FFTs for programmers
algorithms and source code

preliminary draft version

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at February 19, 2001

[1] This document is online at http://www.jjj.de/fxt/. It will stay available online for free.

Contents

List of important Symbols

$\Re x$	real part of x
$\Im x$	imaginary part of x
x^*	complex conjugate of x
a	a sequence, e.g. $\{a_0, a_1, ..., a_{n-1}\}$, the index always starts with zero.
$\hat{a}$	transformed (e.g. Fourier transformed) sequence
$\overset{m}{=}$	emphasize that the sequences to the left and right are all of length m
$\mathcal{F}\left[a\right] \quad (= c)$	(discrete) Fourier transform (FT) of a, $c_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{x\,k}$ where $z = e^{\pm 2\,\pi\,i/n}$
$\mathcal{F}^{-1}\left[a\right]$	inverse (discrete) Fourier transform (IFT) of a, $\mathcal{F}^{-1}\left[a\right]_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{-x\,k}$
$\mathcal{S}^k a$	a sequence c with elements $c_x := a_x \, e^{\pm\,k\,2\,\pi\,i\,x/n}$
$\mathcal{H}\left[a\right]$	discrete Hartley transform (HT) of a
$\overline{a}$	sequence reversed around element with index $n/2$
a_S	the symmetric part of a sequence: $a_S := a + \overline{a}$
a_A	the antisymmetric part of a sequence: $a_A := a - \overline{a}$
$\mathcal{Z}\left[a\right]$	discrete z-transform (ZT) of a
$\mathcal{W}_v\left[a\right]$	discrete weighted transform of a, weight (sequence) v
$\mathcal{W}_v^{-1}\left[a\right]$	inverse discrete weighted transform of a, weight v
$a \circledast b$	cyclic (or circular) convolution of sequence a with sequence b
$a \circledast_{ac} b$	acyclic (or linear) convolution of sequence a with sequence b
$a \circledast_{-} b$	negacyclic (or skew circular) convolution of sequence a with sequence b
$a \circledast_{\{v\}} b$	weighted convolution of sequence a with sequence b, weight v
$n \backslash N$	n divides N

Chapter 1

The Fourier transform

1.1 The discrete Fourier transform

The *discrete Fourier transform* (DFT or simply FT) of a complex sequence a of length n is defined as

$$c = \mathcal{F}[a] \tag{1.1}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{+x\,k} \qquad \text{where} \quad z = e^{\pm 2\,\pi\, i/n} \tag{1.2}$$

z is an n-th root of unity: $z^n = 1$.

Backtransform (or *inverse discrete Fourier transform* IDFT or simply IFT) is then

$$a = \mathcal{F}^{-1}[c] \tag{1.3}$$

$$a_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, z^{-x\,k} \tag{1.4}$$

That this is really true is not straightforward. Consider element y of the IFT of the FT of a:

$$\mathcal{F}^{-1}\left[\mathcal{F}[a]\right]_y = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} (a_x \, z^{x\,k}) \, z^{-y\,k} \tag{1.5}$$

$$= \frac{1}{n} \sum_x a_x \sum_k (z^{x-y})^k \tag{1.6}$$

As $\sum_k (z^{x-y})^k = n$ for $x = y$ and zero else (because z is an n-th root of unity). Therefore the whole expression is equal to

$$\frac{1}{n} n \sum_x a_x \, \delta_{x,y} = a_y \tag{1.7}$$

where

$$\delta_{x,y} = \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases} \tag{1.8}$$

In this book the FT with the plus in the exponent is called forward transform, the one with the minus is called the backward transform, the choice is arbitrary[1].

[1] Electrical engineers prefer the minus for the forward transform, mathematicians the plus.

The FT is a linear transform, i.e. for $\alpha, \beta \in \mathbb{C}$

$$\mathcal{F}[\alpha\, a + \beta\, b] \quad = \quad \alpha\, \mathcal{F}[a] + \beta\, \mathcal{F}[b] \tag{1.9}$$

For the FT Parseval's equation holds, let $c = \mathcal{F}[a]$, then

$$\sum_{x=0}^{n-1} a_x^2 \quad = \quad \sum_{k=0}^{n-1} c_k^2 \tag{1.10}$$

The normalisation factor $\frac{1}{\sqrt{n}}$ in front of the FT sums is sometimes replaced by a single $\frac{1}{n}$ in front of the inverse FT sum which is often convenient in computation. Then, of course, Parseval's equation has to be modified accordingly.

A straightforward implementation of the discrete Fourier transform, i.e. the computation of n sums each of length n requires $\sim n^2$ operations.

Code 1.1 (Fourier transform by definition) *Compute the Fourier transform of the complex sequence* `a[]`*, the result is returned in* `c[]`

```
procedure ft(a[],c[],n,is)
{
    for k:=0 to n-1
    {
        s := 0
        for x:=0 to n-1
        {
            s := s + a[x]*exp(is*2.0*I*PI*x*k/n)
        }
        c[k] := s
    }
}
```

[FXT: `slow_ft` in file `slow/slowft.cc`]

A *fast Fourier transform* (FFT) algorithm is an algorithm that improves the operation count to proportional $n \sum_{k=1}^{m} (p_k - 1)$, where $n = p_1 p_2 \cdots p_m$ is a factorization of n. In case of a power $n = p^m$ the value computes to $n\,(p-1)\,\log_p(n)$. In the special case $p = 2$ even $n/2\,\log_2(n)$ multiplications are enough. There are several different FFT algorithms with many variants.

1.2 Symmetries of the Fourier transform

The FT has several symmetry properties, a bit of notation turns out to be useful becoming written down. Let $\overline{a}$ be the sequence a (length n) reversed around element with index $n/2$:

$$\overline{a}_0 \quad := \quad a_0 \tag{1.11}$$

$$\overline{a}_{n/2} \quad := \quad a_{n/2} \qquad \text{if } n \text{ even} \tag{1.12}$$

$$\overline{a}_k \quad := \quad a_{n-k} \tag{1.13}$$

Let a_S, a_A be the symmetric, antisymmetric part of the sequence a, respectively:

$$a_S \quad := \quad a + \overline{a} \tag{1.14}$$

$$a_A \quad := \quad a - \overline{a} \tag{1.15}$$

(The elements with indices 0 and $n/2$ of a_A are zero). Now let $a \in \mathbb{R}$ (meaning that each element of a is $\in \mathbb{R}$), then

$$\mathcal{F}[a_S] \quad \in \quad \mathbb{R} \tag{1.16}$$

$$\mathcal{F}[a_S] \quad = \quad \overline{\mathcal{F}[a_S]} \tag{1.17}$$

$$\mathcal{F}[a_A] \quad \in \quad i\,\mathbb{R} \tag{1.18}$$

$$\mathcal{F}[a_A] \quad = \quad -\overline{\mathcal{F}[a_A]} \tag{1.19}$$

i.e. the FT of a real symmetric sequence is real and symmetric and the FT of a real antisymmetric sequence is purely imaginary and antisymmetric. Thereby the FT of a general real sequence is the complex conjugate of its reversed:

$$\mathcal{F}[a] \quad = \quad \overline{\mathcal{F}[a]}^* \qquad for \quad a \in \mathbb{R} \tag{1.20}$$

Similarly, for a purely imaginary sequence $b \in i\mathbb{R}$:

$$\mathcal{F}[b_S] \quad \in \quad i\,\mathbb{R} \tag{1.21}$$
$$\mathcal{F}[b_S] \quad = \quad \overline{\mathcal{F}[b_S]} \tag{1.22}$$
$$\mathcal{F}[b_A] \quad \in \quad \mathbb{R} \tag{1.23}$$
$$\mathcal{F}[b_A] \quad = \quad -\overline{\mathcal{F}[b_A]} \tag{1.24}$$

The FT of a complex symmetric/antisymmetric sequence is symmetric/antisymmetric, respectively.

1.3 Radix 2 FFT algorithms

1.3.1 A little bit of notation

Always assume a is a length-n sequence (n even) in what follows:

Let $a^{(even)}$, $a^{(odd)}$ denote the (length-$n/2$) subsequences of those elements of a that have even or odd indices, respectively.

Let $a^{(left)}$ denote the subsequence of those elements of a that have indices $0...n/2-1$.

Similarly, $a^{(right)}$ for indices $n/2...n-1$.

Let $\mathcal{S}^k a$ denote the sequence with elements $a_x\, e^{\pm k\, 2\,\pi\, i\, x/n}$ where n is the length of the sequence a and the sign is that of the transform. The symbol $\mathcal{S}$ shall suggest a shift operator. In the next two sections only $\mathcal{S}^{1/2}$ will appear. $\mathcal{S}^0$ is the identity operator.

1.3.2 Decimation in time (DIT) FFT

The following observation is the key to the decimation in time (DIT) FFT[2] algorithm:
For n even the k-th element of the Fourier transform is

$$\sum_{x=0}^{n-1} a_x\, z^{x\,k} \quad = \quad \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{(2\,x+1)\,k} \tag{1.25}$$
$$= \quad \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{2\,x\,k} \tag{1.26}$$

where $z = e^{\pm i\,2\,\pi/n}$ and $k \in \{0, 1, ..., n-1\}$.

The last identity tells us how to compute the k-th element of the length-n Fourier transform from the length-$n/2$ Fourier transforms of the even and odd indexed subsequences.

To actually rewrite the length-n FT in terms of length-$n/2$ FTs one has to distinguish the cases $0 \le k < n/2$ and $n/2 \le k < n$, therefore we rewrite $k \in \{0, 1, 2, ..., n-1\}$ as $k = j + \delta\,\frac{n}{2}$ where $j \in \{0, 1, ..., n/2-1\}$, $\quad \delta \in \{0, 1\}$.

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(j+\delta\,\frac{n}{2})} \quad = \quad \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} + z^{j+\delta\,\frac{n}{2}} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} \tag{1.27}$$

[2] also called Cooley-Tukey FFT.

$$= \begin{cases} \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} + z^j \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} - z^j \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{1.28}$$

Noting that z^2 is just the root of unity that appears in a length-$n/2$ FT one can rewrite the last two equations as the

Idea 1.1 (FFT radix 2 DIT step) *Radix 2 decimation in time step for the FFT:*

$$\mathcal{F}\left[a\right]^{(left)} \overset{n/2}{=} \mathcal{F}\left[a^{(even)}\right] + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \tag{1.29}$$

$$\mathcal{F}\left[a\right]^{(right)} \overset{n/2}{=} \mathcal{F}\left[a^{(even)}\right] - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \tag{1.30}$$

(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 proportional $n/2\ \log_2(n)$.

Code 1.2 (recursive radix 2 DIT FFT) *Pseudo code for a recursive procedure of the (radix 2) DIT FFT algorithm,* is *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then  // end of recursion
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1  // copy to workspace
    {
        s[k] := a[2*k]     // even indexed elements
        t[k] := a[2*k+1]   // odd  indexed elements
    }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[],nh,b[],is)
    rec_fft_dit2(t[],nh,c[],is)

    fourier_shift(c[],nh,is*1/2)

    for k:=0 to nh-1  // copy back from workspace
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The data length n must be a power of 2. The result is in x[]. Note that normalisation (i.e. multiplication of each element of x[] by $1/\sqrt{n})$ is not included here.

[FXT: recursive_dit2_fft in file learn/recfftdit2.cc] The procedure uses the subroutine

Code 1.3 (Fourier shift) *For each element in c[0..n-1] replace c[k] by c[k] times $e^{v\,2\,\pi\,i\,k/n}$. Used with $v = \pm 1/2$ for the Fourier transform.*

```
procedure fourier_shift(c[], n, v)
{
    for k:=0 to n-1
    {
        c[k] := c[k] * exp(v*2.0*PI*I*k/n)
    }
}
```

cf. [FXT: fourier_shift in file fft/fouriershift.cc]

The recursive FFT-procedure involves a lot of function calls, this can be avoided by rewriting it in a non-recursive way. One can even do all operations *in place*, no workspace array is needed at all. The price is the necessity of an additional data reordering: The procedure `revbin_permute(a[],n)` rearranges 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. This is discussed in section 1.7.

Code 1.4 (radix 2 DIT FFT, naive) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* `is` *must be -1 or +1: (naive version, needs to be improved)*

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[],n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for r:=0 to n-m step m  // n/m iterations
        {
            for j:=0 to mh-1  // m/2 iterations
            {
                e := exp(is*2*PI*I*j/m)  // log_2(n)*n/m*m/2 = log_2(n)*n/2 computations

                u := a[r+j]
                v := a[r+j+mh] * e

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

[FXT: dit2_fft_localized in file learn/fftdit2.cc]

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in place. It works as given, but is a bit wasteful. The (expensive!) computation `e := exp(is*2*PI*I*j/m)` is done `log_2(n) n/2` times. To reduce the number of trigonometric computations, one can swap the two inner loops, leading to the first 'real world' FFT procedure presented here:

Code 1.5 (radix 2 DIT FFT) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* `is` *must be -1 or +1:*

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn

    revbin_permute(a[],n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for j:=0 to mh-1  // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m)  // 1 + 2 + ... + n/8 + n/4 + n/2 = n-1 computations
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e

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

[FXT: dit2_fft in file learn/fftdit2.cc]

Swapping the two inner loops reduces the number of trigonometric (exp()) computations to n but leads to a feature that many FFT implementations share: Memory access is highly nonlocal. For each recursion stage (value of ldm) the array is traversed mh times with n/m accesses in strides of mh. As mh is a power of 2 this can (on computers that use memory cache) have a very negative performance impact for large values of n. On a computer where the CPU clock (366MHz, AMD K6/2) is 5.5 times faster than the memory clock (66MHz, EDO-RAM) I found that indeed for small n the naive FFT is slower by a factor of about 0.66, but for large n the same ratio is in favour of the 'naive' procedure!

It is a good idea to extract the ldm==1 stage of the outermost loop, this avoids complex multiplications with the trivial factors $1 + 0\,i$: Replace

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

by

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
    for ldm:=2 to ldn
    {
```

1.3.3 Decimation in frequency (DIF) FFT

The simple splitting of the Fourier sum into a left and right half (for n even) leads to the decimation in frequency (DIF) FFT[3]:

$$\begin{aligned}
\sum_{x=0}^{n-1} a_x\, z^{x\,k} &= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n} a_x\, z^{x\,k} && (1.31)\\
&= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=0}^{n/2-1} a_{x+n/2}\, z^{(x+n/2)\,k} && (1.32)\\
&= \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{k\,n/2}\, a_x^{(right)}\right) z^{x\,k} && (1.33)
\end{aligned}$$

(where $z = e^{\pm i\,2\,\pi/n}$ and $k \in \{0, 1, ..., n-1\}$)

Here one has to distinguish the cases k even or odd, therefore we rewrite $k \in \{0, 1, 2, ..., n-1\}$ as $k = 2\,j+\delta$ where $j \in \{0, 2, ..., \frac{n}{2} - 1\}$, $\delta \in \{0, 1\}$.

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(2\,j+\delta)} = \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{(2\,j+\delta)\,n/2}\, a_x^{(right)}\right) z^{x\,(2\,j+\delta)} \qquad (1.34)$$

[3] also called Sande-Tukey FFT, cf. [28].

$$= \begin{cases} \sum\limits_{x=0}^{n/2-1} (a_x^{(left)} + a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum\limits_{x=0}^{n/2-1} z^x (a_x^{(left)} - a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{1.35}$$

$z^{(2\,j+\delta)\,n/2} = e^{\pm \pi\, i\, \delta}$ is equal to plus/minus 1 for $\delta = 0/1$ (k even/odd), respectively.

The last two equations are, more compactly written, the

Idea 1.2 (radix 2 DIF step) *Radix 2 decimation in frequency step for the FFT:*

$$\mathcal{F}\,[a]^{(even)} \overset{n/2}{=} \mathcal{F}\left[a^{(left)} + a^{(right)}\right] \tag{1.36}$$

$$\mathcal{F}\,[a]^{(odd)} \overset{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{1.37}$$

Code 1.6 (recursive radix 2 DIF FFT) *Pseudo code for a recursive procedure of the (radix 2) decimation in frequency FFT algorithm,* `is` *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dif2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace
    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2
    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left'  elements
        t[k] := a[k+nh]  // 'right' elements
    }
    for k:=0 to nh-1
    {
        {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
    }
    fourier_shift(t[],nh,is*0.5)
    rec_fft_dif2(s[],nh,b[],is)
    rec_fft_dif2(t[],nh,c[],is)
    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

The data length `n` must be a power of 2. The result is in `x[]`.

[FXT: `recursive_dif2_fft` in file `learn/recfftdif2.cc`]

The non-recursive procedure looks like this:

Code 1.7 (radix 2 DIF FFT) *Pseudo code for a non-recursive procedure of the (radix 2) DIF algorithm,* `is` *must be -1 or +1:*

```
procedure fft_dif2(a[],ldn,is)
```

```
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-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
            }
        }
    }
    revbin_permute(a[],n)
}
```

cf. [FXT: dif2_fft in file learn/fftdif2.cc]

In DIF FFTs the revbin_permute()-procedure is called after the main loop, in the DIT code it was called before the main loop. As in the procedure 1.5 the inner loops where swapped to save unnecessary trigonometric computations.

Extracting the ldm==1 stage of the outermost loop is again a good idea:
Replace the line

