Algorithms for programmers
ideas and source code

This document is work in progress: read the "important remarks" near the beginning

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at March 26, 2003

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

Contents

Some important remarks

. . . about this document.

This is a draft of what is intended to turn into a book about selected algorithms. The audience in mind are programmers who are interested in the treated algorithms and actually want to create and understand working and reasonably optimized code.

The printable full version will always stay online for free download. The referenced sources are online as part of FXT (fast transforms and low level routines) or hfloat (arithmetical algorithms).

The reader is welcome to criticize and make suggestions. Thanks go to those[1] who helped to improve this document so far! Thanks also to the people who share their ideas or source code on the net. I try to give due references to original sources and authors wherever I can. However, I am in no way an expert for history of algorithms and I pretty sure will never be one. So if you feel that a reference is missing somewhere, let me know.

New sections appear as soon as they contain anything useful, sometimes just listings or remarks outlining what is to appear. A "TBD: *something to be done*" is a reminder to myself to fill in something that is missing or would be nice to have.

The style varies from chapter to chapter which I do not consider bad per se: while some topics (as fast Fourier transforms) need a clear and explicit introduction others (like the bit wizardry chapter) seem to be best presented by basically showing the code with just a few comments. Still other parts (like the chapter about sorting and searching) are presented elsewhere extremely well so I only the basic ideas are introduced shortly (accompanied by code) and references for further studies are given.

The pseudo language Sprache is used when I see a clear advantage to do so, mainly when the corresponding C++ does not appear to be self explanatory. Larger pieces of code are presented in C++. A tiny starter about C++ (some good reasons in favor of C++ and some of the very basics of classes/overloading/templates) might be included. C programmers do not need to be shocked by the '++': only an rather minimal set of the C++ features is used.

Enjoy reading !

[1] in particular André Piotrowski and Edith Parzefall.

"Why make things difficult, when it is possible to make them cryptic and totally illogic, with just a little bit more effort?"

– Aksel Peter Jørgensen

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

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

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

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

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

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

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

To see this, consider element y of the IFT of the FT of a:

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

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

As $\sum_k \left(z^{x-y}\right)^k = n$ 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

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

Here we will call the FT with the plus in the exponent the forward transform. The choice is actually 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 normalization 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 straight forward implementation of the discrete Fourier transform, i.e. the computation of n sums each of length n requires $\sim n^2$ operations:

```
void slow_ft(Complex *f, long n, int is)
{
    Complex h[n];
    const double ph0 = is*2.0*M_PI/n;
    for (long w=0; w<n; ++w)
    {
        Complex t = 0.0;
        for (long k=0; k<n; ++k)
        {
            t +=  f[k] * SinCos(ph0*k*w);
        }
        h[w] = t;
    }
    copy(h, f, n);
}
```

[FXT: `slow_ft` in `slow/slowft.cc`] `is` must be $+1$ (forward transform) or -1 (backward transform), `SinCos(x)` returns a `Complex(cos(x), sin(x))`.

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 $n/2\,\log_2(n)$ (complex) multiplications suffice. There are several different FFT algorithms with many variants.

1.2 Symmetries of the Fourier transform

A bit of notation turns out to be useful:
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:

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

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

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

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

1.3 Summary of definitions of Fourier transforms *

This section summarizes the definitions of the continuous, semi-continuous and the discrete Fourier transform.

The continuous Fourier transform

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

$$F(\vec{\omega}) := \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{C}^n} f(\vec{x})\, e^{\sigma\, i\, \vec{x}\, \vec{\omega}} d^n x \tag{1.25}$$

where $\sigma = \pm 1$. The FT is is a unitary transform.

Its inverse ('back-transform') is

$$f(\vec{x}) = \frac{1}{\sqrt{2\,\pi}^n} \int_{\mathbb{C}^n} F(\vec{\omega})\, e^{-\sigma\, \vec{x}\, \vec{\omega}} d^n \omega \tag{1.26}$$

i.e. the complex conjugate transform.

For the 1-dimensional case one has

$$F(\omega) = \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} f(x) e^{\sigma\, x\, \omega} dx \tag{1.27}$$
$$f(x) = \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} F(\omega)\, e^{-\sigma\, x\, \omega} d\omega \tag{1.28}$$

The 'frequency'-form is

$$\hat{f}(\nu) = \int_{-\infty}^{+\infty} f(x) e^{\sigma\, 2\,\pi\, i\, x\, \nu} dx \tag{1.29}$$
$$f(x) = \int_{-\infty}^{+\infty} \hat{f}(\nu)\, e^{-\sigma\, 2\,\pi\, i\, x\, \nu} d\nu \tag{1.30}$$

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*:

$$c_k := \frac{1}{\sqrt{L}} \int_L f(x)\, e^{\sigma\, 2\,\pi\, i\, k\, x/L} dx \tag{1.31}$$

Then

$$\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k\, e^{-\sigma\, 2\,\pi\, i\, k\, x/L} = \begin{cases} f(x) & \text{if } f \text{ continuous at } x \\ \frac{f(x+0)+f(x-0)}{2} & \text{else} \end{cases} \tag{1.32}$$

Another (equivalent) form is given by

$$a_k := \frac{1}{\sqrt{L}} \int_L f(x)\, \cos \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 0, 1, 2, \ldots \tag{1.33}$$

$$b_k := \frac{1}{\sqrt{L}} \int_L f(x)\, \sin \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 1, 2, \ldots \tag{1.34}$$

$$f(x) = \frac{1}{\sqrt{L}} \left[\frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k\, \cos \frac{2\,\pi\, k\, x}{L} + b_k\, \sin \frac{2\,\pi\, k\, x}{L} \right) \right] \tag{1.35}$$

with

$$c_k = \begin{cases} \frac{a_0}{2} & (k = 0) \\ \frac{1}{2}(a_k - i b_k) & (k > 0) \\ \frac{1}{2}(a_k + i b_k) & (k < 0) \end{cases} \tag{1.36}$$

The discrete Fourier transform

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

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x\, e^{\sigma\, 2\,\pi\, i\, x\, k/n} \tag{1.37}$$

Back-transform is

$$f_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k\, e^{\sigma\, 2\,\pi\, i\, x\, k/n} \tag{1.38}$$

1.4 Radix 2 FFT algorithms

1.4.1 A little bit of notation

Always assume a is a length-n sequence (n a power of two) 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 $0 \ldots n/2 - 1$.

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

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.

1.4.2 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

$$\sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{(2\,x+1)\,k} \tag{1.39}$$

$$= \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{2\,x\,k} \tag{1.40}$$

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

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 $0 \leq k < n/2$ and $n/2 \leq k < n$, therefore we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = j + \delta\,\frac{n}{2}$ where $j \in \{0, 1, \ldots, n/2-1\}$, $\quad \delta \in \{0, 1\}$.

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(j+\delta\,\frac{n}{2})} = \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} + z^{j+\delta\,\frac{n}{2}} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} \tag{1.41}$$

$$= \begin{cases} \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} + z^j \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} - z^j \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{1.42}$$

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:*

$$\mathcal{F}\,[a]^{(left)} \overset{n/2}{=} \mathcal{F}\left[a^{(even)}\right] + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \tag{1.43}$$

$$\mathcal{F}\,[a]^{(right)} \overset{n/2}{=} \mathcal{F}\left[a^{(even)}\right] - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \tag{1.44}$$

(Here it is silently assumed that '+' or '−' between two sequences denotes element-wise 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 $n \cdot \log_2(n)$: There are $\log_2(n)$ splitting steps, the work in each step is proportional to n.

Code 1.1 (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
```

[2] also called Cooley-Tukey FFT.

```
    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 normalization (i.e. multiplication of each element of `x[]` by $1/\sqrt{n}$) is not included here.

[FXT: `recursive_fft_dit2` in `slow/recfft2.cc`] The procedure uses the subroutine

Code 1.2 (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 `fft/fouriershift.cc`]

The recursive FFT-procedure involves $n \log_2(n)$ function calls, which can be avoided by rewriting it in a non-recursive way. One can even do all operations *in-place*, no temporary workspace 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 7.1.

Code 1.3 (radix 2 DIT FFT, localized) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* `is` *must be -1 or +1:*

```
procedure fft_dit2_localized(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: `fft_localized_dit2` in `fft/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 $n/2 \cdot \log_2(n)$ times. To reduce the number of trigonometric computations, one can simply swap the two inner loops, leading to the first 'real world' FFT procedure presented here:

Code 1.4 (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: `fft_dit2` in `fft/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 localized FFT is slower by a factor of about 0.66, but for large `n` the same ratio is in favor 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
    {
```

1.4.3 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]:

$$\sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n} a_x\, z^{x\,k} \tag{1.45}$$

$$= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=0}^{n/2-1} a_{x+n/2}\, z^{(x+n/2)\,k} \tag{1.46}$$

$$= \sum_{x=0}^{n/2-1} (a_x^{(left)} + z^{k\,n/2}\, a_x^{(right)})\, z^{x\,k} \tag{1.47}$$

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

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

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(2\,j+\delta)} = \sum_{x=0}^{n/2-1} (a_x^{(left)} + z^{(2\,j+\delta)\,n/2}\, a_x^{(right)})\, z^{x\,(2\,j+\delta)} \tag{1.48}$$

$$= \begin{cases} \sum_{x=0}^{n/2-1} (a_x^{(left)} + a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum_{x=0}^{n/2-1} z^x (a_x^{(left)} - a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{1.49}$$

$z^{(2\,j+\delta)\,n/2} = e^{\pm\pi\,i\,\delta}$ 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:*

$$\mathcal{F}\,[a]^{(even)} \stackrel{n/2}{=} \mathcal{F}\left[a^{(left)} + a^{(right)}\right] \tag{1.50}$$

$$\mathcal{F}\,[a]^{(odd)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{1.51}$$

Code 1.5 (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
    {
```

[3] also called Sande-Tukey FFT, cf. [18].

```
        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_fft_dif2` in `slow/recfft2.cc`]

The non-recursive procedure looks like this:

Code 1.6 (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: `fft_dif2` in `fft/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 1.4 the inner loops where swapped to save 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)`.

TBD: *extraction of the j=0 case*

1.5 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.

1.5.1 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 can introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a notable slowdown is possible. However, for a length-n FFT one does not need to store all the (n complex or $2\,n$ real) sin/cos-values $\exp(2\,\pi\,i\,k/n),\ k = 0, 1, 2, 3, \ldots, n-1$. Already a table $\cos(2\,\pi\,i\,k/n),\ k = 0, 1, 2, 3, \ldots, n/4-1$ (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

$$\cos(\pi + x) = -\cos(x) \tag{1.52}$$
$$\sin(\pi + x) = -\sin(x) \tag{1.53}$$

(reducing the interval from $0 \ldots 2\pi$ to $0 \ldots \pi$),

$$\cos(\pi/2 + x) = -\sin(x) \tag{1.54}$$
$$\sin(\pi/2 + x) = +\cos(x) \tag{1.55}$$

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

$$\sin(x) = \cos(\pi/2 - x) \tag{1.56}$$

(only cos()-table needed).

1.5.2 Recursive generation of the *sin/cos*-values

In the computation of FFTs one typically needs the values

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

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

$$c = \cos\omega, \tag{1.57}$$
$$s = \sin\omega, \tag{1.58}$$
$$\alpha = 1 - \cos\delta \qquad \text{cancellation!} \tag{1.59}$$
$$= 2\,(\sin\frac{\delta}{2})^2 \qquad \text{ok.} \tag{1.60}$$
$$\beta = \sin\delta \tag{1.61}$$

Then compute the next power from the previous as:

$$c_{next} = c - (\alpha\, c + \beta\, s); \tag{1.62}$$
$$s_{next} = s - (\alpha\, s - \beta\, c); \tag{1.63}$$

(The underlying idea is to use (with $e(x) := \exp(2\,\pi\, i\, x)$) the ansatz $e(\omega+\delta) = e(\omega) - e(\omega)\cdot z$ which leads to $z = 1 - \cos\delta - i\,\sin\delta = 2\,(\sin\frac{\delta}{2})^2 - i\,\sin\delta$.)

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 `double`s it might be a good idea to use the type `long double` with the trig recursion: the sin and cos will then always be accurate within the `double`-precision.

A real-world example from [FXT: `fht_dif_core` in `fht/fhtdif.cc`], the recursion is used if `TRIG_REC` is `#defined`:

```
        [...]
        double tt = M_PI_4/kh;
#if defined TRIG_REC
        double s1 = 0.0,  c1 = 1.0;
        double al = sin(0.5*tt);
        al *= (2.0*al);
        double be = sin(tt);
#endif // TRIG_REC
        for (ulong i=1; i<kh; i++)
        {
#if defined TRIG_REC
            c1 -= (al*(tt=c1)+be*s1);
            s1 -= (al*s1-be*tt);
#else
            double s1, c1;
            SinCos(tt*i, &s1, &c1);
#endif // TRIG_REC
            [...]
```

1.5.3 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. (Less bookkeeping overhead, less loads/stores.)

1.6 Higher radix DIT and DIF algorithms

1.6.1 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 $a^{(even)}$, $a^{(3\%4)} = \{a_3, a_7, a_{11}, a_{15}, \dots\}$. 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 $\frac{r\,n}{m} \dots \frac{(r+1)\,n}{m} - 1$; 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.

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

1.6.2 Decimation in time

First reformulate the radix 2 DIT step (formulas 1.43 and 1.44) in the new notation:

$$\mathcal{F}\left[a\right]^{(0/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right] + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right] \tag{1.64}$$

$$\mathcal{F}\left[a\right]^{(1/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right] - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right] \tag{1.65}$$

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

The radix 4 step, whose derivation is analogous to the radix 2 step, it just involves more writing and does not give additional insights, is

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

$$\mathcal{F}\left[a\right]^{(0/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.66}$$

$$\mathcal{F}\left[a\right]^{(1/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.67}$$

$$\mathcal{F}\left[a\right]^{(2/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.68}$$

$$\mathcal{F}\left[a\right]^{(3/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.69}$$

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 compactly:

$$\begin{aligned}\mathcal{F}\left[a\right]^{(j/4)} \overset{n/4}{=} &+e^{\sigma\, 2\, i\, \pi\, 0\, j/4}\cdot\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + e^{\sigma\, 2\, i\, \pi\, 1\, j/4}\cdot\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] \\ &+e^{\sigma\, 2\, i\, \pi\, 2\, j/4}\cdot\mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + e^{\sigma\, 2\, i\, \pi\, 3\, j/4}\cdot\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.70}$$

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

Still more compactly:

$$\mathcal{F}\left[a\right]^{(j/4)} \overset{n/4}{=} \sum_{k=0}^{3} e^{\sigma 2\, i\, \pi\, k\, j/4}\cdot\mathcal{S}^{\sigma k/4}\mathcal{F}\left[a^{(k\%4)}\right] \qquad j = 0, 1, 2, 3 \tag{1.71}$$

where the summation symbol denotes *element-wise* 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:

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

$$\mathcal{F}\left[a\right]^{(j/r)} \overset{n/r}{=} \sum_{k=0}^{r-1} e^{\sigma\, 2\, i\, \pi\, k\, j/r}\cdot\mathcal{S}^{\sigma\, k/r}\mathcal{F}\left[a^{(k\%r)}\right] \qquad j = 0, 1, 2, \ldots, r-1 \tag{1.72}$$

1.6.3 Decimation in frequency

The radix 2 DIF step (formulas 1.50 and 1.51) was

$$\mathcal{F}\left[a\right]^{(0\%2)} \overset{n/2}{=} \mathcal{F}\left[\mathcal{S}^{0/2}\left(a^{(0/2)} + a^{(1/2)}\right)\right] \tag{1.73}$$

$$\mathcal{F}\left[a\right]^{(1\%2)} \overset{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(0/2)} - a^{(1/2)}\right)\right] \tag{1.74}$$

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:*

$$\mathcal{F}\,[a]^{(0\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{0/4}\Big(a^{(0/4)} + a^{(1/4)} + a^{(2/4)} + a^{(3/4)}\Big)\right] \tag{1.75}$$

$$\mathcal{F}\,[a]^{(1\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{1/4}\Big(a^{(0/4)} + i\,\sigma\,a^{(1/4)} - a^{(2/4)} - i\,\sigma\,a^{(3/4)}\Big)\right] \tag{1.76}$$

$$\mathcal{F}\,[a]^{(2\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{2/4}\Big(a^{(0/4)} - a^{(1/4)} + a^{(2/4)} - a^{(3/4)}\Big)\right] \tag{1.77}$$

$$\mathcal{F}\,[a]^{(3\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{3/4}\Big(a^{(0/4)} - i\,\sigma\,a^{(1/4)} - a^{(2/4)} + i\,\sigma\,a^{(3/4)}\Big)\right] \tag{1.78}$$

Or, more compactly:

$$\mathcal{F}\,[a]^{(j\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\,j/4}\sum_{k=0}^{3} e^{\sigma\,2\,i\,\pi\,k\,j/4}\cdot a^{(k/4)}\right] \qquad j = 0,1,2,3 \tag{1.79}$$

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:*

$$\mathcal{F}\,[a]^{(j\%r)} \stackrel{n/r}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\,j/r}\sum_{k=0}^{r-1} e^{\sigma\,2\,i\,\pi\,k\,j/r}\cdot a^{(k/r)}\right] \qquad j = 0,1,2,\ldots,r-1 \tag{1.80}$$

1.6.4 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 (see section 7.2).

Code 1.7 (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.

With $r = p^x$ there is a pitfall: 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), 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 would not 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 1.8 (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
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];
```

[5] cf. section 4.3.

```
            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;
            }
        }
    }
}
```

[FXT: `fft_dit4l` in `fft/fftdit4l.cc`]

[FXT: `fft_dit4` and `fft_dit4_core` in `fft/fftdit4.cc`]

Code 1.9 (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)
}
```

[FXT: `fft_dif4l` in `fft/fftdif4l.cc`]

[FXT: `fft_dif4` and `fft_dif4_core` in `fft/fftdif4.cc`]

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 1.8 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.

The `radix_permute()` procedure is given in section 7.2 on page 126.

1.7 Split radix Fourier transforms (SRFT)

The idea underlying the *split radix FFT* is to use both radix-2 and radix-4 decompositions at the same time.

From the radix-2 (DIF) decomposition (relations 1.73 and 1.74) we use the first, the one for the even indices. For the odd indices we use the radix-4 splitting (relations 1.76 and 1.78, slightly reordered).

Idea 1.7 (split radix 4/2 DIF step) *Radix 4 decimation in frequency step for the split radix FFT:*

$$\mathcal{F}\,[a]^{(0\%2)} \stackrel{n/2}{=} \mathcal{F}\left[\left(a^{(0/2)} + a^{(1/2)}\right)\right] \tag{1.81}$$

$$\mathcal{F}\,[a]^{(1\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{1/4}\left(\left(a^{(0/4)} - a^{(2/4)}\right) + i\,\sigma\,\left(a^{(1/4)} - a^{(3/4)}\right)\right)\right] \tag{1.82}$$

$$\mathcal{F}\,[a]^{(3\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{3/4}\left(\left(a^{(0/4)} - a^{(2/4)}\right) - i\,\sigma\,\left(a^{(1/4)} - a^{(3/4)}\right)\right)\right] \tag{1.83}$$

Now we have expressed the length-$N = 2^n$ FFT as one length-$N/2$ and two length-$N/4$ FFTs. Note that $\mathcal{S}^{3/4} = \mathcal{S}^{-1/4}$ which means a saving in the trigonometric computations. The nice feature is that the operation count of the split radix FFT is actually lower than that of the radix-4 FFT.

Using our nice notation it is almost trivial to write down the DIT version of the algorithm:

Idea 1.8 (split radix 4/2 DIT step) *Radix 4 decimation in frequency step for the split radix FFT:*

$$\mathcal{F}\,[a]^{(0/2)} \stackrel{n/2}{=} \left(\mathcal{F}\left[a^{(0\%2)}\right] + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right]\right) \tag{1.84}$$

$$\mathcal{F}\,[a]^{(1/4)} \stackrel{n/4}{=} \left(\mathcal{F}\left[a^{(0\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right]\right) + i\sigma\mathcal{S}^{1/4}\left(\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(3\%4)}\right]\right) \tag{1.85}$$

$$\mathcal{F}\,[a]^{(3/4)} \stackrel{n/4}{=} \left(\mathcal{F}\left[a^{(0\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right]\right) - i\sigma\mathcal{S}^{1/4}\left(\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(3\%4)}\right]\right) \tag{1.86}$$

Code 1.10 (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 `fft/fftsplitradix.cc`] uses a DIF core as above (and given in [28]).

For the (type-) complex version of the SRFT [FXT: `split_radix_fft` in `fft/cfftsplitradix.cc`] both DIF and DIT core routines have been implemented.

1.8 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 separate 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 cannot 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`$= -1$.

To see, why this works, we first note that

$$\begin{aligned} \mathcal{F}\,[a + i\,b] &= \mathcal{F}\,[a_S] + i\,\sigma\,\mathcal{F}\,[a_A] + i\,\mathcal{F}\,[b_S] + \sigma\,\mathcal{F}\,[b_A] && (1.87)\\ &= \mathcal{F}\,[a_S] + i\,\mathcal{F}\,[b_S] + i\,\sigma\,\left(\mathcal{F}\,[a_A] - i\,\mathcal{F}\,[b_A]\right) && (1.88)\end{aligned}$$

and the computation with swapped real- and imaginary parts gives

$$\mathcal{F}\,[b + i\,a] = \mathcal{F}\,[b_S] + i\,\mathcal{F}\,[a_S] + i\,\sigma\,\left(\mathcal{F}\,[b_A] - i\,\mathcal{F}\,[a_A]\right) \quad (1.89)$$

... but these are implicitly swapped at the end of the computation, giving

$$\mathcal{F}\,[a_S] + i\,\mathcal{F}\,[b_S] - i\,\sigma\,\left(\mathcal{F}\,[a_A] - i\,\mathcal{F}\,[b_A]\right) = \mathcal{F}^{-1}\,[a + i\,b] \quad (1.90)$$

When the type `Complex` is used then the best way to achieve the inverse transform may be to reverse the sequence according to the symmetry of the FT ([FXT: `reverse_nh` in `perm/reverse.h`], reordering by $k \mapsto k^{-1} \bmod n$). While not really 'free' the additional work shouldn't matter in most cases.

With real-to-complex FTs (R2CFT) the trick is to reverse the imaginary part after the transform. Obviously for the complex-to-real FTs (R2CFT) one has to reverse the imaginary part before the transform. Note that in the latter two cases the modification does not yield the inverse transform but the one with the 'other' sign in the exponent. Sometimes it may be advantageous to reverse the input of the R2CFT before transform, especially if the operation can be fused with other computations (e.g. with copying in or with the revbin-permutation).

1.9 Real valued Fourier transforms

The Fourier transform of a purely real sequence $c = \mathcal{F}\,[a]$ where $a \in \mathbb{R}$ has[6] 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. All procedures presented here use the following scheme for the real part of the transformed sequence c in the output array `a[]`:

$$\begin{aligned} \texttt{a[0]} &= \Re c_0 && (1.91)\\ \texttt{a[1]} &= \Re c_1 \\ \texttt{a[2]} &= \Re c_2 \\ &\cdots \\ \texttt{a[n/2]} &= \Re c_{n/2}\end{aligned}$$

[6] cf. relation 1.20

For the imaginary part of the result there are two schemes:
Scheme 1 ('parallel ordering') is

$$\begin{aligned} \mathtt{a[n/2+1]} &= \Im c_1 \\ \mathtt{a[n/2+2]} &= \Im c_2 \\ \mathtt{a[n/2+3]} &= \Im c_3 \\ &\dots \\ \mathtt{a[n-1]} &= \Im c_{n/2-1} \end{aligned} \tag{1.92}$$

Scheme 2 ('antiparallel ordering') is

$$\begin{aligned} \mathtt{a[n/2+1]} &= \Im c_{n/2-1} \\ \mathtt{a[n/2+2]} &= \Im c_{n/2-2} \\ \mathtt{a[n/2+3]} &= \Im c_{n/2-3} \\ &\dots \\ \mathtt{a[n-1]} &= \Im c_1 \end{aligned} \tag{1.93}$$

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

1.9.1 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 post-processing 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.

TBD: *formulas for realFFTwrap*

Here is the 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); zip(f, n); }
//
{
    if ( ldn==0 )  return;

    fht_fft((Complex *)f, ldn-1, +1);

    const ulong n = 1UL<<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 ( nh>=2 )  f[nh+1] *= is;
}
```

TBD: *eliminate if-statement in loop*

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 = 1UL<<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 ( nh>=2 )  { 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 realfft/realfftwrap.cc]

[FXT: wrap_complex_real_fft in realfft/realfftwrap.cc]

1.9.2 Real valued split radix Fourier transforms

Real to complex SRFT

Code 1.11 (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
  }
}
```

[FXT: `split_radix_real_complex_fft` in `realfft/realfftsplitradix.cc`]

Complex to real SRFT

Code 1.12 (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
}
```

[FXT: `split_radix_complex_real_fft` in `realfft/realfftsplitradix.cc`]

See [29].

1.10 Multidimensional FTs

1.10.1 Definition

Let $a_{x,y}$ ($x = 0, 1, 2, \ldots, C-1$ and $y = 0, 1, 2, \ldots, R-1$) be a 2-dimensional array of data[7]. Its 2-dimensional Fourier transform $c_{k,h}$ is defined by:

$$c = \mathcal{F}[a] \tag{1.94}$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{x=0}^{R-1} a_{x,y}\, z^{x\,k+y\,h} \quad \text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = R\,C \tag{1.95}$$

Its inverse is

$$a = \mathcal{F}^{-1}[c] \tag{1.96}$$

$$a_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h}\, z^{-(x\,k+y\,h)} \tag{1.97}$$

For a m-dimensional array $a_{\vec{x}}$ ($\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$, $x_i \in 0, 1, 2, \ldots, S_i$) the m-dimensional Fourier

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

transform $c_{\vec{k}}$ ($\vec{k} = (k_1, k_2, k_3, \ldots, k_m)$, $k_i \in 0, 1, 2, \ldots, S_i$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \cdots \sum_{x_m=0}^{S_m-1} a_{\vec{x}}\, z^{\vec{x}.\vec{k}} \quad \text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = S_1\, S_2\, \ldots\, S_m \tag{1.98}$$

$$= \frac{1}{\sqrt{n}} \sum_{\vec{x}=\vec{0}}^{\vec{S}} a_{\vec{x}}\, z^{\vec{x}.\vec{k}} \quad \text{where} \quad \vec{S} = (S_1 - 1, S_2 - 1, \ldots, S_m - 1)^T \tag{1.99}$$

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

1.10.2 The row column algorithm

The equation of the definition of the two dimensional FT (1.94) can be recast as

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} z^{x\,k} \sum_{x=0}^{R-1} a_{x,y}\, z^{y\,h} \tag{1.100}$$

which shows that the 2-dimensional FT can be accomplished by using 1-dimensional FTs to transform first the rows and then the columns[8]. This leads us directly to the row column algorithm:

Code 1.13 (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
    }
}
```

Here it is assumed that the rows lie in contiguous memory (as in the C language). [FXT: `twodim_fft` in `ndimfft/twodimfft.cc`]

Transposing the array before the column pass in order to avoid the copying of the columns to extra scratch space will do good for the performance in most cases. The transposing back at the end of the routine can be avoided if a back-transform will follow[9], the back-transform must then be called with `R` and `C` swapped.

The generalization to higher dimensions is straight forward. [FXT: `ndim_fft` in `ndimfft/ndimfft.cc`]

1.11 The matrix Fourier algorithm (MFA)

The matrix Fourier algorithm[10] (MFA) works for (composite) data lengths $n = R\,C$. Consider the input array as a $R \times C$-matrix (R rows, C columns).

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

[9] as typical for convolution etc.

[10] A variant of the MFA is called 'four step FFT' in [50].

Idea 1.9 (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.*

Note the elegance!

It is trivial to rewrite the MFA as the

Idea 1.10 (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).*

TBD: *MFA = radix-sqrt(n) DIF/DIT FFT*

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. TBD: *move cache discussion to appendix* 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. [50]), 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 optimized FFTs) can be observed.

1.12 Automatic generation of FFT codes

FFT generators are programs that output FFT routines, usually for fixed (short) lengths. In fact the thoughts here a not at all restricted to FFT codes, but FFTs and several unroll-able routines like matrix multiplications and convolutions are prime candidates for automated generation. Writing such a program

is easy: Take an existing FFT and change all computations into print statements that emit the necessary code. The process, however, is less than delightful and error-prone.

It would be much better to have another program that takes the existing FFT code as input and emit the code for the generator. Let us call this a *meta-generator*. Implementing such a meta-generator of course is highly nontrivial. It actually is equivalent to writing an interpreter for the language used plus the necessary data flow analysis[11].

A practical compromise is to write a program that, while theoretically not even close to a meta-generator, creates output that, after a little hand editing, is a usable generator code. The implemented perl script [FXT: file `scripts/metagen.pl`] is capable of converting a (highly pedantically formatted) piece of C++ code[12] into something that is reasonable close to a generator.

Further one may want to print the current values of the loop variables inside comments at the beginning of a block. Thereby it is possible to locate the corresponding part (both wrt. file and temporal location) of a piece of generated code in the original file. In addition one may keep the comments of the original code.

With FFTs it is necessary to identify ('reverse engineer') the trigonometric values that occur in the process in terms of the corresponding argument (rational multiples of π). The actual values should be inlined to some greater precision than actually needed, thereby one avoids the generation of multiple copies of the (logically) same value with differences only due to numeric inaccuracies. Printing the arguments, both as they appear and gcd-reduced, inside comments helps to understand (or further optimize) the generated code:

```
double c1=.98078528040323044912618223613 4; // == cos(Pi*1/16) == cos(Pi*1/16)
double s1=.19509032201612826784828486847 6; // == sin(Pi*1/16) == sin(Pi*1/16)
double c2=.92387953251128675612818318939 7; // == cos(Pi*2/16) == cos(Pi*1/8)
double s2=.38268343236508977172845998402 9; // == sin(Pi*2/16) == sin(Pi*1/8)
```

Automatic verification of the generated codes against the original is a mandatory part of the process.

A level of abstraction for the array indices is of great use: When the print statements in the generator emit some function of the index instead of its plain value it is easy to generate modified versions of the code for permuted input. That is, instead of

```
cout<<"sumdiff(f0, f2, g["<<k0<<"], g["<<k2<<"]);" <<endl;
cout<<"sumdiff(f1, f3, g["<<k1<<"], g["<<k3<<"]);" <<endl;
```

use

```
cout<<"sumdiff(f0, f2, "<<idxf(g,k0)<<", "<<idxf(g,k2)<<");" <<endl;
cout<<"sumdiff(f1, f3, "<<idxf(g,k1)<<", "<<idxf(g,k3)<<");" <<endl;
```

where `idxf(g, k)` can be defined to print a modified (e.g. the revbin-permuted) index `k`.

Here is the length-8 DIF FHT core as an example of some generated code:

```
template <typename Type>
inline void fht_dit_core_8(Type *f)
// unrolled version for length 8
{
 { // start initial loop
  { // fi = 0  gi = 1
   Type g0, f0, f1, g1;
   sumdiff(f[0], f[1], f0, g0);
   sumdiff(f[2], f[3], f1, g1);
   sumdiff(f0, f1);
   sumdiff(g0, g1);
   Type s1, c1, s2, c2;
   sumdiff(f[4], f[5], s1, c1);
   sumdiff(f[6], f[7], s2, c2);
   sumdiff(s1, s2);
   sumdiff(f0, s1, f[0], f[4]);
```

[11] If you know how to utilize gcc for that, please let me know.

[12] Actually only a small subset of C++.

```
   sumdiff(f1, s2, f[2], f[6]);
   c1 *= M_SQRT2;
   c2 *= M_SQRT2;
   sumdiff(g0, c1, f[1], f[5]);
   sumdiff(g1, c2, f[3], f[7]);
  }
 } // end initial loop
}
// -------------------------
// opcount by generator:  #mult=2=0.25/pt   #add=22=2.75/pt
```

The generated codes can be of great use when one wants to spot parts of the original code that need further optimization. Especially repeated trigonometric values and unused symmetries tend to be apparent in the unrolled code.

It is a good idea to let the generator count the number of operations (e.g. multiplications, additions, load/stores) of the code it emits. Even better if those numbers are compared to the corresponding values found in the compiled assembler code.

It is possible to have gcc produce the assembler code with the original source interlaced (which is a great tool with code optimization, cf. the target `asm` in the FXT makefile). The necessary commands are (include- and warning flags omitted)

```
# create assembler code:
c++ -S -fverbose-asm -g -O2  test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s  > test.lst
```

As an example the (generated)

```
template <typename Type>
inline void fht_dit_core_4(Type *f)
// unrolled version for length 4
{
 { // start initial loop
  { // fi = 0
   Type f0, f1, f2, f3;
   sumdiff(f[0], f[1], f0, f1);
   sumdiff(f[2], f[3], f2, f3);
   sumdiff(f0, f2, f[0], f[2]);
   sumdiff(f1, f3, f[1], f[3]);
  }
 } // end initial loop
}
// -------------------------
// opcount by generator:  #mult=0=0/pt   #add=8=2/pt
```

defined in [FXT: file `fht/shortfhtditcore.h`] results, using

```
// file test.cc:
int main()
{
    double f[4];
    fht_dit_core_4(f);
    return 0;
}
```

in (some lines deleted plus some editing for readability)

```
  11:test.cc   @     fht_dit_core_4(f);
  23:shortfhtditcore.h @ fht_dit_core_4(Type *f)
  24:shortfhtditcore.h @ // unrolled version for length 4
  25:shortfhtditcore.h @ {
  27:shortfhtditcore.h @  { // start initial loop
  28:shortfhtditcore.h @   { // fi = 0
  29:shortfhtditcore.h @    Type f0, f1, f2, f3;
  30:shortfhtditcore.h @    sumdiff(f[0], f[1], f0, f1);
  45:sumdiff.h @ template <typename Type>
  46:sumdiff.h @ static inline void
  47:sumdiff.h @ sumdiff(Type a, Type b, Type &s, Type &d)
```

```
 48:sumdiff.h @ // {s, d}  <--| {a+b, a-b}
 49:sumdiff.h @ { s=a+b; d=a-b; }
305 0006 DD442408              fldl 8(%esp)
306 000a DD442410              fldl 16(%esp)
 31:shortfhtditcore.h @    sumdiff(f[2], f[3], f2, f3);
319 000e DD442418              fldl 24(%esp)
320 0012 DD442420              fldl 32(%esp)
 32:shortfhtditcore.h @    sumdiff(f0, f2, f[0], f[2]);
333 0016 D9C3                  fld %st(3)
334 0018 D8C3                  fadd %st(3),%st
335 001a D9C2                  fld %st(2)
336 001c D8C2                  fadd %st(2),%st
339 001e D9C1                  fld %st(1)
340 0020 D8C1                  fadd %st(1),%st
341 0022 DD5C2408              fstpl 8(%esp)
342 0026 DEE9                  fsubrp %st,%st(1)
343 0028 DD5C2418              fstpl 24(%esp)
344 002c D9CB                  fxch %st(3)
349 002e DEE2                  fsubp %st,%st(2)
350 0030 DEE2                  fsubp %st,%st(2)
353 0032 D9C0                  fld %st(0)
354 0034 D8C2                  fadd %st(2),%st
355 0036 DD5C2410              fstpl 16(%esp)
356 003a DEE1                  fsubp %st,%st(1)
357 003c DD5C2420              fstpl 32(%esp)
 33:shortfhtditcore.h @    sumdiff(f1, f3, f[1], f[3]);
```

Note that the assembler code is not always in sync with the corresponding source lines which is especially true with higher levels of optimization.

1.13 Optimization considerations for fast transforms

- Reduce operations: use higher radix, at least radix 4 (with high radix algorithms note that the intel x86-architecture is severely register impaired)
- Mass storage FFTs: use MFA as described
- Trig recursion: loss of precision (not with mod FFTs), use stable versions, use table for initial values of recursion.
- Trig table: only for small lengths, else cache problem.
- Fused routines: combine first/last (few) step(s) in transforms with squaring/normalization/revbin/transposition etc. e.g. revbin-squaring in convolutions,
- Use explicit last/first step with radix as high a possible
- Write special versions for zero padded data (e.g. for convolutions), also write a special version of revbin_permute for zero padded data
- Integer stuff (e.g. exact convolutions): consider NTTs but be prepared for work & disappointments
- Image processing & effects: also check Walsh transform etc.
- Direct mapped cache: Avoid stride-2^n access (e.g. use gray-FFTs, gray-Walsh); try to achieve unit stride data access. Use the general prime factor algorithm. Improve memory locality (e.g. use the matrix Fourier algorithm (MFA))
- Vectorization: SIMD versions often boost performance
- For correlations/convolutions save two revbin_permute (or transpose) operations by combining DIF and DIT algorithms.

- Real-valued transforms & convolution: use Hartley transform (also for computation of spectrum). Even use complex FHT for forward step in real convolution.
- Reducing multiplications: Winograd FFT, mainly of theoretical interest (today the speed of multiplication is almost that of addition, often multiplies go parallel to adds)
- Only general rule for big sizes: better algorithms win.
- Do NOT blindly believe that some code is fast without profiling. Statements that some code is "the fastest" are always bogus.

1.14 Eigenvectors of the Fourier transform *

For $a_S := a + \overline{a}$, the symmetric part of a sequence a:

$$\mathcal{F}[\mathcal{F}[a_S]] = a_S \tag{1.101}$$

Now let $u_+ := a_S + \mathcal{F}[a_S]$ and $u_- := a_S - \mathcal{F}[a_S]$ then

$$\mathcal{F}[u_+] = \mathcal{F}[a_S] + a_S = a_S + \mathcal{F}[a_S] = +1 \cdot u_+ \tag{1.102}$$
$$\mathcal{F}[u_-] = \mathcal{F}[a_S] - a_S = -(a_S - \mathcal{F}[a_S]) = -1 \cdot u_- \tag{1.103}$$

u_+ and u_- are symmetric.

For $a_A := a - \overline{a}$, the antisymmetric part of a we have

$$\mathcal{F}[\mathcal{F}[a_A]] = -a_A \tag{1.104}$$

Therefore with $v_+ := a_A + i\,\mathcal{F}[a_A]$ and $v_- := a_A - i\,\mathcal{F}[a_A]$:

$$\mathcal{F}[v_+] = \mathcal{F}[a_A] - i\,a_A = -i\,(a_A + i\,\mathcal{F}[a_A]) = -i \cdot v_+ \tag{1.105}$$
$$\mathcal{F}[v_-] = \mathcal{F}[a_A] + i\,a_A = +i\,(a_A - i\,\mathcal{F}[a_A]) = +i \cdot v_- \tag{1.106}$$

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.

Using

$$a = \frac{1}{2}(u_+ + u_- + v_+ + v_-) \tag{1.107}$$

we can, for a given sequence, find a transform that is the 'square root' of the FT: Simply compute u_+, u_-, v_+, v_-. Then for $\lambda \in \mathbb{R}$ one can define a transform $\mathcal{F}^\lambda[a]$ as

$$\mathcal{F}^\lambda[a] = \frac{1}{2}\left((+1)^\lambda u_+ + (-1)^\lambda u_- + (-i)^\lambda v_+ + (+i)^\lambda v_-\right) \tag{1.108}$$

$\mathcal{F}^0[a]$ is the identity, $\mathcal{F}^1[a]$ is the (usual) FT, $\mathcal{F}^{1/2}[a]$ (which is not unique) is a transform so that $\mathcal{F}^{1/2}\left[\mathcal{F}^{1/2}[a]\right] = \mathcal{F}[a]$, that is, a 'square root' of the FT.

The eigenvectors of the Hartley Transform are $u_+ := a + \mathcal{H}[a]$ (with eigenvalue $+1$) and $u_+ := a - \mathcal{H}[a]$ (with eigenvalue -1).

Chapter 2

Convolutions

2.1 Definition and computation via FFT

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

$$\begin{aligned} h &= a \circledast b \\ h_\tau &:= \sum_{x+y \equiv \tau \pmod n} a_x \, b_y \end{aligned} \tag{2.1}$$

The last equation may be rewritten as

$$h_\tau := \sum_{x=0}^{n-1} a_x \, b_{\tau - x} \tag{2.2}$$

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

Code 2.1 (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:

$$\mathcal{F}[a \circledast b] = \mathcal{F}[a] \, \mathcal{F}[b] \tag{2.3}$$

i.e. convolution in original space is ordinary (element-wise) multiplication in Fourier space.

Here is the proof:

$$\begin{aligned}
\mathcal{F}[a]_k \, \mathcal{F}[b]_k &= \sum_x a_x \, z^{k\,x} \sum_y b_y \, z^{k\,y} \\
&\quad \text{with} \quad y := \tau - x \\
&= \sum_x a_x \, z^{k\,x} \sum_{\tau-x} b_{\tau-x} \, z^{k\,(\tau-x)} \\
&= \sum_x \sum_{\tau-x} a_x \, z^{k\,x} b_{\tau-x} \, z^{k\,(\tau-x)} \\
&= \sum_\tau \left(\sum_x a_x \, b_{\tau-x} \right) z^{k\,\tau} \\
&= \left(\mathcal{F} \left[\sum_x a_x \, b_{\tau-x} \right] \right)_k \\
&= (\mathcal{F}[a \circledast b])_k
\end{aligned} \tag{2.4}$$

Rewriting formula 2.3 as

$$a \circledast b = \mathcal{F}^{-1} [\mathcal{F}[a] \, \mathcal{F}[b]] \tag{2.5}$$

tells us how to proceed:

Code 2.2 (cyclic convolution via FFT) *Pseudo code for the cyclic convolution of two complex valued sequences* `x[]` *and* `y[]`*, result is returned in* `y[]`*:*

```
procedure fft_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 normalization. In the normalization loop you precompute `1.0/n` and multiply as divisions are much slower than multiplications. [FXT: `fht_fft_convolution` and `split_radix_fft_convolution` in `fft/fftcnvl.cc`]

Auto (or self) convolution is defined as

$$\begin{aligned}
h &= a \circledast a \\
h_\tau &:= \sum_{x+y \equiv \tau (n)} a_x \, a_y
\end{aligned} \tag{2.6}$$

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

In the definition of the cyclic convolution (2.1) 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 [36]) denoted by $h^{(1)}$ and $h^{(0)}$ respectively. Then

$$h \quad = \quad h^{(0)} + h^{(1)} \tag{2.7}$$

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 separate $h^{(0)}$ 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:

$$A \quad := \quad \{a_0, a_1, a_2, \ldots, a_{n-1}, 0, 0, \ldots, 0\} \tag{2.8}$$

Same for B. Then

$$h_\tau \quad := \quad \sum_{x=0}^{2\,n-1} A_x \, B_{\tau - x} \qquad \tau = 0, 1, 2, \ldots, 2\,n-1 \tag{2.9}$$

$$\sum_{\substack{x+y \equiv \tau \,(2n) \\ x,y<2n}} a_x \, b_y \quad = \quad \sum_{0 \le x < n} a_x \, b_y + \sum_{n \le x < 2n} a_x \, b_y \tag{2.10}$$

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

$$\sum_{0 \le x < n} a_x \, b_y \quad = \quad \sum_{x \le \tau} a_x \, b_{\tau - x} + \sum_{x > \tau} a_x \, b_{2n+\tau-x} =: R_\tau + S_\tau \tag{2.11}$$

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 element-wise added to the left half.

By the way, relation 2.3 is also true for the more general z-transform, but there is no (simple) back-transform, so we cannot turn

$$a \circledast b \quad = \quad \mathcal{Z}^{-1} \left[\mathcal{Z} \left[a \right] \, \mathcal{Z} \left[b \right] \right] \tag{2.12}$$

(the equivalent of 2.5) into a practical algorithm.

A convenient way to illustrate the cyclic convolution of to sequences is the following semi-symbolical table:

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
```

```
 1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15  0
 2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15  0  1
 3:   3  4  5  6   7  8  9 10  11 12 13 14  15  0  1  2

 4:   4  5  6  7   8  9 10 11  12 13 14 15   0  1  2  3
 5:   5  6  7  8   9 10 11 12  13 14 15  0   1  2  3  4
 6:   6  7  8  9  10 11 12 13  14 15  0  1   2  3  4  5
 7:   7  8  9 10  11 12 13 14  15  0  1  2   3  4  5  6

 8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
 9:   9 10 11 12  13 14 15  0   1  2  3  4   5  6  7  8
10:  10 11 12 13  14 15  0  1   2  3  4  5   6  7  8  9
11:  11 12 13 14  15  0  1  2   3  4  5  6   7  8  9 10

12:  12 13 14 15   0  1  2  3   4  5  6  7   8  9 10 11
13:  13 14 15  0   1  2  3  4   5  6  7  8   9 10 11 12
14:  14 15  0  1   2  3  4  5   6  7  8  9  10 11 12 13
15:  15  0  1  2   3  4  5  6   7  8  9 10  11 12 13 14
```

The entries denote where in the convolution the products of the input elements can be found:

```
 +--   0   1   2   3   ...
 |
 0:    0   1   2   4   ...
 1:    1   3   5 <--= h[5] contains a[2]*b[1]
 2:    4   8   9 <--= h[9] contains a[2]*b[b]
 3:    ...
```

Acyclic convolution (where there are 32 buckets 0..31) looks like:

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15 16
 2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15 16 17
 3:   3  4  5  6   7  8  9 10  11 12 13 14  15 16 17 18

 4:   4  5  6  7   8  9 10 11  12 13 14 15  16 17 18 19
 5:   5  6  7  8   9 10 11 12  13 14 15 16  17 18 19 20
 6:   6  7  8  9  10 11 12 13  14 15 16 17  18 19 20 21
 7:   7  8  9 10  11 12 13 14  15 16 17 18  19 20 21 22

 8:   8  9 10 11  12 13 14 15  16 17 18 19  20 21 22 23
 9:   9 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
10:  10 11 12 13  14 15 16 17  18 19 20 21  22 23 24 25
11:  11 12 13 14  15 16 17 18  19 20 21 22  23 24 25 26

12:  12 13 14 15  16 17 18 19  20 21 22 23  24 25 26 27
13:  13 14 15 16  17 18 19 20  21 22 23 24  25 26 27 28
14:  14 15 16 17  18 19 20 21  22 23 24 25  26 27 28 29
15:  15 16 17 18  19 20 21 22  23 24 25 26  27 28 29 30
```

. . . the elements in the lower right triangle do not 'wrap around' anymore, they go to extra buckets. Note that bucket 31 does not appear, it is always zero.

The equivalent table for a (cyclic) correlation is

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0 15 14 13  12 11 10  9   8  7  6  5   4  3  2  1
 1:   1  0 15 14  13 12 11 10   9  8  7  6   5  4  3  2
 2:   2  1  0 15  14 13 12 11  10  9  8  7   6  5  4  3
 3:   3  2  1  0  15 14 13 12  11 10  9  8   7  6  5  4

 4:   4  3  2  1   0 15 14 13  12 11 10  9   8  7  6  5
 5:   5  4  3  2   1  0 15 14  13 12 11 10   9  8  7  6
 6:   6  5  4  3   2  1  0 15  14 13 12 11  10  9  8  7
 7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   8  7  6  5   4  3  2  1   0 15 14 13  12 11 10  9
 9:   9  8  7  6   5  4  3  2   1  0 15 14  13 12 11 10
10:  10  9  8  7   6  5  4  3   2  1  0 15  14 13 12 11
11:  11 10  9  8   7  6  5  4   3  2  1  0  15 14 13 12
```

```
12:  12 11 10  9   8  7  6  5   4  3  2  1   0 15 14 13
13:  13 12 11 10   9  8  7  6   5  4  3  2   1  0 15 14
14:  14 13 12 11  10  9  8  7   6  5  4  3   2  1  0 15
15:  15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

while the acyclic counterpart is:

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0 31 30 29  28 27 26 25  24 23 22 21  20 19 18 17
 1:   1  0 31 30  29 28 27 26  25 24 23 22  21 20 19 18
 2:   2  1  0 31  30 29 28 27  26 25 24 23  22 21 20 19
 3:   3  2  1  0  31 30 29 28  27 26 25 24  23 22 21 20

 4:   4  3  2  1   0 31 30 29  28 27 26 25  24 23 22 21
 5:   5  4  3  2   1  0 31 30  29 28 27 26  25 24 23 22
 6:   6  5  4  3   2  1  0 31  30 29 28 27  26 25 24 23
 7:   7  6  5  4   3  2  1  0  31 30 29 28  27 26 25 24

 8:   8  7  6  5   4  3  2  1   0 31 30 29  28 27 26 25
 9:   9  8  7  6   5  4  3  2   1  0 31 30  29 28 27 26
10:  10  9  8  7   6  5  4  3   2  1  0 31  30 29 28 27
11:  11 10  9  8   7  6  5  4   3  2  1  0  31 30 29 28

12:  12 11 10  9   8  7  6  5   4  3  2  1   0 31 30 29
13:  13 12 11 10   9  8  7  6   5  4  3  2   1  0 31 30
14:  14 13 12 11  10  9  8  7   6  5  4  3   2  1  0 31
15:  15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

Note that bucket 16 does not appear, it is always zero.

Two-dimensional convolution (here 4x4) looks like

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  2  3  0   5  6  7  4   9 10 11  8  13 14 15 12
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   3  0  1  2   7  4  5  6  11  8  9 10  15 12 13 14

 4:   4  5  6  7   8  9 10 11  12 13 14 15   0  1  2  3
 5:   5  6  7  4   9 10 11  8  13 14 15 12   1  2  3  0
 6:   6  7  4  5  10 11  8  9  14 15 12 13   2  3  0  1
 7:   7  4  5  6  11  8  9 10  15 12 13 14   3  0  1  2

 8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
 9:   9 10 11  8  13 14 15 12   1  2  3  0   5  6  7  4
10:  10 11  8  9  14 15 12 13   2  3  0  1   6  7  4  5
11:  11  8  9 10  15 12 13 14   3  0  1  2   7  4  5  6

12:  12 13 14 15   0  1  2  3   4  5  6  7   8  9 10 11
13:  13 14 15 12   1  2  3  0   5  6  7  4   9 10 11  8
14:  14 15 12 13   2  3  0  1   6  7  4  5  10 11  8  9
15:  15 12 13 14   3  0  1  2   7  4  5  6  11  8  9 10
```

while 4x4 correlation would be

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   3  0  1  2   7  4  5  6  11  8  9 10  15 12 13 14
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   1  2  3  0   5  6  7  4   9 10 11  8  13 14 15 12

 4:  12 13 14 15   0  1  2  3   4  5  6  7   8  9 10 11
 5:  15 12 13 14   3  0  1  2   7  4  5  6  11  8  9 10
 6:  14 15 12 13   2  3  0  1   6  7  4  5  10 11  8  9
 7:  13 14 15 12   1  2  3  0   5  6  7  4   9 10 11  8

 8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
 9:  11  8  9 10  15 12 13 14   3  0  1  2   7  4  5  6
10:  10 11  8  9  14 15 12 13   2  3  0  1   6  7  4  5
11:   9 10 11  8  13 14 15 12   1  2  3  0   5  6  7  4

12:   4  5  6  7   8  9 10 11  12 13 14 15   0  1  2  3
13:   7  4  5  6  11  8  9 10  15 12 13 14   3  0  1  2
14:   6  7  4  5  10 11  8  9  14 15 12 13   2  3  0  1
15:   5  6  7  4   9 10 11  8  13 14 15 12   1  2  3  0
```

2.2 Mass storage convolution using the MFA

The matrix Fourier algorithm is also an ideal candidate for mass storage FFTs, i.e. FFTs for data sets that do not fit into physical RAM[1].

In convolution computations it is straight forward 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 the sake of simplicity auto convolution is considered here:

Idea 2.1 (matrixfft convolution algorithm) *The matrix FFT 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 (element-wise).*
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_fft_convolution` in `matrixfft/matrixfftcnvl.cc`]

[FXT: `matrix_fft_convolution0` in `matrixfft/matrixfftcnvl.cc`]

[FXT: `matrix_fft_auto_convolution` in `matrixfft/matrixfftcnvla.cc`]

[FXT: `matrix_fft_auto_convolution0` in `matrixfft/matrixfftcnvla.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

Minimization 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[2]. We now analyze 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.

[1] The naive idea to simply try such an FFT with the virtual memory mechanism will of course, due to the non-locality of FFTs, end in eternal hard disk activity

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

In step 1 and 3 of algorithm 2.5 one reads from disk (row by row, involving R seeks) the number of columns that just fit into RAM, does the (many, short) column-FFTs[3], 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 minimized 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[4] 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 multi-threading 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.

Using a mass storage convolution as described the calculation of the number $9^{9^9} \approx 0.4281247 \cdot 10^{369,693,100}$ could be done on a 32 bit machine in 1999. The computation used two files of size 2GigaBytes each and took less than eight hours on a system with a AMD K6/2 CPU at 366MHz with 66MHz memory.
Cf. [hfloat: `examples/run1-pow999.txt`]

2.3 Weighted Fourier transforms

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

$$\begin{aligned} c &= \mathcal{W}_v\,[a] \\ c_k &:= \sum_{x=0}^{n-1} v_x\,a_x\,z^{x\,k} \qquad v_x \neq 0 \quad \forall x \end{aligned} \tag{2.13}$$

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

$$\begin{aligned} a &= \mathcal{W}_v^{-1}\,[c] \\ a_x &= \frac{1}{n\,v_x} \sum_{k=0}^{n-1} c_k\,z^{-x\,k} \end{aligned} \tag{2.14}$$

[3] real-complex FFTs in step 1 and complex-real FFTs in step 3.
[4] allowing for 8 million 8 byte floats, so the total FFT size is $S = 16 \cdot 64 = 1024$ MB or 32 million floats

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 2.3 (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 2.4 (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 `weighted/weightedfft.cc`]

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

Introducing a *weighted (cyclic) convolution* h_v by

$$\begin{aligned}
h_v &= a \circledast_{\{v\}} b \\
&= \mathcal{W}_v^{-1}\left[\mathcal{W}_v\left[a\right]\,\mathcal{W}_v\left[b\right]\right]
\end{aligned} \tag{2.15}$$

(cf. formula 2.5)

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

$$h_v = h^{(0)} + V^n\,h^{(1)} \tag{2.16}$$

($h^{(0)}$ and $h^{(1)}$ were defined by formula 2.7). 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

$$\sum_{x+y\equiv\tau\,(\mathrm{mod}\;n)} \left(a_x\,V^x\right)\left(b_y\,V^y\right) = \sum_{x\le\tau} a_x\,b_{\tau-x}\,V^\tau + \sum_{x>\tau} a_x\,b_{n+\tau-x}\,V^{n+\tau} \tag{2.17}$$

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*:

$$h_v \quad = \quad h^{(0)} \mp i\, h^{(1)} \tag{2.18}$$

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

$$h_v \quad = \quad h^{(0)} - h^{(1)} \tag{2.19}$$

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. [2]).

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

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

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

The semi-symbolic table (cf. table 2.1) for the negacyclic convolution is

```
 +--   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 |
 0:    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 1:    1   2   3   4   5   6   7   8   9  10  11  12  13  14  15   0-
 2:    2   3   4   5   6   7   8   9  10  11  12  13  14  15   0-  1-
 3:    3   4   5   6   7   8   9  10  11  12  13  14  15   0-  1-  2-
 4:    4   5   6   7   8   9  10  11  12  13  14  15   0-  1-  2-  3-
 5:    5   6   7   8   9  10  11  12  13  14  15   0-  1-  2-  3-  4-
 6:    6   7   8   9  10  11  12  13  14  15   0-  1-  2-  3-  4-  5-
 7:    7   8   9  10  11  12  13  14  15   0-  1-  2-  3-  4-  5-  6-
 8:    8   9  10  11  12  13  14  15   0-  1-  2-  3-  4-  5-  6-  7-
 9:    9  10  11  12  13  14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-
10:   10  11  12  13  14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9-
11:   11  12  13  14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9- 10-
12:   12  13  14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9- 10- 11-
13:   13  14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9- 10- 11- 12-
14:   14  15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9- 10- 11- 12- 13-
15:   15   0-  1-  2-  3-  4-  5-  6-  7-  8-  9- 10- 11- 12- 13- 14-
```

Here the products that enter with negative sign are indicated with a postfix minus at the corresponding entry.

With right-angle convolution the minuses have to be replaced by $i = \sqrt{-1}$ which means the wrap-around (i.e. $h^{(1)}$) elements go to the imaginary part. With real input one thereby effectively separates $h^{(0)}$ and $h^{(1)}$.

Note that once one has routines for both cyclic and negacyclic convolution the parts $h^{(0)}$ and $h^{(1)}$ can be computed as sum and difference, respectively. Thereby all expressions of the form $\alpha\, h^{(0)} + \beta\, h^{(1)}$ can be trivially computed.

2.4 Half cyclic convolution for half the price?

The computation of $h^{(0)}$ from formula 2.7 (without computing $h^{(1)}$) is called *half cyclic convolution*. Apparently, 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[5].

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

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.

2.5 Convolution using the MFA

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm (idea 2.1):

1. Apply a FFT on each column.
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

2.5.1 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[6] $s := x^{(0/2)} + x^{(1/2)}$ and $d := x^{(0/2)} - x^{(1/2)}$ using the formula

$$x \circledast x \quad = \quad \frac{1}{2} \{s \circledast s + d \circledast_- d, \quad s \circledast s - d \circledast_- d\} \tag{2.20}$$

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

$$x \circledast y \quad = \quad \frac{1}{2} \{s_x \circledast s_y + d_x \circledast_- d_y, \quad s_x \circledast s_y - d_x \circledast_- d_y\} \tag{2.21}$$

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 $z_x := \{x_0, x_1, \ldots, n_{n-1}, 0, 0, \ldots, 0\}$ (i.e. x with n zeros appended). Using formula 2.20 one gets for the two sequences x and y (with $s_x = d_x = x$, $s_y = d_y = y$):

$$x \circledast_{ac} y \quad = \quad z_x \circledast z_y \quad = \quad \frac{1}{2} \{x \circledast y + x \circledast_- y, \quad x \circledast y - x \circledast_- y\} \tag{2.22}$$

And for the acyclic auto convolution:

$$x \circledast_{ac} x \quad = \quad z \circledast z \quad = \quad \frac{1}{2} \{x \circledast x + x \circledast_- x, \quad x \circledast x - x \circledast_- x\} \tag{2.23}$$

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

2.5.2 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

$$\begin{aligned} x^{(0/3)} &= A \circledast A + B \circledast_{\{\omega\}} B + C \circledast_{\{\omega^2\}} C \\ x^{(1/3)} &= A \circledast A + \omega^2 (B \circledast_{\{\omega\}} B) + \omega (C \circledast_{\{\omega^2\}} C) \\ x^{(2/3)} &= A \circledast A + \omega (B \circledast_{\{\omega\}} B) + \omega^2 (C \circledast_{\{\omega^2\}} C) \end{aligned} \tag{2.24}$$

For real valued data C is the complex conjugate (*cc.*) 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.

2.6 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[7], the code for that task is given on page 68.

All other weighted convolutions involve complex computations, but it is easy to see how to reduce 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 $1, 2, \ldots, R/2 - 1$)

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

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).

2.7 Convolution without transposition using the MFA *

Section 7.4 explained the connection between revbin-permutation and transposition. Equipped with that knowledge an algorithm for convolution using the MFA that uses `revbin_permute` instead of transpose is almost straight forward:

```
rows=8  columns=4
```

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

```
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

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 (do not 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
```

As shown works for sizes that are a power of two, generalizes for sizes a power of some prime. TBD: *add text*

2.8 The z-transform (ZT)

In this section we will learn a technique to compute the FT by a (linear) convolution. In fact, the transform computed is the z-transform, a more general transform that in a special case is identical to the FT.

2.8.1 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

$$\hat{a}_k := \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{2.25}$$

The z-transform is a linear transformation, its most important property is the convolution property (formula 2.3): Convolution in original space corresponds to ordinary (element-wise) multiplication in z-space. (See [13] and [17].)

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

2.8.2 Computation of the ZT via convolution

In the definition of the (discrete) z-transform we rewrite[8] the product $x\,k$ as

$$x\,k = \frac{1}{2}\left(x^2 + k^2 - (k-x)^2\right) \tag{2.26}$$

$$\hat{f}_k = \sum_{x=0}^{n-1} f_x \, z^{x\,k} = z^{k^2/2} \sum_{x=0}^{n-1} \left(f_x \, z^{x^2/2}\right) z^{-(k-x)^2/2} \tag{2.27}$$

This leads to the following

Idea 2.2 (chirp z-transform) *Algorithm for the chirp z-transform:*

1. *Multiply f element-wise 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 element-wise 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.

[8] cf. [2]

2.8.3 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 becomes 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 $2\,n$.

[FXT: `fft_arblen` in `chirp/fftarblen.cc`]

TBD: *show shortcuts for n even/odd*

2.8.4 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 not integer multiples of the lowest frequency of the DFT. A thorough discussion can be found in [51].

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

Chapter 3

The Hartley transform (HT)

3.1 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

$$c = \mathcal{H}[a] \tag{3.1}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left(\cos \frac{2\pi k x}{n} + \sin \frac{2\pi k x}{n} \right) \tag{3.2}$$

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

$$\mathcal{H}[a] \in \mathbb{R} \quad \text{for} \quad a \in \mathbb{R} \tag{3.3}$$

It also is its own inverse:

$$\mathcal{H}[\mathcal{H}[a]] = a \tag{3.4}$$

The symmetries of the HT are simply:

$$\mathcal{H}[a_S] = \overline{\mathcal{H}[a_S]} = \mathcal{H}[\overline{a_S}] \tag{3.5}$$

$$\mathcal{H}[a_A] = \overline{\mathcal{H}[a_A]} = -\mathcal{H}[\overline{a_A}] \tag{3.6}$$

i.e. symmetry is, like for the Fourier transform, conserved.

The $n \log(n)$-implementations of the HT are called *fast Hartley transforms* (FHT).

3.2 Radix 2 FHT algorithms

3.2.1 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 3.1 (FHT radix 2 DIT step) *Radix 2 decimation in time step for the FHT:*

$$\mathcal{H}[a]^{(left)} \overset{n/2}{=} \mathcal{H}\left[a^{(even)}\right] + \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{3.7}$$

$$\mathcal{H}[a]^{(right)} \overset{n/2}{=} \mathcal{H}\left[a^{(even)}\right] - \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{3.8}$$

Code 3.1 (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];
    }
}
```

[FXT: `recursive_fht_dit2` in `slow/recfht2.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 3.2 (Hartley shift) `procedure hartley_shift_05(c[], n)`

```
// real c[0..n-1] input, result
{
    nh := n/2
    j := n-1
    for k:=1 to nh-1
    {
        c := cos( PI*k/n )
        s := sin( 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 `fht/hartleyshift.cc`]

Code 3.3 (radix 2 DIT FHT, localized) *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
            }
        }
    }
}
```

The derivation of the 'usual' DIT2 FHT algorithm starts by fusing the shift with the sum/diff step:

```
void fht_localized_dit2(double *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            for (ulong j=1, k=mh-1; j<k; ++j,--k)
            {
                double s, c;
                SinCos(phi0*j, &s, &c);

                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

[FXT: fht_localized_dit2 in fht/fhtdit2.cc] Swapping the innermost loops then yields (considerations as for DIT FFT, page 16, hold)

```
void fht_dit2(double *f, ulong ldn)
// decimation in time radix 2 fht
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }

        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

[FXT: fht_dit2 in fht/fhtdit2.cc]

3.2.2 Decimation in frequency (DIF) FHT

Idea 3.2 (FHT radix 2 DIF step) *Radix 2 decimation in frequency step for the FHT:*

$$\mathcal{H}\left[a\right]^{(even)} \stackrel{n/2}{=} \mathcal{H}\left[a^{(left)} + a^{(right)}\right] \tag{3.9}$$

$$\mathcal{H}\left[a\right]^{(odd)} \stackrel{n/2}{=} \mathcal{H}\left[\mathcal{X}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{3.10}$$

Code 3.4 (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
    }
}
```

[FXT: `recursive_fht_dif2` in `slow/recfht2.cc`]

Code 3.5 (radix 2 DIF FHT, localized) *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: fht_localized_dif2 in fht/fhtdif2.cc]

The 'usual' DIF2 FHT algorithm then is

```
void fht_dif2(double *f, ulong ldn)
// decimation in frequency radix 2 fht
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }

        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;
            }
        }
    }

    revbin_permute(f, n);
}
```

[FXT: fht_dif2 in fht/fhtdif2.cc]

TBD: *higher radix FHT*

3.3 Complex 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:

$$\mathcal{F}[d] = \frac{1}{2}\left(\mathcal{H}[d] + \overline{\mathcal{H}[d]} + \sigma\, i\, \left(\mathcal{H}[d] - \overline{\mathcal{H}[d]}\right)\right) \tag{3.11}$$

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

$$\Re\mathcal{F}[a + i\,b] = \frac{1}{2}\left(\mathcal{H}[a] + \overline{\mathcal{H}[a]} - \sigma\, \left(\mathcal{H}[b] - \overline{\mathcal{H}[b]}\right)\right) \tag{3.12}$$

$$\Im\mathcal{F}[a + i\,b] = \frac{1}{2}\left(\mathcal{H}[b] + \overline{\mathcal{H}[b]} + \sigma\, \left(\mathcal{H}[a] - \overline{\mathcal{H}[a]}\right)\right) \tag{3.13}$$

Alternatively, one can recast the relations (using the symmetry relations 3.5 and 3.6) as

$$\Re\mathcal{F}[a + i\,b] = \frac{1}{2}\mathcal{H}[a_S - \sigma\, b_A] \tag{3.14}$$

$$\Im\mathcal{F}[a + i\,b] = \frac{1}{2}\mathcal{H}[b_S + \sigma\, a_A] \tag{3.15}$$

Both formulations lead to the very same

Code 3.6 (complex FT by HT conversion)

```
fht_fft_conversion(a[],b[],n,is)
// preprocessing to use two length-n FHTs
// to compute a length-n complex FFT
// or
// postprocessing to use two length-n FHTs
// to compute a length-n complex FFT
//
// self-inverse
{
    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)
    }
}
```

[FXT: `fht_fft_conversion` in `fht/fhtfft.cc`] [FXT: `fht_fft_conversion` in `fht/fhtcfft.cc`]

Now we have two options to compute a complex FT by two HTs:

Code 3.7 (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)
    fht_fft_conversion(a[], b[], n, is)
}
```

and

Code 3.8 (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)
{
    fht_fft_conversion(a[], b[], n, is)
    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 3.7 for the forward and 3.8 for the backward transform. The complex squarings are then combined with the pre- and post-processing steps, thereby interleaving the most nonlocal memory accesses with several arithmetic operations.

