T3 = (S6 + S4) * C76
S6 = (S6 + S5 - S4) * C75
S5 = (S5 + S4) * C77
S4 = S6 - T3 + S5
S5 = S6 - S5 - T
S6 = S6 + T3 + T
C
A(2) = R3 + S4
A(7) = R3 - S4
A(3) = R1 + S6
A(6) = R1 - S6
A(4) = R2 - S5
A(5) = R2 + S5
B(4) = S2 + R5
B(5) = S2 - R5
B(2) = S3 - R4
B(7) = S3 + R4
B(3) = S1 - R6
B(6) = S1 + R6
C
20 IT = J
DO 31 L=1, N1
I(L) = IT
X(IT) = A(L)
Y(IT) = B(L)
IT = IT + N2
IF (IT.GT.N) IT = IT - N
31 CONTINUE
15 CONTINUE
10 CONTINUE
C
C-----------------UNSCRAMBLING----------------------
C
L = 1
DO 2 K=1, N
A(K) = X(L)
B(K) = Y(L)
L = L + UNSC
IF (L.GT.N) L = L - N
2 CONTINUE
RETURN
```

```
END
C
```

In Place, In Order Prime Factor FFT Algorithm

Below is the Fortran code for a Prime-Factor Algorithm (PFA) FFT allowing factors of the length of 2, 3, 4, 5, 7, 8, 9, and 16. It is both in-place and in-order, so requires no unscrambler.

```
C
C A PRIME FACTOR FFT PROGRAM
C IN-PLACE AND IN-ORDER
C COMPLEX INPUT DATA IN ARRAYS X AND Y
C LENGTH N WITH M FACTORS IN ARRAY NI
C N = NI(1)*NI(2)*...*NI(M)
C REDUCED TEMP STORAGE IN SHORT WFTA MODULES
C Has modules 2,3,4,5,7,8,9,16
C PROGRAM BY C. S. BURRUS, RICE UNIVERSITY
C SEPT 1983
C-------------------------------------------------------
C
SUBROUTINE PFA(X,Y,N,M,NI)
INTEGER NI(4), I(16), IP(16), LP(16)
REAL X(1), Y(1)
DATA C31, C32 / -0.86602540,-1.50000000 /
DATA C51, C52 / 0.95105652,-1.53884180 /
DATA C53, C54 / -0.36327126, 0.55901699 /
DATA C55 / -1.25 /
DATA C71, C72 / -1.16666667,-0.79015647 /
DATA C73, C74 / 0.055854267, 0.7343022 /
DATA C75, C76 / 0.44095855,-0.34087293 /
DATA C77, C78 / 0.53396936, 0.87484229 /
DATA C81 / 0.70710678 /
DATA C95 / -0.50000000 /
DATA C92, C93 / 0.93969262, -0.17364818 /
DATA C94, C96 / 0.76604444, -0.34202014 /
DATA C97, C98 / -0.98480775, -0.64278761 /
DATA C162,C163 / 0.38268343, 1.30656297 /
DATA C164,C165 / 0.54119610, 0.92387953 /
C
C-----------------NESTED LOOPS----------------------------------
C
DO 10 K=1, M
N1 = NI(K)
N2 = N/N1
L = 1
N3 = N2 - N1*(N2/N1)
DO 15 J = 1, N1
LP(J) = L
L = L + N3
IF (L.GT.N1) L = L - N1
15 CONTINUE
C
DO 20 J=1, N, N1
IT = J
```

```
      DO 30 L=1, N1
      I(L) = IT
      IP(LP(L)) = IT
      IT = IT + N2
      IF (IT.GT.N) IT = IT - N
   30 CONTINUE
      GOTO (20,102,103,104,105,20,107,108,109,
     + 20,20,20,20,20,20,116),N1