```
    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
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
```

before the call of revbin_permute(a[],n).

1.4 Saving trigonometric computations

The trigonometric (sin()- and cos()-) computations are an expensive part of any FFT. There are two apparent ways for saving the involved cpu-cycles, the use of lookup-tables and recursive methods for trig-computations.

Using lookup tables

The idea is to save all necessary sin/cos-values in an array and later looking up the values needed. This is a good idea if one wants to compute many FFTs of the same (small) length. For FFTs of large sequences one gets large lookup tables that likely introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a slowdown isn't unlikely. However, for a length-n FFT one doesn't need to store all the (n complex or $2\,n$ real) sin/cos-values $\exp(2\,\pi\,i\,k/n)$, $k = 0, 1, 2, 3, ..., n-1$. Already a table $\cos(2\,\pi\,i\,k/n)$, $k = 0, 1, 2, 3, ..., n/4-1$ (of $n/4$ reals) contains all different trig-values that occur in the computation. The size of the trig-table is thereby cut by a factor of 8. For the lookups one can use the symmetry relations

$$\begin{aligned} \cos(\pi + x) &= -\cos(x) && (1.38) \\ \sin(\pi + x) &= -\sin(x) && (1.39) \end{aligned}$$

(reducing the interval from 0...2π to 0...π),

$$\begin{aligned} \cos(\pi/2 + x) &= -\sin(x) && (1.40) \\ \sin(\pi/2 + x) &= +\cos(x) && (1.41) \end{aligned}$$

(reducing the interval to 0...$\pi/2$) and

$$\sin(x) = \cos(\pi/2 - x) \qquad (1.42)$$

(only cos()-table needed).

Recursive trig-computation

In the computation of FFTs one typically needs the values

$$\{\exp(i\,\omega\,0) = 1, \quad \exp(i\,\omega\,\delta), \quad \exp(i\,\omega\,2\,\delta), \quad \exp(i\,\omega\,3\,\delta), \quad ...\}$$

in sequence. The naive idea for a recursive computation of these values is to precompute $d = \exp(i\,\omega\,\delta)$ and then compute the next following value using the identity $\exp(i\,\omega\,k\,\delta)) = d \cdot \exp(i\,\omega\,(k-1)\,\delta)$. This method, however, is of no practical value because the numerical error grows (exponentially) in the process.

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

$$\begin{aligned} c &= \cos\omega, && (1.43) \\ s &= \sin\omega, && (1.44) \\ \alpha &= 2\,(\sin\frac{\delta}{2})^2 && (1.45) \\ \beta &= \sin\delta && (1.46) \end{aligned}$$

Then compute the next power from the previous as:

$$\begin{aligned} c_{next} &= c - (\alpha\,c + \beta\,s); && (1.47) \\ s_{next} &= s - (\alpha\,s - \beta\,c); && (1.48) \end{aligned}$$

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 (in C) working with `doubles` it might be a good idea to use the type `long double` with the trig recursion: the sin and cos will than always be accurate within the `double`-precision.

Using higher radix algorithms

It may be less apparent, that the use of higher radix FFT algorithms also saves trig-computations. The radix-4 FFT algorithms presented in the next sections replace all multiplications with complex factors $(0, \pm i)$ by the obvious simpler operations. Radix-8 algorithms also simplify the special cases where $\sin(\phi)$ or $\cos(\phi)$ are $\pm\sqrt{1/2}$. Apart from the trig-savings higher radix also brings a performance gain by their more unrolled structure.

1.5 Higher radix DIT and DIF algorithms

1.5.1 More notation

Again some useful notation, again let a be a length-n sequence.

Let $a^{(r\%m)}$ denote the subsequence of those elements of a that have subscripts $x \equiv r \ (mod\, m)$; e.g. $a^{(0\%2)}$ is $a^{(even)}$, $a^{(3\%4)} = \{a_3, a_7, a_{11}, a_{15}, ...\}$. The length of $a^{(r\%m)}$ is[4] n/m.

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. The length of $a^{(r/m)}$ is also n/m.

1.5.2 Decimation in time

First reformulate the radix 2 DIT step (formulas 1.29 and 1.30) in the new notation:

$$\mathcal{F}\left[a\right]^{(0/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right]_{n/2} + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right]_{n/2} \tag{1.49}$$

$$\mathcal{F}\left[a\right]^{(1/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right]_{n/2} - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right]_{n/2} \tag{1.50}$$

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

The radix 4 step, whose derivation is analogue as for the radix 2 step, it just involves more writing and doesn't give additional insights, is

Idea 1.3 (radix 4 DIT step) *Radix 4 decimation in time step for the FFT:*

$$\mathcal{F}\left[a\right]^{(0/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.51}$$

$$\mathcal{F}\left[a\right]^{(1/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.52}$$

$$\mathcal{F}\left[a\right]^{(2/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.53}$$

$$\mathcal{F}\left[a\right]^{(3/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.54}$$

where $\sigma = \pm 1$ is the sign in the exponent. In contrast to the radix 2 step, that happens to be identical for forward and backward transform (with both decimation frequency/time) the sign of the transform appears here.

Or, more compact:

$$\begin{aligned}\mathcal{F}\left[a\right]^{(j/4)} \overset{n/4}{=} & +e^{\sigma\,2\,i\,\pi\,0\,j/4} \cdot \mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + e^{\sigma\,2\,i\,\pi\,1\,j/4} \cdot \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] \\ & +e^{\sigma\,2\,i\,\pi\,2\,j/4} \cdot \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + e^{\sigma\,2\,i\,\pi\,3\,j/4} \cdot \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.55}$$

where $j = 0, 1, 2, 3$ and n is a multiple of 4.

Still more compact:

$$\mathcal{F}\left[a\right]^{(j/4)} \overset{n/4}{=} \sum_{k=0}^{3} e^{\sigma 2\,i\,\pi\,k\,j/4} \cdot \mathcal{S}^{\sigma k/4}\mathcal{F}\left[a^{(k\%4)}\right] \tag{1.56}$$

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, applicable when n is a multiple of r, is:

[4] Throughout this book will m divide n, so the statement is correct.

Idea 1.4 (FFT general DIT step) *General decimation in time step for the FFT:*

$$\mathcal{F}[a]^{(j/r)} \stackrel{n/r}{=} \sum_{k=0}^{r-1} e^{\sigma\, 2\, i\, \pi\, k\, j/r} \cdot \mathcal{S}^{\sigma\, k/r} \mathcal{F}\left[a^{(k\%r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.57}$$

1.5.3 Decimation in frequency

The radix 2 DIF step (formulas 1.36 and 1.37) was

$$\mathcal{F}[a]_n^{(0\%2)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{0/2}\left(a^{(0/2)} + a^{(1/2)}\right)\right] \tag{1.58}$$

$$\mathcal{F}[a]_n^{(1\%2)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(0/2)} - a^{(1/2)}\right)\right] \tag{1.59}$$

The radix 4 DIF step, applicable for n divisible by 4, is

Idea 1.5 (radix 4 DIF step) *Radix 4 decimation in frequency step for the FFT:*

$$\mathcal{F}[a]^{(0\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{0/4}\left(a^{(0/4)} + a^{(1/4)} + a^{(2/4)} + a^{(3/4)}\right)\right] \tag{1.60}$$

$$\mathcal{F}[a]^{(1\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{1/4}\left(a^{(0/4)} + i\,\sigma\, a^{(1/4)} - a^{(2/4)} - i\,\sigma\, a^{(3/4)}\right)\right] \tag{1.61}$$

$$\mathcal{F}[a]^{(2\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{2/4}\left(a^{(0/4)} - a^{(1/4)} + a^{(2/4)} - a^{(3/4)}\right)\right] \tag{1.62}$$

$$\mathcal{F}[a]^{(3\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{3/4}\left(a^{(0/4)} - i\,\sigma\, a^{(1/4)} - a^{(2/4)} + i\,\sigma\, a^{(3/4)}\right)\right] \tag{1.63}$$

Or, more compact:

$$\mathcal{F}[a]^{(j\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\, j/4} \sum_{k=0}^{3} e^{\sigma\, 2\, i\, \pi\, k\, j/4} \cdot a^{(k/4)}\right] \tag{1.64}$$

where $j = 0, 1, 2, 3$ and the sign of the exponent and in the shift operator is the same as in the transform.

The general radix r DIF step is

Idea 1.6 (FFT general DIF step) *General decimation in frequency step for the FFT:*

$$\mathcal{F}[a]^{(j\%r)} \stackrel{n/r}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\, j/r} \sum_{k=0}^{r-1} e^{\sigma\, 2\, i\, \pi\, k\, j/r} \cdot a^{(k/r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.65}$$

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

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

Code 1.8 (radix p^x DIT FFT) *Pseudo code for a radix* `r:=`p^x *decimation in time FFT:*

```
procedure fftdit_r(a[], n, is)
// complex a[0..n-1] input, result
// p (hardcoded)
// r == power of p (hardcoded)
// n == power of p (not necessarily a power of r)
{
    radix_permute(a[], n, p)

    lx := log(r) / log(p)  // r == p ** lx
    ln := log(n) / log(p)
    ldm := (log(n)/log(p)) % lx

    if ( ldm != 0 )  // n is not a power of p
    {
        xx := p**lx
        for z:=0 to n-1 step xx
        {
            fft_dit_xx(a[z..z+xx-1], is)  // inlined length-xx dit fft
        }
    }

    for ldm:=ldm+lx to ln step lx
    {
        m  := p**ldm
        mr := m/r

        for j := 0 to mr-1
        {
            e := exp(is*2*PI*I*j/m)

            for k:=0 to n-1 step m
            {
                // all code in this block should be
                // inlined, unrolled and fused:

                // temporary  u[0..r-1]

                for z:=0 to r-1
                {
                    u[z] := a[k+j+mr*z]
                }

                radix_permute(u[], r, p)

                for z:=1 to r-1  // e**0 = 1
                {
                    u[z] := u[z] * e**z
                }

                r_point_fft(u[], is)

                for z:=0 to r-1
                {
                    a[k+j+mr*z] := u[z]
                }
            }
        }
    }
}
```

Of course the loops that use the variable z have to be unrolled, the (length-p^x) scratch space u[] has to be replaced by explicit variables (e.g. u0, u1, ...) and the r_point_fft(u[],is) shall be an inlined p^x-point FFT.

If $r = p^x$ than there is a pitfall one must now: if one uses the radix_permute() procedure instead of a radix-p^x revbin_permute procedure (e.g. radix-2 revbin_permute for a radix-4 FFT), then some additional reordering is necessary in the innermost loop: in the above pseudo code this is indicated by the radix_permute(u[],p) just before the p_point_fft(u[],is) line. One wouldn't really use a call to a procedure, but change indices in the loops where the a[z] are read/written for the DIT/DIF respectively. In the code below the respective lines have the comment // (!).

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[5]. 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.

Code 1.9 (radix 4 DIT FFT) *C++ code for a radix 4 DIF FFT on the array* f[]*, the data length* n *must be a power of 2,* is *must be +1 or -1:*

```
static const ulong RX = 4;  // == r
```

[5] cf. section 5.3.

```
static const ulong LX = 2;  // == log(r)/log(p) == log_2(r)
void
dit4l_fft(Complex *f, ulong ldn, int is)
// decimation in time radix 4 fft
// ldn == log_2(n)
{
    double s2pi = ( is>0 ? 2.0*M_PI : -2.0*M_PI );
    const ulong n = (1<<ldn);
    revbin_permute(f, n);
    ulong ldm = (ldn&1);  // == (log(n)/log(p)) % LX
    if ( ldm!=0 )  // n is not a power of 4, need a radix 2 step
    {
        for (ulong r=0; r<n; r+=2)
        {
            Complex a0 = f[r];
            Complex a1 = f[r+1];
            f[r]   = a0 + a1;
            f[r+1] = a0 - a1;
        }
    }
    ldm += LX;
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        ulong m = (1<<ldm);
        ulong m4 = (m>>LX);
        double ph0 = s2pi/m;
        for (ulong j=0; j<m4; j++)
        {
            double phi = j*ph0;
            double c, s, c2, s2, c3, s3;
            sincos(phi, &s, &c);
            sincos(2.0*phi, &s2, &c2);
            sincos(3.0*phi, &s3, &c3);
            Complex e  = Complex(c,s);
            Complex e2 = Complex(c2,s2);
            Complex e3 = Complex(c3,s3);

            for (ulong r=0, i0=j+r;  r<n;  r+=m, i0+=m)
            {
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;
                Complex a0 = f[i0];
                Complex a1 = f[i2]; // (!)
                Complex a2 = f[i1]; // (!)
                Complex a3 = f[i3];
                a1 *= e;
                a2 *= e2;
                a3 *= e3;
                Complex t0 = (a0+a2) + (a1+a3);
                Complex t2 = (a0+a2) - (a1+a3);
                Complex t1 = (a0-a2) + Complex(0,is) * (a1-a3);
                Complex t3 = (a0-a2) - Complex(0,is) * (a1-a3);
                f[i0] = t0;
                f[i1] = t1;
                f[i2] = t2;
                f[i3] = t3;
            }
        }
    }
}
```

Code 1.10 (radix 4 DIF FFT) *Pseudo code for 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[],ldn,is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm := ldn to 2 step -2
    {
        m  := 2**ldm
        mr := m/4
        for j := 0 to mr-1
        {
            e  := exp(is*2*PI*I*j/m)
            e2 := e * e
            e3 := e2 * e
            for r := 0 to n-1 step m
            {
                u0 := a[r+j]
                u1 := a[r+j+mr]
                u2 := a[r+j+mr*2]
                u3 := a[r+j+mr*3]
                x := u0 + u2
                y := u1 + u3
                t0 := x + y  // == (u0+u2) + (u1+u3)
                t1 := x - y  // == (u0+u2) - (u1+u3)
                x := u0 - u2
                y := (u1 - u3)*I*is
                t2 := x + y  // == (u0-u2) + (u1-u3)*I*is
                t3 := x - y  // == (u0-u2) - (u1-u3)*I*is
                t1 := t1 * e
                t2 := t2 * e2
                t3 := t3 * e3
                a[r+j]      := t0
                a[r+j+mr]   := t2  // (!)
                a[r+j+mr*2] := t1  // (!)
                a[r+j+mr*3] := t3
            }
        }
    }
    if is_odd(ldn) then  // n not a power of 4
    {
        for r:=0 to n-1 step 2
        {
            {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
        }
    }
    revbin_permute(a[],n)
}
```

Note the 'swapped' order in which t1, t2 are copied back in the innermost loop, this is what radix_permute(u[], r, p) was supposed to do.

The multiplication by the imaginary unit (in the statement y := (u1 - u3)*I*is) should of course be implemented without any multiplication statement: one could unroll it as

```
(dr,di) := u1 - u2   // dr,di = real,imag part of difference
      if is>0 then  y := (-di,dr)  // use (a,b)*(0,+1) == (-b,a)
      else          y := (di,-dr)  // use (a,b)*(0,-1) == (b,-a)
```

In section 1.6 it is shown how the if-statement can be eliminated.

If n is not a power of 4, then ldm is odd during the procedure and at the last pass of the main loop one has ldm=1.

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 := ldn to 3 step -X
and at the last pass of the main loop one has ldm=3 or ldm=2.

[FXT: dit4l_fft in file learn/fftdit4l.cc] [FXT: dif4l_fft in file learn/fftdif4l.cc] [FXT: dit4_fft in file fft/fftdit4.cc] [FXT: dif4_fft in file fft/fftdif4.cc]

Code 1.11 (radix permute) *C++ code for the radix permutation of the array* f[],

```
extern ulong nt[];  // nt[] = 9, 90, 900  for r=10, x=3
extern ulong kt[];  // kt[] = 1, 10, 100  for r=10, x=3

template <typename Type>
void
radix_permute(Type *f, ulong n, ulong r)
//
// swap elements with index pairs i, j were the
// radix-r representation of i and j are mutually
// digit-reversed (e.g. 436 <--> 634)
//
// This is a radix-r generalization of revbin_permute()
// revbin_permute(f, n) =^= radix_permute(f, n, 2)
//
// must be:
//  n == p**x for some x>=1
//  r >= 2
//
{
    ulong x = 0;
    nt[0] = r-1;
    kt[0] = 1;
    while ( 1 )
    {
        ulong z = kt[x] * r;
        if ( z>n )  break;
        ++x;
        kt[x] = z;

        nt[x] = nt[x-1] * r;
    }
    // here: n == p**x

    for (ulong i=0, j=0;  i < n-1;  i++)
    {
        if ( i<j )  swap(f[i], f[j]);

        ulong t = x - 1;
        ulong k = nt[t];  // =^=  k = (r-1) * n / r;

        while ( k<=j )
        {
            j -= k;
            k = nt[--t];  // =^=  k /= r;
        }

        j += kt[t]; // =^=  j += (k/(r-1));
    }
}
```

[FXT: radix_permute in file permute/radixpermute.h]

1.6 Inverse FFT for free

Suppose you programmed some FFT algorithm just for one value of is, the sign in the exponent. There is a nice trick that gives the inverse transform for free, if your implementation uses seperate arrays for real and imaginary part of the complex sequences to be transformed. If your procedure is something like

```
procedure my_fft(ar[], ai[], ldn)  // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
    // incredibly complicated code
    // that you can't see how to modify
    // for is==-1
}
```

Then you *don't* need to modify this procedure at all in order to get the inverse transform. If you want the inverse transform somewhere then just, instead of

```
    my_fft(ar[], ai[], ldn)  // forward fft
```

type

```
    my_fft(ai[], ar[], ldn)  // backward fft
```

Note the swapped real- and imaginary parts ! The same trick works if your procedure coded for fixed `is`$= -1$.

1.7 The revbin permute operation

The procedure `revbin_permute(a[],n)` used in the DIF and DIT FFT algorithms rearranges 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. For example if $n = 256$ and $x = 43_{10} = 00101011_2$ then $\tilde{x} = 11010100_2 = 212_{10}$. Note that $\tilde{x}$ depends both on x and on n.

A naive version

Code 1.12 (revbin_permute, naive)

```
procedure revbin_permute(a[],n)
// a[0..n-1] input,result
{
    for x:=0 to n-1
    {
        r := revbin(x,n)
        if r>x then  swap(a[x],a[r])
    }
}
```

The function `revbin(x,n)` shall return the reversed bits of `x`.

Code 1.13 (revbin)

```
function revbin(x,n)
{
    j := 0
    ldn := log2(n)  // is an integer
    while ldn>0
    {
        j := j << 1
        j := j + (x & 1)
        x := x >> 1
        ldn := ldn - 1
    }
    return j
}
```

The condition `r>x` before the `swap()` statement makes sure that the swapping isn't undone when the loop variable `x` has the value of the present `r`. This version of the `revbin_permute`-routine is pretty unefficient (even if `revbin()` is inlined and `ldn` is only computed once). Each execution of `revbin()` costs proportional `ldn` operations, giving a total of proportional $\frac{n}{2}\log_2(n)$ operations (neglecting the swaps for the moment). One can do better.

A fast version

The key idea is to compute the value $\tilde{x}$ from the value $\widetilde{x-1}$. As x is one added to $x-1$, $\tilde{x}$ is one 'reversed' added to $\widetilde{x-1}$ if one finds a routine for that 'reversed add' update much of the computation can be saved.

Code 1.14 (revbin update) *Update* `r`*, that must be the same as the the result of* `revbin(x-1,n)` *to what would be the result of* `revbin(x,n)`

```
function revbin_update(r,n)
{
    do
    {
        n := n >> 1
        r := r^n  // bitwise exor
    } while ((r&n) == 0)
    return r
}
```

In C this can be cryptified to an efficient piece of code:

```
inline unsigned revbin_update(unsigned r, unsigned n)
{
    for (unsigned m=n>>1; (!((r^=m)&m)); m>>=1);
    return r;
}
```

Now we are ready for

Code 1.15 (revbin_permute, fast) *Put data in revbin order*

```
procedure revbin_permute(a[],n)
// a[0..n-1] input,result
{
    if n<=2  return
    r := 0  // the reversed 0
    for x:=1 to n-1
    {
        r := revbin_update(r,n)  // inline me
        if r>x then  swap(a[x],a[r])
    }
}
```

This routine is several times faster than the naive version. `revbin_update()` does for half of the calls just one iteration because in half of the updates just the leftmost bit changes[6], in half of the remaining updates it does two iterations, in half of the still remaining updates it three and so on. The total number operations done by `revbin_update()` is therefore proportional to $n\,(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots+\frac{\log_2(n)}{n})$ which is $n\sum_{j=1}^{\log_2(n)}\frac{j}{2^j}$ for n large this sum converges against $2n$. Thereby the asymptotics of `revbin_permute()` is improved from proportional $n\,\log(n)$ to proportional n.

How many swaps ?

How many `swap()`-statements will be executed in total for different n ? About $n-\sqrt{n}$, as there are only few numbers with symmetric bit patterns: for even $log_2(n) =: 2\,b$ the left half of the bit pattern must be the reversed of the right half. There are $2^b = \sqrt{2^{2b}}$ such numbers. For odd $log_2(n) =: 2\,b+1$ there are twice as much symmetric patterns, the bit in the middle does not matter and can be 0 or 1.

[6] corresponding to the change in only the rightmost bit if one is added to an even number

n	2 # swaps	# symm. pairs
2	0	2
4	2	2
8	4	4
16	12	4
32	24	8
64	56	8
2^{10}	992	32
2^{20}	$0.999 \cdot 2^{20}$	2^{10}
∞	$n - \sqrt{n}$	$\sqrt{n}$

Summarizing: almost all 'revbin-pairs' will be swapped by `revbin_permute()`.

A still faster version

x	x_2	$\tilde{x}_2$	$\tilde{x}$	Δ	$\tilde{x} > x$?
0	00000	00000	0	-31	
1	00001	10000	16	16	y
2	00010	01000	8	-8	y
3	00011	11000	24	16	y
4	00100	00100	4	-20	
5	00101	10100	20	16	y
6	00110	01100	12	-8	y
7	00111	11100	28	16	y
8	01000	00010	2	-26	
9	01001	10010	18	16	y
10	01010	01010	10	-8	
11	01011	11010	26	16	y
12	01100	00110	6	-20	
13	01101	10110	22	16	y
14	01110	01110	14	-8	
15	01111	11110	30	16	y
16	10000	00001	1	-29	
17	10001	10001	17	16	
18	10010	01001	9	-8	
19	10011	11001	25	16	y
20	10100	00101	5	-20	
21	10101	10101	21	16	
22	10110	01101	13	-8	
23	10111	11101	29	16	y
24	11000	00011	3	-26	
25	11001	10011	19	16	
26	11010	01011	11	-8	
27	11011	11011	27	16	
28	11100	00111	7	-20	
29	11101	10111	23	16	
30	11110	01111	15	-8	
31	11111	11111	31	16	

where the subscript 2 indicates printing in base 2, $\Delta := \tilde{x} - \widetilde{x-1}$ and an 'y' in the last column marks index pairs where `revbin_permute()` will swap elements.

Observation one: $\Delta = \frac{n}{2}$ for all odd x.

Observation two: if for even $x < \frac{n}{2}$ there is a swap (for the pair x, $\tilde{x}$) then there is also a swap for the pair $n-1-x$, $n-1-\tilde{x}$. As $x < \frac{n}{2}$ and $\tilde{x} < \frac{n}{2}$ one has $n-1-x > \frac{n}{2}$ and $n-1-\tilde{x} > \frac{n}{2}$, i.e. the swaps are independent.

There should be no difficulties to cast these observations into

Code 1.16 (revbin_permute, faster) *Put data in revbin order*

```
procedure revbin_permute(a[],n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        swap(a[x],a[r])
        x := x + 1

        // x even:
        r := revbin_update(r,n)  // inline me
        if r>x then
        {
            swap(a[x],a[r])
            swap(a[n-1-x],a[n-1-r])
        }
        x := x + 1
    }
}
```

The `revbin_update()` would be in C, inlined and the first stage of the loop extracted

```
        r^=nh;  for (unsigned m=(nh>>1); !((r^=m)&m); m>>=1)  {}
```

The code above is an ideal candidate to derive an optimised version for zero padded data:

Code 1.17 (revbin_permute for zero padded data) *Put zero padded data in revbin order*

```
procedure revbin_permute0(a[],n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        a[r] := a[x]
        a[x] := 0
        x := x + 1

        // x even:
        r := revbin_update(r,n)  // inline me
        if r>x then  swap(a[x],a[r])
        x := x + 1
    }
}
```

One could carry the scheme that lead to the 'faster' revbin_permute procedures further, e.g. using 3 hardcoded constants Δ_1, Δ_2, Δ_3 depending on whether $x \bmod 4 = 1, 2, 3$ only calling `revbin_update()` for $x \bmod 4 = 0$. However, the code quickly gets quite complicated and there seems to be no measurable gain in speed, even for very large sequences.

