The Fast Fourier Transform

Per Brinch Hansen
Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu

SU-CIS-91-46

THE FAST FOURIER TRANSFORM

PER BRINCH HANSEN

December 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

The Fast Fourier Transform [1]

PER BRINCH HANSEN

Syracuse University, Syracuse, New York 13244

December 1991

This tutorial discusses the fast Fourier transform, which has numerous applications in signal and image processing. The FFT computes the frequency components of a signal that has been sampled at n points in $O(n \log n)$ time. We explain the FFT and illustrate it by examples and Pascal algorithms. We assume that you are familiar with elementary calculus.

Categories and Subject Descriptors: F.2.1 [**Numerical Algorithms and Problems**]: Computation of transforms

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Fast Fourier transform, Pascal algorithms

CONTENTS

[1]Copyright©1991 Per Brinch Hansen

INTRODUCTION

This tutorial discusses one of the most important algorithms in science and technology: the discrete Fourier transform (DFT), which has numerous applications in signal and image processing.

After a brief summary of the continuous Fourier transform we define the DFT. A straightforward DFT computation for n sampled points takes $O(n^2)$ time. The DFT is illustrated by examples and a Pascal algorithm.

The fast Fourier transform (FFT) computes the DFT in $O(n \log n)$ time using the divide-and-conquer paradigm. We explain the FFT and develop recursive and iterative FFT algorithms in Pascal.

The FFT has a long history [Cooley et al. 1967]. It became widely known when James Cooley and John Tukey rediscovered it in 1965. The vast literature on the FFT and its applications include Brigham [1974], Macnaghten and Hoare [1977], and Press et al. [1989].

We assume that you are familiar with elementary calculus.

1. MATHEMATICAL BACKGROUND

We begin by summarizing the theory of the Fourier transform but will only attempt to make the results plausible. You will find a rigorous analysis in [Courant and John, 1989].

1.1 Fourier Series

We consider a physical process that can be described as a continuous function of time. We will call this function a *signal.*

A *periodic* signal $a(t)$ repeats itself after a period T

$$a(t+T) = a(t) \quad \text{for all } t$$

as illustrated by Fig. 1.

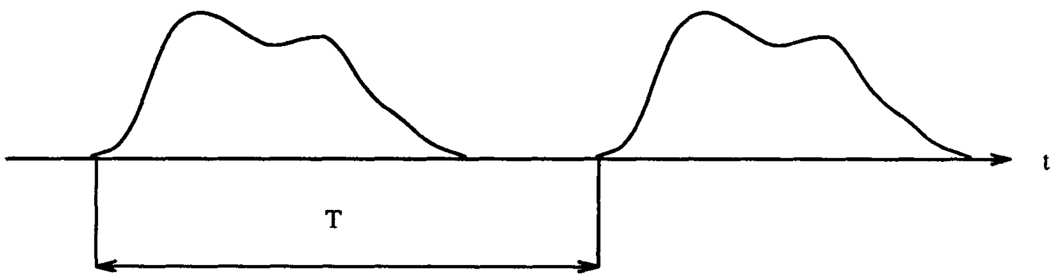

Fig. 1. A periodic signal

Any periodic signal $a(t)$ that we ordinarily encounter in physics or engineering can be written as a *Fourier series*—the sum of an infinite number of cosine and sine waves.

Since the algebra of complex exponentials is much simpler than that of cosines and sines we will express the Fourier series as the sum of *complex harmonics*

$$a(t) = \sum_{k=-\infty}^{\infty} c_k e^{-i2\pi f_k t} \tag{1}$$

where

$$e^{\pm i2\pi f_k t} = \cos(2\pi f_k t) \pm i \sin(2\pi f_k t)$$

Here $e = 2.71828\ldots$ is the base of the natural logarithm, and $i = \sqrt{-1}$ is the imaginary unit.

The discrete *frequencies*

$$f_k = k/T \text{ for } k = 0, \pm1, \pm2 \ldots.$$

are multiples of the lowest frequency $1/T$.

The Fourier coefficients c_k are generally complex numbers. To find a particular coefficient c_j we multiply both sides of Eq. (1) by

$$e^{i2\pi f_j t}$$

and average both sides over one period.

