UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 282 **Prof. R. Fateman**
Spring, 2000

The (finite field) Fast Fourier Transform

0.1 Introduction

There are numerous directions from which one can approach the subject of the fast Fourier Transform (FFT). It can be explained via numerous connections to convolution, signal processing, and various other properties and applications of the algorithm.

We (along with Geddes/Czapor/Labahn) take a rather simple view from the algebraic manipulation standpoint. As will be apparent shortly, we relate the FFT to the evaluation of a polynomial. We also consider it of interest primarily as an algorithm in a discrete (finite) computation structure rather than over the complex numbers.

Given a univariate polynomial which we wish to evaluate at a point x, there are many ways to rearrange the calculation. For example, we can re-write

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

in the form

$$A(x) = (x^0 x^1 ... x^{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_{n-1} \end{pmatrix}$$

Using Horner's rule, we can rewrite this as

$$A(x) = a_0 + x(a_1 + x(a_2 + ... + x a_{n-1} ...)).$$

which can be shown to be optimal under reasonable conditions for evaluating a polynomial at a point. As explained for example in Borodin and Munro [1] p. 8–9, it will require $n-1$ multiplications and $n-1$ additions. Although there are various tricks possible to pre-condition the polynomial we cannot in our circumstances, generally use such schemes economically: the algorithms we deal with will be handed the polynomial coefficients and the point x at the same time. We also prefer to do exact arithmetic over the integers. (see for example, Brassard [2]

pp. 209 which illustrates how a monic polynomial p of degree $n = 2^k - 1$ can be re-expressed as

$$p(x) = (x^{(n+1)/2} + a)q(x) + r(x)$$

where a is a constant and $q(x)$ and $r(x)$ are monic polynomials of degree $2^{k-1} - 1$. Recursively rearranged in this way, evaluation takes about $(n-3)/2 + \log_2(n)$ operations, but these are not integer operations any more!).

Nevertheless, if we keep the polynomial fixed, and evaluate at many points, we can do better, as we will see. For specificity in notation, let us evaluate $A(x)$ at n given points, $x_0, ..., x_{n-1}$.

Let us represent the problem as a matrix multiplication:

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ ... \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} x_0^0 & x_0^1 & ... & x_0^{n-1} \\ x_1^0 & x_1^1 & ... & x_1^{n-1} \\ ... & ... & ... & ... \\ x_{n-1}^0 & x_{n-1}^1 & ... & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix}$$

It is especially convenient for subsequent operations to choose the points specially:
Definition: ω_n is a primitive n^{th} root of unity in a computation structure (e.g. finite field), if $\omega_n^n = 1$ but for no m such that $0 < m < n$ is $\omega_n^m = 1$.
Examples: In the finite field of the integers mod 41 ($\mathbf{Z}_{41}$), the element 3 is a primitive 8^{th} root of unity. In $\mathbf{Z}_3$ the element 2 is a square-root of unity, since $2^2 = 4 \equiv 1 \bmod 3$.
Definition: A Fourier Transform (FT) is a sequence of n points constituting the evaluation of a polynomial with coefficients $\{a_0, ..., a_{n-1}\}$ at the points $\{\omega^0, ..., \omega^{n-1}\}$ where ω is a primitive n^{th} root of unity. That is, the Fourier Transform can be thought of as the left hand side of the equation below:

$$\begin{pmatrix} A(1) \\ A(\omega) \\ ... \\ A(\omega^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega & \omega^2 & ... & \omega^{n-1} \\ ... & ... & ... & ... & ... \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & ... & \omega^{(n-1)^2} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_{n-1} \end{pmatrix}$$

For convenience, we will denote the matrices above as:

$$\hat{\mathbf{A}} \;=\; \mathbf{F}_n \cdot \mathbf{A}$$

Note that if ω is a primitive n^{th} root of unity then in the complex plane $\omega = e^{2i\pi/n}$ and we have the usual "engineering" Fourier transform. We will not consider ω to be a complex number, but rather an element of a finite field.