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

3.4 Complex FT by complex HT and vice versa

A complex valued HT is simply two HTs (one of the real, one of the imaginary part). So we can use both of 3.7 or 3.8 and there is nothing new. Really? If one writes a type complex version of both the conversion and the FHT the routine 3.7 will look like

```
fft_by_fht1(c[], n, is)
// complex c[0..n-1] input,result
{
    fht(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

(the 3.8 equivalent is hopefully obvious)

This may not make you scream but here is the message: it makes sense to do so. It is pretty easy to derive a complex FHT from the real (i.e. usual) version[1] and with a well optimized FHT you get an even better optimized FFT. Note that this trivial rewrite virtually gets you a length-n FHT with the book keeping and trig-computation overhead of a length-$n/2$ FHT.

[FXT: `fht_dit_core` in `fht/cfhtdit.cc`]

[FXT: `fht_dif_core` in `fht/cfhtdif.cc`]

[FXT: `fht_fft_conversion` in `fht/fhtcfft.cc`]

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

Vice versa: Let T be the operator corresponding to the `fht_fft_conversion`, T is its own inverse: $T = T^{-1}$, or, equivalently $T \cdot T = 1$. We have seen that

$$\mathcal{F} = \mathcal{H} \cdot T \qquad \text{and} \qquad \mathcal{F} = T \cdot \mathcal{H} \tag{3.16}$$

[1] in fact this is done automatically in FXT

Therefore trivially

$$\mathcal{H} = T \cdot \mathcal{F} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F} \cdot T \tag{3.17}$$

Hence we have either

```
fht_by_fft(c[], n, is)
// complex c[0..n-1] input,result
{
    fft(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

or the same thing with swapped lines. Of course the same ideas also work for separate real- and imaginary-parts.

3.5 Real FT by HT and vice versa

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 3.12 and 3.13 and set $b = 0$:

$$\Re\mathcal{F}[a] = \frac{1}{2}\left(\mathcal{H}[a] + \overline{\mathcal{H}[a]}\right) \tag{3.18}$$

$$\Im\mathcal{F}[a] = \frac{1}{2}\left(\mathcal{H}[a] - \overline{\mathcal{H}[a]}\right) \tag{3.19}$$

The pseudo code is straight forward:

Code 3.9 (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

$$\begin{aligned}
\mathtt{a}[0] &= \Re c_0 \\
\mathtt{a}[1] &= \Re c_1 \\
\mathtt{a}[2] &= \Re c_2 \\
&\dots \\
\mathtt{a}[n/2] &= \Re c_{n/2} \\
\mathtt{a}[n/2+1] &= \Im c_{n/2-1} \\
\mathtt{a}[n/2+2] &= \Im c_{n/2-2} \\
\mathtt{a}[n/2+3] &= \Im c_{n/2-3} \\
&\dots \\
\mathtt{a}[n-1] &= \Im c_1
\end{aligned} \tag{3.20}$$

[FXT: `fht_real_complex_fft` in `realfft/realfftbyfht.cc`]

The inverse procedure is:

Code 3.10 (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)
}
```

[FXT: `fht_complex_real_fft` in `realfft/realfftbyfht.cc`]

Vice versa: same line of thought as for complex versions. Let T_{rc} be the operator corresponding to the post-processing in `real_complex_fft_by_fht`, and T_{cr} correspond to the preprocessing in `complex_real_fft_by_fht`. That is

$$\mathcal{F}_{cr} = \mathcal{H} \cdot T_{cr} \qquad \text{and} \qquad \mathcal{F}_{rc} = T_{rc} \cdot \mathcal{H} \tag{3.21}$$

It should be no surprise that $T_{rc} \cdot T_{cr} = 1$, or, equivalently $T_{rc} = T_{cr}^{-1}$ and $T_{cr} = T_{rc}^{-1}$. Therefore

$$\mathcal{H} = T_{cr} \cdot \mathcal{F}_{rc} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F}_{cr} \cdot T_{rc} \tag{3.22}$$

The corresponding code should be obvious. Watch out for real/complex FFTs that use a different ordering than 3.20.

3.6 Discrete cosine transform (DCT) by HT

The discrete cosine transform wrt. the basis

$$u(k) \quad = \quad \nu(k) \cdot \cos \frac{\pi\, k\, (i + 1/2)}{n} \tag{3.23}$$

(where $\nu(k) = 1$ for $k = 0$, $\nu(k) = \sqrt{2}$ else) can be computed from the FHT using an auxiliary routine named `cos_rot`. TBD: *give cosrot's action mathematically*

```
procedure cos_rot(x[], y[], n)
// real x[0..n-1] input
// real y[0..n-1] result
{
    nh := n/2
    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]
    }
}
```

which is its own inverse. (cf. [FXT: `cos_rot` in `dctdst/cosrot.cc`]) Then

Code 3.11 (DCT via FHT) *Pseudo code for the computation of the DCT via FHT:*

```
procedure dcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    real y[0..n-1]  // workspace
    unzip_rev(x, y, n)
    fht(y[],ldn)
    cos_rot(y[], x[], n)
}
```

where

```
procedure unzip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

(see section 7.6, page 131).

The inverse routine is

Code 3.12 (IDCT via FHT) *Pseudo code for the computation of the IDCT via FHT:*

```
procedure idcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**n
    real y[0..n-1]  // workspace
    cos_rot(x[], y[], n);
    fht(y[],ldn)
    zip_rev(y[], x[], n)
}
```

where

```
procedure zip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

The implementation of both the forward and the backward transform (cf. [FXT: `dcth` and `idcth` in `dctdst/dcth.cc`]) avoids the temporary array `y[]` if no scratch space is supplied.

Cf. [32], [33].

An alternative variant for the computation of the DCT that also uses the FHT is given in [FXT: `dcth_zapata` in `dctdst/dctzapata.cc`]. The algorithm is described in [34].

3.7 Discrete sine transform (DST) by DCT

TBD: *definition dst, idst*

Code 3.13 (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 `dctdst/dsth.cc`]

Code 3.14 (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 `dctdst/dsth.cc`]

3.8 Convolution via FHT

The convolution property of the Hartley transform is

$$\mathcal{H}\left[a \circledast b\right] \;=\; \frac{1}{2}\Big(\mathcal{H}\left[a\right]\mathcal{H}\left[b\right] - \overline{\mathcal{H}\left[a\right]}\,\overline{\mathcal{H}\left[b\right]} + \mathcal{H}\left[a\right]\overline{\mathcal{H}\left[b\right]} + \overline{\mathcal{H}\left[a\right]}\,\mathcal{H}\left[b\right]\Big) \tag{3.24}$$

or, written element-wise:

$$\begin{aligned}\mathcal{H}\left[a \circledast b\right]_k &= \frac{1}{2}\left(c_k\,d_k - \overline{c_k}\,\overline{d_k} + c_k\,\overline{d_k} + \overline{c_k}\,d_k\right)\\ &= \frac{1}{2}\left(c_k\left(d_k + \overline{d_k}\right) + \overline{c_k}\left(d_k - \overline{d_k}\right)\right) \quad \text{where} \quad c = \mathcal{H}\left[a\right],\quad d = \mathcal{H}\left[b\right]\end{aligned} \tag{3.25}$$

Code 3.15 (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 normalization. Cf. [FXT: `fht_convolution` in `fht/fhtcnvl.cc`]

Equation 3.25 (slightly optimized) for the auto convolution is

$$\begin{aligned} \mathcal{H}\left[a \circledast a\right]_k &= \frac{1}{2}\left(c_k\left(c_k + \overline{c_k}\right) + \overline{c_k}\left(c_k - \overline{c_k}\right)\right) \\ &= c_k\,\overline{c_k} + \frac{1}{2}\left(c_k^2 - \overline{c_k}^2\right) \quad \text{where} \quad c = \mathcal{H}\left[a\right] \end{aligned} \tag{3.26}$$

Code 3.16 (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. Cf. [FXT: `fht_auto_convolution` in `fht/fhtcnvla.cc`]

3.9 Negacyclic convolution via FHT

Code 3.17 (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_05(x, n)

    // 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_05(x, n)
}
```

(The code for `hartley_shift_05()` was given on page 56.)

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

Code for the negacyclic convolution (without the 'self'):

[FXT: `fht_negacyclic_convolution` in `fht/fhtnegacnvl.cc`]

The underlying idea can be derived by closely looking at the convolution of real sequences by the radix-2 FHT.

The FHT-based negacyclic convolution turns out to be extremely useful for the computation of weighted transforms, e.g. in the MFA-based convolution for real input.

Chapter 4

Number theoretic transforms (NTTs)

How to make a number theoretic 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 *number theoretic 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 NTT 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:

$$n \backslash R \iff \exists \sqrt[n]{1} \tag{4.1}$$

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 back-transform. 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:

$$n \perp m \iff \exists n^{-1} \text{ in } \mathbb{Z}/m\mathbb{Z} \tag{4.2}$$

Cf. [1], [3], [19] or [2] and books on number theory.

4.1 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 lengths $n < p$: 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 $(1 \le e < p)$ 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 does not 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. This is because the order of any element divides the maximal order. To find a primitive root in $\mathbb{F}_p$ proceed as indicated by the following pseudo code:

Code 4.1 (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 4.2 (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
}
```

4.2 Composite modulus: $\mathbb{Z}/m\mathbb{Z}$

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

$$\varphi(p^k) \quad = \quad p^k - p^{k-1} \tag{4.3}$$

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

[2]Primes of that form are not 'exceptional', cf. Lipson [3]

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

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

Code 4.4 (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 $\varphi(m)$:

$$|\mathbb{Z}/m\mathbb{Z}^*| \quad = \quad \varphi(m) \tag{4.4}$$

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

$$|\mathbb{Z}/m\mathbb{Z}^*| \quad = \quad \varphi(2^{k_0}) \cdot \varphi(p_1^{k_1}) \cdot \ldots \cdot \varphi(p_q^{k_q}) \tag{4.5}$$

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

$$R(p^k) \quad = \quad \varphi(p^k) \tag{4.6}$$

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

$$R(2^k) \quad = \quad \begin{cases} 1 & \text{for } k = 1 \\ 2 & \text{for } k = 2 \\ 2^{k-2} & \text{for } k \geq 3 \end{cases} \tag{4.7}$$

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

$$R(m) \quad = \quad \text{lcm}(R(2^{k_0}), R(p_1^{k_1}), \ldots, R(p_q^{k_q})) \tag{4.8}$$

where lcm() denotes the least common multiple.

Pseudo code to compute $R(m)$:

Code 4.5 (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:

$$\mathbb{Z}/m\mathbb{Z}^* \quad \text{cyclic for} \quad m = 2,\ 4,\ p^k,\ 2 \cdot p^k \tag{4.9}$$

where p is an odd prime. If m contains two different odd primes p_a, p_b then $R(m) = \text{lcm}(\ldots, \varphi(p_a), \varphi(p_b), \ldots)$ is at least by a factor of two smaller than $\varphi(m) = \ldots \cdot \varphi(p_a) \cdot \varphi(p_b) \cdot \ldots$ because both $\varphi(p_a)$ and $\varphi(p_b)$ are even, so $\mathbb{Z}/m\mathbb{Z}^*$ cannot 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 $R(2^k)$.

Pseudo code (following [19]) for a function that returns the order of some element x in $\mathbb{Z}/m\mathbb{Z}$:

Code 4.6 (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 4.7 (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 4.8 (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 `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.

4.3 Pseudocode for NTTs

To implement NTTs ('mod-m FFTs') one basically must implement modulo-arithmetics 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 `mod/mod.h`]

For the back-transform 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.

4.3.1 Radix 2 DIT NTT

Code 4.9 (radix 2 DIT NTT) *Pseudo code for the radix 2 decimation in time NTT (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)
        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 1.4.2 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
    {
```

[FXT: `ntt_dit2` in `mod/nttdit2.cc`]

4.3.2 Radix 2 DIF NTT

Code 4.10 (radix 2 DIF NTT) *Pseudo code for the radix 2 decimation in frequency NTT:*

```
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 1.4.3 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)`.

[FXT: `ntt_dif2` in `mod/nttdif2.cc`]

4.3.3 Radix 4 NTTs

C++ code for a radix-4 decimation in time NTT:

```
static const ulong LX = 2;

void ntt_dit4(mod *f, ulong ldn, int is)
//
// radix 4 decimation in time NTT
//
{
    const ulong n = (1UL<<ldn);

    revbin_permute(f, n);

    // n is not a power of 4, need a radix 2 step:
    if ( ldn & 1 )
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    ulong ldm = LX + (ldn&1);
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a2 = f[i1] * w2;
                mod a1 = f[i2] * w;
                mod a3 = f[i3] * w3;

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = t02 + t13;
                f[i2] = t02 - t13;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i1] = t02 + t13;
                f[i3] = t02 - t13;
            }

            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }
}
```

The function `mod::root2pow(x)` returns a primitive root of order 2^x. This is [FXT: `ntt_dit4` in `mod/nttdit4.cc`].

The radix-4 DIF variant is [FXT: `ntt_dif4` in `mod/nttdif4.cc`].

For applications of the NTT see the survey article [35] which also has a good bibliography.

4.4 Convolution with NTTs

The NTTs are natural candidates for (exact) integer convolutions, as used e.g. in (high precision) multiplications. One must keep in mind that 'everything is mod p', the largest value that can be represented is $p-1$. As an example consider the multiplication of n-digit radix R numbers[3]. 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[4]).

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[5] *modulus* must be an odd number.

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

4.5 The Chinese Remainder Theorem (CRT)

The Chinese remainder theorem (CRT):

Let $m_1, m_2, \ldots, m_f$ be pairwise relatively[6] prime (i.e. $gcd(m_i, m_j) = 1,\ \forall i \neq j$)
If $x \equiv x_i \ (mod\ m_i)\ i = 1, 2, \ldots, f$ then x is unique modulo the product $m_1 \cdot m_2 \cdot \ldots \cdot m_f$.

For only two moduli m_1, m_2 compute x as follows[7]:

Code 4.11 (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 4.12 (CRT) *Code to perform the CRT for several moduli:*

```
function crt(x[],m[],f)
{
    x1 := x[0]
    m1 := m[0]
```

[3] Multiplication is a convolution of the digits followed by the 'carry' operations.
[4] A radix R 'nine' is $R-1$, nine in radix 10 is 9.
[5] for length-2^k FFTs
[6] note that it is not assumed that any of the m_i is prime
[7] cf. [3]

```
    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
}
```

To see why these functions really work we have to formulate a more general CRT procedure that specializes to the functions above.

Define

$$T_i \quad := \quad \prod_{k!=i} m_k \tag{4.10}$$

and

$$\eta_i \quad := \quad T_i^{-1} \mod m_i \tag{4.11}$$

then for

$$X_i \quad := \quad x_i \, \eta_i \, T_i \tag{4.12}$$

one has

$$X_i \mod m_j \quad = \quad \begin{cases} x_i & \text{for} \quad j = i \\ 0 & \text{else} \end{cases} \tag{4.13}$$

and so

$$\sum_k X_k \quad = \quad x_i \mod m_i \tag{4.14}$$

For the special case of two moduli m_1, m_2 one has

$$T_1 \quad = \quad m_2 \tag{4.15}$$
$$T_2 \quad = \quad m_1 \tag{4.16}$$
$$\eta_1 \quad = \quad m_2^{-1} \mod m_1 \tag{4.17}$$
$$\eta_2 \quad = \quad m_1^{-1} \mod m_2 \tag{4.18}$$

which are related by[8]

$$\eta_1 \, m_2 + \eta_2 \, m_1 \quad = \quad 1 \tag{4.19}$$

$$\sum_k X_k \quad = \quad x_1 \, \eta_1 \, T_1 + x_2 \, \eta_2 \, T_2 \tag{4.20}$$
$$= \quad x_1 \, \eta_1 \, m_2 + x_2 \, \eta_2 \, m_1 \tag{4.21}$$
$$= \quad x_1 \, (1 - \eta_2 \, m_1) + x_2 \, \eta_2 \, m_1 \tag{4.22}$$
$$= \quad x_1 + (x_2 - x_1) \, (m_1^{-1} \mod m_2) \, m_1 \tag{4.23}$$

as given in the code. The operation count of the CRT implementation as given above is significantly better than that of a straight forward implementation.

[8] cf. extended Euclidean algorithm

4.6 A modular multiplication technique

When implementing a mod class on a 32 bit machine the following trick can be useful: It allows easy multiplication of two integers a, b modulo m even if the product $a \cdot b$ does not 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 $0 \le a, b < m$:

$$a \cdot b \quad = \quad \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m + \langle a \cdot b \rangle_m \tag{4.24}$$

rearranging and taking both sides modulo $z > m$:

$$\left\langle a \cdot b - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \quad = \quad \langle \langle a \cdot b \rangle_m \rangle_z \tag{4.25}$$

where the rhs. equals $\langle a \cdot b \rangle_m$ because $m < z$.

$$\langle a \cdot b \rangle_m \quad = \quad \left\langle \langle a \cdot b \rangle_z - \left\langle \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \right\rangle_z \tag{4.26}$$

the expression on the rhs. can be translated into a few lines of 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 )  // normalization 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 normalization is not 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:

Precompute: `float64 i = (float64)1/m;`

and replace the line `uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);`

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.

TBD: *montgomery mult.*

4.7 Number theoretic 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

$$\cos\frac{2\,\pi}{n} \equiv \frac{r^2+1}{2\,r} \tag{4.27}$$

$$i\,\sin\frac{2\,\pi}{n} \equiv \frac{r^2-1}{2\,r} \tag{4.28}$$

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.

Chapter 5

The Walsh transform and its relatives

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

TBD: *complex Walsh transform*

5.1 The Walsh transform: Walsh-Kronecker basis

Removing all *exp*(something) from the radix-2, decimation in time Fourier transform we get

```
void slow_walsh_wak_dit2(double *f, ulong ldn)
// (this routine has a problem)
{
    ulong n = (1<<(ulong)ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1<<ldm);
        const ulong mh = (m>>1);
        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r+j;
                const ulong t2 = t1+mh;
                double v = f[t2];
                double u = f[t1];
                f[t1] = u+v;
                f[t2] = u-v;
            }
        }
    }
}
```

The transform involves proportional $n \log_2(n)$ additions (and subtractions) and no multiplication at all. The transform is its own inverse, so there is nothing like the `is` in the FFT procedures here.

As the `slow` in the name shall suggest, the implementation as a problem as given. The memory access is highly non-local. Let's make a slight improvement: Here we just took the code 1.4 and threw away all trigonometric computations (and multiplications). But the swapping of the inner loops, that we did for the FFT in order to save trigonometric computations is now of no advantage anymore. So we try this piece of code:

```
template <typename Type>
void walsh_wak_dit2(Type *f, ulong ldn)
```

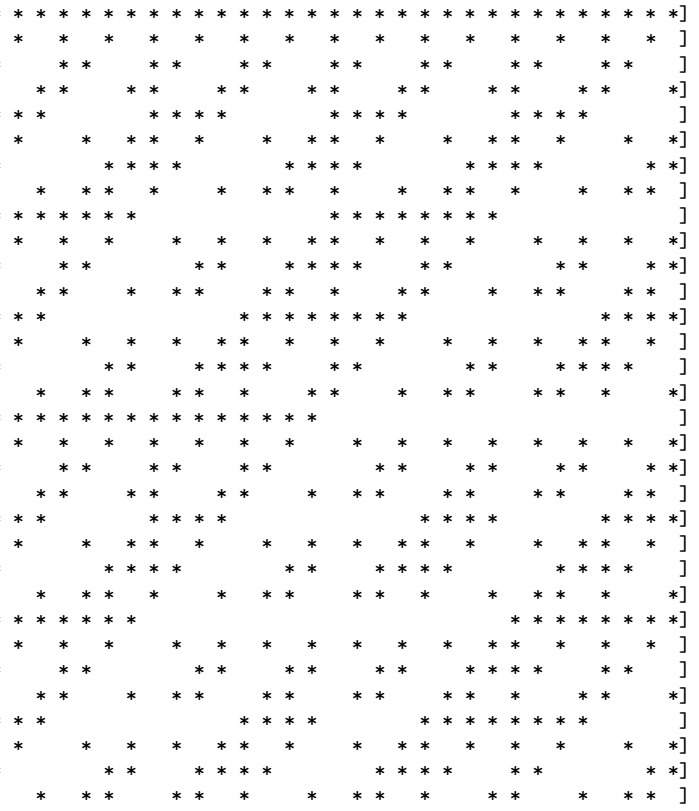


Figure 5.1: Basis functions for the Walsh transform (Walsh-Kronecker basis). Asterisks denote the value +1, blank entries denote −1.

```
// transform wrt. to walsh-kronecker basis (wak-functions)
// decimation in time (DIT) algorithm
{
    ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `walsh_wak_dit2` in `walsh/walshwak.h`]

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. [FXT: `walsh_wak_dit2` in `walsh/walshwak.h`]

The equivalent code for the decimation in frequency algorithm is

```
template <typename Type>
void walsh_wak_dif2(Type *f, ulong ldn)
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `walsh_wak_dif2` in `walsh/walshwak.h`]

The Walsh transform of integer input is integral, cf. section 6.6.

A function that computes the k-th base function of the transform is

```
template <typename Type>
void walsh_wak_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

[FXT: `walsh_wak_basefunc` in `walsh/walshbasefunc.h`]

5.2 The Kronecker product

The length-2 Walsh transform can be seen to be equivalent to the multiplication of a 2-component vector by the matrix

$$\mathbf{W}_2 \;=\; \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \tag{5.1}$$

The length-4 Walsh transform corresponds to

$$\mathbf{W}_4 \;=\; \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \tag{5.2}$$

One might be tempted to write

$$\mathbf{W}_4 \;=\; \begin{bmatrix} +\mathbf{W}_2 & +\mathbf{W}_2 \\ +\mathbf{W}_2 & -\mathbf{W}_2 \end{bmatrix} \tag{5.3}$$

This idea can indeed be turned into a well-defined notation which turns out to be quite powerful when dealing with orthogonal transforms and their fast algorithms.

Let $\mathbf{A}$ be an $m \times n$-matrix

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{5.4}$$

then the *Kronecker product* (or Tensor product) with a matrix $\mathbf{B}$ is

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} a_{0,0}\mathbf{B} & a_{0,1}\mathbf{B} & \cdots & a_{0,n-1}\mathbf{B} \\ a_{1,0}\mathbf{B} & a_{1,1}\mathbf{B} & \cdots & a_{1,n-1}\mathbf{B} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0}\mathbf{B} & a_{m-1,1}\mathbf{B} & \cdots & a_{m-1,n-1}\mathbf{B} \end{bmatrix} \tag{5.5}$$

There is no restriction on the dimensions of $\mathbf{B}$. If $\mathbf{B}$ is a $r \times s$-matrix then the dimensions of the given Kronecker product is $mr \times ns$ and $c_{k+ir,l+js} = a_{i,j}b_{k,l}$.

For a scalar factor α the relations

$$(\alpha\mathbf{A}) \otimes \mathbf{B} = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{5.6}$$
$$\mathbf{A} \otimes (\alpha\mathbf{B}) = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{5.7}$$

should be immediate.

The Kronecker product is not commutative, that is, $\mathbf{A} \otimes \mathbf{B} \neq \mathbf{B} \otimes \mathbf{A}$ in general. The following relations are the same as for the ordinary matrix product. Bilinearity (the matrices left and right from a plus sign must be of the same dimensions):

$$(\mathbf{A} + \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes \mathbf{C} + \mathbf{B} \otimes \mathbf{C} \tag{5.8}$$
$$\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C} \tag{5.9}$$

Associativity:

$$\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \tag{5.10}$$

The matrix product (indicated by a dot) of Kronecker products can be rewritten as

$$(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A} \cdot \mathbf{C}) \otimes (\mathbf{B} \cdot \mathbf{D}) \tag{5.11}$$
$$(\mathbf{L}_1 \otimes \mathbf{R}_1) \cdot (\mathbf{L}_2 \otimes \mathbf{R}_2) \cdot \ldots \cdot (\mathbf{L}_n \otimes \mathbf{R}_n) = (\mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \ldots \cdot \mathbf{L}_n) \otimes (\mathbf{R}_1 \cdot \mathbf{R}_2 \cdot \ldots \cdot \mathbf{R}_n) \tag{5.12}$$
$$(\mathbf{A} \cdot \mathbf{B}) \otimes (\mathbf{C} \cdot \mathbf{D}) = (\mathbf{A} \otimes \mathbf{C}) \cdot (\mathbf{B} \otimes \mathbf{D}) \tag{5.13}$$
$$(\mathbf{L}_1 \cdot \mathbf{R}_1) \otimes (\mathbf{L}_2 \cdot \mathbf{R}_2) \otimes \ldots \otimes (\mathbf{L}_n \cdot \mathbf{R}_n) = (\mathbf{L}_1 \otimes \mathbf{L}_2 \otimes \ldots \otimes \mathbf{L}_n) \cdot (\mathbf{R}_1 \otimes \mathbf{R}_2 \otimes \ldots \otimes \mathbf{R}_n) \tag{5.14}$$

Here the matrices left and right from a dot must be compatible for ordinary matrix multiplication.

One has

$$(\mathbf{A} \otimes \mathbf{B})^T = \mathbf{A}^T \otimes \mathbf{B}^T \tag{5.15}$$
$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{5.16}$$

Back to the Walsh transform, we have $\mathbf{W}_1 = [1]$ and for $n = 2^k$, $n > 1$:

$$\mathbf{W}_n = \begin{bmatrix} +\mathbf{W}_{n/2} & +\mathbf{W}_{n/2} \\ +\mathbf{W}_{n/2} & -\mathbf{W}_{n/2} \end{bmatrix} = \mathbf{W}_2 \otimes \mathbf{W}_{n/2} \tag{5.17}$$

In order to see that this relation is the statement of a fast algorithm split the (to be transformed) vector x into halves

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{5.18}$$

and write

$$\mathbf{W}_n\, x \quad = \quad \left[\begin{array}{l} \mathbf{W}_{n/2}\, x_0 + \mathbf{W}_{n/2}\, x_1 \\ \mathbf{W}_{n/2}\, x_0 - \mathbf{W}_{n/2}\, x_1 \end{array} \right] = \left[\begin{array}{l} \mathbf{W}_{n/2}\, (x_0 + x_1) \\ \mathbf{W}_{n/2}\, (x_0 - x_1) \end{array} \right] \tag{5.19}$$

That is, a length-n transform can be computed by two length-$n/2$ transforms of the sum and difference of the first and second half of x.

Now when $\mathbf{A} = \mathbf{B}$ in relation 5.16 we have $(\mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$, $(\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$ and so on. Using a notation equivalent to the sum (or product) sign this is

$$\left(\bigotimes_{k=1}^{n} \mathbf{A} \right)^{-1} \quad = \quad \bigotimes_{k=1}^{n} \mathbf{A}^{-1} \tag{5.20}$$

Using the notation for the Walsh transform

$$\mathbf{W}_n \quad = \quad \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2 \tag{5.21}$$

(where the empty product equals $[1] = \mathbf{W}_1$) and

$$\mathbf{W}_n^{-1} \quad = \quad \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2^{-1} \tag{5.22}$$

The latter relation isn't that exciting as $\mathbf{W}_2^{-1} = \mathbf{W}_2$, however, it obviously also holds when the inverse transform is different from the forward transform. Thereby, given a fast algorithm for some transform in form of a Kronecker product, the fast algorithm for the backward transform is immediate.

The direct sum of two matrices is defined as

$$\mathbf{A} \oplus \mathbf{B} \quad := \quad \left[\begin{array}{ll} \mathbf{A} & 0 \\ 0 & \mathbf{B} \end{array} \right] \tag{5.23}$$

We have

$$\bigoplus_{k=1}^{n} \mathbf{A} \quad := \quad \mathbf{I}_n \otimes \mathbf{A} \tag{5.24}$$

where $\mathbf{I}_n$ is the $n \times n$-identity matrix.

5.3 Computing the Walsh transform faster

All operations necessary for the Walsh transform are cheap: loads, stores, additions and subtractions. The memory access pattern is a major concern with direct mapped cache, as we have verified comparing the first two implementations in this chapter. Even the one found to be superior due to its more localized access is guaranteed to have a performance problem as soon as the array is long enough: all accesses are separated by a power-of-two distance and cache misses will occur beyond a certain limit. Rather bizarre attempts like inserting 'pad data' have been reported in order to mitigate the problem. The Gray code permutation described in section 7.8 allows a very nice and elegant solution where the sub-arrays are always accessed in mutually reversed order.

```
template <typename Type>
void walsh_gray(Type *f, ulong ldn)
// decimation in frequency (DIF) algorithm
{
    const ulong n = (1UL<<ldn);
```

```
    for (ulong ldm=ldn; ldm>0; --ldm) // dif
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
            for (  ;  t1<t2;  ++t1,--t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `walsh_gray` in `walsh/walshgray.h`]

The transform is not self-inverse, however its inverse can be implemented trivially:

```
template <typename Type>
void inverse_walsh_gray(Type *f, ulong ldn)
// decimation in time (DIT) algorithm
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm) // dit
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
            for (  ;  t1<t2;  ++t1,--t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `inverse_walsh_gray` in `walsh/walshgray.h`]

The relation between `walsh_wak()` and `walsh_gray()` is that

```
   inverse_gray_permute(f, n);
   walsh_gray(f, ldn);
   grs_negate(f, n);
```

is equivalent to the call `walsh_wak(f, ldn)`. The third line is a necessary fix-up for certain elements that have the wrong sign if uncorrected:

```
template <typename Type>
void grs_negate(Type *f, ulong n)
// negate elements at indices where the
// Golay-Rudin-Shapiro is negative.
{
    for (ulong k=0; k<n; ++k)
    {
        if ( grs_negative_q(k) )  f[k] = -f[k];
    }
}
```

[FXT: `grs_negate` in `aux1/grsnegate.h`] The function `grs_negative_q()` is described in section 8.12. Using Q for the `grs_negative_q`-line we have

$$W_k \;=\; Q\,W_g\,G^{-1} \;=\; G\,W_g^{-1}\,Q \tag{5.25}$$

The same idea can be used with the Fast Fourier Transform. However, the advantage of the improved access pattern is usually more than compensated by the increased number of sin/cos-computations (the twiddle factors appear reordered so $n \cdot \log n$ instead of n computations are necessary) cf. [FXT: file `fft/gfft.cc`].

5.4 Dyadic convolution

Walsh's convolution has XOR where the usual one has plus

Using

```
template <typename Type>
void dyadic_convolution(Type * restrict f, Type * restrict g, ulong ldn)
{
    walsh_wak(f, ldn);
    walsh_wak(g, ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    walsh_wak(g, ldn);
}
```

one gets the so called *dyadic* convolution defined by

$$h = a \circledast_{\wedge} b \tag{5.26}$$

$$h_\tau := \sum_{x \wedge y = \tau} a_x \, b_y \tag{5.27}$$

where the symbol $\wedge$ stands for the XOR operator.

The table equivalent to 2.1 is

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

 4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
 5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
 6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
 7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
 9:   9  8 11 10  13 12 15 14   1  0  3  2   5  4  7  6
10:  10 11  8  9  14 15 12 13   2  3  0  1   6  7  4  5
11:  11 10  9  8  15 14 13 12   3  2  1  0   7  6  5  4

12:  12 13 14 15   8  9 10 11   4  5  6  7   0  1  2  3
13:  13 12 15 14   9  8 11 10   5  4  7  6   1  0  3  2
14:  14 15 12 13  10 11  8  9   6  7  4  5   2  3  0  1
15:  15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

The nearest equivalent to the acyclic convolution can be computed using a sequence that has both prepended and appended runs of $n/2$ zeros:

```
 +--   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:    0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:    1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
 2:    2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:    3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

 4:    4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
 5:    5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
 6:    6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
 7:    7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   16 17 18 19  20 21 22 23  24 25 26 27  28 29 30 31
```

```
 9:   17 16 19 18  21 20 23 22  25 24 27 26  29 28 31 30
10:   18 19 16 17  22 23 20 21  26 27 24 25  30 31 28 29
11:   19 18 17 16  23 22 21 20  27 26 25 24  31 30 29 28

12:   20 21 22 23  16 17 18 19  28 29 30 31  24 25 26 27
13:   21 20 23 22  17 16 19 18  29 28 31 30  25 24 27 26
14:   22 23 20 21  18 19 16 17  30 31 28 29  26 27 24 25
15:   23 22 21 20  19 18 17 16  31 30 29 28  27 26 25 24
```

An scheme similar to that of the weighted convolution can be obtained via

```
    walsh_wal_dif2_core(f, ldn);
    walsh_wal_dif2_core(g, ldn);
    ulong n = (1UL<<ldn);
    for (ulong i=0,j=n-1; i<j; --j,++i)  fht_mul(f[i], f[j], g[i], g[j], 0.5);
    walsh_wal_dit2_core(g, ldn);
```

where `fht_mul` is the operation used for the convolution with fast Hartley transforms ([FXT: `fht_mul` in `include/fhtmulsqr.h`]):

```
static inline  void
fht_mul(double xi, double xj, double &yi, double &yj, double v)
{
    double h1p = xi,  h1m = xj;
    double s1 = h1p + h1m,  d1 = h1p - h1m;
    double h2p = yi,  h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

One gets (negative contributions to a bucket have a minus appended):

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

 4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
 5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
 6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
 7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   8  9 10 11  12 13 14 15   0- 1- 2- 3-  4- 5- 6- 7-
 9:   9  8 11 10  13 12 15 14   1- 0- 3- 2-  5- 4- 7- 6-
10:  10 11  8  9  14 15 12 13   2- 3- 0- 1-  6- 7- 4- 5-
11:  11 10  9  8  15 14 13 12   3- 2- 1- 0-  7- 6- 5- 4-

12:  12 13 14 15   8  9 10 11   4- 5- 6- 7-  0- 1- 2- 3-
13:  13 12 15 14   9  8 11 10   5- 4- 7- 6-  1- 0- 3- 2-
14:  14 15 12 13  10 11  8  9   6- 7- 4- 5-  2- 3- 0- 1-
15:  15 14 13 12  11 10  9  8   7- 6- 5- 4-  3- 2- 1- 0-
```

Speedup using the Gray-variant

The `walsh_gray()`-variant and its inverse can be utilized for a faster implementation of the dyadic convolution:

```
template <typename Type>
void dyadic_convolution(Type * restrict f, Type * restrict g, ulong ldn)
{
    walsh_gray(f, ldn);
    walsh_gray(g, ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    for (ulong k=0; k<n; ++k)  if ( grs_negative_q(k) )  g[k] = -g[k];
    inverse_walsh_gray(g, ldn);
}
```

The observed saving is about 25 percent for large arrays:

```
ldn=20  n=1048576 repetitions: m=5 memsize=16384 kiloByte
                reverse(f,n2);     dt=0.0418339      rel=       1
              walsh_wak(f,ldn);    dt=0.505863       rel= 12.0922
             walsh_gray(f,ldn);    dt=0.378223       rel= 9.04108
  dyadic_convolution(f, g, ldn);   dt= 1.54834       rel= 37.0117 << wak
  dyadic_convolution(f, g, ldn);   dt= 1.19474       rel= 28.5436 << gray
ldn=21  n=2097152 repetitions: m=5 memsize=32768 kiloByte
                reverse(f,n2);     dt=0.0838011      rel=       1
              walsh_wak(f,ldn);    dt=1.07741        rel= 12.8567
             walsh_gray(f,ldn);    dt=0.796644       rel= 9.50636
  dyadic_convolution(f, g, ldn);   dt=3.28062        rel= 39.1477 << wak
  dyadic_convolution(f, g, ldn);   dt=2.49583        rel= 29.7401 << gray
```

Analogue tables for matrix multiplication

It may be interesting to note that the table for matrix multiplication (4x4 matrices) looks like

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  .  .  .   4  .  .  .   8  .  .  .  12  .  .  .
 1:   1  .  .  .   5  .  .  .   9  .  .  .  13  .  .  .
 2:   2  .  .  .   6  .  .  .  10  .  .  .  14  .  .  .
 3:   3  .  .  .   7  .  .  .  11  .  .  .  15  .  .  .

 4:   .  0  .  .   .  4  .  .   .  8  .  .   . 12  .  .
 5:   .  1  .  .   .  5  .  .   .  9  .  .   . 13  .  .
 6:   .  2  .  .   .  6  .  .   . 10  .  .   . 14  .  .
 7:   .  3  .  .   .  7  .  .   . 11  .  .   . 15  .  .

 8:   .  .  0  .   .  .  4  .   .  .  8  .   .  . 12  .
 9:   .  .  1  .   .  .  5  .   .  .  9  .   .  . 13  .
10:   .  .  2  .   .  .  6  .   .  . 10  .   .  . 14  .
11:   .  .  3  .   .  .  7  .   .  . 11  .   .  . 15  .

12:   .  .  .  0   .  .  .  4   .  .  .  8   .  .  . 12
13:   .  .  .  1   .  .  .  5   .  .  .  9   .  .  . 13
14:   .  .  .  2   .  .  .  6   .  .  . 10   .  .  . 14
15:   .  .  .  3   .  .  .  7   .  .  . 11   .  .  . 15
```

But when the problem is made symmetric, i.e. the second matrix is indexed in transposed order, we get:

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  .  .  .   4  .  .  .   8  .  .  .  12  .  .  .
 1:   .  0  .  .   .  4  .  .   .  8  .  .   . 12  .  .
 2:   .  .  0  .   .  .  4  .   .  .  8  .   .  . 12  .
 3:   .  .  .  0   .  .  .  4   .  .  .  8   .  .  . 12

 4:   1  .  .  .   5  .  .  .   9  .  .  .  13  .  .  .
 5:   .  1  .  .   .  5  .  .   .  9  .  .   . 13  .  .
 6:   .  .  1  .   .  .  5  .   .  .  9  .   .  . 13  .
 7:   .  .  .  1   .  .  .  5   .  .  .  9   .  .  . 13

 8:   2  .  .  .   6  .  .  .  10  .  .  .  14  .  .  .
 9:   .  2  .  .   .  6  .  .   . 10  .  .   . 14  .  .
10:   .  .  2  .   .  .  6  .   .  . 10  .   .  . 14  .
11:   .  .  .  2   .  .  .  6   .  .  . 10   .  .  . 14

12:   3  .  .  .   7  .  .  .  11  .  .  .  15  .  .  .
13:   .  3  .  .   .  7  .  .   . 11  .  .   . 15  .  .
14:   .  .  3  .   .  .  7  .   .  . 11  .   .  . 15  .
15:   .  .  .  3   .  .  .  7   .  .  . 11   .  .  . 15
```

Thereby dyadic convolution can be used to compute matrix products. The 'un-polished' algorithm is $\sim n^3 \cdot \log n$ as with the FT (-based correlation).

If the above was $F \cdot G$ then $G \cdot F$ is

Multiplication of quaternions

Quaternion multiplication can be achieved in eight real multiplication using the dyadic convolution: The scheme in figure 5.2 suggests to use the dyadic convolution with bucket zero negated as a starting

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0  1  2  3   .  .  .  .   .  .  .  .   .  .  .  .
1:   4  5  6  7   .  .  .  .   .  .  .  .   .  .  .  .
2:   8  9 10 11   .  .  .  .   .  .  .  .   .  .  .  .
3:  12 13 14 15   .  .  .  .   .  .  .  .   .  .  .  .

4:   .  .  .  .   0  1  2  3   .  .  .  .   .  .  .  .
5:   .  .  .  .   4  5  6  7   .  .  .  .   .  .  .  .
6:   .  .  .  .   8  9 10 11   .  .  .  .   .  .  .  .
7:   .  .  .  .  12 13 14 15   .  .  .  .   .  .  .  .

8:   .  .  .  .   .  .  .  .   0  1  2  3   .  .  .  .
9:   .  .  .  .   .  .  .  .   4  5  6  7   .  .  .  .
10:  .  .  .  .   .  .  .  .   8  9 10 11   .  .  .  .
11:  .  .  .  .   .  .  .  .  12 13 14 15   .  .  .  .

12:  .  .  .  .   .  .  .  .   .  .  .  .   0  1  2  3
13:  .  .  .  .   .  .  .  .   .  .  .  .   4  5  6  7
14:  .  .  .  .   .  .  .  .   .  .  .  .   8  9 10 11
15:  .  .  .  .   .  .  .  .   .  .  .  .  12 13 14 15
```

```
+--  0   1   2   3      +--  0   1   2   3        +--  0   1   2   3
|                       |                        |
0:   0   1   2   3      0:  -0   1   2   3      1 0:   0*  1   2   3
1:   1   0   3   2      1:   1  -0   3   2      i 1:   1  -0   3  -2*
2:   2   3   0   1      2:   2   3  -0   1      j 2:   2  -3* -0   1
3:   3   2   1   0      3:   3   2   1  -0      k 3:   3   2  -1* -0
```

Figure 5.2: Scheme for the length-4 dyadic convolution (left), same with bucket zero negated (middle) and the multiplication table for the units of the quaternions (entries correspond to 'row unit' times 'column unit', right). The asterisks mark those entries where the sign is different from the scheme in the middle.

point which costs 4 multiplications. Some entries have to be corrected then which costs four more multiplications.

```
// f[] == [ re1, i1, j1, k1 ]
// g[] == [ re2, i2, j2, k2 ]
c0 := f[0] * g[0]
c1 := f[3] * g[2]
c2 := f[1] * g[3]
c3 := f[2] * g[1]

// length-4 dyadic convolution:
walsh(f[]);
walsh(g[]);
for i:=0 to 3  g[i] := (f[i] * g[i])
walsh(g[]);

// normalization and correction:
g[0] :=   2 * c0 - g[0] / 4
g[1] := - 2 * c1 + g[1] / 4
g[2] := - 2 * c2 + g[2] / 4
g[3] := - 2 * c3 + g[3] / 4
```

The algorithm is taken from [60] which also gives a second variant.

The complex multiplication by three real multiplications corresponds to one length-2 Walsh dyadic convolution and the correction for the product of the imaginary units:

```
// f[] == [ re1, im1 ]
// g[] == [ re2, im2 ]
c0 := f[1] * g[1]  // == im1 * im2
// length-2 dyadic convolution:
{ f[0], f[1] } := { f[0]+f[1], f[0]-f[1] }
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
g[0] := f[0] * g[0]
g[1] := f[1] * g[1]
```

```
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
// normalization:
f[0] := f[0] / 2
g[0] := g[0] / 2
// correction:
g[0] := - 2 * c0 + g[0]
// here:  g[] == [ re1 * re2 - im1 * im2,  re1 * im2 + im1 * re2 ]
```

For complex numbers of high-precision multiplication is asymptotically equivalent to two real multiplications as one FFT based (complex linear) convolution can be used for the computation. Similarly, high precision quaternion multiplication is as expensive as four real multiplications.

5.5 The Walsh transform: Walsh-Paley basis

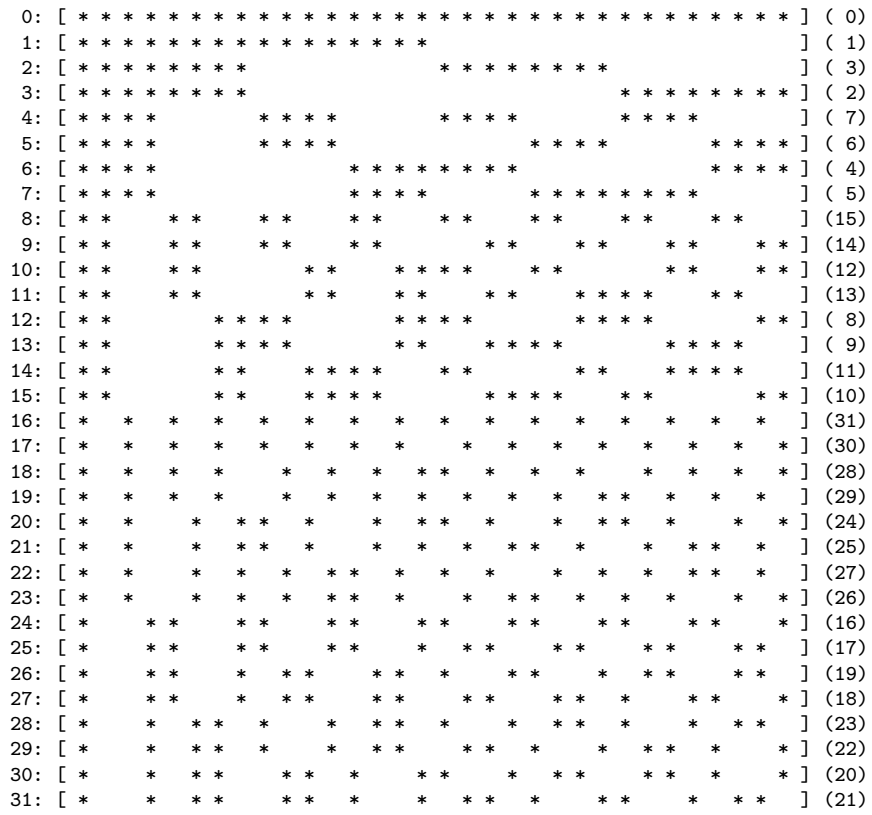


Figure 5.3: Basis functions for the Walsh transform (Walsh-Paley basis). Asterisks denote the value +1, blank entries denote −1.

A Walsh transform with a different (ordering of the) basis can be obtained by

```
template <typename Type>
void walsh_pal(Type *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    walsh_wak(f, ldn);
// =^=
//    walsh_wak(f, ldn);
//    revbin_permute(f, n);
}
```

[FXT: `walsh_pal` in `walsh/walshpal.h`] Actually one can also `revbin_transform` before the transform. That is,

$$W_p \;=\; W_k\, R \;=\; R\, W_k \tag{5.28}$$

One has for W_p

$$W_p \;=\; G\, W_p\, G \;=\; G^{-1}\, W_p\, G^{-1} \tag{5.29}$$
$$\;=\; Z\, W_p\, Z \;=\; Z^{-1}\, W_p\, Z^{-1} \tag{5.30}$$

A function that computes the k-th base function of the transform is

```
template <typename Type>
void walsh_pal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

[FXT: `walsh_pal_basefunc` in `walsh/walshbasefunc.h`]

5.6 Sequency ordered Walsh transforms

A term analogue to the frequency of the Fourier basis functions is the so called *sequency* of the Walsh functions, the number[1] of the changes of sign of the individual functions. If one wants the basis functions ordered by their sequency one can use the procedure

```
    const ulong n = (1UL<<ldn);
    walsh_wak_dif2(f, ldn);
    revbin_permute(f, n);
    inverse_gray_permute(f, n);
```

That is

$$W_w \;=\; G^{-1}\, R\, W_k \;=\; W_k\, R\, G \tag{5.31}$$

A function that computes the k-th base function of the transform is

```
template <typename Type>
void walsh_wal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n)+1);
    k = gray_code(k);
//    // =^=
//    k = revbin(k, ld(n));
//    k = green_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

[FXT: `walsh_wal_basefunc` in `walsh/walshbasefunc.h`]

A version of the transform that avoids the Gray-permutation is based on

[1] Note that the sequency of a signal with frequency f usually is $2\,f$.

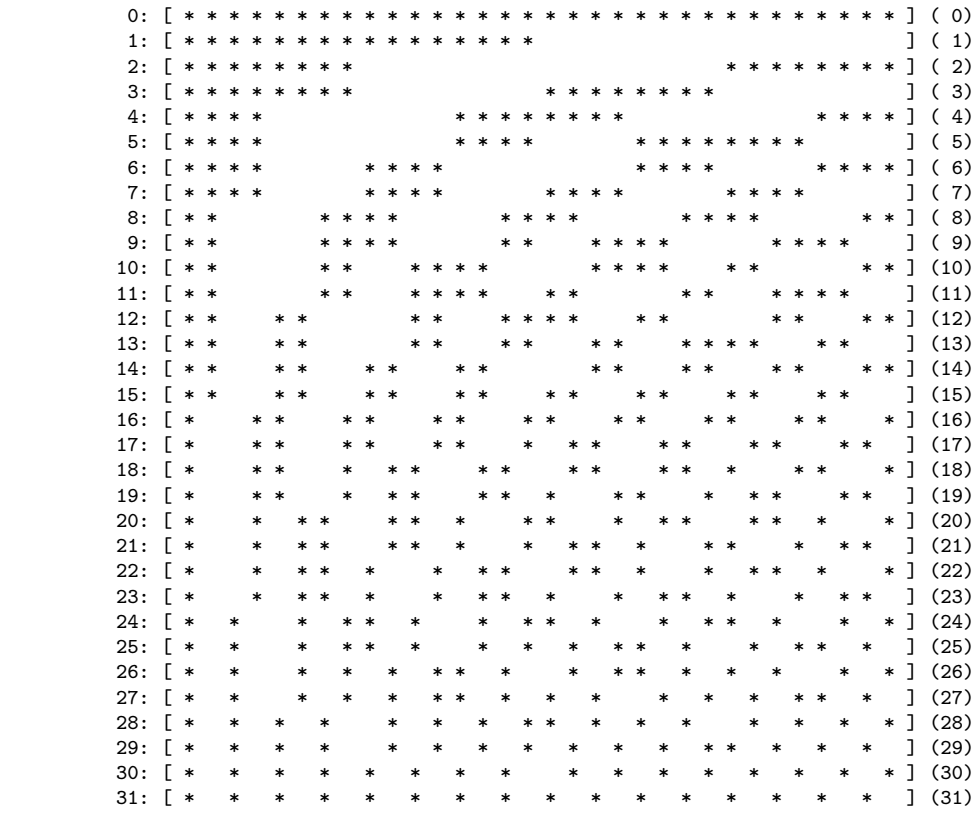


Figure 5.4: Basis functions for the sequency-ordered Walsh transform (Walsh-Kacmarz basis). Asterisks denote the value +1, blank entries denote −1.