C-----------------WFTA N=2--------------------------------
C
  102 R1 = X(I(1))
      X(I(1)) = R1 + X(I(2))
      X(I(2)) = R1 - X(I(2))
C
      R1 = Y(I(1))
      Y(IP(1)) = R1 + Y(I(2))
      Y(IP(2)) = R1 - Y(I(2))
C
      GOTO 20
C
C-----------------WFTA N=3--------------------------------
C
  103 R2 = (X(I(2)) - X(I(3))) * C31
      R1 = X(I(2)) + X(I(3))
      X(I(1))= X(I(1)) + R1
      R1 = X(I(1)) + R1 * C32
C
      S2 = (Y(I(2)) - Y(I(3))) * C31
      S1 = Y(I(2)) + Y(I(3))
      Y(I(1))= Y(I(1)) + S1
      S1 = Y(I(1)) + S1 * C32
C
      X(IP(2)) = R1 - S2
      X(IP(3)) = R1 + S2
      Y(IP(2)) = S1 + R2
      Y(IP(3)) = S1 - R2
C
      GOTO 20
C
C-----------------WFTA N=4--------------------------------
C
  104 R1 = X(I(1)) + X(I(3))
      T1 = X(I(1)) - X(I(3))
      R2 = X(I(2)) + X(I(4))
      X(IP(1)) = R1 + R2
      X(IP(3)) = R1 - R2
C
      R1 = Y(I(1)) + Y(I(3))
      T2 = Y(I(1)) - Y(I(3))
      R2 = Y(I(2)) + Y(I(4))
      Y(IP(1)) = R1 + R2
```

```
Y(IP(3)) = R1 - R2
C
R1 = X(I(2)) - X(I(4))
R2 = Y(I(2)) - Y(I(4))
C
X(IP(2)) = T1 + R2
X(IP(4)) = T1 - R2
Y(IP(2)) = T2 - R1
Y(IP(4)) = T2 + R1
C
GOTO 20

C----------------WFTA N=5--------------------------------
C
105 R1 = X(I(2)) + X(I(5))
R4 = X(I(2)) - X(I(5))
R3 = X(I(3)) + X(I(4))
R2 = X(I(3)) - X(I(4))
C
T = (R1 - R3) * C54
R1 = R1 + R3
X(I(1)) = X(I(1)) + R1
R1 = X(I(1)) + R1 * C55
C
R3 = R1 - T
R1 = R1 + T
C
T = (R4 + R2) * C51
R4 = T + R4 * C52
R2 = T + R2 * C53
C
S1 = Y(I(2)) + Y(I(5))
S4 = Y(I(2)) - Y(I(5))
S3 = Y(I(3)) + Y(I(4))
S2 = Y(I(3)) - Y(I(4))
C
T = (S1 - S3) * C54
S1 = S1 + S3
Y(I(1)) = Y(I(1)) + S1
S1 = Y(I(1)) + S1 * C55
C
S3 = S1 - T
S1 = S1 + T
C
T = (S4 + S2) * C51
S4 = T + S4 * C52
S2 = T + S2 * C53
C
X(IP(2)) = R1 + S2
X(IP(5)) = R1 - S2
X(IP(3)) = R3 - S4
X(IP(4)) = R3 + S4
```

```
C
Y(IP(2)) = S1 - R2
Y(IP(5)) = S1 + R2
Y(IP(3)) = S3 + R4
Y(IP(4)) = S3 - R4
C
GOTO 20

C-----------------WFTA N=7--------------------------
C
107 R1 = X(I(2)) + X(I(7))
R6 = X(I(2)) - X(I(7))
S1 = Y(I(2)) + Y(I(7))
S6 = Y(I(2)) - Y(I(7))
R2 = X(I(3)) + X(I(6))
R5 = X(I(3)) - X(I(6))
S2 = Y(I(3)) + Y(I(6))
S5 = Y(I(3)) - Y(I(6))
R3 = X(I(4)) + X(I(5))
R4 = X(I(4)) - X(I(5))
S3 = Y(I(4)) + Y(I(5))
S4 = Y(I(4)) - Y(I(5))
C
T3 = (R1 - R2) * C74
T = (R1 - R3) * C72
R1 = R1 + R2 + R3
X(I(1)) = X(I(1)) + R1
R1 = X(I(1)) + R1 * C71
R2 =(R3 - R2) * C73
R3 = R1 - T + R2
R2 = R1 - R2 - T3
R1 = R1 + T + T3
T = (R6 - R5) * C78
T3 =(R6 + R4) * C76
R6 =(R6 + R5 - R4) * C75
R5 =(R5 + R4) * C77
R4 = R6 - T3 + R5
R5 = R6 - R5 - T
R6 = R6 + T3 + T
C
T3 = (S1 - S2) * C74
T = (S1 - S3) * C72
S1 = S1 + S2 + S3
Y(I(1)) = Y(I(1)) + S1
S1 = Y(I(1)) + S1 * C71
S2 =(S3 - S2) * C73
S3 = S1 - T + S2
S2 = S1 - S2 - T3
S1 = S1 + T + T3
T = (S6 - S5) * C78
T3 = (S6 + S4) * C76
S6 = (S6 + S5 - S4) * C75
```