The right side is the sum of averages of the form

$$\frac{1}{T}\int_{-T/2}^{T/2} c_k e^{i2\pi(f_j - f_k)t} dt$$

For $k = j$ the exponential is 1 and the corresponding term has the value c_j. For any other k the average value of a harmonic wave over $j - k$ periods is zero.

Consequently

$$c_j = \frac{1}{T}\int_{-T/2}^{T/2} a(t) e^{i2\pi f_j t} dt \tag{2}$$

1.2 Fourier Transform

A *pulse* is a signal of finite duration as shown in Fig. 2.

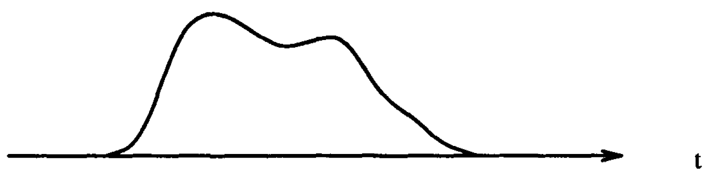

Fig. 2. A nonperiodic signal.

How do we handle such a *nonperiodic* signal? The trick is to pretend that a pulse is periodic as shown in Fig. 1 and then let the period T approach infinity without changing the shape and width of the pulse.

For a periodic signal the frequency increment is

$$\triangle f = 1/T$$

As $T \rightarrow \infty$ and $\triangle f \rightarrow 0$ we obtain a continuous spectrum of frequencies f.

To help in making the transition towards infinity, we will use Eq. (2) to express a Fourier coefficient c_k as the product of a function value $b(f_k)$ and the frequency increment $\triangle f$

$$c_k = b(f_k)\triangle f$$

$b(f_k)$ is the value of a function $b(f)$ for the discrete frequency $f = f_k$.

From Eq. (2) it follows that the appropriate function is

$$b(f) = \int_{-T/2}^{T/2} a(t)e^{i2\pi ft}dt$$

The Fourier series (1) can now be expressed as

$$a(t) = \sum_{k=-\infty}^{\infty} b(f_k)e^{-i2\pi f_k t}\triangle f$$

As $T \rightarrow \infty$ we obtain the *Fourier transform*

$$b(f) = \int_{-\infty}^{\infty} a(t)e^{i2\pi ft}dt \tag{3}$$

which defines the frequency *spectrum* $b(f)$ of the signal $a(t)$.

The *inverse transform*

$$a(t) = \int_{-\infty}^{\infty} b(f)e^{-i2\pi ft}df \tag{4}$$

defines the signal $a(t)$ as a function of its spectrum [Press et al. 1989].

2. DISCRETE FOURIER TRANSFORM

The Fourier transform defines the frequency components of a *continuous* signal. When a signal is sampled and analyzed on a computer we must use the corresponding *discrete Fourier transform* (DFT).

2.1 Definition

Figure 3 shows a signal that is sampled at n discrete intervals of fixed length $\triangle t$. We assume that the signal has been sampled competently, as explained by Brigham [1974].

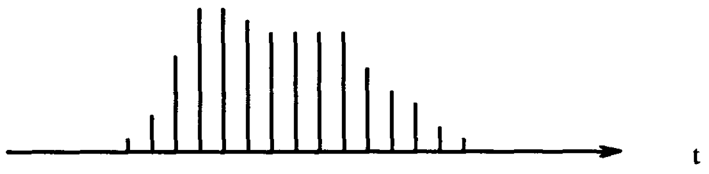

Fig. 3. A sampled signal.

The n sampled points are kept in an array

$$a = [a_0\ a_1 \ldots a_{n-1}]$$

where

$$a_k = a(t_k) \text{ and } t_k = k\triangle t \quad \text{for } k = 0..n-1$$

We will use the sampled points to compute an approximation to the Fourier transform $b(f)$ at n discrete points. The discrete Fourier transform will be stored in another array

$$b = [b_0\ b_1 \ldots b_{n-1}]$$

where

$$b_j = b(f_j) \quad \text{and} \quad f_j = \frac{j}{n\triangle t} \quad \text{for } j = 0..n-1$$

Each discrete frequency f_j is a multiple of f_1, the inverse of the total sampling time $n\triangle t$.

