remarks on FFT algorithms

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at September 29, 1997

[1]This document is online at http://www.jjj.de/fxt/

Contents

Abstract

This is a collection of remarks about fast Fourier transform algorithms.

The object of this document is to show the basic ideas of some FFT algorithms so that the reader can code them in his/her favorite programming language.

Do not expect mathematics here.

For more information I recommend to see the books of Lipson [2], Nussbaumer [1], van Loan [3] and the original papers.

I am asking for feedback !

This preliminary version, i am still enhancing and improving it.

You will find unfinished or empty sections.

Please report errors, inconsitencies, typos and misleading things.

Chapter 1

The Fourier transform

1.1 Definition of the DFT and the FFT

The *discrete Fourier transform*[1] (DFT) of a sequence a of length n is defined as

$$c_k \quad := \quad \sum_{x=0}^{n-1} a_x \, e^{-2\,\pi\,i\,x\,k/n} \tag{1.1}$$

Backtransform is then

$$a_x \quad = \quad \frac{1}{n} \sum_{k=0}^{n-1} c_k \, e^{+2\,\pi\,i\,x\,k/n} \tag{1.2}$$

A straightforward implementation of the (discrete) Fourier transform, i.e. the computation of n sums each of length n would require $\sim n^2$ operations. A *fast Fourier transform* (FFT) algorithm is an algorithm that improves the operation count to $\sim n \log(n)$. There are several different FFT algorithms, with many variants.

1.2 Decimation in time (DIT) FFT

Some notation is useful here:
Let $\mathcal{F}_n[a]$ denote the Fourier transform of a sequence a of length n.
Let $a^{(even)}$ denote the subsequence of those elements of a that have even subscripts,
$a^{(odd)}$ respectively.
Let $a^{(left)}$ denote the subsequence of those elements of a that have indices $0...n/2-1$,
$a^{(right)}$ for indices $n/2...n-1$.
Let $\mathcal{S}a$ denote the sequence with elements $a_x \, e^{-\pi\,i\,x/n}$ where n is the length of the sequence a (the letter $\mathcal{S}$ shall suggest a shift operator).

The following observation is the key to the decimation in time (DIT) FFT[2] algorithm:
For n even

$$\sum_{x=0}^{n-1} a_x \, e^{-2\,\pi\,i\,x\,k/n} \quad = \quad \sum_{x=0}^{n/2-1} a_{2\,x} \, e^{-2\,\pi\,i\,(2\,x)\,k/n} + \sum_{x=0}^{n/2-1} a_{2\,x+1} \, e^{-2\,\pi\,i\,(2\,x+1)\,k/n} \tag{1.3}$$

[1] cf. the definition of the continuous and semi-continuous FT in appendix B
[2] also called Cooley-Tukey FFT.

$$= \sum_{x=0}^{n/2-1} a_x^{(even)}\, e^{-2\,\pi\, i\, x\, k/(n/2)} + e^{-\pi\, i\, k/(n/2)} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, e^{-2\,\pi\, i\, x\, k/(n/2)} \quad (1.4)$$

To rewrite the length-n FT in terms of length-$n/2$ FTs one has to distinguish the cases $0 \le k < n/2$ and $n/2 \le k < n$:

IDEA *radix 2 decimation in time step*: radix 2 DIT step

$$\mathcal{F}_n^{(left)}[a] = \mathcal{F}_{n/2}\left[a^{(even)}\right] + \mathcal{SF}_{n/2}\left[a^{(odd)}\right] \quad (1.5)$$

$$\mathcal{F}_n^{(right)}[a] = \mathcal{F}_{n/2}\left[a^{(even)}\right] - \mathcal{SF}_{n/2}\left[a^{(odd)}\right] \quad (1.6)$$

(Here it is silently assumed that '+' or '-' between two sequences denotes elementwise addition or subtraction.)

The length-n transform has been replaced by two transforms of length $n/2$. If n is a power of 2 this scheme can be applied recursively until length-one transforms (identity operation) are reached. Thereby the operation count is improved to $\sim n\,\log(n)$.

Pseudo code for a recursive procedure of the (radix 2) DIT algorithm:

```
procedure rec_fftdit2(a[],n,e,ao[])
{
    if n == 1 then return

    n := n/2;

    rec_fftdit2(even(a[]),n,e^2,b[])
    rec_fftdit2( odd(a[]),n,e^2,c[])

    foreach k in {0,1, ..., n-1}
    {
        ao[k]   := b[k] + c[k] * e^k
        ao[k+n] := b[k] - c[k] * e^k
    }
}
```

the procedure must be called with `e` $= exp(-2\,\pi\, i/n)$, the result is in `ao[]`. `b[]` and `c[]` are workspace arrays. `even()` and `odd()` shall denote what you expect.

Pseudo code[3] for a non-recursive procedure of the (radix 2) DIT algorithm, the data length `n` must be o power of 2, `is` must be +1 or -1:

```
procedure fftdit2(a[],n,is)
{
    scramble(a[],n)

    for ldm := 1 to log2(n) step 1
    {
        m  := 2^ldm
        mh := m/2

        for j := 0 to mh-1 step 1
        {
```

[3] adapted from [12]

```
            e := exp(is*2*pi*i*j/m)

            for r := 0 to n-1 step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e

                a[r+j] :=    u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

`n` must be a power of 2. The procedure `scramble(a[],n)` is assumed to rearrange the array `a[]` in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. (e.g. for $n = 256$: $x = 43_{10} = 00101011_2$ gives $\tilde{x} = 11010100_2 = 212_{10}$; note that $\tilde{x}$ depends both on x and on n)

```
procedure scramble(a[],n)
{
    for i=0 to n/2-1 step 1
    {
        r := revbin(i,n)
        if r > i then swap(a[i],a[r])
    }
}
```

The function `revbin(i,n)` is assumed to return the reversed bits of `i`.

The non-recursive procedure avoids the calling overhead and it works in-place (i.e. no extra workspace is required).

For the innermost loop we could have said equivalently `for r := 0 to n-m step m` instead of `for r := 0 to n-1 step m` .

1.3 Decimation in frequency (DIF) FFT

The simple splitting
(for n even)

$$\sum_{x=0}^{n-1} a_x \, e^{-2\,\pi\,i\,x\,k/n} \;=\; \sum_{x=0}^{n/2-1} a_x \, e^{-2\,\pi\,i\,x\,k/n} + \sum_{x=n/2}^{n} a_x \, e^{-2\,\pi\,i\,x\,k/n} \tag{1.7}$$

leads to the decimation in frequency (DIF) FFT[4].

Here one has to distinguish the cases k even and k odd:

IDEA *radix 2 decimation in frequency step*: radix 2 DIF step

$$\mathcal{F}_n^{(even)}[a] \;=\; \mathcal{F}_{n/2}\left[a^{(left)} + a^{(right)}\right] \tag{1.8}$$

$$\mathcal{F}_n^{(odd)}[a] \;=\; \mathcal{F}_{n/2}\left[\mathcal{S}\left(a^{(left)} - a^{(right)}\right)\right] \tag{1.9}$$

[4]also called Sande-Tukey FFT, cf. [12].

Pseudo code for a recursive procedure of the (radix 2) DIF algorithm:

```
procedure rec_fftdit2(a[],n,e,ao[])
{
    if n == 1 then return

    n := n/2;

    rec_fftdif2( left(a[]),n,e^2,b[])
    rec_fftdif2(right(a[]),n,e^2,c[])

    foreach k in {0,1, ..., n-1}
    {
        ao[k]   :=  b[k] + c[k]
        ao[k+n] := (b[k] - c[k]) * e^k
    }
}
```

the procedure must be called with `e` $= exp(-2\,\pi\,i/n)$, the result is in `ao[]`. `b[]` and `c[]` are workspace arrays. `left()` and `right()` shall denote what you expect.

Pseudo code[5] for a non-recursive procedure of the (radix 2) DIF algorithm, the data length `n` must be a power of 2, `is` must be +1 or -1:

```
procedure fftdif2(a[],n,is)
{
    for ldm := log2(n) to 1 step -1
    {
        m  := 2^ldm
        mh := m/2

        for j := 0 to mh-1 step 1
        {
            e := exp(is*2*pi*i*j/m)

            for r := 0 to n-1 step m
            {
                u := a[r+j]
                v := a[r+j+mh]

                a[r+j] :=    (u + v)
                a[r+j+mh] := (u - v) * e
            }
        }
    }

    scramble(a[],n)
}
```

note that the `scramble()`-procedure is called after the main loop, in the DIT code it was called before the main loop.

[5] adapted from [12]

1.4 Higher radix DIT and DIF algorithms

Operations can be saved if two or more steps of the radix 2 algorithms are combined.

For convenient formulation the following notation is useful:

Let $a^{(r\%m)}$ denote the subsequence of those elements of a that have subscripts x for which $x \equiv r(mod\ m)$;
e.g. $a^{(0\%2)}$ is $a^{(even)}$, $a^{(3\%4)} = \{a_3, a_7, a_{11}, a_{15}, ...\}$.
Let $a^{(r/m)}$ denote the subsequence of those elements of a that have indices $\frac{r\,n}{m}...\frac{(r+1)\,n}{m} - 1$;
e.g. $a^{(1/2)}$ is $a^{(right)}$, $a^{(2/3)}$ is the last third of a.
Let $\mathcal{S}^r a$ denote the sequence with elements $a_x\, e^{-r\,\pi\,i\,x/n}$ (there is no factor 2 in the exponential) where n is the length of the sequence a:
e.g. $\mathcal{S}^1 = \mathcal{S}$, $\mathcal{S}^0$ is the identity operator.
Let $\{a, b, c, ..., z\}$ denote the sequence that is the concatenation of the sequences[6] $a, b, c, ..., z$.

1.4.1 Decimation in time

First reformulate the radix 2 DIT step (formulas 1.5 and 1.6) in the new notation:

$$\mathcal{F}_n^{(0/2)}[a] = \mathcal{S}^0\mathcal{F}_{n/2}\left[a^{(0\%2)}\right] + \mathcal{S}^1\mathcal{F}_{n/2}\left[a^{(1\%2)}\right] \tag{1.10}$$

$$\mathcal{F}_n^{(1/2)}[a] = \mathcal{S}^0\mathcal{F}_{n/2}\left[a^{(0\%2)}\right] - \mathcal{S}^1\mathcal{F}_{n/2}\left[a^{(1\%2)}\right] \tag{1.11}$$

(Note that $\mathcal{S}^0$ is the identity operator).

The radix 4 step is[7]

$$\begin{aligned}\mathcal{F}_n^{(0/4)}[a] &= +\mathcal{S}^0\mathcal{F}_{n/4}\left[a^{(0\%4)}\right] + \mathcal{S}^1\mathcal{F}_{n/4}\left[a^{(1\%4)}\right] \\ &\quad +\mathcal{S}^2\mathcal{F}_{n/4}\left[a^{(2\%4)}\right] + \mathcal{S}^3\mathcal{F}_{n/4}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.12}$$

$$\begin{aligned}\mathcal{F}_n^{(1/4)}[a] &= +\mathcal{S}^0\mathcal{F}_{n/4}\left[a^{(0\%4)}\right] - i\mathcal{S}^1\mathcal{F}_{n/4}\left[a^{(1\%4)}\right] \\ &\quad -\mathcal{S}^2\mathcal{F}_{n/4}\left[a^{(2\%4)}\right] + i\mathcal{S}^3\mathcal{F}_{n/4}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.13}$$

$$\begin{aligned}\mathcal{F}_n^{(2/4)}[a] &= +\mathcal{S}^0\mathcal{F}_{n/4}\left[a^{(0\%4)}\right] - \mathcal{S}^1\mathcal{F}_{n/4}\left[a^{(1\%4)}\right] \\ &\quad +\mathcal{S}^2\mathcal{F}_{n/4}\left[a^{(2\%4)}\right] - \mathcal{S}^3\mathcal{F}_{n/4}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.14}$$

$$\begin{aligned}\mathcal{F}_n^{(3/4)}[a] &= +\mathcal{S}^0\mathcal{F}_{n/4}\left[a^{(0\%4)}\right] + i\mathcal{S}^1\mathcal{F}_{n/4}\left[a^{(1\%4)}\right] \\ &\quad -\mathcal{S}^2\mathcal{F}_{n/4}\left[a^{(2\%4)}\right] - i\mathcal{S}^3\mathcal{F}_{n/4}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.15}$$

or, more compact

$$\begin{aligned}\mathcal{F}_n^{(j/4)}[a] &= +e^{-2\,i\,\pi\,0\,j/4}\cdot\mathcal{S}^0\mathcal{F}_{n/4}\left[a^{(0\%4)}\right] + e^{-2\,i\,\pi\,1\,j/4}\cdot\mathcal{S}^1\mathcal{F}_{n/4}\left[a^{(1\%4)}\right] \\ &\quad +e^{-2\,i\,\pi\,2\,j/4}\cdot\mathcal{S}^2\mathcal{F}_{n/4}\left[a^{(2\%4)}\right] + e^{-2\,i\,\pi\,3\,j/4}\cdot\mathcal{S}^3\mathcal{F}_{n/4}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.16}$$

where $j = 0, 1, 2, 3, 4$.

Still more compact:

$$\mathcal{F}_n^{(j/4)}[a] = \sum_{k=0}^{3} e^{-2\,i\,\pi\,k\,j/4}\cdot\mathcal{S}^k\mathcal{F}_{n/r}\left[a^{(k\%4)}\right] \tag{1.17}$$

[6] which in this document are always of same length
[7] for the case of negative sign in the exponential function

where the summation symbol denotes *elementwise* summation of the sequences. (The dot indicates multiplication of every element of the rhs. sequence by the lhs. exponential).

The general radix r DIT step is

IDEA *general decimation in time step*: general DIT step

$$\mathcal{F}_n^{(j/r)}[a] = \sum_{k=0}^{r-1} e^{-2\,i\,\pi\,k\,j/r} \cdot \mathcal{S}^k \mathcal{F}_{n/r}\left[a^{(k\%r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.18}$$

Of course must r divide n.

1.4.2 Decimation in frequency

The radix 2 DIF step (formulas 1.5 and 1.6) was

$$\mathcal{F}_n^{(0\%2)}[a] = \mathcal{F}_{n/2}\left[\mathcal{S}^0\left(a^{(0/2)} + a^{(1/2)}\right)\right] \tag{1.19}$$

$$\mathcal{F}_n^{(1\%2)}[a] = \mathcal{F}_{n/2}\left[\mathcal{S}^1\left(a^{(0/2)} - a^{(1/2)}\right)\right] \tag{1.20}$$

The radix 4 DIF step is

$$\mathcal{F}_n^{(0\%4)}[a] = \mathcal{F}_{n/4}\left[\mathcal{S}^0\left(a^{(0/4)} + a^{(1/4)} + a^{(2/4)} + a^{(3/4)}\right)\right] \tag{1.21}$$

$$\mathcal{F}_n^{(1\%4)}[a] = \mathcal{F}_{n/4}\left[\mathcal{S}^1\left(a^{(0/4)} - i\,a^{(1/4)} - a^{(2/4)} + i\,a^{(3/4)}\right)\right] \tag{1.22}$$

$$\mathcal{F}_n^{(2\%4)}[a] = \mathcal{F}_{n/4}\left[\mathcal{S}^2\left(a^{(0/4)} - a^{(1/4)} + a^{(2/4)} - a^{(3/4)}\right)\right] \tag{1.23}$$

$$\mathcal{F}_n^{(3\%4)}[a] = \mathcal{F}_{n/4}\left[\mathcal{S}^3\left(a^{(0/4)} + i\,a^{(1/4)} - a^{(2/4)} - i\,a^{(3/4)}\right)\right] \tag{1.24}$$

or, more compact

$$\mathcal{F}_n^{(j\%4)}[a] = \mathcal{F}_{n/4}\left[\mathcal{S}^j \sum_{k=0}^{3} e^{-2\,i\,\pi\,k\,j/4} \cdot a^{(k/4)}\right] \tag{1.25}$$

where $j = 0, 1, 2, 3, 4$.

The general radix r DIF step is

IDEA *general decimation in frequency step*: general DIF step

$$\mathcal{F}_n^{(j\%r)}[a] = \mathcal{F}_{n/r}\left[\mathcal{S}^j \sum_{k=0}^{r-1} e^{-2\,i\,\pi\,k\,j/r} \cdot a^{(k/r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.26}$$

1.4.3 Implementation of radix $r = p^x$ DIF/DIT FFTs

If $r = p \neq 2$ (p prime) then the `scramble()` function has to be replaced by its radix-p version: the reordering now swaps elements x with $\tilde{x}$ where $\tilde{x}$ is obtained from x by reversing its radix-p expansion.

Pseudocode for a radix `px:=`p^x DIT FFT:

```
procedure fftdit_p(a[],n,is)
{
    radix_p_scramble(a[],n)
```

```
    for ldm := 1 to log_px(n) step x
    {
        m  := px^ldm
        mh := m/px

        for j := 0 to mh-1 step 1
        {
            e := exp(-2*pi*i*j/m)

            for r := 0 to n-1 step m
            {
                for z := 0 to px-1 step 1
                {
                    u[z] := a[r+j+mh*z]*e^z
                }

                radix_p_scramble(u[],px)

                px_point_fft(u[],is)

                for z := 0 to px-1 step 1
                {
                    a[z] := u[z]
                }
            }
        }
    }
}
```

Of course the loops that use the variable `z` have to be written out, the (length-p^x) scratch space `u[]` has to be replaced by explicite variables (e.g. `u0, u1, ...`) and the `px_point_fft(u[],is)` shall be a written out p^x-point FFT.

If $r = p^x$ than there is a pitfall one must now: if one uses the `radix_p_scramble()` procedure[8] before (DIT) or after (DIF) the main loop then some additional reordering is necessary in the innermost loop: in the above pseudo code this is indicated by the `radix_p_scramble(u[],p)` just before the `p_point_fft(u[],is)` line. Again, one won't use a call to a procedure call, but change indices in the loops where the `a[z]` are read/written for the DIT/DIF respectively.

It is wise to extract the stage of the main loop where the exp()-function always has the value 1, which is the case when `ldm==1` in the outermost loop[9]. In order not to restrict the possible array sizes to powers of p^x but only to powers of p one will supply adapted versions of the `ldm==1` -loop: e.g. for a radix-4 DIF FFT append a radix 2 step after the main loop if the array size is not a power of 4.

The following pseudo code does a radix 4 DIF FFT on the array `a[]`, the data length `n` must be a power of 2, `is` must be +1 or -1:

```
procedure fftdif4(a[],n,is)
{
    for ldm := log2(n) to 2 step -2
    {
        m  := 2^ldm
        mh := m/4
```

[8] as opposed to a radix p^x scramble procedure

[9] cf. section 4.3.

```
        for j := 0 to mh-1 step 1
        {
            e := exp(is*2*pi*i*j/m)

            for r := 0 to n-1 step m
            {
                u0 := a[r+j]
                u1 := a[r+j+mh]
                u2 := a[r+j+mh*2]
                u3 := a[r+j+mh*3]

                t0 := (u0+u2) + (u1+u3)
                t1 := (u0+u2) - (u1+u3)
                t2 := (u0-u2) + (u1+u3)*sqrt(-1)*is
                t3 := (u0-u2) - (u1+u3)*sqrt(-1)*is

                t1 := t1*e
                t2 := t2*e^2
                t3 := t3*e^3

                a[r+j]      := u0
                a[r+j+mh]   := u2
                a[r+j+mh*2] := u1
                a[r+j+mh*3] := u3
            }
        }
    }

    if is_odd(ldm) then
    {
        for r:=0 to n-1 step 2
        {
            t      := a[r]+a[r+1];
            a[r+1] := a[r]-a[r+1];
            a[r]   := t
        }
    }