```
template <typename Type>
void walsh_wal_dif2_core(Type *f, ulong ldn)
// decimation in frequency (DIF) algorithm
// gray_permute is absorbed
//
// walsh_wal(f, ldn)
//   =^=
// revbin_permute(f, n);  walsh_wal_dif2_core(f, ldn);
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=2; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong j;
            for (j=0; j<m4; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }

            for (  ; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
```

```
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = v - u;  // reversed
            }
        }
    }

    if ( ldn )
    {
        // ulong ldm=1;
        const ulong m = 2; //(1UL<<ldm);
        const ulong mh = 1; //(m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong j = 0;
//            for (ulong j=0; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The transform still needs the revbin-permutation:

```
template <typename Type>
inline void walsh_wal(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dif2_core(f, ldn);
//  =^=
//    walsh_wal_dit2_core(f, ldn);
//    revbin_permute(f, (1UL<<ldn));
}
```

A decimation in time (DIT) version of the core-routine is also given in [FXT: file `walsh/walshwal.h`]. The procedure `gray_permute()` is introduced in section 7.8.

For the variant

```
walsh_gray(f, ldn);
grs_negate(f, n);
revbin_permute(f, n);
```

is equivalent to `walsh_wal(f, ldn)` and might be faster for large arrays. We have

$$W_w \;=\; R\,Q\,W_g \;=\; W_g^{-1}\,R\,Q \tag{5.32}$$

An alternative ordering of the base functions (first even sequencies ascending then odd sequencies descending [FXT: `walsh_wal_rev` in `walsh/walshwalrev.h`]) can be obtained by either of

```
    revbin_permute(f, 1UL<<ldn);
    gray_permute(f, n);
    walsh_wak(f,ldn);
```

```
    walsh_wak(f,ldn);
    inverse_gray_permute(f, n);
    revbin_permute(f, 1UL<<ldn);
```

```
    zip_rev(f, n);
    walsh_wal(f, ldn);
```

```
    walsh_wal(f, ldn);
    unzip_rev(f, n);
```

```
    walsh_wak(f, ldn);
```

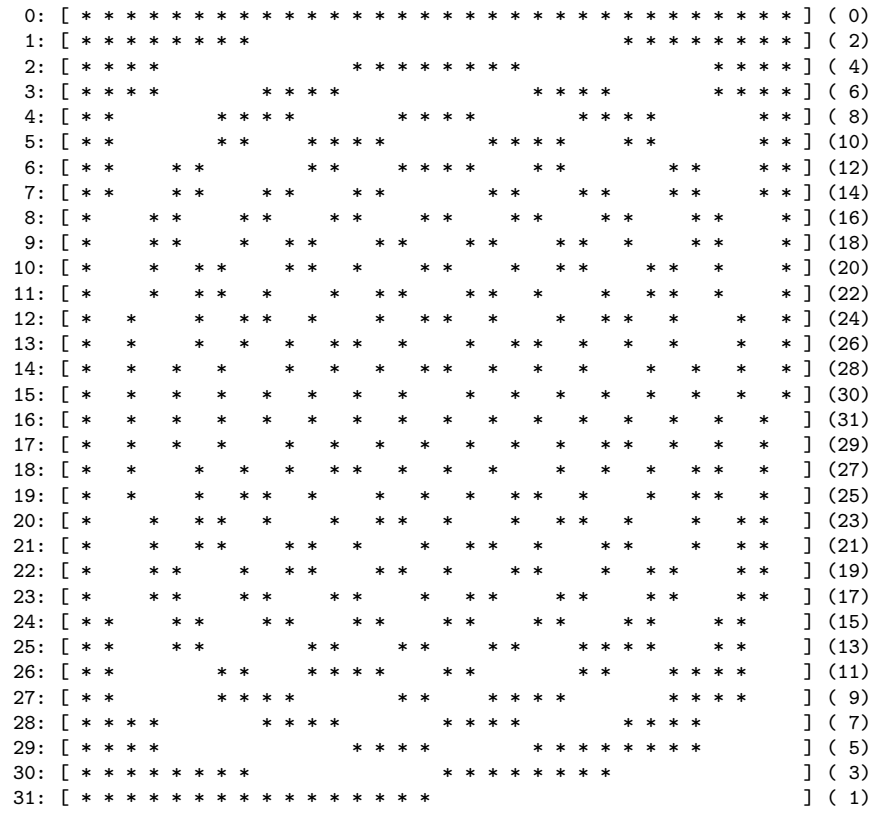


Figure 5.5: Basis functions for the reversed sequency ordered Walsh transform. Asterisks denote the value $+1$, blank entries denote -1.

```
inverse_gray_permute(f, n);
revbin_permute(f, n);
revbin_permute(f, 1UL<<ldn);
walsh_gray(f, ldn);
grs_negate(f, n);
```

That is,

$$\begin{aligned} W_{\bar{w}} &= W_k\, G\, R \;=\; R\, G^{-1}\, W_k && (5.33)\\ &= W_w\, \bar{Z} \;=\; \bar{Z}^{-1}\, W_w && (5.34)\\ &= Q\, W_g\, R && (5.35) \end{aligned}$$

However, an implementation that is more efficient uses the core-routines that have the Gray-permutation 'absorbed':

```
template <typename Type>
inline void walsh_wal_rev(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dit2_core(f, ldn);
    // =^=
//    walsh_wal_dif2_core(f, ldn);
//    revbin_permute(f, n);
}
```

[FXT: `walsh_wal_rev` in `walsh/walshwalrev.h`].

This implementation uses the fact that

$$\bar{W_w} \quad = \quad R\,W_w\,R \tag{5.36}$$

We can do still better:

```
    revbin_permute(f, n);
    walsh_gray(f, ldn);
    grs_negate(f, n);
```

gives the same transform.

Similar relations as for the transform with Walsh-Paley basis (5.29 and 5.30) hold for W_w:

$$W_w \quad = \quad G\,W_w\,G \;=\; G^{-1}\,W_w\,G^{-1} \tag{5.37}$$

$$\quad = \quad \bar{Z}\,W_w\,\bar{Z} \;=\; \bar{Z}^{-1}\,W_w\,\bar{Z}^{-1} \tag{5.38}$$

The k-th base function of the transform can be computed as

```
template <typename Type>
void walsh_wal_rev_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    k = gray_code(k);
    // =^=
//    k = green_code(k);
//    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

[FXT: `walsh_wal_rev_basefunc` in `walsh/walshbasefunc.h`]

The next variant of the Walsh transform has the interesting feature that the basis functions for a length-n transform have only sequencies $n/2$ and $n/2-1$ at the even and odd indices, respectively. The transform is self-inverse (the basis is shown in figure 5.6) and can be obtained via

```
template <typename Type>
void walsh_q1(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    grs_negate(f, n);
    walsh_gray(f, ldn);
    revbin_permute(f, n);
}
```

[FXT: `walsh_q1` in `walsh/walshq.h`]

A different transform with sequency $n/2$ for the first half of the basis, sequency $n/2-1$ for the second half ([FXT: `walsh_q2` in `walsh/walshq.h`], basis shown in figure 5.7) is computed by

```
template <typename Type>
void walsh_q2(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    revbin_permute(f, n);
    grs_negate(f, n);
    walsh_gray(f, ldn);
// =^=
//    grs_negate(f, n);
//    revbin_permute(f, n);
//    walsh_gray(f, ldn);
}
```

```
 0: [ * * *   * *   * * * *       *   * * *   * *   *       * * *   * ] (16)
 1: [ * * *   * *   * * * *       *         *     *   * * *       *   ] (15)
 2: [ * * *   * *   *       * * *   *       *     *         * * *   * ] (16)
 3: [       *     *   * * *       *         *     *         * * *   * ] (15)
 4: [ * * *       *         *     *   * * *       *   * * *   * *   * ] (16)
 5: [ * * *       *         *     *         * * *   *       *     *   ] (15)
 6: [       * * *   *       *     *   * * *       *         *     *   ] (16)
 7: [ * * *       *   * * *   * *   * * * *       *         *     *   ] (15)
 8: [ * *   *       * * *   * * * *   * *   *       *     *         * ] (16)
 9: [ * *   *       * * *   * * * *       *   * * *   * *   * * * *   ] (15)
10: [ * *   *       *     *         *     *   * * *       *         * ] (16)
11: [     *   * * *   * *   * * * *       *   * * *       *         * ] (15)
12: [     *         * * *   *       *     *         *     *   * * *   ] (16)
13: [     *         * * *   *       * * *   * * * *   * *   *       * ] (15)
14: [ * *   * * * *   * *   *       *     *         * * *   *       * ] (16)
15: [     *         *     *   * * *       *         * * *   *       * ] (15)
16: [ *       *   * * *         *     *       *   * *   * * * *   * * ] (16)
17: [ *       *   * * *         *       * * *   *     *         *     ] (15)
18: [ *       *   * *   * * * *   * *   * * *   *       * * * *   * * ] (16)
19: [   * * *   *     *         *       * * *   *       * * * *   * * ] (15)
20: [ *         *       * * *   *     *         *     *       *   * * ] (16)
21: [ *         *       * * *   *       * * * *   * *   * * *   *     ] (15)
22: [   * * * *   * *   * * *   *     *         *       * * *   *     ] (16)
23: [ *         *     *       *   * * *         *       * * *   *     ] (15)
24: [   *     *         *       * * *   *     *       *   * * *       ] (16)
25: [   *     *         *       * * * *   * *   * * *   *       * * *   ] (15)
26: [   *     *       *   * * *       *   * *   * * * *   * * *       ] (16)
27: [ *   * *   * * *   *       * * * *   * *   * * * *   * * *       ] (15)
28: [ *   * * *         *     *       *   * * *       *   * *   * * * ] (16)
29: [ *   * * *         *     *         *       * * *   *     *       ] (15)
30: [   *       * * *   *     *       *   * * *         *     *       ] (16)
31: [ *   * * *       *   * *   * * * *   * * *         *     *       ] (15)
```

Figure 5.6: Basis functions for a self-inverse Walsh transform that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

One has:

$$W_{q2} \quad = \quad R\, W_{q1}\, R \tag{5.39}$$

The base functions of the transforms can be computed as

```
template <typename Type>
void walsh_q1_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = gray_code(k);
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

and

```
template <typename Type>
void walsh_q2_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = revbin(k, ld(n));
    k = gray_code(k);
```

```
 0: [ * * *   * *   * * * *       *   * * *   * *   *       * * *   * ] (16)
 1: [ * * *   * *   *       * * *   *       *     *       *       * * *   * ] (16)
 2: [ * * *       *       *     *   *   * * *       *   * * *   * *   * ] (16)
 3: [       * * *   *       *     *   *   * * *       *       *     *   ] (16)
 4: [ * *   *       * * *   * * * *   * *   *       *     *       *       * ] (16)
 5: [ * *   *       *     *       *     *   *   * * *       *       *       * ] (16)
 6: [     *       * * *   *       *     *       *     *     *   * * *   ] (16)
 7: [ * *   * * * *   * *   *       *     *       *       * * *   *       * ] (16)
 8: [ *       *   * * *         *     *       *   *   * *   * * * *   * * ] (16)
 9: [ *       *   * *   * * * *   * *   * * *   *       * * * *   * * ] (16)
10: [ *         *       * * *   *     *         *     *       *   * * ] (16)
11: [   * * * *   * *   * * *   *     *         *       * * *   *     ] (16)
12: [   *     *         *       * * *   *     *       *   * * *       ] (16)
13: [   *     *       *   * * *       *   * *   * * * *   * * *       ] (16)
14: [ *   * * *         *     *       *   * * *       *   * *   * * * ] (16)
15: [   *       * * *   *     *       *   * * *         *     *       ] (16)
16: [ *   * * *       *   * *   * * * *   * * *         *     *       ] (15)
17: [ *   * * *         *     *         *       * * *   *     *       ] (15)
18: [ *   * *   * * *   *       * * * *   * *   * * * *   * * *       ] (15)
19: [   *     *         *       * * * *   * *   * * *   *       * * * ] (15)
20: [ *         *     *       *   * * *         *       * * *   *     ] (15)
21: [ *         *       * * *   *       * * * *   * *   * * *   *     ] (15)
22: [   * * *   *     *         *       * * *   *       * * * *   * * ] (15)
23: [ *       *   * * *         *       * * *   *     *         *     ] (15)
24: [     *         *     *   * * *       *         * * *   *       * ] (15)
25: [     *         * * *   *       * * *   * * * *   * *   *       * ] (15)
26: [     *   * * *   * *   * * * *       *   * * *       *         * ] (15)
27: [ * *   *       * * *   * * * *       *   * * *   * *   * * * *   ] (15)
28: [ * * *       *   * * *   * *   * * * *       *         *     *   ] (15)
29: [ * * *       *         *     *         * * *   *       *     *   ] (15)
30: [       *     *     *   * * *       *         *     *         * * *   * ] (15)
31: [ * * *   * *   * * * *       *         *     *   * * *       *   ] (15)
```

Figure 5.7: Basis functions for a self-inverse Walsh transform (second form) that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

```
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

[FXT: `walsh_q1_basefunc` and `walsh_q2_basefunc` in `walsh/walshbasefunc.h`]

5.7 The slant transform

The slant transform can be implemented using a Walsh Transform and just a little pre/post-processing:

```
void slant(double *f, ulong ldn)
// slant transform
{
    walsh_wak(f, ldn);
    ulong n = 1UL<<ldn;
    for (ulong ldm=0; ldm<ldn-1; ++ldm)
    {
        ulong m = 1UL<<ldm;  // m = 1, 2, 4, 8, ..., n/4
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
```

```
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = a * f1 - b * f2;
            f[t2] = b * f1 + a * f2;
        }
    }
}
```

The `ldm`-loop executes `ldn`-1 times, the inner loop is executed is $n/2 - 1$ times. That is, apart from the Walsh transform only an amount of work linear with the array size has to be done. [FXT: `slant` in `walsh/slant.cc`]

The inverse transform is:

```
void inverse_slant(double *f, ulong ldn)
// inverse of slant()
{
    ulong n = 1UL<<ldn;
    ulong ldm=ldn-2;
    do
    {
        ulong m = 1UL<<ldm;  // m = n/4, n/2, ..., 4, 2, 1
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = b * f2 + a * f1;
            f[t2] = a * f2 - b * f1;
        }
    }
    while ( ldm-- );

    walsh_wak(f, ldn);
}
```

A sequency ordered version of the transform can be implemented as follows:

```
void slant_seq(double *f, ulong ldn)
// sequency ordered slant transform
{
    slant(f, ldn);

    ulong n = 1UL<<ldn;
    inverse_gray_permute(f, n);
    unzip_rev(f, n);
    revbin_permute(f, n);
}
```

This implementation could be optimized by fusing the involved permutations, cf. [40].

The inverse is trivially derived by calling the inverse operations in reversed order:

```
void inverse_slant_seq(double *f, ulong ldn)
// inverse of slant_seq()
{
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    zip_rev(f, n);
    gray_permute(f, n);

    inverse_slant(f, ldn);
}
```

TBD: *figure: slant basis funcs*

5.8 The Reed-Muller transform (RMT)

How to make a Reed-Muller transform out of a Walsh transform:
'*Replace* `u+v` *by* `u` *and* `u-v` *by* `u XOR v`*, done.*'

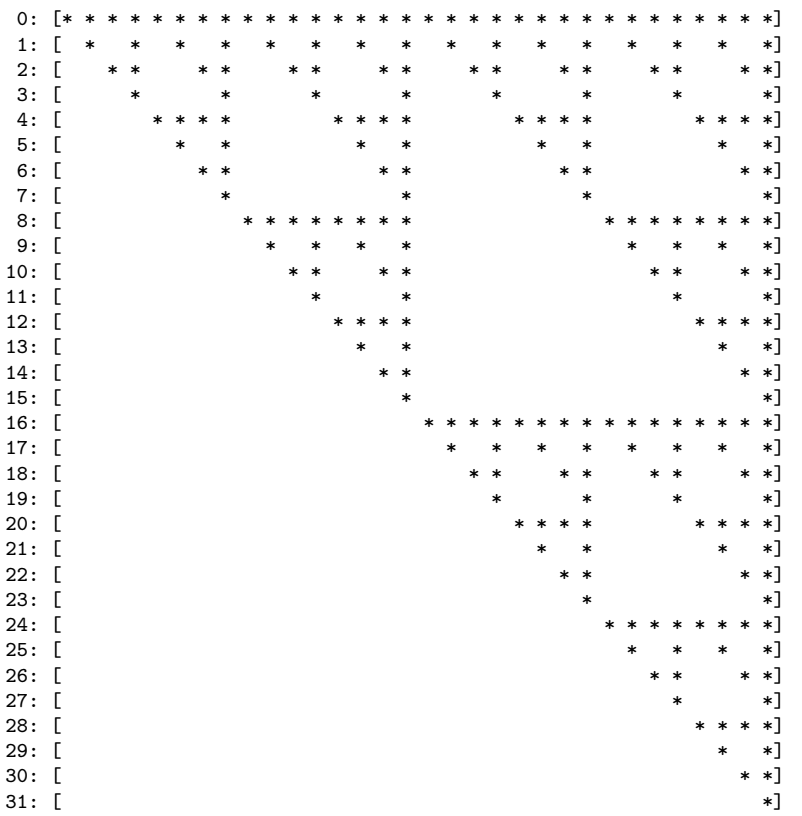


Figure 5.8: Basis functions for the Reed-Muller transform (RMT). Asterisks denote positions where the functions are equal to one. Blank entries correspond to zero.

There we go:

```
template <typename Type>
void word_reed_muller_dif2(Type *f, ulong ldn)
// Reed-Muller Transform
// Type must have the XOR operator.
// decimation in frequency (DIF) algorithm
// self-inverse
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u ^ v;
            }
```

```
        }
    }
}
```

[FXT: `word_reed_muller_dif2` in `walsh/reedmuller.h`].

This routine is almost identical to [FXT: `walsh_wak_dif2` in `walsh/walshwak.h`]. The only changes are

```
    walsh          f[t1] = u + v;
    walsh          f[t2] = u - v;

    reedmuller     f[t1] = u;
    reedmuller     f[t2] = u ^ v;
```

The decimation in time algorithm can be obtained from [FXT: `walsh_wak_dit2` in `walsh/walshwak.h`] by the very same changes.

As given, the transforms work word-wise, if the bit-wise transform is wanted use

```
template <typename Type>
inline void bit_reed_muller(Type *f, ulong ldn)
{
    word_reed_muller_dif2(f, ldn);
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n; ++k)  f[k] = yellow_code(f[k]);
}
```

The `yellow_code` (see section 8.23) can also be applied before the main loop. In fact, the yellow code *is* the Reed-Muller transform (RMT) on a binary word.

The other 'color-transforms' of section 8.23 lead to variants of the RMT, the blue-code gives another self-inverse transform, the red-code and the cyan-code give transforms R and C so that

$$R\,R\,R = \mathrm{id} \qquad R^{-1} = R\,R = C \tag{5.40}$$

$$C\,C\,C = \mathrm{id} \qquad C^{-1} = C\,C = R \tag{5.41}$$

$$R\,C = C\,R = \mathrm{id} \tag{5.42}$$

In fact, all relations given in the referenced section hold.

As can be seen from the 'atomic' matrices (relations 8.40 ... 8.43) the four transforms corresponding to the 'color-codes' are obtained by

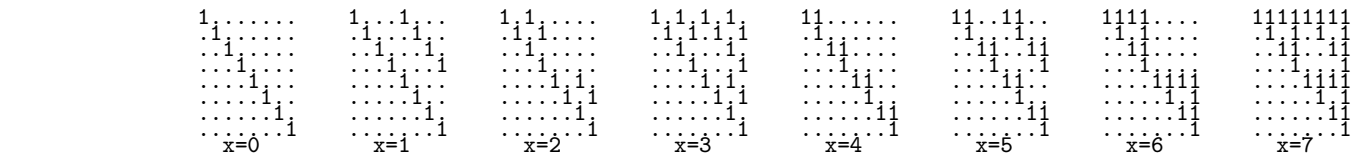
```
 Walsh:   f[t1] = u + v;    f[t2] = u - v;

    B:    f[t1] = u ^ v;    f[t2] = v;
    Y:    f[t1] = u;        f[t2] = u ^ v;  (Reed-Muller transform)
    R:    f[t1] = v;        f[t2] = u ^ v;
    C:    f[t1] = u ^ v;    f[t2] = u;
```

The blue-code equivalent leads to a basis that is obtained by transposing the shown one, the red- and cyan- variants are the mutually inverse

```
      red                    cyan
1111111111111111        ...............1
1.1.1.1.1.1.1.1.        ..............11
11..11..11..11..        .............1.1
1...1...1...1...        ............1111
1111....1111....        ............1...1
1.1.....1.1.....        ..........11..11
11......11......        ..........1.1.1.1
1.......1.......        ........11111111
11111111........        .......1.......1
1.1.1.1.........        ......11......11
11..11..........        .....1.1.....1.1
1...1...........        ....1111....1111
1111............        ...1...1...1...1
1.1.............        ..11..11..11..11
11..............        .1.1.1.1.1.1.1.1
1...............        1111111111111111
```

The symbolic powering idea from section 8.23 leads to transforms with bases (using eight element arrays):



[FXT: file `demo/bitxtransforms-demo.cc`] gives the list for 32-bit words.

[FXT: file `aux1/wordgray.h`] contains some functions that are the Gray code equivalents for $GF(2^n)$:

```
template <typename Type>
void word_gray(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong k=0;  k<n-1;  ++k)  f[k] ^= f[k+1];
}

template <typename Type>
void inverse_word_gray(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong s=1; s<n; s*=2)
    {
        // word_gray ** s:
        for (ulong k=0, j=k+s;  j<n;  ++k,++j)  f[k] ^= f[j];
    }
}
```

As one might suspect, these are related to the Reed-Muller transform. Writing Y ('yellow') for the RMT, g for the word- Gray code and S_k for the cyclic shift by k words (word zero is moved to position k) one has

$$Y\,S_{+1}\,Y = g \tag{5.43}$$
$$Y\,S_{-1}\,Y = g^{-1} \tag{5.44}$$
$$Y\,S_{k}\,Y = g^{k} \tag{5.45}$$

These are exactly the relations 8.32 given on page 196 for the bit-wise transforms.

For $k >= 0$ the operator S_k corresponds to the shift toward element zero (use [FXT: `rotate_sgn` in `perm/rotate.h`]).

The power of the word-wise Gray code is perfectly equivalent to the bit-wise version:

```
template <typename Type>
void word_gray_pow(Type *f, ulong ldn, ulong x)
{
    ulong n = 1UL<<ldn;
    x &= (n-1);  // modulo n
    for (ulong s=1; s<n; s*=2)
    {
        if ( x & 1)
        {
            // word_gray ** s:
            for (ulong k=0, j=k+s;  j<n;  ++k,++j)  f[k] ^= f[j];
        }
        x >>= 1;
    }
}
```

With e be the green code operator, then for the 'blue-variant':

$$B\,S_{+1}\,B = e^{-1} \tag{5.46}$$
$$B\,S_{-1}\,B = e \tag{5.47}$$
$$B\,S_{k}\,B = e^{-k} \tag{5.48}$$

Further,

$$C\,S_k\,R = e^k \tag{5.49}$$
$$C\,e_k\,R = S^k \tag{5.50}$$

The transforms as Kronecker products (all operations are modulo two):

$$B_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad B_n = \bigotimes_{k=1}^{\log_2(n)} B_2 \tag{5.51}$$

$$Y_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad Y_n = \bigotimes_{k=1}^{\log_2(n)} Y_2 \tag{5.52}$$

$$R_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \qquad R_n = \bigotimes_{k=1}^{\log_2(n)} R_2 \tag{5.53}$$

$$C_2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \qquad C_n = \bigotimes_{k=1}^{\log_2(n)} C_2 \tag{5.54}$$

A function that computes the k-th base function of the word-wise transform is

```
template <typename Type>
inline void word_reed_muller_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
        ulong x = (n-1-i) & (k);
        f[i] = ( 0==x ? +1 : 0 );
    }
}
```

[FXT: `word_reed_muller_basefunc` in `walsh/reedmuller.h`]

5.9 The arithmetic transform

How to make a arithmetic transform out of a Walsh transform:
'*Forward: replace* `u+v` *by* `u` *and* `u-v` *by* `v-u`. *Backward: replace* `u+v` *by* `u` *and* `u-v` *by* `u+v`.'
On to the code:

```
template <typename Type>
void arith_transform_plus(Type *f, ulong ldn)
// Arithmetic Transform
// Decimation In Frequency (DIF) algorithm
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u + v;
```

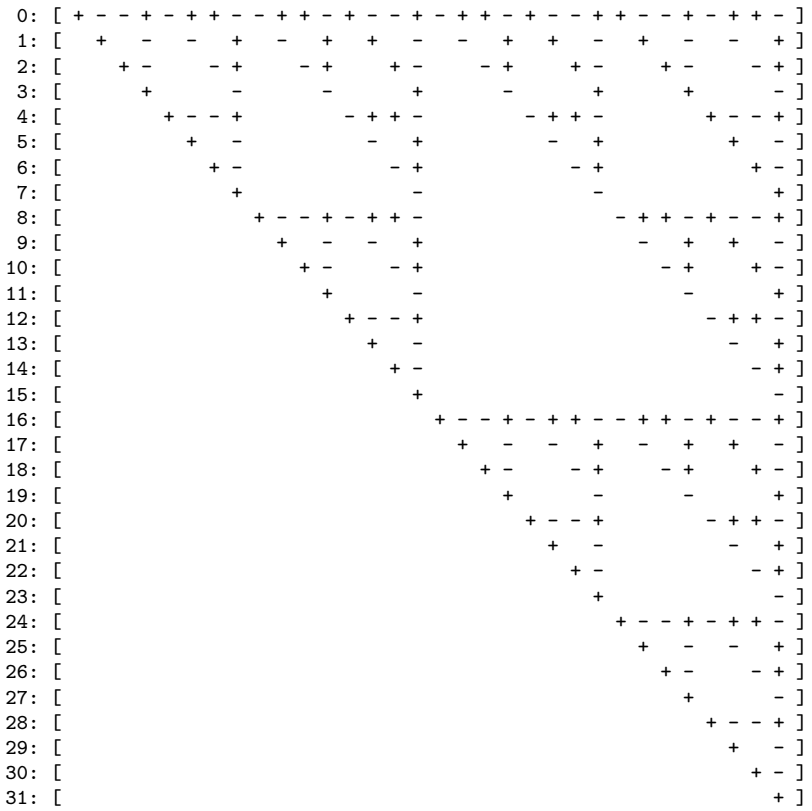


Figure 5.9: Basis functions for the Arithmetic transform (A^-, the one with the minus sign). The values are ± 1, blank entries denote 0.

```
            }
        }
    }
}
```

and

```
template <typename Type>
void arith_transform_minus(Type *f, ulong ldn)
// Inverse of arith_transform_plus
{
  -- snip --
                f[t1] = u;
                f[t2] = v - u;
  -- snip --
}
```

The length-2 transforms can be written as

$$A_2^+ v = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{5.55}$$

$$A_2^- v = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b-a \end{bmatrix} \tag{5.56}$$

That the one with the minus is called the forward transform is tradition. Similar to the Fourier transform we avoid the forward- backward- naming scheme and put:

```
template <typename Type>
```

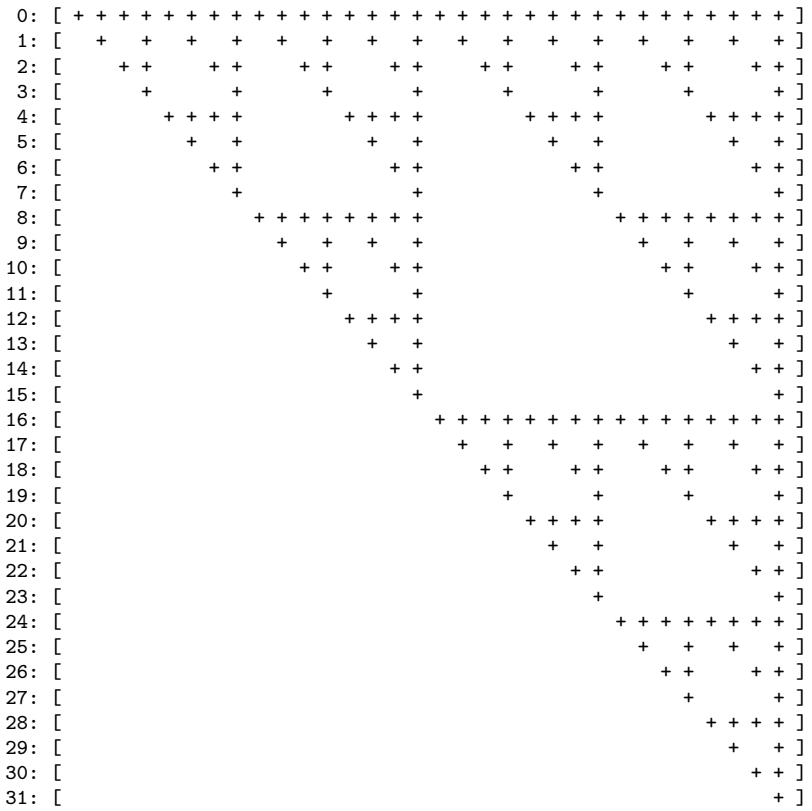


Figure 5.10: Basis functions for the inverse Arithmetic transform (A^+, the one without minus sign). The values are ± 1, blank entries denote 0.

```
inline void arith_transform(Type *f, ulong ldn, int is)
{
    if ( is>0 )  arith_transform_plus(f, ldn);
    else  arith_transform_minus(f, ldn);
}
```

In Kronecker product notation the arithmetic transform and its inverse can be written as

$$A_2^+ = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \qquad A_n^+ = \bigotimes_{k=1}^{\log_2(n)} A_2^+ \tag{5.57}$$

$$A_2^- = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \qquad A_n^- = \bigotimes_{k=1}^{\log_2(n)} A_2^- \tag{5.58}$$

Chapter 6

The Haar transform

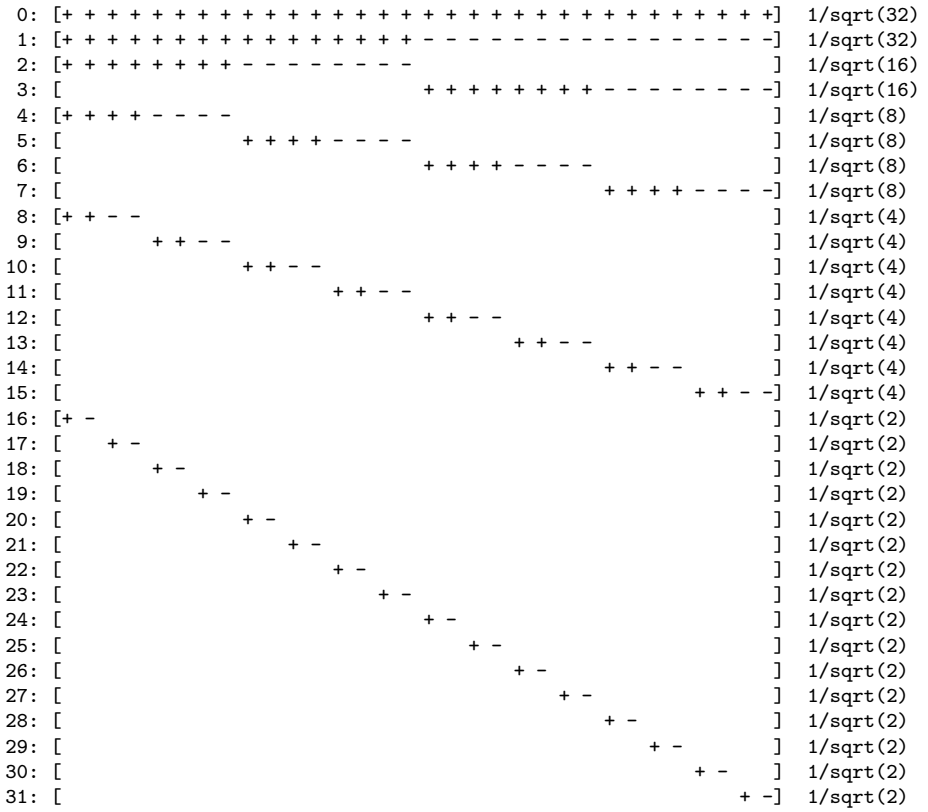


Figure 6.1: Basis functions for the Haar transform. Only the sign of the basis functions is shown. The absolute value of the non-zero entries in each row is given at the right. The norm of each row is one. At the blank entries the functions are zero.

Code for the Haar transform:

```
template <typename Type>
void haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    Type *g = ws;
```

```
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        v *= s2;
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] = (x - y) * v;
        }
        copy(g, f, m);
    }
    f[0] *= v; // v == 1.0/sqrt(n);
    if ( !ws )  delete [] g;
}
```

The above routine uses a temporary workspace that can be supplied by the caller. The computational cost is only $\sim n$. [FXT: `haar` in `haar/haar.h`]

Code for the inverse Haar transform:

```
template <typename Type>
void inverse_haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}
```

[FXT: `inverse_haar` in `haar/haar.h`]

That the given routines use a temporary storage may be seen as a disadvantage. A rather simple reordering of the basis functions, however, allows for to an in-place algorithm. This leads to the in-place Haar transform.

6.1 In-place Haar transform

Code for the in-place version of the Haar transform:

```
template <typename Type>
void haar_inplace(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    for (ulong js=2; js<=n; js<<=1)
    {
        v *= s2;
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
```

```
 0: [+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +]  1/sqrt(32)
 1: [+ -                                                           ]  1/sqrt(4)
 2: [+ + - -                                                       ]  1/sqrt(4)
 3: [    + -                                                       ]  1/sqrt(4)
 4: [+ + + + - - - -                                               ]  1/sqrt(8)
 5: [        + -                                                   ]  1/sqrt(4)
 6: [        + + - -                                               ]  1/sqrt(4)
 7: [            + -                                               ]  1/sqrt(4)
 8: [+ + + + + + + + - - - - - - - -                               ]  1/sqrt(16)
 9: [                + -                                           ]  1/sqrt(4)
10: [                + + - -                                       ]  1/sqrt(4)
11: [                    + -                                       ]  1/sqrt(4)
12: [                + + + + - - - -                               ]  1/sqrt(8)
13: [                        + -                                   ]  1/sqrt(4)
14: [                        + + - -                               ]  1/sqrt(4)
15: [                            + -                               ]  1/sqrt(4)
16: [+ + + + + + + + + + + + + + + + - - - - - - - - - - - - - - - -]  1/sqrt(32)
17: [                                + -                           ]  1/sqrt(4)
18: [                                + + - -                       ]  1/sqrt(4)
19: [                                    + -                       ]  1/sqrt(4)
20: [                                + + + + - - - -               ]  1/sqrt(8)
21: [                                        + -                   ]  1/sqrt(4)
22: [                                        + + - -               ]  1/sqrt(4)
23: [                                            + -               ]  1/sqrt(4)
24: [                                + + + + + + + + - - - - - - - -]  1/sqrt(16)
25: [                                                + -           ]  1/sqrt(4)
26: [                                                + + - -       ]  1/sqrt(4)
27: [                                                    + -       ]  1/sqrt(4)
28: [                                                + + + + - - - -]  1/sqrt(8)
29: [                                                        + -   ]  1/sqrt(4)
30: [                                                        + + - -]  1/sqrt(4)
31: [                                                            + -]  1/sqrt(4)
```

Figure 6.2: Haar basis functions, in-place order. Only the sign of the basis functions is shown. The absolute value of the non-zero entries in each row is given at the right. The norm of each row is one. At the blank entries the functions are zero.

```
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  = (x - y) * v;
        }
    }
    f[0] *= v; // v==1.0/sqrt(n);
}
```

[FXT: `haar_inplace` in `haar/haar.h`]

... and its inverse:

```
template <typename Type>
void inverse_haar_inplace(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
```

```
 0: [+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +]  1/sqrt(32)
 1: [+ + + + + + + + + + + + + + + + - - - - - - - - - - - - - - - -]  1/sqrt(32)
 2: [+ + + + + + + + - - - - - - - -                               ]  1/sqrt(16)
 3: [                           + + + + + + + + - - - - - - - -]  1/sqrt(16)
 4: [+ + + + - - - -                                             ]  1/sqrt(8)
 5: [                           + + + + - - - -                   ]  1/sqrt(8)
 6: [               + + + + - - - -                               ]  1/sqrt(8)
 7: [                                           + + + + - - - -]  1/sqrt(8)
 8: [+ + - -                                                     ]  1/sqrt(4)
 9: [                           + + - -                           ]  1/sqrt(4)
10: [               + + - -                                       ]  1/sqrt(4)
11: [                                           + + - -           ]  1/sqrt(4)
12: [       + + - -                                               ]  1/sqrt(4)
13: [                                     + + - -                 ]  1/sqrt(4)
14: [                       + + - -                               ]  1/sqrt(4)
15: [                                                 + + - -]  1/sqrt(4)
16: [+ -                                                        ]  1/sqrt(2)
17: [                           + -                             ]  1/sqrt(2)
18: [               + -                                         ]  1/sqrt(2)
19: [                                           + -             ]  1/sqrt(2)
20: [       + -                                                 ]  1/sqrt(2)
21: [                                     + -                   ]  1/sqrt(2)
22: [                       + -                                 ]  1/sqrt(2)
23: [                                                 + -       ]  1/sqrt(2)
24: [    + -                                                    ]  1/sqrt(2)
25: [                                 + -                       ]  1/sqrt(2)
26: [                   + -                                     ]  1/sqrt(2)
27: [                                               + -         ]  1/sqrt(2)
28: [           + -                                             ]  1/sqrt(2)
29: [                                       + -                 ]  1/sqrt(2)
30: [                           + -                             ]  1/sqrt(2)
31: [                                                   + -]  1/sqrt(2)
```

Figure 6.3: Basis functions of the in-place order Haar transform followed by a revbin permutation. Note that the ordering is such that basis functions that are identical up to a shift appear consecutively.

```
        v *= s2;
    }
}
```

[FXT: `inverse_haar_inplace` in `haar/haar.h`]

The in-place Haar transform H_i is related to the 'usual' Haar transform H by a permutation P_H via the relations

$$H = P_H \cdot H_i \tag{6.1}$$

$$H^{-1} = H_i^{-1} \cdot P_H^{-1} \tag{6.2}$$

P_H can be programmed as

```
template <typename Type>
void haar_permute(Type *f, ulong ldn)
{
    revbin_permute(f, 1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn-1; ++ldm)
    {
        ulong m = (1UL<<ldm);  // m=2, 4, 8, ..., n/2
        revbin_permute(f+m, m);
    }
}
```

while its inverse is

```
template <typename Type>
```

```
void inverse_haar_permute(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn-1; ++ldm)
    {
        ulong m = (1UL<<ldm);  // m=2, 4, 8, ..., n/2
        revbin_permute(f+m, m);
    }
    revbin_permute(f, 1UL<<ldn);
}
```

(cf. [FXT: file perm/haarpermute.h])

Then, as given above, `haar` is equivalent to

```
    inplace_haar();
    haar_permute();
```

and `inverse_haar` is equivalent to

```
    inverse_haar_permute();
    inverse_inplace_haar();
```

6.2 Non-normalized Haar transforms

Versions of the Haar transform without normalization are given in [FXT: file haar/haarnn.h]. The basis functions are the same as for the normalized versions, only the absolute value of the non-zero entries are different.

```
template <typename Type>
void haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

[FXT: haar_nn in haar/haarnn.h]

```
template <typename Type>
void inverse_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = 2.0;
    Type v = 1.0/n;
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
```

```
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}

template <typename Type>
void haar_inplace_nn(Type *f, ulong ldn)
//
// transform wrt. to non-normalized haar base
// in-place operation
//
// the sequence
//   haar_inplace_nn();  haar_permute();
// is equivalent to
//   haar_nn()
//
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

[FXT: haar_inplace_nn in haar/haarnn.h]

```
template <typename Type>
void inverse_haar_inplace_nn(Type *f, ulong ldn)
//
// inverse transform wrt. to haar base
// in-place operation
//
// the sequence
//   inverse_haar_permute();
//   inverse_haar_inplace();
// is equivalent to
//   inverse_haar()
//
{
    ulong n = 1UL<<ldn;
    Type s2 = 2.0;
    Type v = 1.0/n;
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
        v *= s2;
    }
}
```

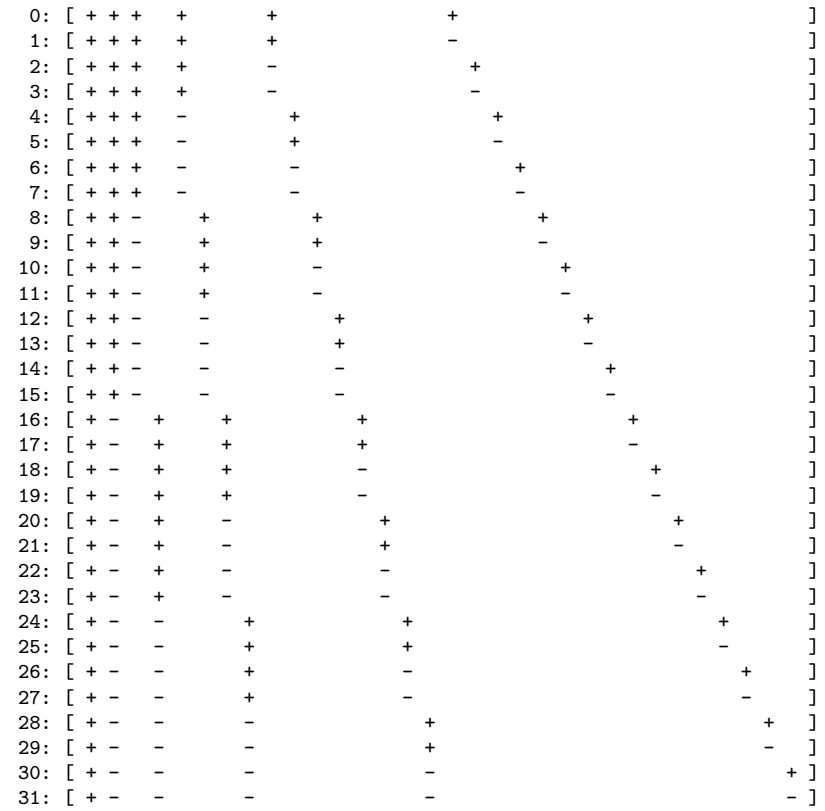


Figure 6.4: Basis functions for the transposed Haar transform. Only the sign of the basis functions is shown. At the blank entries the functions are zero.

6.3 Transposed Haar transforms

```
template <typename Type>
void transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k];
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

[FXT: `transposed_haar_nn` in `haar/transposedhaarnn.h`]

```
template <typename Type>
void inverse_transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
```

```
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j]   * 0.5;
            Type y = f[j+1] * 0.5;
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

```
 0: [ + + +   +       +              +                           ]
 1: [ + - +   +       +              +                           ]
 2: [ +   - + +       +              +                           ]
 3: [ +   - - +       +              +                           ]
 4: [ +       - + +   +              +                           ]
 5: [ +       - - +   +              +                           ]
 6: [ +       -   - + +              +                           ]
 7: [ +       -   - - +              +                           ]
 8: [ +               - + +   +      +                           ]
 9: [ +               - - +   +      +                           ]
10: [ +               -   - + +      +                           ]
11: [ +               -   - - +      +                           ]
12: [ +               -       - + +  +                           ]
13: [ +               -       - - +  +                           ]
14: [ +               -       -   - + +                          ]
15: [ +               -       -   - - +                          ]
16: [ +                               - + +   +       +          ]
17: [ +                               - - +   +       +          ]
18: [ +                               -   - + +       +          ]
19: [ +                               -   - - +       +          ]
20: [ +                               -       - + +   +          ]
21: [ +                               -       - - +   +          ]
22: [ +                               -       -   - + +          ]
23: [ +                               -       -   - - +          ]
24: [ +                               -               - + +   +  ]
25: [ +                               -               - - +   +  ]
26: [ +                               -               -   - + +  ]
27: [ +                               -               -   - - +  ]
28: [ +                               -               -       - + +]
29: [ +                               -               -       - - +]
30: [ +                               -               -       -   - +]
31: [ +                               -               -       -   - -]
```

Figure 6.5: Basis functions for the transposed in-place Haar transform. Only the sign of the basis functions is shown. At the blank entries the functions are zero.

```
template <typename Type>
void transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

[FXT: transposed_haar_inplace_nn in haar/transposedhaarnn.h]

```
template <typename Type>
void inverse_transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j] * 0.5;
            Type y = f[t] * 0.5;
            f[j]  =  x + y;
            f[t]  =  x - y;;
        }
    }
}
```

6.4 The reversed Haar transform

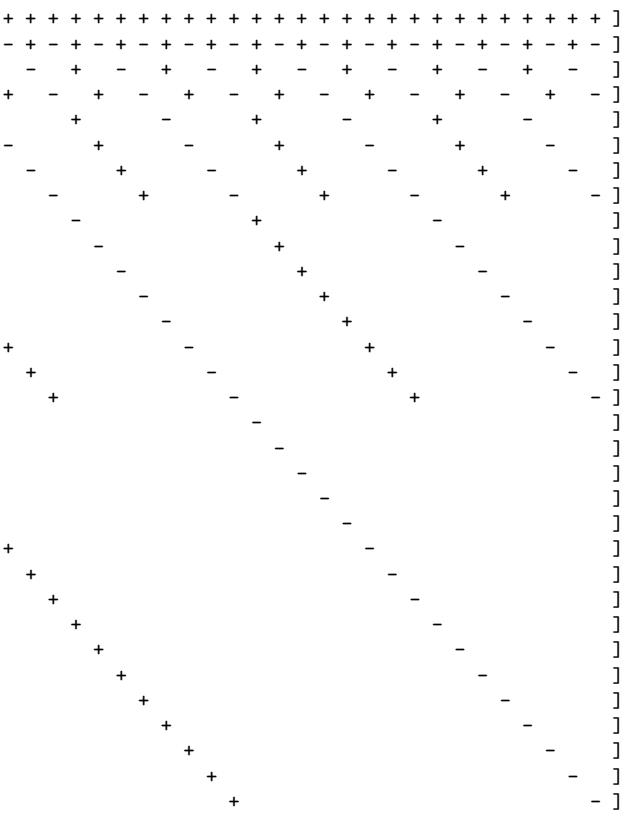


Figure 6.6: Basis functions for the reversed Haar transform. Only the sign of the non-zero entries is shown.

Let H_{ni} denote the non-normalized in-place Haar transform (`haar_inplace_nn`), Let H_{tni} denote the transposed non-normalized in-place Haar transform (`transposed_haar_inplace_nn`), R the revbin-

permutation, $\bar{H}$ the reversed Haar transform and $\bar{H}_t$ the transposed reversed Haar transform. Then

$$\bar{H} = R\,H_{ni}\,R \tag{6.3}$$
$$\bar{H}_t = R\,H_{tni}\,R \tag{6.4}$$
$$\bar{H}^{-1} = R\,H_{ni}^{-1}\,R \tag{6.5}$$
$$\bar{H}_t^{-1} = R\,H_{tni}^{-1}\,R \tag{6.6}$$

Code for the reversed Haar transform:

```
template <typename Type>
void haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `haar_rev_nn` in `haar/haarrevnn.h`] Note that this is almost the DIF implementation for the Walsh transform (W_k implemented as `walsh_wak_dif2`): The only thing that changed is that the line `for (ulong r=0; r<n; r+=m)` was replaced by `ulong r = 0`.

The transposed transform is:

```
template <typename Type>
void transposed_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

[FXT: `transposed_haar_rev_nn` in `haar/transposedhaarrevnn.h`] With the identical change as above this is almost the DIT implementation for the Walsh transform (W_k implemented as `walsh_wak_dit2`).

The inverse transforms are

```
template <typename Type>
```

```
void inverse_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

and

```
template <typename Type>
void inverse_transposed_haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

6.5 Relations between Walsh- and Haar- transforms

6.5.1 Walsh transforms from Haar transforms

A length-n Walsh transform can be obtained from one length-n Haar transform, one transform of length-$\frac{n}{2}$, two transforms of length-$\frac{n}{4}$, four transforms of length-$\frac{n}{8}$, ... and $\frac{n}{4}$ transforms of length-2. Using the reversed Haar transform the implementation is most straightforward: A Walsh transform (W_k, the one with the Walsh Kronecker base) can be implemented as

```
    // algorithm WH1:
    ulong n = 1UL<<ldn;
    haar_rev_nn(f, ldn);
    for (ulong ldk=ldn-1; ldk>0; --ldk)
    {
        ulong k = 1UL << ldk;
        for (ulong j=k; j<n; j+=2*k)  haar_rev_nn(f+j, ldk);
    }
```

The idea can be drawn symbolically as in figure 6.5.1.

Equivalently, one can compute W_k using the transposed version:

```
Haar transforms:
    H(16)                  H(8)          H(4)      H(2)
  AAAAAAAAaaaaaaaa        BBBBbbbb       CCcc      Dd
  AAAAaaaa                BBbb           Cc
  AAaa                    Bb
  Aa

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AAAAAAAAaaaaaaaa
  AAAAaaaaBBBBbbbb
  AAaaCCccBBbbCCcc
  AaDdCcDdBbDdCcDd
```

Figure 6.7: Symbolic description of how to build a Walsh transform from Haar transforms.

```
Transposed Haar transforms:
    H(16)                  H(8)          H(4)      H(2)
  Aa
  AAaa                    Bb
  AAAAaaaa                BBbb           Cc
  AAAAAAAAaaaaaaaa        BBBBbbbb       CCcc      Dd

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AaDdCcDdBbDdCcDd
  AAaaCCccBBbbCCcc
  AAAAaaaaBBBBbbbb
  AAAAAAAAaaaaaaaa
```

Figure 6.8: Symbolic description of how to build a Walsh transform from Haar transforms, transposed version.

```
// algorithm WH1T:
ulong n = 1UL<<ldn;
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  transposed_haar_rev_nn(f+j, ldk);
}
transposed_haar_rev_nn(f, ldn);
```

The symbolic scheme is obtained by reversing the lines in the non-transposed scheme, this is shown in figure 6.5.1.

Moreover, the inverse Walsh transform ($W_k^{-1} = \frac{1}{n} W_k$) can be computed as

```
// algorithm WH2T:
ulong n = 1UL<<ldn;
inverse_transposed_haar_rev_nn(f, ldn);
for (ulong ldk=ldn-1; ldk>0; --ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  inverse_transposed_haar_rev_nn(f+j, ldk);
}
```

or as

```
// algorithm WH2:
ulong n = 1UL<<ldn;
for (ulong ldk=1; ldk<ldn; ++ldk)
```

```
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  inverse_haar_rev_nn(f+j, ldk);
}
inverse_haar_rev_nn(f, ldn);
```

6.5.2 Haar transforms from Walsh transforms

```
Walsh transform:
        W(16)
      AaDdCcDdBbDdCcDd
      AAaaCCccBBbbCCcc
      AAAAaaaaBBBBbbbb
      AAAAAAAAaaaaaaaa

Inverse (or transposed) Walsh transforms:
       W(8):           W(4):      W(2):
      BBBBbbbb           CCcc        Dd
      BBbbCCcc           CcDd
      BbDdCcDd


                                      BBBBbbbb
                                    CCccBBbbCCcc
                                  DdCcDdBbDdCcDd
      Aa                        AaDdCcDdBbDdCcDd
      AAaa                      AAaaCCccBBbbCCcc
      AAAAaaaa                  AAAAaaaaBBBBbbbb
      AAAAAAAAaaaaaaaa          AAAAAAAAaaaaaaaa
        Haar(16)      =^=  W(16) + W(8) + W(4) + W(2)
```

Figure 6.9: Symbolic description of how to build a Haar transform from Walsh transforms.

The non-normalized transposed reversed Haar transform can (up to normalization) be obtained via

```
// algorithm HW1:  transposed_haar_rev_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
walsh_wak(f, ldn);
```

and its inverse as

```
// algorithm HW1I:  inverse_transposed_haar_rev_nn(f, ldn); =^=
walsh_wak(f, ldn);
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_wak(f+k, ldk);
}
```

The non-normalized transposed Haar transform can (again, up to normalization) be obtained via

```
// algorithm HW2:  transposed_haar_nn(f, ldn); =^=
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
```

```
        walsh_pal(f+k, ldk);
    }
    walsh_pal(f, ldn);
```

and its inverse as

```
    // algorithm HW2I:  inverse_transposed_haar_nn(f, ldn); =^=
    walsh_pal(f, ldn);  // =^= revbin_permute(f, n); walsh_wak(f, ldn);
    for (ulong ldk=1; ldk<ldn; ++ldk)
    {
        ulong k = 1UL << ldk;
        walsh_pal(f+k, ldk);
    }
```

The symbolic scheme is given in figure 6.5.2.

6.6 Integer to integer Haar transform

Code 6.1 (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
    }
}
```

Omit floor() with integer types. [FXT: `haar_i2i` in `haar/haari2i.cc`]

Code 6.2 (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
    }
}
```

[FXT: `inverse_haar_i2i` in `haar/haari2i.cc`]

Chapter 7

Permutations

7.1 The revbin permutation

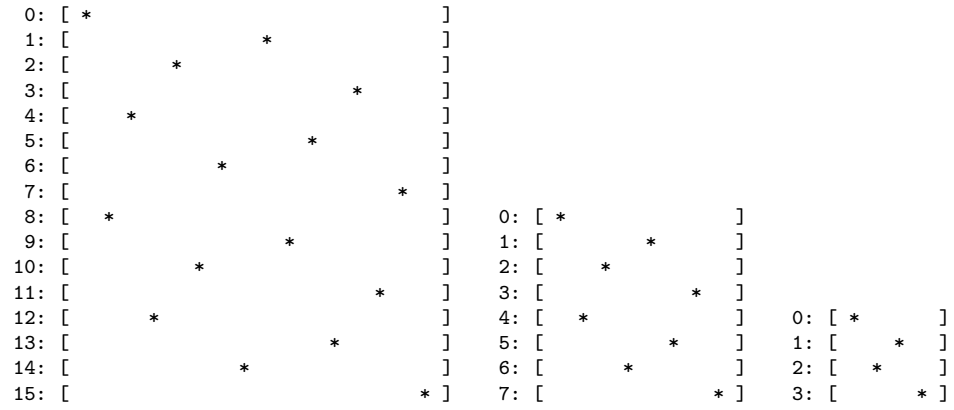


Figure 7.1: Permutation matrices of the revbin-permutation for sizes 16, 8 and 4. The permutation is self-inverse.

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 on both x and on n.