The remaining step is to approximate the integral in (3) by a discrete sum

$$b_j = \sum_{k=0}^{n-1} a(t_k) e^{i2\pi f_j t_k} \triangle t$$

Without loss of generality we assume $\triangle t = 1$ and rewrite the sum as

$$b_j = \sum_{k=0}^{n-1} a_k e^{i2\pi jk/n} \tag{5}$$

We can simplify this equation by introducing the complex number

$$w(n) = e^{i2\pi/n} = \cos(2\pi/n) + i\sin(2\pi/n) \tag{6}$$

This number is an n^{th} root of unity in the complex plane, since

$$w(n)^n = e^{i2\pi} = 1$$

When the value of n is obvious from the context, we will write w instead of $w(n)$.

Examples

$$\begin{array}{lclcr} w(1) & = & e^{i2\pi} & = & 1 \\ w(2) & = & e^{i\pi} & = & -1 \\ w(4) & = & e^{i\pi/2} & = & i \end{array}$$

Using $w(n)$ we can express Eq. (5) as

$$b_j = \sum_{k=0}^{n-1} a_k w(n)^{jk} \quad \text{for} \quad j = 0..n-1 \tag{7}$$

[Press et al. 1989].

This formula shows that the DFT of n points is the product

$$b = F(n)a^T$$

of a matrix $F(n)$ and a transposed vector a^T.

The elements of the *Fourier matrix* $F(n)$ are powers of $w(n)$

$$F(n) = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w & w^2 & \ldots & w^{n-1} \\ 1 & w^2 & w^4 & \ldots & w^{2(n-1)} \\ & & & \ldots & \\ 1 & w^{n-1} & w^{2(n-1)} & \ldots & w^{(n-1)^2} \end{bmatrix}$$

The elements of the signal vector are the signal points

$$a^T = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \ldots \\ a_{n-1} \end{bmatrix}$$

Example

$$F(1) = [1]$$

$$a = [a_0]$$

$$b = [a_0]$$

Example

$$F(2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$a = [a_0 \ a_1]$$

$$b = [a_0 + a_1 \ a_0 - a_1]$$

Example

$$F(4) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

$$a = [a_0 \ a_1 \ a_2 \ a_3]$$

$$b = \begin{bmatrix} a_0 + a_2 + & a_0 - a_2 + & a_0 + a_2 - & a_0 - a_2 - \\ (a_1 + a_3) & i(a_1 - a_3) & (a_1 + a_3) & i(a_1 - a_3) \end{bmatrix}$$

The pairwise similarity of DFT points is no coincidence. It is the main idea behind the fast Fourier transform, which will be discussed later.

A numerical example may be helpful. The DFT of the four points

$$a = [0.07 \ \ 0.91 \ \ 0.32 \ \ 0.29]$$

is

$$b = [1.59, \ -0.25 + i0.62, \ -0.81, \ -0.25 - i0.62]$$

2.2 Iterative DFT

Algorithm 1 defines an iterative DFT based on Eq. (7). It is written in Pascal, which does not support complex arithmetic.

```
type
  complex = record re, im: real end;
  table = array [0..n-1] of complex;

procedure DFT(var a, b: table);
const pi = 3.14159265358979;
var j, k: integer; v: real;
  ak, bj, wj, wjk, x: complex;
begin
  for j := 0 to n − 1 do
  begin
    v := 2.0*pi*j/n;
    wj.re := cos(v); wj.im := sin(v);
    wjk.re := 1.0; wjk.im := 0.0;
    bj.re := 0.0; bj.im := 0.0;
    for k := 0 to n − 1 do
    begin
      { bj := bj + ak*wjk }
      ak := a[k];
      bj.re := bj.re +
        ak.re*wjk.re − ak.im*wjk.im;
      bj.im := bj.im +
        ak.re*wjk.im + ak.im*wjk.re;
      { wjk := wjk*wj }
      x.re :=
        wjk.re*wj.re − wjk.im*wj.im;
      x.im :=
        wjk.re*wj.im + wjk.im*wj.re;
      wjk := x
    end;
    b[j] := bj
  end
end
```

Algorithm 1

The local variables include

v	denoting	$2\pi j/n$
wj	denoting	$w^j = \cos(v) + i\sin(v)$
wjk	denoting	$w^{jk} = (w^j)^k$

The computation of powers of w_j by complex multiplication may accumulate rounding errors. So it is a good idea to use double-precision arithmetic for the DFT.