    scramble(a[],n)
}
```

Note the 'swapped' order in which `u1, u2` are copied back in the innermost loop, this is what `radix_p_scramble(u[],p)` was supposed to do. If `n` is not a power of 2, then `ldm` is odd during the procedure and the at the last stage of main loop one has `ldm`=3.

To improve the performance one will instead of the (extracted) radix 2 loop supply extracted radix 8 and radix 4 loops. Then depending on whether `n` is a power of 4 or not one will use the radix 4 or the radix 8 loop, respectively. The start of the main loop then has to be `for ldm := log2(n) to 3 step -X` and on exit of the main loop one has `ldm`=3 or `ldm`=2.

1.5 The four step algorithm (FSA)

The four step FFT algorithm[10] works for data lengths $n = n_1\, n_2$. Consider the input array as a $n_1 \times n_2$-matrix (n_1 rows, n_2 columns).

four step algorithm

IDEA *the four step algorithm (FSA)*:

1. Apply a (length n_1) FFT on each column.
2. Multiply each matrix element (index r, c) by $\exp(\pm 2\,\pi\, i\, r\, c/n)$.
3. Apply a (length n_2) FFT on each row.
4. Transpose the matrix.

note the elegance !

It is trivial to rewrite the FSA as the

IDEA *transposed four step algorithm (TFSA)*:

1. Transpose the matrix.
2. Apply a (length n_2) FFT on each row.
3. Multiply each matrix element (index r, c) by $\exp(\pm 2\,\pi\, i\, r\, c/n)$.
4. Apply a (length n_1) FFT on each column.

FFT algorithms are usually very memory nonlocal, i.e. the data is accessed very randomly (as opposed to e.g. in strides). This algorithm has a much better memory locality[11], because the work is done in the short FFTs over the rows and columns. For the computation of the column FFTs it is important to know that it is[12] a very bad idea to do them in place, i.e. with memory locations that are equidistant with an offset that is a power of 2. One would get 100% cache misses and therefore get a memory performance that corresponds to the access time of the main memory[13]. The easy way around is to use an additional scratch space for the column FFTs, then only the copying form and to the scratch space will be slow. If one interleaves the copying back with the exp()-multiplications (to let the CPU do some work during the wait for the memory access) the performance should be ok.

It is usually a good idea to use factors of the data length n that are close to $\sqrt{n}$. Of course one can apply the same algorithm for the row (or column) FFTs again: it might be a good idea to split n into 3 factors (as close to $n^{1/3}$ as possible) if a length-$n^{1/3}$ FFT fits completely into the second level cache (or even the first level cache) of the computer used. Especially for systems with very high CPU clock the performance may increase drastically.

The four step algorithm is also an ideal candidate for (adaption for) mass storage FFTs, i.e. FFTs for data sets that do not fit into physical RAM[14].

1.6 Mass storage convolution using FSA

In convolution computations it is straightforward to save the transpositions. (The data is assumed to be in memory as row_0, row_1, ..., row_{n_1-1}, i.e. the way array data is stored in memory in the C language, as opposed to the Fortran language.) For simplicity auto convolution is shown here:

[10] The name 'four step FFT' (FSA) is borrowed from the paper [7].
[11] cf. [7]
[12] at least on computer systems that use direct mapped cache memory
[13] which is very long compared to the clock of modern CPUs
[14] The naive idea to simply try such an FFT with the virtual memory mechanism will of course, due to the nonlocality of FFTs, end in eternal harddisk activity

1. Apply a (length n_1) FFT on each column.
   (*memory access with n_2-skips*)
2. Multiply each matrix element (index r, c) by $\exp(\pm 2\,\pi\,i\,r\,c/n)$.
3. Apply a (length n_2) FFT on each row.
   (*memory access without skips*)
4. Complex square row.
5. Apply a (length n_2) FFT on each row (of the transposed matrix).
   (*memory access is without skips*)
6. Multiply each matrix element (index r, c) by $\exp(\mp 2\,\pi\,i\,r\,c/n)$.
7. Apply a (length n_1) FFT on each column (of the transposed matrix).
   (*memory access with n_2-skips*)

note that steps 3,4,5 constitute a length-n_2 convolution.

A simple consideration lets one use the above algorithm for mass storage convolutions, i.e. convolutions of data sets that do not fit into the main memory. Simply using the above with the data in a file and reading the data to memory for the required operations wouldn't be a good idea at all because the excessive number of seek operations on the file would degrade performance unacceptably.

Solution is simple: use as few rows as possible, i.e. choose the length of the rows so that they just fit into the main memory[15].

Let us count the number of seek operations to see why: (assume the data is x times bigger than the main memory, i.e. there are x rows)
Step 1 will be done in x passes, each pass accesses the file at x positions, therefore we have $2\,x^2$ seeks here (read and write).
Steps 3,4,5 consist of one pass for each row, $2\,x$ seeks.
Step 7: same as step 1, $2\,x^2$ seeks.
So one has a total of $4\,x^2 + 2\,x$ seek operations. If one seek takes 10 milliseconds this means for $x = 16$ (probably quite a big FFT) a total of approximately 10 seconds. With a main memory of 64 Megabytes[16] the CPU time alone might be in the order of several minutes. The overhead for the (linear) read and write would be (throughput of 10MB/sec assumed) $6\,S/10$ seconds or approximately 10 minutes.

With a multithreading OS one might produce a 'double buffer' variant: choose the row length so that it fits twice into RAM; then let always one (CPU-intensive) thread do the FFTs in one of the scratch spaces and another (hard disk intensive) thread write back the data from the other scratch-space and read the next data to be processed. With not too small main memory (and not too slow hard disk) and some fine tuning this should allow to completely do away with hard disk overhead.

[15] more accurate: the biggest possible amount of RAM where no swapping will occur.
[16] allowing for 8 million 8 byte floats, so the total FFT size is $S = 16 \cdot 64 = 1024$MB or 32million floats

Chapter 2

The Hartley transform (HT)

2.1 Definition of the HT

The Hartley transform (HT) is defined like the Fourier transform with 'cos + sin' instead of 'cos $+i \cdot$ sin'. Let $A = \mathcal{H}[a]$ denote the hartley transform of a then

$$A_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left(\cos \frac{2\pi k x}{n} + \sin \frac{2\pi k x}{n} \right) \tag{2.1}$$

It has the advantage that real input produces real output. It also is its own inverse.

notation:
For a sequence a of length n let $\bar{a}$ denote the reversed sequence: $\bar{a}_0 := a_0, \bar{a}_{n/2} := a_{n/2}, \bar{a}_k := a_{n-k}$

The convolution property of the HT is

$$\mathcal{H}[a \otimes b]_k = \frac{1}{2} \left(A_k B_k - \bar{A}_k \bar{B}_k + A_k \bar{B}_k + \bar{A}_k B_k \right) \tag{2.2}$$

(the symbol $\otimes$ denotes convolution).

The connection to Fourier transforms is expressed by

$$\mathcal{H}[a] = \Re\mathcal{F}[a] - \Im\mathcal{F}[a] \tag{2.3}$$

2.2 The fast Hartley transform

notation:
Let $\mathcal{X}a$ denote the sequence with elements $a_x \cos \pi x/n + \bar{a}_x \sin \pi x/n$ where n is the length of the sequence a (this is the 'shift operator' for the Hartley transform).

IDEA *Hartley radix 2 decimation in time step*:

$$\mathcal{H}_n^{(left)}[a] = \mathcal{H}_{n/2}\left[a^{(even)}\right] + \mathcal{X}\mathcal{H}_{n/2}\left[a^{(odd)}\right] \tag{2.4}$$

$$\mathcal{H}_n^{(right)}[a] = \mathcal{H}_{n/2}\left[a^{(even)}\right] - \mathcal{X}\mathcal{H}_{n/2}\left[a^{(odd)}\right] \tag{2.5}$$

IDEA *Hartley radix 2 decimation in frequency step*:

$$\mathcal{H}_n^{(even)}[a] = \mathcal{H}_{n/2}\left[a^{(left)} + a^{(right)}\right] \tag{2.6}$$

$$\mathcal{H}_n^{(odd)}[a] = \mathcal{H}_{n/2}\left[\mathcal{X}\left(a^{(left)} - a^{(right)}\right)\right] \tag{2.7}$$

2.3 Real valued FT by HT

The Fourier transform of a purely real sequence has a symmetric real part ($\Re\bar{c} = \Re c$) and an antisymmetric imaginary part ($\Im\bar{c} = -\Im c$). Simply using a complex FFT for real input is basically a waste of a factor 2 of memory and CPU cycles. There are several ways out:

- sincos wrappers for complex FFTs, for a real sequence a one feeds the (half length) complex sequence $f = a^{(even)} + i\, a^{(odd)}$ into a complex FFT. Some postprocessing is necessary, cf. [6]
- special real (split radix algorithm) FFTs
- Usage of the fast Hartley transform, only this technique shall be described here.

Use the 'inverse' of 2.3:

$$\Re\mathcal{F}[a] = \frac{1}{2}\left(\mathcal{H}[a] + \overline{\mathcal{H}[a]}\right) \tag{2.8}$$

$$\Im\mathcal{F}[a] = \frac{1}{2}\left(\mathcal{H}[a] - \overline{\mathcal{H}[a]}\right) \tag{2.9}$$

Pseude code for this:

```
procedure real_complex_fft_by_fht(a[],n)
{
    fht(a[],n)

    j := n-1
    for i:=1 to n/2-1 step 1
    {
        u := a[i]
        v := a[j]
        a[i] := 0.5*(u+v)
        a[j] := 0.5*(u-v)
        j := j-1
    }
}
```

At the end of this procedure the ordering of the data is like this:

$$a[0] = \Re c_0, \quad a[1] = \Re c_1, \quad \ldots \quad a[n/2] = \Re c_{n/2}, \tag{2.10}$$

$$a[n/2+1] = -\Im c_{n/2-1}, \quad a[n/2+2] = -\Im c_{n/2-2}, \quad \ldots \quad a[n-1] = -\Im c_1, \tag{2.11}$$

The inverse procedure is:

```
procedure complex_real_fft_by_fht(a[],n)
{
    j := n-1
    for i:=1 to n/2-1 step 1
    {
        u := a[i]
        v := a[j]
        a[i] := u+v
        a[j] := u-v
        j := j-1
```

```
    }

    fht(a[],n)
}
```

Chapter 3

The z-transform (ZT)

3.1 Definition of the ZT

The z-transform (ZT) $\mathcal{Z}[a] = \hat{a}$ of a (length n) sequence a with elements a_x is defined as

$$\hat{a}_k \quad := \quad \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{3.1}$$

The z-transform is a linear transformation, its most important property is the convolution property (formula B): convolution in original space corresponds to ordinary (elementwise) multiplication in z-space. (See [10] and [11]).

Note that the special case $z = e^{-2\,\pi\,i/n}$ is the discrete FT.

3.2 The chirp ZT

In the definition of the (discrete) z-transform we rewrite[1] the product $x\,k$ as

$$x\,k \quad = \quad -(k-x)^2/2 + x^2/2 + k^2/2 \tag{3.2}$$

$$\hat{f}_k = \sum_{x=0}^{n-1} f_x \, z^{x\,k} \quad = \quad z^{k^2/2} \sum_{x=0}^{n-1} \left(f_x \, z^{x^2/2} \right) \, z^{-(k-x)^2/2} \tag{3.3}$$

this leads to the following

IDEA *chirp z-transform*:

1. multiply f elementwise with $z^{x^2/2}$,
2. convolve the resulting sequence with the sequence $z^{-x^2/2}$
3. multiply elementwise with the sequence $z^{k^2/2}$.

The above algorithm constitutes a 'fast' ($\sim n \log(n)$) algorithm for the ZT because fast convolution is possible via FFT.

[1] cf. [1]

3.3 Arbitrary length FFT by ZT

We first note that the length n of the input sequence a for the fast z-transform is not limited to highly composite values (especially n prime is allowed): For values of n where a FFT is not feasible pad the sequence with zeros up to a length L with $L >= 2\,n$ and a length L FFT feasible (e.g. L is a power of 2).

Second remember that the FT is the special case $z = e^{-2\,\pi\,i/n}$ of the ZT: with the chirp ZT algorithm one also has an (arbitrary length) FFT algorithm

The transform takes a few times more than an optimal transform (by direct FFT) would take. The worst case (if only FFTs for n a power of 2 are available) is $n = 2^p + 1$: one must perform 3 FFTs of length $2^{p+2} \approx 4\,n$ for the computation of the convolution. So the total work amounts to about 12 times the work a FFT of length $n = 2^p$ would cost. It is of course possible to lower this 'worst case factor' to 6 by using highly composite L slightly greater than $2\,n$.

3.4 Fractional Fourier transform by ZT

The z-transform with $z = e^{\alpha\,2\,\pi\,i/n}$ and $\alpha \neq 1$ is called the fractional Fourier transform (FRFT). Uses of the FRFT are e.g. the computation of the DFT for data sets that have only few nonzero elements and the detection of frequencies that are no integer multiples of the lowest frequency of the DFT. A thorough discussion can be found in [8].

Chapter 4

Numbertheoretic transforms (NTTs)

How to make a numbertheoretic transform out of your FFT:
'*replace* $exp(-2\,\pi\,i/n)$ *by a primitive* n*-th root of unity, done.*'

We want to do FFTs in $\mathbb{Z}/p\mathbb{Z}$ (the ring of integers modulo some integer p) instead of $\mathbb{C}$, the complex numbers. These FFTs are called numbertheoretic transforms (NTTs), mod p FFTs or prime modulus transforms.

The operations needed in FFTs are addition, subtraction and multiplication. Division is not needed, except for division by n for the final normalization after transform and backtransform. Division by n is multiplication by the inverse of n. Hence n must be invertible in $\mathbb{Z}/p\mathbb{Z}$: p must be coprime to n.

Condition 1: $p \perp n$

There is another restriction for the choice of p: for a length n FFT we need a primitive n-th root of unity. A number r is called an n-th root of unity if $r^n = 1$, it is called a primitive root of unity if $r^k \neq 1 \; \forall \, k < n$.

Condition 2: $\exists\,\text{primit.}\ \sqrt[n]{1}$

In $\mathbb{C}$ matters are simple: $e^{2\,\pi\,i/n}$ is a primitive n-th root of unity for arbitrary n. $e^{2\,\pi\,i/21}$ is a 21-th root of unity. $r = e^{2\,\pi\,i/3}$ is also 21-th root of unity but not a primitive root, because $r^3 = 1$. A primitive n-th root of 1 in $\mathbb{Z}/p\mathbb{Z}$ is also called an 'element of order n'. The 'cyclic' property of primitive roots r lies in the heart of all FFT algorithms: $r^{n+k} = r^k$.

In $\mathbb{Z}/p\mathbb{Z}$ things are not that simple, primitive roots of unity do not exist for arbitrary n. Primitive roots exist for some maximal order m. Roots of unity of an order different from m are available only for the divisors d_i of m: r^{m/d_i} is a d_i-th root of unity because

$$\left(r^{m/d_i}\right)^{d_i} \quad = \quad r^m = 1 \tag{4.1}$$

In what follows we will need the function $\varphi()$, the so called 'totient' function. $\varphi(p)$ counts the number of integers prime to and less than p. For p prime $\varphi(p) = p - 1$. For p composite $\varphi(p) < p - 1$. For $p = p_x^k$ a prime power $\varphi(p_x^k) = p_x^k - p_x^{k-1}$, e.g. $\varphi(2^k) = 2^{k-1}$. For coprime p_1, p_2 (p_1, p_2 not necessarily primes) $\varphi(p_1\,p_2) = \varphi(p_1)\,\varphi(p_2)$, i.e. $\varphi()$ is a multiplicative function.

Cf. [2], [15] or [1] and books on number theory.

4.1 Prime p: $\mathbb{F}_p$

If p is prime then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p$, i.e. all elements except 0 have inverses. The elements of maximal order ($m = \varphi(p) = p - 1$) are then called 'primitive elements' or 'generators'. If r is a generator, then every element in $\mathbb{F}_p$ different from 0 is equal to some power r^x, $1 \leq x < p$ of r. Roots of unity of an order different from $p-1$ are available only for the divisors d_i of $p-1$. Therefore the second condition on p for a length-n mod p FFT being possible is that n divides $p - 1$. This restricts the choice p to primes

of the form $p = v\,n + 1$: For length $n = 2^k$ FFTs one will use primes like $p = 3 \cdot 5 \cdot 2^{27} + 1$ (31 bits), $p = 13 \cdot 2^{28} + 1$ (32 bits), $p = 3 \cdot 29 \cdot 2^{56} + 1$ (63 bits) or $p = 27 \cdot 2^{59} + 1$ (64 bits)[1]. The first condition is trivially fulfilled: a prime p is coprime to all integers $n < p$.

To test whether r is a primitive n-th root of unity (also simply called a 'primitive root') in $\mathbb{F}_p$ one doesn't need to check for all $k < n$ that $r^k \neq 1$: it suffices to do the check for exponents k that are prime factors of n. To find a primitive root in $\mathbb{F}_p$ proceed as indicated by the following pseudo code:

```
procedure find_primroot(p)
{
    f[] := distinct_prime_factors(p-1)

    for r:=2 to p-1 step 1
    {
        x := TRUE

        foreach q in f[]
        {
            if r^((p-1)/q)==1 then x:=FALSE
        }

        if x==TRUE return r
    }

    error("no primitive root found")
}
```

4.2 Composite p

4.2.1 p a product of pairwise destinct primes

For composite $p = \prod_{i=1}^{f} p_i$ (with $p_i \neq p_j \; \forall i \neq j$) $\mathbb{Z}/p\mathbb{Z}$ is isomorphic to the direct product of the $\mathbb{F}_{p_i}$:

$$\mathbb{Z}/p\mathbb{Z} \quad \cong \quad \mathbb{F}_{p_1} \otimes \mathbb{F}_{p_2} \otimes ... \otimes \mathbb{F}_{p_f} \tag{4.2}$$

There exist elements of maximal order $m = lcm(p_1 - 1, p_2 - 1, ..., p_f - 1)$ Therefore the second condition on p for a length-n mod p FFT being possible is that n divides m. Such an element can be found as follows:

1. find primitive elements r_i for the fields $\mathbb{F}_{p_i}$
2. Find the solution of the CRT[2] problem $r \equiv r_i \mod p_i \quad i = 1, ..., f$

Example: $p = 2^{32} + 1 = 4294967297 = 641 \cdot 6700417$
$\varphi(641) = 640 = 2^7 \cdot 5$
$\varphi(6700417) = 6700416 = 2^7 \cdot 3 \cdot 17449$
$r_{641} = 3$ is a primitive root in the field $\mathbb{F}_{641}$
$r_{6700417} = 5$ is a primitive root in the field $\mathbb{F}_{6700417}$
4288266240 is an element of (the maximal)
order $lcm(640, 6700416) = 33502080 = 2^7 \cdot 3 \cdot 5 \cdot 17449$ in the ring $\mathbb{Z}/4294967297\mathbb{Z}$.
Hence the maximum (radix 2) NTT length would be $n = 2^7$.

[1] Primes of that form are not 'exceptional', cf. Lipson [2]
[2] cf. appendix E

In practice (e.g. for p a product of f primes that are only slightly smaller than a machine word) it is unwise to do the computation as indicated by the above. Instead it will be much more efficient to

1. for a given number x find $u_i,\ i = 1, ..., f$ so that $x \equiv u_i \pmod{p_i}$.)
2. do the transforms separately using each p_i as a modulus
3. get the final result by applying the CRT elementwise to the transformed vectors

4.2.2 p a prime power

For a prime power $p = p_x^k$ the maximal possible order of an element is $m = \varphi(p_x^k) = p_x^k - p_x^{k-1}$. Cryptographers like $\mathbb{Z}/2^k\mathbb{Z}$ where $m = 2^{k-1}$.

4.2.3 The general case

For general $p = \prod_{i=1}^{f} p_i^{k_i}$ the maximal possible order of an element is $m = lcm(\varphi(p_1^{k_1}), \varphi(p_2^{k_2}), ..., \varphi(p_f^{k_f}),)$ Any use for this ?

4.3 Pseudocode for NTTs

To implement mod p FFTs one basically must supply a mod p class[3] and replace $e^{2\pi i/n}$ by an n-th root of unity in $\mathbb{Z}/p\mathbb{Z}$ in the code.

For the backtransform one uses the (mod p) inverse $\bar{r}$ of the r (an element of order n) that was used for the (forward-) transform. There are two ways to compute the inverse modulo p: computing $\bar{r} = r^m \pmod{p}$ where $m = \varphi(p) - 1$ or using the extended Euclidean algorithm (cf. [15]). $\bar{r}$ exists because $r \perp p$; $\bar{r}$ is a root of unity because $\bar{r}^m = r^{(m-1)\,m} = 1$; $\bar{r}$ is a primitive root because $\bar{r}^{m-k} = r^{m\,m-m-m\,k-k} = r^{-k} \neq 1\ \forall k < m$

The notion of the Fourier transform as a 'decomposition into frequencies' is of course meaningless for NTTs.

The nice feature of NTTs is that there is no loss of precision in the transform (as there is always with the complex FFTs). The analogue of trig recursion (in its most naive form) is mandatory, as the computation of roots of unity is expensive.

4.3.1 Radix 2 DIT NTT

pseudo code for the radix 2 decimation in time mod fft: (to be called with `ldn=log2(n)`)

```
procedure mod_fft_dit2((mod_type)f[], ldn, is)
{
    n := 2^ldn

    rn := element_of_order(n)

    if is>0 then  rn := inverse(rn)

    scramble(f[],n)

    for ldm:=1 to ldn step 1
```

[3] A class in the C++ meaning: objects that represent numbers in $\mathbb{Z}/p\mathbb{Z}$ together with the operations on them

```
    {
        m  := 2^ldm
        mh := m/2

        (mod_type) dw := rn^(2^(ldn-ldm))
        (mod_type) w  := 1

        for j:=0 to mh-1 step 1
        {
            for r:=0 to n-1 step m
            {
                t1 := r+j
                t2 := t1+mh

                (mod_type) v := f[t2]*w
                (mod_type) u := f[t1]

                f[t1] := u+v
                f[t2] := u-v
            }

            w := w*dw
        }
    }
}
```

it is a good idea to extract the `ldm==1` stage of the outermost loop: replace

```
    for ldm:=1 to ldn step 1
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        t      := f[r]+f[r+1];
        f[r+1] := f[r]-f[r+1];
        f[r]   := t
    }

    for ldm:=2 to ldn step 1
    {
```

4.3.2 Radix 2 DIF NTT

pseudo code for the radix 2 decimation in frequency mod fft:

```
procedure mod_fft_dif2((mod_type) f[], ldn, is)
{
    n := 2^ldn

    dw := element_of_order(n)
```

```
    if is>0 then  dw := inverse(rn)

    for  ldm:=ldn to 1 step -1
    {
        m  := 2^ldm
        mh := m/2

        (mod_type) w := 1

        for j:=0 to mh-1 step 1
        {
            for r:=0 to n-1 step m
            {
                t1 := r+j;
                t2 := t1+mh;