7.1.1 A naive version

A first implementation might look like

```
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 condition `r>x` before the `swap()` statement makes sure that the swapping isn't undone later when the loop variable `x` has the value of the present `r`. The function `revbin(x, n)` shall return the reversed

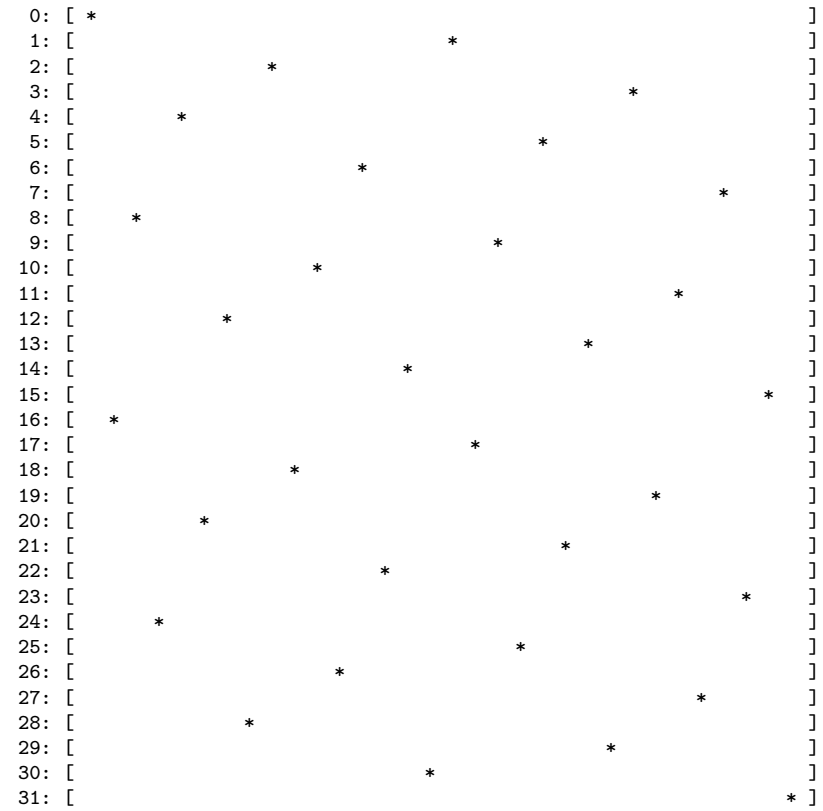


Figure 7.2: Permutation matrix of the revbin-permutation for size 32.

bits of `x`:

```
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
}
```

This version of the `revbin_permute`-routine is pretty inefficient (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 by solving a slightly different problem.

7.1.2 A fast version

The key idea is to *update* 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.
A routine to 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 XOR
    } 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;
}
```

[FXT: `revbin_update` in `auxbit/revbin.h`]

Now we are ready for a fast revbin-permute routine:

```
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()` needs for half of the calls just one iteration because in half of the updates just the leftmost bit changes[1], in half of the remaining updates it needs two iterations, in half of the still remaining updates it needs three and so on. The total number of operations done by `revbin_update()` is therefore proportional to $n\,(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\cdots+\frac{\log_2(n)}{n})$ $= n\sum_{j=1}^{\log_2(n)}\frac{j}{2^j}$. For n large this sum is close to $2n$. Thereby the asymptotics of `revbin_permute()` is improved from proportional $n\,\log(n)$ to proportional n.

7.1.3 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.

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()`.

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

7.1.4 A still faster version

The following table lists indices versus their revbin-counterpart. The subscript 2 indicates printing in base 2, $\Delta := \widetilde{x} - \widetilde{x-1}$ and an 'y' in the last column marks index pairs where `revbin_permute()` will swap elements.

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	

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 a routine to put data into 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 optimized version for zero padded data:

```
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])
        // both a[n-1-x] and a[n-1-r] are zero
        x := x + 1
    }
}
```

One could carry the scheme that lead to the 'faster' revbin_permute procedures further, e.g. using 3 hard-coded 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 separate arrays for real and imaginary part[2] 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 after each statement `swap(a[x],a[r])` has 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 favorable case the cache miss rate will be increased. Do expect to hardly ever win anything noticeable but in most cases to lose big. Think about it, whisper "*direct mapped cache*" and forget it.

7.1.5 The real world version

Finally we remark that the `revbin_update` can be optimized by usage of a small (length `BITS_PER_LONG`) table containing the reflected bursts of ones that change on the lower end with incrementing. A routine that utilizes this idea, optionally uses the CPU-bitscan instruction(cf. section 8.2) and further allows to select the amount of symmetry optimizations looks like

```
#include "inline.h"  // swap()
#include "fxttypes.h"
#include "bitsperlong.h" //  BITS_PER_LONG
#include "revbin.h" // revbin(), revbin_update()

#include "bitasm.h"
#if  defined  BITS_USE_ASM
#include "bitlow.h" // lowest_bit_idx()
#define  RBP_USE_ASM  // use bitscan if available, comment out to disable
```

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

```
#endif // defined  BITS_USE_ASM

#define  RBP_SYMM  4  // 1, 2, 4 (default is 4)
#define  idx_swap(f, k, r)  { ulong kx=(k), rx=(r);  swap(f[kx], f[rx]); }
template <typename Type>
void revbin_permute(Type *f, ulong n)
{
    if ( n<=8 )
    {
        if ( n==8 )
        {
            swap(f[1], f[4]);
            swap(f[3], f[6]);
        }
        else if ( n==4 )  swap(f[1], f[2]);

        return;
    }

    const ulong nh = (n>>1);
    ulong x[BITS_PER_LONG];
    x[0] = nh;
    {  // initialize xor-table:
        ulong i, m = nh;
        for (i=1; m!=0; ++i)
        {
            m >>= 1;
            x[i] = x[i-1] ^ m;
        }
    }

#if  ( RBP_SYMM >= 2 )
    const ulong n1  = n - 1;    // = 11111111
#if  ( RBP_SYMM >= 4 )
    const ulong nx1 = nh - 2;   // = 01111110
    const ulong nx2 = n1 - nx1; // = 10111101
#endif //  ( RBP_SYMM >= 4 )
#endif //  ( RBP_SYMM >= 2 )
    ulong k=0, r=0;
    while ( k<n/RBP_SYMM  )  // n>=16, n/2>=8, n/4>=4
    {
        // ----- k%4 == 0:
        if ( r>k )
        {
            swap(f[k], f[r]);  // <nh, <nh 11
#if  ( RBP_SYMM >= 2 )
            idx_swap(f, n1^k, n1^r);  // >nh, >nh 00
#if  ( RBP_SYMM >= 4 )
            idx_swap(f, nx1^k, nx1^r);  // <nh, <nh 11
            idx_swap(f, nx2^k, nx2^r);  // >nh, >nh 00
#endif //  ( RBP_SYMM >= 4 )
#endif //  ( RBP_SYMM >= 2 )
        }

        r ^= nh;
        ++k;

        // ----- k%4 == 1:
        if ( r>k )
        {
            swap(f[k], f[r]);  // <nh, >nh 10
#if  ( RBP_SYMM >= 4 )
            idx_swap(f, n1^k, n1^r);  // >nh, <nh 01
#endif //  ( RBP_SYMM >= 4 )
        }

        { // scan for lowest unset bit of k:
#ifdef  RBP_USE_ASM
            ulong i = lowest_bit_idx(~k);
#else
            ulong m = 2,  i = 1;
            while ( m & k )  { m <<= 1;  ++i; }
#endif // RBP_USE_ASM
            r ^= x[i];
        }
        ++k;

        // ----- k%4 == 2:
        if ( r>k )
        {
```

```
            swap(f[k], f[r]);  // <nh, <nh 11
#if  ( RBP_SYMM >= 2 )
            idx_swap(f, n1^k, n1^r); // >nh, >nh 00
#endif //  ( RBP_SYMM >= 2 )
        }

        r ^= nh;
        ++k;

        // ----- k%4 == 3:
        if ( r>k )
        {
            swap(f[k], f[r]);    // <nh, >nh 10
#if  ( RBP_SYMM >= 4 )
            idx_swap(f, nx1^k, nx1^r);   // <nh, >nh 10
#endif //  ( RBP_SYMM >= 4 )
        }

        { // scan for lowest unset bit of k:
#ifdef  RBP_USE_ASM
            ulong i = lowest_bit_idx(~k);
#else
            ulong m = 4,  i = 2;
            while ( m & k )  { m <<= 1;  ++i; }
#endif // RBP_USE_ASM
            r ^= x[i];
        }
        ++k;
    }
}
```

. . . not the most readable piece of code but a nice example for a real-world optimized routine.

This is [FXT: `revbin_permute` in `perm/revbinpermute.h`], see [FXT: `revbin_permute0` in `perm/revbinpermute0.h`] for the respective version for zero padded data.

7.2 The radix permutation

The radix-permutation is the generalization of the revbin-permutation (corresponding to radix 2) to arbitrary radices.

C++ code for the radix-`r` 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 have:
//  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 `perm/radixpermute.h`]

TBD: *mixed-radix permute*

7.3 In-place matrix transposition

To transpose a $n_r \times n_c$- matrix first identify the position i of then entry in row r and column c:

$$i = r \cdot n_c + c \tag{7.1}$$

After the transposition the element will be at position i' in the transposed $n'_r \times n'_c$- matrix

$$i' = r' \cdot n'_c + c' \tag{7.2}$$

Obviously, $r' = c$, $c' = r$, $n'_r = n_c$ and $n'_c = n_r$, so:

$$i' = c \cdot n_r + r \tag{7.3}$$

Multiply the last equation by n_c

$$i' \cdot n_c = c \cdot n_r \cdot n_c + r \cdot n_c \tag{7.4}$$

With $n := n_r \cdot n_c$ and $r \cdot n_c = i - c$ we get

$$i' \cdot n_c = c \cdot n + i - c \tag{7.5}$$

$$i = i' \cdot n_c + c \cdot (n-1) \tag{7.6}$$

Take the equation modulo $n-1$ to get[3]

$$i \equiv i' \cdot n_c \mod (n-1) \tag{7.7}$$

That is, the transposition moves the element $i = i' \cdot n_c$ to position i'. Multiply by n_r to get the inverse:

$$i \cdot n_r \equiv i' \cdot n_c \cdot n_r \tag{7.8}$$

$$i \cdot n_r \equiv i' \cdot (n-1+1) \tag{7.9}$$

$$i \cdot n_r \equiv i' \tag{7.10}$$

That is, element i will be moved to $i' = i \cdot n_r \mod (n-1)$.

[FXT: `transpose` in `aux2/transpose.h`]
[FXT: `transpose_ba` in `aux2/transpose_ba.h`]

Note that one should take care of possible overflows in the calculation $i \cdot n_c$.

For the case that n is a power of two (and so are both n_r and n_c) the multiplications modulo $n-1$ are cyclic shifts. Thus any overflow can be avoided and the computation is also significantly cheaper.

[FXT: `transpose2_ba` in `aux2/transpose2_ba.h`]

[3] As the last element of the matrix is a fixed point the transposition moves around only the $n-1$ elements $0 \ldots n-2$

7.4 Revbin permutation vs. transposition

How would you rotate an (length-n) array by s positions (left or right), *without* using any scratch space. If you do not know the solution then try to find it before reading on.

Rotation by triple reversion

The nice little trick is to use `reverse` three times as in the following:

```
template <typename Type>
void rotate_left(Type *f, ulong n, ulong s)
// rotate towards element #0
// shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }

    reverse(f,   s);
    reverse(f+s, n-s);
    reverse(f,   n);
}
```

Likewise for the other direction:

```
template <typename Type>
void rotate_right(Type *f, ulong n, ulong s)
// rotate away from element #0
// shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }

    reverse(f,     n-s);
    reverse(f+n-s, s);
    reverse(f,     n);
}
```

[FXT: `rotate_left` and `rotate_right` in `perm/rotate.h`]

What this has to do with our subject? When transposing an $n_r \times n_c$ matrix whose size is a power of two (thereby both n_r and n_c are also powers of two) the above mentioned rotation is done with the *indices* (written in base two) of the elements. We know how to do a permutation that reverses the complete indices and reversing a few bits at the least significant end is not any harder:

```
template <typename Type>
void revbin_permute_rows(Type *f, ulong ldn, ulong ldnc)
// revbin_permute the length 2**ldnc rows  of f[0..2**ldn-1]
// (f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix)
{
    ulong n = 1UL<<ldn;
    ulong nc = 1UL<<ldnc;
    for (ulong k=0; k<n; k+=nc)  revbin_permute(f+k, nc);
}
```

And there we go:

```
template <typename Type>
void transpose_by_rbp(Type *f, ulong ldn, ulong ldnc)
// transpose f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix
{
    revbin_permute_rows(f, ldn, ldnc);
```

```
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    revbin_permute_rows(f, ldn, ldn-ldnc);  // ... that is, columns
}
```

TBD: *revbin-permute by transposition*

TBD: *2dim generalization: triple shearing*

7.5 The zip permutation

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

Figure 7.3: Permutation matrices of the zip-permutation (left) and its inverse, the unzip-permutation (right). The zip-permutation moves the lower half of the array to the even indices, the upper half to the odd indices.

An important special case of the 'transposition by revbin permutation' is

```
template <typename Type>
void zip(Type *f, ulong n)
//
// lower half --> even indices
// higher half --> odd indices
//
// same as transposing the array as 2 x n/2 - matrix
//
// useful to combine real/imag part into a Complex array
//
// n must be a power of two
{
    ulong nh = n/2;
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
    revbin_permute(f, n);
}
```

[FXT: `zip` in `perm/zip.h`]

The effect of the code is the same as that of

```
template <typename Type>
void zip(const Type * restrict f, Type * restrict g, ulong n)
// n must be even
{
    ulong nh = n/2;
    for (ulong k=0,  k2=0;  k<nh;  ++k, k2+=2)  g[k2] = f[k];
    for (ulong k=nh, k2=1;  k<n;   ++k, k2+=2)  g[k2] = f[k];
}
```

Here the array length does not need to be a power of two.
For the type `double` the 'zip by revbin-permute' technique can be optimized slightly.

```
void zip(double *f, long n)
{
    revbin_permute(f, n);
    revbin_permute((Complex *)f, n/2);
}
```

Here it is assumed that the type `Complex` consists of two `double`s lying contiguous in memory.
The inverse of `zip` is `unzip`:

```
template <typename Type>
void unzip(Type *f, ulong n)
//
// inverse of zip():
// put part of data with even indices
// sorted into the lower half,
// odd part into the higher half
//
// same as transposing the array as n/2 x 2 - matrix
//
// useful to separate a Complex array into real/imag part
//
// n must be a power of two
{
    ulong nh = n/2;
    revbin_permute(f, n);
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
}
```

[FXT: `unzip` in `perm/zip.h`] which can for the type `double` again be optimized as

```
void unzip(double *f, long n)
{
    revbin_permute((Complex *)f, n/2);
    revbin_permute(f, n);
}
```

[FXT: `unzip` in `perm/zip.cc`] TBD: *zip for length not a power of two*

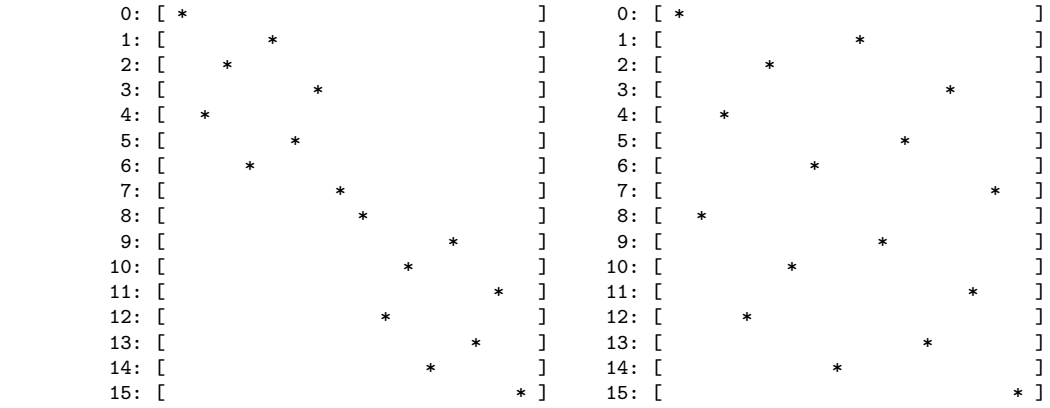


Figure 7.4: Revbin-permutation matrices that, when multiplied together, give the zip-permutation and its inverse. Let L,R be the permutations given on the left,right side, respectively. Then $Z = R\,L$ and $Z^{-1} = L\,R$.

While the above mentioned technique is usually *not* a gain for doing a transposition it may be used to speed up the `revbin_permute` itself. Let us operator-ize the idea to see how. Let R be the

revbin-permutation `revbin_permute`, $T(n_r, n_c)$ the transposition of the $n_r \times n_c$ matrix and $R(n_c)$ the `revbin_permute_rows`. Then

$$T(n_r, n_c) \quad = \quad R(n_r) \cdot R \cdot R(n_c) \tag{7.11}$$

The R-operators are their own inverses while T is in general not self inverse[4].

$$R \quad = \quad R(n_r) \cdot T(n_r, n_c) \cdot R(n_c) \tag{7.12}$$

There is a degree of freedom in this formula: for fixed $n = n_r \times n_c$ one can choose one of n_r and n_c (only their product is given).

7.6 The reversed zip-permutation

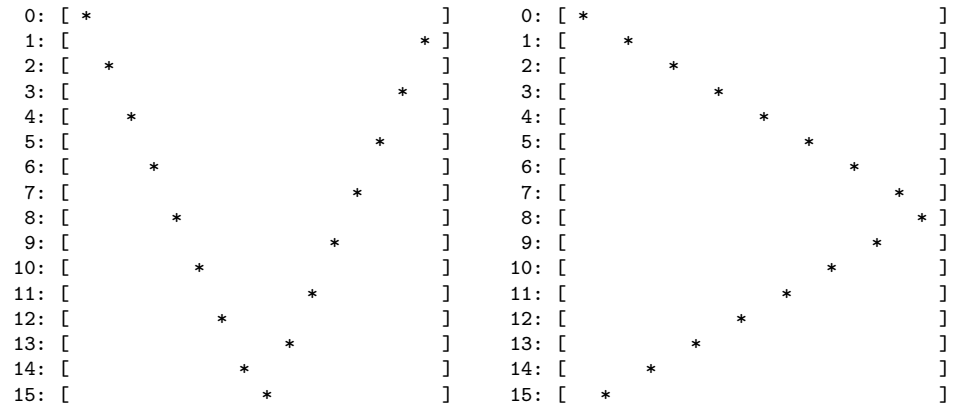


Figure 7.5: Permutation matrices of the reversed zip-permutation (left) and its inverse, the reversed unzip-permutation (right).

A permutation closely related to the zip permutation is

```
template <typename Type>
void zip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k2] = x[k];
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k2] = x[k];
}
```

[FXT: `zip_rev` in `perm/ziprev.h`] The in-place version can (if the array length is a power of two) be implemented as

```
template <typename Type>
void zip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    reverse(x+nh, nh);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    revbin_permute(x, n);
}
```

The inverse permutation, called `unzip_rev`, [FXT: `unzip_rev` in `perm/ziprev.h`] can be implemented as

[4] For $n_r = n_c$ it of course is.

```
template <typename Type>
void unzip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k] = x[k2];
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k] = x[k2];
}
```

The in-place version is

```
template <typename Type>
void unzip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    revbin_permute(x, n);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    reverse(x+nh, nh);
}
```

The given permutation is used in an algorithm where the cosine transform is computed using the Hartley transform (see section 3.6, page 64).

7.7 The XOR permutation

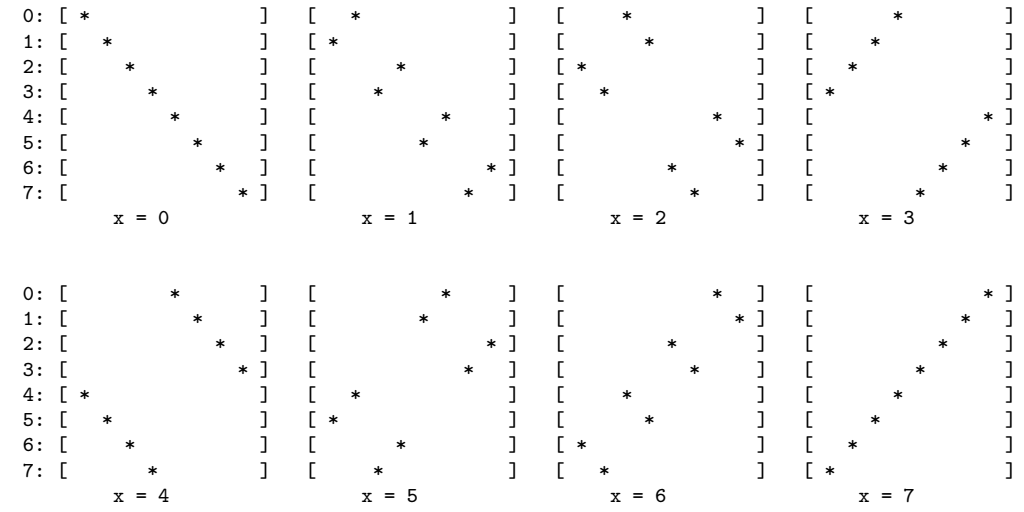


Figure 7.6: Permutation matrices of the XOR permutation for length 8 with parameter $x = 0 \ldots 7$. Compare to the table for the dyadic convolution given on page 86.

The XOR permutation may be explained most simply by its trivial implementation: [FXT: `xor_permute` in `perm/xorpermute.h`]:

```
template <typename Type>
void xor_permute(Type *f, ulong n, ulong x)
{
    if ( 0==x )  return;
    for (ulong k=0; k<n; ++k)
    {
        ulong r = k^x;
        if ( r>k )  swap(f[r], f[k]);
    }
}
```

The XOR permutation is evidently self-inverse. n must be divisible by the smallest power of two that is greater than x: for example, n must be even if $x = 1$, n must be divisible by four if $x = 2$ of $x = 3$. With n a power of two and $x < n$ one is on the safe side.

The XOR permutation contains a few other permutations as important special cases (for simplicity assume that the array length n is a power of two): when the third argument x equals $n-1$ then the permutation is the reversion, with $x = 1$ neighboring even and odd indexed elements are swapped, with $x = n/2$ the upper and the lower half of the array are swapped.

One has

$$X_a\, X_b \;\; = \;\; X_b\, X_a \;=\; X_c \tag{7.13}$$

where $c = a$ XOR b. For the special case $a = b$ the relation expresses the self-inverse property (as $X_0 = \mathrm{id}$).

The XOR permutation often occurs in relations between other permutations where we will use the symbol X_x, the subscript denoting the third argument.

7.8 The Gray code permutation

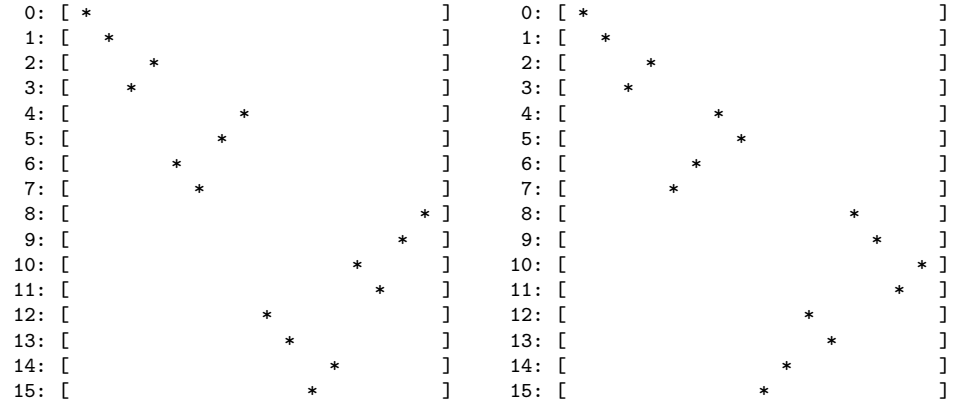


Figure 7.7: Permutation matrices of the Gray code permutation (left) and its inverse (right).

The Gray code permutation reorders (length-2^n) arrays according to the Gray code

```
static inline ulong gray_code(ulong x)
{
    return  x ^ (x>>1);
}
```

which is most easily demonstrated with the according routine that does not work in-place ([FXT: file perm/graypermute.h]):

```
template <typename Type>
inline void gray_permute(const Type *f, Type * restrict g, ulong n)
// after this routine
// g[gray_code(k)] == f[k]
{
    for (ulong k=0; k<n; ++k)  g[gray_code(k)] = f[k];
}
```

Its inverse is

```
template <typename Type>
```

```
inline void inverse_gray_permute(const Type *f, Type * restrict g, ulong n)
// after this routine
// g[k] == f[gray_code(k)]
// (same as: g[inverse_gray_code(k)] == f[k])
{
    for (ulong k=0; k<n; ++k)  g[k] = f[gray_code(k)];
}
```

It also uses calls to `gray_code()` because they are cheaper than the computation of `inverse_gray_code()`, cf. 8.12.

It is actually possible to write an in-place version of the above routines that offers extremely good performance. The underlying observation is that the cycle leaders (cf. 7.13) have an easy pattern and can be efficiently generated using the ideas from 8.4 (detection of perfect powers of two) and 8.9 (enumeration of bit subsets).

```
template <typename Type>
void gray_permute(Type *f, ulong n)
// in-place version
{
    ulong z = 1; // mask for cycle maxima
    ulong v = 0; // ~z
    ulong cl = 1;  // cycle length
    for (ulong ldm=1, m=2;  m<n;  ++ldm, m<<=1)
    {
        z <<= 1;
        v <<= 1;
        if ( is_pow_of_2(ldm) )
        {
            ++z;
            cl <<= 1;
        }
        else  ++v;

        bit_subset b(v);
        do
        {
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                Type tt = f[g];
                f[g] = t;
                t = tt;
                g = gray_code(g);
            }
            f[g] = t;
            // --- end (do cycle) ---
        }
        while ( b.current() );
    }
}
```

The inverse looks similar, the only actual difference is the `do cycle` block:

```
template <typename Type>
void inverse_gray_permute(Type *f, ulong n)
// in-place version
{
    ulong z = 1;
    ulong v = 0;
    ulong cl = 1;
    for (ulong ldm=1, m=2;  m<n;  ++ldm, m<<=1)
    {
        z <<= 1;
        v <<= 1;
        if ( is_pow_of_2(ldm) )
        {
            ++z;
            cl <<= 1;
        }
```

```
        else  ++v;
        bit_subset b(v);
        do
        {
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                f[i] = f[g];
                i = g;
                g = gray_code(i);
            }
            f[i] = t;
            // --- end (do cycle) ---
        }
        while ( b.current() );
    }
}
```

How fast is it? We use the convention that the speed of the trivial (and completely cache-friendly, therefore running at memory bandwidth) `reverse` is 1.0, our hereby declared time unit for comparison. A little benchmark looks like:

```
 CLOCK defined as 1000 MHz  // AMD Athlon 1000MHz with 100MHz DDR RAM
memsize=32768 kiloByte  // permuting that much memory (in chunks of doubles)
               reverse(fr,n2);    dt=  0.0997416   rel=          1  // set to one
        revbin_permute(fr,n2);    dt=   0.594105   rel=    5.95644
               reverse(fr,n2);    dt=  0.0997483   rel=    1.00007
          gray_permute(fr,n2);    dt=   0.119014   rel=    1.19323
               reverse(fr,n2);    dt=  0.0997618   rel=     1.0002
  inverse_gray_permute(fr,n2);    dt=    0.11028   rel=    1.10566
               reverse(fr,n2);    dt=  0.0997424   rel=    1.00001
```

We repeatedly timed `reverse` to get an impression how much we can trust the observed numbers. The bandwidth of the `reverse` is about 320MByte/sec which should be compared to the output of a special memory testing program, revealing that it actually runs at about 83% of the bandwidth one can get without using streaming instructions:

```
avg:   33554432  [ 0]"memcpy"                    305.869 MB/s
avg:   33554432  [ 1]"char *"                    154.713 MB/s
avg:   33554432  [ 2]"short *"                   187.943 MB/s
avg:   33554432  [ 3]"int *"                     300.720 MB/s
avg:   33554432  [ 4]"long *"                    300.584 MB/s
avg:   33554432  [ 5]"long * (4x unrolled)"      306.135 MB/s
avg:   33554432  [ 6]"int64 *"                   305.372 MB/s
avg:   33554432  [ 7]"double *"                  388.695 MB/s  // <--=
avg:   33554432  [ 8]"double * (4x unrolled)"    374.271 MB/s
avg:   33554432  [ 9]"streaming K7"              902.171 MB/s
avg:   33554432  [10]"streaming K7 prefetch"    1082.868 MB/s
avg:   33554432  [11]"streaming K7 clear"       1318.875 MB/s
avg:   33554432  [12]"long * clear"             341.456 MB/s
```

While the `revbin_permute` takes about 6 units (due to its memory access pattern that is very problematic wrt. cache usage) the `gray_permute` only uses 1.20 units, the `inverse_gray_permute` even[5] only 1.10! This is pretty amazing for such a nontrivial permutation.

The described permutation can be used to significantly speed up fast transforms of lengths a power of two, notably the Walsh transform, see chapter 5.

It is instructive to study the complementary masks that occur for cycles off different lengths. The system in the structure of the cycles in `gray_permute(f, 128)` seems non-obvious:

```
  0:  (  2,  3) #=2
  1:  (  4,  7,  5,  6) #=4
```

[5]The observed difference between the forward- and backward version is in fact systematic.

```
  2:  (  8, 15, 10, 12) #=4
  3:  (  9, 14, 11, 13) #=4
  4:  ( 16, 31, 21, 25, 17, 30, 20, 24) #=8
  5:  ( 18, 28, 23, 26, 19, 29, 22, 27) #=8
  6:  ( 32, 63, 42, 51, 34, 60, 40, 48) #=8
  7:  ( 33, 62, 43, 50, 35, 61, 41, 49) #=8
  8:  ( 36, 56, 47, 53, 38, 59, 45, 54) #=8
  9:  ( 37, 57, 46, 52, 39, 58, 44, 55) #=8
 10:  ( 64,127, 85,102, 68,120, 80, 96) #=8
 11:  ( 65,126, 84,103, 69,121, 81, 97) #=8
 12:  ( 66,124, 87,101, 70,123, 82, 99) #=8
 13:  ( 67,125, 86,100, 71,122, 83, 98) #=8
 14:  ( 72,112, 95,106, 76,119, 90,108) #=8
 15:  ( 73,113, 94,107, 77,118, 91,109) #=8
 16:  ( 74,115, 93,105, 78,116, 88,111) #=8
 17:  ( 75,114, 92,104, 79,117, 89,110) #=8
 126 elements in  18 nontrivial cycles.
 cycle lengths:   2 ...   8
2 fixed points: [0. 1]
```

However, one can identify the cycle maxima as the set of different possible values that consist of the bits of an `INV`-mask combined with all variations of bits of the `VAR`-mask (filled in as `max` for $n < 128$):

```
--------------------------
 ldm= 1:     1 cycles of length= 2 [2..3]
max: ......................11 = 3  cl=2
     ......................11 = INV
     ........................ = VAR
--------------------------
 ldm= 2:     1 cycles of length= 4 [4..7]
max: .....................111 = 7  cl=4
     .....................111 = INV
     ........................ = VAR

 ldm= 3:     2 cycles of length= 4 [8..f]
max: ....................1111 = 15  cl=4
max: .....................111. = 14  cl=4
     .....................111. = INV
     ......................1 = VAR
--------------------------
 ldm= 4:     2 cycles of length= 8 [10..1f]
max: ...................11111 = 31  cl=8
max: ...................111.1 = 29  cl=8
     ...................111.1 = INV
     ......................1. = VAR

 ldm= 5:     4 cycles of length= 8 [20..3f]
max: ..................111.11 = 59  cl=8
max: ..................11111. = 62  cl=8
max: ..................111111 = 63  cl=8
max: ..................111.1. = 58  cl=8
     ..................111.1. = INV
     ....................1.1 = VAR

 ldm= 6:     8 cycles of length= 8 [40..7f]
max: ..................111.1.1 = 117  cl=8
max: ..................111.11. = 118  cl=8
max: ..................111.111 = 119  cl=8
max: ..................11111.. = 124  cl=8
max: ..................11111.1 = 125  cl=8
max: ..................111111. = 126  cl=8
max: ..................1111111 = 127  cl=8
max: ..................111.1.. = 116  cl=8
     ..................111.1.. = INV
     ....................1.11 = VAR

 ldm= 7:    16 cycles of length= 8 [80..ff]
max: [...]
     ................111.1... = INV
     ...................1.111 = VAR
--------------------------
 ldm= 8:    16 cycles of length=16 [100..1ff]
max: [...]
     ...............111.1...1 = INV
     ..................1.111. = VAR

 ldm= 9:    32 cycles of length=16 [200..3ff]
max: [...]
     ...............111.1...1. = INV
     .................1.111.1 = VAR
```

7.9 The reversed Gray code permutation

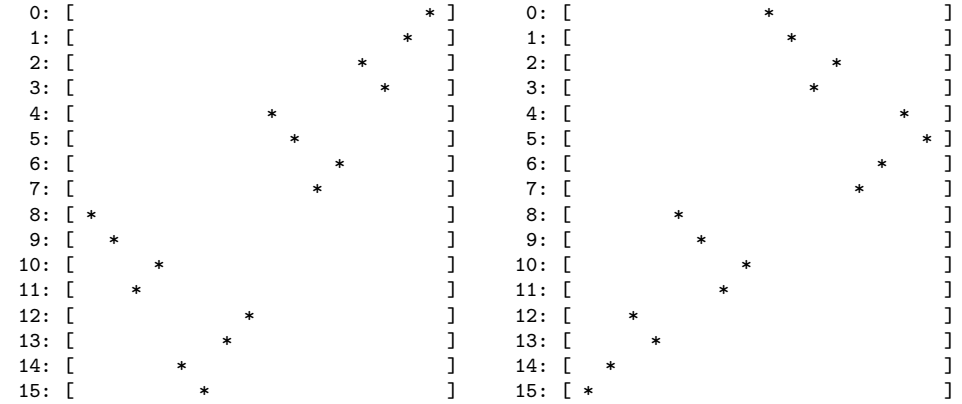


Figure 7.8: Permutation matrices of the reversed Gray code permutation (left) and its inverse (right).

If the length-n array is permuted in the way the upper half of the length-$2n$ array would be permuted by `gray_permute()` then all cycles are of the same length. The resulting permutation is equivalent to the reversed Gray code permutation:

```
template <typename Type>
inline void gray_rev_permute(const Type *f, Type * restrict g, ulong n)
// gray_rev_permute() =^=
//  { reverse(); gray_permute(); }
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)  g[gray_code(m)] = f[k];
}
```

The routine, its inverse and in-place versions can be found in [FXT: file `perm/grayrevpermute.h`].

All cycles have the same length, `gray_rev_permute(f, 64)` gives:

```
 0:  (  0, 63, 21, 38,  4, 56, 16, 32) #=8
 1:  (  1, 62, 20, 39,  5, 57, 17, 33) #=8
 2:  (  2, 60, 23, 37,  6, 59, 18, 35) #=8
 3:  (  3, 61, 22, 36,  7, 58, 19, 34) #=8
 4:  (  8, 48, 31, 42, 12, 55, 26, 44) #=8
 5:  (  9, 49, 30, 43, 13, 54, 27, 45) #=8
 6:  ( 10, 51, 29, 41, 14, 52, 24, 47) #=8
 7:  ( 11, 50, 28, 40, 15, 53, 25, 46) #=8
 64 elements in   8 nontrivial cycles.
 cycle length is == 8
 No fixed points.
```

If 64 is added to the cycle elements then the cycles in the upper half of the array as in of `gray_permute(f, 128)` are reproduced (this is by construction).

Let G denote the Gray code permutation, $\bar{G}$ the reversed Gray code permutation. Symbolically one can write

$$G(n) = \{\dots, \bar{G}(n/8), \bar{G}(n/4), \bar{G}(n/2)\} \tag{7.14}$$

$$G^{-1}(n) = \{\dots, \bar{G}^{-1}(n/8), \bar{G}^{-1}(n/4), \bar{G}^{-1}(n/2)\} \tag{7.15}$$

Now let r the reversion and h the permutation that swaps the upper and the lower half of an array. Then

$$\bar{G} = G\,r \tag{7.16}$$

$$\bar{G}^{-1} = r\,G^{-1} \tag{7.17}$$

$$\bar{G}^{-1}\,G = G^{-1}\,\bar{G} = r = X_{n-1} \tag{7.18}$$

$$G\,\bar{G}^{-1} = \bar{G}\,G^{-1} = h = X_{n/2} \tag{7.19}$$

Throughout it is assumed that the array length n is a power of two.

7.10 The green code permutation

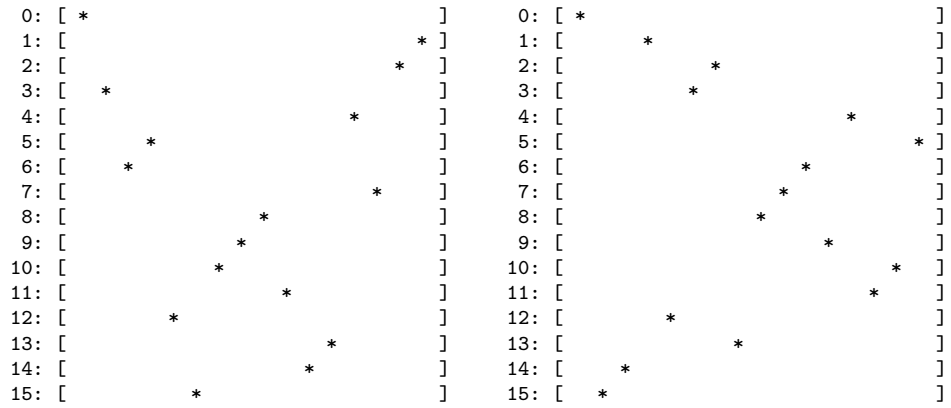


Figure 7.9: Permutation matrices of the green code permutation (left) and its inverse (right).

The green code permutation is obtained by replacing the calls to `gray_code` by `green_code` in the Gray code permutation. An additional step is required: the highest bit must be masked out in order to keep indices in the range.

[FXT: `green_permute` in `perm/greenpermute.h`]:

```
template <typename Type>
inline void green_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0;  k<n;  ++k)
    {
        ulong r = green_code(k);
        r &= (n-1);
        g[r] = f[k];
    }
}
```

The inverse is

```
template <typename Type>
inline void inverse_green_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0;  k<n;  ++k)
    {
        ulong r = green_code(k);
        r &= (n-1);
        g[k] = f[r];
    }
}
```

7.11 The reversed green code permutation

The analogue to `gray_rev_permute` for the green permutation is

```
template <typename Type>
inline void green_rev_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)
```

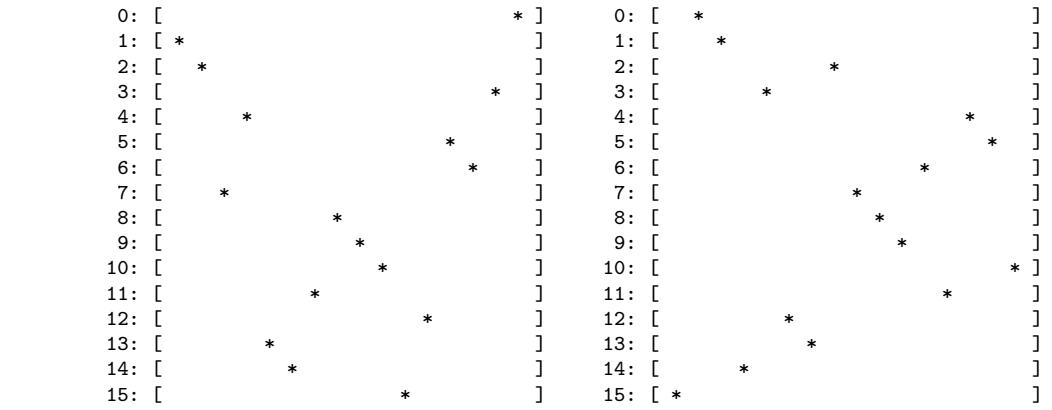


Figure 7.10: Permutation matrices of the reversed green code permutation (left) and its inverse (right).

```
    {
        ulong r = green_code(m);
        r &= (n-1);
        g[r] = f[k];
    }
}
```

The inverse is

```
template <typename Type>
inline void inverse_green_rev_permute(const Type *f, Type * restrict g, ulong n)
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)
    {
        ulong r = green_code(m);
        r &= (n-1);
        g[k] = f[r];
    }
}
```

Let E denote the green code permutation, $\bar{E}$ the reversed green code permutation and R the revbin-permutation. Further assume that the array length n is a power of two. Then

$$E = R\,G\,R \tag{7.20}$$
$$E^{-1} = R\,G^{-1}\,R \tag{7.21}$$
$$\bar{E} = E\,r \tag{7.22}$$
$$\bar{E}^{-1} = r\,E^{-1} \tag{7.23}$$
$$\bar{E} = R\,\bar{G}\,R \tag{7.24}$$
$$\bar{E}^{-1} = R\,\bar{G}^{-1}\,R \tag{7.25}$$

The relations between E and $\bar{E}$ are

$$\bar{E}^{-1}\,E = E^{-1}\,\bar{E} = r = X_{n-1} \tag{7.26}$$
$$E\,\bar{E}^{-1} = \bar{E}\,E^{-1} = X_1 \tag{7.27}$$

where X_x is the XOR permutation (with third argument equal to the subscript x, see section 7.7, page 132).

Let $\bar{Z}$ be the `zip_rev`-permutation described in section 7.6, page 131. Then

$$\bar{Z} = G^{-1}\,E \tag{7.28}$$
$$\bar{Z}^{-1} = E^{-1}\,G \tag{7.29}$$

With an implementation of E and its inverse that is as fast as the Gray code permutation (TBD) the given relations constitute an algorithm that should be significantly faster than the triple revbin permute code given on page 131.

7.12 Factorizing permutations

In this section we will see some algorithms that use a certain type of decomposition (factorization in term of matrices) of some permutations we have studied so far. The resulting algorithms involve proportional $n \cdot \log(n)$ computations for length-n arrays. This might seem to render the schemes worthless (we generally can obtain a permutation with work proportional to n). There are, however, situations where on can use the algorithms advantageously. Firstly, with bit manipulations, where the whole binary words are modified in one (or a few) statements. The corresponding algorithms are therefore only proportional $\log(n)$. Secondly, when the algorithm can be used implicitly in order to integrate the permutation in a fast transform. The work can sometimes be reduced to zero in that case.
The most simple example might the reversion via

```
template <typename Type>
void perm1(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong ldk=1; ldk<=ldn; ++ldk)  // starting with length 2
    {
        ulong k = 1UL<<ldk;
        for (ulong j=0; j<n; j+=k)  func(f+j, ldk);
    }
}
```

where `func` swaps the upper and lower half of an array:

```
template <typename Type>
void func(Type *f, ulong ldn)  { swap(f, f+n/2, n/2); }
```

This idea has been exploited in section 8.7 in order to obtain a bit-reversal routine. The bit-reversal is self-inverse, therefore one can alternatively execute the steps in reverse order and still get the same permutation:

```
template <typename Type>
void perm2(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong ldk=ldn; ldk>0; --ldk)  // starting with full length
    {
        ulong k = 1UL<<ldk;
        for (ulong j=0; j<n; j+=k)  func(f+j, ldk);
    }
}
```

Note that `func` in turn can be obtained via

```
    reverse(f, n);
    reverse(f, n/2);  reverse(f+n/2, n/2);
```

or the same statements in reversed order.
Let us try a not so obvious example, use `perm1` with `func` defined as:

```
    swap(f+n/2, f+n/2+n/4, n/4);
```

The resulting permutation is the Gray-permutation. The other way round (using `perm2`) one gets the inverse Gray-permutation. Using `func` defined as

```
    reverse(f+n/2, n/2);
```

one gets the Gray-permutation through `perm2`, its inverse through `perm1`. This idea has been used in the core-routine for the sequency-ordered Walsh transform described in section 5.6. The work for the Gray-permutation has been completely vaporized there. Note that the routine that swaps the halves of the upper half array could be obtained as either of

```
    inverse_gray_permute(f, n);
    gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
//  =^=
//    inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
//    gray_permute(f, n);
```

Similarly, the routine that reverses the upper half can be obtained as

```
    gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
    inverse_gray_permute(f, n);
//  =^=
//    gray_permute(f, n);
//    inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
```

Using `func` defined as

```
    swap(f+n/4, f+n/2, n/4);
```

and `perm2` one gets the zip-permutation, `perm1` gives the inverse.

Using `func` (lazily implementated as)

```
    Type g[n];
    copy(f, g, n/4);              // quarter: 0 -->
    copy(f+n/4, g+n/2+n/4, n/4);  // 1 --> 3
    copy(f+n/2, g+n/4, n/4);      // 2 --> 1
    copy(f+n/2+n/4, g+n/2, n/4);  // 3 --> 2
    copy(g, f, n);
```

which was obtained using

```
    zip_rev(f, n);
    unzip_rev(f, n/2);  unzip_rev(f+n/2, n/2);
```

both the reversed zip-permutation and its inverse can be computed in the now hopefully obvious way.

The revbin-permutation can be generated through the zip-permutation or its inverse. However, the zip-permutation is the more complicate one, so absorbing the revbin-permutation into fast transforms does not seem to be easy. The other way round it makes more sense:

```
    revbin_permute(f, n/2);  revbin_permute(f+n/2, n/2);
    revbin_permute(f, n);
```

Is a convenient (though not the most effective) way to compute the zip-permutation.

Clearly, the idea presented here is in analogy with the decomposition of linear transforms. Finally the permutations are (very simple forms of) linear transforms.

7.13 General permutations

So far we treated special permutations that occurred as part of other algorithms. It is instructive to study permutations in general with the operations (as composition and inverse) on them.

7.13.1 Basic definitions

A straight forward way to describe a permutation is to consider the array of indices that for the original (un-permuted) data would be the length-n *canonical* sequence 0, 1, 2, ..., $n-1$. The mentioned trivial

sequence describes the 'do-nothing' permutation or *identity* (wrt. composition of permutations). The concept is best described by the routine that *applies* a given permutation x on an array of data f: after the routine has finished the array g will contain the elements of f reordered according to x

```
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// apply x[] on f[]
// i.e.  g[k] <-- f[x[k]]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x[k]];
}
```

[FXT: `apply_permutation` in `perm/permapply.h`] An example using strings (arrays of characters): The permutation described by $x = \{7, 6, 3, 2, 5, 1, 0, 4\}$ and the input data
$f =$`"ABadCafe"` would produce
$g =$`"efdaaBAC"`

All routines in this and the following section are declared in [FXT: file `perm/permutation.h`]

Trivially

```
bool is_identity(const ulong *f, ulong n)
// check whether f[] is the identical permutation,
// i.e. whether f[k]==k for all k= 0...n-1
{
    for (ulong k=0; k<n; ++k)  if ( f[k] != k )  return false;
    return true;
}
```

A fixed point of a permutation is an index where the element is not moved:

```
ulong count_fixed_points(const ulong *f, ulong n)
// return number of fixed points in f[]
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  ++ct;
    return ct;
}
```

A *derangement* is a permutation that has no fixed points (i.e. that moved every element to another position so `count_fixed_points()` returns zero). To check whether a permutation is a derangement of identity use:

```
bool is_derangement(const ulong *f, ulong n)
//
// check whether f[] is a derangement of identity
// i.e. whether f[k]!=k for all k
//
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  return false;
    return true;
}
```

Whether two permutations are derangements of each other is obviously found by:

```
bool is_derangement(const ulong *f, const ulong *g, ulong n)
// check whether f[] is a derangement of g[],
// i.e. whether f[k]!=g[k] for all k
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == g[k] )  return false;
    return true;
}
```

To check whether a given array really describes a valid permutation one has to verify that each index appears exactly once. The `bitarray` class described in 10.6 allows us to do the job without modification of the input (like e.g. sorting):

```
bool is_valid_permutation(const ulong *f, ulong n, bitarray *bp/*=0*/)
// check whether all values 0...n-1 appear exactly once
{
    // check whether any element is out of range:
    for (ulong k=0; k<n; ++k)  if ( f[k]>=n )  return false;

    // check whether values are unique:
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong k;
    for (k=0; k<n; ++k)
    {
        if ( tp->test_set(f[k]) )  break;
    }

    if ( 0==bp )  delete tp;

    return  (k==n);
}
```

7.13.2 Compositions of permutations

One can apply arbitrary many permutations to an array, one by one. The resulting permutation is called the *composition* of the applied permutations. As an example, the check whether some permutation g is equal to f applied twice, or $f \cdot f$, or f *squared* use:

```
bool is_square(const ulong *f, const ulong *g, ulong n)
// whether f * f == g  as a permutation
{
    for (ulong k=0; k<n; ++k)  if ( g[k] != f[f[k]] )  return false;
    return true;
}
```

A permutation f is said to be the *inverse* of another permutation g if it undoes its effect, that is $f \cdot g = id$ (likewise $g \cdot f = id$):

```
bool is_inverse(const ulong *f, const ulong *g, ulong n)
// check whether f[] is inverse of g[]
{
    for (ulong k=0; k<n; ++k)  if ( f[g[k]] != k )  return false;
    return true;
}
```

A permutation that is its own inverse (like the revbin-permutation) is called an *involution*. Checking that is easy:

```
bool is_involution(const ulong *f, ulong n)
// check whether max cycle length is <= 2
{
    for (ulong k=0; k<n; ++k)  if ( f[f[k]] != k )  return false;
    return true;
}
```

Finding the inverse of a given permutation is trivial:

```
void make_inverse(const ulong *f, ulong * restrict g, ulong n)
// set g[] to the inverse of f[]
{
    for (ulong k=0; k<n; ++k)  g[f[k]] = k;
}
```

However, if one wants to do the operation in-place a little bit of thought is required. The idea underlying all subsequent routines working in-place is that every permutation entirely consists of disjoint cycles. A *cycle* (of a permutation) is a subset of the indices that is rotated (by one) by the permutation. The term *disjoint* means that the cycles do not 'cross' each other. While this observation is pretty trivial it allows us to do many operations by following the cycles of the permutation, one by one, and doing the necessary operation on each of them. As an example consider the following permutation of an array originally consisting of the (canonical) sequence 0, 1, ..., 15 (extra spaces inserted for readability):

```
0, 1, 3, 2,   7, 6, 4, 5,   15, 14, 12, 13,   8, 9, 11, 10
```

There are two fixed points (0 and 1) and these cycles:

```
( 2 <--  3 )
( 4 <--  7 <--  5 <--  6 )
( 8 <-- 15 <-- 10 <-- 12 )
( 9 <-- 14 <-- 11 <-- 13 )
```

The cycles do 'wrap around', e.g. the initial 4 of the second cycle goes to position 6, the last element of the second cycle.
Note that the inverse permutation could formally be described by reversing every arrow in each cycle:

```
( 2 -->  3 )
( 4 -->  7 -->  5 -->  6 )
( 8 --> 15 --> 10 --> 12 )
( 9 --> 14 --> 11 --> 13 )
```

Equivalently, one can reverse the order of the elements in each cycle:

```
( 3 <--  2 )
( 6 <--  5 <--  7 <--  4 )
(12 <-- 10 <-- 15 <--  8 )
(13 <-- 11 <-- 14 <--  9 )
```

If we begin each cycle with its smallest element the inverse permutation looks like:

```
( 2 <--  3 )
( 4 <--  6 <--  5 <--  7 )
( 8 <-- 12 <-- 10 <-- 15 )
( 9 <-- 13 <-- 11 <-- 14 )
```

The last three sets of cycles all describe the same permutation:

```
0, 1, 3, 2,   6, 7, 5, 4,   12, 13, 15, 14,   10, 11, 9, 8
```

The maximal cycle-length of an involution is 2, that means it completely consists of fixed points and 2-cycles (swapped pairs of indices).

As a warm-up look at the code used to print the cycles of the above example (which by the way is the Gray-permutation of the canonical length-16 array):

```
ulong print_cycles(const ulong *f, ulong n, bitarray *bp=0)
// print the cycles of the permutation
// return number of fixed points
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong ct = 0;  // # of fixed points
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // follow a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        if ( g==i )  // fixed point ?
        {
            ++ct;
            continue;
        }

        cout << "(" << setw(3) << i;
        while ( 0==(tp->test_set(g)) )
        {
            cout << " <-- " << setw(3) << g;
```

```
            g = f[g];
        }
        cout << " )" << endl;
    }
    if ( 0==bp )  delete tp;
    return  ct;
}
```

The bit-array is used to keep track of the elements already processed.
For the computation of the inverse we have to reverse each cycle:

```
void make_inverse(ulong *f, ulong n, bitarray *bp/*=0*/)
// set f[] to its own inverse
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);
        // invert a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        while ( 0==(tp->test_set(g)) )
        {
            ulong t = f[g];
            f[g] = i;
            i = g;
            g = t;
        }
        f[g] = i;
    }
    if ( 0==bp )  delete tp;
}
```

Similarly for the straightforward

```
void make_square(const ulong *f, ulong * restrict g, ulong n)
// set g[] = f[] * f[]
{
    for (ulong k=0; k<n; ++k)  g[k] = f[f[k]];
}
```

whose in-place version is

```
void make_square(ulong *f, ulong n, bitarray *bp/*=0*/)
// set f[] to f[] * f[]
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);
        // square a cycle:
        ulong i = k;
        ulong t = f[i];  // save
        ulong g = f[i];  // next index
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = f[g];
        }
        f[i] = t;
```

```
    }
    if ( 0==bp )  delete tp;
}
```

Random permutations are sometimes useful:

```
void random_permute(ulong *f, ulong n)
// randomly permute the elements of f[]
{
    for (ulong k=1; k<n; ++k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16;  // avoid using low bits of rand alone
        ulong i = r % (k+1);
        swap(f[k], f[i]);
    }
}
```

and

```
void random_permutation(ulong *f, ulong n)
// create a random permutation
{
    for (ulong k=0; k<n; ++k)  f[k] = k;
    random_permute(f, n);
}
```

7.13.3 Applying permutations to data

The following routines are from [FXT: file `perm/permapply.h`].

The in-place analogue of the routine `apply` shown near the beginning of section 7.13 is:

```
template <typename Type>
void apply_permutation(const ulong *x, Type *f, ulong n, bitarray *bp=0)
// apply x[] on f[]  (in-place operation)
// i.e.  f[k] <-- f[x[k]]  \forall k
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);
        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. inverse_gray_permute()
        {
            f[i] = f[g];
            i = g;
            g = x[i];
        }
        f[i] = t;
        // --- end (do cycle) ---
    }
    if ( 0==bp )  delete tp;
}
```

Often one wants to apply the inverse of a permutation without actually inverting the permutation itself. This leads to

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// apply inverse of x[] on f[]
```

```
// i.e.  g[x[k]] <-- f[k]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[x[k]] = f[k];
}
```

whereas the in-place version is

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// apply inverse of x[] on f[]  (in-place operation)
// i.e.  f[x[k]] <-- f[k]  \forall k
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. gray_permute()
        {
            Type tt = f[g];
            f[g] = t;
            t = tt;
            g = x[g];
        }
        f[g] = t;
        // --- end (do cycle) ---
    }

    if ( 0==bp )  delete tp;
}
```

Finally let us remark that an analogue of the binary powering algorithm exists wrt. composition of permutations. [FXT: `power` in `perm/permutation.cc`]

7.14 Generating all Permutations

In this section a few algorithms for the generation of all permutations are presented. These are typically useful in situations where an exhaustive search over all permutations is needed. At the time of writing the pre-fascicles of Knuth's *The Art of Computer Programming* Volume 4 are available. Therefore (1) the title of this section is not anymore 'Enumerating all permutations' and (2) I will not elaborate on the underlying algorithms, for a thorough discussion see Knuth's text.

7.14.1 Lexicographic order

When generated in lexicographic order the permutations appear as if (read as numbers and) sorted numerically:

```
       permutation  sign
#  0:   0 1 2 3     +
#  1:   0 1 3 2     -
#  2:   0 2 1 3     -
#  3:   0 2 3 1     +
#  4:   0 3 1 2     +
#  5:   0 3 2 1     -
#  6:   1 0 2 3     -
#  7:   1 0 3 2     +
#  8:   1 2 0 3     +
#  9:   1 2 3 0     -
# 10:   1 3 0 2     -
# 11:   1 3 2 0     +
# 12:   2 0 1 3     +
```

```
 # 13:   2 0 3 1     -
 # 14:   2 1 0 3     -
 # 15:   2 1 3 0     +
 # 16:   2 3 0 1     +
 # 17:   2 3 1 0     -
 # 18:   3 0 1 2     -
 # 19:   3 0 2 1     +
 # 20:   3 1 0 2     +
 # 21:   3 1 2 0     -
 # 22:   3 2 0 1     -
 # 23:   3 2 1 0     +
```

The *sign* given is plus or minus if the (minimal) number of transpositions is even or odd, respectively. The minimalistic `class perm_lex` implementing the algorithm is

```
class perm_lex
{
protected:
    ulong n;   // number of elements to permute
    ulong *p;  // p[n] contains a permutation of {0, 1, ..., n-1}
    ulong idx; // incremented with each call to next()
    ulong sgn; // sign of the permutation
public:
    perm_lex(ulong nn)
    {
        n = (nn > 0 ? nn : 1);
        p = new ulong[n];
        first();
    }

    ~perm_lex() { delete [] p; }

    void first()
    {
        for (ulong i=0; i<n; i++)  p[i] = i;
        sgn = 0;
        idx = 0;
    }

    ulong next();
    ulong current()  const  { return idx; }
    ulong sign()  const  { return sgn; }  // 0 for sign +1,  1 for sign -1
    const ulong *data()  const  { return p; }
};
```

[FXT: `class perm_lex` in `perm/permlex.h`] The only nontrivial part is the `next()`-method that computes the next permutation with each call:

```
ulong perm_lex::next()
{
    const ulong n1 = n - 1;
    ulong i = n1;
    do
    {
        --i;
        if ( (long)i<0 )  return  0;  // last sequence is falling seq.
    }
    while ( p[i] > p[i+1] );

    ulong j = n1;
    while ( p[i] > p[j] )  --j;

    swap(p[i], p[j]);  sgn ^= 1;

    ulong r = n1;
    ulong s = i + 1;
    while ( r > s )
    {
        swap(p[r], p[s]);  sgn ^= 1;
        --r;
        ++s;
    }

    ++idx;
    return  idx;
}
```

The routine is based on code by Glenn Rhoads who in turn ascribes the algorithm to Dijkstra. [FXT: `perm_lex::next` in `perm/permlex.cc`]

Using the above is no black magic:

```
perm_lex perm(n);
const ulong *x = perm.data();
do
{
    // do something, e.g. just print the permutation:
    for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
    cout << endl;
}
while ( perm.next() );
```

cf. [FXT: file demo/permlex-demo.cc]

7.14.2 Minimal-change order

When generated in minimal-change order[6] the permutations in a way that between each consecutive two exactly two elements are swapped:

```
       permutation   swap    inverse p.
 #  0:   0 1 2 3    (0, 0)    0 1 2 3
 #  1:   0 1 3 2    (3, 2)    0 1 3 2
 #  2:   0 3 1 2    (2, 1)    0 2 3 1
 #  3:   3 0 1 2    (1, 0)    1 2 3 0
 #  4:   3 0 2 1    (3, 2)    1 3 2 0
 #  5:   0 3 2 1    (0, 1)    0 3 2 1
 #  6:   0 2 3 1    (1, 2)    0 3 1 2
 #  7:   0 2 1 3    (2, 3)    0 2 1 3
 #  8:   2 0 1 3    (1, 0)    1 2 0 3
 #  9:   2 0 3 1    (3, 2)    1 3 0 2
 # 10:   2 3 0 1    (2, 1)    2 3 0 1
 # 11:   3 2 0 1    (1, 0)    2 3 1 0
 # 12:   3 2 1 0    (3, 2)    3 2 1 0
 # 13:   2 3 1 0    (0, 1)    3 2 0 1
 # 14:   2 1 3 0    (1, 2)    3 1 0 2
 # 15:   2 1 0 3    (2, 3)    2 1 0 3
 # 16:   1 2 0 3    (0, 1)    2 0 1 3
 # 17:   1 2 3 0    (3, 2)    3 0 1 2
 # 18:   1 3 2 0    (2, 1)    3 0 2 1
 # 19:   3 1 2 0    (1, 0)    3 1 2 0
 # 20:   3 1 0 2    (2, 3)    2 1 3 0
 # 21:   1 3 0 2    (0, 1)    2 0 3 1
 # 22:   1 0 3 2    (1, 2)    1 0 3 2
 # 23:   1 0 2 3    (2, 3)    1 0 2 3
```

Note that the swapped pairs are always neighboring elements. Often one will only use the indices of the swapped elements to update the visited configurations. A property of the algorithm used is that the inverse permutations are available. The corresponding `class perm_minchange` is

```
class perm_minchange
{
protected:
    ulong n;   // number of elements to permute
    ulong *p;  // p[n] contains a permutation of {0, 1, ..., n-1}
    ulong *ip; // ip[n] contains the inverse permutation of p[]
    ulong *d;  // aux
    ulong *ii; // aux
    ulong sw1, sw2; // index of elements swapped most recently
    ulong idx; // incremented with each call to next()
public:
    perm_minchange(ulong nn);
    ~perm_minchange();
    void first();
    ulong next()  { return make_next(n-1); }
    ulong current()  const  { return idx; }
    ulong sign()  const  { return  idx & 1; }  // 0 for sign +1,  1 for sign -1
```

[6] There is more than one minimal change order, e.g. reversing the order yields another one.