In order to accomplish this matrix multiplication more rapidly, thus making it a *Fast* Fourier transform, we use the clever relationship pointed out by Cooley and Tukey in their 1965 paper [4] popularizing (and naming) the FFT. Let n be an even number, $2k$ and define two subsidiary polynomials:

$$A_{\text{even}}(x) \;=\; \sum_{i=0}^{k-1} a_{2i} x^i$$

$$A_{\text{odd}}(x) \;=\; \sum_{i=0}^{k-1} a_{2i+1}x^i$$

Then we can see that

$$A(x) = A_{\text{even}}(x^2) \;+\; xA_{\text{odd}}(x^2).$$

Hence

$$\hat{\mathbf{A}}_n = \mathbf{F}_{2k}\cdot\mathbf{A}_{2k} = \begin{pmatrix} A(1) \\ A(\omega) \\ \dots \\ A(\omega^{k-1}) \\ A(\omega^k) \\ A(\omega^{k+1}) \\ \dots \\ \\ A(\omega^{2k-1}) \end{pmatrix} = \begin{pmatrix} A_{\text{even}}(1) + A_{\text{odd}}(1) \\ A_{\text{even}}(\omega^2) + \omega A_{\text{odd}}(\omega^2) \\ \dots \\ A_{\text{even}}((\omega^{k-1})^2) + \omega^{k-1}A_{\text{odd}}((\omega^{k-1})^2) \\ A_{\text{even}}(1) + \omega^k A_{\text{odd}}(1) \\ A_{\text{even}}(\omega^2) + \omega^{k+1}A_{\text{odd}}(\omega^2) \\ \dots \\ A_{\text{even}}((\omega^2)^{k-1}) + \omega^{2k-1}A_{\text{odd}}((\omega^2)^{k-1}) \end{pmatrix}$$

Note that we have, in the last three displayed rows, made use of the facts that $(\omega^2)^k = \omega^{2k} = \omega^n = 1$, $(\omega^{k+1})^2 = \omega^{2k+2} = \omega^2$, and $(\omega^{2k-1})^2 = \omega^{4k-2} = \omega^{2k-2} = (\omega^2)^{k-1}$. The pattern in this formula can be made a bit more explicit. Let us use the symbol $\circ$ to denote component-wise matrix multiplication, and think about this in terms of block matrix multiplication where $\mathbf{A}_{\text{even}}$ is a column matrix of the coefficients of A_{even}, and

$$\mathbf{F}_{2k}\cdot\mathbf{A} = \begin{pmatrix} \mathbf{F}_k\cdot\mathbf{A}_{\text{even}} \\ \mathbf{F}_k\cdot\mathbf{A}_{\text{even}} \end{pmatrix} + \begin{pmatrix} 1 \\ \omega \\ \omega^2 \\ \dots \\ w^{2k-1} \end{pmatrix} \circ \begin{pmatrix} \mathbf{F}_k\cdot\mathbf{A}_{\text{odd}} \\ \mathbf{F}_k\cdot\mathbf{A}_{\text{odd}} \end{pmatrix}$$

Now we have replaced our $2k$-transform by two k-transforms, and some ($O(n)$) additions and computations of powers of ω. Actually, we will normally have an opportunity to pre-compute these powers of ω, so that is not considered as the same kind of cost. The smaller transforms can be computed by a recursive application of the same algorithm if n was originally a power of 2. We will find that in applications, we can increase n to fulfill this requirement, or modify the requirement some other way.

More succinctly, where all the $\mathbf{A}$ column vectors are implicitly $\mathbf{A}_n$ we have

$$\widehat{\mathbf{A}} = \begin{pmatrix} \widehat{\mathbf{A}_{\text{even}}} \\ \widehat{\mathbf{A}_{\text{even}}} \end{pmatrix} + \begin{pmatrix} 1 \\ \omega \\ \omega^2 \\ \dots \\ w^{n-1} \end{pmatrix} \circ \begin{pmatrix} \widehat{\mathbf{A}_{\text{odd}}} \\ \widehat{\mathbf{A}_{\text{odd}}} \end{pmatrix}$$

0.2 How fast is it?

A principal item of interest is the running time for the FFT. If the time to perform an FFT of size $2k$ is $T(2k)$, then from our expression above,

$$T(2k) = 2T(k) + O(k).$$

Iterating this formula gives (where $r = \log_2 k$):

$$T(2k) = 2^r T(1) + O(rk) = O(n \log n).$$

Thus this trick reduces the $O(n^2)$ algorithm (of n applications of Horner's rule), to $O(n \log n)$ operations.