If, for complex data, one works with seperate arrays for real and imaginary part[7] one might be tempted to do away with half of the bookkeeping as follows: write a special procedure `revbin_permute(a[],b[],n)` that shall replace the two successive calls `revbin_permute(a[],n)` and `revbin_permute(b[],n)` and has after each statement `swap(a[x],a[r])` inserted a `swap(b[x],b[r])`. If you do so, be prepared for disaster! Very likely the real and imaginary element for the same index lie apart in memory by a power of two, leading to one hundred percent cache miss for the typical computer. Even in the most favourable case the cache miss rate will be increased. Do expect to hardly ever win anything noticable but in most cases to lose big. Think about it, whisper "*direct mapped cache*" and forget it.

[FXT: `revbin_permute` in file `permute/revbinpermute.h`]

[7] as opposed to: using a data type 'complex' with real and imaginary part of each number in consecutive places

1.8 Real valued Fourier transforms

The Fourier transform of a purely real sequence $c = \mathcal{F}[a]$ where $a \in \mathbb{R}$ has[8] 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
- usage of the fast Hartley transform
- a variant of the matrix Fourier algorithm
- special real (split radix algorithm) FFTs

All techniques have in common that they store only half of the complex result to avoid the redundancy due to the symmetries of a complex FT of purely real input. The result of a real to (half-) complex FT (abbreviated R2CFT) must contain the purely real components c_0 (the DC-part of the input signal) and, in case n is even, $c_{n/2}$ (the nyquist frequency part). The inverse procedure, the (half-) complex to real transform (abbreviated C2RFT) must be compatible to the ordering of the R2CFT. The procedures presented here use the following ordering of the real part of the resulting data c in the output array `a[]`:

$$\begin{aligned} \mathtt{a[0]} &= \Re c_0 \\ \mathtt{a[1]} &= \Re c_1 \\ \mathtt{a[2]} &= \Re c_2 \\ &\dots \\ \mathtt{a[n/2]} &= \Re c_{n/2} \end{aligned} \tag{1.66}$$

The imaginary part of the result is stored like

$$\begin{aligned} \mathtt{a[n/2+1]} &= \Im c_1 \\ \mathtt{a[n/2+2]} &= \Im c_2 \\ \mathtt{a[n/2+3]} &= \Im c_3 \\ &\dots \\ \mathtt{a[n-1]} &= \Im c_{n/2-1} \end{aligned} \tag{1.67}$$

except for the Hartley transform based R2CFT, which uses the reversed order for the imaginary part

$$\begin{aligned} \mathtt{a[n/2+1]} &= \Im c_{n/2-1} \\ \mathtt{a[n/2+2]} &= \Im c_{n/2-2} \\ \mathtt{a[n/2+3]} &= \Im c_{n/2-3} \\ &\dots \\ \mathtt{a[n-1]} &= \Im c_1 \end{aligned} \tag{1.68}$$

Note the absence of the elements $\Im c_0$ and $\Im c_{n/2}$ which are zero.

1.8.1 Real valued FT via wrapper routines

A simple way to use a complex length-$n/2$ FFT for a real length-n FFT (n even) is to use some post- and preprocessing routines. 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. This is not the most elegant real FFT available, but it is directly usable to turn complex FFTs of any (even) length into a real-valued FFT.

Here is the

[8] cf. relation 1.20

Code 1.18 (R2CFT with wrap routines) *C++ code for a real to complex FFT (R2CFT):*

```
void
wrap_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
//
// ordering of output:
// f[0]     = re[0]   (DC part, purely real)
// f[1]     = re[n/2] (nyquist freq, purely real)
// f[2]     = re[1]
// f[3]     = im[1]
// f[4]     = re[2]
// f[5]     = im[2]
//          ...
// f[2*i]   = re[i]
// f[2*i+1] = im[i]
//          ...
// f[n-2]   = re[n/2-1]
// f[n-1]   = im[n/2-1]
//
// equivalent:
// { fht_real_complex_fft(f, ldn, is); evenodd_permute(f, n); }
//
{
    if ( ldn==0 )  return;
    fht_fft((Complex *)f, ldn-1, +1);
    const ulong n = 1<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]
        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]
        double f1r, f2i;
        sumdiff05(f[i3], f[i1], f1r, f2i);
        double f2r, f1i;
        sumdiff05(f[i2], f[i4], f2r, f1i);
        double c, s;
        double phi = i*phi0;
        sincos(phi, &s, &c);
        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);
        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);

        // f[i4] = is * (ti + f1i);  // im hi
        // f[i2] = is * (ti - f1i);  // im low
        // =^=
        if ( is>0 )  sumdiff( ti,  f1i, f[i4], f[i2]);
        else         sumdiff(-ti,  f1i, f[i2], f[i4]);
    }
    sumdiff(f[0], f[1]);
    if ( n>=4 )  f[nh+1] *= is;
}
```

Code 1.19 (C2RFT, with wrap routines) *C++ code for a complex to real FFT (C2RFT):*

```
void
wrap_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
//
// inverse of wrap_real_complex_fft()
//
// ordering of input:
// like the output of wrap_real_complex_fft()
{
    if ( ldn==0 )  return;
    const ulong n = 1<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = -M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]
        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]
        double f1r, f2i;
        // double f1r =  f[i1] + f[i3];  // re symm
        // double f2i =  f[i1] - f[i3];  // re asymm
        // =^=
        sumdiff(f[i1], f[i3], f1r, f2i);
        double f2r, f1i;
        // double f2r = -f[i2] - f[i4];  // im symm
        // double f1i =  f[i2] - f[i4];  // im asymm
        // =^=
        sumdiff(-f[i4], f[i2], f1i, f2r);
        double c, s;
        double phi = i*phi0;
        sincos(phi, &s, &c);
        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);
        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);
        // f[i2] = ti - f1i;  // im low
        // f[i4] = ti + f1i;  // im hi
        // =^=
        sumdiff(ti, f1i, f[i4], f[i2]);
    }
    sumdiff(f[0], f[1]);
    if ( n>=4 )  { f[nh] *= 2.0; f[nh+1] *= 2.0; }
    fht_fft((Complex *)f, ldn-1, -1);
    if ( is<0 )  reverse_nh(f, n);
}
```

[FXT: `wrap_real_complex_fft` in file `realfft/realfftwrap.cc`]

[FXT: `wrap_complex_real_fft` in file `realfft/realfftwrap.cc`]

1.9 The matrix algorithm (MFA)

The matrix Fourier algorithm[9] (MFA) works for (composite) data lengths $n = R\,C$. Consider the input array as a $R \times C$-matrix (R rows, C columns).

Idea 1.7 (matrix Fourier algorithm) *The matrix Fourier algorithm (MFA) for the FFT:*

1. *Apply a (length R) FFT on each column.*
2. *Multiply each matrix element (index r, c) by $\exp(\pm 2\,\pi\,i\,r\,c/n)$ (sign is that of the transform).*
3. *Apply a (length C) FFT on each row.*
4. *Transpose the matrix.*

[9] A variant of the MFA is called 'four step FFT' in [127].

Note the elegance!

It is trivial to rewrite the MFA as the

Idea 1.8 (transposed matrix Fourier algorithm) *The transposed matrix Fourier algorithm (TMFA) for the FFT:*

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

FFT algorithms are usually very memory nonlocal, i.e. the data is accessed in strides with large skips (as opposed to e.g. in unit strides). In radix 2 (or 2^n) algorithms one even has skips of powers of 2, which is particularly bad on computer systems that use *direct mapped cache* memory: One piece of cache memory is responsible for caching addresses that lie apart by some power of 2. With an 'usual' FFT algorithm one gets 100% cache misses and therefore a memory performance that corresponds to the access time of the main memory, which is very long compared to the clock of modern CPUs. The matrix Fourier algorithm has a much better memory locality (cf. [127]), because the work is done in the short FFTs over the rows and columns.

For the reason given above the computation of the column FFTs should not be done in place. One can insert additional transpositions in the algorithm to have the columns lie in contiguous memory when they are worked upon. The easy way is to use an additional scratch space for the column FFTs, then only the copying from 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. Moreover, one can insert small offsets (a few unused memory words) at the end of each row in order to avoid the cache miss problem almost completely. Then one should also program a procedure that does a 'mass production' variant of the column FFTs, i.e. for doing computation for all rows at once.

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 can 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 where CPU clock is much higher than memory clock the performance may increase drastically, a performance factor of two (even when compared to else very good optimised FFTs) can be observed.

1.10 Convolutions

The cyclic convolution of two sequences a and b is defined as the sequence h with elements h_τ as follows:

$$\begin{aligned} h &= a \circledast b \qquad (1.69) \\ h_\tau &:= \sum_{x+y\equiv\tau \pmod n} a_x\, b_y \end{aligned}$$

The last equation may be rewritten as

$$h_\tau := \sum_{x=0}^{n-1} a_x\, b_{\tau-x} \qquad (1.70)$$

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

Code 1.20 (cyclic convolution by definition) *Compute the cyclic convolution of* `a[]` *with* `b[]` *using the definition, result is returned in* `c[]`

```
procedure convolution(a[],b[],c[],n)
{
    for tau:=0 to n-1
    {
        s := 0
        for x:=0 to n-1
        {
            tx := tau-x
            if tx<0 then  tx := tx+n
            s := s + a[x]*b[tx]
        }
        c[tau] := s
    }
}
```

This procedure uses (for length-n sequences a, b) proportional n^2 operations, therefore it is slow for large values of n. The Fourier transform provides us with a more efficient way to compute convolutions that only uses proportional $n \log(n)$ operations. First we have to establish the convolution property of the Fourier transform:

$$\mathcal{F}[a \circledast b] \quad = \quad \mathcal{F}[a]\,\mathcal{F}[b] \tag{1.71}$$

i.e. convolution in original space is ordinary (elementwise) multiplication in Fourier space.

Here is the proof:

$$\begin{aligned}
\mathcal{F}[a]_k\,\mathcal{F}[b]_k \quad &= \quad \sum_x a_x\, z^{k\,x} \sum_y b_y\, z^{k\,y} \\
&\qquad \text{with} \quad y := \tau - x \\
&= \quad \sum_x a_x\, z^{k\,x} \sum_{\tau-x} b_{\tau-x}\, z^{k\,(\tau-x)} \\
&= \quad \sum_x \sum_{\tau-x} a_x\, z^{k\,x} b_{\tau-x}\, z^{k\,(\tau-x)} \\
&= \quad \sum_\tau \left(\sum_x a_x\, b_{\tau-x} \right) z^{k\,\tau} \\
&= \quad \left(\mathcal{F}\left[\sum_x a_x\, b_{\tau-x} \right] \right)_k \\
&= \quad \left(\mathcal{F}[a \circledast b] \right)_k
\end{aligned} \tag{1.72}$$

Rewriting formula 1.71 as

$$a \circledast b \quad = \quad \mathcal{F}^{-1}\left[\mathcal{F}[a]\,\mathcal{F}[b]\right] \tag{1.73}$$

tells us how to proceed:

Code 1.21 (cyclic convolution via FFT) *Pseudo code for the cyclic convolution of two complex valued sequences* `x[]` *and* `y[]`*, result is returned in* `y[]`*:*

```
procedure fht_cyclic_convolution(x[],y[],n)
{
    complex x[0..n-1], y[0..n-1]
    // transform data:
    fft(x[],n,+1)
    fft(y[],n,+1)
    // convolution in transformed domain:
    for i:=0 to n-1
    {
        y[i] := y[i] * x[i]
    }
    // transform back:
    fft(y[],n,-1)
    // normalise:
    for i:=0 to n-1
    {
        y[i] := y[i]/n
    }
}
```

It is assumed that the procedure `fft()` does no normalisation. In the normalisation loop you precompute `1.0/n` and multiply as divisions are much slower than multiplications.

real convolutions: ... [FXT: `fht_fft_convolution` in file `fft/fftcnvl.cc`] [FXT: `split_radix_fft_convolution` in file `fft/fftcnvl.cc`]

Auto (or self) convolution is defined as

$$\begin{aligned} h &= a \circledast a \qquad (1.74) \\ h_\tau &:= \sum_{x+y\equiv\tau(n)} a_x\, a_y \end{aligned}$$

The corresponding procedure should be obvious. [FXT: `fht_convolution` in file `fht/fhtcnvl.cc`] [FXT: `fht_convolution0` in file `fht/fhtcnvl.cc`]

In the definition of the cyclic convolution (1.69) one can distinguish between those summands where the $x+y$ 'wrapped around' (i.e. $x+y = n+\tau$) and those where simply $x+y=\tau$ holds. These are (following the notation in [77]) denoted by $h^{(1)}$ and $h^{(0)}$ respectively. Then

$$h = h^{(0)} + h^{(1)} \qquad (1.75)$$

where

$$\begin{aligned} h^{(0)} &= \sum_{x\le\tau} a_x\, b_{\tau-x} \\ h^{(1)} &= \sum_{x>\tau} a_x\, b_{n+\tau-x} \end{aligned}$$

There is a simple way to seperate $h^{(0)}$ and $h^{(1)}$ as the left and right half of a length-$2\,n$ sequence. This is just what the *acyclic* (or *linear*) convolution does: Acyclic convolution of two (length-n) sequences a and b can be defined as that length-$2\,n$ sequence h which is the cyclic convolution of the *zero padded* sequences A and B:

$$A := \{a_0, a_1, a_2, ..., a_{n-1}, 0, 0, ..., 0\} \qquad (1.76)$$

Same for B. Then

$$h_\tau := \sum_{x=0}^{2\,n-1} A_x\, B_{\tau-x} \qquad \tau = 0, 1, 2, ..., 2\,n-1 \qquad (1.77)$$

$$\sum_{\substack{x+y\equiv\tau(2n)\\ x,y<2n}} a_x\, b_y = \sum_{0\le x<n} a_x\, b_y + \sum_{n\le x<2n} a_x\, b_y \qquad (1.78)$$

where the right sum is zero because $a_x = 0$ for $n \le x < 2n$. Now

$$\sum_{0\le x<n} a_x\, b_y = \sum_{x\le\tau} a_x\, b_{\tau-x} + \sum_{x>\tau} a_x\, b_{2n+\tau-x} =: R_\tau + S_\tau \qquad (1.79)$$

where the rhs. sums are silently understood as restricted to $0 \le x < n$.

For $0 \le \tau < n$ the sum S_τ is always zero because $b_{2n+\tau-x}$ is zero ($n \le 2n+\tau-x < 2n$ for $0 \le \tau - x < n$); the sum R_τ is already equal to $h^{(0)}_\tau$. For $n \le \tau < 2n$ the sum S_τ is again zero, this time because it extends over nothing (simultaneous conditions $x < n$ and $x > \tau \ge n$); R_τ can be identified with $h^{(1)}_{\tau'}$ ($0 \le \tau' < n$) by setting $\tau = n + \tau'$.

As an illustration consider the convolution of the sequence $\{1, 1, 1, 1\}$ with itself: its linear self convolution is $\{1, 2, 3, 4, 3, 2, 1, 0\}$, its cyclic self convolution is $\{4, 4, 4, 4\}$, i.e. the right half of the linear convolution elementwise added to the left half.

By the way, relation 1.71 is also true for the more general z-transform, but there is no (simple) back-transform, so we cannot turn (the analogue of 1.73)

$$a \circledast b \quad = \quad \mathcal{Z}^{-1}\left[\mathcal{Z}\left[a\right] \mathcal{Z}\left[b\right]\right] \tag{1.80}$$

into a practical algorithm.

1.11 Mass storage convolution using the MFA

The matrix Fourier 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[10].

In convolution computations it is straightforward to save the transpositions by using the MFA followed by the TMFA. (The data is assumed to be in memory as row_0, row_1, ..., row_{R-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 considered here:

Idea 1.9 (matrix convolution algorithm) *The matrix convolution algorithm:*

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

Note that steps 3, 4 and 5 constitute a length-C convolution.

[FXT: `matrix_convolution` in file `matrix/matrixcnvl.cc`] [FXT: `matrix_convolution0` in file `matrix/matrixcnvl.cc`] [FXT: `matrix_auto_convolution` in file `matrix/matrixcnvla.cc`] [FXT: `matrix_auto_convolution0` in file `matrix/matrixcnvla.cc`]

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 RAM workspace. An important consideration is the

[10] 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

Minimisation of the number of disk seeks

The number of disk seeks has to be kept minimal because these are slow operations which, if occur too often, degrade performance unacceptably.

The crucial modification of the use of the MFA is *not* to choose R and C as close as possible to $\sqrt{n}$ as usually done. Instead one chooses R minimal, i.e. the row length C corresponds to the biggest data set that fits into the RAM memory[11]. We now analyse how the number of seeks depends on the choice of R and C: in what follows it is assumed that the data lies in memory as row_0, row_1, ..., row_{R-1}, i.e. the way array data is stored in the C language, as opposed to the Fortran language convention. Further let $\alpha \geq 2$ be the number of times the data set exceeds the RAM size.

In step 1 and 3 of algorithm 1.14 one reads from disk (row by row, involving R seeks) the number of colums that just fit into RAM, does the (many, short) column-FFTs[12], writes back (again R seeks) and proceeds to the next block; this happens for α of these blocks, giving a total of $4\,\alpha\,R$ seeks for steps 1 and 3.

In step 2 one has to read (α times) blocks of one or more rows, which lie in contiguous portions of the disk, perform the FFT on the rows and write back to disk, leading to a total of $2\,\alpha$ seeks.

Thereby one has a number of $2\,\alpha + 4\,\alpha\,R$ seeks during the whole computation, which is minimised by the choice of maximal C. This means that one chooses a shape of the matrix so that the rows are as big as possible subject to the constraint that they have to fit into main memory, which in turn means there are $R = \alpha$ rows, leading to an optimal seek count of $K = 2\,\alpha + 4\,\alpha^2$.

If one seek takes 10 milliseconds then one has for $\alpha = 16$ (probably quite a big FFT) a total of $K \cdot 10 = 1056 \cdot 10$ milliseconds or approximately 10 seconds. With a RAM workspace of 64 Megabytes[13] 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) $6 \cdot 1024MB/(10MB/sec) \approx 600sec$ or approximately 10 minutes.

With a multithreading OS one may want to produce a 'double buffer' variant: choose the row length so that it fits twice into the RAM workspace; 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 keep the CPU busy during much of the hard disk operations.

The remarks about the computation of the column FFTs on page 25 also apply here.

1.12 Weighted Fourier transforms

Let us define a new kind of transform by slightly modifying the definition of the FT (cf. formula 1.1):

$$\begin{aligned} c &= \mathcal{W}_v\,[a] && (1.81) \\ c_k &:= \sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k} \qquad v_x \neq 0 \quad \forall x \end{aligned}$$

where $z := e^{\pm 2\,\pi\,i/n}$. The sequence c shall be called weighted (discrete) transform of the sequence a with the weight (sequence) v. Note the v_x that entered: the weighted transform with $v_x = \frac{1}{\sqrt{n}}\ \forall x$ is just the usual Fourier transform. The inverse transform is

$$a = \mathcal{W}_v^{-1}\,[c] \qquad (1.82)$$

[11] more precisely: the amount of RAM where no swapping will occur, some programs plus the operating system have to be there, too.

[12] real-complex FFTs in step 1 and complex-real FFTs in step 3.

[13] allowing for 8 million 8 byte floats, so the total FFT size is $S = 16 \cdot 64 = 1024\,\text{MB}$ or 32 million floats

$$a_x = \frac{1}{n\,v_x}\sum_{k=0}^{n-1} c_k\, z^{-x\,k}$$

This can be easily seen:

$$\begin{aligned}\mathcal{W}_v^{-1}\left[\mathcal{W}_v\left[a\right]\right]_y &= \frac{1}{n\,v_y}\sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k}\, z^{-y\,k} \\ &= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\,\frac{1}{v_y}\,a_x\, z^{x\,k}\, z^{-y\,k} \\ &= \frac{1}{n}\sum_{x=0}^{n-1} v_x\,\frac{1}{v_y}\,a_x\,\delta_{x,y}\, n \\ &= a_y\end{aligned}$$

(cf. section 1.1). That $\mathcal{W}_v\left[\mathcal{W}_v^{-1}\left[a\right]\right]$ is also identity is apparent from the definitions.

Given an implemented FFT it is trivial to set up a weighted Fourier transform:

Code 1.22 (weighted transform) *Pseudo code for the discrete weighted Fourier transform*

```
procedure weighted_ft(a[], v[], n, is)
{
    for x:=0 to n-1
    {
        a[x] := a[x] * v[x]
    }
    fft(a[],n,is)
}
```

Inverse weighted transform is also easy:

Code 1.23 (inverse weighted transform) *Pseudo code for the inverse discrete weighted Fourier transform*

```
procedure inverse_weighted_ft(a[], v[], n, is)
{
    fft(a[],n,is)
    for x:=0 to n-1
    {
        a[x] := a[x] / v[x]
    }
}
```

`is` must be negative wrt. the forward transform.