The run time of the DFT is $O(n^2)$.

3. FAST FOURIER TRANSFORM

If n is a power of two, the DFT can be computed by a much faster algorithm called the *fast Fourier Transform* (FFT). The FFT runs in $O(n \log n)$ time.

3.1 Definition

We split the n sampled points into even and odd numbered points

$$\begin{aligned} a'' &= [a_0\ a_2 \ldots a_{n-2}] \\ a' &= [a_1\ a_3 \ldots a_{n-1}] \end{aligned}$$

The DFTs of a' and a'' can be computed separately using the $n/2^{th}$ root of unity

$$w(n/2) = e^{i4\pi/n} = \left(e^{i2\pi/n}\right)^2 = w(n)^2$$

The equations will look less cluttered if we use the abbreviations

$$m = n/2 \quad u = w(n/2) \quad w = w(n)$$

According to Eq. (7) the DFT of a' is

$$b'_j = a_1 + a_3 u^j + a_5 u^{2j} + \cdots + a_{n-1} u^{(m-1)j}$$

which can be rewritten as

$$b'_j = a_1 + a_3 w^{2j} + a_5 w^{4j} + \cdots + a_{n-1} w^{(n-2)j} \tag{8}$$

for $j = 0..m-1$.

Similarly, the DFT of a'' is

$$b''_j = a_0 + a_2 w^{2j} + a_4 w^{4j} + \cdots + a_{n-2} w^{(n-2)j} \tag{9}$$

for $j = 0..m-1$.

The DFT of all n points is

$$\begin{aligned} b_j &= a_0 \quad + a_2 w^{2j} + \cdots + a_{n-2} w^{(n-2)j} \\ &+ \ a_1 w^j + a_3 w^{3j} + \cdots + a_{n-1} w^{(n-1)j} \end{aligned} \tag{10}$$

for $j = 0..n-1$. We have rearranged the terms by writing the even ones on the first line and the odd ones below.

Combining Eqs. (8)–(10) we obtain a method of computing the first $n/2$ points of the complete DFT from the DFTs of the odd and even points

$$b_j = b''_j + w^j b'_j \quad \text{for} \quad j = 0..m-1 \tag{11}$$

To compute the other half of the DFT we start by observing that

$$w^m = \left(e^{i2\pi/n}\right)^{n/2} = e^{i\pi} = -1$$

Consequently

$$w^{k(j+m)} = (-1)^k w^{kj} \tag{12}$$

If we replace j by $j+m$ in Eq. (10) and use Eq. (12) we get

$$\begin{array}{rcrcrcr} b_{j+m} & = & a_0 & + & a_2w^{2j} & + \cdots + & a_{n-2}w^{(n-2)j} \\ & - & a_1w^j & - & a_3w^{3j} & - \cdots - & a_{n-1}w^{(n-1)j} \end{array}$$

In short

$$b_{j+m} = b''_j - w^j b'_j \quad \text{for} \quad j = 0..m{-}1 \tag{13}$$

The computational idea behind the FFT is the combination of Eqs. (11) and (13)

$$\begin{array}{rcll} b_j & = & b''_j + w^j b'_j & \text{for } j = 0..m{-}1 \\ b_{j+m} & = & b''_j - w^j b'_j & \text{where } m = n/2 \end{array} \tag{14}$$

Since $n/2$ is also a power of two, we can use the same formula to compute b''_j and b'_j. The FFT applies this rule recursively until it reaches a level that involves the transforms of single points only.