As in the case of the usual Fourier transform, we would like to have an inverse transform. In our view, the inverse transform corresponds to polynomial interpolation. That is, we are given a set of evaluation points (powers of a root of unity) and asked to provide a polynomial which assumes given values at those points. The inverse of $\mathbf{F}_n$ is a matrix $\mathbf{G}_n$ where the i, j^{th} element is ω_n^{-ij}/n, where we have, somewhat unconventionally, numbered the rows and columns so that the 0,0 element is at the top left. We can write it out as

$$\mathbf{G}_n = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega^{-1} & \omega^{-2} & ... & \omega^{-(n-1)} \\ ... & ... & ... & ... & ... \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & ... & \omega^{-(n-1)^2} \end{pmatrix}$$

A proof is not difficult to construct: in the i, jth position of $\mathbf{F} \cdot \mathbf{G}$ we have

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{ik} \omega^{-jk}.$$

If $i = j$ this is just 1, and if $h = i - j \neq 0$, we have the geometric series

$$\frac{1}{n} \sum_{k=0}^{n-1} \omega^{hk} = \frac{1}{n} \frac{(\omega^h)^n - 1}{(\omega^h) - 1} = 0.$$

It's zero because $(\omega^h)^n - 1 = (\omega^n)^h - 1 = 1^h - 1 = 0$. In general, we would like to perform the FFT and the inverse FFT for some finite computation structure. A sufficiently large finite field has more than adequate structure: some less restrictive conditions are possible. For example the only elements that need to have multiplicative inverses in the computation structure are ω and the number of points, n.

The choice of a root of unity is a somewhat mysterious decision to make. Various tables can be precomputed for various structures, but in fact even if we use fields, we find there are plenty of suitable structures. For example,

> There is a 2^r th root of unity in a field $\mathbf{Z}_m$ (m prime) if $m = c2^r + 1$. E.g. 3 is a primitive 8th root of 1 in $\mathbf{Z}_{41}$.

> There are about 50 primes $m < 2^{31}$ of the form $c2^r + 1$ where $r \leq 20$. That is, even among the "single-computer-word" primes, there are plenty of primes of that form.

Thus if we wish to use an FFT, we can have a pre-computed list of 2^{20} th roots of unity (allowing about a 10^6 point transform). Moreover, $\omega_{2^{19}}, \omega_{2^{18}}$, etc. are trivially computed by squaring $\omega_{2^{20}}$, etc. so that we have roots of all smaller power-of-two sizes.

As an example of how we can use the FFT for evaluation and interpolation in a typical algebraic manipulation context, consider the problem of powering a univariate polynomial: $(A(x))^3 = (x + 10)^3$. We can represent this problem as follows (we will use superscript t to indicate matrix transposes):

$$\mathbf{A} = (10\ 1\ 0\ 0)^t.$$

Note that we padded out the sequence to length 4, not because we needed it to encode $A(x)$, but we anticipate that we will need it to encode $(A(x))^3$, which is of degree 3, and (counting the coefficient of x^0) has 4 terms.