```
S5 = (S5 + S4) * C77
S4 = S6 - T3 + S5
S5 = S6 - S5 - T
S6 = S6 + T3 + T
C
X(IP(2)) = R3 + S4
X(IP(7)) = R3 - S4
X(IP(3)) = R1 + S6
X(IP(6)) = R1 - S6
X(IP(4)) = R2 - S5
X(IP(5)) = R2 + S5
Y(IP(4)) = S2 + R5
Y(IP(5)) = S2 - R5
Y(IP(2)) = S3 - R4
Y(IP(7)) = S3 + R4
Y(IP(3)) = S1 - R6
Y(IP(6)) = S1 + R6
C
GOTO 20

C-----------------WFTA N=8--------------------------
C
108 R1 = X(I(1)) + X(I(5))
R2 = X(I(1)) - X(I(5))
R3 = X(I(2)) + X(I(8))
R4 = X(I(2)) - X(I(8))
R5 = X(I(3)) + X(I(7))
R6 = X(I(3)) - X(I(7))
R7 = X(I(4)) + X(I(6))
R8 = X(I(4)) - X(I(6))
T1 = R1 + R5
T2 = R1 - R5
T3 = R3 + R7
R3 =(R3 - R7) * C81
X(IP(1)) = T1 + T3
X(IP(5)) = T1 - T3
T1 = R2 + R3
T3 = R2 - R3
S1 = R4 - R8
R4 =(R4 + R8) * C81
S2 = R4 + R6
S3 = R4 - R6
R1 = Y(I(1)) + Y(I(5))
R2 = Y(I(1)) - Y(I(5))
R3 = Y(I(2)) + Y(I(8))
R4 = Y(I(2)) - Y(I(8))
R5 = Y(I(3)) + Y(I(7))
R6 = Y(I(3)) - Y(I(7))
R7 = Y(I(4)) + Y(I(6))
R8 = Y(I(4)) - Y(I(6))
T4 = R1 + R5
R1 = R1 - R5
```

```
R5 = R3 + R7
R3 =(R3 - R7) * C81
Y(IP(1)) = T4 + R5
Y(IP(5)) = T4 - R5
R5 = R2 + R3
R2 = R2 - R3
R3 = R4 - R8
R4 =(R4 + R8) * C81
R7 = R4 + R6
R4 = R4 - R6
X(IP(2)) = T1 + R7
X(IP(8)) = T1 - R7
X(IP(3)) = T2 + R3
X(IP(7)) = T2 - R3
X(IP(4)) = T3 + R4
X(IP(6)) = T3 - R4
Y(IP(2)) = R5 - S2
Y(IP(8)) = R5 + S2
Y(IP(3)) = R1 - S1
Y(IP(7)) = R1 + S1
Y(IP(4)) = R2 - S3
Y(IP(6)) = R2 + S3
C
GOTO 20

C-----------------WFTA N=9-----------------------
C
109 R1 = X(I(2)) + X(I(9))
R2 = X(I(2)) - X(I(9))
R3 = X(I(3)) + X(I(8))
R4 = X(I(3)) - X(I(8))
R5 = X(I(4)) + X(I(7))
T8 =(X(I(4)) - X(I(7))) * C31
R7 = X(I(5)) + X(I(6))
R8 = X(I(5)) - X(I(6))
T0 = X(I(1)) + R5
T7 = X(I(1)) + R5 * C95
R5 = R1 + R3 + R7
X(I(1)) = T0 + R5
T5 = T0 + R5 * C95
T3 = (R3 - R7) * C92
R7 = (R1 - R7) * C93
R3 = (R1 - R3) * C94
T1 = T7 + T3 + R3
T3 = T7 - T3 - R7
T7 = T7 + R7 - R3
T6 = (R2 - R4 + R8) * C31
T4 = (R4 + R8) * C96
R8 = (R2 - R8) * C97
R2 = (R2 + R4) * C98
T2 = T8 + T4 + R2
T4 = T8 - T4 - R8
```