Consider the FFT of four points. First, we split the samples into two halves consisting of the even and odd numbered points, respectively. Each half is then split into two samples of one point each. Figure 4 illustrates the recursive splitting of the computation into smaller and smaller FFTs.

$$\begin{tikzcd}
\mathrm{FFT}([a_0]) & & \mathrm{FFT}([a_2]) & & \mathrm{FFT}([a_1]) & & \mathrm{FFT}([a_3]) \\
& \mathrm{FFT}([a_0\ a_2]) \arrow[ul] \arrow[ur] & & & & \mathrm{FFT}([a_1\ a_3]) \arrow[ul] \arrow[ur] & \\
& & & \mathrm{FFT}(([a_0\ a_1\ a_2\ a_3]) \arrow[ull] \arrow[urr] & & &
\end{tikzcd}$$

Fig. 4 Splitting the samples

Since the transform of a single point is the point itself, the recursion stops when the FTTs are of length 1. The computation then combines four FFTs of length 1 into two FFTs of length 2 and, finally, two FFTs of length 2 into a single one of length 4. Figure 5 illustrates the recursive combination of FFTs.

$$\begin{tikzcd}
{[a_0]} \arrow[dr] & & {[a_2]} \arrow[dl] & & {[a_1]} \arrow[dr] & & {[a_3]} \arrow[dl] \\
& {[a_0 + a_2 \;\; a_0 - a_2]} \arrow[drr] & & & & {[a_1 + a_3 \;\; a_1 - a_3]} \arrow[dll] & \\
& & & \begin{bmatrix} a_0 + a_2 + & a_0 - a_2 + & a_0 + a_2 - & a_0 - a_2 - \\ (a_1 + a_3) & i(a_1 - a_3) & (a_1 + a_3) & i(a_1 - a_3) \end{bmatrix} & & &
\end{tikzcd}$$

Fig. 5 Combining the transforms

Before writing an FFT algorithm we will discuss several refinements.

3.2 In-Place Computation

It is possible to compute the FFT in place in a single array. The computation replaces the signal points by the corresponding transform.

The trick is to let the FFT computation operate on smaller and smaller slices of the same array. An array slice is split into two halves by moving the even and odd numbered points to the left and right halves, respectively (Fig. 6).

$$\begin{tikzcd}
{[a_0} & & a_2 & & a_1 & & {a_3]} \\
& \arrow[ul] \arrow[ur] & & & & \arrow[ul] \arrow[ur] & \\
{[a_0} & & a_2 & & a_1 & & {a_3]} \\
& & & \arrow[ul] \arrow[ur] & & & \\
{[a_0} & & a_1 & & a_2 & & {a_4]}
\end{tikzcd}$$

Fig. 6 In-place splitting

The transforms are then combined in place as illustrated by Fig. 7.

$$\begin{tikzcd}
{[a_0} \arrow[dr] & & a_2 \arrow[dl] & & a_1 \arrow[dr] & & {a_3]} \arrow[dl] \\
& {[\; a_0 + a_2 \quad a_0 - a_2} \arrow[drr] & & & & {a_1 + a_3 \quad a_1 - a_3 \;]} \arrow[dll] & \\
& & & \begin{bmatrix} a_0 + a_2 + & a_0 - a_2 + & a_0 + a_2 - & a_0 - a_2 - \\ (a_1 + a_3) & i(a_1 - a_3) & (a_1 + a_3) & i(a_1 - a_3) \end{bmatrix} & & &
\end{tikzcd}$$

Fig. 7 In-place combination

After this example, we will prove by induction that in-place computation is always possible:

Base step: The FFT of a single point can be done in place by leaving the point unchanged.

Induction step: Without loss of generality we consider a slice of n elements with indices from 0 to $n-1$. After splitting the slice into two halves we assume that it is possible to compute the two FFTs in place. We must now show that this hypothesis also makes it possible to compute the combined FFT in place.

Figure 8 shows the j^{th} elements b''_j and b'_j of the left and right FFTs. Their indices are j and $j+m$, respectively. According to Eq. (14) b''_j and b'_j are used once only to compute elements b_j and b_{j+m} of the combined FFT. Since the two "output" elements have the same indices as the two "input" elements, they can replace them in the array.

QED

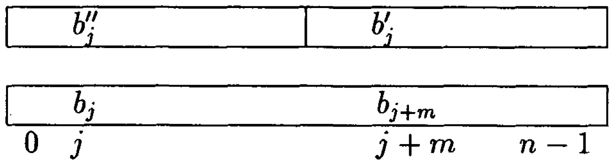

Fig. 8 FFT elements

3.3 Recursive FFT

Algorithm 2 defines a recursive, in-place computation of the FFT in a single array a.

```
procedure FFT(var a: table; first,
  last: integer);
var middle: integer;
begin
  if first < last then
  begin
    split(a, first, last, middle);
    FFT(a, first, middle);
    FFT(a, middle + 1, last);
    combine(a, first, last, middle)
  end
end
```

Algorithm 2

Each activation of the procedure computes the FFT of a slice of the array a. The slice is defined by the indices of its first and last elements, where

$$0 \leq \text{ first } \leq \text{ last } \leq n-1$$

The procedure activation

$$\text{FFT}(a,\ 0,\ n-1)$$

computes the transform of all n points.