The next step is to compute the Fourier Transform, in some domain which we need not specify at the moment:

$$\mathbf{F}_4 \cdot \mathbf{A} = \hat{\mathbf{A}} = (\hat{a}_0\ \hat{a}_1\ \hat{a}_2\ \hat{a}_3)^t$$

Then to cube $A(x)$, we in effect cube $A(1)$, $A(\omega)$, $A(\omega^2)$, ... and then interpolate. The cubing step is the computation of:

$$\hat{\mathbf{A}}^{(3)} = (\hat{a}_0^3\ \hat{a}_1^3\ \hat{a}_2^3\ \hat{a}_3^3)^t$$

and then finally, we interpolate those values, by computing the inverse transform:

$$\mathbf{F}_4^{-1} \cdot \hat{\mathbf{A}}^{(3)} = (1000\ 300\ 30\ 1)^t.$$

Obviously, performing this FFT procedure by hand will be significantly slower than the usual method of multiplication (or expansion by the binomial theorem). And thus if someone asks you to compute the FFT of something, consider going back to some other form – even the matrix definition we started with (the "Slow" Fourier transform) might be faster and less error prone by hand. But even using a computer, the bookkeeping for such a small problem would probably make a straight-forward method of polynomial multiplication faster. However, if A were a polynomial of degree 20, then this FFT might be faster than the conventional method: even though the FFT would have to be bumped up to size 64, a power of two, and therefore larger than necessary to represent the answer. However, if A were not dense, but sparse, say $A(x) = x^{1000} + 10$, then the computation via FFT of A^3 would require a transform of some 4096 elements, even though we can do it "by eye" quite rapidly. This suggests that the FFT is very bad for sparse polynomials because its running time is dependent upon the degree of the answer. This is not a *natural* measure of the size of a sparse polynomial; by constrast the "high school" algorithm depends mostly on the number of non-zero coefficients, which is often more reasonable. This type of behavior (good for dense polynomials, bad for sparse) as well as the nagging details of bignum arithmetic in finite fields, has usually provided a rationale for avoiding the general use of the discrete FFT in general purpose algebra systems.

There are interesting variations on the FFT to speed it up yet more, and to overcome the "power-of-two" restrictions. (for example, see Moenck [6].) The improvements to the FFT

discovered by S. Winograd [**?**] have not, to our knowledge, been implemented in the context of algebraic manipulation.

The generalization of the FFT to multivariate polynomials, and to a variety of polynomial problems, is discussed by R. Bonneau in his MIT thesis [3]. Various papers have tried to identify the sizes of polynomials for which the theoretical advantages actually translate into practical improvement (R. Moenck [6], R. Fateman [5]). The tradeoff points naturally depend on details of implementation.

0.3 The FFT Algorithm

Input: k, A power of 2.

$\mathbf{A}$, a sequence of length k of elements in $\mathbf{S}$, a finite computation structure.

ω_k, a kth root of unity in $\mathbf{S}$.

Output: $\hat{\mathbf{A}}$

0. If $k = 1$ then return $\hat{\mathbf{A}} = \mathbf{A}$.

1. [split] $\mathbf{B} := even(\mathbf{A})$; $\mathbf{C} := odd(\mathbf{A})$; {Separating the original sequence into two subsequences}.

2. [recursive steps] $\hat{\mathbf{B}} := \mathrm{FFT}(k/2, \mathbf{B}, \omega_k^2)$;
   $\hat{\mathbf{C}} := \mathrm{FFT}(k/2, \mathbf{C}, \omega_k^2)$

3. For i:=0 to k/2-1 do
   begin
   $\hat{\mathbf{A}}_i := \hat{\mathbf{B}}_i + \omega_k^i \hat{\mathbf{C}}_i$;
   $\hat{\mathbf{A}}_{i+k/2} := \hat{\mathbf{B}}_i + {\omega_k}^{i+k/2} \hat{\mathbf{C}}_i$
   end

A number of programming tricks can be used to speed this up, in addition to a certain number of less obvious transformations. For example, $\omega_k^{k/2} = -1$ in any ring supporting a k-point FFT, and the computation in the inner loop of step 3 can be improved to:

3′ $m := 1; i := 0; k2 := k/2;$
   Repeat
   $\hat{\mathbf{A}}_i := \hat{\mathbf{B}}_i + m\hat{\mathbf{C}}_i$;
   $\hat{\mathbf{A}}_{i+k2} := \hat{\mathbf{B}}_i - m\hat{\mathbf{C}}_i$;
   $i := i + 1; m := m\omega_k$
   until $m = 1$;