```
    const ulong *data()  const  { return p; }
    const ulong *invdata()  const  { return ip; }
    void get_swap(ulong &s1, ulong &s2)  const  { s1=sw1; s2=sw2; }
protected:
    ulong make_next(ulong m);
};
```

[FXT: `class perm_minchange` in `perm/permminchange.h`]

The algorithm itself can be found in [FXT: `perm_minchange::make_next` in `perm/permminchange.cc`]

```
ulong perm_minchange::make_next(ulong m)
{
    ulong i = ii[m];
    ulong ret = 1;

    if ( i==m )
    {
        d[m] = -d[m];
        if ( 0!=m )  ret = make_next(m-1);
        else         ret = 0;
        i = -1UL;
    }

    if ( (long)i>=0 )
    {
        ulong j = ip[m];
        ulong k = j + d[m];
        ulong z = p[k];
        p[j] = z;
        p[k] = m;
        ip[z] = j;
        ip[m] = k;

        sw1 = j;  // note that sw1 == sw2 +-1 (adjacent positions)
        sw2 = k;
        ++idx;
    }

    ++i;
    ii[m] = i;

    return  ret;
}
```

The central block (`if ( (long)i>=0 ) {...}`) is based on code by Frank Ruskey / Glenn Rhoads. The data is initialized by

```
void perm_minchange::first()
{
    for (ulong i=0; i<n; i++)
    {
        p[i] = ip[i] = i;
        d[i] = -1UL;
        ii[i] = 0;
    }
    sw1 = sw2 = 0;
    idx = 0;
}
```

Usage of the class is straightforward:

```
    perm_minchange perm(n);
    const ulong *x = perm.data();
    const ulong *ix = perm.invdata();
    ulong sw1, sw2;
    do
    {
        // do something, e.g. just print the permutation:
        for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
        // sometimes one only uses the indices swapped ...
        perm.get_swap(sw1, sw2);
        cout << "  swap: (" << sw1 << ", " << sw2 << ") ";
        // ... inverse permutation courtesy of the algorithm
```

```
        for (ulong i=0; i<n; ++i)  cout << ix[i] << " ";
    }
    while ( perm.next() );
```

Cf. also [FXT: file demo/permminchange-demo.cc]

An alternative implementation using the algorithm of Trotter (based on code by Helmut Herold) can be found in [FXT: perm_trotter::make_next in perm/permtrotter.cc]

```
void perm_trotter::make_next()
{
    ++idx_;
    ulong k = 0;
    ulong m = 0;
    yy_ = p_[m] + d_[m];
    p_[m] = yy_;
    while ( (yy_==n_-m) || (yy_==0) )
    {
        if ( yy_==0 )
        {
            d_[m] = 1;
            k++;
        }
        else  d_[m] = -1UL;
        if ( m==n_-2 )
        {
            sw1_ = n_ - 1;
            sw2_ = n_ - 2;
            swap(x_[sw1_], x_[sw2_]);
            yy_ = 1;
            idx_ = 0;
            return;
        }
        else
        {
            m++;
            yy_ = p_[m] + d_[m];
            p_[m] = yy_;
        }
    }
    sw1_ = yy_ + k;  // note that sw1 == sw2 + 1 (adjacent positions)
    sw2_ = sw1_ - 1;
    swap(x_[sw1_], x_[sw2_]);
}
```

The corresponding class perm_trotter, however, does not produce the inverse permutations.

7.14.3 Derangement order

The following enumeration of permutations is characterized by the fact that two successive permutations have no element at the same position:

```
#  0:   0 1 2 3
#  1:   3 0 1 2
#  2:   1 2 3 0
#  3:   2 3 0 1
#  4:   1 0 2 3
#  5:   3 1 0 2
#  6:   0 2 3 1
#  7:   2 3 1 0
#  8:   1 2 0 3
#  9:   3 1 2 0
# 10:   2 0 3 1
# 11:   0 3 1 2
# 12:   2 1 0 3
# 13:   3 2 1 0
# 14:   1 0 3 2
# 15:   0 3 2 1
# 16:   2 0 1 3
# 17:   3 2 0 1
# 18:   0 1 3 2
# 19:   1 3 2 0
# 20:   0 2 1 3
# 21:   3 0 2 1
# 22:   2 1 3 0
# 23:   1 3 0 2
```

There is no such sequence for $n = 3$.

The utility class, that implements the underlying algorithm is [FXT: `class perm_derange` in `perm/permderange.h`]. The central piece of code is [FXT: `perm_derange::make_next` in `perm/permderange.cc`]:

```
void perm_derange::make_next()
{
    ++idx_;
    ++idxm_;

    if ( idxm_>=n_ ) // every n steps: need next perm_trotter
    {
        idxm_ = 0;
        if ( 0==pt->next() )
        {
            idx_ = 0;
            return;
        }

        // copy in:
        const ulong *xx = pt->data();
        for (ulong k=0; k<n_-1; ++k)  x_[k] = xx[k];
        x_[n_-1] = n_-1;  // last element
    }
    else  // rotate
    {
        if ( idxm_==n_-1 )
        {
            rotl1(x_, n_);
        }
        else // last two swapped
        {
            rotr1(x_, n_);
            if ( idxm_==n_-2 )  rotr1(x_, n_);
        }
    }
}
```

The above listing can be generated via

```
    ulong n = 4;
    perm_derange perm(n);
    const ulong *x = perm.data();
    do
    {
        cout << " #"; cout.width(3); cout << perm.current() << ":   ";
        for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
        cout << endl;
    }
    while ( perm.next() );
```

[FXT: file `demo/permderange-demo.cc`]

7.14.4 Star-transposition order

Knuth [fasc2B p.19] gives an algorithm that generates the permutations ordered in a way that each two successive entries in the list differ by a swap of element zero with some other element (star transposition):

```
 #  0:   0 1 2 3   swap: (0, 3)
 #  1:   1 0 2 3   swap: (0, 1)
 #  2:   2 0 1 3   swap: (0, 2)
 #  3:   0 2 1 3   swap: (0, 1)
 #  4:   1 2 0 3   swap: (0, 2)
 #  5:   2 1 0 3   swap: (0, 1)
 #  6:   3 1 0 2   swap: (0, 3)
 #  7:   0 1 3 2   swap: (0, 2)
 #  8:   1 0 3 2   swap: (0, 1)
 #  9:   3 0 1 2   swap: (0, 2)
 # 10:   0 3 1 2   swap: (0, 1)
 # 11:   1 3 0 2   swap: (0, 2)
```

```
# 12:   2 3 0 1   swap: (0, 3)
# 13:   3 2 0 1   swap: (0, 1)
# 14:   0 2 3 1   swap: (0, 2)
# 15:   2 0 3 1   swap: (0, 1)
# 16:   3 0 2 1   swap: (0, 2)
# 17:   0 3 2 1   swap: (0, 1)
# 18:   1 3 2 0   swap: (0, 3)
# 19:   2 3 1 0   swap: (0, 2)
# 20:   3 2 1 0   swap: (0, 1)
# 21:   1 2 3 0   swap: (0, 2)
# 22:   2 1 3 0   swap: (0, 1)
# 23:   3 1 2 0   swap: (0, 2)
```

The implementation of the algorithm, ascribed to Gideon Ehrlich, can be found in [FXT: `class perm_star` in `perm/permstar.h`]

The above listing can be obtained with

```
ulong n = 4;
perm_star perm(n);
const ulong *x = perm.data();
ulong ct = 0;
do
{
    cout << " #"; cout.width(3); cout << ct << ":   ";
    for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
    cout << "  swap: (" << 0 << ", " << perm.get_swap() << ") ";
    cout << endl;
    ++ct;
}
while ( perm.next() );
```

[FXT: file `demo/permstar-demo.cc`]

7.14.5 An order from graph traversal

... to enumerate all permutations of n elements was given in [25]:

```
#  0:   0 1 2 3
#  1:   0 1 3 2
#  2:   0 2 1 3
#  3:   0 3 1 2
#  4:   0 2 3 1
#  5:   0 3 2 1
#  6:   1 0 2 3
#  7:   1 0 3 2
#  8:   2 0 1 3
#  9:   3 0 1 2
# 10:   2 0 3 1
# 11:   3 0 2 1
# 12:   1 2 0 3
# 13:   1 3 0 2
# 14:   2 1 0 3
# 15:   3 1 0 2
# 16:   2 3 0 1
# 17:   3 2 0 1
# 18:   1 2 3 0
# 19:   1 3 2 0
# 20:   2 1 3 0
# 21:   3 1 2 0
# 22:   2 3 1 0
# 23:   3 2 1 0
```

The underlying idea is to find all possible paths that visit all nodes of a totally connected graph: start from each node and repeat the process on the remaining subgraph. The same array is used to mark nodes as not yet visited (-1) or to contain at which point in the path (0 for starting point $\ldots n-1$ for end point) it was visited. A recursive implementation looks like

```
int n;
int v[n];

int main()
{
    for (ulong k=0; k<n; ++k)  v[k] = -1;  // mark as not visited
    visit(0, 0);
```

```
    return 0;
}
void visit(int k, int j)
{
    int i;
    v[k] = j - 1;
    if ( j==n )
    {
        for (i=0; i<n; i++) printf ("%2d", v[i]);
        printf ("\n");
    }
    else
    {
        for (i=0; i<n; i++)
        {
            if ( -1 == v[i] )  visit(i, j+1);
        }
    }
    v[k] = -1;
}
```

The utility class [FXT: `class perm_visit` in `perm/permvisit.h`] is an iterative version of the algorithm that uses the `funcemu` mechanism (cf. section 11.5).

The above list can be created via

```
    ulong n = 4;
    perm_visit perm(n);
    const ulong *x = perm.data();
    do
    {
        cout << " #"; cout.width(3); cout << perm.current() << ":   ";
        for (ulong i=0; i<n; ++i)  cout << x[i] << " ";
        cout << endl;
    }
    while ( perm.next() );
```

Chapter 8

Some bit wizardry

In this chapter low-level functions are presented that operate on the bits of a given input word. It is often not obvious what these are good for and I do not attempt much to motivate why particular functions are here. However, *if* you happen to have a use for a given routine you will love that it is there: The program using it may run significantly faster.

Throughout this chapter it is assumed that `BITS_PER_LONG` (and `BYTES_PER_LONG`) reflect the size of the type `unsigned long` which usually is 32 (and 4) on 32 bit architectures, 64 (and 8) on 64 bit machines. [FXT: file `auxbit/bitsperlong.h`]

Further the type `unsigned long` is abbreviated as `ulong`. [FXT: file `include/fxttypes.h`]

The examples of assembler code are generally for the x86-architecture. They should be simple enough to be understood also by readers that only know the assembler-mnomics of other CPUs. The listings were generated from C-code using gcc's feature described on page 38.

TBD: *scaled int arith*

TBD: *int sincos*

TBD: *CORDIC?*

TBD: *modulo const by mult/shift*

TBD: *float-int conversion*

8.1 Trivia

With twos complement arithmetic (that is: on likely every computer you'll ever touch) division and multiplication by powers of two is right and left shift, respectively. This is true for unsigned types and for multiplication (left shift) with signed types. Division with signed types rounds toward zero, as one would expect, but right shift is a division (by a power of two) that rounds to minus infinity:

```
int a = -1;
int s = a >> 1;   // c == -1
int d = a / 2;    // d ==  0
```

The compiler still uses a shift instruction for the division, but a 'fix' for negative values:

```
  9:test.cc   @ int foo(int a)
 10:test.cc   @ {
285 0003 8B442410              movl 16(%esp),%eax
 11:test.cc   @     int s = a >> 1;
289 0007 89C1                  movl %eax,%ecx
290 0009 D1F9                  sarl $1,%ecx
 12:test.cc   @     int d = a / 2;
293 000b 89C2                  movl %eax,%edx
```

```
 294 000d C1EA1F                shrl $31,%edx  // fix: %edx=(%edx<0?1:0)
 295 0010 01D0                  addl %edx,%eax // fix: add one if a<0
 296 0012 D1F8                  sarl $1,%eax
```

For unsigned types the shift would suffice. One more reason to use unsigned types whenever possible.

There are two types of *right* shifts: a so called logical and an arithmetical shift. The logical version (`shrl` in the above fragment) always fills the higher bits with zeros, corresponding to division[1] of unsigned types. The arithmetical shift (`sarl` in the above fragment) fills in ones or zeros, according to the most significant bit of the original word. C uses the arithmetical or logical shift according to the operand types: This is used in

```
static inline long min0(long x)
// return min(0, x), i.e. return zero for positive input
// no restriction on input range
{
    return  x & (x >> (BITS_PER_LONG-1));
}
```

The trick is that the expression to the right of the "`&`" is 0 or 111...11 for positive or negative `x`, respectively (i.e. arithmetical shift is used). With unsigned type the same expression would be 0 or 1 according to whether the leftmost bit of `x` is set.

Computing residues modulo a power of two with unsigned types is equivalent to a bit-and using a mask:

```
    ulong a = b % 32;  // == b & (32-1)
```

All of the above is done by the compiler's optimization wherever possible.

Division by constants can be replaced by multiplications and shift. The magic machinery inside the compiler does it for you:

```
   5:test.cc   @ ulong foo(ulong a)
   6:test.cc   @ {
   7:test.cc   @     ulong b = a / 10;
 290 0000 8B442404              movl 4(%esp),%eax
 291 0004 F7250000              mull .LC33   // == 0xcccccccd
 292 000a 89D0                  movl %edx,%eax
 293 000c C1E803                shrl $3,%eax
```

Sometimes a good reason to have separate code branches with explicit special values. Similar for modulo computations with a constant modulus:

```
   8:test.cc   @ ulong foo(ulong a)
   9:test.cc   @ {
  53 0000 8B4C2404              movl 4(%esp),%ecx
  10:test.cc   @     ulong b = a % 10000;
  57 0004 89C8                  movl %ecx,%eax
  58 0006 F7250000              mull .LC0   // == 0xd1b71759
  59 000c 89D0                  movl %edx,%eax
  60 000e C1E80D                shrl $13,%eax
  61 0011 69C01027              imull $10000,%eax,%eax
  62 0017 29C1                  subl %eax,%ecx
  63 0019 89C8                  movl %ecx,%eax
```

To test whether at least one of `a` and `b` equals zero use `if ( (a+b)==a )` . This works for signed and unsigned integers. Checking whether both are zero can be done via `if ( (a|b)==0 )` . This obviously generalizes for several variables as `if ( (a|b|c|..|z)==0 )`). Test whether exactly one of two variables is zero using `if ( ((a|b)!=0) && ((a+b)==a) )` .

In order to toggle an integer `x` between two values `a` and `b` do:

[1] So you can think of it as 'unsigned arithmetical' shift.

```
precalculate:  t  = a ^ b;
toggle:        x ^= t;   // a <--> b
```

the equivalent trick for floats is

```
precalculate:  t = a + b;
toggle:        x = t - x;
```

The following functions should be self explaining. In the spirit of the C language there is no check whether the indices used are out of bounds. That is, if any index is greater than `BITS_PER_LONG`, the result is undefined. Find these in [FXT: file `auxbit/bitcopy.h`].

```
inline ulong test_bit(ulong a, ulong i)
// Return whether bit[i] is set
{
    ulong b = 1UL << i;
    return  (a & b);
}

inline ulong set_bit(ulong a, ulong i)
// Return a with bit[i] set
{
    return  (a | (1UL << i));
}

inline ulong delete_bit(ulong a, ulong i)
// Return a with bit[i] set to zero
{
    return  (a & ~(1UL << i));
}

inline ulong change_bit(ulong a, ulong i)
// Return a with bit[i] changed
{
    return  (a ^ (1UL << i));
}

inline ulong copy_bit(ulong a, ulong isrc, ulong idst)
// copy bit at [isrc] to position [idst]
{
    ulong v = a & (1UL << isrc);
    ulong b = 1UL << idst;
    if ( 0==v )  a &= ~b;
    else         a |=  b;
    return  a;
}
```

If the function

```
inline ulong bit_swap_01(ulong a, ulong i, ulong j)
// Swap bits i and j of a
// Bits must have different values (!)
// (i.e. one is zero, the other one)
// i==j is allowed (a is unchanged then)
{
    return  a ^ ( (1UL<<i) ^ (1UL<<j) );
}
```

appears worthless to you then consider

```
inline ulong bit_swap(ulong a, ulong i, ulong j)
// Swap bits i and j of a
// i==j is allowed (a is unchanged then)
{
#if 1  // optimized:
    ulong x = (a >> i) ^ (a >> j);
    // something to do if bit[i]!=bit[j]:
```

```
    if ( 0!=(x&1) )  a = bit_swap_01(a, i, j);
    return  a;
#else  // non-optimized version:
    ulong bi = 1UL << i;
    ulong bj = 1UL << j;
    ulong m = ~0UL;
    m ^= bi;
    m ^= bj;  // use xor to make it work for i==j
    ulong t = a & m;  // delete bits from a
    if ( a & bi )  t |= bj;
    if ( a & bj )  t |= bi;
    return  t;
#endif
}
```

Never ever think that some code is the 'fastest possible', there always another trick that can still improve it. Many factors can have an influence on performance like number of CPU registers or cost of branches. Code that performs well on one machine might perform badly on another. The old trick to swap variables without using a temporary

```
            //   a=0, b=0   a=0, b=1   a=1, b=0   a=1, b=1
    a ^= b; //     0    0     1    1     1    0     0    1
    b ^= a; //     0    0     1    0     1    1     0    1
    a ^= b; //     0    0     1    0     0    1     1    1
    equivalent to:
    tmp = a;  a = b;  b = tmp;
```

is pretty much out of fashion today. However in some specific context (like extreme register pressure) it may be the way to go.

8.2 Operations on low bits/blocks in a word

The following functions are taken from [FXT: file `auxbit/bitlow.h`].

The underlying idea is that addition/subtraction of 1 always changes a burst of bits at the lower end of the word.
Isolation of the lowest set bit is achieved via

```
static inline ulong lowest_bit(ulong x)
// return word where only the lowest set bit in x is set
// return 0 if no bit is set
{
    return  x & -x;  // use: -x == ~x + 1
}
```

The lowest zero (or unset bit) of some word `x` is then trivially isolated using `lowest_bit( ~x )`. [FXT: `lowest_zero` in `auxbit/bitlow.h`]

Unsetting the lowest set bit in a word can be achieved via

```
static inline ulong delete_lowest_bit(ulong x)
// return word were the lowest bit set in x is unset
// returns 0 for input == 0
{
    return  x & (x-1);
}
```

while setting the lowest unset bit is done by

```
static inline ulong set_lowest_zero(ulong x)
// return word were the lowest unset bit in x is set
// returns ~0 for input == ~0
{
    return  x | (x+1);
}
```

Isolate the burst of low bits/zeros as follows:

```
static inline ulong low_bits(ulong x)
// return word where all the (low end) ones
// are set
// e.g.  01011011 --> 00000011
// returns 0 if lowest bit is zero:
//       10110110 --> 0
{
    if ( ~0UL==x )  return ~0UL;

    return (((x+1)^x) >> 1);
}
```

and

```
static inline ulong low_zeros(ulong x)
// return word where all the (low end) zeros
// are set
// e.g.  01011000 --> 00000111
// returns 0 if all bits are set
{
    if ( 0==x )  return ~0UL;

    return (((x-1)^x) >> 1);
}
```

Isolation of the lowest block of ones (which may have zeros to the right of it) can be achieved via:

```
static inline ulong lowest_block(ulong x)
//
// x   = *****011100
// l   = 00000000100
// y   = *****100000
// x^y = 00000111100
// ret = 00000011100
//
{
    ulong l = x & -x;  // lowest bit
    ulong y = x + l;
    x ^= y;
    return  x & (x>>1);
}
```

Extracting the *index* of the lowest bit is easy when the corresponding assembler instruction is used:

```
static inline ulong asm_bsf(ulong x)
// Bit Scan Forward
{
    asm ("bsfl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

The given example uses gcc's wonderful feature of *Assembler Instructions with C Expression Operands*, see the corresponding info page.

Without the assembler instruction an algorithm that uses proportional $\log_2$(`BITS_PER_LONG`) can be used, so the resulting function may look like[2]

```
static inline ulong lowest_bit_idx(ulong x)
// return index of lowest bit set
// return 0 if no bit is set
{
#if defined  BITS_USE_ASM
    return  asm_bsf(x);
#else // BITS_USE_ASM
//    if ( 1>=x )  return  x-1; // 0 if 1, ~0 if 0
//    if ( 0==x )  return  0;
    ulong r = 0;
```

[2]thanks go to Nathan Bullock for emailing this improved (wrt. non-assembler `highest_bit_idx()`) version.

```
    x &= -x;
#if  BITS_PER_LONG >= 64
    if ( x & 0xffffffff00000000 )  r += 32;
    if ( x & 0xffff0000ffff0000 )  r += 16;
    if ( x & 0xff00ff00ff00ff00 )  r += 8;
    if ( x & 0xf0f0f0f0f0f0f0f0 )  r += 4;
    if ( x & 0xcccccccccccccccc )  r += 2;
    if ( x & 0xaaaaaaaaaaaaaaaa )  r += 1;
#else // BITS_PER_LONG >= 64
    if ( x & 0xffff0000 )  r += 16;
    if ( x & 0xff00ff00 )  r += 8;
    if ( x & 0xf0f0f0f0 )  r += 4;
    if ( x & 0xcccccccc )  r += 2;
    if ( x & 0xaaaaaaaa )  r += 1;
#endif // BITS_PER_LONG >= 64

#endif // BITS_USE_ASM
    return r;
}
```

Occasionally one wants to set a rising or falling edge at the position of the lowest bit:

```
static inline ulong lowest_bit_01edge(ulong x)
// return word where a all bits from  (including) the
//   lowest set bit to bit 0 are set
// return 0 if no bit is set
{
    if ( 0==x )  return 0;
    return  x^(x-1);
}

static inline ulong lowest_bit_10edge(ulong x)
// return word where a all bits from (including) the
//   lowest set bit to most significant bit are set
// return 0 if no bit is set
{
    if ( 0==x )  return 0;
    x ^= (x-1);
    // here  x == lowest_bit_01edge(x);
    return  ~(x>>1);
}
```

8.3 Operations on high bits/blocks in a word

The following functions are taken from [FXT: file `auxbit/bithigh.h`].

For the functions operating on the highest bit there is not a way as trivial as with the equivalent task with the lower end of the word. With a bit-reverse CPU-instruction available life would be significantly easier. However, almost no CPU seems to have it.

Isolation of the highest set bit is achieved via the bit-scan instruction when it is available

```
static inline ulong asm_bsr(ulong x)
// Bit Scan Reverse
{
    asm ("bsrl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

else one may use

```
static inline ulong highest_bit_01edge(ulong x)
// return word where a all bits from (including) the
//   highest set bit to bit 0 are set
// returns 0 if no bit is set
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
```

```
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  x;
}
```

so the resulting code may look like

```
static inline ulong highest_bit(ulong x)
// return word where only the highest bit in x is set
// return 0 if no bit is set
{
#if defined  BITS_USE_ASM
    if ( 0==x )  return 0;
    x = asm_bsr(x);
    return  1UL<<x;
#else
    x = highest_bit_01edge(x);
    return  x ^ (x>>1);
#endif // BITS_USE_ASM
}
```

trivially

```
static inline ulong highest_zero(ulong x)
// return word where only the highest unset bit in x is set
// return 0 if all bits are set
{
    return  highest_bit( ~x );
}
```

and

```
static inline ulong set_highest_zero(ulong x)
// return word were the highest unset bit in x is set
// returns ~0 for input == ~0
{
    return  x | highest_bit( ~x );
}
```

Finding the index of the highest set bit uses the equivalent algorithm as with the lowest set bit:

```
static inline ulong highest_bit_idx(ulong x)
// return index of highest bit set
// return 0 if no bit is set
{
#if defined  BITS_USE_ASM
    return  asm_bsr(x);
#else // BITS_USE_ASM
    if ( 0==x )  return  0;
    ulong r = 0;
#if  BITS_PER_LONG >= 64
    if ( x & (~0UL<<32) )  { x >>= 32;  r += 32; }
#endif
    if ( x & 0xffff0000 )  { x >>= 16;  r += 16; }
    if ( x & 0x0000ff00 )  { x >>=  8;  r +=  8; }
    if ( x & 0x000000f0 )  { x >>=  4;  r +=  4; }
    if ( x & 0x0000000c )  { x >>=  2;  r +=  2; }
    if ( x & 0x00000002 )  {            r +=  1; }
    return r;
#endif // BITS_USE_ASM
}
```

Isolation of the high zeros goes like

```
static inline ulong high_zeros(ulong x)
// return word where all the (high end) zeros are set
// e.g.  11001000 --> 00000111
// returns 0 if all bits are set
{
```

```
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  ~x;
}
```

The high bits could be isolated using arithmetical right shift

```
static inline ulong high_bits(ulong x)
// return word where all the (high end) ones are set
// e.g.  11001011 --> 11000000
// returns 0 if highest bit is zero:
//       01110110 --> 0
{
    long y = (long)x;
    y &= y>>1;
    y &= y>>2;
    y &= y>>4;
    y &= y>>8;
    y &= y>>16;
#if  BITS_PER_LONG >= 64
    y &= y>>32;
#endif
    return  (ulong)y;
}
```

However, arithmetical shifts may not be cheap, so we better use

```
static inline ulong high_bits(ulong x)
{
    return  high_zeros( ~x );
}
```

Demonstration of selected functions with two different input words:

```
----------------------------------------------------------------
................1111....1111.111 = 0xf0f7  == word
................1............... = highest_bit
................1111111111111111 = highest_bit_01edge
1111111111111111............... = highest_bit_10edge
                              15 = highest_bit_idx
................................ = low_zeros
...............................111 = low_bits
................................1 = lowest_bit
................................1 = lowest_bit_01edge
11111111111111111111111111111111 = lowest_bit_10edge
                               0 = lowest_bit_idx
...............................111 = lowest_block
................1111....1111.11. = delete_lowest_bit
................................1... = lowest_zero
................1111....11111111 = set_lowest_zero
................................ = high_bits
1111111111111111................ = high_zeros
1............................... = highest_zero
1...............1111....1111.111 = set_highest_zero
----------------------------------------------------------------
1111111111111111....1111....1... = 0xffff0f08  == word
1............................... = highest_bit
11111111111111111111111111111111 = highest_bit_01edge
1............................... = highest_bit_10edge
                             31 = highest_bit_idx
.............................111 = low_zeros
................................ = low_bits
............................1... = lowest_bit
.............................1111 = lowest_bit_01edge
11111111111111111111111111111... = lowest_bit_10edge
                              3 = lowest_bit_idx
............................1... = lowest_block
1111111111111111....1111........ = delete_lowest_bit
.............................1 = lowest_zero
```

```
1111111111111111....1111....1..1 = set_lowest_zero
1111111111111111................ = high_bits
................................ = high_zeros
................1............... = highest_zero
1111111111111111...1111....1... = set_highest_zero
----------------------------------------------------------
```

8.4 Functions related to the base-2 logarithm

The following functions are taken from [FXT: file `auxbit/bit2pow.h`].

The function `ld` that shall return $\lfloor \log_2(x) \rfloor$ can be implemented using the obvious algorithm:

```
static inline ulong ld(ulong x)
// returns k so that 2^k <= x < 2^(k+1)
// if x==0 then 0 is returned (!)
{
    ulong k = 0;
    while ( x>>=1 )  { ++k; }
    return k;
}
```

And then `ld` is the same as `highest_bit_idx`, so one can use

```
static inline ulong ld(ulong x)
{
    return  highest_bit_idx(x);
}
```

Closely related are the functions

```
static inline int is_pow_of_2(ulong x)
// return 1 if x == 0(!) or x == 2**k
{
    return  ((x & -x) == x);
}
```

and

```
static inline int one_bit_q(ulong x)
// return 1 iff x \in {1,2,4,8,16,...}
{
    ulong m = x-1;
    return  (((x^m)>>1) == m);
}
```

Occasionally useful in FFT based computations (where the length of the available FFTs is often restricted to powers of two) are

```
static inline ulong next_pow_of_2(ulong x)
// return x if x=2**k
// else return 2**ceil(log_2(x))
{
    ulong n = 1UL<<ld(x);  // n<=x
    if ( n==x )  return  x;
    else         return  n<<1;
}
```

and

```
static inline ulong next_exp_of_2(ulong x)
// return k if x=2**k
// else return k+1
{
    ulong ldx = ld(x);
    ulong n = 1UL<<ldx;  // n<=x
    if ( n==x )  return  ldx;
    else         return  ldx+1;
}
```

8.5 Counting the bits in a word

The following functions are from [FXT: file `auxbit/bitcount.h`].

If your CPU does not have a bit count instruction (sometimes called 'population count') then you might use an algorithm of the following type

```
static inline ulong bit_count(ulong x)
// return number of bits set
{
#if  BITS_PER_LONG == 32
    x = (0x55555555 & x) + (0x55555555 & (x>> 1));  // 0-2 in 2 bits
    x = (0x33333333 & x) + (0x33333333 & (x>> 2));  // 0-4 in 4 bits
    x = (0x0f0f0f0f & x) + (0x0f0f0f0f & (x>> 4));  // 0-8 in 8 bits
    x = (0x00ff00ff & x) + (0x00ff00ff & (x>> 8));  // 0-16 in 16 bits
    x = (0x0000ffff & x) + (0x0000ffff & (x>>16));  // 0-31 in 32 bits
    return x;
}
```

which can be improved to either

```
    x  = ((x>>1) & 0x55555555) + (x & 0x55555555);  // 0-2 in 2 bits
    x  = ((x>>2) & 0x33333333) + (x & 0x33333333);  // 0-4 in 4 bits
    x  = ((x>>4) + x) & 0x0f0f0f0f;                 // 0-8 in 4 bits
    x +=  x>> 8;                                    // 0-16 in 8 bits
    x +=  x>>16;                                    // 0-32 in 8 bits
    return  x & 0xff;
```

or

```
    x -=  (x>>1) & 0x55555555;
    x  = ((x>>2) & 0x33333333) + (x & 0x33333333);
    x  = ((x>>4) + x) & 0x0f0f0f0f;
    x *= 0x01010101;
    return  x>>24;
```

(From [54].) Which one is better mainly depends on the speed of integer multiplication.

For 64 bit CPUs the masks have to be adapted and one more step must be added (example corresponding to the second variant above):

```
    x = ((x>>1) & 0x5555555555555555) + (x & 0x5555555555555555);  // 0-2 in 2 bits
    x = ((x>>2) & 0x3333333333333333) + (x & 0x3333333333333333);  // 0-4 in 4 bits
    x = ((x>>4) + x) & 0x0f0f0f0f0f0f0f0f;                         // 0-8 in 4 bits
    x +=  x>> 8;                                                   // 0-16 in 8 bits
    x +=  x>>16;                                                   // 0-32 in 8 bits
    x +=  x>>32;                                                   // 0-64 in 8 bits
    return  x & 0xff;
```

When the word is known to have only a few bits set the following sparse count variant may be advantageous

```
static inline ulong bit_count_sparse(ulong x)
// return number of bits set
// the loop will execute once for each bit of x set
{
    if ( 0==x )  return 0;
    ulong n = 0;
    do  { ++n; }  while ( x &= (x-1) );
    return  n;
}
```

More esoteric counting algorithms are

```
static inline ulong bit_block_count(ulong x)
// return number of bit blocks
// e.g.:
// ..1..11111...111.  -> 3
// ...1..11111...111  -> 3
// ......1.....1.1..  -> 3
// .........111.1111  -> 2
{
    return  bit_count( (x^(x>>1)) ) / 2 + (x & 1);
}
```

```
static inline ulong bit_block_ge2_count(ulong x)
// return number of bit blocks with at least 2 bits
// e.g.:
// ..1..11111...111.  -> 2
// ...1..11111...111  -> 2
// ......1.....1.1..  -> 0
// .........111.1111  -> 2
{
    return  bit_block_count( x & ( (x<<1) & (x>>1) ) );
}
```

The slightly weird algorithm

```
static inline ulong bit_count_01(ulong x)
// return number of bits in a word
// for words of the special form 00...0001...11
{
    ulong ct = 0;
    ulong a;
#if  BITS_PER_LONG == 64
    a = (x & (1UL<<32)) >> (32-5);  // test bit 32
    x >>= a;  ct += a;
#endif
    a = (x & (1<<16)) >> (16-4);  // test bit 16
    x >>= a;  ct += a;

    a = (x & (1<<8)) >> (8-3);  // test bit 8
    x >>= a;  ct += a;

    a = (x & (1<<4)) >> (4-2);  // test bit 4
    x >>= a;  ct += a;

    a = (x & (1<<2)) >> (2-1);  // test bit 2
    x >>= a;  ct += a;

    a = (x & (1<<1)) >> (1-0);  // test bit 1
    x >>= a;  ct += a;

    ct += x & 1; // test bit 0

    return ct;
}
```

avoids all branches and may prove to be useful on a planet with pink air.

8.6 Swapping bits/blocks of a word

Functions in this section are from [FXT: file `auxbit/bitswap.h`]

Pairs of adjacent bits may be swapped via

```
static inline ulong bit_swap_1(ulong x)
// return x with neighbour bits swapped
{
#if  BITS_PER_LONG == 32
    ulong m = 0x55555555;
#else
#if  BITS_PER_LONG == 64
    ulong m = 0x5555555555555555;
#endif
#endif
    return  ((x & m) << 1) | ((x & (~m)) >> 1);
}
```

(the 64 bit branch is omitted in the following examples).

Groups of 2 bits are swapped by

```
static inline ulong bit_swap_2(ulong x)
// return x with groups of 2 bits swapped
{
    ulong m = 0x33333333;
    return  ((x & m) << 2) | ((x & (~m)) >> 2);
}
```

Equivalently,

```
static inline ulong bit_swap_4(ulong x)
// return x with groups of 4 bits swapped
{
    ulong m = 0x0f0f0f0f;
    return  ((x & m) << 4) | ((x & (~m)) >> 4);
}
```

and

```
static inline ulong bit_swap_8(ulong x)
// return x with groups of 8 bits swapped
{
    ulong m = 0x00ff00ff;
    return  ((x & m) << 8) | ((x & (~m)) >> 8);
}
```

When swapping half-words (here for32bit architectures)

```
static inline ulong bit_swap_16(ulong x)
// return x with groups of 16 bits swapped
{
    ulong m = 0x0000ffff;
    return  ((x & m) << 16) | ((x & (m<<16)) >> 16);
}
```

gcc is clever enough to recognize that the whole thing is equivalent to a (left or right) word rotation and indeed emits just a single rotate instruction.

The masks used in the above examples (and in many similar algorithms) can be replaced by arithmetic expressions that render the preprocessor statements unnecessary. However, the code does not necessarily gain readability by doing so.

Swapping two selected bits of a word goes like

```
static inline void bit_swap(ulong &x, ulong k1, ulong k2)
// swap bits k1 and k2
// ok even if  k1 == k2
{
    ulong b1 = x & (1UL<<k1);
    ulong b2 = x & (1UL<<k2);
    x ^= (b1 ^ b2);
    x ^= (b1>>k1)<<k2;
    x ^= (b2>>k2)<<k1;
}
```

8.7 Reversing the bits of a word

. . . when there is no corresponding CPU instruction can be achieved via the functions just described, cf. [FXT: file `auxbit/revbin.h`]

Shown is a 32 bit version of `revbin`:

```
static inline ulong revbin(ulong x)
// return x with bitsequence reversed
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
#if defined  BITS_USE_ASM
    x = asm_bswap(x);
#else
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#endif
    return x;
}
```

Here, the last two steps that correspond to a byte-reverse are replaced by the CPU instruction if available. For 64 bit machines a `x = bit_swap_32(x);` would have to be inserted at the end (and possibly a `bswap`-branch entered that can replace the last three `bit_swaps`).

One can generate the masks used in the process:

```
static inline ulong revbin(ulong x)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        x = ( (x & m) << s ) ^ ( (x & (~m)) >> s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  x;
}
```

Note that the above function is pretty expensive and it is not even clear whether it beats the obvious algorithm,

```
static inline ulong revbin(ulong x)
{
    ulong r = 0,  ldn = BITS_PER_LONG;
    while ( ldn-- != 0 )
    {
        r <<= 1;
        r += (x&1);
        x >>= 1;
    }
    return  r;
}
```

especially on 32 bit machines.

Therefore the function

```
static inline ulong revbin(ulong x, ulong ldn)
// return word with the last ldn bits
//   (i.e. bit_0 ... bit_{ldn-1})
//   of x reversed
// the other bits are set to 0
{
    return  revbin(x) >> (BITS_PER_LONG-ldn);
}
```

should only be used when `ldn` is not too small, else replaced by the trivial algorithm.

For practical computations the bit-reversed words usually have to be generated in the (reversed) counting order and there is a significantly cheaper way to do the update:

```
static inline ulong revbin_update(ulong r, ulong ldn)
// let r = revbin(x, ld(n)) at entry
// then return  revbin(x+1, ld(n))
{
    ldn >>= 1;
    while ( !((r^=ldn)&ldn) )  ldn >>= 1;
    return  r;
}
```

8.8 Generating bit combinations

The following functions are taken from [FXT: file `auxbit/bitcombcolex.h`] and [FXT: file `auxbit/bitcomblex.h`]

The ideas above can be used for the generation of bit combinations in colex order:

```
static inline ulong next_colex_comb(ulong x)
```

```
// return smallest integer greater than x with the same number of bits set.
//
// colex order: (5,3);
//  0  1  2   ..111
//  0  1  3   .1.11
//  0  2  3   .11.1
//  1  2  3   .111.
//  0  1  4   1..11
//  0  2  4   1.1.1
//  1  2  4   1.11.
//  0  3  4   11..1
//  1  3  4   11.1.
//  2  3  4   111..
//
//  Examples:
//    000001 -> 000010 -> 000100 -> 001000 -> 010000 -> 100000
//    000011 -> 000101 -> 000110 -> 001001 -> 001010 -> 001100 -> 010001 -> ...
//    000111 -> 001011 -> 001101 -> 001110 -> 010011 -> 010101 -> 010110 -> ...
//
//  Special cases:
//    0 -> 0
//    all bits on the high side (i.e. last combination) -> 0
//
{
    ulong r = x & -x; // lowest set bit
    x += r;           // replace lowest block by a one left to it

    if ( 0==l )  return 0;  // input was last comb

    ulong l = x & -x; // first zero beyond low block
    l -= r;           // low block

    while ( 0==(l&1) )  { l >>= 1; }  // move block to low end of word
    return  x | (l>>1);  // need one bit less of low block
}
```

One might consider replacing the while-loop by a bit scan and shift combination.
Moving backwards goes like

```
static inline ulong prev_colex_comb(ulong x)
// inverse of next_colex_comb()
{
    x = next_colex_comb( ~x);
    if ( 0!=x )  x = ~x;
    return  x;
}
```

The relation to lex order enumeration is

```
static inline ulong next_lex_comb(ulong x)
//
// let the zeros move to the lower end in the same manner
// as the ones go to the higher end in next_colex_comb()
//
// lex order:  (5, 3):
//  0  1  2   ..111
//  0  1  3   .1.11
//  0  1  4   1..11
//  0  2  3   .11.1
//  0  2  4   1.1.1
//  0  3  4   11..1
//  1  2  3   .111.
//  1  2  4   1.11.
//  1  3  4   11.1.
//  2  3  4   111..
//
// start and end combo are the same as for next_colex_comb()
//
{
    x = revbin(~x);
    x = next_colex_comb(x);
    if ( 0!=x )  x = revbin(~x);
    return  x;
}
```

(the bit-reversal routine `revbin` is shown in section 8.7) and

```
static inline ulong prev_lex_comb(ulong x)
// inverse of next_lex_comb()
{
    x = revbin(x);
    x = next_colex_comb(x);
    x = revbin(x);
    return  x;
}
```

Note that the ones in lex-order(k, n) behave like the zeros in reversed colex-order(n-k, n):

```
   Lex( n = 5,  k = 3 )            Colex( n = 5,  k = 2 )
 forward order:                  reverse order:
   [  0  1  2 ]  ..111  #  0       [  3  4 ]  11...  #  9
   [  0  1  3 ]  .1.11  #  1       [  2  4 ]  1.1..  #  8
   [  0  1  4 ]  1..11  #  2       [  1  4 ]  1..1.  #  7
   [  0  2  3 ]  .11.1  #  3       [  0  4 ]  1...1  #  6
   [  0  2  4 ]  1.1.1  #  4       [  2  3 ]  .11..  #  5
   [  0  3  4 ]  11..1  #  5       [  1  3 ]  .1.1.  #  4
   [  1  2  3 ]  .111.  #  6       [  0  3 ]  .1..1  #  3
   [  1  2  4 ]  1.11.  #  7       [  1  2 ]  ..11.  #  2
   [  1  3  4 ]  11.1.  #  8       [  0  2 ]  ..1.1  #  1
   [  2  3  4 ]  111..  #  9       [  0  1 ]  ...11  #  0
 reverse order:                  forward order:
   [  2  3  4 ]  111..  #  9       [  0  1 ]  ...11  #  0
   [  1  3  4 ]  11.1.  #  8       [  0  2 ]  ..1.1  #  1
   [  1  2  4 ]  1.11.  #  7       [  1  2 ]  ..11.  #  2
   [  1  2  3 ]  .111.  #  6       [  0  3 ]  .1..1  #  3
   [  0  3  4 ]  11..1  #  5       [  1  3 ]  .1.1.  #  4
   [  0  2  4 ]  1.1.1  #  4       [  2  3 ]  .11..  #  5
   [  0  2  3 ]  .11.1  #  3       [  0  4 ]  1...1  #  6
   [  0  1  4 ]  1..11  #  2       [  1  4 ]  1..1.  #  7
   [  0  1  3 ]  .1.11  #  1       [  2  4 ]  1.1..  #  8
   [  0  1  2 ]  ..111  #  0       [  3  4 ]  11...  #  9
```

The first and last combination for both colex- and lex order are

```
static inline ulong first_comb(ulong k)
// return the first combination of (i.e. smallest word with) k bits,
// i.e.  00..001111..1 (k low bits set)
// must have:  0 <= k <= BITS_PER_LONG
{
    return  ~0UL >> ( BITS_PER_LONG - k );
}
```

and

```
static inline ulong last_comb(ulong k, ulong n=BITS_PER_LONG)
// return the last combination of (biggest n-bit word with) k bits
// i.e.  1111..100..00 (k high bits set)
// must have:  0 <= k <= n <= BITS_PER_LONG
{
    return  first_comb(k) << (n - k);
}
```

Note that the colex-combinations in reversed order are identical to the bit-reversed lex-combinations, thereby:

```
static inline ulong first_rev_comb(ulong k, ulong n=BITS_PER_LONG)
{
    return  last_comb(k, n);
}
static inline ulong last_rev_comb(ulong k)
{
    return  first_comb(k);
}
static inline ulong next_lexrev_comb(ulong x)
// Return next bit-reversed lex-order combination.
{
    return  prev_colex_comb(x);
}
static inline ulong prev_lexrev_comb(ulong x)
```

```
// Return previous bit-reversed lex-order combination.
{
    return  next_colex_comb( x );
}
```

These are often the functions of choice when fast generation of bit-combinations is required as they allow to restrict the pattern to a certain length without any overhead.

A variant of the presented (colex-) algorithm appears in hakmem [53]. The variant used here avoids the division of the hakmem-version and is given at `http://www.caam.rice.edu/~dougm/` by Doug Moore and Glenn Rhoads `http://remus.rutgers.edu/~rhoads/` (cited in the code is "Constructive Combinatorics" by Stanton and White).

8.9 Generating bit subsets

The sparse counting idea shown on page 164 is used in

```
class bit_subset
// generate all all subsets of bits of  a given word
//
// e.g. for the word ('.' printed for unset bits)
//   ...11.1.
// these words are produced by subsequent next()-calls:
//   ......1.
//   ....1...
//   ....1.1.
//   ...1....
//   ...1..1.
//   ...11...
//   ...11.1.
//   ........
//
{
public:
    ulong u_, v_;
public:
    bit_subset(ulong vv) : u_(0), v_(vv) { ; }
    ~bit_subset() { ; }
    ulong current()  const { return u_; }
    ulong next()      { u_ = (u_ - v_) & v_;  return u_; }
    ulong previous()  { u_ = (u_ - 1 ) & v_;  return u_; }
};
```

which can be found in [FXT: file `auxbit/bitsubset.h`]

TBD: *sparse count in Gray code order*

8.10 Binary words in lexicographic order

The (bit-reversed) binary words in lexicographic order are generated by successive calls to the following function:

```
static inline ulong next_lexrev(ulong x)
// Return next word in (reversed) lex order.
{
    ulong x0 = x & -x;
    if ( 1==x0 )
    {
        x ^= x0;
        x0 = x & -x;
        x0 ^= (x0>>1);
    }
    else  x0 >>= 1;
    x ^= x0;
    return  x;
}
```

The bit-reversed representation was chosen because the isolation of the lowest bit is often cheaper than the same operation on the highest bit. The routine was derived from the code in section 11.10. Starting with a one-bit word at position $n-1$ one generates the 2^n bit-subsets of length n:

```
 n==4:
  1...
  11..
  111.
  1111
  11.1
  1.1.
  1.11
  1..1
  .1..
  .11.
  .111
  .1.1
  ..1.
  ..11
  ...1
```

A similar function allows to go backward:

```
static inline ulong prev_lexrev(ulong x)
// Return previous word in (reversed) lex order.
{
    ulong x0 = x & -x;
    if ( x & (x0<<1) )  x ^= x0;
    else
    {
        x0 ^= (x0<<1);
        x ^= x0;
        x |= 1;
    }
    return x;
}
```

Starting with zero one gets a sequence of words that just before the 2^n-th call has visited every word of length $\le n$.

```
 n==4:
 0:  ....  =  0 *
 1:  ...1  =  1 *
 2:  ..11  =  3
 3:  ..1.  =  2
 4:  .1.1  =  5
 5:  .111  =  7
 6:  .11.  =  6 *
 7:  .1..  =  4
 8:  1..1  =  9
 9:  1.11  = 11
10:  1.1.  = 10 *
11:  11.1  = 13
12:  1111  = 15
13:  111.  = 14
14:  11..  = 12
15:  1...  =  8
```

The '*' mark fixed points of the sequence.

Find the described functions in [FXT: file `auxbit/bitlex.h`].

The sequence of fixed points

The fixed points of the generated sequence are 0, 1, 6, 10, 18, 34, 60, 66, 92, 108, 116, 130, 156, 172, 180, 204, 212, 228, 258, 284, 300, 308, 332, 340, 356, 396, 404, 420, 452, 514, 540, 556, ...

Their values as bit patterns:

```
    0:  ..........
    1:  .........1
    6:  ........11.
   10:  .......1.1.
   18:  ......1..1.
   34:  .....1...1.
   60:  ....1111..
   66:  ....1....1.
   92:  ....1.111..
  108:  ....11.11..
  116:  ....111.1..
  130:  ...1.....1.
  156:  ...1..111..
  172:  ...1.1.11..
```

```
  180:  ...1.11.1..
  204:  ...11..11..
  212:  ...11.1.1..
  228:  ...111..1..
  258:  ..1.....1.
  284:  ..1...111..
  300:  ..1..1.11..
  308:  ..1..11.1..
  332:  ..1.1..11..
  340:  ..1.1.1.1..
  356:  ..1.11..1..
  396:  ..11...11..
  404:  ..11..1.1..
  420:  ..11.1...1.
  452:  .1111...1..
  514:  .1.....1.
  540:  .1....111..
  556:  .1...1.11..
[-snip-]
 1556:  ..11...1.1..
 1572:  ..11..1..1..
 1604:  ..11.1...1..
 1668:  ..11.1....1..
 1796:  ..111.....1..
 2040:  ..11111111..
 2050:  .1.......1.
 2076:  .1......111..
 2092:  .1.....1.11..
 2100:  .1.....11.1..
 2124:  .1....1..11..
 2132:  .1....1.1.1..
 2148:  .1....11..1..
[-snip-]
 4644:  .1..1..1..1..
 4676:  .1..1..1...1..
 4740:  .1..1.1....1..
 4868:  .1..11.....1..
 5112:  .1..1111111..
 5132:  .1.1......11..
 5140:  .1.1.....1.1..
 5156:  .1.1...1..1..
 5188:  .1.1..1...1..
 5252:  .1.1..1....1..
 5380:  .1.1.1.....1..
 5624:  .1.1.11111..
 5636:  .1.11.......1..
 5880:  .1.11.1111...
 6008:  .1.111.1111...
 6072:  .1.1111.111...
 6104:  .1.11111.11...
 6120:  .1.111111.1...
 6156:  .11.......11..
 6164:  .11.....1.1..
 6180:  .11.....1..1..
```

Whether x is a fixed point of the sequence is detected by [FXT: `is_lexrev_fixed_point` in `auxbit/bitlex.h`]:

```
static inline bool is_lexrev_fixed_point(ulong x)
// Return whether x is a fixed point in the prev_lexrev() - sequence
{
    if ( x & 1 )
    {
        if ( 1==x )  return  true;
        else         return  false;
    }
    else
    {
        ulong w = bit_count(x);
        if ( w != (w & -w) )  return  false;
        if ( 0==x )  return  true;
        return  0 != ( (x & -x) & w );
    }
}
```

A little contemplation on the structure of the binary words in lex-order leeds to the routine

```
inline ulong negidx2lexrev(ulong k)
//
//   k:  idx2revlex(k)
//   0:  .....
//   1:  ....1
//   2:  ...11
//   3:  ...1.
//   4:  ..1.1
//   5:  ..111
//   6:  ..11.
//   7:  ..1..
//   8:  .1..1
```

```
//   9:  .1.11
//  10:  .1.1.
//  11:  .11.1
//  12:  .1111
//  13:  .111.
//  14:  .11..
//  15:  .1...
//  16:  1...1
//
{
    ulong z = 0;
    ulong h = highest_bit(k);
    while ( k )
    {
        while ( 0==(h&k) )  h >>= 1;
        z ^= h;
        ++k;
        k &= h - 1;
    }
    return  z;
}
```

8.11 Bit set lookup

There is a nice trick to determine whether some input is contained in a tiny set, e.g. lets determine whether `x` is a tiny prime

```
ulong m = (1UL<<2) | (1UL<<3) | (1UL<<5) | ... | (1UL<<31);  // precomputed
static inline ulong is_tiny_prime(ulong x)
{
    return  m & (1UL << x);
}
```

A function using this idea is

```
static inline bool is_tiny_factor(ulong x, ulong d)
// for x,d < BITS_PER_LONG (!)
// return whether d divides x  (1 and x included as divisors)
// no need to check whether d==0
//
{
    return  ( 0 != ( (tiny_factors_tab[x]>>d) & 1 ) );
}
```

from [FXT: file `auxbit/tinyfactors.h`] that uses the precomputed

```
extern const ulong tiny_factors_tab[] =
{
                   0x0,  // x = 0:           ( bits: ........)
                   0x2,  // x = 1:  1        ( bits: ......1.)
                   0x6,  // x = 2:  1 2      ( bits: .....11.)
                   0xa,  // x = 3:  1 3      ( bits: ....1.1.)
                  0x16,  // x = 4:  1 2 4    ( bits: ...1.11.)
                  0x22,  // x = 5:  1 5      ( bits: ..1...1.)
                  0x4e,  // x = 6:  1 2 3 6  ( bits: .1..111.)
                  0x82,  // x = 7:  1 7      ( bits: 1.....1.)
                 0x116,  // x = 8:  1 2 4 8
                 0x20a,  // x = 9:  1 3 9
...
            0x20000002,  // x = 29:  1 29
            0x4000846e,  // x = 30:  1 2 3 5 6 10 15 30
            0x80000002,  // x = 31:  1 31
#if  ( BITS_PER_LONG > 32 )
           0x100010116,  // x = 32:  1 2 4 8 16 32
           0x20000080a,  // x = 33:  1 3 11 33
...
    0x2000000000000002,  // x = 61:  1 61
    0x4000000080000006,  // x = 62:  1 2 31 62
    0x800000000020028a   // x = 63:  1 3 7 9 21 63
#endif // ( BITS_PER_LONG > 32 )
};
```

8.12 The Gray code of a word

Can easily be computed by

```
static inline ulong gray_code(ulong x)
// Return the Gray code of x
// ('bitwise derivative modulo 2')
{
    return  x ^ (x>>1);
}
```

The inverse is slightly more expensive. The straight forward idea is to use

```
static inline ulong inverse_gray_code(ulong x)
// inverse of gray_code()
{
    // VERSION 1 (integration modulo 2):
    ulong h=1, r=0;
    do
    {
        if ( x & 1 )  r^=h;
        x >>= 1;
        h = (h<<1)+1;
    }
    while ( x!=0 );
    return r;
}
```

which can be improved to

```
    // VERSION 2 (apply graycode BITS_PER_LONG-1 times):
    ulong r = BITS_PER_LONG;
    while ( --r )  x ^= x>>1;
    return x;
```

while the best way to do it is

```
    // VERSION 3 (use: gray ** BITSPERLONG == id):
    x ^= x>>1;  // gray ** 1
    x ^= x>>2;  // gray ** 2
    x ^= x>>4;  // gray ** 4
    x ^= x>>8;  // gray ** 8
    x ^= x>>16; // gray ** 16
    // here: x = gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x>>32;  // for 64bit words
#endif
    return  x;
```

Related to the inverse Gray code is the parity of a word (that is: bit-count modulo two). The inverse Gray code of a word contains at each bit position the parity of all bits of the input left from it (including itself).