If an array slice holds one point only, it is left unchanged since the point is its own transform. Otherwise, the computation splits the array into two halves, computes the FFT of each half separately, and combines them into a single FFT. The two halves have indices in the ranges first..middle and middle+1..last, respectively.

An array slice can be split into two halves using a local array b to rearrange the samples into even and odd numbered points (Algorithm 3).

```
procedure split(var a: table; first,
  last: integer; var middle: integer);
var even, half, size: integer;
  b: table;
begin
  middle := (first + last) div 2;
  size := last - first + 1 ;
  half := size div 2;
  for j := 0 to half - 1 do
  begin
    even := first + 2*j;
    b[j] := a[even];
    b[j + half] := a[even + 1]
  end;
  for j := 0 to size - 1 do
    a[first + j] := b[j]
end
```

Algorithm 3

Since Pascal does not support dynamic arrays, every local array must be of length n to be able to hold all n signal points. Altogether, the local arrays created by nested activations of the FFT and split procedures will occupy a memory space of $O(n \log n)$.

A single array b suffices if we are willing to pass it as a parameter to each call of FFT and split. Rather than doing that, we prefer to eliminate the split procedure completely. We will return to this problem later.

The combination of two FFTs into one based on Eqs. (6) and (14) is defined by Algorithm 4.

```
procedure combine (var a: table; first,
  last: integer);
const pi = 3.14159265358979;
var even, half, odd, j: integer;
  v: real; ae, ao, w, wj, x: complex;
begin
  half := (last - first + 1) div 2;
  v := pi/half;
  w.re := cos(v); w.im := sin(v);
  wj.re := 1.0; wj.im := 0.0;
  for j := 0 to half - 1 do
  begin
    even := first + j;
    odd := even + half;
    ae := a[even]; ao := a[odd];
    { x = wj*a[odd] }
    x.re := wj.re*ao.re - wj.im*ao.im;
    x.im := wj.re*ao.im + wj.im*ao.re;
    { a[odd] := a[even] - x;
      a[even] := a[even] + x }
    ao.re := ae.re - x.re;
    ao.im := ae.im - x.im;
    ae.re := ae.re + x.re;
    ae.im := ae.im + x.im;
    a[even] := ae; a[odd] = ao;
    { wj :=wj*w }
    x.re := wj.re*w.re - wj.im*w.im;
    x.im := wj.re*w.im + wj.im*w.re;
    wj := x
  end
end
```

Algorithm 4

3.4 Initial Permutation

If you compare the top and bottom of Fig. 6, you will see that the net effect of recursive splitting is to permute the sampled points before the FFTs are combined. This suggests that splitting can be replaced by a single permutation of the array before the FFT combinations begin (Algorithm 5).

```
type
  complex = record re, im: real end;
  table = array [0..n−1] of complex;

procedure DFT(var a: table);
begin
  permute(a);
  FFT(a, 0, n − 1)
end
```

Algorithm 5

Algorithm 6 defines the simplified FFT.

```
procedure FFT(var a: table; first,
    last: integer);
var middle: integer;
begin
    if first < last then
    begin
        middle := (first + last) div 2;
        FFT(a, first, middle);
        FFT(a, middle + 1, last);
        combine(a, first, last)
    end
end
```

Algorithm 6

How should we permute the array? We can get a hint from Fig. 9, which shows the end result of splitting eight sampled points in place.

a_0	a_4	a_2	a_6	a_1	a_5	a_3	a_7
0	1	2	3	4	5	6	7

Fig. 9. A permuted array

In an array of n sampled points the initial index of point a_j is j. The final index of a_j after the permutation is denoted

index(j,n) for j=0..n−1

Table 1 shows the initial and final indices of eight points in decimal (and binary) form.