Other tricks worth noting:

a) For $k = 2$, $\hat{\mathbf{A}} = (a_0 + a_1, a_0 - a_1)$, shortening the recursion.

b) If $\mathbf{A} = (0, 0, ..., 0)$, $\hat{\mathbf{A}} = \mathbf{A}$.

c) If $\mathbf{A} = (a_0, 0, ..., 0)$, $\hat{\mathbf{A}} = (a_0, a_0, ..., a_0)$.

d) If $\mathbf{A} = (a, a, ..., a)$, $\hat{\mathbf{A}} = (ka, 0, ..., 0)$. "The DC case."

e) Non-recursive versions of this basic idea are generally used, since the data structure for sequences is usually implemented as sequential array allocation, and the programming language does not implement recursion efficiently. The recursion is therefore turned into clever indexing. Note, for example, that in separating even and odd elements, we could just as easily say "all elements whose last bits are 0" or ".. 1". And that even(even(A)) is "all elements whose last two bits are 00". etc.

   (Bit-reversal indexing is a small industry. For more discussion see e.g. J. Demmel's CS267 notes.)

   The interested reader may find it amusing to compare industrial-strength, or even academic "public-domain" FFT code to the relative simplicity of a direct implementation of the recursive program.

   You should be able to find Fortran or C language versions of the complex FFT, using various clever indexing schemes, in any scientific subroutine library. (see netlib, mit benchmark programs, Bailey's NAS code).

   The very best implementations of the FFT take into account particular features of host machines including the interference of memory-bank references, pipe-lining, instruction and memory hierarchies (cache, external storage), vector or distributed multi-processing communication costs, etc.

   There are also special chips built for FFTs, but these are, so far as we know, intended for signal or image processing, not computer algebra applications!

1 Some actual data

Actual data suggest that if one is in fact interested in multiplying univariate polynomials over finite fields (chosen by the algorithm), then by implementing the FFT in a more suitable language than Lisp, (e.g. C), and using appropriate data structures (arrays rather than lists), a significant speed-up can be obtained. For modest sizes of input with finite-field arithmetic a major speed-up is actually obtained just by using C and arrays, even if we use an n^2 algorithm!. In one test we squared the polynomial $(x + 1)^n$ for various n. We compared (i) Macsyma's standard LISP program, (ii) a program that converted the polynomial to an array and used C to implement standard high-school multiplication, and then converted back to a LISP list, and (iii), an FFT program written in C.

Basically, Macsyma's LISP program was slowest for all n, and some thirty times slower for $n = 64$. The FFT was slower than the high-school method (except for isolated "good sizes") when $n > 96$. After that point, the FFT seemed to be superior.

Asymptotically faster algorithms are often possible where there is some core convolution that can be performed using the FFT rather than conventional n^2 multiplication. Whether this is actually a practical result depends heavily on the circumstances [6].

There are over 10,000 papers on the FFT that have been published in the last decade, and this bibliography references almost none of them. Use your web browser for finding a few more, by looking at

http://science.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/RNR-89-004/RNR-89-004.o.html

Should discuss: "Discrete" Winograd FFT.

2 Exercises

Exercise: Show that nearly the same analysis can be used for a power of 3 FFT, with asymptotically fast running time. Show how this can be used for an FFT over exactly 18 points. And how fast is that?

Exercise: Implement an FFT on your favorite machine and figure out how much of the time is actually spent on arithmetic, and how much on data movement or indexing.

reference: `http://theory.lcs.mit.edu/~fftw`

3 What about higher dimensions?

You can find substantial material on higher-dimensional FFTs, especial 2-dimensions where FFTs of image data provides an interesting encoding that is useful for signal enhancement, compression, etc. How does this map into our polynomial world?