```
T8 = T8 + R8 - R2
C
R1 = Y(I(2)) + Y(I(9))
R2 = Y(I(2)) - Y(I(9))
R3 = Y(I(3)) + Y(I(8))
R4 = Y(I(3)) - Y(I(8))
R5 = Y(I(4)) + Y(I(7))
R6 =(Y(I(4)) - Y(I(7))) * C31
R7 = Y(I(5)) + Y(I(6))
R8 = Y(I(5)) - Y(I(6))
T0 = Y(I(1)) + R5
T9 = Y(I(1)) + R5 * C95
R5 = R1 + R3 + R7
Y(I(1)) = T0 + R5
R5 = T0 + R5 * C95
T0 = (R3 - R7) * C92
R7 = (R1 - R7) * C93
R3 = (R1 - R3) * C94
R1 = T9 + T0 + R3
T0 = T9 - T0 - R7
R7 = T9 + R7 - R3
R9 = (R2 - R4 + R8) * C31
R3 = (R4 + R8) * C96
R8 = (R2 - R8) * C97
R4 = (R2 + R4) * C98
R2 = R6 + R3 + R4
R3 = R6 - R8 - R3
R8 = R6 + R8 - R4
C
X(IP(2)) = T1 - R2
X(IP(9)) = T1 + R2
Y(IP(2)) = R1 + T2
Y(IP(9)) = R1 - T2
X(IP(3)) = T3 + R3
X(IP(8)) = T3 - R3
Y(IP(3)) = T0 - T4
Y(IP(8)) = T0 + T4
X(IP(4)) = T5 - R9
X(IP(7)) = T5 + R9
Y(IP(4)) = R5 + T6
Y(IP(7)) = R5 - T6
X(IP(5)) = T7 - R8
X(IP(6)) = T7 + R8
Y(IP(5)) = R7 + T8
Y(IP(6)) = R7 - T8
C
GOTO 20

C-----------------WFTA N=16-------------------------
C
116 R1 = X(I(1)) + X(I(9))
R2 = X(I(1)) - X(I(9))
```

R3 = X(I(2)) + X(I(10))
R4 = X(I(2)) - X(I(10))
R5 = X(I(3)) + X(I(11))
R6 = X(I(3)) - X(I(11))
R7 = X(I(4)) + X(I(12))
R8 = X(I(4)) - X(I(12))
R9 = X(I(5)) + X(I(13))
R10= X(I(5)) - X(I(13))
R11 = X(I(6)) + X(I(14))
R12 = X(I(6)) - X(I(14))
R13 = X(I(7)) + X(I(15))
R14 = X(I(7)) - X(I(15))
R15 = X(I(8)) + X(I(16))
R16 = X(I(8)) - X(I(16))
T1 = R1 + R9
T2 = R1 - R9
T3 = R3 + R11
T4 = R3 - R11
T5 = R5 + R13
T6 = R5 - R13
T7 = R7 + R15
T8 = R7 - R15
R1 = T1 + T5
R3 = T1 - T5
R5 = T3 + T7
R7 = T3 - T7
X(IP(1)) = R1 + R5
X(IP(9)) = R1 - R5
T1 = C81 * (T4 + T8)
T5 = C81 * (T4 - T8)
R9 = T2 + T5
R11= T2 - T5
R13 = T6 + T1
R15 = T6 - T1
T1 = R4 + R16
T2 = R4 - R16
T3 = C81 * (R6 + R14)
T4 = C81 * (R6 - R14)
T5 = R8 + R12
T6 = R8 - R12
T7 = C162 * (T2 - T6)
T2 = C163 * T2 - T7
T6 = C164 * T6 - T7
T7 = R2 + T4
T8 = R2 - T4
R2 = T7 + T2
R4 = T7 - T2
R6 = T8 + T6
R8 = T8 - T6
T7 = C165 * (T1 + T5)
T2 = T7 - C164 * T1
T4 = T7 - C163 * T5

T6 = R10 + T3
T8 = R10 - T3
R10 = T6 + T2
R12 = T6 - T2
R14 = T8 + T4
R16 = T8 - T4
R1 = Y(I(1)) + Y(I(9))
S2 = Y(I(1)) - Y(I(9))
S3 = Y(I(2)) + Y(I(10))
S4 = Y(I(2)) - Y(I(10))
R5 = Y(I(3)) + Y(I(11))
S6 = Y(I(3)) - Y(I(11))
S7 = Y(I(4)) + Y(I(12))
S8 = Y(I(4)) - Y(I(12))
S9 = Y(I(5)) + Y(I(13))
S10= Y(I(5)) - Y(I(13))
S11 = Y(I(6)) + Y(I(14))
S12 = Y(I(6)) - Y(I(14))
S13 = Y(I(7)) + Y(I(15))
S14 = Y(I(7)) - Y(I(15))
S15 = Y(I(8)) + Y(I(16))
S16 = Y(I(8)) - Y(I(16))
T1 = R1 + S9
T2 = R1 - S9
T3 = S3 + S11
T4 = S3 - S11
T5 = R5 + S13
T6 = R5 - S13
T7 = S7 + S15
T8 = S7 - S15
R1 = T1 + T5
S3 = T1 - T5
R5 = T3 + T7
S7 = T3 - T7
Y(IP(1)) = R1 + R5
Y(IP(9)) = R1 - R5
X(IP(5)) = R3 + S7
X(IP(13)) = R3 - S7
Y(IP(5)) = S3 - R7
Y(IP(13)) = S3 + R7
T1 = C81 * (T4 + T8)
T5 = C81 * (T4 - T8)
S9 = T2 + T5
S11= T2 - T5
S13 = T6 + T1
S15 = T6 - T1
T1 = S4 + S16
T2 = S4 - S16
T3 = C81 * (S6 + S14)
T4 = C81 * (S6 - S14)
T5 = S8 + S12
T6 = S8 - S12