```
static inline ulong parity(ulong x)
// return 1 if the number of set bits is even, else 0
{
    return  inverse_gray_code(x) & 1;
}
```

Be warned that the parity bit of many CPUs is the complement of the above. With the x86-architecture the parity bit also takes in account only the lowest byte, therefore:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addl  $0, %0  \n"
         "setnp %%al    \n"
```

```
        "movzx %%al, %0"
        : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

Cf. [FXT: file auxbit/bitasm.h]

The function

```
static inline ulong grs_negative_q(ulong x)
// Return whether the Golay-Rudin-Shapiro sequence
// (A020985) is negative for index x
// returns 1 for x =
// 3,6,11,12,13,15,19,22,24,25,26,30,35,38,43,44,45,47,48,49,
//  50,52,53,55,59,60,61,63,67,70,75,76,77,79,83,86,88,89,90,94,
//  96,97,98,100,101,103,104,105,106,110,115,118,120,121,122,
//  126,131,134,139,140, ...
//
// algorithm: count bit pairs modulo 2
//
{
    return  parity( x & (x>>1) );
}
```

proves to be useful in specialized versions of the fast Fourier- and Walsh transform.

A byte-wise Gray code can be computed using

```
static inline ulong byte_gray_code(ulong x)
// Return the Gray code of bytes in parallel
{
    return  x ^ ((x & 0xfefefefe)>>1);
}
```

Its inverse is

```
static inline ulong byte_inverse_gray_code(ulong x)
// Return the inverse Gray code of bytes in parallel
{
    x ^= ((x & 0xfefefefe)>>1);
    x ^= ((x & 0xfcfcfcfc)>>2);
    x ^= ((x & 0xf0f0f0f0)>>4);
    return  x;
}
```

Thereby

```
static inline ulong byte_parity(ulong x)
// Return the parities of bytes in parallel
{
    return  byte_inverse_gray_code(x) & 0x01010101;
}
```

The Gray code related functions can be found in [FXT: file auxbit/graycode.h].

Similar to the Gray code and its inverse is the

```
static inline ulong green_code(ulong x)
// Return the green code of x
// ('bitwise derivative modulo 2 towards high bits')
//
// green_code(x) == revbin(gray_code(revbin(x)))
{
    return  x ^ (x<<1);
}
```

and

```
static inline ulong inverse_green_code(ulong x)
// inverse of green_code()
```

```
// note: the returned value contains at each bit position
// the parity of all bits of the input right from it (incl. itself)
{
    // use: green ** BITSPERLONG == id:
    x ^= x<<1;  // green ** 1
    x ^= x<<2;  // green ** 2
    x ^= x<<4;  // green ** 4
    x ^= x<<8;  // green ** 8
    x ^= x<<16; // green ** 16
    // here: x = green**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x<<32;  // for 64bit words
#endif
    return  x;
}
```

Both can be found in [FXT: file `auxbit/greencode.h`] The green code preserves the lowest set bit while the Gray code preserves the highest.

Demonstration of Gray/green code and their inverses with different input words:

```
-------------------------------------------------------------
111.1111....1111.............  = 0xef0f0000  == word
1..11...1...1...1.............. = gray_code
..11...1...1...1............... = green_code
1.11.1.11111.1.1111111111111111 = inverse_gray_code
1.1..1.1.....1.1............... = inverse_green_code
-------------------------------------------------------------
...1....1111....1111111111111111  = 0x10f0ffff  == word
...11...1...1...1.............. = gray_code
..11...1...1...1.............1 = green_code
...11111.1.11111.1.1.1.1.1.1.1 = inverse_gray_code
1111.....1.1.....1.1.1.1.1.1.1 = inverse_green_code
-------------------------------------------------------------
......1.........................  = 0x2000000  == word
......11........................ = gray_code
.....11......................... = green_code
......11111111111111111111111111 = inverse_gray_code
1111111......................... = inverse_green_code
-------------------------------------------------------------
111111.11111111111111111111111  = 0xfdffffff  == word
1.....11........................ = gray_code
.....11.......................1 = green_code
1.1.1..1.1.1.1.1.1.1.1.1.1.1.1 = inverse_gray_code
1.1.1.11.1.1.1.1.1.1.1.1.1.1.1 = inverse_green_code
-------------------------------------------------------------
```

The Gray codes of consecutive values change in one bit. The Gray codes of the Gray codes of consecutive values change in one or two bits. The Gray codes of values that have a difference of two changes in two bit. Gray codes of even/odd values have an even/odd number of bits set, respectively. This is demonstrated in [FXT: file `demo/gray2-demo.cc`]:

```
   k:        g(k)     g(g(k))     g(2*k)   g(2*k+1)
   0:      ......    ......    ......1    ......1
   1:      .....1    .....1    .....11    .....1.
   2:      ....11    ....1.    ....11.    ....111
   3:      ....1.    ....11    ...1.1.    ...1...
   4:      ...11.    ...1.1    ...11..    ...11.1
   5:      ...111    ...1..    ...1111    ...111.
   6:      ...1.1    ...111    ...1.1.    ...1.11
   7:      ...1..    ...11.    ...1..1    ...1...
   8:      ..11..    ..1.1.    ..11...    ..11..1
   9:      ..11.1    ..1.11    ..11.11    ..11.1.
  10:      ..1111    ..1...    ..1111.    ..11111
  11:      ..111.    ..1..1    ..111.1    ..111..
  12:      ..1.1.    ..1111    ..1.1..    ..1.1.1
  13:      ..1.11    ..111.    ..1.111    ..1.11.
  14:      ..1..1    ..11.1    ..1..1.    ..1..11
  15:      .....    .....    .....1    .....
  16:      .11...    .1.1..    .11...    .11...1
  17:      .11..1    .1.1.1    .11..11    .11..1.
  18:      .11.11    .1.11.    .11.11.    .11.111
  19:      .11.1.    .1.111    .11.1.1    .11.1..
  20:      .1111.    .1...1    .1111..    .1111.1
  21:      .11111    .1....    .111111    .11111.
  22:      .111.1    .1..11    .111.1.    .111.11
  23:      .111..    .1..1.    .111..1    .111...
  24:      .1.1..    .1111.    .1.1...    .1.1..1
  25:      .1.1.1    .11111    .1.1.11    .1.1.1.
  26:      .1.111    .111..    .1.111.    .1.1111
  27:      .1.11.    .111.1    .1.11.1    .1.11..
```

```
 28:      ..1..1.    ..11.11    .1..1..    .1..1.1
 29:      ..1..11    ..11.1.    .1..111    .1..11.
 30:      ..1...1    ..11..1    .1...1.    .1...11
 31:      ..1....    ..11...    .1....1    .1.....
```

In order two produce a random value with with even/odd many bits set, set the lowest bit of a random number to zero/one and take the Gray code.

8.13 Generating minimal-change bit combinations

The wonderful

```
static inline ulong igc_next_minchange_comb(ulong x)
// Returns the inverse graycode of the next
// combination in minchange order.
// Input must be the inverse graycode of the
// current combination.
{
    ulong g = green_code(x);
    ulong i = 2;
    ulong cb; // ==candidateBits;
    do
    {
        ulong y = (x & ~(i-1)) + i;
        ulong j = lowest_bit(y) << 1;
        ulong h = !!(y & j);
        cb = ((j-h) ^ g) & (j-i);
        i = j;
    }
    while ( 0==cb );
    return  x + lowest_bit(cb);
}
```

together with

```
static inline ulong igc_last_comb(ulong k, ulong n)
// return the (inverse graycode of the) last combination
// as in igc_next_minchange_comb()
{
    if ( 0==k )  return 0;
    else  return  ((1UL<<n) - 1) ^ (((1UL<<k) - 1) / 3);
}
```

could be used as demonstrated in

```
static inline ulong next_minchange_comb(ulong x, ulong last)
// not efficient, just to explain the usage
// of igc_next_minchange_comb()
// Must have: last==igc_last_comb(k, n)
//
// Example with  k==3, n==5:
//      x       inverse_gray_code(x)
//    ..111       ..1.1 == first_sequency(k)
//    .11.1       .1..1
//    .111.       .1.11
//    .1.11       .11.1
//    11..1       1...1
//    11.1.       1..11
//    111..       1.111
//    1.1.1       11..1
//    1.11.       11.11
//    1..11       111.1 == igc_last_comb(k, n)
{
    x = inverse_gray_code(x);
    if ( x==last )  return 0;
    x = igc_next_minchange_comb(x);
    return  gray_code(x);
}
```

Each combination is different from the preceding one in exactly two positions. The same run of bit combinations could be obtained by going through the Gray codes and omitting all words where the bit-count is $\neq k$. The algorithm shown here, however, is much more efficient.

For reasons of efficiency one may prefer code as

```
    ulong last = igc_last_comb(k, n);
    ulong c, nc = first_sequency(k);
    do
    {
        c = nc;
        nc = igc_next_minchange_comb(c);
        ulong g = gray_code(c);
        // Here g contains the bitcombination
    }
    while ( c!=last );
```

which avoids the repeated computation of the inverse Gray code.

As Doug Moore explains [priv.comm.], the algorithm in `igc_next_minchange_comb` uses the fact that the difference of two (inverse gray codes of) successive combinations is always a power of two. Using this observation one can derive a different version that checks the pattern of the change:

```
static inline ulong igc_next_minchange_comb(ulong x)
// Alternative version.
// Amortized time = O(1).
{
    ulong gx = gray_code( x );
    ulong y, i = 2;
    do
    {
        y = x + i;
        ulong gy = gray_code( y );
        ulong r = gx ^ gy;
        // Check that change consists of exactly one bit
        // of the new and one bit of the old pattern:
        if ( is_pow_of_2( r & gy ) && is_pow_of_2( r & gx ) )  break;
        // is_pow_of_2(x):=((x & -x) == x)  returns 1 also for x==0.
        // But this cannot happen for both tests at the same time
        i <<= 1;
    }
    while ( 1 );
    return  y;
}
```

Still another version which needs `k`, the number of set bits, as a second parameter:

```
static inline ulong igc_next_minchange_comb(ulong x, ulong k)
// Alternative version, uses the fact that the difference
// of two successive x is the smallest possible power of 2.
// Should be fast if the CPU has a bitcount instruction.
// Amortized time = O(1).
{
    ulong y, i = 2;
    do
    {
        y = x + i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

The necessary modification for the generation of the previous combination is minimal:

```
static inline ulong igc_prev_minchange_comb(ulong x, ulong k)
// Returns the inverse graycode of the previous combination in minchange order.
// Input must be the inverse graycode of the current combination.
// Amortized time = O(1).
// With input==first the output is the last for n=BITS_PER_LONG
{
```

```
    ulong y, i = 2;
    do
    {
        y = x - i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

8.14 Bitwise rotation of a word

Neither C nor C++ have a statement for bitwise rotation[3]. The operations can be 'emulated' like this

```
static inline ulong bit_rotate_left(ulong x, ulong r)
// return word rotated r bits
// to the left (i.e. toward the most significant bit)
{
    return  (x<<r) | (x>>(BITS_PER_LONG-r));
}
```

As already mentioned, GCC emits exactly the one CPU instruction that is *meant* here, even with non-constant `r`. Well done, GCC folks!

Of course the explicit use of the corresponding assembler instruction cannot do any harm:

```
static inline ulong bit_rotate_right(ulong x, ulong r)
// return word rotated r bits
// to the right (i.e. toward the least significant bit)
//
// gcc 2.95.2 optimizes the function to asm 'rorl %cl,%ebx'
{
#if defined  BITS_USE_ASM    // use x86 asm code
    return asm_ror(x, r);
#else
    return  (x>>r) | (x<<(BITS_PER_LONG-r));
#endif
}
```

where (see [FXT: file `auxbit/bitasm.h`]):

```
static inline ulong asm_ror(ulong x, ulong r)
{
    asm ("rorl   %%cl, %0" : "=r" (x) : "0" (x), "c" (r));
    return x;
}
```

Rotations using only a part of the word length are achieved by

```
static inline ulong bit_rotate_left(ulong x, ulong r, ulong ldn)
// return ldn-bit word rotated r bits
// to the left (i.e. toward the most significant bit)
// must have  0 <= r <= ldn
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x<<r) | (x>>(ldn-r));
    x &= m;
    return  x;
}
```

and

```
static inline ulong bit_rotate_right(ulong x, ulong r, ulong ldn)
// return ldn-bit word rotated r bits
// to the right (i.e. toward the least significant bit)
// must have  0 <= r <= ldn
```

[3] which I consider a missing feature.

```
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x>>r) | (x<<(ldn-r));
    x &= m;
    return  x;
}
```

Finally, the functions

```
static inline ulong bit_rotate_sgn(ulong x, long r, ulong ldn)
// positive r --> shift away from element zero
{
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r, ldn);
    else          return  bit_rotate_right(x, (ulong)-r, ldn);
}
```

and

```
static inline ulong bit_rotate_sgn(ulong x, long r)
// positive r --> shift away from element zero
{
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r);
    else          return  bit_rotate_right(x, (ulong)-r);
}
```

are often convenient.

8.15 Functions related to bitwise rotation

Some functions related to bitwise rotation can be found in [FXT: file `auxbit/bitcyclic.h`]:

```
static inline ulong bit_cyclic_match(ulong x, ulong y)
// return  r if x==rotate_right(y, r)
// else return ~0UL
// in other words: returns, how often
//   the right arg must be rotated right (to match the left)
// or, equivalently: how often
//   the left arg must be rotated left (to match the right)
{
    ulong r = 0;
    do
    {
        if ( x==y )  return r;
        y = bit_rotate_right(y, 1);
    }
    while ( ++r < BITS_PER_LONG );
    return ~0UL;
}

static inline ulong bit_cyclic_min(ulong x)
// return  minimum of all rotations of x
{
    ulong r = 1;
    ulong m = x;
    do
    {
        x = bit_rotate_right(x, 1);
        if ( x<m )  m = x;
    }
    while ( ++r < BITS_PER_LONG );
    return  m;
}
```

Selecting from all n-bit words those that are equal to their cyclic minimum gives the sequence of the binary length-n necklaces, see section 12.6.

From [FXT: file `auxbit/bitcyclic2.h`]:

```
static inline ulong bit_cyclic_period(ulong x, ulong ldn)
// return minimal positive bit-rotation
//   that transforms x into itself.
//  (using ldn-bit words)
// Returned value is a divisor of ldn.
//
// Examples for ldn=6:
//  ......  1
//  .....1  6
//  ....11  6
//  ...1.1  6
//  ...111  6
//  ..1..1  3
//  ..1.11  6
//  ..11.1  6
//  ..1111  6
//  .1.1.1  2
//  .1.111  6
//  .11.11  3
//  .11111  6
//  111111  1
{
    ulong y = bit_rotate_right(x, 1, ldn);
    return  bit_cyclic_match(x, y, ldn) + 1;
}
```

The version for `ldn==BITS_PER_LONG` can be optimized:

```
static inline ulong bit_cyclic_period(ulong x)
// return minimal positive bit-rotation
//   that transforms x into itself.
// (same as bit_cyclic_period(x, BITS_PER_LONG) )
//
// Returned value is a divisor of the word length,
//   i.e. 1,2,4,8,...,BITS_PER_LONG.
{
    ulong r = 1;
    do
    {
        ulong y = bit_rotate_right(x, r);
        if ( x==y )  return r;
        r <<= 1;
    }
    while ( r < BITS_PER_LONG );
    return  r;  // == BITS_PER_LONG
}
```

An equivalent optimization is possible for arbitrary word length using the mechanism described in section 8.11.

```
inline ulong bit_cyclic_dist(ulong a, ulong b)
// Return minimal bitcount of (t ^ b)
// where t runs through the cyclic rotations.
{
    ulong d = ~0UL;
    ulong t = a;
    do
    {
        ulong z = t ^ b;
        ulong e = bit_count( z );
        if ( e < d )  d = e;
        t = bit_rotate_right(t, 1);
    }
    while ( t!=a );
    return  d;  // not reached
}
```

and a equivalent function `ulong bit_cyclic_dist(ulong a, ulong b, ulong ldn)` for length-`ldn` words.

8.16 Bitwise zip

The bitwise zip operation, when straight forward implemented, is

```
ulong bit_zip(ulong a, ulong b)
// put lower half bits to even indexes, higher half to odd
{
    ulong x = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        x |= (a & m) << s;
        ++s;
        x |= (b & m) << s;
        m <<= 1;
    }
    return  x;
}
```

Its inverse is

```
void bit_unzip(ulong x, ulong &a, ulong &b)
// put even indexed bits to lower hald, odd indexed to higher half
{
    a = 0;  b = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        a |= (x & m) >> s;
        ++s;
        m <<= 1;
        b |= (x & m) >> s;
        m <<= 1;
    }
}
```

The optimized versions (cf. [FXT: file `auxbit/bitzip.h`]), using ideas similar to those in `revbin` and `bit_count`, are

```
static inline ulong bit_zip(ulong x)
{
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    x = butterfly_8(x);
    x = butterfly_4(x);
    x = butterfly_2(x);
    x = butterfly_1(x);
    return  x;
}
```

and

```
static inline ulong bit_unzip(ulong x)
{
    x = butterfly_1(x);
    x = butterfly_2(x);
    x = butterfly_4(x);
    x = butterfly_8(x);
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    return  x;
}
```

Both use the `butterfly_*()`-functions which look like

```
static inline ulong butterfly_4(ulong x)
{
    ulong t, ml, mr, s;
#if  BITS_PER_LONG == 64
    ml = 0x0f000f000f000f00;
```

```
#else
    ml = 0x0f000f00;
#endif
    s = 4;
    mr = ml >> s;
    t = ((x & ml) >> s ) | ((x & mr) << s );
    x = (x & ~(ml | mr)) | t;
    return  x;
}
```

The version given by Torsten Sillke (cf. `http://www.mathematik.uni-bielefeld.de/~sillke/`)

```
static inline ulong Butterfly4(ulong x)
{
   ulong m = 0x00f000f0;
   return ((x & m) << 4) | ((x >> 4) & m) | (x & ~(0x11*m));
}
```

looks much nicer, but seems to use one more register (4 instead of 3) when compiled.

8.17 Bit sequency

Some doubtful functions of questionable usefulness can be found in [FXT: file `auxbit/bitsequency.h`]:

```
static inline ulong bit_sequency(ulong x)
// return the number of zero-one (or one-zero)
// transitions (sequency) of x.
{
    return bit_count( gray_code(x) );
}


static inline ulong first_sequency(ulong k)
// return the first (i.e. smallest) word with sequency k,
// e.g.  00..00010101010 (seq 8)
// e.g.  00..00101010101 (seq 9)
// must be:  1 <= k <= BITS_PER_LONG
{
    return inverse_gray_code( first_comb(k) );
}


static inline ulong last_sequency(ulong k)
// return the lasst (i.e. biggest) word with sequency k,
{
    return inverse_gray_code( last_comb(k) );
}


static inline ulong next_sequency(ulong x)
// return smallest integer with highest bit at greater or equal
// position than the highest bit of x that has the same number
// of zero-one transitions (sequency) as x.
// The value of the lowest bit is conserved.
//
// Zero is returned when there is no further sequence.
//
// e.g.:
//  ...1.1.1 ->
//  ..11.1.1 ->
//  ..1..1.1 ->
//  ..1.11.1 ->
//  ..1.1..1 ->
//  ..1.1.11 ->
//  .111.1.1 ->
//  .11..1.1 ->
//  .11.11.1 ->
//  .11.1..1 ->
//  .11.1.11 -> ...
//
{
```

```
    x = gray_code(x);
    x = next_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

8.18 Misc

...there is always some stuff that does not fit into any conceivable category. That goes to [FXT: file auxbit/bitmisc.h], e.g. the occasionally useful

```
static inline ulong bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  x << p;
}
```

and

```
static inline ulong cyclic_bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// The result is possibly wrapped around the word boundary.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  (x<<p) | (x>>(BITS_PER_LONG-p));
}
```

Rather weird functions like

```
static inline ulong single_bits(ulong x)
// Return word were only the single bits from x are set
{
    return  x & ~( (x<<1) | (x>>1) );
}
```

or

```
static inline ulong single_values(ulong x)
// Return word were only the single bits and the
// single zeros from x are set
{
    return  (x ^ (x<<1)) & (x ^ (x>>1));
}
```

or

```
static inline ulong border_values(ulong x)
// Return word were those bits/zeros from x are set
// that lie next to a zero/bit
{
    ulong g = x ^ (x>>1);
    g |= (g<<1);
    return   g | (x & 1);
}
```

or

```
static inline ulong block_bits(ulong x)
// Return word were only those bits from x are set
// that are part of a block of at least 2 bits
{
    return  x & ( (x<<1) | (x>>1) );
}
```

or

```
static inline ulong interior_bits(ulong x)
// Return word were only those bits from x are set
// that do not have a zero to their left or right
{
    return  x & ( (x<<1) & (x>>1) );
}
```

might not be the most often needed functions on this planet, but if you can use them you will love them.

[FXT: file `auxbit/branchless.h`] contains functions that avoid branches. With modern CPUs and their conditional move instructions these are not necessarily optimal:

```
static inline long max0(long x)
// Return max(0, x), i.e. return zero for negative input
// No restriction on input range
{
    return  x & ~(x >> (BITS_PER_LONG-1));
}
```

or

```
static inline ulong upos_abs_diff(ulong a, ulong b)
// Return abs(a-b)
// Both a and b must not have the most significant bit set
{
    long d1 = b - a;
    long d2 = (d1 & (d1>>(BITS_PER_LONG-1)))<<1;
    return  d1 - d2; // == (b - d) - (a + d);
}
```

The ideas used are sometimes interesting on their own:

```
static inline ulong average(ulong x, ulong y)
// Return (x+y)/2
// Result is correct even if (x+y) wouldn't fit into a ulong
// Use the fact that x+y == ((x&y)<<1) + (x^y)
//    that is:       sum ==  carries   + sum_without_carries
{
    return  (x & y) + ((x ^ y) >> 1);
}
```

or

```
static inline void upos_sort2(ulong &a, ulong &b)
// Set {a, b} := {minimum(a, b), maximum(a,b)}
// Both a and b must not have the most significant bit set
{
    long d = b - a;
    d &= (d>>(BITS_PER_LONG-1));
    a += d;
    b -= d;
}
```

Note that the `upos_*()` functions only work for a limited range (highest bit must not be set) in order to have the highest bit emulate the carry flag.

```
static inline ulong contains_zero_byte(ulong x)
// Determine if any sub-byte of x is zero.
// Returns zero when x contains no zero byte and nonzero when it does.
// The idea is to subtract 1 from each of the bytes and then look for bytes
//   where the borrow propagated all the way to the most significant bit.
// To scan for other values than zero (e.g. 0xa5) use:
//  contains_zero_byte( x ^ 0xa5a5a5a5UL )
{
```

```
#if  BITS_PER_LONG == 32
    return ((x-0x01010101UL)^x) & (~x) & 0x80808080UL;
    // return ((x-0x01010101UL) ^ x) & 0x80808080UL;
    // ...  gives false alarms when a byte of x is 0x80:
    //   hex: 80-01 = 7f,  7f^80 = ff,  ff & 80 = 80
#endif
#if  BITS_PER_LONG == 64
    return ((x-0x0101010101010101UL) ^ x) & (~x) & 0x8080808080808080UL;
#endif
}
```

from [FXT: file `auxbit/zerobyte.h`] may only be a gain for $\geq$128 bit words (cf. [FXT: `long_strlen` and `long_memchr` in `aux1/bytescan.cc`]), however, the underlying idea is nice enough to be documented here.

8.19 Hilbert's space-filling curve

The famous space-filling curve of Hilbert can be used to define a mapping from a one-dimensional coordinate (linear coordinate of the curve) to the pair of coordinates x and y. The function ([FXT: `hilbert` in `auxbit/hilbert.cc`]) that does the job is a state-engine as suggested in [53], item 115:

```
void
hilbert(ulong t, ulong &x, ulong &y)
// Transform linear coordinate to hilbert x and y
{
    ulong xv = 0, yv = 0;
    ulong c01 = (0<<2);  // (2<<2) for transposed output (swapped x, y)
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        ulong abi = t >> (BITS_PER_LONG-2);
        t <<= 2;
        ulong st = htab[ (c01<<2) | abi ];
        c01 = st & 3;
        yv <<= 1;
        yv |= ((st>>2) & 1);
        xv <<= 1;
        xv |= (st>>3);
    }
    x = xv;  y = yv;
}
```

The table used is defined as

```
static const ulong htab[] = {
#define HT(xi,yi,c0,c1) ((xi<<3)+(yi<<2)+(c0<<1)+(c1))
    // index == HT(c0,c1,ai,bi)
    HT( 0, 0,  1, 0 ),
    HT( 0, 1,  0, 0 ),
    HT( 1, 1,  0, 0 ),
    HT( 1, 0,  0, 1 ),
    HT( 1, 1,  1, 1 ),
    HT( 0, 1,  0, 1 ),
    HT( 0, 0,  0, 1 ),
    HT( 1, 0,  0, 0 ),
    HT( 0, 0,  0, 0 ),
    HT( 1, 0,  1, 0 ),
    HT( 1, 1,  1, 0 ),
    HT( 0, 1,  1, 1 ),
    HT( 1, 1,  0, 1 ),
    HT( 1, 0,  1, 1 ),
    HT( 0, 0,  1, 1 ),
    HT( 0, 1,  1, 0 )
#undef HT
};
```

Apart from the nice images the algorithm can be used to generate a Gray code using the following observation: with each increment by one of the linear coordinate t exactly one of the generated x and y

changes its value be ± 1. Therefore the concatenation of the[4] Gray codes of x and y gives another Gray code. It turns out that the routine is most effective if the bits of (the Gray code of) x and y are interlaced as with the bit-zip function given in section 8.16:

```
ulong
hilbert_gray_code(ulong t)
//
// A Gray code based on the function hilbert().
// Equivalent to the following sequence of statements:
//   hilbert(t, x, y);
//   x=gray_code(x);
//   y=gray_code(y);
//   return bitzip(y, x);
{
    ulong g = 0;
    ulong c01 = (0<<2);  // (2<<2) for transposed output (swapped x, y)
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        ulong abi = t >> (BITS_PER_LONG-2);
        t <<= 2;

        ulong st = htab[ (c01<<2) | abi ];
        c01 = st & 3;

        g <<= 2;
        g |= (st>>2);
    }
#if  ( BITS_PER_LONG <= 32 )
    g ^= ( (g & 0x55555555UL) >> 2 );
    g ^= ( (g & 0xaaaaaaaaUL) >> 2 );
#else
    g ^= ( (g & 0x5555555555555555UL) >> 2 );
    g ^= ( (g & 0xaaaaaaaaaaaaaaaaUL) >> 2 );
#endif
    return g;
}
```

The demo ([FXT: file `demo/hilbert-demo.cc`]) should clarify the idea, it gives the output

```
    t ->    binary(x)  binary(y)= ( x,  y)   gx=gray(x)  gy=gray(y)  z=bitzip(gx,gy) == g
 #  0 ->    x=......   y=...... = ( 0,  0)   gx=......   gy=......   z=......   g=......
 #  1 ->    x=.....1   y=...... = ( 1,  0)   gx=.....1   gy=......   z=....1.   g=....1.
 #  2 ->    x=.....1   y=.....1 = ( 1,  1)   gx=.....1   gy=.....1   z=....11   g=....11
 #  3 ->    x=......   y=.....1 = ( 0,  1)   gx=......   gy=.....1   z=.....1   g=.....1
 #  4 ->    x=......   y=....1. = ( 0,  2)   gx=......   gy=....11   z=...1.1   g=...1.1
 #  5 ->    x=......   y=....11 = ( 0,  3)   gx=......   gy=....1.   z=...1..   g=...1..
 #  6 ->    x=.....1   y=....11 = ( 1,  3)   gx=.....1   gy=....1.   z=...11.   g=...11.
 #  7 ->    x=.....1   y=....1. = ( 1,  2)   gx=.....1   gy=....11   z=...111   g=...111
 #  8 ->    x=....1.   y=....1. = ( 2,  2)   gx=....11   gy=....11   z=..1111   g=..1111
 #  9 ->    x=....1.   y=....11 = ( 2,  3)   gx=....11   gy=....1.   z=..111.   g=..111.
 # 10 ->    x=....11   y=....11 = ( 3,  3)   gx=....1.   gy=....1.   z=..11..   g=..11..
 # 11 ->    x=....11   y=....1. = ( 3,  2)   gx=....1.   gy=....11   z=..11.1   g=..11.1
 # 12 ->    x=....11   y=.....1 = ( 3,  1)   gx=....1.   gy=.....1   z=..1..1   g=..1..1
 # 13 ->    x=....1.   y=.....1 = ( 2,  1)   gx=....11   gy=.....1   z=..1.11   g=..1.11
 # 14 ->    x=....1.   y=...... = ( 2,  0)   gx=....11   gy=......   z=..1.1.   g=..1.1.
 # 15 ->    x=....11   y=...... = ( 3,  0)   gx=....1.   gy=......   z=..1...   g=..1...
 # 16 ->    x=...1..   y=...... = ( 4,  0)   gx=...11.   gy=......   z=1.1...   g=1.1...
 # 17 ->    x=...1..   y=.....1 = ( 4,  1)   gx=...11.   gy=.....1   z=1.1..1   g=1.1..1
 # 18 ->    x=...1.1   y=.....1 = ( 5,  1)   gx=...111   gy=.....1   z=1.1.11   g=1.1.11
 # 19 ->    x=...1.1   y=...... = ( 5,  0)   gx=...111   gy=......   z=1.1.1.   g=1.1.1.
 # 20 ->    x=...11.   y=...... = ( 6,  0)   gx=...1.1   gy=......   z=1...1.   g=1...1.
 # 21 ->    x=...111   y=...... = ( 7,  0)   gx=...1..   gy=......   z=1.....   g=1.....
 # 22 ->    x=...111   y=.....1 = ( 7,  1)   gx=...1..   gy=.....1   z=1....1   g=1....1
 # 23 ->    x=...11.   y=.....1 = ( 6,  1)   gx=...1.1   gy=.....1   z=1...11   g=1...11
 # 24 ->    x=...11.   y=....1. = ( 6,  2)   gx=...1.1   gy=....11   z=1..111   g=1..111
 # 25 ->    x=...111   y=....1. = ( 7,  2)   gx=...1..   gy=....11   z=1..1.1   g=1..1.1
 # 26 ->    x=...111   y=....11 = ( 7,  3)   gx=...1..   gy=....1.   z=1..1..   g=1..1..
 # 27 ->    x=...11.   y=....11 = ( 6,  3)   gx=...1.1   gy=....1.   z=1..11.   g=1..11.
 # 28 ->    x=...1.1   y=....11 = ( 5,  3)   gx=...111   gy=....1.   z=1.111.   g=1.111.
 # 29 ->    x=...1.1   y=....1. = ( 5,  2)   gx=...111   gy=....11   z=1.1111   g=1.1111
```

[4] actually any Gray code does it, we use the standard binary version here.

```
 # 30 ->    x=...1..   y=....1. = ( 4,  2)   gx=...11.   gy=....11   z=1.11.1   g=1.11.1
 # 31 ->    x=...1..   y=....11 = ( 4,  3)   gx=...11.   gy=....1.   z=1.11..   g=1.11..
```

8.20 Manipulation of colors

In the following it is assumed that the type `uint` (unsigned integer) contains at least 32 bit. In this section This data type is exclusively used as a container for three color channels that are assumed to be 8 bit each and lie at the lower end of the word. The functions do not depend on how the channels are ordered (e.g. RGB or BGR).

The following functions are obviously candidates for your CPUs SIMD-extensions (if it has any). However, having the functionality in a platform independent manner that is sufficiently fast for most practical purposes[5] is reason enough to include this section.

Scaling a color by an integer value:

```
static inline uint color01(uint c, ulong v)
// return color with each channel scaled by v
// 0 <= v <= (1<<16)  corresponding to 0.0 ... 1.0
{
    uint t;
    t = c & 0xff00ff00;  // must include alpha channel bits ...
    c ^= t;  // ... because they must be removed here
    t *= v;
    t >>= 24; t <<= 8;
    v >>= 8;
    c *= v;
    c >>= 8;
    c &= 0xff00ff;
    return  c | t;
}
```

... used in the computation of the weighted average of colors:

```
static inline uint color_mix(uint c1, uint c2, ulong v)
// return channelwise average of colors
//  (1.0-v)*c1 and v*c2
//
// 0 <= v <= (1<<16)  corresponding to 0.0 ... 1.0
// c1   ...    c2
{
    ulong w = ((ulong)1<<16)-v;
    c1 = color01(c1, w);
    c2 = color01(c2, v);
    return  c1 + c2;  // no overflow in color channels
}
```

Channel-wise average of two colors:

```
static inline uint color_mix_50(uint c1, uint c2)
// return channelwise average of colors c1 and c2
//
// shortcut for the special case (50% transparency)
//   of color_mix(c1, c2, "0.5")
//
// least significant bits are ignored
{
    return  ((c1 & 0xfefefe) + (c2 & 0xfefefe)) >> 1;  // 50% c1
}
```

... and with higher weight of the first color:

```
static inline uint color_mix_75(uint c1, uint c2)
// least significant bits are ignored
```

[5] The software rendering program that uses these functions operates at a not too small fraction of memory bandwidth when all of environment mapping, texture mapping and translucent objects are shown with (very) simple scenes.

```
{
    return  color_mix_50(c1, color_mix_50(c1, c2));  // 75% c1
}
```

Saturated addition of color channels:

```
static inline uint color_sum(uint c1, uint c2)
// least significant bits are ignored
{
    uint s = color_mix_50(c1, c2);
    return  color_sum_adjust(s);
}
```

which uses:

```
static inline uint color_sum_adjust(uint s)
// set color channel to max (0xff) iff an overflow occured
//  (that is, leftmost bit in channel is set)
{
    uint m = s & 0x808080;  // 1000 0000 // overflow bits
    s ^= m;
    m >>= 7;    // 0000 0001
    m *= 0xff;  // 1111 1111  // optimized to (m<<8)-m by gcc
    return  (s << 1) | m;
}
```

Channel-wise product of two colors:

```
static inline uint color_mult(uint c1, uint c2)
// corresponding to an object of color c1
//  illuminated by a light of color c2
{
    uint t = ((c1 & 0xff) * (c2 & 0xff)) >> 8;
    c1 >>= 8;  c2 >>= 8;
    t |= ((c1 & 0xff) * (c2 & 0xff)) & 0xff00;
    c1 &= 0xff00;  c2 >>= 8;
    t |= ((c1 * c2) & 0xff0000);
    return  t;
}
```

When one does not want to discard the lowest channel bits (e.g. because numerous such operations appear in a row) a more 'perfect' version is required:

```
static inline uint perfect_color_mix_50(uint c1, uint c2)
// return channelwise average of colors c1 and c2
{
//    uint t = (c1 | c2) & 0x010101; // lowest channels bits in any arg
    uint t = (c1 & c2) & 0x010101; // lowest channels bits in both args
    return  color_mix_50(c1, c2) + t;
}


static inline uint perfect_color_sum(uint c1, uint c2)
{
    uint srb = (c1 & 0xff00ff) + (c2 & 0xff00ff) + 0x010001;
    uint mrb = srb & 0x01000100;
    srb ^= mrb;
    uint sg = (c1 & 0xff00) + (c2 & 0xff00) + 0x0100;
    uint mg = (sg & 0x010000);
    sg ^= mg;
    uint m = (mrb | mg) >> 1;  // 1000 0000 // overflow bits
    m |= (m >> 1);  // 1100 0000
    m |= (m >> 2);  // 1111 0000
    m |= (m >> 4);  // 1111 1111
    return  srb | sg | m;
}
```

Note that the last two functions are overkill for most practical purposes.

8.21 2-adic inverse and root

The 2-adic inverse can be computed using an iteration (see 13.3) with quadratic convergence. The number to be inverted has to be odd.

```
inline ulong inv2adic(ulong x)
// Return inverse modulo 2**BITS_PER_LONG
// x must be odd
// number of correct bits are doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 3, 6, 12, 24, 48, 96, ... bits (or better)
{
    if ( 0==(x&1) )  return 0;  // not invertible
    ulong i = x;  // correct to three bits at least
    ulong p;
    do
    {
        p = i * x;
        i *= (2UL - p);
    }
    while ( p!=1 );
    return  i;
}
```

With the inverse square root we choose the start value to match $\lfloor d/2 \rfloor + 1$ as that guarantees four bits of initial precision. Moreover, we get control to which of the two possible values the inverse square root is finally reached. The argument modulo 8 has to be equal to one.

```
inline ulong invsqrt2adic(ulong d)
// Return inverse square root modulo 2**BITS_PER_LONG
// must have d==1 mod 8
// number of correct bits are doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 4, 8, 16, 32, 64, ... bits (or better)
{
    if ( 1 != (d&7) )  return 0;  // no inverse sqrt
    // start value: if d == ****10001 ==> x := ****1001
    ulong x = (d >> 1) | 1;
    ulong p, y;
    do
    {
        y = x;
        p = (3 - d * y * y);
        x = (y * p) >> 1;
    }
    while ( x!=y );
    return  x;
}
```

The square root can be obtained by final multiplication with d.

```
inline ulong sqrt2adic(ulong d)
// Return square root modulo 2**BITS_PER_LONG
// must have d==1 mod 8  or  d==4 mod 32,  d==16 mod 128
//  ... d==4**k mod 4**(k+3)
// undefined return if condition does not hold
{
    if ( 0==d )  return  0;

    ulong s = 0;
    while ( 0==(d&1) )  { d >>= 1; ++s; }
    d *= invsqrt2adic(d);
    d <<= (s>>1);

    return   d;
}
```

Note that the 2-adic square root is something completely different from the integer square root.

The described functions can be found in [FXT: file `auxbit/bit2adic.h`]. A little demo is [FXT: file `demo/bit2adic-demo.cc`], where no inverse or square root is given, it does not exit:

```
    x   = ...............................1  =    1
    inv = ...............................1
   sqrt = ...............................1

    x   = 11111111111111111111111111111111  =   -1
    inv = 11111111111111111111111111111111

    x   = ..............................1.  =    2

    x   = 11111111111111111111111111111111.  =   -2

    x   = .............................11  =    3
    inv = i.i.i.i.i.i.i.i.i.i.i.i.i.i.i.11

    x   = 11111111111111111111111111111111.1  =   -3
    inv = .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1

    x   = ............................1..  =    4
   sqrt = ............................1.

    x   = 11111111111111111111111111111111..  =   -4

    x   = ...........................1.1  =    5
    inv = ii..ii..ii..ii..ii..ii..ii..ii.1

    x   = 11111111111111111111111111111111.11  =   -5
    inv = ..11..11..11..11..11..11..11..11

    x   = ............................11.  =    6

    x   = 11111111111111111111111111111111.1.  =   -6

    x   = ..........................111  =    7
    inv = i.ii.ii.ii.ii.ii.ii.ii.ii.ii.111

    x   = 11111111111111111111111111111111..1  =   -7
    inv = .1..1..1..1..1..1..1..1..1..1..1
   sqrt = i..111..i..ii...ii......1.ii.i.1

    x   = ............................1...  =    8

    x   = 11111111111111111111111111111111...  =   -8

    x   = ..........................1..1  =    9
    inv = ..iii...iii...iii...iii...iii..1
   sqrt = iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii.1
```

For further information on 2-adic (more generally p-adic) numbers see [11], [15] and also [67].

8.22 Powers of the Gray code

Consider the Gray code as a transform of a binary word. The following list shows the eight vectors of the 'canonical basis' in the leftmost column. The columns to the right show the powers (repeated applications) of the Gray code for 8-bit words:

```
    .......1    .......1    .......1    .......1    .......1    .......1    .......1    .......1
    ......i.    ......ii    ......i.    ......ii    ......i.    ......ii    ......i.    ......ii
    .....i..    .....ii.    .....i.i    .....iii    .....i..    .....ii.    .....i.i    .....iii
    ....i...    ....ii..    ....i.i.    ....iiii    ....i...    ....ii..    ....i.i.    ....iiii
    ...i....    ...ii...    ...i.i..    ...iiiii    ...i....    ...ii...    ...i.i..    ...iiiii
    ..i.....    ..ii....    ..i.i...    ..iiiii.    ..i.....    ..ii....    ..i.i...    ..iiiiii
    .i......    .ii.....    .i.i....    .iiiii..    .i.....i    .ii.....    .i.i....    .iiiiiii
    i.......    ii......    i.i.....    iiiii...    i.....i.    ii.....i    i.i.....    iiiiiiii
    g**0=id     g**1=g      g**2        g**3        g**4        g**5        g**6        g**7=g**(-1)
```

This motivates the

```
inline ulong gray_pow(ulong x, ulong e)
// Return (gray_code**e)(x)
// gray_pow(x, 1) == gray_code(x)
// gray_pow(x, BITS_PER_LONG-1) == inverse_gray_code(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 )  x ^= x >> s;  // gray ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
```

The powers of the inverse Gray code cycle through the columns in the opposite direction, so

```
inline ulong inverse_gray_pow(ulong x, ulong e)
// Return (inverse_gray_code**(e))(x)
//   == (gray_code**(-e))(x)
// inverse_gray_pow(x, 1) == inverse_gray_code(x)
// gray_pow(x, BITS_PER_LONG-1) == gray_code(x)
{
    return  gray_pow(x, -e);
}
```

Same for the green code (this time using `g` for the green code):

```
    .......1    ......11    ....._1_1    ...1111    ..1..1    .11..11    _1_1_1_1    11111111
    ......1.    .....11.    ...1_1_.    ..1111.    .1..1.    11..11.    1_1_1_1.    1111111.
    .....1..    ....11..    ..1_1_..    .1111..    1..1..    .11..11    _1_1_1..    111111..
    ....1...    ...11...    .1_1_1...    1111...    ..1..1    11..11.    1_1_1_1.    11111...
    ...1....    ..11....    1_1_1_1.    111.....    .1..1.    .11..11    _1_1_1_.    1111....
    ..1.....    .11.....    _1_1_1_.    11.....    1..1..    11..11.    1_1_1_1.    111.....
    .1......    11......    1_1_1_1.    11.....    .1..1.    .11..11    _1_1_1_.    11......
    1.......    1.......    _1_1_1_.    1.....1    1..1..    1..11..    1_1_1_1.    1.......
    g**0=id     g**1=g      g**2        g**3       g**4      g**5       g**6        g**7=G**(-1)
```

We just have to reverse the shift operator for `green ** s`

```
inline ulong green_pow(ulong x, ulong e)
// Return (green_code**e)(x)
// green_pow(x, 1) == green_code(x)
// green_pow(x, BITS_PER_LONG-1) == inverse_green_code(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 )  x ^= x << s;  // green ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
inline ulong inverse_green_pow(ulong x, ulong e)
// Return (inverse_green_code**(e))(x)
//   == (green_code**(-e))(x)
// inverse_green_pow(x, 1) == inverse_green_code(x)
// green_pow(x, BITS_PER_LONG-1) == green_code(x)
{
    return  green_pow(x, -e);
}
```

8.23 Invertible transforms on words

The functions presented in this section are 'transforms' on binary words that are invertible.
Consider

```
inline ulong blue_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        a ^= ( (a&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

and

```
inline ulong yellow_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
```

```
    ulong m = ~0UL >> s;
    while ( s )
    {
        a ^= ( (a&m) << s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

[FXT: file `auxbit/bittransforms.h`] Both involve a computational work $\sim \log_2(b)$ where b is the number of bits per word (`BITS_PER_LONG`). The `blue_code` can be used as a fast implementation for the composition of a binary polynomial with $x+1$, see page 267.

[FXT: file `demo/bittransforms-blue-demo.cc`] gives (leftmost column gives Gray code for comparison, bit-counts at the right of each column):

```
 0:   g=......  0   b=......  0   y=................................  0
 1:   g=.....1  1   b=.....1  1   y=11111111111111111111111111111111 32
 2:   g=....11  2   b=....11  2   y=1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
 3:   g=....1.  1   b=....1.  1   y=.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1 16
 4:   g=...11.  2   b=...1.1  2   y=11..11..11..11..11..11..11..11.. 16
 5:   g=...111  3   b=...1..  1   y=..11..11..11..11..11..11..11..11 16
 6:   g=...1.1  2   b=...11.  2   y=.11..11..11..11..11..11..11..11. 16
 7:   g=...1..  1   b=...111  3   y=1..11..11..11..11..11..11..11..1 16
 8:   g=..11..  2   b=..1111  4   y=1...1...1...1...1...1...1...1...  8
 9:   g=..11.1  3   b=..111.  3   y=.111.111.111.111.111.111.111.111 24
10:   g=..1111  4   b=..11..  2   y=..1...1...1...1...1...1...1...1.  8
11:   g=..111.  3   b=..11.1  3   y=11.111.111.111.111.111.111.111.1 24
12:   g=..1.1.  2   b=..1.1.  2   y=.1...1...1...1...1...1...1...1..  8
13:   g=..1.11  3   b=..1.11  3   y=1.111.111.111.111.111.111.111.11 24
14:   g=..1..1  2   b=..1..1  2   y=111.111.111.111.111.111.111.111. 24
15:   g=..1...  1   b=..1...  1   y=...1...1...1...1...1...1...1...1  8
16:   g=.11...  2   b=.1...1  2   y=1111....1111....1111....1111.... 16
17:   g=.11..1  3   b=.1....  1   y=....1111....1111....1111....1111 16
18:   g=.11.11  4   b=.1..1.  2   y=.1.11.1..1.11.1..1.11.1..1.11.1. 16
19:   g=.11.1.  3   b=.1..11  3   y=1.1..1.11.1..1.11.1..1.11.1..1.1 16
20:   g=.1111.  4   b=.1.1..  2   y=..1111....1111....1111....1111.. 16
21:   g=.11111  5   b=.1.1.1  3   y=11....1111....1111....1111....11 16
22:   g=.111.1  4   b=.1.111  4   y=1..1.11.1..1.11.1..1.11.1..1.11. 16
23:   g=.111..  3   b=.1.11.  3   y=.11.1..1.11.1..1.11.1..1.11.1..1 16
24:   g=.1.1..  2   b=.1111.  4   y=.1111....1111....1111....1111... 16
25:   g=.1.1.1  3   b=.11111  5   y=1....1111....1111....1111....111 16
26:   g=.1.111  4   b=.111.1  4   y=11.1..1.11.1..1.11.1..1.11.1..1. 16
27:   g=.1.11.  3   b=.111..  3   y=..1.11.1..1.11.1..1.11.1..1.11.1 16
28:   g=.1..1.  2   b=.11.11  4   y=1.11.1..1.11.1..1.11.1..1.11.1.. 16
29:   g=.1..11  3   b=.11.1.  3   y=.1..1.11.1..1.11.1..1.11.1..1.11 16
30:   g=.1...1  2   b=.11...  2   y=...1111....1111....1111....1111. 16
31:   g=.1....  1   b=.11..1  3   y=111....1111....1111....1111....1 16
```

The parity of $B(a)$ is equal to the lowest bit of a. Up to the $a = 47$ the bit-count varies by ± 1 between successive values of $B(a)$, the transition $B(47) \to B(48)$ changes the bit-count by 3. The sequence of the indices a where the bit-count changes by more than one is 47, 51, 59, 67, 75, 79, 175, 179, 187, 195, 203, 207, 291, 299, 339, 347, 419, 427, 467, 475, 531, 539, ...

The sequence of fixed points is 0, 1, 6, 7, 18, 19, 20, 21, 106, 107, 108, 109, 120, 121, 126, 127, 258, 259, 260, 261, 272, 273, 278, 279, 360, 361, 366, 367, 378, 379, 380, 381, 1546, 1547, 1548, 1549, 1560, 1561, 1566, 1567, 1632, 1633, ...

The yellow-code might be a good candidate for 'randomization' of binary words.

The blue-code maps any range $[0 \dots 2^k - 1]$ onto itself. Both blue-code and yellow-code are involutions (self-inverse).

The transforms

```
inline ulong red_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
```

```
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u<<s);
        a ^= (v>>s);
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

and

```
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u>>s);
        a ^= (v<<s);
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
```

look like

```
 0:   r=i...............................  0   c=                                0
 1:   r=1i..............................  1   c=iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii 32
 2:   r=11..............................  2   c=.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
 3:   r=.1..............................  1   c=1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1. 16
 4:   r=1.1.............................  2   c=..ii..ii..ii..ii..ii..ii..ii..ii 16
 5:   r=..1.............................  1   c=ii..ii..ii..ii..ii..ii..ii..ii.. 16
 6:   r=.11.............................  2   c=.ii..ii..ii..ii..ii..ii..ii..ii. 16
 7:   r=111............................  3   c=i..ii..ii..ii..ii..ii..ii..ii..i 16
 8:   r=1111...........................  4   c=.......1.......1.......1.......1  8
 9:   r=.111...........................  3   c=iii.iii.iii.iii.iii.iii.iii.iii. 24
10:   r=..11...........................  2   c=.......1.......1.......1.......1  8
11:   r=1.11...........................  3   c=i.iii.iii.iii.iii.iii.iii.iii.ii 24
12:   r=.1.1...........................  2   c=.......1.......1.......1.......1  8
13:   r=11.1...........................  3   c=ii.iii.iii.iii.iii.iii.iii.iii.i 24
14:   r=1..1...........................  2   c=.iii.iii.iii.iii.iii.iii.iii.iii 24
15:   r=...1...........................  1   c=1...1...1...1...1...1...1...1.  8
16:   r=1...1.........................  2   c=....iiii....iiii....iiii....iiii 16
17:   r=....1.........................  1   c=iiii....iiii....iiii....iiii.... 16
18:   r=.1..1.........................  2   c=.1.ii.i..1.ii.i..1.ii.i..1.ii.i. 16
19:   r=11..1.........................  3   c=i.i..i.ii.i..i.ii.i..i.ii.i..i.i 16
20:   r=..11.1.........................  2   c=.iiii....iiii....iiii....iiii.... 16
21:   r=1.11.1.........................  3   c=ii..iiii..iiii..iiii..iiii..iiii 16
22:   r=111.1.........................  4   c=.ii.i..i.ii.i..i.ii.i..i.ii.i..i 16
23:   r=.11.1.........................  3   c=i..i.ii.i..i.ii.i..i.ii.i..i.ii. 16
24:   r=.1111.........................  4   c=...iiii....iiii....iiii....iiii. 16
25:   r=11111.........................  5   c=iii....iiii....iiii....iiii....i 16
26:   r=1.111.........................  4   c=.i..i.ii.i..i.ii.i..i.ii.i..i.ii 16
27:   r=..111.........................  3   c=i.ii.i..i.ii.i..i.ii.i..i.ii.i.. 16
28:   r=11.11.........................  4   c=.i.ii.i..i.ii.i..i.ii.i..i.ii.i. 16
29:   r=.1.11.........................  3   c=ii..i..i.ii..i..i.ii..i..i.ii..i. 16
30:   r=...11.........................  2   c=.iiii....iiii....iiii....iiii.... 16
31:   r=1..11.........................  3   c=i....iiii....iiii....iiii....iii 16
```

[FXT: file `demo/bittransforms-red-demo.cc`]

Relations between the transforms

We write B for the blue-code (transform), Y for the yellow-code and r for bit-reversal (the `revbin`-function). Then B and Y are connected by the relations

$$B = Y\,r\,Y \tag{8.1}$$
$$Y = B\,r\,B \tag{8.2}$$
$$B = r\,Y\,r \tag{8.3}$$
$$Y = r\,B\,r \tag{8.4}$$
$$r = Y\,B\,Y \tag{8.5}$$
$$r = B\,Y\,B \tag{8.6}$$

As said, B and Y are self-inverse:

$$B^{-1} = B \qquad B\,B = \mathrm{id} \tag{8.7}$$
$$Y^{-1} = Y \qquad Y\,Y = \mathrm{id} \tag{8.8}$$

The red-code and the cyan-code are not involutions ('square roots of identity') but third roots of identity (Using R for the red-code, C for the cyan-code):

$$R\,R\,R = \mathrm{id} \qquad R^{-1} = R\,R = C \tag{8.9}$$
$$C\,C\,C = \mathrm{id} \qquad C^{-1} = C\,C = R \tag{8.10}$$
$$R\,C = C\,R = \mathrm{id} \tag{8.11}$$

By construction

$$R = r\,B \tag{8.12}$$
$$C = r\,Y \tag{8.13}$$

Similar inter-relations as for B and Y hold for R and C:

$$R = C\,r\,C \tag{8.14}$$
$$C = R\,r\,R \tag{8.15}$$
$$R = r\,C\,r \tag{8.16}$$
$$C = r\,R\,r \tag{8.17}$$
$$R = R\,C\,R \tag{8.18}$$
$$C = C\,R\,C \tag{8.19}$$

One has

$$r = Y\,R = R\,B = B\,C = C\,Y \tag{8.20}$$

Further

$$B = R\,Y = Y\,C = R\,B\,R = C\,B\,C \tag{8.21}$$
$$Y = C\,B = B\,R = R\,Y\,R = C\,Y\,C \tag{8.22}$$

$$R = B\,Y = B\,C\,B = Y\,C\,Y \tag{8.23}$$
$$C = Y\,B = B\,R\,B = Y\,R\,Y \tag{8.24}$$

$$\mathrm{id} = B\,Y\,C = R\,Y\,B \tag{8.25}$$
$$\mathrm{id} = C\,B\,Y = B\,R\,Y \tag{8.26}$$
$$\mathrm{id} = Y\,C\,B = Y\,B\,R \tag{8.27}$$

The multiplication table lists $Z = Y\,X$. The R in the third column of the second row says that $r\,B = R$. The letter i is used for identity (id). An asterisk says that $X\,Y = Y\,X$.

	i	r	B	Y	R	C
i	i*	r*	B*	Y*	R*	C*
r	r*	i*	R	C	B	Y
B	B*	C	i*	R	Y	r
Y	Y*	R	C	i*	r	B
R	R*	Y	r	B	C*	i*
C	C*	B	Y	r	i*	R*

Relations to Gray code and green code

Write g for the Gray code, then:

$$g\,B\,g\,B = \mathrm{id} \tag{8.28}$$
$$g\,B\,g = B \tag{8.29}$$
$$g^{-1}\,B\,g^{-1} = B \tag{8.30}$$
$$g\,B = B\,g^{-1} \tag{8.31}$$

Let S_k be the operator that rotates a word by k bits (bit zero is moved to position k, use [FXT: `bit_rotate_sgn` in `auxbit/bitrotate.h`]) then

$$Y\,S_{+1}\,Y = g \tag{8.32}$$
$$Y\,S_{-1}\,Y = g^{-1} \tag{8.33}$$
$$Y\,S_k\,Y = g^k \tag{8.34}$$

'Shift in the frequency domain is derivative in time domain'.

Let e be the green code operator, then

$$B\,S_{+1}\,B = e^{-1} \tag{8.35}$$
$$B\,S_{-1}\,B = e \tag{8.36}$$
$$B\,S_k\,B = e^{-k} \tag{8.37}$$

More transforms by symbolic powering

The idea of powering a transform (as done for the Gray code in section 8.22) can be applied to the 'color'-transforms as exemplified for the blue-code:

```
inline ulong blue_xcode(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        if ( x & 1 )  a ^= ( (a&m) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

The result is *not* the power of the blue-code which would be pretty boring as $B\,B = \mathrm{id}$. Instead the transform (and the equivalents for Y, R and C, see [FXT: file `auxbit/bitxtransforms.h`]) are more interesting: All relations between the transforms are still valid, if the symbolic exponent is identical with all terms. For example, we had $B\,B = \mathrm{id}$, now $B^x\,B^x = \mathrm{id}$ is true for all x (there are essentially `BITS_PER_LONG` different x). Similarly, $C\,C = R$ now has to be $C^x\,C^x = R^x$. That is, we have `BITS_PER_LONG` different versions of our four transforms that share their properties with the 'simple' versions. Among them `BITS_PER_LONG` transforms B^x and Y^x that are involutions and C^x and R^x that are third roots of the identity: $C^x\,C^x\,C^x = R^x\,R^x\,R^x = \mathrm{id}$.

While not powers of the simple versions, we still have $B^0 = Y^0 = R^0 = C^0 = \mathrm{id}$. Further, let e be the 'exponent' of all ones and Z be any of the transforms, then $Z^e = Z$, Writing '+' for the XOR operation, then $Z^x\,Z^y = Z^{x+y}$ and so $Z^x\,Z^y = Z$ whenever $x + y = e$.

The building blocks of the transforms

Consider the following transforms on two-bit words where addition is over $GF(2)$ (that is, addition is XOR)

$$\mathrm{id}_2\, v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{8.38}$$

$$r_2\, v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \tag{8.39}$$

$$B_2\, v = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ b \end{bmatrix} \tag{8.40}$$

$$Y_2\, v = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{8.41}$$

$$R_2\, v = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a+b \end{bmatrix} \tag{8.42}$$

$$C_2\, v = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ a \end{bmatrix} \tag{8.43}$$

It can easily be verified that for these the same relations hold as for id, r, B, Y, R, C. In fact the 'color-transforms' and (trivially) id are the transforms obtained by the repeated Kronecker-products of the matrices.
The corresponding version of the bit-reversal is [FXT: `xrevbin` in `auxbit/revbin.h`]:

```
inline ulong xrevbin(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        if ( x & 1 )  a = ( (a & m) << s ) ^ ( (a & (~m)) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

Then, for example $R^x = r^x\, B^x$ (cf. 8.12).

Talking about the blue-code, the symbolic powering is equivalent to selecting individual levels of the Reed-Muller transform (section 5.8). Y, R and C are variants that transform.

The symbolic-powering idea can be carried further. The values of the mask `m` in the blue-code for the successive stages are (for 32-bit words)

```
1111111111111111........................
11111111....11111111....11111111....1111
1111..11..1111..1111..1111..1111..1111..
11..11..11..11..11..11..11..11..11..11..
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.
```

Considering the lowest bit of each block

```
................1...............1.......
...1...1...1...1...1...1...1...1...1...1...
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.
```

it is obvious that one could use all bits of a word to select the blocks where the 'atomic' transform should be applied.

8.24 CPU instructions often missed

Essential

- A bit-reverse instruction. Every general purpose CPU should have one. Considering that the necessary circuitry is the most simple imaginable it is hard to understand CPU manufacturers tend to 'forget' it.
- A parity bit for the complete machine word. The parity of a word is the number of bits modulo two, not the complement of it. Even better would be a instruction for the inverse Gray-code which can (among other things) serve as building block for fast transforms over $GF(2)$.
- A bit-count instruction. Clearly, this would also give the parity at bit zero.
- Primitives for permutations of bits. The alpha processor has such. A instruction using the equivalent of the idea presented at the end of section 8.23 might be the most powerful approach.

Nice to have

- A fast conversion from integer to float and double (both directions).
- Multiplication corresponding to XOR as addition. That is, a multiplication without carries, the one used for polynomials over $GF(2)$. See [FXT: `bitpol_mult` in `auxbit/bitpol.h`].
- A bit-zip and a bit-unzip instruction, see [FXT: file `auxbit/bitzip.h`].
- A bit-gather and a bit-scatter instruction, see [FXT: file `auxbit/bitgather.h`]. This would include bit-zip and its inverse.

Luxury items

- The mentioned multiplication for polynomials over $GF(2)$ with a modulus, see [FXT: `bitpolmod_mult` in `auxbit/bitpolmodmult.h`].
- Instructions for the multiplication of complex floating point numbers.
- Primitives for the fast orthogonal transforms like the radix -two, -four and possibly -eight butterflies for the Walsh and Hartley transform.
- Multiplication modulo the prime $2^{64} - 2^{32} + 1$.

Chapter 9

Sorting and searching

TBD: *chapter outline*

TBD: *counting sort, radix sort, merge sort*

9.1 Sorting

There are a few straight forward algorithms for sorting that scale with $\sim n^2$ (where n is the size of the array to be sorted).

Here we use selection sort whose idea is to find the minimum of the array, swap it with the first element and repeat for all elements but the first:

```
template <typename Type>
void selection_sort(Type *f, ulong n)
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( f[j]<v )
            {
                m = j;
                v = f[m];
            }
        }
        swap(f[i], f[m]);
    }
}
```

A verification routine is always handy:

```
template  <typename Type>
int is_sorted(const Type *f, ulong n)
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 2
    {
        if ( f[n] < f[n-1] )  break;
    }
    return  !n;
}
```

While the quicksort-algorithm presented below scales $\sim n\log(n)$ (in the average case) it does not just obsolete the more simple schemes because (1) for arrays small enough the 'simple' algorithm is usually

the fastest method because of its minimal bookkeeping overhead and (2) therefore it is used inside the quicksort for lengths below some threshold.

The main ingredient of quicksort is to *partition* the array: The corresponding routine reorders some elements where needed and returns some partition index k so that $max(f_0, \ldots, f_{k-1}) \leq min(f_k, \ldots, f_{n-1})$:

```
template <typename Type>
ulong partition(Type *f, ulong n)
// rearrange array, so that for some index p
// max(f[0] ... f[p]) <= min(f[p+1] ... f[n-1])
{
    swap( f[0], f[n/2]);  // avoid worst case with already sorted input
    const Type v = f[0];
    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do  { ++i; }  while ( f[i]<v );
        do  { --j; }  while ( f[j]>v );
        if ( i<j )  swap(f[i], f[j]);
        else        return j;
    }
}
```

which we want to be able to verify:

```
template <typename Type>
Type inline min(const Type *f, ulong n)
// returns minimum of array
{
    Type v = f[0];
    while ( n-- )  if ( f[n]<v )  v = f[n];
    return v;
}
template <typename Type>
inline Type max(const Type *f, ulong n)
// returns maximum of array
{
    Type v = f[0];
    while ( n-- )  if ( f[n]>v )  v = f[n];
    return v;
}
template  <typename Type>
int is_partitioned(const Type *f, ulong n, ulong k)
{
    ++k;
    Type lmax = max(f,   k);
    Type rmin = min(f+k, n-k);
    return  ( lmax<=rmin );
}
```

Quicksort calls `partition` on the whole array, then on the parts left and right from the partition index and repeat. When the size of the subproblems is smaller than a certain threshold selection sort is used.