[FXT: `weighted_fft` in file `weighted/weightedfft.cc`]

[FXT: `weighted_inverse_fft` in file `weighted/weightedfft.cc`]

Introducing a *weighted (cyclic) convolution* h_v by

$$\begin{aligned}h_v &= a \circledast_{\{v\}} b \\ &= \mathcal{W}_v^{-1}\left[\mathcal{W}_v\left[a\right]\,\mathcal{W}_v\left[b\right]\right]\end{aligned} \tag{1.83}$$

(cf. formula 1.73)

Then for the special case $v_x = V^x$ one has

$$h_v = h^{(0)} + V^n\, h^{(1)} \tag{1.84}$$

($h^{(0)}$ and $h^{(1)}$ were defined by formula 1.75). It is not hard to see why: Up to the final division by the weight sequence, the weighted convolution is just the cyclic convolution of the two weighted sequences, which is for the element with index τ equal to

$$\sum_{x+y\equiv\tau (\mathrm{mod}\ n)} (a_x\, V^x)\,(b_y\, V^y) \quad = \quad \sum_{x\leq\tau} a_x\, b_{\tau-x}\, V^\tau + \sum_{x>\tau} a_x\, b_{n+\tau-x}\, V^{n+\tau} \tag{1.85}$$

Final division of this element (by V^τ) gives $h^{(0)} + V^n\, h^{(1)}$ as stated.

The cases when V^n is some root of unity are particularly interesting: For $V^n = \pm i = \pm\sqrt{-1}$ one gets the so called *right-angle convolution*:

$$h_v \quad = \quad h^{(0)} \mp i\, h^{(1)} \tag{1.86}$$

This gives a nice possibility to directly use complex FFTs for the computation of a linear (acycclic) convolution of two real sequences: for length-n sequences the elements of the linear convolution with indices $0, 1, ..., n-1$ are then found in the real part of the result, the elements $n, n+1, ..., 2\,n-1$ are the imaginary part. Choosing $V^n = -1$ leads to the *negacyclic convolution* (or skew circular convolution):

$$h_v \quad = \quad h^{(0)} - h^{(1)} \tag{1.87}$$

Cyclic, negacyclic and right-angle convolution can be understood as a polynomial product modulo $z^n - 1$, $z^n + 1$ and $z^n \pm i$, respectively (cf. [3]).

[FXT: `weighted_complex_auto_convolution` in file `weighted/weightedconv.cc`]

[FXT: `negacyclic_complex_auto_convolution` in file `weighted/weightedconv.cc`]

[FXT: `right_angle_complex_auto_convolution` in file `weighted/weightedconv.cc`]

1.13 Half cyclic convolution for half the price ?

The computation of $h^{(0)}$ from formula 1.75 (without computing $h^{(1)}$) is called *half cyclic convolution*. Clearly, one asks for less information than one gets from the acyclic convolution. One might hope to find an algorithm that computes $h^{(0)}$ and uses only half the memory compared to the linear convolution or that needs half the work, possibly both. It may be a surprise that no such algorithm seems to be known currently[14].

Here is a clumsy attempt to find $h^{(0)}$ alone: Use the weighted transform with the weight sequence $v_x = V^x$ where V^n is very small. Then $h^{(1)}$ will in the result be multiplied with a small number and we hope to make it almost disappear. Indeed, using $V^n = 1000$ for the cyclic self convolution of the sequence $\{1, 1, 1, 1\}$ (where for the linear self convolution $h^{(0)} = \{1, 2, 3, 4\}$ and $h^{(1)} = \{3, 2, 1, 0\}$) one gets $\{1.003, 2.002, 3.001, 4.000\}$. At least for integer sequences one could choose V^n (more than two times) bigger than biggest possible value in $h^{(1)}$ and use rounding to nearest integer to isolate $h^{(0)}$. Alas, even for modest sized arrays numerical overflow and underflow gives spurious results. Careful analysis shows that this idea leads to an algorithm far worse than simply using linear convolution.

1.14 Convolution using the MFA

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm (section 1.9):

1. Apply a FFT on each column.

[14] If you know one, tell me about it!

2. On each row apply the weighted convolution with $V^C = e^{2\,\pi\,i\,r/R} = 1^{r/R}$ where R is the total number of rows, $r = 0..R-1$ the index of the row, C the length of each row (or, equivalently the total number columns)
3. Apply a FFT on each column (of the transposed matrix).

First consider

The case $R = 2$

The cyclic auto convolution of the sequence x can be obtained by two half length convolutions (one cyclic, one negacyclic) of the sequences[15] $s := x^{(0/2)} + x^{(1/2)}$ and $d := x^{(0/2)} - x^{(1/2)}$ using the formula

$$x \circledast x \quad = \quad \frac{1}{2} \left\{ s \circledast s + d \circledast_{-} d, \quad s \circledast s - d \circledast_{-} d \right\} \tag{1.88}$$

The equivalent formula for the cyclic convolution of two sequences x and y is

$$x \circledast y \quad = \quad \frac{1}{2} \left\{ s_x \circledast s_y + d_x \circledast_{-} d_y, \quad s_x \circledast s_y - d_x \circledast_{-} d_y \right\} \tag{1.89}$$

where

$$\begin{aligned} s_x &:= x^{(0/2)} + x^{(1/2)} \\ d_x &:= x^{(0/2)} - x^{(1/2)} \\ s_y &:= y^{(0/2)} + y^{(1/2)} \\ d_y &:= y^{(0/2)} - y^{(1/2)} \end{aligned}$$

For the acyclic (or linear) convolution of sequences one can use the cyclic convolution of the zero padded sequences $z_x := \{x_0, x_1, ..., n_{n-1}, 0, 0, ..., 0\}$ (i.e. x with n zeros appended). Using formula 1.88 one gets for the two sequences x and y (with $s_x = d_x = x$, $s_y = d_y = y$):

$$x \circledast_{ac} y \quad = \quad z_x \circledast z_y \quad = \quad \frac{1}{2} \left\{ x \circledast y + x \circledast_{-} y, \quad x \circledast y - x \circledast_{-} y \right\} \tag{1.90}$$

And for the acyclic auto convolution:

$$x \circledast_{ac} x \quad = \quad z \circledast z \quad = \quad \frac{1}{2} \left\{ x \circledast x + x \circledast_{-} x, \quad x \circledast x - x \circledast_{-} x \right\} \tag{1.91}$$

The case $R = 3$

Let $\omega = \frac{1}{2}\,(1 + \sqrt{3})$ and define

$$\begin{aligned} A &:= x^{(0/3)} + x^{(1/3)} + x^{(2/3)} \\ B &:= x^{(0/3)} + \omega\, x^{(1/3)} + \omega^2\, x^{(2/3)} \\ C &:= x^{(0/3)} + \omega^2\, x^{(1/3)} + \omega\, x^{(2/3)} \end{aligned}$$

Then, if $h := x \circledast_{ac} x$, there is

$$\begin{aligned} x^{(0/3)} &= A \circledast A + B \circledast_{\{\omega\}} B + C \circledast_{\{\omega^2\}} C \\ x^{(1/3)} &= A \circledast A + \omega^2\, (B \circledast_{\{\omega\}} B) + \omega\, (C \circledast_{\{\omega^2\}} C) \\ x^{(2/3)} &= A \circledast A + \omega\, (B \circledast_{\{\omega\}} B) + \omega^2\, (C \circledast_{\{\omega^2\}} C) \end{aligned} \tag{1.92}$$

For real valued data C is the complex conjugate of B and (with $\omega^2 = cc.\omega$) $B \circledast_{\{\omega\}} B$ is the cc. of $C \circledast_{\{\omega^2\}} C$ and therefore every $B \circledast_{\{\}} B$-term is the cc. of the $C \circledast_{\{\}} C$-term in the same line. Is there a nice and general scheme for real valued convolutions based on the MFA? Read on for the positive answer.

[15] s, d lower half plus/minus higher half of x

1.15 Convolution of real valued data using the MFA

For row 0 (which is real after the column FFTs) one needs to compute the (usual) cyclic convolution; for row $R/2$ (also real after the column FFTs) a negacyclic convolution is needed[16], the pseudo code for that task is given on page 56.

All other weighted convolutions involve complex computations, but it is easy to see how cut the work by 50 percent: As the result must be real the data in row number $R - r$ must, because of the symmetries of the real and imaginary part of the (inverse) Fourier transform of real data, be the complex conjugate of the data in row r. Therefore one can use real FFTs (R2CFTs) for all column-transforms for step 1 and half-complex to real FFTs (C2RFTs) for step 3.

Let the computational cost of a cyclic (real) convolution be q, then

For R even one must perform 1 cyclic (row 0), 1 negacyclic (row $R/2$) and $R/2 - 2$ complex (weighted) convolutions (rows $1, 2, ..., R/2 - 1$)

For R odd one must perform 1 cyclic (row 0) and $(R - 1)/2$ complex (weighted) convolutions (rows $1, 2, ..., (R - 1)/2$)

Now assume, slightly simplifying, that the cyclic and the negacyclic real convolution involve the same number of computations and that the cost of a weighted complex convolution is twice as high. Then in both cases above the total work is exactly half of that for the complex case, which is about what one would expect from a real world real valued convolution algorithm.

For acyclic convolution one may want to use the right angle convolution (and complex FFTs in the column passes).

1.16 Convolution without transposition using MFA

An algorithm for convolution using the MFA that uses revbin_permute instead of transpose (works for sizes that are a power of two, generalizes for sizes a power of some prime):

```
rows=8  columns=4
input data (symbolic format: R00C):
  0:        0     1     2     3
  1:     1000  1001  1002  1003
  2:     2000  2001  2002  2003
  3:     3000  3001  3002  3003
  4:     4000  4001  4002  4003
  5:     5000  5001  5002  5003
  6:     6000  6001  6002  6003
  7:     7000  7001  7002  7003

FULL REVBIN_PERMUTE for transposition:
  0:        0  4000  2000  6000  1000  5000  3000  7000
  1:        2  4002  2002  6002  1002  5002  3002  7002
  2:        1  4001  2001  6001  1001  5001  3001  7001
  3:        3  4003  2003  6003  1003  5003  3003  7003

DIT FFTs on revbin_permuted rows (in revbin_permuted sequence), i.e. unrevbin_permute rows:
 (apply weight after each FFT)
  0:        0  1000  2000  3000  4000  5000  6000  7000
  1:        2  1002  2002  3002  4002  5002  6002  7002
  2:        1  1001  2001  3001  4001  5001  6001  7001
  3:        3  1003  2003  3003  4003  5003  6003  7003
```

[16] For R odd there is no such row and no negacyclic convolution is needed.

```
FULL REVBIN_PERMUTE for transposition:
  0:       0     1     2     3
  1:    4000  4001  4002  4003
  2:    2000  2001  2002  2003
  3:    6000  6001  6002  6003
  4:    1000  1001  1002  1003
  5:    5000  5001  5002  5003
  6:    3000  3001  3002  3003
  7:    7000  7001  7002  7003

CONVOLUTIONS on rows (don't care revbin_permuted sequence), no reordering.

FULL REVBIN_PERMUTE for transposition:
  0:       0  1000  2000  3000  4000  5000  6000  7000
  1:       2  1002  2002  3002  4002  5002  6002  7002
  2:       1  1001  2001  3001  4001  5001  6001  7001
  3:       3  1003  2003  3003  4003  5003  6003  7003

 (apply inverse weight before each FFT)
DIF FFTs on rows (in revbin_permuted sequence), i.e. revbin_permute rows:
  0:       0  4000  2000  6000  1000  5000  3000  7000
  1:       2  4002  2002  6002  1002  5002  3002  7002
  2:       1  4001  2001  6001  1001  5001  3001  7001
  3:       3  4003  2003  6003  1003  5003  3003  7003

FULL REVBIN_PERMUTE for transposition:
  0:       0     1     2     3
  1:    1000  1001  1002  1003
  2:    2000  2001  2002  2003
  3:    3000  3001  3002  3003
  4:    4000  4001  4002  4003
  5:    5000  5001  5002  5003
  6:    6000  6001  6002  6003
  7:    7000  7001  7002  7003
```

1.17 Split radix Fourier transforms (SRFT)

Code 1.24 (split radix DIF FFT) *Pseudo code for the split radix DIF algorithm,* is *must be -1 or +1:*

```
procedure fft_splitradix_dif(x[],y[],ldn,is)
{
    n := 2**ldn
    if n<=1  return
    n2 := 2*n
    for k:=1 to ldn
    {
        n2 := n2 / 2
        n4 := n2 / 4
        e := 2 * PI / n2
        for j:=0 to n4-1
        {
            a := j * e
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

            ix := j
            id := 2*n2

            while ix<n-1
            {
                i0 := ix
                while i0 < n
                {
                    i1 := i0 + n4
                    i2 := i1 + n4
                    i3 := i2 + n4

                    {x[i0], r1} := {x[i0] + x[i2], x[i0] - x[i2]}
                    {x[i1], r2} := {x[i1] + x[i3], x[i1] - x[i3]}

                    {y[i0], s1} := {y[i0] + y[i2], y[i0] - y[i2]}
                    {y[i1], s2} := {y[i1] + y[i3], y[i1] - y[i3]}

                    {r1, s3} := {r1+s2, r1-s2}
                    {r2, s2} := {r2+s1, r2-s1}

                    // complex mult: (x[i2],y[i2]) := -(s2,r1) * (ss1,cc1)
                    x[i2] :=  r1*cc1 - s2*ss1
                    y[i2] := -s2*cc1 - r1*ss1

                    // complex mult: (y[i3],x[i3]) := (r2,s3) * (cc3,ss3)
                    x[i3] :=  s3*cc3 + r2*ss3
                    y[i3] :=  r2*cc3 - s3*ss3

                    i0 := i0 + id
                }

                ix := 2 * id - n2 + j
                id := 4 * id
            }
        }
    }

    ix := 1
    id := 4

    while ix<n
    {
        for i0:=ix-1 to n-id step id
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            {y[i0], y[i1]} := {y[i0]+y[i1], y[i0]-y[i1]}
        }

        ix := 2 * id - 1
        id := 4 * id
    }

    revbin_permute(x[],n)
    revbin_permute(y[],n)

    if is>0
    {
        for j:=1 to n/2-1
        {
            swap(x[j],x[n-j])
            swap(y[j],y[n-j])
        }
    }
}
```

[FXT: split_radix_fft in file fft/fftsplitradix.cc] [FXT: split_radix_fft in file fft/cfftsplitradix.cc]

1.17.1 Real to complex SRFT

Code 1.25 (split radix R2CFT) *Pseudo code for the split radix R2CFT algorithm*

```
procedure r2cft_splitradix_dit(x[],ldn)
{
    n := 2**ldn

    ix := 1;
    id := 4;
```

```
do
{
    i0 := ix-1
    while i0<n
    {
        i1 := i0 + 1

        {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}

        i0 := i0 + id
    }

    ix := 2*id-1
    id := 4 * id
}
while ix<n

n2 := 2
nn := n/4
while nn!=0
{
    ix := 0
    n2 := 2*n2
    id := 2*n2
    n4 := n2/4
    n8 := n2/8

    do  // ix loop
    {
        i0 := ix
        while i0<n
        {
            i1 := i0
            i2 := i1 + n4
            i3 := i2 + n4
            i4 := i3 + n4

            {t1, x[i4]} := {x[i4]+x[i3], x[i4]-x[i3]}

            {x[i1], x[i3]} := {x[i1]+t1, x[i1]-t1}

            if  n4!=1
            {
                i1 := i1 + n8
                i2 := i2 + n8
                i3 := i3 + n8
                i4 := i4 + n8

                t1 := (x[i3]+x[i4]) * sqrt(1/2)
                t2 := (x[i3]-x[i4]) * sqrt(1/2)

                {x[i4], x[i3]} := {x[i2]-t1, -x[i2]-t1}
                {x[i1], x[i2]} := {x[i1]+t2,  x[i1]-t2}
            }

            i0 := i0 + id
        }

        ix := 2*id - n2
        id := 2*id
    }
    while ix<n

    e := 2.0*PI/n2
    a := e

    for j:=2 to n8
    {
        cc1 := cos(a)
        ss1 := sin(a)
        cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
        ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

        a := j*e

        ix := 0
        id := 2*n2

        do  // ix-loop
        {
            i0 := ix
            while  i0<n
            {
                i1 := i0 + j - 1
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                i5 := i0 + n4 - j + 1
                i6 := i5 + n4
                i7 := i6 + n4
                i8 := i7 + n4
```

```
                    // complex mult: (t2,t1) := (x[i7],x[i3]) * (cc1,ss1)
                    t1 := x[i3]*cc1 + x[i7]*ss1
                    t2 := x[i7]*cc1 - x[i3]*ss1

                    // complex mult: (t4,t3) := (x[i8],x[i4]) * (cc3,ss3)
                    t3 := x[i4]*cc3 + x[i8]*ss3
                    t4 := x[i8]*cc3 - x[i4]*ss3

                    t5 := t1 + t3
                    t6 := t2 + t4
                    t3 := t1 - t3
                    t4 := t2 - t4

                    {t2, x[i3]} := {t6+x[i6], t6-x[i6]}

                    x[i8] := t2

                    {t2,x[i7]} := {x[i2]-t3, -x[i2]-t3}

                    x[i4] := t2

                    {t1, x[i6]} := {x[i1]+t5, x[i1]-t5}

                    x[i1] := t1

                    {t1, x[i5]} := {x[i5]+t4, x[i5]-t4}

                    x[i2] := t1

                    i0 := i0 + id
                }

                ix := 2*id - n2
                id := 2*id

            }
            while ix<n
        }
        nn := nn/2
    }
}
```

1.17.2 Complex to real SRFT

Code 1.26 (split radix C2RFT) *Pseudo code for the split radix C2RFT algorithm*

```
procedure c2rft_splitradix_dif(x[],ldn)
{
    n := 2**ldn

    n2 := n/2
    nn := n/4
    while nn!=0
    {
        ix := 0
        id := n2
        n2 := n2/2
        n4 := n2/4
        n8 := n2/8

        do  // ix loop
        {
            i0 := ix
            while i0<n
            {
                i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                {x[i1], t1} := {x[i1]+x[i3], x[i1]-x[i3]}

                x[i2] := 2*x[i2]

                x[i4] := 2*x[i4]
                {x[i3], x[i4]} := {t1+x[i4], t1-x[i4]}

                if  n4!=1
                {
                    i1 := i1 + n8
                    i2 := i2 + n8
                    i3 := i3 + n8
                    i4 := i4 + n8

                    {x[i1], t1} := {x[i2]+x[i1], x[i2]-x[i1]}
                    {t2, x[i2]} := {x[i4]+x[i3], x[i4]-x[i3]}

                    x[i3] := -sqrt(2)*(t2+t1)
                    x[i4] :=  sqrt(2)*(t1-t2)
                }
                i0 := i0 + id
            }
            ix := 2*id - n2
            id := 2*id
        }
        while ix<n

        e := 2.0*PI/n2
        a := e

        for j:=2 to n8
        {
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

            a := j*e

            ix := 0
            id := 2*n2

            do  // ix-loop
            {
                i0 := ix
                while  i0<n
                {
                    i1 := i0 + j - 1
                    i2 := i1 + n4
                    i3 := i2 + n4
                    i4 := i3 + n4

                    i5 := i0 + n4 - j + 1
                    i6 := i5 + n4
                    i7 := i6 + n4
                    i8 := i7 + n4

                    {x[i1], t1} := {x[i1]+x[i6], x[i1]-x[i6]}

                    {x[i5], t2} := {x[i5]+x[i2], x[i5]-x[i2]}

                    {t3, x[i6]} := {x[i8]+x[i3], x[i8]-x[i3]}

                    {t4, x[i2]} := {x[i4]+x[i7], x[i4]-x[i7]}

                    {t1, t5} := {t1+t4, t1-t4}

                    {t2, t4} := {t2+t3, t2-t3}

                    // complex mult: (x[i7],x[i3]) := (t5,t4)  * (ss1,cc1)
                    x[i3] := t5*cc1 + t4*ss1
                    x[i7] := -t4*cc1 + t5*ss1

                    // complex mult: (x[i4],x[i8]) := (t1,t2)  * (cc3,ss3)
                    x[i4] := t1*cc3 - t2*ss3
                    x[i8] := t2*cc3 + t1*ss3

                    i0 := i0 + id
                }

                ix := 2*id - n2
                id := 2*id
            }
            while ix<n
        }

        nn := nn/2
    }

    ix := 1;
    id := 4;
    do
    {
        i0 := ix-1
        while i0<n
        {
            i1 := i0 + 1

            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}

            i0 := i0 + id
        }

        ix := 2*id-1
        id := 4 * id
    }
    while ix<n
}
```

1.18 Multidimensional FTs

1.18.1 Definition

Let $a_{x,y}$ ($x = 0, 1, 2, .., C$ and $y = 0, 1, 2, ..., R$) be a 2-dimensional array of data[17]. Its 2-dimensional Fourier transform $c_{k,h}$ is defined by:

$$\begin{aligned} c &= \mathcal{F}[a] && (1.93) \\ c_{k,h} &:= \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{x=0}^{R-1} a_{x,y} \, z^{x\,k + y\,h} \qquad \text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = R\,C && (1.94) \end{aligned}$$

Its inverse is

$$\begin{aligned} a &= \mathcal{F}^{-1}[c] && (1.95) \\ a_x &= \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h} \, z^{-(x\,k + y\,h)} && (1.96) \end{aligned}$$

For a m-dimensional array $a_{\vec{x}}$ ($\vec{x} = (x_1, x_2, x_3, ..., x_m)$, $x_i \in 0, 1, 2, ..., S_i$) the m-dimensional Fourier transform $c_{\vec{k}}$ ($\vec{k} = (k_1, k_2, k_3, ..., k_m)$, $k_i \in 0, 1, 2, ..., S_i$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \ldots \sum_{x_m=0}^{S_m-1} a_{\vec{x}} \, z^{\vec{x}.\vec{k}} \qquad (1.97)$$

$$\text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = S_1\, S_2 \ldots Sm$$

The inverse transform is again the one with the minus in the exponent of z.