```
T7 = C162 * (T2 - T6)
T2 = C163 * T2 - T7
T6 = C164 * T6 - T7
T7 = S2 + T4
T8 = S2 - T4
S2 = T7 + T2
S4 = T7 - T2
S6 = T8 + T6
S8 = T8 - T6
T7 = C165 * (T1 + T5)
T2 = T7 - C164 * T1
T4 = T7 - C163 * T5
T6 = S10 + T3
T8 = S10 - T3
S10 = T6 + T2
S12 = T6 - T2
S14 = T8 + T4
S16 = T8 - T4
X(IP( 2)) = R2 + S10
X(IP(16)) = R2 - S10
Y(IP( 2)) = S2 - R10
Y(IP(16)) = S2 + R10
X(IP( 3)) = R9 + S13
X(IP(15)) = R9 - S13
Y(IP( 3)) = S9 - R13
Y(IP(15)) = S9 + R13
X(IP( 4)) = R8 - S16
X(IP(14)) = R8 + S16
Y(IP( 4)) = S8 + R16
Y(IP(14)) = S8 - R16
X(IP( 6)) = R6 + S14
X(IP(12)) = R6 - S14
Y(IP( 6)) = S6 - R14
Y(IP(12)) = S6 + R14
X(IP( 7)) = R11 - S15
X(IP(11)) = R11 + S15
Y(IP( 7)) = S11 + R15
Y(IP(11)) = S11 - R15
X(IP( 8)) = R4 - S12
X(IP(10)) = R4 + S12
Y(IP( 8)) = S4 + R12
Y(IP(10)) = S4 - R12
C
GOTO 20
C
20 CONTINUE
10 CONTINUE
RETURN
END
```

Contributor

- ContribEEBurrus

14.4: Appendix 4 - Programs for Short FFTs

This appendix will discuss efficient short FFT programs that can be used in both the The Cooley-Tukey Fast Fourier Transform Algorithm and the The Prime Factor and Winograd Fourier Transform Algorithms. Links and references are given to Fortran listings that can be used "as is" or put into the indexed loops of existing programs to give greater efficiency and/or a greater variety of allowed lengths. Special programs have been written for lengths: $N = 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19, 25, \ldots$

In the early days of the FFT, multiplication was done in software and was, therefore, much slower than an addition. With modem hardware, a floating point multiplication can be done in one clock cycle of the computer, microprocessor, or DSP chip, requiring the same time as an addition. Indeed, in some computers and many DSP chips, both a multiplication and an addition (or accumulation) can be done in one cycle while the indexing and memory access is done in parallel. Most of the algorithms described here are not hardware architecture specific but are designed to minimize both multiplications and additions.

The most basic and often used length FFT (or DFT) is for N=2N=2" role="presentation" style="position:relative;" tabindex="0">$N = 2$. In the Cooley Tukey FFT, it is called a "butterfly" and its reason for fame is requiring no multiplications at all, only one co

Code for other short lengths such as the primes N=3, 5, 7, 11, 13, 17, and 19N=3, 5, 7, 11, 13, 17, and 19" role="presentation" style="position:relative;" tabindex="0">$N = 3, 5, 7, 11, 13, 17, 19$and the

If these short FFTs are used as modules in the basic prime factor algorithm (PFA), then the straight forward development used for the modules in Figure 17.12 are used. However if the more complicated

For each of the indicated lengths, the computer code is given in a Connexions module.

They are not in the collection Fast Fourier Transforms as the printed version would be too long. However, one can link to them on-line from the following buttons: `N=2` `N=3` `N=4` `N=5` `N=7` `N= 8`

Contributor

- ContribEEBurrus

Index

Index

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Fast Fourier Transforms (Burrus)

Webpages: 95

All licenses found:

- CC BY 4.0: 78.9% (75 pages)
- Undeclared: 21.1% (20 pages)

By Page