Table 1

j	index(j,8)
0 (000)	0 (000)
1 (001)	4 (100)
2 (010)	2 (010)
3 (011)	6 (110)
4 (100)	1 (001)
5 (101)	5 (101)
6 (110)	3 (011)
7 (111)	7 (111)

The table suggests that the final index of a sampled point is obtained by reversing the order of the bits in the initial index

$$\text{index(j,n)} = \text{reverse(j,n)} \quad \text{for } \text{j=0..n}-1 \tag{15}$$

The reverse of a binary number consists of the last bit followed by the rest of the bits (if any) in reverse order

$$\text{reverse}(0,1) = 0$$

$$\text{reverse(2k,n)} = \text{reverse(k,n/2)}$$

$$\text{reverse(2k+1,n)} = \text{n/2} + \text{reverse(k,n/2)}$$

These rules define a recursive function (Algorithm 7)

```
function reverse(j, n: integer)
  : integer;
var half: integer;
begin
  if n = 1 then reverse := j else
  begin
    half := n div 2;
    reverse := (j mod 2)*half +
        reverse(j div 2, half)
  end
end
```

Algorithm 7

We will prove Eq. (15) by induction.

Base step: In an array of size 1 the single point with index 0 is already correctly placed

$$\text{index}(0,1) = 0 = \text{reverse}(0,1)$$

Induction step: We assume that Eq. (15) holds for arrays of size $n/2$ and will prove that it also holds for arrays of size n.

Consider a sampled point a_j when an array of size n is split into two halves. If j is even, say $j = 2k$, a_j is placed in the left half with index k. The left half is then split into half. In formal terms

$$\begin{aligned}\text{index}(2k,n) &= \text{index}(k,n/2)\\ &= \text{reverse}(k,n/2)\\ &= \text{reverse}(2k,n)\end{aligned}$$

If j is odd, say $j = 2k+1$, a_j is placed in the right half with index $n/2+k$. The right half is then split again. Consequently

$$\begin{aligned}\text{index}(2k{+}1,n) &= n/2 + \text{index}(k,n/2)\\ &= n/2 + \text{reverse}(k,n/2)\\ &= \text{reverse}(2k{+}1,n)\end{aligned}$$

Bit reversal is obviously symmetric: if $j = \text{reverse}(k,n)$, then $k = \text{reverse}(j,n)$, and vice versa. So the recursive splitting interchanges $a[j]$ and $a[k]$. This insight leads to Algorithm 8.

```
procedure permute(var a: table;
  n: integer);
var j, k: integer; aj: complex;
begin
  for j := 0 to n - 1 do
  begin
    k := reverse(j,n);
    if j < k then
    begin { swap(a[j],a[k]) }
      aj := a[j]; a[j] := a[k];
      a[k] := aj
    end;
  end
end
```

Algorithm 8

The condition $j < k$ prevents a pair of points from being swapped twice. It also eliminates superfluous swapping when an index and its reverse are the same.

Since every index is reversed in time $O(\log n)$, the permutation takes $O(n \log n)$ time.

The recursive FFT is the combination of Algorithms 4–8.

3.5 Fast Permutation

The divide-and-conquer nature of the FFT makes it well-suited for parallel computation [Brinch Hansen 1991; Fox et al. 1988]. However, if a parallel FFT is preceded by a sequential $O(n \log n)$ permutation, the combination is still an $O(n \log n)$ algorithm.

Fortunately, it is possible to permute n points in $O(n)$ time. A fast permutation begins by constructing a map of size n that holds the bit-reversed values of the indices $0..n-1$.

We will illustrate the computational pattern for an array of eight points (see Table 1):

The first element of the permutation map is

$$\mathrm{rev}_0 = 0$$

The next element is

$$\mathrm{rev}_1 = \mathrm{rev}_0 + 4 = 4$$

The next two elements are

$$\begin{aligned}\mathrm{rev}_2 &= \mathrm{rev}_0 + 2 = 2\\ \mathrm{rev}_3 &= \mathrm{rev}_1 + 2 = 6\end{aligned}$$

The last four elements are

$$\begin{aligned}\mathrm{rev}_4 &= \mathrm{rev}_0 + 1 = 1\\ \mathrm{rev}_5 &= \mathrm{rev}_1 + 1 = 5\\ \mathrm{rev}_6 &= \mathrm{rev}_2 + 1 = 3\\ \mathrm{rev}_7 &= \mathrm{rev}_3 + 1 = 7\end{aligned}$$

In each step the map is doubled by adding a power of two to each of the previous elements. The increment is halved after each step. This method was suggested by Tapas Som.