1.18.2 The row column algorithm

The equation of the definition of the two dimensional FT (1.93) can be recast as

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} z^{x\,k} \sum_{x=0}^{R-1} a_{x,y} \, z^{y\,h} \qquad (1.98)$$

which shows that the 2-dimensional FT can be accomplished by using 1-dimensional FTs to transform first the rows and then the columns[18]. This leads us directly to the row column algorithm:

Code 1.27 (row column FFT) *Compute the two dimensional FT of* `a[][]` *using the row column method*

```
procedure rowcol_ft(a[][],R,C)
{
    complex a[R][C]   // R (length-C) rows, C (length-R) columns

    for r:=0 to R-1   // FFT rows
    {
        fft(a[r][],C,is)
    }

    complex t[R]      // scratch array for columns
    for c:=0 to C-1   // FFT columns
    {
        copy a[0,1,...,R-1][c] to t[]   // get column
        fft(t[],R,is)
        copy t[] to a[0,1,...,R-1][c]   // write back column
    }
}
```

[17] Imagine a $R \times C$ matrix of R rows (of length C) and C columns (of length R).

[18] or first the rows, then the columns, the result is the same

Here it is assumed that the rows lie in contiguous memory (as in the C language).

Transposing the array before the column pass in order to avoid the copying of the columns to extra scratch space will probably do good for the performance. The transposing back before returning can be avoided if a backtransform will follow[19], the backtransform must then be called with `R` and `C` swapped.

[19] as typical for convolution etc.

Chapter 2

The z-transform (ZT)

2.1 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

$$\hat{a}_k \quad := \quad \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{2.1}$$

The z-transform is a linear transformation, its most important property is the convolution property (formula 1.71): Convolution in original space corresponds to ordinary (elementwise) multiplication in z-space. (See [26] and [27].)

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

2.2 The chirp ZT

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

$$x\,k \quad = \quad \frac{1}{2}\left(x^2 + k^2 - (k-x)^2\right) \tag{2.2}$$

$$\hat{f}_k = \sum_{x=0}^{n-1} f_x \, z^{x\,k} \quad = \quad z^{k^2/2} \sum_{x=0}^{n-1} \left(f_x \, z^{x^2/2}\right) \, z^{-(k-x)^2/2} \tag{2.3}$$

This leads to the following

Idea 2.1 (chirp z-transform) *Algorithm for the chirp z-transform:*

1. *Multiply f elementwise with $z^{x^2/2}$.*
2. *Convolve (acyclically) the resulting sequence with the sequence $z^{-x^2/2}$, zero padding of the sequences is required here.*
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.

TEST: ref chirpzt: 2.1, pageref: 41

[1] cf. [3]

2.3 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 $z = e^{\pm 2\,\pi\,i/n}$ 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 $2\,n$.

2.4 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 [128].

[FXT: `fft_fract` in file `chirp/fftfract.cc`]

Chapter 3

Walsh transforms

How to make a Walsh transform out of your FFT:
'*Replace exp(something) by* 1*, done.*'

Very simple, so we are ready for

Code 3.1 (radix 2 DIT Walsh transform, first trial) *Pseudo code for a radix 2 decimation in time Walsh transform: (has a flaw)*

```
procedure walsh_wak_dit2(a[],ldn)
{
    n := 2**ldn
    for ldm := 1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        for j := 0 to mh-1
        {
            for r := 0 to n-1 step m
            {
                t1 := r + j
                t2 := t1 + mh
                u := a[t1]
                v := a[t2]
                a[t1] := u + v
                a[t2] := u - v
            }
        }
    }
}
```

The transform involves proportional $n \log_2(n)$ additions (and subtractions) and no multiplication at all. Note the absence of any `permute(a[],n)` function call. The transform is its own inverse, so there is nothing like the `is` in the FFT procedures here. Let's make a slight improvement: Here we just took the code 1.5 and threw away all trig computations. But the swapping of the inner loops, that caused the nonlocality of the memory access is now of no advantage, so we try this piece of

Code 3.2 (radix 2 DIT Walsh transform) *Pseudo code for a radix 2 decimation in time Walsh transform:*

```
procedure walsh_wak_dit2(a[],ldn)
{
    n := 2**ldn
    for ldm := 1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        for r := 0 to n-1 step m
        {
            t1 = r
            t2 = r + mh
            for j := 0 to mh-1
            {
                u := a[t1]
                v := a[t2]

                a[t1] := u + v
                a[t2] := u - v

                t1 := t1 + 1
                t2 := t2 + 1
            }
        }
    }
}
```

Which performance impact can this innocent change in the code have? For large n it gave a speedup by a factor of more than three when run on a computer with a main memory clock of 66 Megahertz and a 5.5 times higher CPU clock of 366 Megahertz.

The equivalent code for the decimation in frequency algorithm looks like this:

Code 3.3 (radix 2 DIF Walsh transform) *Pseudo code for a radix 2 decimation in frequency Walsh transform:*

```
procedure walsh_wak_dif2(a[],ldn)
{
    n := 2**ldn

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

        for r := 0 to n-1 step m
        {
            t1 = r
            t2 = r + mh
            for j := 0 to mh-1
            {
                u := a[t1]
                v := a[t2]

                a[t1] := u + v
                a[t2] := u - v

                t1 := t1 + 1
                t2 := t2 + 1
            }
        }
    }
}
```

The basis functions look like this (for $n = 16$):

Here is a formula for the Walsh basis functions

$$wak_i(x) \quad := \quad XXX \tag{3.1}$$

A term analogue to the frequency of the Fourier basis functions is the so called 'sequency' of the Walsh functions, the number of the changes of sign of the individual functions. If one wants the basis functions ordered with respect to sequency one can use a procedure like this:

Code 3.4 (sequency ordered Walsh transform (wal))

```
procedure walsh_wal_dif2(a[],n)
{
    gray_permute(a[],n)
    permute(a[],n)
    walsh_wak_dif2(a[],n)
}
```

permute(a[],n) is what it used to be (cf. section 1.7). The procedure gray_permute(a[],n) reorders data element with index m by the element with index gray_code(m).

Code 3.5 (Gray permute)

```
procedure gray_permute(a[],n)
// real a[0..n] input, result
{
    real t[]  // workspace
    for i:=0 to n-1
    {
        g := graycode(i)
        t[g] := a[i]
    }
    copy t[] to a[]
}
```

The graycode reordering can't be (easily) done inplace, therefore the temporary array t[]. The function graycode(i) shall return the graycode of its (integer) argument, i.e. i exor'd with i/2. In C one can write this compactly as

```
inline unsigned long graycode(unsigned long x)  { return x^(x>>1); }
```

The Walsh transform of integer input is integral, cf. section 6.3.

```
 0:     [* * * * * * * * * * * * * * * *] ( 0)    [* * * * * * * * * * * * * * * *] ( 0)
 1:     [*   *   *   *   *   *   *   *  ] (15)    [* * * * * * * *                ] ( 1)
 2:     [* *     * *     * *     * *    ] ( 7)    [* * * *         * * * *        ] ( 3)
 3:     [*     * *     * *     * *     *] ( 8)    [* * * *                 * * * *] ( 2)
 4:     [* * * *         * * * *        ] ( 3)    [* *     * *     * *     * *    ] ( 7)
 5:     [*   *     *   * *   *     *   *] (12)    [* *     * *         * *     * *] ( 6)
 6:     [* *         * * * *         * *] ( 4)    [* *         * * * *         * *] ( 4)
 7:     [*     *   * *   *     *   * *  ] (11)    [* *         * *     * * * *    ] ( 5)
 8:     [* * * * * * * *                ] ( 1)    [*   *   *   *   *   *   *   *  ] (15)
 9:     [*   *   *   *     *   *   *   *] (14)    [*   *   *   *     *   *   *   *] (14)
10:     [* *     * *         * *     * *] ( 6)    [*   *     *   * *   *     *   *] (12)
11:     [*     * *     *   * *     * *  ] ( 9)    [*   *     *   *   *   * *   *  ] (13)
12:     [* * * *                 * * * *] ( 2)    [*     * *     * *     * *     *] ( 8)
13:     [*   *     *   *   *   * *   *  ] (13)    [*     * *     *   * *     * *  ] ( 9)
14:     [* *         * *     * * * *    ] ( 5)    [*     *   * *   *     *   * *  ] (11)
15:     [*     *   * *     * *   *     *] (10)    [*     *   * *     * *   *     *] (10)
```

WAK (Walsh-Kronecker base)

PAL (Walsh-Paley base)

```
 0:     [* * * * * * * * * * * * * * * *] ( 0)    [* * * * * * * * * * * * * * * *] ( 0)
 1:     [* * * * * * * *                ] ( 1)    [* * * *                 * * * *] ( 2)
 2:     [* * * *                 * * * *] ( 2)    [* *         * * * *         * *] ( 4)
 3:     [* * * *         * * * *        ] ( 3)    [* *     * *         * *     * *] ( 6)
 4:     [* *         * * * *         * *] ( 4)    [*     * *     * *     * *     *] ( 8)
 5:     [* *         * *     * * * *    ] ( 5)    [*     *   * *     * *   *     *] (10)
 6:     [* *     * *         * *     * *] ( 6)    [*   *     *   * *   *     *   *] (12)
 7:     [* *     * *     * *     * *    ] ( 7)    [*   *   *   *     *   *   *   *] (14)
 8:     [*     * *     * *     * *     *] ( 8)    [*   *   *   *   *   *   *   *  ] (15)
 9:     [*     * *     *   * *     * *  ] ( 9)    [*   *     *   *   *   * *   *  ] (13)
10:     [*     *   * *     * *   *     *] (10)    [*     *   * *   *     *   * *  ] (11)
11:     [*     *   * *   *     *   * *  ] (11)    [*     * *     *   * *     * *  ] ( 9)
12:     [*   *     *   * *   *     *   *] (12)    [* *     * *     * *     * *    ] ( 7)
13:     [*   *     *   *   *   * *   *  ] (13)    [* *         * *     * * * *    ] ( 5)
14:     [*   *   *   *     *   *   *   *] (14)    [* * * *         * * * *        ] ( 3)
15:     [*   *   *   *   *   *   *   *  ] (15)    [* * * * * * * *                ] ( 1)
```

WAL (Walsh-Kaczmarz base)

CIRCLE

```
 0:     [* * * * * * * * * * * * * * * *] ( 0)    [* *   *       *     *         *] ( 8)
 1:     [* * * * * * * *                ] ( 1)    [* *   *       * * *   * * * *  ] ( 7)
 2:     [        * * * * * * * *        ] ( 2)    [* *   * * * *   * *   *       *] ( 8)
 3:     [* * * *         * * * *        ] ( 3)    [* *   * * * *       *   * * *  ] ( 7)
 4:     [    * * * *         * * * *    ] ( 4)    [* * *   * *   *       * * *   *] ( 8)
 5:     [    * * * *     * *         * *] ( 5)    [* * *   * *   * * * *       *  ] ( 7)
 6:     [    * *     * * * *     * *    ] ( 6)    [* * *       *   * * *   * *   *] ( 8)
 7:     [* *     * *     * *     * *    ] ( 7)    [* * *       *         *     *  ] ( 7)
 8:     [  * *     * *     * *     * *  ] ( 8)    [*   * *   * * *   *       * * *] ( 8)
 9:     [  * *     * *   *     * *     *] ( 9)    [*   * *   * * * *   * * *      ] ( 7)
10:     [*     *   * *     * *   *     *] (10)    [*   * * *       *   * *   * * *] ( 8)
11:     [  * *   *     *   * *   *     *] (11)    [*   * * *         *     *      ] ( 7)
12:     [  *   * *   *     *   * *   *  ] (12)    [*       *   * *   * * * *   * *] ( 8)
13:     [  *   * *   *   *   *     *   *] (13)    [*       *   * * *         *    ] ( 7)
14:     [  *   *   *   * *   *   *   *  ] (14)    [*         *     *       *   * *] ( 8)
15:     [*   *   *   *   *   *   *   *  ] (15)    [*         *       * * *   *    ] ( 7)
```

SEQ

INVERSE SEQ

Chapter 4

The Hartley transform (HT)

4.1 Definition of the HT

The Hartley transform (HT) is defined like the Fourier transform with '$\cos + \sin$' instead of '$\cos + i \cdot \sin$'. The (discrete) Hartley transform of a is defined as

$$\begin{aligned} c &= \mathcal{H}\left[a\right] && (4.1) \\ c_k &:= \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left(\cos \frac{2\,\pi\,k\,x}{n} + \sin \frac{2\,\pi\,k\,x}{n} \right) && (4.2) \end{aligned}$$

It has the obvious property that real input produces real output,

$$\mathcal{H}\left[a\right] \in \mathbb{R} \qquad \text{for} \quad a \in \mathbb{R} \tag{4.3}$$

It also is its own inverse:

$$\mathcal{H}\left[\mathcal{H}\left[a\right]\right] = a \tag{4.4}$$

The symmetries of the HT are simply:

$$\begin{aligned} \mathcal{H}\left[a_S\right] &= \overline{\mathcal{H}\left[a_S\right]} = \mathcal{H}\left[\overline{a_S}\right] && (4.5) \\ \mathcal{H}\left[a_A\right] &= \overline{\mathcal{H}\left[a_A\right]} = -\mathcal{H}\left[\overline{a_A}\right] && (4.6) \end{aligned}$$

i.e. symmetry is, like for the FT, conserved.

4.2 Complex valued FT by HT

The relations between the HT and the FT can be read off directly from their definitions and their symmetry relations. Let σ be the sign of the exponent in the FT, then the HT of a complex sequence $d \in \mathbb{C}$ is:

$$\mathcal{F}\left[d\right] = \frac{1}{2} \left(\mathcal{H}\left[d\right] + \overline{\mathcal{H}\left[d\right]} + \sigma\, i \left(\mathcal{H}\left[d\right] - \overline{\mathcal{H}\left[d\right]} \right) \right) \tag{4.7}$$

Written out for the real and imaginary part $d = a + i\,b$ $(a, b \in \mathbb{R})$:

$$\begin{aligned} \Re\mathcal{F}\left[a + i\,b\right] &= \frac{1}{2} \left(\mathcal{H}\left[a\right] + \overline{\mathcal{H}\left[a\right]} - \sigma \left(\mathcal{H}\left[b\right] - \overline{\mathcal{H}\left[b\right]} \right) \right) && (4.8) \\ \Im\mathcal{F}\left[a + i\,b\right] &= \frac{1}{2} \left(\mathcal{H}\left[b\right] + \overline{\mathcal{H}\left[b\right]} + \sigma \left(\mathcal{H}\left[a\right] - \overline{\mathcal{H}\left[a\right]} \right) \right) && (4.9) \end{aligned}$$

This leads to the following

Code 4.1 (complex FT by HT, version 1) *Pseudo code for the complex Fourier transform that uses the Hartley transform,* is *must be -1 or +1:*

```
fft_by_fht1(a[],b[],n,is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    fht(a[],n)
    fht(b[],n)

    for k:=1 to n/2-1
    {
        t := n-k

        as := a[k] + a[t]
        aa := a[k] - a[t]

        bs := b[k] + b[t]
        ba := b[k] - b[t]

        aa := is * aa
        ba := is * ba

        a[k] := 1/2 * (as - ba)
        a[t] := 1/2 * (as + ba)

        b[k] := 1/2 * (bs + aa)
        b[t] := 1/2 * (bs - aa)
    }
}
```

Alternatively, one can recast the relations (using the symmetry relations 4.5 and 4.6) as

$$\Re\mathcal{F}\left[a + i\,b\right] = \frac{1}{2}\,\mathcal{H}\left[a_S - \sigma\,b_A\right] \tag{4.10}$$

$$\Im\mathcal{F}\left[a + i\,b\right] = \frac{1}{2}\,\mathcal{H}\left[b_S + \sigma\,a_A\right] \tag{4.11}$$

which leads to this

Code 4.2 (complex FT by HT, version 2) *Pseudo code for the complex Fourier transform that uses the Hartley transform,* is *must be -1 or +1:*

```
fft_by_fht2(a[],b[],n,is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    for k:=1 to n/2-1
    {
        t := n-k

        as := a[k] + a[t]
        aa := a[k] - a[t]

        bs := b[k] + b[t]
        ba := b[k] - b[t]

        aa := is * aa
        ba := is * ba

        a[k] := 1/2 * (as - ba)
        a[t] := 1/2 * (as + ba)

        b[k] := 1/2 * (bs + aa)
        b[t] := 1/2 * (bs - aa)
    }

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

Note that the real and imaginary parts of the FT are computed independently by this procedure.

For convolutions it would be sensible to use procedure 4.1 for the forward and 4.2 for the backward transform. The complex squarings are then combined with the pre- and postprocessing steps, thereby interleaving the most nonlocal memory accesses with several arithmetic operations. In effect the routine is (about) twice as memory local as the direct FFT implementation.

[FXT: fht_fft in file fht/fhtcfft.cc]

[FXT: fht_fft0 in file fht/fhtcfft.cc]

4.3 Real valued FT by HT

To express the real and imaginary part of a Fourier transform of a purely real sequence $a \in \mathbb{R}$ by its Hartley transform use relations 4.8 and 4.9 and set $b = 0$:

$$\begin{aligned} \Re\mathcal{F}[a] &= \frac{1}{2}\left(\mathcal{H}[a] + \overline{\mathcal{H}[a]}\right) && (4.12) \\ \Im\mathcal{F}[a] &= \frac{1}{2}\left(\mathcal{H}[a] - \overline{\mathcal{H}[a]}\right) && (4.13) \end{aligned}$$

The pseude code is straightforward:

Code 4.3 (real to complex FFT via FHT)

```
procedure real_complex_fft_by_fht(a[],n)
// real a[0..n-1] input,result
{
    fht(a[],n)
    for i:=1 to n/2-1
    {
        t := n - i
        u := a[i]
        v := a[t]
        a[i] := 1/2 * (u+v)
        a[t] := 1/2 * (u-v)
    }
}
```

At the end of this procedure the ordering of the output data $c \in \mathbb{C}$ is

$$\begin{aligned} \mathsf{a[0]} &= \Re c_0 && (4.14) \\ \mathsf{a[1]} &= \Re c_1 \\ \mathsf{a[2]} &= \Re c_2 \\ & \dots \\ \mathsf{a[n/2]} &= \Re c_{n/2} \\ \mathsf{a[n/2+1]} &= \Im c_{n/2-1} \\ \mathsf{a[n/2+2]} &= \Im c_{n/2-2} \\ \mathsf{a[n/2+3]} &= \Im c_{n/2-3} \\ & \dots \\ \mathsf{a[n-1]} &= \Im c_1 \end{aligned}$$

The inverse procedure is:

Code 4.4 (complex to real FFT via FHT)

```
procedure complex_real_fft_by_fht(a[],n)
// real a[0..n-1] input,result
{
    for i:=1 to n/2-1
    {
        t := n - i
        u := a[i]
        v := a[t]
        a[i] := u+v
        a[t] := u-v
    }
    fht(a[],n)
}
```

4.4 HT by real valued FT

4.5 radix 2 FHT algorithms

4.5.1 Decimation in time (DIT) FHT

For a sequence a of length n let $\mathcal{X}^{1/2}a$ denote the sequence with elements $a_x \cos \pi\, x/n + \overline{a}_x \sin \pi\, x/n$ (this is the 'shift operator' for the Hartley transform).

Idea 4.1 (FHT radix 2 DIT step) *Radix 2 decimation in time step for the FHT:*

$$\mathcal{H}\left[a\right]^{(left)} \overset{n/2}{=} \mathcal{H}\left[a^{(even)}\right] + \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \qquad (4.15)$$

$$\mathcal{H}\left[a\right]_n^{(right)} \overset{n/2}{=} \mathcal{H}\left[a^{(even)}\right] - \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \qquad (4.16)$$

Code 4.5 (recursive radix 2 DIT FHT) *Pseudo code for a recursive procedure of the (radix 2) DIT FHT algorithm:*

```
procedure rec_fht_dit2(a[],n,x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace
    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2;
    for k:=0 to nh-1
    {
        s[k] := a[2*k]     // even indexed elements
        t[k] := a[2*k+1]   // odd  indexed elements
    }
    rec_fht_dit2(s[],nh,b[])
    rec_fht_dit2(t[],nh,c[])
    hartley_shift(c[],nh,1/2)
    for k:=0 to nh-1
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The result is in `x[]`.