                (mod_type) v := f[t2];
                (mod_type) u := f[t1];

                f[t1] :=  u+v;
                f[t2] := (u-v)*w;
            }

            w := w*dw;
        }

        dw := dw*dw;
    }

    scramble(f[],n);
}
```

Again, extract the `ldm==1` stage of the outermost loop: replace

```
    for  ldm:=ldn to 1 step -1
```

by

```
    for  ldm:=ldn to 2 step -1
```

and insert

```
    for r:=0 to n-1 step 2
    {
        t      := f[r]+f[r+1];
        f[r+1] := f[r]-f[r+1];
        f[r]   := t
    }
```

before the call of `scramble(f[],n)`.

4.4 Convolution with NTTs

The NTTs are natural candidates for (exact) integer convolutions, as used e.g. in (high precision) multiplications. One must keep in mind that 'everything is mod p', the largest value that can be represented is $p-1$. As an example consider the multiplication of n-digit radix R numbers[4]. The largest possible value in the convolution is the 'central' one, it can be as large as $M = n\,(R-1)^2$ (which will occur if both numbers consist of 'nines' only[5]).

One has to choose $p > M$ to get rid of this problem. If p does not fit into a single machine word this may slow down the computation unacceptably. The way out is to choose p as the product of several distinct primes that are all just below machine word size and use the Chinese Remainder Theorem (CRT) afterwards.

If using length-N FFTs for convolution there must be an inverse element for N. This imposes the condition $gcd(N, modulus) = 1$, i.e. the modulus must be prime to N. Usually[6] $modulus$ must be an odd number.

[4]Multiplication is a convolution of the digits followed by the 'carry' operations.
[5]A radix R 'nine' is $R-1$.
[6]for length-2^k FFTs

Appendix A

Continuous Fourier transforms

The (continuous) *Fourier transform* (FT) of a function $f : \mathbb{R}^n \to \mathbb{R}^n, \quad \vec{x} \mapsto f(\vec{x})$ is defined by

$$F(\vec{\omega}) \quad := \quad \frac{1}{\sqrt{2\,\pi}^n} \int_{\mathbb{R}^n} f(\vec{x})\, e^{-i\,\vec{x}\,\vec{\omega}} d^n x \tag{A.1}$$

The FT is is a unitary transform.
Its inverse ('backtransform') is

$$f(\vec{x}) \quad = \quad \frac{1}{\sqrt{2\,\pi}^n} \int_{\mathbb{R}^n} F(\vec{\omega})\, e^{+i\,\vec{x}\,\vec{\omega}} d^n \omega \tag{A.2}$$

i.e. the complex conjugate transform.
For the 1-dimensional case one has

$$F(\omega) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} f(x) e^{-i\,x\,\omega} dx \tag{A.3}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} F(\omega)\, e^{+i\,x\,\omega} d\omega \tag{A.4}$$

The 'frequency'-form is

$$\hat{f}(\nu) \quad = \quad \int_{-\infty}^{+\infty} f(x) e^{-2\,\pi\,i\,x\,\nu} dx \tag{A.5}$$

$$f(x) \quad = \quad \int_{-\infty}^{+\infty} \hat{f}(\nu)\, e^{+2\,\pi\,i\,x\,\nu} d\nu \tag{A.6}$$

The Poisson summation formula (cf. [13], p.45) is

$$\sum_{x \in \mathbb{Z}} f(x) \quad = \quad \sum_{\nu \in \mathbb{Z}} \hat{f}(\nu) \tag{A.7}$$

More general

$$\sum_{x \in \mathbb{Z}} f(x + r)\, e^{-\pi\,i\,s\,(2x+r)} \quad = \quad \sum_{\nu \in \mathbb{Z}} \hat{f}(\nu + s)\, e^{+\pi\,i\,r\,(2\nu+s)} \tag{A.8}$$

For periodic functions defined on a interval $L \in \mathbb{R}$, $f : L \to \mathbb{R}, \quad \vec{x} \mapsto f(\vec{x})$ one has the *semi-continuous Fourier transform*:

$$c_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, e^{-2\,\pi\,i\,k\,x/L} dx \tag{A.9}$$

Then

$$\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k \, e^{+2\,\pi\,i\,k\,x/L} \quad = \quad \begin{cases} f(x) & \text{if } f \text{ continuous at } x \\ \frac{f(x+0)+f(x-0)}{2} & \text{else} \end{cases} \tag{A.10}$$

Another form is given by

$$a_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x) \, \cos \frac{2\,\pi\,k\,x}{L} dx, \qquad k = 0, 1, 2, ... \tag{A.11}$$

$$b_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x) \, \sin \frac{2\,\pi\,k\,x}{L} dx, \qquad k = 1, 2, ... \tag{A.12}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{L}} \left[\frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \, \cos \frac{2\,\pi\,k\,x}{L} + b_k \, \sin \frac{2\,\pi\,k\,x}{L} \right) \right] \tag{A.13}$$

with

$$c_k \quad = \quad \begin{cases} \frac{a_0}{2} & (k = 0) \\ \frac{1}{2}(a_k - i b_k) & (k > 0) \\ \frac{1}{2}(a_k + i b_k) & (k < 0) \end{cases} \tag{A.14}$$

The *discrete Fourier transform* (DFT) of a sequence f of length n with elements f_x is defined by

$$c_k \quad := \quad \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x \, e^{-2\,\pi\,i\,x\,k/n} \tag{A.15}$$

Backtransform is

$$f_x \quad = \quad \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, e^{+2\,\pi\,i\,x\,k/n} \tag{A.16}$$

Cf. [1] and [11].

note that in the rest of this document the factors $\frac{1}{\sqrt{n}}$ in front of the sums are omitted. To make $\mathcal{F}^{-1}\left[\mathcal{F}\left[.\right]\right]$ identity one has to multiply by a factor $\frac{1}{n}$ after the backtransform.

Appendix B

The convolution property of ZT and FT

Recall the definition of the z-transform $\mathcal{Z}[a] = \hat{a}$ of a (length n) sequence a with elements a_x:

$$\hat{a}_k := \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{B.1}$$

The convolution (sequence) h (with elements h_τ) of the two sequences a and b is defined as

$$h = a \otimes b \tag{B.2}$$

$$h_\tau := \sum_x a_x \, b_{\tau - x} \tag{B.3}$$

The convolution property of the z-transform can be written as

$$\mathcal{Z}[a \otimes b] = \mathcal{Z}[a] \; \mathcal{Z}[b] \tag{B.4}$$

i.e. convolution in original space is ordinary (elementwise) multiplication in z-space.
Proofoid:

$$\hat{a}_k \, \hat{b}_k = \sum_x a_x \, z^{k\,x} \sum_y b_y \, z^{k\,y} \tag{B.5}$$

$$\text{with} \quad y := \tau - x$$

$$= \sum_x a_x \, z^{k\,x} \sum_{\tau - x} b_{\tau - x} \, z^{k\,(\tau - x)} \tag{B.6}$$

$$= \sum_x \sum_{\tau - x} a_x \, z^{k\,x} b_{\tau - x} \, z^{k\,(\tau - x)} \tag{B.7}$$

$$= \sum_\tau \left(\sum_x a_x \, b_{\tau - x} \right) z^{k\,\tau} \tag{B.8}$$

$$= \left(\mathcal{Z} \left[\sum_x a_x \, b_{\tau - x} \right] \right)_k \tag{B.9}$$

$$= (\mathcal{Z}[a \otimes a])_k \tag{B.10}$$

The Fourier transform is the z-transform for $z = e^{-i\,2\,\pi/n}$, the convolution property rewrites as

$$a \otimes b = \mathcal{F}^{-1}[\mathcal{F}[a] \; \mathcal{F}[a]] \tag{B.11}$$

Written out for finite sequences

$$\sum_{x=0}^{n-1} a_x\, b_{\tau-x} \quad = \quad \sum_{k=0}^{n-1} \left(\sum_{x=0}^{n-1} a_x\, e^{-2\,\pi\, i\, x\, k/n} \sum_{x=0}^{n-1} b_x\, e^{-2\,\pi\, i\, x\, k/n} \right) e^{+2\,\pi\, i\, k\, \tau/n} \tag{B.12}$$

negative indices $\tau - x$ must be understood as $n + \tau - x$, it's a cyclic convolution.

For the usual (linear) convolution one must use 'zero padding' of the data: to convolve two seqences a, b of lengths n, m one extends (by appending zeros) both series to length $\geq n + m - 1$. The resulting series are transformed, elementwise multiplied and transformed back to get the linear convolution.

Appendix C

Optimisation considerations

reduce operations: use higher radix

reduce multiplications: Winograd FFT

number of processor registers: (intel sucks)

memory locality, cache size: four step algorithm

trig recursion: loss of precision (not with mod FFTs), use stable versions,

trig table: (only for small lengths)

explicit last/first step with radix as high a possible

write special versions for zero padded data (e.g. for convolutions), also write a special version of scramble for zero padded data

integer stuff: consider mod FFT

dsp: ...

image processing & effects: check also Walsh transform etc.

super computers: unit stride data access, vectorising

for correlations/convolutions save two scramble (or transpose) operations by combining DIF and DIT algorithms.

ONLY general rule: better algorithms win !

Appendix D

Trigonometric recursion

In the computation of FFTs one typically needs the values $\{\exp(i\,\omega), \exp(i\,(\omega+\delta)), \exp(i\,(\omega+2\,\delta)), ...\}$ in sequence.

The naive idea to do this is to precompute $d = \exp(i\,\delta)$ and recursively compute the next following value using the identity $\exp(i\,(\omega + k\,\delta)) = d\,\exp(i\,(\omega + (k-1)\,\delta))$, This method, however, is of no practical value because the numerical error grows in the process.

Here is a stable version of a trigonometric recursion for the computation of the sequence:

Precompute

$$c = \cos(\omega), \tag{D.1}$$
$$s = \sin(\omega), \tag{D.2}$$
$$\alpha = 2\,(\sin(\delta/2))^2 \tag{D.3}$$
$$\beta = \sin(\delta) \tag{D.4}$$

Then compute the next power from the previous as:

$$c_{next} = c - (\alpha\,c + \beta\,s); \tag{D.5}$$
$$s_{next} = s - (\alpha\,s - \beta\,c); \tag{D.6}$$

Do not expect to get all the precision you would get with the repeated call of the sin and cos functions, but even for very long FFTs less than 3 bits of precision are lost.

When working with `doubles` it is a good idea to use the type `long double` with the trig recursion: the sin and cos will than be accurate within the full `double`-precision.

Appendix E

The Chinese Remainder Theorem (CRT)

The Chinese remainder theorem (CRT):

Let $p_1, p_2, ..., p_f$ be pairwise relatively[1] prime (i.e. $gcd(p_i, p_j) = 1,\ \forall i \neq j$) CRT fact
If $x \equiv u_i\ (mod\ p_i)\ i = 1, 2, ..., f$ then x is unique modulo the product $p_1 \cdot p_2 \cdot ... \cdot p_f$.

For only two moduli p_1, p_2 compute x as follows[2]:

pseudo code to find unique $x\ (mod\ p_1\, p_2)$ with $x \equiv u_1\ (mod\ p_1)\ x \equiv u_2\ (mod\ p_2)$:

```
function crt2(u1,p1,u2,p2)
{
    c := invmod(p1,p2)

    s := ((u2-u1)*c) mod p2

    x := u1+s*p1

    return x
}
```

here `invmod(a,b)` shall return $\bar{a}$, the inverse of a mod b. For repeated CRT calculations with the same moduli one will use precomputed `c`.

For more more than two moduli use the above (two moduli) algorithm repeatedly: CRT algorithm

```
function crt(u[],p[],f)
{
    u1 := u[0]
    p1 := p[0]

    i := 1

    do
    {
        u2 := u[i]
        p2 := p[i]
```

[1] note that it is not assumed that any of the p_i is prime
[2] cf. [2]

```
        u1 := crt2(u1,p1,u2,p2)
        p1 := p1*p2

        i := i+1
    }
    while i<f

    return u1
}
```

Bibliography

[1] H.J.Nussbaumer: Fast Fourier Transform and Convolution Algorithms, 2.ed, Springer 1982

[2] J.D.Lipson: Elements of algebra and algebraic computing, Addison-Wesley 1981

[3] C. van Loan: Computational Frameworks for the Fast Fourier Transform, SIAM Frontiers in Applied Mathematics, 1992

[4] L.P.Jaroslavskij: Einführung in die digitale Bildverarbeitung, german translation of the russian 'Vvedenie v cifrovuju obrabotku izobraženij', Hüthig Buch Verlag GmbH, 2.ed, Heidelberg 1990

[5] E.Oran Brigham: The Fast Fourier Transform, Prentice-Hall 1974

[6] W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery: Numerical Recipes in C, Cambridge University Press, 1988, 2nd Edition 1992
online: `http://nr.harvard.edu/nr/`, be careful with the code !

[7] D.H.Bailey: FFTs in External or Hierarchical Memory, 1989
online at `http://www.nas.nasa.gov/~dbailey/`

[8] D.H.Bailey: The Fractional Fourier Transform and Applications, 1990
online at `http://www.nas.nasa.gov/~dbailey/`

[9] M.Hegland: A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing
online at `XXX`

[10] R.L.Graham, D.E.Knuth, O.Patashnik: Concrete Mathematics, Addison-Wesley, New York 1988

[11] I.N.Bronstein, K.A.Semendjajew, G.Grosche, V.Ziegler, D.Ziegler, ed: E.Zeidler: Teubner-Taschenbuch der Mathematik, vol. 1+2, B.G.Teubner Stuttgart, Leipzig 1996, the new edition of Bronstein's Handbook of Mathematics, english edition in preparation.

[12] J.Stoer, R.Bulirsch: Introduction to Numerical Analysis, Springer-Verlag, New York, Heidelberg, Berlin 1980

[13] M.Waldschmidt, P.Moussa, J.-M. Luck, C.Itzykson (Eds.): From Number Theory to Physics, Springer Verlag 1992

[14] Mikko Tommila: apfloat, A High Performance Arbitrary Precision Arithmetic Package, 1996,
online at `http://www.hut.fi/~mtommila/apfloat/`

[15] H.Cohen: A Course in Computational Algebraic Number Theory, Springer Verlag, Berlin Heidelberg 1993,
online errata list at `http://XXX`