```
template <typename Type>
void quick_sort(Type *f, ulong n)
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        selection_sort(f, n);
        return;
    }
    ulong p = partition(f, n);
    ulong ln = p + 1;
    ulong rn = n - ln;
    if ( ln>rn )  // recursion for shorter subarray
    {
        quick_sort(f+ln, rn);  // f[ln] ... f[n-1]   right
```

```
        n = ln;
    }
    else
    {
        quick_sort(f, ln);  // f[0]  ... f[ln-1]  left
        n = rn;
        f += ln;
    }
    goto start;
}
```

[FXT: file sort/sort.h]

TBD: *worst case and how to avoid it*

9.2 Searching

The reason why some data was sorted may be that a fast search has to be performed repeatedly. The following bsearch is $\sim \log(n)$ and works by the obvious subdivision of the data:

```
template <typename Type>
ulong bsearch(const Type *f, ulong n, const Type v)
// return index of first element in f[] that is == v
// return ~0 if there is no such element
// f[] must be sorted in ascending order
// must have  n!=0
{
    ulong nlo=0, nhi=n-1;
    while ( nlo != nhi )
    {
        ulong t = (nhi+nlo)/2;

        if ( f[t] < v )  nlo = t + 1;
        else             nhi = t;
    }

    if ( f[nhi]==v )  return nhi;
    else              return ~0UL;
}
```

A simple modification of bsearch makes it search the first element greater than v: Replace the operator == in the above code by >= and you have it: [FXT: bsearch_ge in sort/search.h].

Approximate matches are found by

```
template <typename Type>
ulong bsearch_approx(const Type *f, ulong n, const Type v, Type da)
// return index of first element x in f[] for which  |(x-v)| <= da
// return ~0 if there is no such element
// f[] must be sorted in ascending order
// da must be positive
// makes sense only with inexact types (float or double)
// must have  n!=0
{
    ulong k = bsearch_ge(f, n, v-da);
    if ( k<n )  k = bsearch_le(f+k, n-k, v+da);
    return k;
}
```

where bsearch_le() searches for an element less or equal to a given value.

When the values to be searched will themselves appear in monotone order you can reduce the total time used for searching with:

```
template <typename Type>
inline long search_down(const Type *f, const Type v, ulong &i)
// search v in f[], starting at i (so i must be < length)
// f[i] must be greater or equal v
// f[] must be sorted in ascending order
```

```
// returns index k if f[k]==v or ~0 if no such k is found
// i is updated so that it can be used for a following
// search for an element u where u < v
{
    while ( (f[i]>v) && (i>0) )  --i;

    if ( f[i]==v )  return i;
    else            return ~0UL;
}
```

[FXT: file sort/search.h]

9.3 Index sorting

While the 'plain' sorting reorders an array f so that, after it has finished, $f_k \leq f_{k+1}$ the following routines sort an array of indices without modifying the actual data:

```
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x)
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( f[x[j]]<v )
            {
                m = j;
                v = f[x[m]];
            }
        }

        swap(x[i], x[m]);
    }
}
```

Apart from the 'read only'-feature the index-sort routines have the nice property to perfectly work on non-contiguous data.

The verification code looks like:

```
template  <typename Type>
int is_idx_sorted(const Type *f, ulong n, const ulong *x)
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 1
    {
        if ( f[x[n]] < f[x[n-1]] )  break;
    }

    return  !n;
}
```

The index-sort routines reorder the indices in `x` such that `x` applied to `f` as a permutation (in the sense of section 7.13.3) will render `f` a sorted array.

While the transformation of `partition` is straight forward:

```
template <typename Type>
ulong idx_partition(const Type *f, ulong n, ulong *x)
// rearrange index array, so that for some index p
// max(f[x[0]] ... f[x[p]]) <= min(f[x[p+1]] ... f[x[n-1]])
{
    swap( x[0], x[n/2]);
    const Type v = f[x[0]];

    ulong i = 0UL - 1;
```

```
    ulong j = n;
    while ( 1 )
    {
        do  ++i;
        while ( f[x[i]]<v );
        do  --j;
        while ( f[x[j]]>v );
        if ( i<j )  swap(x[i], x[j]);
        else        return j;
    }
}
```

The index-quicksort itself deserves a minute of contemplation comparing it to the plain version:

```
template <typename Type>
void idx_quick_sort(const Type *f, ulong n, ulong *x)
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        idx_selection_sort(f, n, x);
        return;
    }
    ulong p = idx_partition(f, n, x);
    ulong ln = p + 1;
    ulong rn = n - ln;
    if ( ln>rn )  // recursion for shorter subarray
    {
        idx_quick_sort(f, rn, x+ln);  // f[x[ln]] ... f[x[n-1]]  right
        n = ln;
    }
    else
    {
        idx_quick_sort(f, ln, x);  // f[x[0]]  ... f[x[ln-1]]  left
        n = rn;
        x += ln;
    }
    goto start;
}
```

[FXT: file `sort/sortidx.h`]

The index-analogues of bsearch etc. are again straight forward, they can be found in [FXT: file `sort/searchidx.h`].

9.4 Pointer sorting

Pointer sorting is an idea similar to index sorting which is even less restricted than index sort: The data may be unaligned in memory. And overlapping. Or no data at all but port addresses controlling some highly dangerous machinery.

Thereby pointer sort is the perfect way to highly cryptic and powerful programs that seg-fault when you least expect it. Admittedly, all the 'dangerous' features of pointer sort except the unaligned one are also there in index sort. However, with index sort you will not so often use them *by accident*.

Just to make the idea clear, the array of indices is replaced by an array of pointers:

```
template <typename Type>
void ptr_selection_sort(const Type *f, ulong n, Type **x)
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
```

```
            if ( *x[j]<v )
            {
                m = j;
                v = *x[m];
            }
        }
        swap(x[i], x[m]);
    }
}
```

Find the pointer sorting code in [FXT: file `sort/sortptr.h`] and the pointer search routines in [FXT: file `sort/searchptr.h`].

9.5 Sorting by a supplied comparison function

The routines in [FXT: file `sort/sortfunc.h`] are similar to the C-quicksort `qsort` that is part of the standard library. A comparison function `cmp` has to be supplied by the called so that compound data types can be sorted with respect to some key contained. Citing the manual page for `qsort`:

```
The comparison function must return an integer less than, equal to, or greater than
zero if the first argument is considered to be respectively less than, equal to, or
greater than the second.  If two members compare  as  equal,  their  order  in  the
sorted array is undefined.
```

Note that the numerous calls to `cmp` do have a negative impact on the performance. And then with C++ you can provide a comparison 'function' for compound data by overloading the operators `<`, `<`, `<=` and `>=` and use the plain version. Back in performance land. Isn't C++ nice? TBD: *add a compile-time inlined version?*
As a prototypical example here the version of selection sort:

```
template <typename Type>
void selection_sort(Type *f, ulong n, int (*cmp)(const Type &, const Type &))
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(f[j],v) < 0 )
            {
                m = j;
                v = f[m];
            }
        }
        swap(f[i], f[m]);
    }
}
```

The rest of the supplied routines a rather straight forward translation of the (plain-) sort analogues, the function one will most likely use being

```
template <typename Type>
void quick_sort(Type *f, ulong n, int (*cmp)(const Type &, const Type &))
```

Sorting complex numbers

You want to sort complex numbers? Fine for me, but *don't* tell your local mathematician. To see the mathematical problem we ask whether i is smaller or greater than zero. Assume $i > 0$: follows $i \cdot i > 0$

(we multiplied with a positive value) which is $-1 > 0$ and that is false. So, is $i < 0$? Then $i \cdot i > 0$ (multiplication with a negative value, as assumed). So $-1 > 0$, oops! The lesson is that there is no way to impose an arrangement on the complex numbers that would justify the usage of the symbols $<$ and $>$ in the mathematical sense.

Nevertheless we can invent a relation that allows us to sort: arranging (sorting) the complex numbers according to their absolute value (modulus) leaves infinitely many numbers in one 'bucket', namely all those that have the same distance to zero. However, one could use the modulus as the *major* ordering parameter, the angle as the *minor*. Or the real part as the major and the imaginary part as the minor.

The latter is realized in

```
static inline int
cmp_complex(const Complex &f, const Complex &g)
{
    int  ret = 0;
    double fr = f.real();
    double gr = g.real();
    if ( fr==gr )
    {
        double fi = f.imag();
        double gi = g.imag();
        if ( fi!=gi )  ret = (fi>gi ? +1 : -1);
    }
    else    ret = (fr>gr ? +1 : -1);
    return  ret;
}
```

which, when used as comparison with the above function-sort as in

```
void complex_sort(Complex *f, ulong n)
// major order wrt. real part
// minor order wrt. imag part
{
    quick_sort(f, n, cmp_complex);
}
```

can indeed be the practical tool you had in mind.

9.6 Unique

This section presents a few utility functions that revolve around whether values in a (sorted) array are repeated or unique.

Testing whether all values are unique:

```
template  <typename Type>
int test_unique(const Type *f, ulong n)
// for a sorted array test whether all values are unique
//   (i.e. whether no value is repeated)
//
// returns 0 if all values are unique
// else returns index of the second element in the first pair found
//
// this function is not called "is_unique()" because it
// returns 0 (=="false") for a positive answer
{
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] == f[k-1] )  return  k;  // k != 0
    }
    return  0;
}
```

The same thing, but for inexact types (floats): the maximal (absolute) difference within which two contiguous elements will still be considered equal can be provided as additional parameter. One subtle

point is that the values can slowly 'drift away' unnoticed by this implementation: Consider a long array where each difference computed has the same sign and is just smaller than `da`, say it is $d = 0.6 \cdot$`da`. The difference of the first and last value then is $0.6 \cdot (n-1) \cdot d$ which is greater than `da` for $n \geq 3$.

```
template  <typename Type>
int test_unique_approx(const Type *f, ulong n, Type da)
// for a sorted array test whether all values are
// unique within some tolerance
//   (i.e. whether no value is repeated)
//
// returns 0 if all values are unique
// else returns index of the second element in the first pair found
//
// makes mostly sense with inexact types (float or double)
{
    if ( da<=0 )  da = -da;  // want positive tolerance

    for (ulong k=1; k<n; ++k)
    {
        Type d = (f[k] - f[k-1]);
        if ( d<=0 )  d = -d;

        if ( d < da )  return k;  // k != 0
    }

    return  0;
}
```

An alternative way to deal with inexact types is to apply

```
template  <typename Type>
void quantise(Type *f, ulong n, double q)
//
// in f[] set each element x to q*floor(1/q*(x+q/2))
// e.g.: q=1 ==> round to nearest integer
//       q=1/1000 ==> round to nearest multiple of 1/1000
// For inexact types (float or double)
{
    Type qh = q * 0.5;
    Type q1 = 1.0 / q;
    while ( n-- )
    {
        f[n] = q * floor( q1 * (f[n]+qh) );
    }
}
```

[FXT: `quantise` in `aux1/quantise.h`] before using `test_unique_approx`. One should use a quantization parameter `q` that is greater than the value used for `da`.

Minimalistic demo:

```
Random values:
   0:  0.9727750243
   1:  0.2925167845
   2:  0.7713576982
   3:  0.5267449795
   4:  0.7699138366
   5:  0.4002286223

Quantization with q=0.01

Quantised & sorted :
   0:  0.2900000000
   1:  0.4000000000
   2:  0.5300000000
   3:  0.7700000000
   4:  0.7700000000
   5:  0.9700000000
First REPEATED value at index 4  (and 3)

Unique'd array:
   0:  0.2900000000
   1:  0.4000000000
   2:  0.5300000000
   3:  0.7700000000
   4:  0.9700000000
```

`quantise()` turns out to be also useful in another context, cf. [FXT: `symbolify_by_size` and `symbolify_by_order` in `aux1/symbolify.h`].

Counting the elements that appear just once:

```
template  <typename Type>
int unique_count(const Type *f, ulong n)
// for a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
//   values is n - unique_count(f, n);
{
    if ( 1>=n )  return n;
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] != f[k-1] )  ++ct;
    }
    return  ct;
}
```

Removing repeated elements:

```
template  <typename Type>
ulong unique(Type *f, ulong n)
// for a sorted array squeeze all repeated values
// and return the number of unique values
// e.g.:  [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// the routine also works for unsorted arrays as long
//   as identical elements only appear in contiguous blocks
//   e.g. [4, 4, 3, 7, 7] --> [4, 3, 7]
// the order is preserved
{
    ulong u = unique_count(f, n);
    if ( u == n )  return n;  // nothing to do
    Type v = f[0];
    for (ulong j=1, k=1;  j<u;  ++j)
    {
        while ( f[k] == v )  ++k;  // search next different element
        v = f[j] = f[k];
    }
    return u;
}
```

9.7 Misc

A sequence is called *monotone* if it is either purely ascending or purely descending. This includes the case where subsequent elements are equal. Whether a constant sequence is considered ascending or descending in this context is a matter of convention.

```
template  <typename Type>
int is_monotone(const Type *f, ulong n)
// return
//   +1 for ascending order
//   -1 for descending order
//   else 0
{
    if ( 1>=n )  return  +1;
    ulong k;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }
    if ( k==n )  return +1;  // constant is considered ascending here
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
```

```
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }
    return  s;
}
```

A *strictly monotone* sequence is a monotone sequence that has no identical pairs of elements. The test turns out to be slightly easier:

```
template  <typename Type>
int is_strictly_monotone(const Type *f, ulong n)
// return
//   +1 for strictly ascending order
//   -1 for strictly descending order
//   else 0
{
    if ( 1>=n )  return  +1;
    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }
    return  s;
}
```

[FXT: file `sort/monotone.h`]

A sequence is called *convex* if it starts with an ascending part and ends with a descending part. A *concave* sequence starts with a descending and ends with an ascending part. Whether a monotone sequence is considered convex or concave again is a matter of convention (i.e. you have the choice to consider the first or the last element as extremum). Lacking a term that contains both convex and concave the following routine is called `is_convex`:

```
template <typename Type>
long is_convex(Type *f, ulong n)
//
// return
//   +val for convex sequence (first rising then falling)
//   -val for concave sequence (first falling then rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is monotone.
//
// note: a constant sequence is considered any of rising/falling
//
{
    if ( 1>=n )  return +1;
    ulong k = 1;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }
    if ( k==n )  return  +n;  // constant is considered convex here
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
```

```
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  break;
        s = -k;
    }

    if ( k==n )  return s;  // sequence is monotone

    // check that the ordering does not change again:
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }

    return  s;
}
```

The test for *strictly convex* (or concave) sequences is:

```
template <typename Type>
long is_strictly_convex(Type *f, ulong n)
//
// return
//   +val for strictly convex sequence
//        (i.e. first strictly rising then strictly falling)
//   -val for strictly concave sequence
//        (i.e. first strictly falling then strictly rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is strictly monotone.
//
{
    if ( 1>=n )  return +1;

    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  break;
        s = -k;
    }

    if ( k==n )  return s;  // sequence is monotone
    else  if ( f[k] == f[k-1] )  return 0;

    // check that the ordering does not change again:
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
```

```
    }
    return  s;
}
```

[FXT: file sort/convex.h]

The tests given are mostly useful as assertions used inside more complex algorithms.

9.8 Heap-sort

An alternative algorithm for sorting uses the heap data structure introduced in section 10.5 (p.218).

A heap can be sorted by swapping the first (and biggest) element with the last and 'repairing' the array of size $n-1$ by a call to `heapify1`. Applying this idea recursively until there is nothing more to sort leads to the routine

```
template <typename Type>
void heap_sort_ascending(Type *x, ulong n)
// sort an array that has the heap-property into ascending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        swap(p[1], p[k]);   // move largest to end of array
        --n;                // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

that needs time $O(n \log(n))$. That is, a call to

```
template <typename Type>
void heap_sort(Type *x, ulong n)
{
    build_heap(x, n);
    heap_sort_ascending(x, n);
}
```

will sort the array `x[]` into ascending order. Note that sorting into descending order is not any harder:

```
template <typename Type>
void heap_sort_decending(Type *x, ulong n)
// sort an array that has the heap-property into descending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        ++p;  --n;          // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

[FXT: file sort/heapsort.h]

Find a demo in [FXT: file demo/heapsort-demo.cc], its output in [FXT: file demo/heapsort-out.txt].

Chapter 10

Data structures

10.1 Stack (LIFO)

A *stack* (or LIFO for last-in, first-out) is a data structure that supports the operations: *push* to saves an entry and *pop* to retrieve and remove the entry that was entered last. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it. Similarly, *poke* modifies the last entry.

An implementation with the option to let the stack grow when necessary can be found in [FXT: file ds/stack.h]

```
template <typename Type>
class stack
{
public:
    Type  *x_;  // data
    ulong  s_;  // size
    ulong  p_;  // stack pointer (position of next write), top entry @ p-1
    ulong  gq_; // grow gq elements if necessary, 0 for "don't grow"
public:
    stack(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        p_ = 0;  // stack is empty
        gq_ = growq;
    }

    ~stack()  { delete [] x_; }
private:
    stack & operator = (const stack &);  // forbidden
public:
    ulong n()  const
    // return number of entries
    {
        return p_;
    }

    ulong push(Type z)
    // return size of stack, zero on stack overflow
    // if gq_ is non-zero the stack grows
    {
        if ( p_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow

            ulong ns = s_ + gq_;  // new size
            Type *nx = new Type[ns];
            copy(x_, nx, s_);
            delete [] x_;  x_ = nx;
            s_ = ns;
        }
```

```
        x_[p_] = z;
        ++p_;
        return  s_;
    }
    ulong pop(Type &z)
    // read top entry and remove it
    // return number of entries at function start
    // if empty return zero is returned and z is undefined
    {
        ulong ret = p_;
        if ( 0!=p_ )  { --p_;  z = x_[p_]; }
        return  ret;
    }
    ulong poke(Type z)
    // modify top entry
    // return number of entries
    // if empty return zero is returned and nothing done
    {
        if ( 0!=p_ )  x_[p_-1] = z;
        return p_;
    }
    ulong peek(Type &z)
    // read top entry (without removing it)
    // return number of entries
    // if empty zero is returned z is undefined
    {
        if ( 0!=p_ )  z = x_[p_-1];
        return p_;
    }
};
```

Its working is demonstrated in [FXT: file `demo/stack-demo.cc`]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) looks like:

```
push( 1)  1  -  -  -   #=1
push( 2)  1  2  -  -   #=2
push( 3)  1  2  3  -   #=3
push( 4)  1  2  3  4   #=4
push( 5)  1  2  3  4  5  -  -  -   #=5
push( 6)  1  2  3  4  5  6  -  -   #=6
push( 7)  1  2  3  4  5  6  7  -   #=7
pop== 7   1  2  3  4  5  6  -  -   #=6
pop== 6   1  2  3  4  5  -  -  -   #=5
push( 8)  1  2  3  4  5  8  -  -   #=6
pop== 8   1  2  3  4  5  -  -  -   #=5
pop== 5   1  2  3  4  -  -  -  -   #=4
push( 9)  1  2  3  4  9  -  -  -   #=5
pop== 9   1  2  3  4  -  -  -  -   #=4
pop== 4   1  2  3  -  -  -  -  -   #=3
push(10)  1  2  3 10  -  -  -  -   #=4
pop==10   1  2  3  -  -  -  -  -   #=3
pop== 3   1  2  -  -  -  -  -  -   #=2
push(11)  1  2 11  -  -  -  -  -   #=3
pop==11   1  2  -  -  -  -  -  -   #=2
pop== 2   1  -  -  -  -  -  -  -   #=1
push(12)  1 12  -  -  -  -  -  -   #=2
pop==12   1  -  -  -  -  -  -  -   #=1
pop== 1   -  -  -  -  -  -  -  -   #=0
push(13) 13  -  -  -  -  -  -  -   #=1
pop==13   -  -  -  -  -  -  -  -   #=0
pop== 0   -  -  -  -  -  -  -  -   #=0
  (stack was empty)
push(14) 14  -  -  -  -  -  -  -   #=1
pop==14   -  -  -  -  -  -  -  -   #=0
pop== 0   -  -  -  -  -  -  -  -   #=0
  (stack was empty)
push(15) 15  -  -  -  -  -  -  -   #=1
```

10.2 Ring buffer

A *ring buffer* is a array plus read- and write operations that wrap around. That is, if the last position of the array is reached writing continues at the begin of the array, thereby erasing the oldest entries. The read operation should start at the oldest entry in the array.

The `ringbuffer` utility class in [FXT: `class ringbuffer` in `ds/ringbuffer.h`] is as simple as:

```
template <typename Type>
class ringbuffer
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong fpos_;  // first position to read in buffer
public:
    explicit ringbuffer(ulong n)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        wpos_ = 0;
        fpos_ = 0;
    }
    ~ringbuffer()  { delete [] x_; }
    ulong n()  const  { return n_; }
    void insert(const Type &z)
    {
        x_[wpos_] = z;
        if ( ++wpos_>=s_ )  wpos_ = 0;
        if ( n_ < s_ )  ++n_;
        else  fpos_ =  wpos_;
    }
    ulong read(ulong n, Type &z)  const
    // read entry n (that is, [(fpos_ + n)%s_])
    // return 0 if entry #n is last in buffer
    // else return n+1
    {
        if ( n>=n_ )  return 0;
        ulong j = fpos_ + n;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  n + 1;
    }
};
```

Reading from the ring buffer goes like:

```
    ulong k = 0;
    Type z;  // type of entry
    while ( (k = f.read(k, z)) )
    {
        // do something with z
    }
```

A demo is in [FXT: file `demo/ringbuffer-demo.cc`], its output is

```
insert( 1)  1   #=1   r=1   w=0
insert( 2)  1  2   #=2   r=2   w=0
insert( 3)  1  2  3   #=3   r=3   w=0
insert( 4)  1  2  3  4   #=4   r=0   w=0
insert( 5)  2  3  4  5   #=4   r=1   w=1
insert( 6)  3  4  5  6   #=4   r=2   w=2
insert( 7)  4  5  6  7   #=4   r=3   w=3
insert( 8)  5  6  7  8   #=4   r=0   w=0
insert( 9)  6  7  8  9   #=4   r=1   w=1
```

Ring buffers can be useful for storing a constant amount of history-data such as for logging purposes. For that one would enhance the `ringbuffer` class so that it uses an additional array of (fixed width) strings. The message to log would be copied into the array and the pointer set accordingly. Read should then just return the pointer to the string.

10.3 Queue (FIFO)

A *queue* (or LIFO for first-in, first-out) is a data structure that supports two operations: *push* saves an entry and *pop* retrieves (and removes) the entry that was entered the longest time ago. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it.

A utility class with the optional feature of growing if necessary is [FXT: `class queue` in `ds/queue.h`]:

```
template <typename Type>
class queue
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong rpos_;  // next position to read in buffer
    ulong gq_;  // grow gq elements if necessary, 0 for "don't grow"
public:
    explicit queue(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        wpos_ = 0;
        rpos_ = 0;
        gq_ = growq;
    }

    ~queue()  { delete [] x_; }

    ulong n()  const  { return n_; }

    ulong push(const Type &z)
    // returns number of entries
    // zero is returned on failure
    //   (i.e. space exhausted and 0==gq_)
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // growing disabled

            ulong ns = s_ + gq_;  // new size
            Type *nx = new Type[ns];

            // move read-position to zero:
            rotate_left(x_, s_, rpos_);
            wpos_ = s_;
            rpos_ = 0;

            copy(x_, nx, s_);

            delete [] x_;  x_ = nx;
            s_ = ns;
        }

        x_[wpos_] = z;
        ++wpos_;
        if ( wpos_>=s_ )  wpos_ = 0;

        ++n_;
        return n_;
    }

    ulong peek(Type &z)
    // returns number of entries
    // if zero is returned the value of z is undefined
    {
        z = x_[rpos_];
        return n_;
```

```
    }
    ulong pop(Type &z)
    // returns number of entries before pop
    // i.e. zero is returned if queue was empty
    // if zero is returned the value of z is undefined
    {
        ulong ret = n_;
        if ( 0!=n_ )
        {
            z = x_[rpos_];
            ++rpos_;
            if ( rpos_ >= s_ )  rpos_ = 0;
            --n_;
        }
        return ret;
    }
};
```

Its working is demonstrated in [FXT: file `demo/queue-demo.cc`]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) looks like:

```
push( 1)  1  -  -  -   #=1   r=0   w=1
push( 2)  1  2  -  -   #=2   r=0   w=2
push( 3)  1  2  3  -   #=3   r=0   w=3
push( 4)  1  2  3  4   #=4   r=0   w=0
push( 5)  1  2  3  4  5  -  -  -   #=5   r=0   w=5
push( 6)  1  2  3  4  5  6  -  -   #=6   r=0   w=6
push( 7)  1  2  3  4  5  6  7  -   #=7   r=0   w=7
pop== 1   -  2  3  4  5  6  7  -   #=6   r=1   w=7
pop== 2   -  -  3  4  5  6  7  -   #=5   r=2   w=7
push( 8)  -  -  3  4  5  6  7  8   #=6   r=2   w=0
pop== 3   -  -  -  4  5  6  7  8   #=5   r=3   w=0
pop== 4   -  -  -  -  5  6  7  8   #=4   r=4   w=0
push( 9)  9  -  -  -  5  6  7  8   #=5   r=4   w=1
pop== 5   9  -  -  -  -  6  7  8   #=4   r=5   w=1
pop== 6   9  -  -  -  -  -  7  8   #=3   r=6   w=1
push(10)  9 10  -  -  -  -  7  8   #=4   r=6   w=2
pop== 7   9 10  -  -  -  -  -  8   #=3   r=7   w=2
pop== 8   9 10  -  -  -  -  -  -   #=2   r=0   w=2
push(11)  9 10 11  -  -  -  -  -   #=3   r=0   w=3
pop== 9   - 10 11  -  -  -  -  -   #=2   r=1   w=3
pop==10   -  - 11  -  -  -  -  -   #=1   r=2   w=3
push(12)  -  - 11 12  -  -  -  -   #=2   r=2   w=4
pop==11   -  -  - 12  -  -  -  -   #=1   r=3   w=4
pop==12   -  -  -  -  -  -  -  -   #=0   r=4   w=4
push(13)  -  -  -  - 13  -  -  -   #=1   r=4   w=5
pop==13   -  -  -  -  -  -  -  -   #=0   r=5   w=5
pop== 0   -  -  -  -  -  -  -  -   #=0   r=5   w=5
  (queue was empty)
push(14)  -  -  -  -  - 14  -  -   #=1   r=5   w=6
pop==14   -  -  -  -  -  -  -  -   #=0   r=6   w=6
pop== 0   -  -  -  -  -  -  -  -   #=0   r=6   w=6
  (queue was empty)
push(15)  -  -  -  -  -  - 15  -   #=1   r=6   w=7
```

This is [FXT: file `demo/queue-out.txt`]. You might want to compare this to the stack-demo at page 212.

10.4 Deque (double-ended queue)

A *deque* (for *double-ended queue*) combines the data structures stack and queue: insertion and deletion is possible both at the first- and the last postion, all in time-$O(1)$. An implementation with the option to let the stack grow when necessary can be found in [FXT: file `ds/deque.h`]

```
template <typename Type>
class deque
{
public:
```

```
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong fpos_;  // position of first element in buffer
    // insert_first() will write to (fpos-1)%n
    ulong lpos_;  // position of last element in buffer plus one
    // insert_last() will write to  lpos,  n==(lpos-fpos) (mod s)
    // entries are at [fpos, ... , lpos-1]  (range may be empty)
    ulong  gq_; // grow gq elements if necessary, 0 for "don't grow"
public:
    explicit deque(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        fpos_ = 0;
        lpos_ = 0;
        gq_ = growq;
    }
    ~deque()  { delete [] x_; }
    ulong n()  const  { return n_; }
    ulong insert_first(const Type &z)
    // returns number of entries after insertion
    // zero is returned on failure
    //   (i.e. space exhausted and 0==gq_)
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // growing disabled
            grow();
        }
        --fpos_;
        if ( fpos_ == -1UL )  fpos_ = s_ - 1;
        x_[fpos_] = z;
        ++n_;
        return  n_;
    }

    ulong insert_last(const Type &z)
    // returns number of entries after insertion
    // zero is returned on failure
    //   (i.e. space exhausted and 0==gq_)
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // growing disabled
            grow();
        }
        x_[lpos_] = z;
        ++lpos_;
        if ( lpos_>=s_ )  lpos_ = 0;
        ++n_;
        return  n_;
    }

    ulong extract_first(Type & z)
    // return number of elements before extract
    // return 0 if extract on empty deque was attempted
    {
        if ( 0==n_ )  return 0;
        z = x_[fpos_];
        ++fpos_;
        if ( fpos_ >= s_ )  fpos_ = 0;
        --n_;
        return  n_ + 1;
    }
    ulong extract_last(Type & z)
    // return number of elements before extract
    // return 0 if extract on empty deque was attempted
    {
```

```
        if ( 0==n_ )  return 0;
        --lpos_;
        if ( lpos_ == -1UL )  lpos_ = s_ - 1;
        z = x_[lpos_];
        --n_;
        return  n_ + 1;
    }

    ulong read_first(Type & z)  const
    // read (but don't remove) first entry
    // return number of elements (i.e. on error return zero)
    {
        if ( 0==n_ )  return 0;
        z = x_[fpos_];
        return n_;
    }

    ulong read_last(Type & z)  const
    // read (but don't remove) last entry
    // return number of elements (i.e. on error return zero)
    {
        return  read(n_-1, z);  // ok for n_==0
    }

    ulong read(ulong k, Type & z)  const
    // read entry k (that is, [(fpos_ + k)%s_])
    // return 0 if k>=n_
    // else return k+1
    {
        if ( k>=n_ )  return 0;
        ulong j = fpos_ + k;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  k + 1;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        Type *nx = new Type[ns];

        // move read-position to zero:
        // rotate_left(x_, s_, fpos_);  copy(x_, nx, s_);
        copy_cyclic(x_, nx, s_, fpos_);
        fpos_ = 0;
        lpos_ = n_;

        delete [] x_;  x_ = nx;
        s_ = ns;
    }
};
```

Its working is demonstrated in [FXT: file `demo/deque-out.txt`]:

```
insert_first( 1)      1
insert_last(51)       1 51
insert_first( 2)      2  1 51
insert_last(52)       2  1 51 52
insert_first( 3)      3  2  1 51 52
insert_last(53)       3  2  1 51 52 53
insert_first( 4)      4  3  2  1 51 52 53
insert_last(54)       4  3  2  1 51 52 53 54
extract_first()= 4    3  2  1 51 52 53 54
extract_last() =54    3  2  1 51 52 53
insert_first( 5)      5  3  2  1 51 52 53
insert_last(55)       5  3  2  1 51 52 53 55
extract_first()= 5    3  2  1 51 52 53 55
extract_last() =55    3  2  1 51 52 53
extract_first()= 3    2  1 51 52 53
extract_last() =53    2  1 51 52
extract_first()= 2    1 51 52
extract_last() =52    1 51
insert_first( 6)      6  1 51
insert_last(56)       6  1 51 56
extract_first()= 6    1 51 56
extract_last() =56    1 51
extract_first()= 1   51
```

```
extract_last() =51
 (dequeue empty)
 (dequeue empty)
insert_first( 7)      7
insert_last(57)       7 57
extract_first()= 7   57
extract_last() =57
 (dequeue empty)
 (dequeue empty)
insert_first( 9)      9
insert_last(59)       9 59
```

10.5 Heap and priority queue

A *heap* is a binary tree where the left and right children are smaller or equal than their parent node. The function

```
template <typename Type>
ulong test_heap(const Type *x, ulong n)
// return 0 if x[] has heap property
// else index of node found to be bigger than its parent
{
    const Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        ulong t = (k>>1); // parent(k)
        if ( p[t]<p[k] )  return k;
    }
    return 0;  // has heap property
}
```

finds out whether a given array has the described heap-property.

It turns out that a unordered array of size n can be reordered to a heap in $O(n)$ time[1]. The routine

```
template <typename Type>
void build_heap(Type *x, ulong n)
// reorder data to a heap
{
    for (ulong j=(n>>1); j>0; --j)  heapify1(x-1, n, j);
}
```

does the trick. It uses the subroutine

```
template <typename Type>
void heapify1(Type *z, ulong n, ulong k)
// subject to the condition that the trees below the children of node
// k are heaps, move the element z[k] (down) until the tree below node
// k is a heap.
//
// data expected in z[1..n] (!)
{
    ulong m = k;
    ulong l = (k<<1); // left(k);
    if ( (l <= n) && (z[l] > z[k]) )  m = l;
    ulong r = (k<<1) + 1; // right(k);
    if ( (r <= n) && (z[r] > z[m]) )  m = r;
    if ( m != k )
    {
        swap(z[k], z[m]);
        heapify1(z, n, m);
    }
}
```

That runs in time $O(\log(n))$.

[1] This is slightly non-obvious, see [21] (p.145) for an explanation.

Heaps are useful to build a so-called *priority queue*. This is a data structure that supports insertion of an element and extraction of the maximal element it contains both in an efficient manner. A priority queue can be used to 'schedule' a certain event for a given point in time and return the next pending event.

A new element can be inserted into a heap in $O(\log(n))$ time by appending it and moving it towards the root (that is, the first element) as necessary:

```
bool heap_insert(Type *x, ulong n, ulong s, Type t)
// with x[] a heap of current size n
// and max size s (i.e. space for s elements allocated)
// insert t and restore heap-property.
//
// Return true if successful
// else (i.e. space exhausted) false
//
// Takes time \log(n)
{
    if ( n > s )  return false;
    ++n;
    Type *x1 = x - 1;
    ulong j = n;
    while ( j > 1 )
    {
        ulong k = (j>>1);  // k==parent(j)
        if ( x1[k] >= t )  break;
        x1[j] = x1[k];
        j = k;
    }
    x1[j] = t;
    return true;
}
```

Similarly, the maximal element can be removed in time $O(\log(n))$:

```
template <typename Type>
Type heap_extract_max(Type *x, ulong n)
// Return maximal element of heap and
// restore heap structure.
// Return value is undefined for 0==n
{
    Type m = x[0];
    if ( 0 != n )
    {
        Type *x1 = x - 1;
        x1[1] = x1[n];
        --n;
        heapify1(x1, n, 1);
    }
    // else error
    return m;
}
```

Two modifications seem appropriate: First, replacement of `extract_max()` by a `extract_next()`, leaving it as an (compile time) option whether to extract the minimal or the maximal element. This is achieved by changing the comparison operators at a few strategic places so that the heap is built either as described above or with its minimum as first element:

```
#if  1
// next() is the one with the smallest key
// i.e.  extract_next()  is  extract_min()
#define _CMP_ <
#define _CMPEQ_ <=
#else
// next() is the one with the biggest key
// i.e.  extract_next()  is  extract_max()
#define _CMP_ >
#define _CMPEQ_ >=
#endif
```

Second, augmenting the elements used by a event-description that can be freely defined. The resulting utility class is

```
template <typename Type1, typename Type2>
```

```
class priority_queue
{
public:
    Type1 *t_;  // time:   t[0..s-1]
    Type2 *e_;  // events: e[0..s-1]
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of events
    ulong gq_;  // grow gq elements if necessary, 0 for "don't grow"
public:
    priority_queue(ulong n, ulong growq=0)
    {
        s_ = n;
        t_ = new Type1[s_];
        e_ = new Type2[s_];
        n_ = 0;
        gq_ = growq;
    }
    ~priority_queue()  { delete [] t_;  delete [] e_; }
    ulong n()  const  { return n_; }
    Type1 get_next(Type2 &e)  const
    // No check if empty
    {
        e = e_[0];
        return t_[0];
    }
    void heapify1(Type1 *t1, ulong n, ulong k, Type2 *e1)
    // events in         e1[1..n]
    // time of events in t1[1..n]
    {
        ulong m = k;
        ulong l = (k<<1); // left(k);
        if ( (l <= n) && (t1[l] _CMP_ t1[k]) )  m = l;
        ulong r = (k<<1) + 1; // right(k);
        if ( (r <= n) && (t1[r] _CMP_ t1[m]) )  m = r;
        if ( m != k )
        {
            swap(t1[k], t1[m]);  swap(e1[k], e1[m]);
            heapify1(t1, n, m, e1);
        }
    }
    Type1 extract_next(Type2 &e)
    {
        Type1 m = get_next(e);
        if ( 0 != n_ )
        {
            Type1 *t1 = t_ - 1;
            Type2 *e1 = e_ - 1;
            t1[1] = t1[n_];  e1[1] = e1[n_];
            --n_;
            heapify1(t1, n_, 1, e1);
        }
        // else error
        return m;
    }
    bool insert(const Type1 &t, const Type2 &e)
    // Insert event e at time t
    // Return true if successful
    // else (i.e. space exhausted and 0==gq_) false
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return false;  // growing disabled
            ulong ns = s_ + gq_;  // new size
            Type1 *nt = new Type1[ns];
            copy(t_, nt, s_);
            Type2 *ne = new Type2[ns];
            copy(e_, ne, s_);
            delete [] t_;  t_ = nt;
            delete [] e_;  e_ = ne;
            s_ = ns;
        }
```

```
        ++n_;
        Type1 *t1 = t_ - 1;
        Type2 *e1 = e_ - 1;
        ulong j = n_;
        while ( j > 1 )
        {
            ulong k = (j>>1);  // k==parent(j)
            if ( t1[k] _CMPEQ_ t )  break;
            t1[j] = t1[k];  e1[j] = e1[k];
            j = k;
        }
        t1[j] = t;
        e1[j] = e;
        return true;
    }
};
```

The second argument of the constructor determines the number of elements added in case of growth, it is disabled (equals zero) by default. [FXT: `class priority_queue` in `ds/priorityqueue.h`]

A demo that inserts events at random times $0 \le t < 1$ and for the sake of simplicity uses the negative time values as 'events' gives the output

```
 inserting into piority_queue:...
# :        event  @        time
 0:    -0.840188  @    0.840188
 1:    -0.394383  @    0.394383
 2:    -0.783099  @    0.783099
 3:     -0.79844  @     0.79844
 4:    -0.911647  @    0.911647
 5:    -0.197551  @    0.197551
 6:    -0.335223  @    0.335223
 7:     -0.76823  @     0.76823
 8:    -0.277775  @    0.277775
 9:     -0.55397  @     0.55397
 extracting from piority_queue:...
# :        event  @        time
10:    -0.197551  @    0.197551
 9:    -0.277775  @    0.277775
 8:    -0.335223  @    0.335223
 7:    -0.394383  @    0.394383
 6:     -0.55397  @     0.55397
 5:     -0.76823  @     0.76823
 4:    -0.783099  @    0.783099
 3:     -0.79844  @     0.79844
 2:    -0.840188  @    0.840188
 1:    -0.911647  @    0.911647
```

Here all events were inserted, then all were extracted, which does not quite reflect the typical use where the inserts and extracts occur intermixed. [FXT: file `demo/priorityqueue-demo.cc`]

10.6 Bit-array

The use of *bit-arrays* should be obvious: an array of tag values (like 'seen' versus 'unseen') where all standard data types would be a waste of space. Besides reading and writing individual bits one should implement a convenient search for the next set (or cleared) bit.

In FXT the class ([FXT: `class bitarray` in `ds/bitarray.h`]) is used for example for lists of small primes ([FXT: file `mod/primes.cc`]), in transpositions ([FXT: `transpose_ba` in `aux2/transpose_ba.h`] and [FXT: `transpose2_ba` in `aux2/transpose2_ba.h`]) and several operations on permutations. The public methods are

```
// operations on bit n:
    ulong test(ulong n)  const // test whether n-th bit set
    void set(ulong n)          // set n-th bit
    void clear(ulong n)        // clear n-th bit
    void change(ulong n)       // toggle n-th bit
    ulong test_set(ulong n)    // test whether n-th bit is set and set it
    ulong test_clear(ulong n)  // test whether n-th bit is set and clear it
    ulong test_change(ulong n) // test whether n-th bit is set and toggle it
// operations on all bits:
```

```
    void clear_all()           // clear all bits
    void set_all()             // set all bits
    int all_set_q()  const;    // return whether all bits are set
    int all_clear_q()  const;  // return whether all bits are clear
// scanning the array:
    // Note: the given index n is included in the search
    ulong next_set_idx(ulong n)  const   // return index of next set or value beyond end
    ulong next_clear_idx(ulong n)  const // return index of next clear or value beyond end
```

Combined operations like 'test-and-set-bit', 'test-and-clear-bit', 'test-and-change-bit' are often needed in applications that use bit-arrays. This is underlined by the fact that modern CPUs typically have instructions implementing these operations.

On the x86 architecture the corresponding CPU instructions as

```
static inline ulong asm_bts(ulong *f, ulong i)
// Bit Test and Set
{
    ulong ret;
    asm ( "btsl %2, %1 \n"
          "sbbl %0, %0"
          : "=r" (ret)
          : "m" (*f), "r" (i) );
    return ret;
}
```

(cf. [FXT: file auxbit/bitasm.h]) are used. If no specialized CPU instructions are available macros as

```
#define  DIVMOD_TEST(n, d, bm) \
ulong d = n / BITS_PER_LONG; \
ulong bm = 1UL << (n % BITS_PER_LONG); \
ulong t = bm & f_[d];
```

are used, performance is still good with these (the compiler of course replaces the '%' by the corresponding bit-and with BITS_PER_LONG-1 and the '/' by a right shift by log_2(BITS_PER_LONG) bits).

The class does not supply overloading of the array-index operator [] because the writing variant would cause a performance penalty.

One might want to add 'sparse'-versions of the scan functions for large bit-arrays with only few bits set or unset.

10.7 Resizable array

A resizable array is a container utility that, besides the array itself, uses a minimal (constant) amount of meta-data. Thereby it is useful in situations where linked lists might come to mind. Linked lists have the disadvantage of the additional data (next- and previous -pointers) proportional to the total size together with the inherent book keeping. Moreover, linked lists that use a call to the systems memory-allocator for the insertion of additional entries often suffer from severe performance problems.

Here we go [FXT: class rarray in ds/rarray.h]:

```
template <typename Type>
class rarray
// array that grows in adjustable steps when necessary
// rarray := "Resizing array"
//
// all operations maintain the relative order between elements
{
public:
    Type  *x_;  // data
    ulong  s_;  // size
    ulong  n_;  // position of next write, top entry @ n-1
    ulong  gq_; // grow gq elements if necessary, 0 for "don't grow"
```

```
public:
    explicit rarray(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;  // rarray is empty
        gq_ = growq;
    }
    ~rarray()  { delete [] x_; }

private:
    rarray & operator = (const rarray &);  // forbidden

public:
    Type * data()  { return x_; }
    ulong n()  const
    // Return number of entries
    { return n_; }
    ulong size()  const
    // Return number of allocated entries
    { return s_; }

    ulong append(const Type & z)
    // Return size of rarray, zero on rarray overflow
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }
        x_[n_] = z;
        ++n_;
        return  s_;
    }
    ulong remove_last()
    {
        ulong ret = n_;
        if ( n_!=0 )  --n_;
        return  ret;
    }
    ulong prepend(const Type & z)
    // Return size of rarray, zero on rarray overflow
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }
        for (ulong k=n_; k!=0; --k)  x_[k] = x_[k-1];  // shift right
        x_[0] = z;
        ++n_;
        return  s_;
    }
    ulong remove_first()
    // returns how many elements were there before remove
    // zero indicates error
    {
        ulong ret = n_;
        if ( n_!=0 )  --n_;
        for (ulong k=0; k<n_; ++k)  x_[k] = x_[k+1];  // shift left
        return  ret;
    }

    ulong search(const Type& x, ulong k=0)  const
    {
        for (  ; k<n_; ++k)  if ( x_[k]==x )  return k;
        return  s_;
    }
```

```
    void sort()  { ::quick_sort(x_, n_); }
    void uniq()  { sort();  ::uniq(x_, n_); }

    ulong insert_at(const Type & v, ulong j)
    // Return size of rarray
    // Return zero on error: (rarray overflow or access beyond range)
    // (i.e. no action unless  0<=j<=n_)
    {
        if ( j >= n_ )  return 0;  // beyond bounds

        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }

        for (ulong k=n_; k!=j; --k)  x_[k] = x_[k-1];
        x_[j] = v;
        ++n_;

        return  s_;
    }

    ulong remove_at(ulong j)
    // returns how many elements were there before remove
    // zero indicates error
    {
        ulong ret = n_;
        if ( j>=n_ )  return 0;
        --n_;
        for (ulong k=j; k<n_; ++k)  x_[k] = x_[k+1];
        return ret;
    }

private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        Type *nx = new Type[ns];
        copy(x_, nx, s_);
        delete [] x_;
        x_ = nx;
        s_ = ns;
    }
};
```

Note that while the operations `append()` and `remove_last()` are $O(1)$, the equivalent tasks manipulation the start of the array, `prepend()` and `remove_first()`, are $O(n)$ where n is the number of entries.

A simple demo in [FXT: file `demo/rarray-demo.cc`] prints:

```
append ( 1)     1  -  -  -   #=1
prepend( 2)     2  1  -  -   #=2
append ( 3)     2  1  3  -   #=3
prepend( 4)     4  2  1  3   #=4
append ( 5)     4  2  1  3  5  -  -  -   #=5
prepend( 6)     6  4  2  1  3  5  -  -   #=6
append ( 7)     6  4  2  1  3  5  7  -   #=7
remove_last     6  4  2  1  3  5  -  -   #=6
remove_first    4  2  1  3  5  -  -  -   #=5
prepend( 8)     8  4  2  1  3  5  -  -   #=6
remove_last     8  4  2  1  3  -  -  -   #=5
remove_first    4  2  1  3  -  -  -  -   #=4
append ( 9)     4  2  1  3  9  -  -  -   #=5
remove_last     4  2  1  3  -  -  -  -   #=4
remove_first    2  1  3  -  -  -  -  -   #=3
prepend(10)    10  2  1  3  -  -  -  -   #=4
remove_last    10  2  1  -  -  -  -  -   #=3
remove_first    2  1  -  -  -  -  -  -   #=2
append (11)     2  1 11  -  -  -  -  -   #=3
remove_last     2  1  -  -  -  -  -  -   #=2
remove_first    1  -  -  -  -  -  -  -   #=1
prepend(12)    12  1  -  -  -  -  -  -   #=2
remove_last    12  -  -  -  -  -  -  -   #=1
remove_first    -  -  -  -  -  -  -  -   #=0
append (13)    13  -  -  -  -  -  -  -   #=1
remove_last     -  -  -  -  -  -  -  -   #=0
```

```
remove_first    -  -  -  -  -  -  -  -   #=0
  (rarray was empty)
prepend(14)    14  -  -  -  -  -  -  -   #=1
remove_last     -  -  -  -  -  -  -  -   #=0
remove_first    -  -  -  -  -  -  -  -   #=0
  (rarray was empty)
append (15)    15  -  -  -  -  -  -  -   #=1
```

10.8 Ordered resizable array

Sometimes it is desirable to maintain the order of elements in an array during insertion and deletion.

The utility class providing the operations on an ordered (resizable) array is [FXT: `class ordered_rarray` in `ds/orderedrarray.h`]:

```
template <typename Type>
class ordered_rarray : private rarray<Type>
// rarray that maintains ascending order of elements
//
// private inheritance to hide those functions that
// might destroy the order
{
public:
    explicit ordered_rarray(ulong n, ulong growq=0)
        : rarray<Type>(n, growq)
    {;}

    ~ordered_rarray()  {;}

private:
    ordered_rarray & operator = (const ordered_rarray &);  // forbidden

public:
    using rarray<Type>::n;
    using rarray<Type>::data;
    using rarray<Type>::size;
    using rarray<Type>::remove_last;
    using rarray<Type>::remove_first;
    using rarray<Type>::remove_at;

    ulong search(const Type & v)  const
    // Return index of first element in that is == v
    // Return ~0 if there is no such element
    { return ::bsearch(x_, n_, v); }

    ulong search_ge(const Type & v)  const
    // Return index of first element in that is >= v
    // Return ~0 if there is no such element
    { return ::bsearch_ge<Type>(x_, n_, v); }

    ulong insert(const Type & v)
    // Insert v so that ascending order is kept.
    // Return size, zero on rarray overflow.
    {
        if ( 0==n_ )  { return append(v); }

        ulong j = search_ge(v);
        if ( j>=n_ )  return  append(v);
        else          return  rarray<Type>::insert_at(v, j);
    }

    ulong insert_uniq(const Type & v)
    // Insert v so that ascending order is kept.
    // Don't insert if element already in array.
    // Return size, zero on rarray overflow.
    {
        if ( 0==n_ )  { return append(v); }

        ulong j = search_ge(v);
        if ( j>=n_ )  return  append(v);
        else
        {
            if ( x_[j] = v )  return 0;
            return  rarray<Type>::insert_at(v, j);
```

```
        }
    }
};
```

If `insert_uniq()` is used exclusively for insertion, the class can be used for ordered sets without repetitions.
The demo in [FXT: file `demo/orderedrarray-demo.cc`] prints:

```
insert( 0)    0  -  -  -   #=1
insert( 0)    0  0  -  -   #=2
insert( 2)    0  0  2  -   #=3
insert( 1)    0  0  1  2   #=4
insert( 3)    0  0  1  2  3  -  -  -   #=5
insert( 2)    0  0  1  2  2  3  -  -   #=6
insert( 5)    0  0  1  2  2  3  5  -   #=7
remove @ 3    0  0  1  2  3  5  -  -   #=6
remove @ 3    0  0  1  3  5  -  -  -   #=5
insert( 2)    0  0  1  2  3  5  -  -   #=6
remove @ 3    0  0  1  3  5  -  -  -   #=5
remove @ 2    0  0  3  5  -  -  -  -   #=4
insert( 7)    0  0  3  5  7  -  -  -   #=5
remove @ 2    0  0  5  7  -  -  -  -   #=4
remove @ 2    0  0  7  -  -  -  -  -   #=3
insert( 3)    0  0  3  7  -  -  -  -   #=4
remove @ 2    0  0  7  -  -  -  -  -   #=3
remove @ 1    0  7  -  -  -  -  -  -   #=2
insert( 8)    0  7  8  -  -  -  -  -   #=3
remove @ 1    0  8  -  -  -  -  -  -   #=2
remove @ 1    0  -  -  -  -  -  -  -   #=1
insert( 4)    0  4  -  -  -  -  -  -   #=2
remove @ 1    0  -  -  -  -  -  -  -   #=1
remove @ 0    -  -  -  -  -  -  -  -   #=0
insert(10)   10  -  -  -  -  -  -  -   #=1
remove @ 0    -  -  -  -  -  -  -  -   #=0
remove @ 0    -  -  -  -  -  -  -  -   #=0
  (ordered_rarray was empty)
insert( 4)    4  -  -  -  -  -  -  -   #=1
remove @ 0    -  -  -  -  -  -  -  -   #=0
remove @ 0    -  -  -  -  -  -  -  -   #=0
  (ordered_rarray was empty)
insert(12)   12  -  -  -  -  -  -  -   #=1
```

10.9 Resizable set

A data structure for sets that, for the sake of $O(1)$ deletion, does not maintain the relative order of its entries can be implemented as [FXT: `class rset` in `ds/rset.h`]

```
template <typename Type>
class rset
// set that grows in adjustable steps when necessary
// rset := "Resizing set"
{
public:
    Type  *x_;  // data
    ulong  s_;  // size
    ulong  n_;  // position of next write, top entry @ n-1
    ulong  gq_; // grow gq elements if necessary, 0 for "don't grow"
public:
    explicit rset(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;  // rset is empty
        gq_ = growq;
    }
    ~rset()  { delete [] x_; }

private:
    rset & operator = (const rset &);  // forbidden

public:
    Type * data()  { return x_; }
    ulong n()  const
```

```
    // Return number of entries
    { return n_; }

    ulong size()  const
    // Return number of allocated entries
    { return s_; }

    ulong insert(const Type & z)
    // Insert element v (append v).
    // Return size of rset, zero on rset overflow
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }

        x_[n_] = z;
        ++n_;

        return  s_;
    }

    ulong search(const Type & x, ulong k=0)  const
    {
        for (  ; k<n_; ++k)  if ( x_[k]==x )  return k;
        return  s_;
    }

    void sort()  { ::quick_sort(x_, n_); }
    void uniq()  { sort();  ::uniq(x_, n_); }

    ulong remove_at(ulong j)
    // returns how many elements were there before remove
    // zero indicates error
    {
        ulong ret = n_;
        if ( j>=n_ )  return 0;
        --n_;
        x_[j] = x_[n_];
        return ret;
    }

    ulong remove(const Type & z)
    {
        ulong j = search(z);
        if ( j<n_ )  remove_at(j);
        return j;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        Type *nx = new Type[ns];
        copy(x_, nx, s_);
        delete [] x_;
        x_ = nx;
        s_ = ns;
    }
};
```

The output of [FXT: file `demo/rset-demo.cc`] demonstrates how deleted elements are simply swapped with the last entry:

```
insert (11)     11  -  -  -   #=1
insert ( 2)     11  2  -  -   #=2
insert (13)     11  2 13  -   #=3
insert ( 4)     11  2 13  4   #=4
insert (15)     11  2 13  4 15  -  -  -   #=5
insert ( 6)     11  2 13  4 15  6  -  -   #=6
insert (17)     11  2 13  4 15  6 17  -   #=7
remove_at( 3)   11  2 13 17 15  6  -  -   #=6
remove_at( 3)   11  2 13  6 15  -  -  -   #=5
insert ( 8)     11  2 13  6 15  8  -  -   #=6
remove_at( 3)   11  2 13  8 15  -  -  -   #=5
remove_at( 2)   11  2 15  8  -  -  -  -   #=4
insert (19)     11  2 15  8 19  -  -  -   #=5
```

```
remove_at( 2)   11  2 19  8  -  -  -  -   #=4
remove_at( 2)   11  2  8  -  -  -  -  -   #=3
insert (10)     11  2  8 10  -  -  -  -   #=4
remove_at( 2)   11  2 10  -  -  -  -  -   #=3
remove_at( 1)   11 10  -  -  -  -  -  -   #=2
insert (21)     11 10 21  -  -  -  -  -   #=3
remove_at( 1)   11 21  -  -  -  -  -  -   #=2
remove_at( 1)   11  -  -  -  -  -  -  -   #=1
insert (12)     11 12  -  -  -  -  -  -   #=2
remove_at( 1)   11  -  -  -  -  -  -  -   #=1
remove_at( 0)    -  -  -  -  -  -  -  -   #=0
insert (23)     23  -  -  -  -  -  -  -   #=1
remove_at( 0)    -  -  -  -  -  -  -  -   #=0
remove_at( 0)    -  -  -  -  -  -  -  -   #=0
  (rset was empty)
insert (14)     14  -  -  -  -  -  -  -   #=1
remove_at( 0)    -  -  -  -  -  -  -  -   #=0
remove_at( 0)    -  -  -  -  -  -  -  -   #=0
  (rset was empty)
insert (25)     25  -  -  -  -  -  -  -   #=1
```

Chapter 11

Selected combinatorial algorithms

This chapter presents selected combinatorial algorithms. The generation of combinations, subsets, partitions, and pairings of parentheses (as example for the use of 'funcemu') are treated here. Permutations are treated in a separate chapter because of the not so combinatorial viewpoint taken with most of the material (especially the specific examples like the revbin-permutation) there.

11.1 Combinations in lexicographic order

The combinations of three elements out of six in *lexicographic* order are

```
[  0  1  2 ]    ...111  #  0
[  0  1  3 ]    ..1.11  #  1
[  0  1  4 ]    .1..11  #  2
[  0  1  5 ]    1...11  #  3
[  0  2  3 ]    ..11.1  #  4
[  0  2  4 ]    .1.1.1  #  5
[  0  2  5 ]    1..1.1  #  6
[  0  3  4 ]    .11..1  #  7
[  0  3  5 ]    1.1..1  #  8
[  0  4  5 ]    11...1  #  9
[  1  2  3 ]    ..111.  # 10
[  1  2  4 ]    .1.11.  # 11
[  1  2  5 ]    1..11.  # 12
[  1  3  4 ]    .11.1.  # 13
[  1  3  5 ]    1.1.1.  # 14
[  1  4  5 ]    11..1.  # 15
[  2  3  4 ]    .111..  # 16
[  2  3  5 ]    1.11..  # 17
[  2  4  5 ]    11.1..  # 18
[  3  4  5 ]    111...  # 19
```

A bit of contemplation (staring at the ".1"-strings might help) leads to the code implementing a simple utility class that supplies the methods `first()`, `last()`, `next()` and `prev()`:

```
class comb_lex
{
public:
    ulong n_;
    ulong k_;
    ulong *x_;
public:
    comb_lex(ulong n, ulong k)
    {
        n_ = (n ? n : 1);  // not zero
        k_ = (k ? k : 1);  // not zero
        x_ = new ulong[k_];
        first();
```

```
    }
    ~comb_lex()  { delete [] x_; }

    void first()
    {
        for (ulong k=0; k<k_; ++k)  x_[k] = k;
    }
    void last()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = n_ - k_ + i;
    }
    ulong next()  // return zero if previous comb was the last
    {
        if ( x_[0] == n_ - k_ )  { first();  return 0; }
        ulong j = k_ - 1;
        // trivial if highest element != highest possible value:
        if ( x_[j] < (n_-1) )  { ++x_[j];  return 1; }
        // find highest falling edge:
        while ( 1 == (x_[j] - x_[j-1]) )  { --j; }
        // move lowest element of highest block up:
        ulong z = ++x_[j-1];
        // ... and attach rest of block:
        while ( j < k_ )  { x_[j] = ++z;  ++j; }
        return  1;
    }
    ulong prev()  // return zero if current comb is the first
    {
        if ( x_[k_-1] == k_-1 )  { last();  return 0; }
        // find highest falling edge:
        ulong j = k_ - 1;
        while ( 1 == (x_[j] - x_[j-1]) )  { --j; }
        --x_[j];  // move down edge element
        // ... and move rest of block to high end:
        while ( ++j < k_ )  x_[j] = n_ - k_ + j;
        return  1;
    }
    const ulong * data()  { return x_; }
    friend ostream & operator << (ostream &os, const comb_lex &x);
};
```

[FXT: `class comb_lex` in `comb/comblex.h`]

The listing at the beginning of this section can then be produced by a simple fragment like

```
    ulong ct = 0, n = 6,  k = 3;
    comb_lex comb(n, k);
    do
    {
        cout << endl;
        cout << "   [ " << comb << " ]  ";
        print_set_as_bitset("", comb.data(), k, n );
        cout << "  #" << setw(3) << ct;
        ++ct;
    }
    while ( comb.next() );
```

Cf. [FXT: file `demo/comblex-demo.cc`].

11.2 Combinations in co-lexicographic order

The combinations of three elements out of six in *co-lexicographic* ('colex') order are

```
[  0  1  2 ]    ...111  #  0
[  0  1  3 ]    ..1.11  #  1
[  0  2  3 ]    ..11.1  #  2
[  1  2  3 ]    ..111.  #  3
[  0  1  4 ]    .1..11  #  4
[  0  2  4 ]    .1.1.1  #  5
[  1  2  4 ]    .1.11.  #  6
[  0  3  4 ]    .11..1  #  7
[  1  3  4 ]    .11.1.  #  8
[  2  3  4 ]    .111..  #  9
[  0  1  5 ]    1...11  # 10
[  0  2  5 ]    1..1.1  # 11
[  1  2  5 ]    1..11.  # 12
[  0  3  5 ]    1.1..1  # 13
[  1  3  5 ]    1.1.1.  # 14
[  2  3  5 ]    1.11..  # 15
[  0  4  5 ]    11...1  # 16
[  1  4  5 ]    11..1.  # 17
[  2  4  5 ]    11.1..  # 18
[  3  4  5 ]    111...  # 19
```

Again, the algorithm is pretty straight forward:

```
class comb_colex
{
public:
    ulong n_;
    ulong k_;
    ulong *x_;
public:
    comb_colex(ulong n, ulong k)
    {
        n_ = (n ? n : 1);  // not zero
        k_ = (k ? k : 1);  // not zero
        x_ = new ulong[k_];
        first();
    }
    ~comb_colex()  { delete [] x_; }

    void first()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = i;
    }
    void last()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = n_ - k_ + i;
    }
    ulong next()  // return zero if previous comb was the last
    {
        if ( x_[0] == n_ - k_ )  { first();  return 0; }
        ulong j = 0;
        // until lowest rising edge ...
        while ( 1 == (x_[j+1] - x_[j]) )
        {
            x_[j] = j;  // attach block at low end
            ++j;
        }
        ++x_[j];  // move edge element up
        return  1;
    }
    ulong prev()  // return zero if current comb is the first
    {
        if ( x_[k_-1] == k_-1 )  { last();  return 0; }
        // find lowest falling edge:
        ulong j = 0;
        while ( j == x_[j] )  ++j;
        --x_[j];  // move edge element down
        // attach rest of low block:
        while ( 0!=j-- )  x_[j] = x_[j+1] - 1;
```

```
        return  1;
    }
    const ulong * data()  { return x_; }
    friend ostream & operator << (ostream &os, const comb_colex &x);
};
```

[FXT: `class comb_colex` in `comb/combcolex.h`]

For the connection between lex-order and colex-order see section 8.8

Usage is completely analogue to that of the `class comb_lex`, cf. [FXT: file `demo/combcolex-demo.cc`].

11.3 Combinations in minimal-change order

The combinations of three elements out of six in *minimal-change* order are

```
   ...111   [  0  1  2 ]    swap: (0, 0)   #  0
   ..11.1   [  0  2  3 ]    swap: (3, 1)   #  1
   ..111.   [  1  2  3 ]    swap: (1, 0)   #  2
   ..1.11   [  0  1  3 ]    swap: (2, 0)   #  3
   .11..1   [  0  3  4 ]    swap: (4, 1)   #  4
   .11.1.   [  1  3  4 ]    swap: (1, 0)   #  5
   .111..   [  2  3  4 ]    swap: (2, 1)   #  6
   .1.1.1   [  0  2  4 ]    swap: (3, 0)   #  7
   .1.11.   [  1  2  4 ]    swap: (1, 0)   #  8
   .1..11   [  0  1  4 ]    swap: (2, 0)   #  9
   11...1   [  0  4  5 ]    swap: (5, 1)   # 10
   11..1.   [  1  4  5 ]    swap: (1, 0)   # 11
   11.1..   [  2  4  5 ]    swap: (2, 1)   # 12
   111...   [  3  4  5 ]    swap: (3, 2)   # 13
   1.1..1   [  0  3  5 ]    swap: (4, 0)   # 14
   1.1.1.   [  1  3  5 ]    swap: (1, 0)   # 15
   1.11..   [  2  3  5 ]    swap: (2, 1)   # 16
   1..1.1   [  0  2  5 ]    swap: (3, 0)   # 17
   1..11.   [  1  2  5 ]    swap: (1, 0)   # 18
   1...11   [  0  1  5 ]    swap: (0, 2)   # 19
```

The algorithm used in the utility class [FXT: `class comb_minchange` in `comb/combminchange.h`] is based on inlined versions of the routines that were explained in the corresponding bit-magic section (8.13).