[FXT: `recursive_dit2_fht` in file `learn/recfhtdit2.cc`]

The procedure `hartley_shift()` replaces element c_k of the input sequence c by $c_k\, cos(\pi\, k/n) + c_{n-k}\, sin(\pi\, k/n)$. Here is the pseudo code:

Code 4.6 (Hartley shift)

```
procedure hartley_shift(c[],n,v)
// real c[0..n-1] input, result
{
    nh := n/2

    j := n-1
    for k:=1 to nh-1
    {
        c := cos(v*2*PI*k/n)
        s := sin(v*2*PI*k/n)

        {c[k], c[j]} := {c[k]*c+c[j]*s, c[k]*s-c[j]*c}

        j := j-1
    }
}
```

[FXT: hartley_shift_05 in file fht/hartleyshift.cc]

Code 4.7 (radix 2 DIT FHT, naive) *Pseudo code for a non-recursive procedure of the (radix 2) DIT FHT algorithm:*

```
procedure fht_dit2(a[],ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[],n)

    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4

        for r:=0 to n-m step m
        {
            for j:=1 to m4-1  // hartley_shift(a+r+mh,mh,1/2)
            {
                k := mh - j

                u := a[r+mh+j]
                v := a[r+mh+k]

                c := cos(j*PI/mh)
                s := sin(j*PI/mh)

                {u, v} := {u*c+v*s, u*s-v*c}

                a[r+mh+j] := u
                a[r+mh+k] := v
            }
            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]

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

[FXT: dit2_fht_localized in file learn/fhtdit2.cc]

4.5.2 Decimation in frequency (DIF) FHT

Idea 4.2 (FHT radix 2 DIF step) *Radix 2 decimation in frequency step for the FHT:*

$$\mathcal{H}\left[a\right]^{(even)} \stackrel{n/2}{=} \mathcal{H}\left[a^{(left)} + a^{(right)}\right] \tag{4.17}$$

$$\mathcal{H}\left[a\right]^{(odd)} \stackrel{n/2}{=} \mathcal{H}\left[\mathcal{X}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{4.18}$$

Code 4.8 (recursive radix 2 DIF FHT) *Pseudo code for a recursive procedure of the (radix 2) DIF FHT algorithm:*

```
procedure rec_fht_dif2(a[],n,x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace
    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2;
    for k:=0 to nh-1
    {
        s[k] := a[k]      // 'left'  elements
        t[k] := a[k+nh]   // 'right' elements
    }
    for k:=0 to nh-1
    {
        {s[k], t[k]} := {s[k]+t[k], s[k]-t[k]}
    }
    hartley_shift(t[],nh,1/2)
    rec_fht_dif2(s[],nh,b[])
    rec_fht_dif2(t[],nh,c[])
    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

The result is found in x[].

[FXT: recursive_dit2_fht in file learn/recfhtdit2.cc]

Code 4.9 (radix 2 DIF FHT, naive) *Pseudo code for a non-recursive procedure of the (radix 2) DIF FHT algorithm:*

```
procedure fht_dif2(a[],ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4
        for r:=0 to n-m step m
        {
            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
            for j:=1 to m4-1
            {
                k := mh - j
                u := a[r+mh+j]
                v := a[r+mh+k]
                c := cos(j*PI/mh)
                s := sin(j*PI/mh)

                {u, v} := {u*c+v*s, u*s-v*c}

                a[r+mh+j] := u
                a[r+mh+k] := v
            }
        }
    }
    revbin_permute(a[],n)
}
```

[FXT: dif2_fht_localized in file learn/fhtdif2.cc]

4.6 Discrete cosine transform (DCT) by HT

Code 4.10 (DCT via FHT) *Pseudo code for the computation of the DCT via FHT:*

```
procedure dct(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    nh := n/2

    real y[0..n-1]  // workspace

    for k:=0 to nh-1
    {
        k2 := 2*k
        y[k]    := x[k2]
        y[nh+k] := x[n-1-k2]
    }

    fht(y[],ldn)

    x[0]  := y[0]
    x[nh] := y[nh]
    phi := PI/2/n
    for (ulong k:=1; k<nh; k++)
    {
        c := cos(phi*k)
        s := sin(phi*k)

        cps := (c+s)*sqrt(1/2)
        cms := (c-s)*sqrt(1/2)

        x[k]   := cms*y[k] + cps*y[n-k]
        x[n-k] := cps*y[k] - cms*y[n-k]
    }
}
```

[FXT: dcth in file dctdst/dcth.cc]

Code 4.11 (IDCT via FHT) *Pseudo code for the computation of the IDCT via FHT:*

```
procedure idct(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    nh := n/2

    real y[0..n-1]  // workspace

    y[0]  := x[0]
    y[nh] := x[nh]
    phi := PI/2/n
    for (ulong k:=1; k<nh; k++)
    {
        c := cos(phi*k)
        s := sin(phi*k)

        cps := (c+s)*sqrt(1/2)
        cms := (c-s)*sqrt(1/2)

        y[k]   := cms*x[k] + cps*x[n-k]
        y[n-k] := cps*x[k] - cms*x[n-k]
    }

    fht(y[],ldn)

    for k:=0 to nh-1
    {
        k2 := 2*k
        x[k]    := y[k2]
        x[nh+k] := y[n-1-k2]
    }
}
```

[FXT: idcth in file dctdst/dcth.cc]

4.7 Discrete sine transform (DST) by DCT

Code 4.12 (DST via DCT) *Pseudo code for the computation of the DST via DCT:*

```
procedure dst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    nh := n/2

    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }

    dct(x,ldn)

    for k:=0 to nh-1
    {
        swap(x[k],x[n-1-k])
    }
}
```

[FXT: dsth in file dctdst/dsth.cc]

Code 4.13 (IDST via IDCT) *Pseudo code for the computation of the inverse sine transform (IDST) using the inverse cosine transform (IDCT):*

```
procedure idst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    nh := n/2

    for k:=0 to nh-1
    {
        swap(x[k],x[n-1-k])
    }

    idct(x,ldn)

    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }
}
```

[FXT: idsth in file dctdst/dsth.cc]

4.8 Convolution via FHT

The convolution property of the HT is

$$\mathcal{H}[a \circledast b] = \frac{1}{2}\left(\mathcal{H}[a]\,\mathcal{H}[b] - \overline{\mathcal{H}[a]}\,\overline{\mathcal{H}[b]} + \mathcal{H}[a]\,\overline{\mathcal{H}[b]} + \overline{\mathcal{H}[a]}\,\mathcal{H}[b]\right) \tag{4.19}$$

or, written elementwise:

$$\begin{aligned}\mathcal{H}[a \circledast b]_k &= \frac{1}{2}\left(c_k\,d_k - \overline{c_k}\,\overline{d_k} + c_k\,\overline{d_k} + \overline{c_k}\,d_k\right) \\ &= \frac{1}{2}\left(c_k\,(d_k + \overline{d_k}) + \overline{c_k}\,(d_k - \overline{d_k})\right) \qquad \text{where} \quad c = \mathcal{H}[a], \quad d = \mathcal{H}[b]\end{aligned} \tag{4.20}$$

Code 4.14 (cyclic convolution via FHT) *Pseudo code for the cyclic convolution of two real valued sequences* `x[]` *and* `y[]`, `n` *must be even, result is found in* `y[]`*:*

```
procedure fht_cyclic_convolution(x[],y[],n)
// real x[0..n-1] input, modified
// real y[0..n-1] result
{
    // transform data:
    fht(x[],n)
    fht(y[],n)

    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        xi := x[i]
        xj := x[j]

        yp := y[i] + y[j]    // =   y[j] + y[i]
        ym := y[i] - y[j]    // = -(y[j] - y[i])

        y[i] := (xi*yp + xj*ym)/2
        y[j] := (xj*yp - xi*ym)/2

        j := j-1
    }
    y[0] := y[0]*y[0]
    if n>1 then  y[n/2] := y[n/2]*y[n/2]

    // transform back:
    fht(y[],n)

    // normalise:
    for i:=0 to n-1
    {
        y[i] := y[i]/n
    }
}
```

It is assumed that the procedure `fht()` does no normalisation.

Equation 4.20 (slightly optimised) for the auto convolution is

$$\begin{aligned}\mathcal{H}[a \circledast a]_k &= \frac{1}{2}\left(c_k\,(c_k + \overline{c_k}) + \overline{c_k}\,(c_k - \overline{c_k})\right) \\ &= c_k\,\overline{c_k} + \frac{1}{2}\left(c_k^2 - \overline{c_k}^2\right) \qquad \text{where} \quad c = \mathcal{H}[a]\end{aligned} \tag{4.21}$$

Code 4.15 (cyclic auto convolution via FHT) *Pseudo code for an auto convolution that uses a fast Hartley transform,* `n` *must be even:*

```
procedure cyclic_self_convolution(x[],n)
// real x[0..n-1] input, result
```

```
{
    // transform data:
    fht(x[],n)

    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        ci := x[i]
        cj := x[j]

        t1 := ci*cj                 // = cj*ci
        t2 := 1/2*(ci*ci-cj*cj)     // = -1/2*(cj*cj-ci*ci)

        x[i] := t1 + t2
        x[j] := t1 - t2

        j := j-1
    }
    x[0]   := x[0]*x[0]

    if n>1 then  x[n/2] := x[n/2]*x[n/2]

    // transform back:
    fht(x[],n)

    // normalise:
    for i:=0 to n-1
    {
        x[i] := x[i]/n
    }
}
```

For odd n replace the line

```
    for i:=1 to n/2-1
```

by

```
    for i:=1 to (n-1)/2
```

and omit the line

```
    if n>1 then  x[n/2] := x[n/2]*x[n/2]
```

in both procedures above.

4.9 Negacyclic convolution via FHT

Code 4.16 (negacyclic auto convolution via FHT) *Code for the computation of the negacyclic (auto-) convolution:*

```
procedure negacyclic_self_convolution(x[],n)
// real x[0..n-1]  input, result
{
    // preprocessing:
    hartley_shift(x,n,1/2)

    // transform data:
    fht(x,n)

    // convolution in transformed domain:
    j := n-1
    for i:=0 to n/2-1  // here i starts from zero
    {
        a := x[i]
        b := x[j]

        x[i] := a*b+(a*a-b*b)/2
        x[j] := a*b-(a*a-b*b)/2
        j := j-1
    }
    // transform back:
    fht(x,n)

    // postprocessing:
    hartley_shift(x,n,1/2)
}
```

[FXT: `fht_negacyclic_auto_convolution` in file `fht/fhtnegacnvla.cc`]

(The code for `hartley_shift()` was given on page 51.)

Chapter 5

Numbertheoretic transforms (NTTs)

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

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

There is a restriction for the choice of m: 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* n*-th root* if $r^k \neq 1 \; \forall \, k < n$.

In $\mathbb{C}$ matters are simple: $e^{\pm 2\,\pi\, i/n}$ 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 $\mathbb{Z}/m\mathbb{Z}$ is also called an *element of order* n. The 'cyclic' property of the elements r of order n lies in the heart of all FFT algorithms: $r^{n+k} = r^k$.

In $\mathbb{Z}/m\mathbb{Z}$ things are not that simple since primitive roots of unity do not exist for arbitrary n, they exist for some maximal order R only. Roots of unity of an order different from R are available only for the divisors d_i of R: r^{R/d_i} is a d_i-th root of unity because $(r^{R/d_i})^{d_i} = r^R = 1$.

Therefore n must divide R, the first condition for NTTs:

$$n \backslash R \iff \exists \sqrt[n]{1} \tag{5.1}$$

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 $\mathbb{Z}/m\mathbb{Z}$: n must be coprime[1] to m, the second condition for NTTs:

$$n \perp m \iff \exists n^{-1} \text{ in } \mathbb{Z}/m\mathbb{Z} \tag{5.2}$$

Cf. [1], [6], [30] or [3] and books on number theory.

5.1 Prime modulus: $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p$

If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p$: All elements except 0 have inverses and 'division is possible' in $\mathbb{Z}/p\mathbb{Z}$. Thereby the second condition is trivially fulfilled for all FFT lengthes $n < p$: a prime p is coprime to all integers $n < p$.

[1] n coprime to $m \iff \gcd(n, m) = 1$

Roots of unity are available for the maximal order $R = p-1$ and its divisors: Therefore the first condition on n for a length-n mod p FFT being possible is that n divides $p-1$. This restricts the choice for 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)[2]. The elements of maximal order in $\mathbb{Z}/p\mathbb{Z}$ are called primitive elements, generators or primitive roots modulo p. If r is a generator, then every element in $\mathbb{F}_p$ different from 0 is equal to some power r^e $(1 \leq e < p)$ of r and its order is R/e. To test whether r is a primitive n-th root of unity in $\mathbb{F}_p$ one doesn't need to check $r^k \neq 1$ for all $k < n$. 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:

Code 5.1 (Primitive root modulo p) *Return a primitive root in* $\mathbb{F}_p$

```
function primroot(p)
{
    if p==2 then  return 1
    f[] := distinct_prime_factors(p-1)
    for r:=2 to p-1
    {
        x := TRUE
        foreach q in f[]
        {
            if r**((p-1)/q)==1 then x:=FALSE
        }
        if x==TRUE then  return r
    }
    error("no primitive root found")  // p cannot be prime !
}
```

An element of order n is returned by this function:

Code 5.2 (Find element of order n) *Return an element of order n in* $\mathbb{F}_p$*:*

```
function element_of_order(n,p)
{
    R := p-1  // maxorder
    if (R/n)*n != R then  error("order n must divide maxorder p-1")
    r := primroot(p)
    x := r**(R/n)
    return x
}
```

5.2 Composite modulus: $\mathbb{Z}/m\mathbb{Z}$, cyclic vs. noncyclic

In what follows we will need the function $\varphi()$, the so-called 'totient' function. $\varphi(m)$ counts the number of integers prime to and less than m. For $m = p$ prime $\varphi(p) = p - 1$. For m composite $\varphi(m)$ is always less than $m - 1$. For $m = p^k$ a prime power

$$\varphi(p^k) \quad = \quad p^k - p^{k-1} \tag{5.3}$$

e.g. $\varphi(2^k) = 2^{k-1}$. $\varphi(1) = 1$. For coprime p_1, p_2 (p_1, p_2 not necessarily primes) $\varphi(p_1\,p_2) = \varphi(p_1)\,\varphi(p_2)$, $\varphi()$ is a so-called *multiplicative* function.

For the computation of $\varphi(m)$ for m a prime power one can use this simple piece of code

Code 5.3 (Compute phi(m) for m a prime power) *Return* $\varphi(p^x)$

```
function phi_pp(p,x)
```

[2] Primes of that form are not 'exceptional', cf. Lipson [6]

```
{
    if x==1 then  return p - 1
    else          return  p**x - p**(x-1)
}
```

Pseudo code to compute $\varphi(m)$ for general m:

Code 5.4 (Compute phi(m)) *Return* $\varphi(m)$

```
function phi(m)
{
    {n, p[], x[]} := factorization(m)  // m==product(i=0..n-1,p[i]**x[i])
    ph := 1
    for i:=0 to n-1
    {
        ph := ph * phi_pp(p[i],x[i])
    }
}
```

Further we need the notion of $\mathbb{Z}/m\mathbb{Z}^*$, the ring of units in $\mathbb{Z}/m\mathbb{Z}$. $\mathbb{Z}/m\mathbb{Z}^*$ contains all invertible elements ('units') of $\mathbb{Z}/m\mathbb{Z}$, i.e. those which are coprime to m. Evidently the total number of units is given by $\varphi(m)$:

$$|\mathbb{Z}/m\mathbb{Z}^*| \quad = \quad \varphi(m) \tag{5.4}$$

If m factorizes as $m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q}$ then

$$|\mathbb{Z}/m\mathbb{Z}^*| \quad = \quad \varphi(2^{k_0}) \cdot \varphi(p_1^{k_1}) \cdot \ldots \cdot \varphi(p_q^{k_q}) \tag{5.5}$$

It turns out that the maximal order R of an element can be equal to or less than $|\mathbb{Z}/m\mathbb{Z}^*|$, the ring $\mathbb{Z}/m\mathbb{Z}^*$ is then called *cyclic* or *noncyclic*, respectively. For m a power of an odd prime p the maximal order R in $\mathbb{Z}/m\mathbb{Z}^*$ (and also in $\mathbb{Z}/m\mathbb{Z}$) is

$$R(p^k) \quad = \quad \varphi(p^k) \tag{5.6}$$

while for m a power of two a tiny irregularity enters:

$$R(2^k) \quad = \quad \begin{cases} 1 & \text{for } k = 1 \\ 2 & \text{for } k = 2 \\ 2^{k-2} & \text{for } k \geq 3 \end{cases} \tag{5.7}$$

i.e. for powers of two greater than 4 the maximal order deviates from $\varphi(2^k) = 2^{k-1}$ by a factor of 2. For the general modulus $m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q}$ the maximal order is

$$R(m) \quad = \quad \operatorname{lcm}(R(2^{k_0}), R(p_1^{k_1}), \ldots, R(p_q^{k_q})) \tag{5.8}$$

where lcm() denotes the least common multiple.

Pseudo code to compute $R(m)$:

Code 5.5 (Maximal order modulo m) *Return* $R(m)$*, the maximal order in* $\mathbb{Z}/m\mathbb{Z}$

```
function maxorder(m)
{
    {n, p[], k[]} := factorization(m)  // m==product(i=0..n-1,p[i]**k[i])
    R := 1
    for i:=0 to n-1
    {
        t := phi_pp(p[i],k[i])
        if p[i]==2 AND k[i]>=3 then  t := t / 2
        R := lcm(R,t)
    }
    return R
}
```

Now we can see for which m the ring $\mathbb{Z}/m\mathbb{Z}^*$ will be cyclic:

$$\mathbb{Z}/m\mathbb{Z}^* \quad \text{cyclic for} \quad m = 2,\ 4,\ p^k,\ 2 \cdot p^k \tag{5.9}$$

where p is an odd prime. If m contains two different odd primes p_a, p_b then $R(m) = lcm(..., \varphi(p_a), \varphi(p_b), ...)$ is at least by a factor of two smaller than $\varphi(m) = ... \cdot \varphi(p_a) \cdot \varphi(p_b) \cdot ...$ because both $\varphi(p_a)$ and $\varphi(p_b)$ are even, so $\mathbb{Z}/m\mathbb{Z}^*$ can't be cyclic in that case. The same argument holds for $m = 2^{k_0} \cdot p^k$ if $k_0 > 1$. For $m = 2^k$ $\mathbb{Z}/m\mathbb{Z}^*$ is cyclic only for $k = 1$ and $k = 2$ because of the above mentioned irregularity of $R(2^k)$.

The underlying mathematical proofs can be found in .

Pseudo code (following [30]) for a function that returns the order of some element x in $\mathbb{Z}/m\mathbb{Z}$:

Code 5.6 (Order of an element in $\mathbb{Z}/m\mathbb{Z}$) *Return the order of an element x in $\mathbb{Z}/m\mathbb{Z}$*

```
function order(x,m)
{
    if gcd(x,m)!=1 then  return 0  // x not a unit
    h := phi(m)  // number of elements of ring of units
    e := h
    {n, p[], k[]} := factorization(h)  // h==product(i=0..n-1,p[i]**k[i])
    for i:=0 to n-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := x**e mod m
        while g1!=1
        {
            g1 := g1**p[i] mod m
            e := e * p[i]
            p[i] := p[i] - 1
        }
    }
    return e
}
```

Pseudo code for a function that returns some element x in $\mathbb{Z}/m\mathbb{Z}$ of maximal order:

Code 5.7 (Element of maximal order in $\mathbb{Z}/m\mathbb{Z}$) *Return an element that has maximal order in $\mathbb{Z}/m\mathbb{Z}$*

```
function maxorder_element(m)
{
    R := maxorder(m)
    for x:=1 to m-1
    {
        if order(x,m)==R then  return x
    }
    // never reached
}
```

For prime m the function returns a primitive root. It is a good idea to have a table of small primes stored (which will also be useful in the factorization routine) and restrict the search to small primes and only if the modulus is greater than the largest prime of the table proceed with a loop as above:

Code 5.8 (Element of maximal order in $\mathbb{Z}/m\mathbb{Z}$) *Return an element that has maximal order in $\mathbb{Z}/m\mathbb{Z}$, use a precomputed table of primes*

```
function maxorder_element(m,pt[],np)
// pt[0..np-1] = 2,3,5,7,11,13,17,...
{
    if m==2 then  return 1
    R := maxorder(m)
    for i:=0 to np-1
    {
        if order(pt[i],m)==R then  return x
    }
    // hardly ever reached
    for x:=pt[np-1] to m-1 step 2
    {
        if order(x,m)==R then  return x
    }
    // never reached
}
```

[FXT: maxorder_element_mod in file mod/maxorder.cc]

There is no problem if the prime table contains primes $\geq m$: The first loop will finish before order() is called with an element $\geq m$, because before that can happen, the element of maximal order is found.