The formal basis of the method is the rule

$$\mathrm{reverse}(j+2^k, n) = \mathrm{reverse}(j, n) + n/2^{k+1}$$

for $k = 0..\log n-1$ and $j = 0..2^k-1$. The proof of this rule is left as an exercise for you.

The fast permutation constructs a map according to this rule and uses it to swap pairs of points in a signal array of size n (Algorithm 9).

```
procedure permute(var a: table);
type map = array [0..n−1] of integer;
var rev: map; aj: complex;
 incr, size, j, k: integer;
begin
 rev[0] := 0; size := 1;
 while size < n do
 begin
   incr := n div (2*size);
   for j := 0 to size − 1 do
     rev[j + size] := rev[j] + incr;
   size := 2*size
 end;
 for j := 0 to n − 1 do
 begin
   k := rev[j];
   if j < k then
   begin { swap(a[j], a[k]) }
     aj := a[j]; a[j] := a[k];
     a[k] := aj
   end
 end
end
```

Algorithm 9

If a complex number is represented by two 64-bit reals and an index by a 32-bit integer, the permutation map increases the memory requirement by 25%.

This recursive FFT is defined by Algorithms 4–6 and 9.

There are FFT algorithms that work when n is not a power of two. However, Press et al. [1989] recommend using the FFT only with n a power of two, if necessary by padding the data with zeros up to the next power of two.

3.6 Iterative FFT

The FFT computation can also be defined by an iterative algorithm that transforms the entire array $\log n$ times. The first transformation combines n slices of size 1 into $n/2$ slices of size 2. The second transformation combines $n/2$ slices of size 2 into $n/4$ slices of size 4, and so on. The last transformation combines 2 slices of size $n/2$ into a single slice of size n (Algorithm 10).

```
procedure FFT(var a: table; first,
  last: integer);
var size, k, m: integer;
begin
  m := last − first + 1;
  size := 2;
  while size <= m do
  begin
    k := first + size − 1;
    while k <= last do
    begin
      combine (a, k − size + 1, k);
      k := k + size
    end;
    size := 2*size
  end
end
```

Algorithm 10

The iterative FFT is the combination of Algorithms 4, 5, 8, and 10. (Algorithm 8 may be replaced by the equivalent Algorithm 9.)

4. SUMMARY

We have explained the fast Fourier transform (FFT) and have illustrated the algorithm by examples and Pascal algorithms. The FFT is yet another example of a fundamental computation with a subtle theory and a short algorithm.

ACKNOWLEDGEMENTS

I thank Jonathan Greenfield and Erik Hemmingsen for valuable comments that improved the paper.

REFERENCES

BRIGHAM, E. O. 1974. *The Fast Fourier Transform.* Prentice-Hall, Englewood Cliffs, NJ.

BRINCH HANSEN, P. 1991. Parallel divide and conquer. School of Computer and Information Science, Syracuse University, Syracuse, NY.

COOLEY, J. W., and TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19*, 297–301.

COOLEY, J. M., LEWIS, P. A., and WELSH, P. D. 1967. The history of the fast Fourier transform. *Proceedings of the IEEE 55*, 1675–1679.

COURANT, R., and JOHN, F. 1989. *Introduction to Calculus and Analysis.* Vols. I & II. Springer-Verlag, New York.

FOX, G. C., JOHNSON, M. A., LYZENGA, G. A., OTTO, S. W., SALMON, J. K., and WALKER, D. W. 1988. *Solving Problems on Concurrent Processors.* Vol. I. Prentice-Hall, Englewood Cliffs, NJ.

MACNAGHTEN, A. M., and HOARE, C. A. R. 1977. Fast Fourier transform. *Computer Journal* 20, 1, 78–83.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., and VETTERLING, W. T. 1989. *Numerical Recipes in Pascal. The Art of Scientific Computing.* Cambridge University Press, New York.