For a 2-D polynomial, consider the following extension: from

$$A(x) = \sum_{i=0}^{mn-1} a_i x^i$$

to

$$B(x,y) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{m-1} b_{i,j} y^j \right) x^i.$$

Note that if we set $b_{i,j}$ to a_{im+j}, then $B(x, x^n)$ is the same function as $A(x)$. That is, we can think of B as a 2-dimensional polynomial with a main variable x and a secondary variable y that appears in the coefficients of powers of x, or alternatively, as another way of encoding A. Taking an FFT of B can be done by converting it, with suitable padding of n and m, into A. The rest of the FFT discussion still holds.

**** are we omitted some twiddle factors here?? ***

For some computational purposes (in particular, to take advantage of locality of memory reference) it is advantageous to map single-variable polynomials like A into 2-D ones. Or in the matrix version, to take A in an array and consider it as an n by m matrix. By naming

the subsidiary polynomials that are functions only of y, we can rewrite the equation for B above as:

$$B(x,y) + \sum_{i=0}^{n-1} q_i(y)x^i.$$

We compute an n point FFT on this. Instead of the usual adds we must add vectors q_i. Instead of the usual scalar multiply, we must multiply vectors by the same old scalars ω^i. Let this result be

$$B'(x,y) + \sum_{i=0}^{n-1} r_i(y)x^i.$$

We don't use the notation $\hat{B}$ here because we haven't yet completed the transform. In fact to complete the transform we can rewrite B' as

$$B'(x,y) = \sum_{j=0}^{m-1} \left(\sum_{i=0}^{n-1} b'_{i,j}x^i \right) y^j$$

and then perform an m-point FFT on this system. From a polynomial perspective, we have re-written the system as a polynomial with main variable y and subsidiary variable x. From the more common perspective, this rewriting of the polynomial is a matrix transpose, an operation which does no arithmetic but merely re-orients data that was arranged by rows into columns. The reason to do it is based on the observation that sequential access to memory locations that whose addresses are spaced apart by certain powers of two will cause contention between banks of memory or promote cache misses. A careful analysis of such a rearrangement in light of a particular machine architecture (cache or vector-register sizes) can speed the computation of an FFT considerably (e.g. for Cray-YMP, 30%, D. Bailey).

Note also that the padding of one (or more) dimensions of a higher-dimensional FFT multiplies the excess size by the the other dimensions, and runs up the costs substantially, so it may pay more to keep the sizes "tighter" than for the 1-D FFT.

References

[1] A. B. Borodin and J. I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, NY, NY. 1975.

[2] Gilles Brassard and Paul Bratley, *Algorithmics: Theory and Practice*, Prentice Hall, 1988. 1 1

[3] Richard J. Bonneau, *Fast Polynomial Operations using the Fast Fourier Transform*, Dept of Mathematics, MIT, PhD dissertation, 1974. 6

[4] J. W. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series" *Math Comp. 19* 29–301 (1965) 2

[5] R. Fateman, "A Case Study in Interlanguage Communication: Fast LISP Polynomial Operations written in 'C' " *Proc. of 1981 ACM Symp. on Symbolic and Algebr. Computation*, SYMSAC'81, 122–125. 6

[6] R. T. Moenck "Practical Fast Polynomial Multiplication" *Proc. of 1976 ACM Symp. on Symbolic and Algebr. Computation*, SYMSAC'76, 136–145. 5, 6, 8

4 Appendix

Here is a self-contained Macsyma version of an FFT that will work for sequences of points. e.g. `myfft(4,[f0,f1,f2,f3],%i)`. This is not actually fast, but it is remarkably small, and shows the result of computation. It can be tried out for various special cases, and can be used for an inverse-FFT, as well. Perhaps of more practical interest is a version we wrote this that spews out straight-line code for the FFT.

```
/* k is a power of 2, a is a sequence of length k, wk is a kth root of unity */

myfft(k,a,w) :=
if k=1 then a else
 block([b:myfft(k/2,even(a),w^2),
        c:myfft(k/2, odd(a),w^2)],
   append (makelist(b[i]+w^(i-1)*c[i],i,1,k/2),
           makelist(b[i]-w^(i-1)*c[i],i,1,k/2)))$

even(a):= if a=[] then a else cons(first(a),odd(rest(a))) $
odd(a) := if a=[] then a else even(rest(a))$
```