5.2.1 Cyclic rings

5.2.2 Noncyclic rings

5.3 Pseudocode for NTTs

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

For the backtransform one uses the (mod m) inverse $\bar{r}$ of r (an element of order n) that was used for the forward transform. To check whether $\bar{r}$ exists one tests whether $gcd(r, m) = 1$. To compute the inverse modulo m one can use the relation $\bar{r} = r^{\varphi(p)-1}$ $(mod\, m)$. Alternatively one may use the extended Euclidean algorithm, which for two integers a and b finds $d = gcd(a, b)$ and u, v so that $a\,u + b\,v = d$. Feeding $a = r$, $b = m$ into the algorithm gives u as the inverse: $r\,u + m\,v \equiv r\,u \equiv 1$ $(mod\, m)$.

While the notion of the Fourier transform as a 'decomposition into frequencies' seems to be meaningless for NTTs the algorithms are denoted with 'decimation in time/frequency' in analogy to those in the complex domain.

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

5.3.1 Radix 2 DIT NTT

Code 5.9 (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(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn
    rn := element_of_order(n)  // (mod_type)
    if is<0 then  rn := rn**(-1)
    revbin_permute(f[],n)
    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        dw := rn**(2**(ldn-ldm))    // (mod_type)
        w  := 1                     // (mod_type)
```

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

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

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

                f[t1] := u+v
                f[t2] := u-v
            }
            w := w*dw
        }
    }
}
```

Like in 1.3.2 it is a good idea to extract the `ldm==1` stage of the outermost loop:
Replace

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

by

```
    for r:=0 to n-1 step 2
    {
        {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
    }
    for ldm:=2 to ldn
    {
```

5.3.2 Radix 2 DIF NTT

Code 5.10 (radix 2 DIF NTT) *Pseudo code for the radix 2 decimation in frequency mod fft:*

```
procedure mod_fft_dif2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn
    dw := element_of_order(n)  // (mod_type)

    if is<0 then  dw := rn**(-1)

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

        w := 1  // (mod_type)

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

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

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

    revbin_permute(f[],n)
}
```

As in section 1.3.3 extract the `ldm==1` stage of the outermost loop:
Replace the line

```
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
{
    {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
}
```

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

5.4 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.

Integer convolution: Split input mod $m1$, $m2$, do 2 FFT convolutions, combine with CRT.

5.5 Numbertheoretic Hartley transform

Let r be an element of order n, i.e. $r^n = 1$ (but there is no $k < n$ so that $r^k = 1$) we like to identify r with $\exp(2\,i\,\pi/n)$.

Then one can set

$$\cos\frac{2\,\pi}{n} \equiv \frac{r^2+1}{2\,r} \tag{5.10}$$

$$i\,\sin\frac{2\,\pi}{n} \equiv \frac{r^2-1}{2\,r} \tag{5.11}$$

For This choice of sin and cos the relations $\exp() = \cos() + i\,\sin()$ and $\sin()^2 + \cos()^2 = 1$ should hold. The first check is trivial: $\frac{x^2+1}{2\,x} + \frac{x^2-1}{2\,x} = x$. The second is also easy if we allow to write i for some element that is the square root of -1: $(\frac{x^2+1}{2\,x})^2 + (\frac{x^2-1}{2\,x\,i})^2 = \frac{(x^2+1)^1-(x^2-1)^2}{4\,x^2} = 1$. Ok, but what is i in the modular ring ? Simply $r^{n/4}$, then we have $i^2 = -1$ and $i^4 = 1$ as we are used to. This is only true in cyclic rings .

[4] Multiplication is a convolution of the digits followed by the 'carry' operations.

[5] A radix R 'nine' is $R-1$, nine in radix 10 is 9.

[6] for length-2^k FFTs

Chapter 6

Wavelet transforms

6.1 The Haar transform

basis functions have compact support

combination step (haar step: nur DC neu + 1.freq aus den 2 alten DC) −¿ complexity proportional n

as matrix mult

pyramid algorithms

Code 6.1 (Haar transform) *pseudo code for the Haar transform:*

```
procedure haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**ldn
    real g[0..n-1]  // workspace

    s2 := sqrt(0.5)
    v  := 1.0

    for m:=n to 2 div_step 2
    {
        v := v * s2

        mh = m/2

        k := 0
        for j=0 to m-1 step 2
        {
            x := f[j]
            y := f[j+1]
            g[k]     :=  x+y
            g[mh+k] := (x-y)*v

            k := k+1
        }

        copy g[0..m-1] to f[0..m-1]
    }

    f[0] := f[0]*v   // v==1.0/sqrt(n)
}
```

Code 6.2 (inverse Haar transform) *pseudo code for the inverse Haar transform:*

```
procedure inverse_haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**ldn
    real g[0..n-1]  // workspace
    s2 := sqrt(0.5)
    v := 1.0/sqrt(n)

    f[0] := f[0]*v

    for m:=2 to n mul_step 2
    {
        mh := m/2

        k := 0
        for j=0 to m-1 step 2
        {
            x := f[k]
            y := f[mh+k] * v
            g[j]   :=  x+y
            g[j+1] :=  x-y

            k := k+1
        }

        copy g[0..m-1] to f[0..m-1]

        v := v * s2
    }
}
```

6.2 Inplace Haar transform

localized ordering of basis functions

Code 6.3 (inplace Haar transform) *pseudo code for the inplace Haar transform:*

```
procedure inplace_haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**n

    s2 := sqrt(0.5)
    v  := 1.0

    for js:=2 to n mul_step 2
    {
        v := v * s2

        t := j + js/2
        for j:=0 to n-1 step js
        {
            // {f[j], f[t]} := {f[j]+f[t], (f[j]-f[t])*v}
            x := f[j]
            y := f[t]
            f[j] := x + y
            f[t] := (x - y) * v

            t := t + js
        }
    }

    f[0] := f[0]*v   // v==1.0/sqrt(n)

    revbin_permute(f[],n)
}
```

Code 6.4 (inplace inverse Haar transform) *pseudo code for the inverse inplace Haar transform:*

```
procedure inverse_inplace_haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**n

    revbin_permute(f[],n)

    s2 := sqrt(0.5)
    v  := 1.0/sqrt(n)

    f[0] := f[0]*v
    for js:=n to 2 div_step 2
    {
        t := j + js/2
        for j:=0 to n-1 step js
        {
            // {f[j], f[t]} := {f[j]+f[t]*v, f[j]-f[t]*v}
            x := f[j]
            y := f[t] * v
            f[j] := x + y
            f[t] := x - y
            t := t + js
        }
        v := v * s2
    }
}
```

6.3 Integer to integer Haar transform

Code 6.5 (integer to integer Haar transform)

```
procedure int_haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**n
    real g[0..n-1]  // workspace
    for m:=n to 2 div_step 2
    {
        mh = m/2
        k := 0
        for j=0 to m-1 step 2
        {
            x := f[j]
            y := f[j+1]
            d := x - y
            s := y + floor(d/2) // == floor((x+y)/2)
            g[k]    :=  s
            g[mh+k] :=  d
            k := k + 1
        }
        copy g[0..m-1] to f[0..m-1]
        m := m/2
    }
}
```

jjnote: one can omit floor() with type integer

Code 6.6 (inverse integer to integer Haar transform)

```
procedure inverse_int_haar(f[],ldn)
// real f[0..2**ldn-1]  // input, result
{
    n := 2**n
    real g[0..n-1]  // workspace
    for m:=2 to n mul_step 2
    {
        mh := m/2
        k := 0
        for j=0 to m-1 step 2
        {
            s := f[k]
            d := f[mh+k]
            y := s - floor(d/2)
            x := d + y  // == s+floor((d+1)/2)
            g[j]   := x
            g[j+1] := y
            k := k + 1
        }
        copy g[0..m-1] to f[0..m-1]
        m := m * 2
    }
}
```

Appendix A

Definition of Fourier transforms

The continuous Fourier transform

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

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

where $\sigma = \pm 1$. The FT is is a unitary transform.
Its inverse ('backtransform') is

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

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

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

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

The 'frequency'-form is

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

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

The semi-continuous Fourier transform

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

$$c_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, e^{\sigma\, 2\, \pi\, i\, k\, x / L} dx \tag{A.7}$$

Then

$$\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k\, e^{-\sigma\, 2\, \pi\, i\, k\, x / L} \quad = \quad \begin{cases} f(x) & \text{if } f \text{ continuous at } x \\ \frac{f(x+0)+f(x-0)}{2} & \text{else} \end{cases} \tag{A.8}$$

Another form is given by

$$a_k := \frac{1}{\sqrt{L}} \int_L f(x)\, \cos \frac{2\,\pi\,k\,x}{L} dx, \qquad k = 0, 1, 2, ... \tag{A.9}$$

$$b_k := \frac{1}{\sqrt{L}} \int_L f(x)\, \sin \frac{2\,\pi\,k\,x}{L} dx, \qquad k = 1, 2, ... \tag{A.10}$$

$$f(x) = \frac{1}{\sqrt{L}} \left[\frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k\, \cos \frac{2\,\pi\,k\,x}{L} + b_k\, \sin \frac{2\,\pi\,k\,x}{L} \right) \right] \tag{A.11}$$

with

$$c_k = \begin{cases} \frac{a_0}{2} & (k = 0) \\ \frac{1}{2}(a_k - i b_k) & (k > 0) \\ \frac{1}{2}(a_k + i b_k) & (k < 0) \end{cases} \tag{A.12}$$

The discrete Fourier transform

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

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x\, e^{\sigma\,2\,\pi\,i\,x\,k/n} \tag{A.13}$$

Backtransform is

$$f_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k\, e^{\sigma\,2\,\pi\,i\,x\,k/n} \tag{A.14}$$

Cf. [3] and [27].

Appendix B

The pseudo language Sprache

Many algorithms in this book are given in a pseudo language called Sprache. Sprache is meant to be immediately understandable for everyone who ever had contact with programming languages like C, FORTRAN, pascal or algol. Sprache is hopefully self explanatory. The intention of using Sprache instead of e.g. mathematical formulas (cf. [9]) or description by words (cf. [18] or [30]) was to minimize the work it takes to translate the given algorithm to one's favorite programming language, it should be mere syntax adaptation.

By the way 'Sprache' is the german word for language,

```
// a comment:
// comments are useful.

// assignment:
t := 2.71

// parallel assignment:
{s, t, u} := {5, 6, 7}
// same as:
s := 5
t := 6
u := 7

{s, t} := {s+t, s-t}
// same as (avoid temporary):
temp := s + t
t := s - t;
s := temp


// if conditional:
if a==b then  a:=3

// with block
if a>=3 then
{
    // do something ...
}


// a function returns a value:
function plus_three(x)
{
    return  x + 3;
}

// a procedure works on data:
procedure increment_copy(f[],g[],n)
// real f[0..n-1] input
// real g[0..n-1] result
{
    for k:=0 to n-1
    {
        g[k] := f[k] + 1
    }
}
```

```
// for loop with stepsize:
for i:=0 to n step 2  // i:=0,2,4,6,...
{
    // do something
}

// for loop with multiplication:
for i:=1 to 32 mul_step 2
{
    print i, ", "
}
```

will print `1, 2, 4, 8, 16, 32,`

```
// for loop with division:
for i:=32 to 8 div_step 2
{
    print i, ", "
}
```

will print `32, 16, 8,`

```
// while loop:
i:=5
while i>0
{
    // do something 5 times...
    i := i - 1
}
```

The usage of `foreach` emphasizes that no particular order is needed in the array acces (so parallelization is possible):

```
procedure has_element(f[],x)
{
    foreach t in f[]
    {
        if t==x then return TRUE
    }
    return FALSE
}
```

Emphasize type and range of arrays:

```
real     a[0..n-1],    // has n elements (floating point reals)
complex  b[0..2**n-1]  // has 2**n elements (floating point complex)
mod_type m[729..1728]  // has 1000 elements (modular integers)
integer  i[]           // has ? elements (integers)
```

Arithmetical operators: `+, -, *, /, %` and `**` for powering. Arithmetical functions: `min()`, `max()`, `gcd()`, `lcm()`, `...`

Mathematical functions: `sqr()`, `sqrt()`, `pow()`, `exp()`, `log()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `...`

Bitwise operators: `~, &, |, ^` for negation, and, or, exor, respectively. Bit shift operators: `a<<3` shifts (the integer) `a` 3 bits to the left `a>>1` shifts `a` 1 bits to the right.

Comparison operators: `==, !=, <, > ,<=, >=`

There is no operator '`=`' in Sprache, only '`==`' (for testing equality) and '`:=`' (assignment operator).

A well known constant: `PI` $= 3.14159265...$

The complex square root of minus one in the upper half plane: `I` $= \sqrt{-1}$

Boolean values `TRUE` and `FALSE`

Logical operators: `NOT, AND, OR, EXOR`

```
// copying arrays of same length:
copy a[] to b[]

// more copying arrays:
copy a[n..n+m] to b[0..m]

// skip copy array:
copy a[0,2,4,...,n-1] to b[0,1,2,...,n/2-1]
```

Modular arithmetic: `x := a * b mod m` shall do what it says, `i := a**(-1) mod m` shall set `i` to the modular inverse of `a`.

Appendix C

Eigenvectors of the discrete Fourier transform

For $a_S := a + \overline{a}$:

$$\mathcal{F}\left[\mathcal{F}\left[a_S\right]\right] \;=\; a_S \tag{C.1}$$

Let $u_+ := a_S + \mathcal{F}\left[a_S\right]$ then

$$\begin{aligned} \mathcal{F}\left[u_+\right] &= \mathcal{F}\left[a_S\right] + a_S && \text{(C.2)} \\ &= a_S + \mathcal{F}\left[a_S\right] = +1 \cdot u_+ && \text{(C.3)} \end{aligned}$$

Let $u_- := a_S - \mathcal{F}\left[a_S\right]$ then

$$\begin{aligned} \mathcal{F}\left[u_-\right] &= \mathcal{F}\left[a_S\right] - a_S && \text{(C.4)} \\ &= -(a_S - \mathcal{F}\left[a_S\right]) = -1 \cdot u_- && \text{(C.5)} \end{aligned}$$

u_+ and u_- are symmetric.

For $a_A := a - \overline{a}$:

$$\mathcal{F}\left[\mathcal{F}\left[a_A\right]\right] \;=\; -a_A \tag{C.6}$$

$v_+ := a_A + i\,\mathcal{F}\left[a_A\right]$

$$\begin{aligned} \mathcal{F}\left[v_+\right] &= \mathcal{F}\left[a_A\right] - i\,a_A && \text{(C.7)} \\ &= -i\left(a_A + i\,\mathcal{F}\left[a_A\right]\right) = -i \cdot v_+ && \text{(C.8)} \end{aligned}$$

$v_- := a_A - i\,\mathcal{F}\left[a_A\right]$

$$\begin{aligned} \mathcal{F}\left[v_-\right] &= \mathcal{F}\left[a_A\right] + i\,a_A && \text{(C.9)} \\ &= +i\left(a_A - i\,\mathcal{F}\left[a_A\right]\right) = +i \cdot v_- && \text{(C.10)} \end{aligned}$$

v_+ and v_- are antisymmetric.

u_+, u_-, v_+ and v_- are *eigenvectors* of the FT, with *eigenvalues* $+1$, -1, $-i$ and $+i$ respectively. The eigenvectors are pairwise perpendicular.

How to find sequences u_+, u_-, v_+, v_- and numbers ($\in \mathbb{C}$) α_+, α_-, β_+, β_- that for a given sequence a

$$\begin{aligned} a &= \alpha_+\, u_+ + \alpha_-\, u_- + \beta_+\, v_+ + \beta_-\, v_- \\ &\quad \text{where } \alpha_+^2 + \alpha_-^2 + \beta_+^2 + \beta_-^2 = 1 \end{aligned} \tag{C.11}$$

first compute a_S then with $a_S/2 = \alpha_+\, u_+ + \alpha_-\, u_-$ and $\mathcal{F}\left[a_S/2\right] = +1\, \alpha_+\, u_+ - 1\, \alpha_-\, u_-$ one has $1/4(a_S + \mathcal{F}\left[a_S\right]) = \alpha_+\, u_+$ and $1/4(a_S - \mathcal{F}\left[a_S\right]) = \alpha_-\, u_-$
Analogue with a_A for v_+, v_-, β_+ and β_-.

Thereby we can compute a transform that is the 'square root' of the FT: for some sequence a compute u_+, u_-, v_+, v_- and α_+, α_-, β_+, β_- as above then for $\lambda \in \mathbb{R}$ one can define a transform $\mathcal{F}^\lambda\left[a\right]$ as

$$\mathcal{F}^\lambda\left[a\right] = (+1)^\lambda\, \alpha_+\, u_+ + (-1)^\lambda\, \alpha_-\, u_- + (-i)^\lambda\, \beta_+\, v_+ + (+i)^\lambda\, \beta_-\, v_- \tag{C.12}$$

$\mathcal{F}^0\left[a\right]$ is identity

$\mathcal{F}^1\left[a\right]$ is the FT

$\mathcal{F}^{1/2}\left[a\right]$ (which is not unique) is a transform so that $\mathcal{F}^{1/2}\left[\mathcal{F}^{1/2}\left[a\right]\right] = \mathcal{F}\left[a\right]$.

Appendix D

The Chinese Remainder Theorem (CRT)

The Chinese remainder theorem (CRT):

Let $m_1, m_2, ..., m_f$ be pairwise relatively[1] prime (i.e. $gcd(m_i, m_j) = 1$, $\forall i \neq j$)
If $x \equiv x_i \ (mod\, m_i)$ $i = 1, 2, ..., f$ then x is unique modulo the product $m_1 \cdot m_2 \cdot ... \cdot m_f$.

For only two moduli m_1, m_2 compute x as follows[2]:

Code D.1 (CRT for two moduli) *pseudo code to find unique* $x \ (mod\, m_1\, m_2)$ *with* $x \equiv x_1 \ (mod\, m_1)$ $x \equiv x_2 \ (mod\, m_2)$*:*

```
function crt2(x1,m1,x2,m2)
{
    c := m1**(-1) mod m2    // inverse of m1 modulo m2
    s := ((x2-x1)*c) mod m2
    return  x1 + s*m1
}
```

For repeated CRT calculations with the same moduli one will use precomputed c.

For more more than two moduli use the above algorithm repeatedly.

Code D.2 (CRT) *Code to perform the CRT for several moduli:*

```
function crt(x[],m[],f)
{
    x1 := x[0]
    m1 := m[0]

    i := 1
    do
    {
        x2 := x[i]
        m2 := m[i]

        x1 := crt2(x1,m1,x2,m2)
        m1 := m1 * m2

        i := i + 1
    }
    while i<f

    return x1
}
```

[1] note that it is not assumed that any of the m_i is prime

[2] cf. [6]

To see why these functions really work we have to formulate a more general CRT procedure that specialises to the functions above.

Define

$$T_i \; := \; \prod_{k!=i} m_k \tag{D.1}$$

and

$$\eta_i \; := \; T_i^{-1} \mod m_i \tag{D.2}$$

then for

$$X_i \; := \; x_i \, \eta_i \, T_i \tag{D.3}$$

one has

$$X_i \mod m_j \; = \; \begin{cases} x_i & \text{for} \quad j = i \\ 0 & \text{else} \end{cases} \tag{D.4}$$

and so

$$\sum_k X_k \; = \; x_i \mod m_i \tag{D.5}$$

For the special case of two moduli m_1, m_2 one has

$$\begin{aligned} T_1 &= m_2 && \text{(D.6)} \\ T_2 &= m_1 && \text{(D.7)} \\ \eta_1 &= m_2^{-1} \bmod m_1 && \text{(D.8)} \\ \eta_2 &= m_1^{-1} \bmod m_2 && \text{(D.9)} \end{aligned}$$

which are related by[3]

$$\eta_1 \, m_2 + \eta_2 \, m_1 \; = \; 1 \tag{D.10}$$

$$\begin{aligned} \sum_k X_k &= x_1 \, \eta_1 \, T_1 + x_2 \, \eta_2 \, T_2 && \text{(D.11)} \\ &= x_1 \, \eta_1 \, m_2 + x_2 \, \eta_2 \, m_1 && \text{(D.12)} \\ &= x_1 \, (1 - \eta_2 \, m_1) + x_2 \, \eta_2 \, m_1 && \text{(D.13)} \\ &= x_1 + (x_2 - x_1) \, (m_1^{-1} \bmod m_2) \, m_1 && \text{(D.14)} \end{aligned}$$

as given in the code. The operation count of the CRT implementation as given above is significantly better than that of a straightforward implementation.

[3] cf. extended euclidean algorithm

Appendix E

A modular multiplication trick

The following trick allows easy multiplication of two integers a, b modulo some modulus m even if the product $a \cdot b$ doesn't fit into a machine integer (that is assumed to have some maximal value $z-1, z = 2^k$). Let $\langle x \rangle_y$ denote x modulo y, $\lfloor x \rfloor$ denote the integer part of x. For $0 \leq a, b < m$:

$$a \cdot b \quad = \quad \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m + \langle a \cdot b \rangle_m \tag{E.1}$$

rearranging and taking both sides modulo $z > m$:

$$\left\langle a \cdot b - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \quad = \quad \langle \langle a \cdot b \rangle_m \rangle_z \tag{E.2}$$

where the rhs. equals $\langle a \cdot b \rangle_m$ because $m < z$.

$$\langle a \cdot b \rangle_m \quad = \quad \left\langle \langle a \cdot b \rangle_z - \left\langle \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \right\rangle_z \tag{E.3}$$

the expression on the rhs. can be translated into a few lines fo C-code. The code given here assumes that one has 64 bit integer types `int64` (signed) and `uint64` (unsigned) and a floating point type with 64 bit mantissa, `float64` (typically `long double`).

```
uint64 mul_mod(uint64 a, uint64 b, uint64 m)
{
    uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);  // floor(a*b/m)
    y = y * m;           // m*floor(a*b/m) mod z
    uint64 x = a * b;    // a*b mod z
    uint64 r = x - y;    // a*b mod z - m*floor(a*b/m) mod z
    if ( (int64)r < 0 )  // normalisation needed ?
    {
        r = r + m;
        y = y - 1;       // (a*b)/m  quotient, omit line if not needed
    }
    return  r;           // (a*b)%m  remnant
}
```

It uses the fact that integer multiplication computes the least significant bits of the result $\langle a \cdot b \rangle_z$ whereas float multiplication computes the most significant bits of the result. The above routine works if $0 <= a, b < m < 2^{63} = \frac{z}{2}$. The normalisation isn't necessary if $m < 2^{62} = \frac{z}{4}$.

When working with a fixed modulus the division by `p` may be replaced by a multiplication with the inverse modulus, that only needs to be computed once:

```
float64 i = (float64)1/m;
```

the line

```
uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);
```

is the replaced by

```
uint64 y = (uint64)((float64)a*(float64)b*i+(float64)1/2);
```

so any division inside the routine avoided. But beware, the routine then cannot be used for $m >= 2^{62}$: it very rarely fails for moduli of more than 62 bits. This is due to the additional error when inverting and multiplying as compared to dividing alone.

This trick is ascribed to Peter Montgomery.

Bibliography

— BOOKS & THESIS —

[1] H.S.Wilf: Algorithms and Complexity, internet edition, 1994,
online at `ftp://ftp.cis.upenn.edu/pub/wilf/AlgComp.ps.Z`

[2] P.Duhamel, ed.: Papers on the Fast Fourier Transform, IEEE Press, New York 1995

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

[4] J.McClellan, C.Rader: Number Theory in Digital Signal Processing, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979.

[5] D.Myers: Digital Signal Processing, Efficient Convolution and Fourier Transform Techniques, Prentice-Hall, 1990

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

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

[8] 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

[9] R.Tolimieri, M.An, C.Lu: Algorithms for Discrete Fourier Transform and Convolution, Springer 1997 (second edition)

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

[11] W.Briggs, V.Henson: The DFT: An Owner's Manual for the Discrete Fourier Transform, Philadelphia: SIAM, 1995

[12] W.Smith, J.Smith: Handbook of Real-Time Fast Fourier Transforms, New York: IEEE Press, 1995

[13] J.Lim, A.Oppenheim: Advanced Topics in Signal Processing, ch. 4. Prentice-Hall, 1988

[14] H.Wesnikoff, R.Wells jr.: Wavelet Analysis, Springer 1998

[15] R.Crandall: Projects in Scientific Computation, Springer/TELOS 1994

[16] R.Crandall: Topics in Advanced Scientific Computation, Springer/TELOS 1996

[17] M.Heideman: Muliplicative Complexity, Convolution and the DFT, Springer

[18] D.E.Knuth: The Art of Computer Programming, 2.edition, Volume 2: Seminumerical Algorithms, Addison-Wesley 1981,
online errata list at `http://www-cs-staff.stanford.edu/~knuth/`

[19] J.Mendel: Maximum Likelihood Deconvolution, Springer

[20] R.Blahut: Algebraic Methods for Signal Processing and Communications Coding, Springer

[21] R.Tolimieri, M.An, C.Lu: Mathematics of Multidimensional Fourier Transform Algorithms, Springer

[22] R.Bucy: Lectures on Discrete Time Filtering, Springer

[23] C.Burrus, T.Parks: DFT/FFT and Convolution Algorithms, Wiley 1985

[24] P.Besslich, L.Tian: Diskrete Orthogonaltransformationen, Springer 1990

[25] 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 !

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

[27] 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.

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

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

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

[31] B.Fino: Recursive definition and computation of fast unitary transforms, Ph.D. dissertation, Univ. California, Berkeley, Nov. 1973

— PAPERS —

[32] C.Rader: Discrete Fourier Transforms When the Number of Data Samples is Prime, Proc. IEEE 56, 1968 pp.1107-1108

[33] J.Johnson, R.Johnson, D.Rodriguez, R.Tolimieri: A Methodology for Designing, Modifying and Implementing Fourier Transform Algorithms on Various Architectures, IEEE Trans. Circuits Sys. 9, 1990

[34] C.Temperton: Self-Sorting Mixed Radix Fast Fourier Transforms, J. ACM 10, 1967 pp.647-654

[35] C.Temperton: Implementation of a Self-Sorting In-Place Prime Factor FFT Algorithm, J. Comp. Physics 58, 1985 pp.283-299

[36] C.Temperton: A Note on a Prime Factor FFT, J. Comp. Physics 52, 1983 pp.198-204

[37] C.Burrus, P.Eschenbacher: An In-place IN-order Prime factor FFT Algorithm, IEEE Trans. on Acoustics, Speech and Signal Processing, 29, Aug. 1981 pp.806-817

[38] D.Kolba, T.Parks: A Prime factor FFT Algorithm Using High-speed Convolution, IEEE Trans. on Acoustics, Speech and Signal Processing, 25, Aug. 1977 pp.281-294

[39] S.Chu, C.Burrus: A Prime Factor FFT Algorithm Using Distributed Arithmetic, IEEE Trans. on Acoustics, Speech and Signal Processing, 30, April 1982 pp.217-227

[40] J.Cooley, O.Tukey: An Algorithm for the Machine Calculation of Complex Fourier Series, Math. Comp. 19 pp.297-301, 1965

[41] G.Duhamel, M.Vetterli: Fast Fourier Transforms: A Tutorial Review, Signal Processing 19 pp.259-299, 1990

[42] G.Strang: Wavelet Transforms Versus Fourier Transforms, Bull. Amer. Math. Soc. 28 pp.288-305, 1993

[43] A.Saidi: Decimation-in-time-frequency FFT algorithm, Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, (IEEE ICASSP-94, Adelaide, Australia), pp.III:453-456, Apr.1994

[44] H.Guo, G.Sitton, C.Burrus: The quick discrete Fourier transform, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, (IEEE ICASSP-94, Adelaide, Australia), pp.III:445-448, Apr.1994

[45] H.Guo, G.Sitton, C.Burrus: The quick Fourier transform, an FFT based on symmetries, IEEE Transactions on Signal Processing, submitted Oct. 1994

[46] H.Sorensen, D.Jones, M.Heideman, C.Burrus: Real-Valued Fast Fourier Transform algorithms, IEEE Trans. on Acoustics, Speech and Signal Processing, Vol ASSP-35, no.6 pp.849-863, 1987

[47] R.Crochiere, L.Rabiner Interpolation and Decimation of Digital signals - A tutorial Review, Proc. of the IEEE, Vol 69, no.3 pp.300-331, 1981

[48] M.Heideman, D.Johnson, C.Burrus: Gauss and the history of the fast Fourier transformation, IEEE ASSP Magazine 1 pp.14-21, 1984

[49] P.Duhamel, H.Hollmann: Split radix FFT algorithm, Electronis Letters 20 pp.14-16, 1984

[50] P.Duhamel: Implementation of 'split-radix' FFT algorithms for complex, real and real-symmetric data, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-34 pp.285-295, 1986

[51] H.Sorensen, M.Heideman, C.Burrus: Oncomputing the split-radix FFT, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-34 152-156, 1986

[52] S.Winograd: On computing the discrete Fourier transform, Math. of Comp. 32, Jan. 1978 pp.175-199

[53] C.Lu, R.Tolimieri: Extension of Winograd Multiplicative Algorithm to Transform size $N = p^2q, p^2qr$ and Their Implementation, Proc. ICASSP 89, 19(D.3), Scotland

[54] R.Tolimieri, C.Lu, W.Johnson: Modified Winograd FFT Algorithm and Its Variants for Transform Size $N = p^n$ and Their Implementations, Advances in apllied Mathematics, 10, 1989 pp.228-251

[55] H.Silverman: An introduction to programming the Winograd Fourier transform algorithm (WFTA), IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-25 pp.152-164, 1977

[56] L.Auslander, E.Feig, S.Winograd: The Multiplicative Complexity of the Discrete Fourier Transform, Adv. in Appl. Math. 5, 1984 pp.87-109

[57] Y.Tadokoro, T.Higuchi: Discrete Fourier transform computation via the Walsh transform, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-26 pp.236-240, 1978

[58] Y.Tadokoro, T.Higuchi: Comments on "Discrete Fourier transform computation via the Walsh transform", ASSP-27 pp.295-296, 1979

[59] Y.Tadokoro, T.Higuchi: Another discrete Fourier transform computation with small multiplications via the Walsh transform, ICASSP'81 Proceedings of the 1981 IEEE International Conference on Acoustics, Speech and Signal Processing, 1 pp.308-309

[60] R.Storn: Fast algorithms for the discrete Hartley transform, Archiv für Elektronik & Übertragungstechnik 40 pp.233-240, 1986

[61] S.Pei, J.Wu: Split-radix fast Hartley transform, Electronics Letters 22 pp.26-27, 1986

[62] H.Hou: The fast Hartley transform algorithm, IEEE Trans. Comp. C-36 pp.147-156, Feb.1987

[63] H.Hou: Correction to: The fast Hartley transform algorithm, IEEE Trans. Comp. C-36 pp.1135-1136, 1987

[64] H.Sorensen, D.Jones, C.Burrus, M.Heideman: On computing the discrete Hartley transform, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-33 pp.1231-1238, Oct.1985

[65] H.Meckelburg, D.Lipka: Fast Hartley transform algorithm, Electronics Letters 21 pp.341-343, 1985

[66] C.Hsu, J.Wu: Fast computation of the discrete Hartley transform via Walsh-Hadamard transform, Electronics Letters 23 pp.466-468, 1987

[67] C.Hsu, J.Wu: The Walsh-Hadamard /discrete Hartley transform, Int. J. Electronics 62 pp.744-755, 1987

[68] J.Fine: On the Walsh Functions, Transactions of the American Math. Soc. vol.65 pp.372-414, 1949

[69] J.Fine: The generalised Walsh-Functions, Transactions of the American Math. Soc. vol.69 pp.66-77, 1950

[70] O.Buneman: Conversion of FFT's to fast hartley transforms, SIAM J. Sci. Stat. Comput. pp.624-639, 1986

[71] H.Malvar: Fast computation of the discrete cosine transform through fast Hartley transform, Electronics Letters 22 pp.352-353, 1986

[72] H.Malvar: Fast Computation of the discrete cosine transform and the discrete Hartley transform, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-35 pp.1484-1485, 1987

[73] Z.Mou, P.Duhamel: In-place butterfly style FFT of 2-D real sequences, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-36 pp.1642-1650, 1988

[74] R.Bracewell, O.Buneman, H.Hao, J.Villasenor: fast two-dimensional Hartley transform, Proc. IEEE 74 pp.1282-1283, 1986

[75] R.Kumaresan, P.Gupta: Vector radix algorithm for a 2-D discrete Hartley transform, Proc. IEEE 74 pp.755-757, 1986

[76] M.Haque: A two-dimensional fast cosine transform, IEEE Trans. on Acoustics, Speech and Signal Processing, ASSP-33 pp.1532-1539, 1985

[77] R.Crandall, B.Fagin: Discrete Weighted Transforms and Large Integer Arithmetic, Math. Comp. (62) 1994 pp.305-324

[78] P.Roeser, M.Jernigan: Fast Haar transform algorithms, IEEE Trans. Comput. C-31 pp.175-177, 1982

[79] Z.Wang: New algorithm for the slant transform, IEEE Trans. Pattern Anal. Mach. Intell. PAMI-4 pp.551-555, 1982

[80] Z.Wang: A fast algorithm for the discrete sine transform implemented through the fast cosine transform, IEEE Trans. on Acoustics, Speech and Signal Processing, vol. 30, pp.814-815, 1982

[81] B.Fino, V.Algazi: Slant-Haar transform, Proc. IEEE 62 pp.653-654, 1974

[82] B.Fino, V.Algazi: A unified treatment for fast unitary transforms, SIAM J.Comput.,vol.6 no.4, pp.700-717, 1977

[83] H.Jones, D.Hein, S.Knauer: The Karhunen-Loeŷe, discrete cosine, and related transforms obtained via the hadamard transform, presented at the Int. Telemetring Conf. Nov.1978

[84] Z.Wang: Fast algorithms for the discrete W transform and the discrete fourier transform, IEEE Trans., Acoust., Speech, Signal Processing, ASSP-32 pp.803-816, Aug.1984

[85] J.Martens: Recursive cyclotomic factorization - a new algorithm for calculating the discrete Fourier transform, IEEE Trans. on ASSP, vol.32, pp.750-762, Aug.1984

[86] M.Vetterli, H.Nussbaumer: imple FFT and DCT algorithms with reduced number of operations, Signal Processing, vol.6, pp.267-278, Aug.1984

[87] M.Vetterli, P.Duhamel: Split-radix algorithms for length - pm DFT's, IEEE Trans. on ASSP, vol.37, pp.57-64, Jan.1989. Also in ICASSP-88 Proceedings, pp.1415-1418, Apr.1988

[88] R.Stasinski: The techniques of the generalized fast Fourier transform algorithm, IEEE Transactions on Signal Processing, vol.39, pp.1058-1069, May 1991

[89] M.Heideman, C.Burrus: On the number of multiplications necessary to compute a length-2n DFT, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.34, pp.91-95, Feb.1986

[90] J.Beard: An in-place, self-reordering FFT, Proceedings of the ICASSP-78, (Tulsa), pp.632-633, Apr.1978.

[91] H.Johnson, C.Burrus: An in-place, in-order radix-2 FFT, in ICASSP-84 Proceedings, p.28A.2, Mar.1984

[92] C.Burrus: Unscrambling for fast DFT algorithms, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.36, pp.1086-1087, Jul.1988

[93] P.Rösel: Timing of some bit reversal algorithms, Signal Processing, vol.18, pp.425-433, Dec.1989

[94] J.Jeong, W.Williams: A fast recursive bit-reversal algorithm, in Proceedings of the ICASSP-90, (Albuquerque, NM), pp.1511-1514, Apr.1990

[95] D.Evans: A second improved digit-reversal permutation algorithm for fast transforms, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.37, pp.1288-1291, Aug.1989

[96] J.Rodriguez: An improved FFT digit-reversal algorithm, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.37, pp.1298-1300, Aug.1989

[97] J.Walker: A new bit reversal algorithm, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.38, pp.1472-1473, Aug.1990

[98] A.Yong: A better FFT bit-reversal algorithm without tables, IEEE Transactions on Signal Processing, vol.39, pp.2365-2367, Oct.1991

[99] D.Sundararajan, M.Ahamad, M.Swamy: A fast FFT bit-reversal algorithm, IEEE Transactions on Circuits and Systems, II, vol.41, pp.701-703, Oct.1994

[100] J.Rius, R.De Porrata-Doria: New FFT bit-reversal algorithm, IEEE Transactions on Signal Processing, vol.43, pp.991-994, Apr.1995

[101] C.Temperton: Nesting strategies for prime factor FFT algorithms, Journal of Computational Physics, vol.82, pp.247-268, Jun.1989

[102] C.Temperton: A generalized prime factor FFT algorithm for any n = 2p 3q5r, SIAM Journal of Sci. Stat. Comp., 1992

[103] R.Stasinski: Prime factor DFT algorithms for new small-N DFT modules, IEEE Proceedings, Part G, vol.134, no.3, pp.117-126, 1987

[104] H.Johnson, C.Burrus: The design of optimal DFT algorithms using dynamic programming, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.31, pp.378-387, Apr.1983

[105] H.Johnson, C.Burrus: On the structure of efficient DFT algorithms, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.33, pp.248-254, Feb.1985

[106] H.Johnson, C.Burrus: Large DFT modules: N = 11, 13, 17, 19, and 25, Tech. Rep. 8105, Department of Electrical Engineering, Rice University, Houston, TX 77251-1892, 1981

[107] C.Temperton: A new set of minimum-add small-n rotated DFT modules, Journal of Computational Physics, vol.75, pp.190-198, 1988

[108] F.Wang, P.Yip: Fast prime factor decomposition algorithms for a family of discrete trigonometric transforms, Circuits, Systems, and Signal Processing, vol.8, no.4, pp.401-419, 1989

[109] P.Duhamel, M.Vetterli: Improved Fourier and Hartley transfrom algorithms, application to cyclic convolution of real data, IEEE Trans. on ASSP, vol.35, pp.818-824, Jun.1987

[110] M.Popovic, D.Sevic: A new look at the comparison of the fast Hartley and Fourier transforms, IEEE Transactions on Signal Processing, vol.42, pp.2178-2182, Aug.1994

[111] P.Uniyal: Transforming real-valued sequences: fast Fourier versus fast Hartley transform algorithms, IEEE Transactions on Signal Processing, vol.42, pp.3249-3254, Nov.1994

[112] G.Bruun: Z-transform DFT filters and FFTs, IEEE Transactions on ASSP, vol.26, pp.56-63, Feb.1978

[113] R.Storn: On the Bruun algorithm and its inverse, Frequenz, vol.46, pp.110-116, 1992

[114] C.Rader, N.Brenner A new principle for fast Fourier transformation, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol.ASSP-24, pp.264-266, Jun.1976

[115] K.Cho, G.Temes: Real-factor FFT algorithms, in Proceedings of IEEE ICASSP-78, (Tulsa, OK), pp.634-637, Apr.1978

[116] J.Glassman: A generalization of the fast Fourier transform, IEEE Transactions on Computers, vol.C-19, pp.105-116, Feb.1970

[117] W.Ferguson, Jr.: A simple derivation of Glassman general-n fast Fourier transform, Comput. and Math. with Appls., vol.8, no.6, pp.401-411, 1982. Also, in Report AD-A083811, NTIS, Dec.1979

[118] L.Rabiner, R.Schafer, C.Rader: The chirp z-transform algorithm, IEEE Transactions on Audio Electroacoustics, vol.AU-17, pp.86-92, Jun.1969

[119] I.Selesnick, C.Burrus: Multidimensional mapping techniques for convolution, in Proceedings of the IEEE International Conference on Signal Processing, (IEEE ICASSP-93, Minneapolis), pp.III-288-291, Apr.1993

[120] I.Selesnick, C.Burrus: Automating the design of prime length FFT programs, in Proceedings of the IEEE International Symposium on Circuits and Systems, (ISCAS-92, San Diego, CA), pp.133-136, May 1992

[121] I.Selesnick, C.Burrus: Automatic generation of prime length FFT programs, IEEE Transactions on Signal Processing, 1995

[122] W.Hocking: Performing Fourier transforms on extremely long data streams, Computers in Physics, vol.3, pp.59-65, Jan.1989

[123] R.Agarwal, C.Burrus: Number theoretic transforms to implement fast digital convolution, Proceedings of the IEEE, vol.63, pp.550-560, Apr.1975. Also in IEEE Press DSP Reprints II, 1979

[124] H.Sorensen, C.Burrus, D.Jones: A new efficient algorithm for computing a few DFT points, in Proceedings of the IEEE International Symposium on Circuits and Systems, (Espoo, Finland), pp.1915-1918, Jun.1988

[125] C.Roche: A split-radix partial input/output fast Fourier transform algorithm, IEEE Transactions on Signal Processing, vol.40, pp.1273-1276, May 1992

[126] H.Sorensen, C.Burrus: Efficient computation of the DFT with only a subset of input or output points, IEEE Transactions on Signal Processing, vol.41, pp.1184-1200, Mar.1993

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

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

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

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

Index