```
class comb_minchange
{
public:
    ulong n_;  // number of elements to choose from
    ulong k_;  // number of elements of subsets
    ulong igc_bits_;
    ulong bits_;
    ulong igc_last_;
    ulong igc_first_;
    ulong sw1_, sw2_;
    ulong *x_;
public:
    comb_minchange(ulong n, ulong k)
    {
        n_ = (n ? n : 1);  // not zero
        k_ = (k ? k : 1);  // not zero
        x_ = new ulong[k_];

        igc_last_ =  igc_last_comb(k_, n_);
        igc_first_ =  first_sequency(k_);

        first();
    }

    ~comb_minchange()
    {
        delete [] x_;
    }
```

```
    const ulong * data()  const  { return x_; }

    ulong first()
    {
        igc_bits_ = igc_first_;
        bits_ = gray_code( igc_last_ );  // to get sw1_, sw2_ right
        sync_x();
        return  bits_;
    }

    ulong last()
    {
        igc_bits_ = igc_last_;
        bits_ = gray_code( igc_first_ );  // to get sw1_, sw2_ right
        sync_x();
        return  bits_;
    }

    ulong next()  // return zero if current comb is the last
    {
        if ( igc_bits_ == igc_last_ )  return 0;

        ulong gy, y, i = 2;
        do
        {
            y = igc_bits_ + i;
            gy = gray_code( y );
            i <<= 1;
        }
        while ( bit_count( gy ) != k_ );
        igc_bits_ = y;

        sync_x();
        return  bits_;
    }

    ulong prev()  // return zero if current comb is the first
    {
        if ( igc_bits_ == igc_first_ )  return 0;

        ulong gy, y, i = 2;
        do
        {
            y = igc_bits_ - i;
            gy = gray_code( y );
            i <<= 1;
        }
        while ( bit_count( gy ) != k_ );
        igc_bits_ = y;

        sync_x();
        return  bits_;
    }

    void sync_x()  // aux
    // Sync bits into array and
    // set sw1_ and sw2_
    {
        ulong tbits = gray_code( igc_bits_ );
        ulong sw = bits_ ^ tbits;

        bits_ = tbits;

        ulong xi = 0, bi = 0;
        while ( bi < n_ )
        {
            if ( tbits & 1 )  x_[xi++] = bi;
            ++bi;
            tbits >>= 1;
        }

        sw1_ = 0;
        while ( 0==(sw&1) )  { sw >>= 1;  ++sw1_; }
        sw2_ = sw1_;
        do  { sw >>= 1;  ++sw2_; }  while ( 0==(sw&1) );
    }

    friend ostream & operator << (ostream &os, const comb_minchange &x);
};
```

The listing at the beginning of this section can be generated via code like:

```
ulong ct = 0, n = 6, k = 3;
comb_minchange comb(n, k);
comb.first();
do
{
    for (long k=n-1; k>=0; --k)  cout << ((bits>>k)&1 ? '1' : '.');
    cout << "   [ " << comb << " ]  ";
    cout << "  swap: (" << comb.sw1_ << ", " << comb.sw2_ << ") ";
    cout << "  #" << setw(3) << ct;
    ++ct;
    cout << endl;
}
while ( comb.next() );
```

cf. [FXT: file `demo/combminchange-demo.cc`].

11.4 Combinations in alternative minimal-change order

There is more than one minimal-change order. Consider the sequence of bit-sets generated in section 8.13: alternative orderings that have the minimal-change property are e.g. described by 1) the sequence with each word reversed or, more general 2) every permutation of the bits 3) the sequence with its bits negated 4) cyclical rotations of (1) ... (3)

Here we use the negated and bit-reversed sequence for $\binom{n-k}{n}$ in order to generate the combinations corresponding to $\binom{k}{n}$:

```
n = 6  k = 3:
   ...111   [  0  1  2 ]    swap: (3, 0)   #  0
   .1..11   [  0  1  4 ]    swap: (4, 2)   #  1
   1...11   [  0  1  5 ]    swap: (5, 4)   #  2
   ..1.11   [  0  1  3 ]    swap: (5, 3)   #  3
   .11..1   [  0  3  4 ]    swap: (4, 1)   #  4
   1.1..1   [  0  3  5 ]    swap: (5, 4)   #  5
   11...1   [  0  4  5 ]    swap: (4, 3)   #  6
   .1.1.1   [  0  2  4 ]    swap: (5, 2)   #  7
   1..1.1   [  0  2  5 ]    swap: (5, 4)   #  8
   ..11.1   [  0  2  3 ]    swap: (5, 3)   #  9
   .111..   [  2  3  4 ]    swap: (4, 0)   # 10
   1.11..   [  2  3  5 ]    swap: (5, 4)   # 11
   11.1..   [  2  4  5 ]    swap: (4, 3)   # 12
   111...   [  3  4  5 ]    swap: (3, 2)   # 13
   .11.1.   [  1  3  4 ]    swap: (5, 1)   # 14
   1.1.1.   [  1  3  5 ]    swap: (5, 4)   # 15
   11..1.   [  1  4  5 ]    swap: (4, 3)   # 16
   .1.11.   [  1  2  4 ]    swap: (5, 2)   # 17
   1..11.   [  1  2  5 ]    swap: (5, 4)   # 18
   ..111.   [  1  2  3 ]    swap: (5, 3)   # 19
```

The interesting feature is that the last combination is identical to the first shifted left by one. This makes it easy to generate the subsets of a set with n elements in monotonic min-change order by concatenating the sequences for $k = 1, 2, \ldots, n$.

The usage of the utility class [FXT: `class comb_alt_minchange` in `comb/combaltminchange.h`] is identical to that of the "standard" min-change order.

The above listing can be produced via

```
ulong n = 6,  k = 3,  ct = 0;
comb_alt_minchange comb(n, k);
comb.first();
do
{
    ulong bits = revbin( ~comb.bits_, n);  // reversed and negated
    cout << "   ";
    for (long k=n-1; k>=0; --k)  cout << ((bits>>k)&1 ? '1' : '.');
    cout << "   [ " << comb << " ]  ";
```

```
        cout << "  swap: (" << comb.sw1_ << ", " << comb.sw2_ << ") ";
        cout << "  #" << setw(3) << ct;
        ++ct;
        cout << endl;
    }
    while ( comb.next() );
```

11.5 Offline functions: funcemu

Sometimes it is possible to find recursive algorithm for solving some problem that is not easily solved iteratively. However the recursive implementations might produce the results in midst of its calling graph. When a utility class providing a the results one by one with some `next` call is required there is an apparent problem: There is only one stack available for function calls[1]. We do not have *offline* functions.

As an example consider the following recursive code[2]

```
int n = 4;
int v[n];

int main()
{
    paren(0, 0);
    return 0;
}

void paren(long i, long s)
{
    long k, t;
    if ( i<n )
    {
        for (k=0; k<=i-s;  ++k)
        {
            a[i-1] = k;
            t = s + a[i-1];
            q[t + i] = '(';
            paren(i + 1, t);  // recursion
            q[t + i] = ')';
        }
    }
    else
    {
        a[i-1] = n - s;
        Visit();  // next set of parens available
    }
}
```

that generates following output (the different ways to group four pairs of parenthesis):

```
(((())))
(((())))
((()())) 
((())()) 
(()((())) 
(()(()))
(()())()
(())(())
(())()()
()((()))
()(()())
()(())()
()()(())
()()()()
```

A reasonable way to create offline functions[3] is to rewrite the function as a state engine and utilize a class [FXT: `class funcemu` in `aux0/funcemu.h`] that provides two stacks, one for local variables and one for the state of the function:

[1] True for the majority of the programming languages.

[2] given by Glenn Rhoads

[3] A similar mechanism is called *coroutines* in languages that offer it.

```
template <typename Type>
class funcemu
{
public:
    ulong tp_;  // sTate stack Pointer
    ulong dp_;  // Data stack Pointer
    ulong *t_;  // sTate stack
    Type  *d_;  // Data stack
public:
    funcemu(ulong maxdepth, ulong ndata)
    {
        t_ = new ulong[maxdepth];
        d_ = new Type[ndata];
        init();
    }
    ~funcemu()
    {
        delete [] d_;
        delete [] t_;
    }
    void init()  { dp_=0; tp_=0; }
    void stpush(ulong x)  { t_[tp_++] = x; }
    ulong stpeek()  const  { return  t_[tp_-1]; }
    void stpeek(ulong &x)  { x = t_[tp_-1]; }
    void stpoke(ulong x)  { t_[tp_-1] = x; }
    void stpop()  { --tp_; }
    void stpop(ulong ct)  { tp_-=ct; }
    void stnext()  { ++t_[tp_-1]; }
    void stnext(ulong x)  { t_[tp_-1] = x; }
    bool more()  const  { return (0!=dp_); }
    void push(Type x)  { d_[dp_++] = x; }
    void push(Type x, Type y)  { push(x); push(y); }
    void push(Type x, Type y, Type z)  { push(x); push(y); push(z); }
    void push(Type x, Type y, Type z, Type u)
    { push(x); push(y); push(z); push(u); }
    void peek(Type &x)  { x = d_[dp_-1]; }
    void peek(Type &x, Type &y)
    { y = d_[dp_-1]; x = d_[dp_-2]; }
    void peek(Type &x, Type &y, Type &z)
    { z = d_[dp_-1]; y = d_[dp_-2]; x = d_[dp_-3]; }
    void peek(Type &x, Type &y, Type &z, Type &u)
    { u = d_[dp_-1]; z = d_[dp_-2]; y = d_[dp_-3]; x = d_[dp_-4]; }
    void poke(Type x)  { d_[dp_-1] = x; }
    void poke(Type x, Type y)
    { d_[dp_-1] = y; d_[dp_-2] = x; }
    void poke(Type x, Type y, Type z)
    { d_[dp_-1] = z; d_[dp_-2] = y; d_[dp_-3] = x; }
    void poke(Type x, Type y, Type z, Type u)
    { d_[dp_-1] = u; d_[dp_-2] = z; d_[dp_-3] = y; d_[dp_-4] = x; }
    void pop(ulong ct=1) { dp_-=ct; }
};
```

Rewriting the function in question (as part of a utility class, [FXT: file `comb/paren.h`] and [FXT: file `comb/paren.cc`]) only requires the understanding of the language, not of the algorithm. The process is straight forward but needs a bit of concentration, `#define`s are actually useful to slightly beautify the code:

```
#define PAREN    0  // initial state
#define RETURN  20
//               args=(i, s)(k, t)=locals
#define EMU_CALL(func, i, s, k, t)  fe_->stpush(func);  fe_->push(i, s, k, t);
paren::next_recursion()
{
    int i, s; // args
    int k, t; // locals
 redo:
```

```
    fe_->peek(i, s, k, t);
 loop:
    switch ( fe_->stpeek() )
    {
    case 0:
        if ( i>=n )
        {
            x[i-1] = n - s;
            fe_->stnext( RETURN );  return 1;
        }
        fe_->stnext();
    case 1:
        if ( k>i-s ) // loop end ?
        {
            break;  // shortcut: nothing to do at end
        }
        fe_->stnext();
    case 2: // start of loop body
        x[i-1] = k;
        t = s + x[i-1];
        str[t+i] = '('; // OPEN_CHAR;
        fe_->poke(i, s, k, t);  fe_->stnext();
        EMU_CALL( PAREN, i+1, t, 0, 0 );
        goto redo;
    case 3:
        str[t+i] = ')'; // CLOSE_CHAR;
        ++k;
        if ( k>i-s ) // loop end ?
        {
            break;  // shortcut: nothing to do at end
        }
        fe_->stpoke(2);  goto loop; // shortcut: back to loop body
    default: ;
    }
    fe_->pop(4);  fe_->stpop();  // emu_return to caller
    if ( fe_->more() )  goto redo;
    return 0;  // return from top level emu_call
}
```

The constructor initializes the funcemu and pushes the needed variables and parameters on the data stack and the initial state on the state stack:

```
paren::paren(int nn)
{
    n = (nn>0 ? nn : 1);
    x = new int[n];
    str = new char[2*n+1];
    for (int i=0; i<2*n; ++i)  str[i] = ')';
    str[2*n] = 0;
    fe_ = new funcemu<int>(n+1, 4*(n+1));
    //               i, s, k, t
    EMU_CALL( PAREN, 0, 0, 0, 0 );
    idx = 0;
    q = next_recursion();
}
```

The `EMU_CALL` actually only initializes the data for the state engine, the following call to `next_recursion` then lets the thing run.

The method `next` of the paren class lets the offline function advance until the next result is available:

```
int paren::next()
{
    if ( 0==q )  return 0;
    else
    {
        q = next_recursion();
```

```
        return  ( q ? ++idx : 0 );
    }
}
```

Performance wise the `funcemu`-rewritten functions are close to the original (state engines are fast and the operations within `funcemu` are cheap).

The shown method can also applied when the recursive algorithm consists of more than one function by merging the functions into one state engine.

The presented mechanism is also useful for unmaintainable code insanely cluttered with `goto` statements.

Further, investigating the contents of the data stack can be of help in the search of a iterative solution.

11.6 Parenthesis

An iterative scheme to generate all valid ways to group parenthesis can be comes from a modified version of the combinations in co-lexicographic order (see section 11.2). For $n = 5$ pairs the possible combinations are (the right column has a one where a closing parenthesis occurs, else a dot):

```
  ((((()))))  .....11111  #  0
  (((()())))  ....1.1111  #  1
  (((())())))  ....11.111  #  2
  (((()))()))  ....111.11  #  3
  (((())))()))  ....1111.1  #  4
  ((()(())))  ...1..1111  #  5
  ((()()()))  ...1.1.111  #  6
  ((()())()))  ...1.11.11  #  7
  ((()()))()))  ...1.111.1  #  8
  ((())(()))  ...11..111  #  9
  ((())()()))  ...11.1.11  # 10
  ((())())()))  ...11.11.1  # 11
  ((()))(()))  ...111..11  # 12
  ((()))()()))  ...111.1.1  # 13
  (()((())))  ..1...1111  # 14
  (()(()()))  ..1..1.111  # 15
  (()(())())))  ..1..11.11  # 16
  (()(()))()))  ..1..111.1  # 17
  (()()(()))  ..1.1..111  # 18
  (()()()()))  ..1.1.1.11  # 19
  (()()())()))  ..1.1.11.1  # 20
  (()())(()))  ..1.11..11  # 21
  (()())()()))  ..1.11.1.1  # 22
  (())((()))  ..11...111  # 23
  (())(()()))  ..11..1.11  # 24
  (())(())()))  ..11..11.1  # 25
  (())()(()))  ..11.1..11  # 26
  (())()()()))  ..11.1.1.1  # 27
  ()(((())))  .1....1111  # 28
  ()(((())))  .1...1.111  # 29
  ()(((())))  .1...11.11  # 30
  ()(((())))()))  .1...111.1  # 31
  ()(()(()))  .1..1..111  # 32
  ()(()()())  .1..1.1.11  # 33
  ()(()())()))  .1..1.11.1  # 34
  ()(())(()))  .1..11..11  # 35
  ()(())()()))  .1..11.1.1  # 36
  ()()((()))  .1.1...111  # 37
  ()()(()())  .1.1..1.11  # 38
  ()()(())()))  .1.1..11.1  # 39
  ()()()(()))  .1.1.1..11  # 40
  ()()()()())  .1.1.1.1.1  # 41
```

Observe that whenever a repeated pattern of `.1` is at the right end the next combination is generated by gathering these ones and the block of ones to its left, moving the leftmost one position to the left and all the others as a block to the right end (see the transitions from `#13` to `#14` or `#36` to `#37`).

The code of the resulting utility class [FXT: `class paren2` in `comb/paren2.h`] is concise and fast:

```
class paren2
// parentheses by a modified comb_colex procedure
```

```
{
public:
    ulong k_;  // number of paren pairs
    ulong n_;  // ==2*k
    ulong *x_; // (negated) positions where a close paren occurs
    char *str_;  // string representation,  e.g. "((())())()"
public:
    paren2(ulong k)
    {
        k_ = (k>1 ? k : 2);  // not zero (empty) or one (trivial: "()")
        n_ = 2 * k_;
        x_ = new ulong[k_ + 1];
        x_[k_] = 999;  // sentinel
        str_ = new char[n_ + 1];
        str_[n_] = 0;
        first();
    }

    ~paren2()
    {
        delete [] x_;
        delete [] str_;
    }

    void first()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = i;
    }

    void last()
    {
        for (ulong i=0; i<k_; ++i)  x_[i] = 2*i;
    }

    ulong next()  // return zero if previous paren was the last
    {
        ulong j = 0;
        if ( x_[1] == 2 )
        {
            // scan for low end == 010101:
            j = 2;
            while ( (j<=k_) && (x_[j]==2*j) )  ++j;  // can touch sentinel

            if ( j==k_ )
            {
                first();
                return  0;
            }
        }

        // if ( k_==1 )  return 0;  // uncomment to make algorithm work for k_==1

        // scan block:
        while ( 1 == (x_[j+1] - x_[j]) )  { ++j; }

        ++x_[j];  // move edge element up
        for (ulong i=0; i<j; ++i)  x_[i] = i; // attach block at low end

        return  1;
    }

    const ulong * data()  { return x_; }

    const char * string()  // generate on demand
    {
        for (ulong j=0; j<n_; ++j)  str_[j] = '(';
        for (ulong j=0; j<k_; ++j)  str_[n_-1-x_[j]] = ')';
        return str_;
    }
};
```

The number of valid combinations of n parenthesis pairs is

$$C_n \;=\; \frac{\binom{2\,n}{n}}{n+1} \tag{11.1}$$

as nicely explained in (chapter 7.5, example 4 on page 343 of) [13]. These are the Catalan numbers:

n	C_n
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1430
9	4862
10	16796
11	58786
12	208012

11.7 Partitions

An integer x is the sum of the positive integers less or equal to itself in various ways ($x = 4$ in this example):

```
  4* 1  +  0* 2  +  0* 3  +  0* 4  ==   4
  2* 1  +  1* 2  +  0* 3  +  0* 4  ==   4
  0* 1  +  2* 2  +  0* 3  +  0* 4  ==   4
  1* 1  +  0* 2  +  1* 3  +  0* 4  ==   4
  0* 1  +  0* 2  +  0* 3  +  1* 4  ==   4
```

The left hand side expressions are called the *partitions* of the number x. We want to attack a slightly more general problem and find all partitions of a number x with respect to a set $V = \{v_0, v_1, \ldots, v_{n-1}\}$, that is all decompositions of the form $x = \sum_{k=0}^{n-1} c_k \cdot v_k$.

The utility class is

```
class partition
{
public:
    ulong ct_;  // # of partitions found so far
    ulong n_;   // # of values
    ulong i_;   // level in iterative search

    long *pv_;  // values into which to partition
    ulong *pc_; // multipliers for values
    ulong pci_; // temporary for pc_[i_]
    long *r_;   // rest
    long ri_;   // temporary for r_[i_]
    long x_;    // value to partition
public:
    partition(const ulong *pv, ulong n)
        : n_(n==0?1:n)
    {
        pv_ = new long[n_+1];
        for (ulong j=0; j<n_; ++j)  pv_[j] = pv[j];
        pc_ = new ulong[n_+1];
        r_  = new long[n_+1];
    }

    ~partition()
    {
        delete [] pv_;
        delete [] pc_;
        delete [] r_;
    }

    void init(ulong x);  // reset state

    ulong next();  // generate next partition
    ulong next_func(ulong i);  // aux

    ulong count(ulong x);  // count number of partitions
```

```
    ulong count_func(ulong i);  // aux
    void dump()  const;
    int  check(ulong i=0)  const;
};
```

[FXT: `class partition` in `comb/partition.h`]

The algorithm to count the partitions is to assign to the first bucket a multiple $c_0 \cdot p_0 \le x$ of the first set element p_0. If $c_0 \cdot p_0 = x$ we already found a partition, else if $c_0 \cdot p_0 < x$ solve the problem for $x' := x - c_0 \cdot p_0$ and $V' := \{v_1, v_2, \ldots, v_{n-1}\}$.

```
ulong
partition::count(ulong x)
// count number of partitions
{
    init(x);
    count_func(n_-1);
    return ct_;
}

ulong
partition::count_func(ulong i)
{
    if ( 0!=i )
    {
        while ( r_[i]>0 )
        {
            pc_[i-1] = 0;
            r_[i-1] = r_[i];
            count_func(i-1);  // recursion
            r_[i] -= pv_[i];
            ++pc_[i];
        }
    }
    else  // recursion end
    {
        if ( 0!=r_[i] )
        {
            long d = r_[i] / pv_[i];
            r_[i] -= d * pv_[i];
            pc_[i] = d;
        }
    }

    if ( 0==r_[i] )  // valid partition found
    {
        // if ( whatever )  ++ct_;  // restricted count
        ++ct_;
        return 1;
    }
    else  return 0;
}
```

The algorithm, when rewritten iteratively, can supply the partitions one by one:

```
ulong
partition::next()
// generate next partition
{
    if ( i_>=n_ )  return  n_;

    r_[i_] = ri_;
    pc_[i_] = pci_;
    i_ = next_func(i_);

    for (ulong j=0; j<i_; ++j)  pc_[j] = r_[j] = 0;

    ++i_;
    ri_ = r_[i_] - pv_[i_];
    pci_ = pc_[i_] + 1;

    return  i_ - 1;  // >=0
}

ulong
partition::next_func(ulong i)
```

```
{
 start:
    if ( 0!=i )
    {
        while ( r_[i]>0 )
        {
            pc_[i-1] = 0;
            r_[i-1] = r_[i];
            --i;  goto start;   // iteration
        }
    }
    else  // iteration end
    {
        if ( 0!=r_[i] )
        {
            long d = r_[i] / pv_[i];
            r_[i] -= d * pv_[i];
            pc_[i] = d;
        }
    }

    if ( 0==r_[i] )  // valid partition found
    {
        ++ct_;
        return i;
    }

    ++i;
    if ( i>=n_ )  return n_;  // search finished

    r_[i] -= pv_[i];
    ++pc_[i];

    goto start;  // iteration
}
```

[FXT: file comb/partition.cc]

The routines can easily adapted to the generation of partitions satisfying certain restrictions, e.g. partitions into unequal parts (i.e. $c_i \leq 1$).

Cf. [FXT: file demo/partition-demo.cc]

11.8 Compositions

The compositions of n into at most k parts are the ordered tuples $(x_0, x_1, \ldots, x_{k-1})$ where $x_0 + x_1 + \ldots + x_{k-1} = n$ and $0 \leq x_i \leq n$. Order matters: one 3-composition of 4 is $(0, 1, 2, 1)$, a different one is $(2, 0, 1, 1)$.

11.8.1 Compositions in lexicographic order

For the 4-compositions of 4 in lexicographic order are

```
 0:   4 0 0 0
 1:   3 1 0 0
 2:   2 2 0 0
 3:   1 3 0 0
 4:   0 4 0 0
 5:   3 0 1 0
 6:   2 1 1 0
 7:   1 2 1 0
 8:   0 3 1 0
 9:   2 0 2 0
10:   1 1 2 0
11:   0 2 2 0
12:   1 0 3 0
13:   0 1 3 0
14:   0 0 4 0
15:   3 0 0 1
16:   2 1 0 1
17:   1 2 0 1
18:   0 3 0 1
19:   2 0 1 1
20:   1 1 1 1
21:   0 2 1 1
22:   1 0 2 1
23:   0 1 2 1
```

```
24:   0 0 3 1
25:   2 0 0 2
26:   1 1 0 2
27:   0 2 0 2
28:   1 0 1 2
29:   0 1 1 2
30:   0 0 2 2
31:   1 0 0 3
32:   0 1 0 3
33:   0 0 1 3
34:   0 0 0 4
```

This is the output of [FXT: file demo/compositionlex-demo.cc].

The corresponding utility class is [FXT: class composition_lex in comb/compositionlex.h]:

```
class composition_lex
{
public:
    ulong n_;  // number of elements to choose from
    ulong *x_;  // data
public:
    composition_lex(ulong n)
    {
        n_ = (n ? n : 1);  // not zero
        x_ = new ulong[n_];
        first();
    }

    ~composition_lex()  { delete [] x_; }

    const ulong *data()  const  { return x_; }

    void first()
    {
        x_[0] = n_;
        for (ulong k=1; k<n_; ++k)  x_[k] = 0;
    }

    void last()
    {
        for (ulong k=0; k<n_; ++k)  x_[k] = 0;
        x_[n_-1] = n_;
    }

    ulong next()
    // Nijenhuis, Wilf
    {
        // return zero if current comp. is last:
        if ( n_==x_[n_-1] )  return 0;

        ulong j = 0;
        while ( 0==x_[j] )  ++j;
        ulong v = x_[j];  // first nonzero
        x_[j] = 0;
        x_[0] = v - 1;
        ++x_[j+1];

        return  1;
    }

    ulong prev()
    {
        // return zero if current comp. is first:
        if ( n_==x_[0] )  return 0;

        ulong v0 = x_[0];
        x_[0] = 0;
        ulong j = 1;
        while ( 0==x_[j] )  ++j;
        --x_[j];
        x_[j-1] = 1 + v0;

        return  1;
    }
};
```

11.8.2 Compositions from combinations

There are $\binom{2\,n-1}{n}$ n-compositions of n which indicates that there should be a connection to the combinations of n out of $2\,n-1$ items. In the following listing the (reversed) lex-order combinations $\binom{7}{4}$ are opposed by the compositions of 4:

```
    1111...         4 0 0 0
    111.1..         3 1 0 0
    11.11..         2 2 0 0
    1.111..         1 3 0 0
    .1111..         0 4 0 0
    111..1.         3 0 1 0
    11.1.1.         2 1 1 0
    1.11.1.         1 2 1 0
    .111.1.         0 3 1 0
    11...11         2 0 2 0
    1.1.11.         1 1 2 0
    .11.11.         0 2 2 0
    1...111         1 0 3 0
    .1.1111         0 1 3 0
    ..1111.         0 0 4 0
    111...1         3 0 0 1
    11.1..1         2 1 0 1
    1.11..1         1 2 0 1
    .111..1         0 3 0 1
    11..1.1         2 0 1 1
    1.1.1.1         1 1 1 1
    .11.1.1         0 2 1 1
    1...11.1        1 0 2 1
    .1.11.1         0 1 2 1
    ..111.1         0 0 3 1
    11....11        2 0 0 2
    1.1..11         1 1 0 2
    .11..11         0 2 0 2
    1...1.11        1 0 1 2
    .1.1.11         0 1 1 2
    ..11.11         0 0 2 2
    1....111        1 0 0 3
    .1...111        0 1 0 3
    ..1..111        0 0 1 3
    ...1111         0 0 0 4
```

The entries in the compositions can be identified by the runs of consecutive bits in the combinations. One bit that follows each run of ones is considered as a separator. This leads to the function

```
static inline void bit2composition(ulong w, ulong *x, ulong n)
{
    for (ulong k=0; k<n; ++k)
    {
        ulong ct = 0;
        ulong b;
        do
        {
            b = w & 1;
            ct += b;
            w >>= 1;
        }
        while ( 0!=b );
        x[k] = ct;
    }
}
```

The given function enables us to use any routine that generates combinations to produce compositions. The minimal-change combinations (see section 11.3) give the following sequence of compositions:

```
    ....1111    [4 0 0 0 ]    #  0
    ...11.11    [2 2 0 0 ]    #  1
    ...1111.    [0 4 0 0 ]    #  2
    ...111.1    [1 3 0 0 ]    #  3
    ...1.111    [3 1 0 0 ]    #  4
    ..11..11    [2 0 2 0 ]    #  5
    ..11.11.    [0 2 2 0 ]    #  6
    ..11.1.1    [1 1 2 0 ]    #  7
    ..1111..    [0 0 4 0 ]    #  8
    ..111.1.    [0 1 3 0 ]    #  9
    ..111..1    [1 0 3 0 ]    # 10
    ..1.1.11    [2 1 1 0 ]    # 11
    ..1.111.    [0 3 1 0 ]    # 12
    ..1.11.1    [1 2 1 0 ]    # 13
    ..1..111    [3 0 1 0 ]    # 14
    .11...11    [2 0 0 2 ]    # 15
    .11..11.    [0 2 0 2 ]    # 16
    .11..1.1    [1 1 0 2 ]    # 17
```

```
.11.11..    [0 0 2 2 ]    # 18
.11.1.1.    [0 1 1 2 ]    # 19
.11.1..1    [1 0 1 2 ]    # 20
.1111...    [0 0 0 4 ]    # 21
.111.1..    [0 0 1 3 ]    # 22
.111..1.    [0 1 0 3 ]    # 23
.111...1    [1 0 0 3 ]    # 24
.1.1..11    [2 0 1 1 ]    # 25
.1.1.11.    [0 2 1 1 ]    # 26
.1.1.1.1    [1 1 1 1 ]    # 27
.1.111..    [0 0 3 1 ]    # 28
.1.11.1.    [0 1 2 1 ]    # 29
.1.11..1    [1 0 2 1 ]    # 30
.1..1.11    [2 1 0 1 ]    # 31
.1..111.    [0 3 0 1 ]    # 32
.1..11.1    [1 2 0 1 ]    # 33
.1...111    [3 0 0 1 ]    # 34
```

This is the output of [FXT: file `demo/compositionalt-demo.cc`].

Note that the minimum-change property of the combinations does not lead to the equivalent for the compositions.

11.8.3 Compositions in minimal-change order

The modified idea to select from the radix-r Gray codes with $r = n+1$ those that are valid as composition (that is, the sum of digits is one) can be used for the generation of minimal-change compositions. With $n = 4$, $k = 4$ one gets

```
...1111     4 0 0 0
..1.111     3 1 0 0
..11.11     2 2 0 0
..111.1     1 3 0 0
..1111.     0 4 0 0
.1.111.     0 3 1 0
.1.11.1     1 2 1 0
.1.1.11     2 1 1 0
.1..111     3 0 1 0
.11..11     2 0 2 0
.11.1.1     1 1 2 0
.11.11.     0 2 2 0
.111.1.     0 1 3 0
.111..1     1 0 3 0
.1111..     0 0 4 0
1.111..     0 0 3 1
1.11.1.     0 1 2 1
1.11..1     1 0 2 1
1.1.1.11    2 0 1 1
1.1.1.1.    1 1 1 1
1.1.11.     0 2 1 1
1..111.     0 3 0 1
1..11.1     1 2 0 1
1..1.11     2 1 0 1
1...111     3 0 0 1
11...11     2 0 0 2
11..1.1     1 1 0 2
11..11.     0 2 0 2
11.1.1.     0 1 1 2
11.1..1     1 0 1 2
11.11..     0 0 2 2
111.1..     0 0 1 3
111..1.     0 1 0 3
111...1     1 0 0 3
```

However, the brute-force scan of the Gray codes is horribly ineffective, as $r^n = (n+1)^n$ candidates are scanned for the $\binom{2n-1}{n}$ compositions.

TBD: *fast algorithm for minchange-compositions*

11.9 Numbers in lexicographic order

A generalization of the lexicographic ordering for binary words (given in section 8.10) for radix 3 (using 4 digits and printing a dot for each zero) is

```
1 . . .
1 1 . .        2 . . .
1 1 1 .        2 1 . .          . 1
1 1 1 1        2 1 1 .          . 1 1 .
```

```
1 1 1 1        2 1 1 1        . 1 1 1
1 1 1 2        2 1 1 2        . 1 1 2
1 1 2 .        2 1 2 .        . 1 2 .
1 1 2 1        2 1 2 1        . 1 2 1
1 1 2 2        2 1 2 2        . 1 2 2
1 1 . 1        2 1 . 1        . 1 . 1
1 1 . 2        2 1 . 2        . 1 . 2
1 2 . .        2 2 . .        . 2 . .
1 2 1 .        2 2 1 .        . 2 1 .
1 2 1 1        2 2 1 1        . 2 1 1
1 2 1 2        2 2 1 2        . 2 1 2
1 2 2 .        2 2 2 .        . 2 2 .
1 2 2 1        2 2 2 1        . 2 2 1
1 2 2 2        2 2 2 2        . 2 2 2
1 2 . 1        2 2 . 1        . 2 . 1
1 2 . 2        2 2 . 2        . 2 . 2
1 . 1 .        2 . 1 .        . . 1 .
1 . 1 1        2 . 1 1        . . 1 1
1 . 1 2        2 . 1 2        . . 1 2
1 . 2 .        2 . 2 .        . . 2 .
1 . 2 1        2 . 2 1        . . 2 1
1 . 2 2        2 . 2 2        . . 2 2
1 . . 1        2 . . 1        . . . 1
1 . . 2        2 . . 2        . . . 2
```

The sequence can be generated by using a list of all 4-digit radix-3 numbers, replacing all trailing zeroes by a character that is sorted before all numbers (a dot works with ascii characters), replacing the remaining zeros by a character that is sorted after all numbers (a 'z' with ascii characters), and sorting the usual way. For the friends of the command line:

```
for n in {0,1,2}{0,1,2}{0,1,2}{0,1,2}; do echo $n; done \
  | sed  's/0\+$//' | sed 's/0/z/g' | sort \
  | sed 's/z/ /g'
```

A routine for generating successive words in lexicographic order for arbitrary radices is implemented in [FXT: `class mixed_radix_lex` in `comb/mixedradixlex.h`]. As the name might suggest, the base can be set for each individual digit. Successive words differ in at most three positions.

TBD: *applications*

11.10 Subsets in lexicographic order

The (nonempty) subsets of a set of five elements enumerated in *lexicographic* order are:

```
 0  #= 1:    ....1  {0}
 1  #= 2:    ...11  {0, 1}
 2  #= 3:    ..111  {0, 1, 2}
 3  #= 4:    .1111  {0, 1, 2, 3}
 4  #= 5:    11111  {0, 1, 2, 3, 4}
 5  #= 4:    1.111  {0, 1, 2, 4}
 6  #= 3:    .1.11  {0, 1, 3}
 7  #= 4:    11.11  {0, 1, 3, 4}
 8  #= 3:    1..11  {0, 1, 4}
 9  #= 2:    ..1.1  {0, 2}
10  #= 3:    .11.1  {0, 2, 3}
11  #= 4:    111.1  {0, 2, 3, 4}
12  #= 3:    1.1.1  {0, 2, 4}
13  #= 2:    .1..1  {0, 3}
14  #= 3:    11..1  {0, 3, 4}
15  #= 2:    1...1  {0, 4}
16  #= 1:    ...1.  {1}
17  #= 2:    ..11.  {1, 2}
18  #= 3:    .111.  {1, 2, 3}
19  #= 4:    1111.  {1, 2, 3, 4}
20  #= 3:    1.11.  {1, 2, 4}
21  #= 2:    .1.1.  {1, 3}
22  #= 3:    11.1.  {1, 3, 4}
23  #= 2:    1..1.  {1, 4}
24  #= 1:    ..1..  {2}
25  #= 2:    .11..  {2, 3}
26  #= 3:    111..  {2, 3, 4}
27  #= 2:    1.1..  {2, 4}
28  #= 1:    .1...  {3}
29  #= 2:    11...  {3, 4}
30  #= 1:    1....  {4}
```

Clearly there are 2^n subsets (including the empty set) of an n-element set.
The corresponding utility class is not too complicated

```
class subset_lex
{
protected:
    ulong *x;  // subset data
    ulong n;   // number of elements in set
    ulong k;   // index of last element in subset
    // number of elements in subset == k+1

public:
    subset_lex(ulong nn)
    {
        n = (nn ? nn : 1);  // not zero
        x = new ulong[n+1];
        first();
    }

    ~subset_lex()  { delete [] x; }

    ulong first()
    {
        k = 0;
        x[0] = 0;
        return  k + 1;
    }

    ulong last()
    {
        k = 0;
        x[0] = n - 1;
        return  k + 1;
    }

    ulong next()
    // Generate next subset
    // Return number of elements in subset
    // Return zero if current == last
    {
        if ( x[k] == n-1 )  // last element is max ?
        {
            if ( 0==k )  { return 0; }  // note: user has to call first() again

            --k;     // remove last element
            x[k]++;  // increase last element
        }
        else  // add next element from set:
        {
            ++k;
            x[k] = x[k-1] + 1;
        }

        return  k + 1;
    }

    ulong prev()
    // Generate previous subset
    // Return number of elements in subset
    // Return zero if current == first
    {
        if ( k == 0 )  // only one lement ?
        {
            if ( x[0]==0 )  { return 0; }  // note: user has to call last() again

            x[0]--;  // decr first element
            x[++k] = n - 1;     // add element
        }
        else  // remove last element:
        {
            if ( x[k] == x[k-1]+1 )  --k;
            else
            {
                x[k]--;  // decr last element
                x[++k] = n - 1;     // add element
            }
        }

        return  k + 1;
```

```
    }
    const ulong * data()  { return x; }
};
```

[FXT: `class subset_lex` in `comb/subsetlex.h`]

One can generate the list at the beginning of this sections by a code fragment like:

```
    ulong n = 5;
    subset_lex sl(n);
    ulong idx = 0;
    ulong num = sl.first();
    do
    {
        cout << setw(2) << idx;
        ++idx;
        cout << "  #=" << setw(2) << num << ":  ";
        print_set_as_bitset("  ", sl.data(), num, n);
        print_set("  ", sl.data(), num);
        cout << endl;
    }
    while ( (num = sl.next()) );
```

cf. [FXT: file `demo/subsetlex-demo.cc`]

11.11 Subsets in minimal-change order

The subsets of a set with 5 elements in minimal-change order:

```
 1:  1....   chg @ 0   num=1   set={0}
 2:  11...   chg @ 1   num=2   set={0, 1}
 3:  .1...   chg @ 0   num=1   set={1}
 4:  .11..   chg @ 2   num=2   set={1, 2}
 5:  111..   chg @ 0   num=3   set={0, 1, 2}
 6:  1.1..   chg @ 1   num=2   set={0, 2}
 7:  ..1..   chg @ 0   num=1   set={2}
 8:  ..11.   chg @ 3   num=2   set={2, 3}
 9:  1.11.   chg @ 0   num=3   set={0, 2, 3}
10:  1111.   chg @ 1   num=4   set={0, 1, 2, 3}
11:  .111.   chg @ 0   num=3   set={1, 2, 3}
12:  .1.1.   chg @ 2   num=2   set={1, 3}
13:  11.1.   chg @ 0   num=3   set={0, 1, 3}
14:  1..1.   chg @ 1   num=2   set={0, 3}
15:  ...1.   chg @ 0   num=1   set={3}
16:  ...11   chg @ 4   num=2   set={3, 4}
17:  1..11   chg @ 0   num=3   set={0, 3, 4}
18:  11.11   chg @ 1   num=4   set={0, 1, 3, 4}
19:  .1.11   chg @ 0   num=3   set={1, 3, 4}
20:  .1111   chg @ 2   num=4   set={1, 2, 3, 4}
21:  11111   chg @ 0   num=5   set={0, 1, 2, 3, 4}
22:  1.111   chg @ 1   num=4   set={0, 2, 3, 4}
23:  ..111   chg @ 0   num=3   set={2, 3, 4}
24:  ..1.1   chg @ 3   num=2   set={2, 4}
25:  1.1.1   chg @ 0   num=3   set={0, 2, 4}
26:  111.1   chg @ 1   num=4   set={0, 1, 2, 4}
27:  .11.1   chg @ 0   num=3   set={1, 2, 4}
28:  .1..1   chg @ 2   num=2   set={1, 4}
29:  11..1   chg @ 0   num=3   set={0, 1, 4}
30:  1...1   chg @ 1   num=2   set={0, 4}
31:  ....1   chg @ 0   num=1   set={4}
32:  .....   chg @ 4   num=0   set={}
```

Generation is easy, for a set with n elements go through the binary gray codes of the numbers from 1 to 2^{n-1} and sync the bits into the array to be used:

```
class subset_minchange
{
protected:
    ulong *x;  // current subset as delta-set
    ulong n;   // number of elements in set
    ulong num; // number of elements in current subset
    ulong chg; // element that was chnged with latest call to next()
    ulong idx;
    ulong maxidx;
public:
    subset_minchange(ulong nn)
    {
        n = (nn ? nn : 1);  // not zero
        x = new ulong[n];
        maxidx = (1<<nn) - 1;
        first();
    }
    ~subset_minchange()  { delete [] x; }

    ulong first()  // start with empty set
    {
        idx = 0;
        num = 0;
        chg = n - 1;
        for (ulong k=0; k<n; ++k)  x[k] = 0;
        return  num;
    }
    ulong next()  // return number of elements in subset
    {
        make_next();
        return  num;
    }
    const ulong * data() const { return x; }
    ulong get_change()  const  { return chg; }
    const ulong current()  const  { return idx; }
protected:
    void make_next()
    {
        ++idx;
        if ( idx > maxidx )
        {
            chg = n - 1;
            first();
        }
        else  // x[] essentially runs through the binary graycodes
        {
            chg = lowest_bit_idx( idx );
            x[chg] = 1 - x[chg];
            num += (x[chg] ? 1 : -1);
        }
    }
};
```

[FXT: `class subset_minchange` in `comb/subsetminchange.h`] The above list was created via

```
    ulong n = 5;
    subset_minchange sm(n);
    const ulong *x = sm.data();
    ulong num, idx = 0;
    do
    {
        num = sm.next();  // omit empty set
        ++idx;
        cout << setw(2) << idx << ":  ";
        // print as bit set:
        for (ulong k=0; k<n; ++k)  cout << (x[k]?'1':'.');
        cout << "   chg @ " << sm.get_change();
        cout << "   num=" << num;
        print_delta_set_as_set("   set=", x, n);
        cout << endl;
```

```
    }
    while ( num );
```

Cf. [FXT: file demo/subsetminchange-demo.cc]

11.12 Subsets ordered by number of elements

Sometimes it is useful to generate all subsets ordered with respect to the number of elements, that is starting with the 1-element subsets, continuing with 2-element subsets and so on until the full set is reached. For that purpose one needs to generate the combinations of 1 form n, 2 from n and so on. There are of course many orderings of that type, practical choices are limited by the various generators for combinations one wants to use. Here we use the colex-order for the combinations:

```
 1:  1....   #=1   set={0}
 2:  .1...   #=1   set={1}
 3:  ..1..   #=1   set={2}
 4:  ...1.   #=1   set={3}
 5:  ....1   #=1   set={4}
 6:  11...   #=2   set={0, 1}
 7:  1.1..   #=2   set={0, 2}
 8:  .11..   #=2   set={1, 2}
 9:  1..1.   #=2   set={0, 3}
10:  .1.1.   #=2   set={1, 3}
11:  ..11.   #=2   set={2, 3}
12:  1...1   #=2   set={0, 4}
13:  .1..1   #=2   set={1, 4}
14:  ..1.1   #=2   set={2, 4}
15:  ...11   #=2   set={3, 4}
16:  111..   #=3   set={0, 1, 2}
17:  11.1.   #=3   set={0, 1, 3}
18:  1.11.   #=3   set={0, 2, 3}
19:  .111.   #=3   set={1, 2, 3}
20:  11..1   #=3   set={0, 1, 4}
21:  1.1.1   #=3   set={0, 2, 4}
22:  .11.1   #=3   set={1, 2, 4}
23:  1..11   #=3   set={0, 3, 4}
24:  .1.11   #=3   set={1, 3, 4}
25:  ..111   #=3   set={2, 3, 4}
26:  1111.   #=4   set={0, 1, 2, 3}
27:  111.1   #=4   set={0, 1, 2, 4}
28:  11.11   #=4   set={0, 1, 3, 4}
29:  1.111   #=4   set={0, 2, 3, 4}
30:  .1111   #=4   set={1, 2, 3, 4}
31:  11111   #=5   set={0, 1, 2, 3, 4}
32:  .....   #=0   set={}
```

The class implementing the obvious algorithm is [FXT: `class subset_monotone` in `comb/subsetmonotone.h`]. The above list can be generated via

```
    ulong n = 5;
    subset_monotone so(n);
    const ulong *x = so.data();
    ulong num, idx = 0;
    do
    {
        num = so.next();
        ++idx;
        cout << setw(2) << idx << ":  ";

        // print as bit set:
        for (ulong k=0; k<n; ++k)  cout << (x[k]?'1':'.');
        cout << "   #=" << num;

        // print as set:
        print_delta_set_as_set("   set=", x, n);
        cout << endl;
    }
    while ( num );
```

cf. [FXT: file demo/subsetmonotone-demo.cc]

Replacing the colex-comb engine by alt-minchange-comb engine(s) (as described in section 11.4) gives the additional feature of minimal changes between the subsets.

11.13 Subsets ordered with shift register sequences

A curious sequence of all subsets of a given set can be generated using a binary *de Bruijn* (or shift register) sequence, that is a cyclical sequence of zeros and ones that contains each n-bit word once. In the following example (where $n = 5$) the empty places of the subsets are included to make the nice property apparent:

```
{0,  ,  ,  ,  }   #=1   0
{ , 1,  ,  ,  }   #=1   1
{ ,  , 2,  ,  }   #=1   2
{ ,  ,  , 3,  }   #=1   3
{0,  ,  ,  , 4}   #=2   4
{0, 1,  ,  ,  }   #=2   5
{ , 1, 2,  ,  }   #=2   6
{ ,  , 2, 3,  }   #=2   7
{0,  ,  , 3, 4}   #=3   8
{ , 1,  ,  , 4}   #=2   9
{0,  , 2,  ,  }   #=2   10
{ , 1,  , 3,  }   #=2   11
{ ,  , 2,  , 4}   #=2   12
{0,  ,  , 3,  }   #=2   13
{0, 1,  ,  , 4}   #=3   14
{0, 1, 2,  ,  }   #=3   15
{ , 1, 2, 3,  }   #=3   16
{0,  , 2, 3, 4}   #=4   17
{ , 1,  , 3, 4}   #=3   18
{0,  , 2,  , 4}   #=3   19
{0, 1,  , 3,  }   #=3   20
{ , 1, 2,  , 4}   #=3   21
{0,  , 2, 3,  }   #=3   22
{0, 1,  , 3, 4}   #=4   23
{0, 1, 2,  , 4}   #=4   24
{0, 1, 2, 3,  }   #=4   25
{0, 1, 2, 3, 4}   #=5   26
{ , 1, 2, 3, 4}   #=4   27
{ ,  , 2, 3, 4}   #=3   28
{ ,  ,  , 3, 4}   #=2   29
{ ,  ,  ,  , 4}   #=1   30
{ ,  ,  ,  ,  }   #=0   31
```

The underlying shift register sequence (SRS) is

$$0000010001100101001110101101111 1$$

(rotated left in the example so that the empty set appears at the end). Each subset is made from its predecessor by shifting it to the right and inserting the current element from the SRS.

The utility class [FXT: `class subset_debruijn` in `comb/subsetdebruijn.h`] uses [FXT: `class debruijn` in `comb/debruijn.h`] (which in turn uses [FXT: `class prime_string` in `comb/primestring.h`]). An algorithm for the generation of binary SRS is given in section 12.2 on page 254

The list above was created via

```
ulong n = 5;
subset_debruijn sdb(n);

for (ulong j=0; j<=n; ++j)  sdb.next();  // cosmetics: end with empty set

ulong ct = 0;
do
{
    ulong num = print_delta_set_as_set("", sdb.data(), n, 1);;
    cout << "   #=" << num;
    cout << "   " << ct;
    cout << endl;

    sdb.next();
```

```
    }
    while ( ++ct < (1UL<<n) );
```

Chapter 12

Shift register sequences

12.1 Linear feedback shift register (LFSR)

A binary shift register sequence (SRS) is a (periodic) sequence of zeros and ones that contains all nonzero words of length n. As an example the following sequence (that is periodic with period 15)

```
100110101111000
```

has the stated property. This can be seen more easily when displayed together with the words corresponding to the position (dots for zeros):

```
111.  .
11..  .
1...  .

..i1  1
.i1.  .
i1..  .
1..i  1
..ii  1
.ii.  .
ii.i  1
1.i.  .
.i.i  1
i.ii  1
.iii  1
iiii  1
iii.  .
ii..  .
i...  .

..i1  1
.i1.  .
i1..  .
```

The words to the left are obtained by combining the current bit and the $n-1=3$ bits left of it.

The sequence can be obtained by computing $A_k = x^k$, $k = 0, 1, \ldots, 2^n - 1$ modulo the polynomial $C = x^4 + x + 1$ and setting bit k of the SRS to the least significant bit of (equivalently: constant term of) A_k.

This is demonstrated in [FXT: file `demo/lfsr-demo.cc`], which for $n = 4$ gives the output:

```
poly = 1..11 == 0x13 == 19  (deg = 4)
   0     a= ..i1  w= 1111 = 15
   1     a= .i1.  w= 111. = 14
   2     a= i1..  w= 11.. = 12
   3     a= 1...  w= 1... = 8
   4     a= ..ii  w= ...1 = 1
   5     a= .ii.  w= ..i. = 2
   6     a= ii..  w= .i.. = 4
   7     a= 1.ii  w= 1..i = 9
   8     a= .1.1  w= ..11 = 3
   9     a= i.i.  w= .11. = 6
  10     a= .iii  w= 11.i = 13
  11     a= iii.  w= 1.i. = 10
  12     a= 111i  w= .1.i = 5
  13     a= 11.1  w= i.11 = 11
  14     a= 1..1  w= .111 = 7
```

12.2 Generation of binary shift register sequences

We restrict our attention to *binary* polynomials (that is, polynomials over $Z/2Z$) as computations are especially easy with those: when represented as binary words (bits as coefficients) both addition and subtraction are XOR, multiplication by x is just a shift to the left.
Consider the sequence of successive (polynomial) values $A_k := A_0\,x^k$ modulo C for $A_0 \neq 0$, $k = 0, 1, 2, 3 \ldots$. Here is id a polynomial over $GF(2)$, that is, with coefficients modulo two. Note that the computations are modulo a polynomial with coefficients modulo two.

The multiplication (of A_k) be x modulo C is easily done by shifting to the left, then, if coefficient n of A is non-zero, subtract C (that is, XOR it).

The utility class that can be used is [FXT: `class lfsr` in `auxbit/lfsr.h`], where the crucial computation is implemented as

```
ulong next()
{
    ulong s = a_ & h_;
    a_ <<= 1;
    w_ <<= 1;
    if ( 0!=s )
    {
        a_ ^= c_;
        w_ |= 1;
    }
    w_ &= m_;
    return w_;
}
```

This needs about 10 cycles per call. The underlying mechanism (shifting and feeding back the bit just shifted out) is called a *linear feedback shift register* (LFSR).

Using $n = 5$ and $c = x^5 + x^2 + 1$ we find

```
k       A(k)
1    a= ..1.1
2    a= .1.1.
3    a= 1.1..
4    a= .11.1
5    a= 11.1.
6    a= 1...1
7    a= ..111
8    a= .111.
9    a= 111..
10   a= 111.1
11   a= 11111
12   a= 11.11
13   a= 1..11
14   a= ...11
15   a= ..11.
16   a= .11..
17   a= 11...
18   a= 1...1
19   a= .1111
20   a= 1111.
21   a= 11...1
22   a= 1.111
23   a= .1.11
24   a= 1.11.
25   a= .1...1
26   a= 1...1.
27   a= ....1
28   a= ...1.
29   a= ..1..
30   a= .1...
31   a= 1....
32   a= ..1.1 == A(1)
```

The sequence A_k is periodic with period $31 = 2^5 - 1$. In fact $m = 2^n - 1$ is the maximal possible period for a polynomial C of degree n as there are just m non-zero binary polynomials at all. The choice of C is not arbitrary, with 'bad' polynomials C one gets sequences with a period less than $2^n - 1$ and therefore only a subset of the binary words.

First, the polynomials must be irreducible: A polynomial $C = c_0 + c_1\,x + c_2\,x^2 + \ldots + c_n\,x^n$ is called *irreducible* if it has no non-trivial factorization.

Consider only polynomials with a constant term (else x would be a factor and the polynomial would not be

irreducible). Then the period is determined by the minimal m for which C divides $Q(m) := x^m - 1$ (which is equal to $x^m + 1$ for binary polynomials). Polynomials that give maximal period are called *primitive*. We'll abbreviate primitive polynomials as PP. PP are always irreducible, irreducible polynomials are not necessarily primitive.

Lists of primitive binary polynomials look like

```
 2,1,0
 3,2,0
 4,3,0  // == x^4 + x^3 + 1
 5,3,2,1,0
 6,5,0
 7,6,3,1,0
 8,6,5,3,0
 9,5,3,2,0
 10,8,4,3,0
 11,10,8,7,6,4,2,1,0
 12,11,10,9,4,3,0
...
 31,29,23,21,19,17,15,12,10,8,7,5,4,2,0
 32,31,29,28,26,24,23,18,10,9,8,7,6,5,4,1,0
 33,31,29,28,27,25,20,19,17,13,11,10,8,6,5,1,0
 34,33,31,27,25,24,22,18,16,15,14,12,11,10,9,8,7,6,4,2,0
...
```

Each line consists of those powers of x where C has a non-zero coefficient. The coefficients c_0 and c_n are never zero so the first entry in each line gives the degree. In fact the above list gives 'random' PPs that where created by testing random binary words for primitivity when used as a binary polynomial; cf. [FXT: file `data/rand-primpoly.txt`].

The SRS generated with all PPs for degree $2 \le n \le 8$ are computed with [FXT: file `demo/allprimpoly-demo.cc`]. For $n = 6$ there are 6 different PPs and the output is:

```
degree = 6
c= 1...11  srs=.....1...11..1.1..1111.1...111..1..1.11.111.11..11.1.1.111111
c= 1.11.1  srs=.....1.111111..1.1.1...11..1111.111.1.11.1..11.11...1..1...111
c= 11...1  srs=.....111111.1.1.11..11.111.11.1..1..111...1.1111..1.1...11....1
c= 11..111  srs=.....1111..1..1.1.1..11.1....1...1.11.111111.1.111...11..111.11
c= 11.11.1  srs=.....111....1..1...11.11..1.11.1.111.1111..11...1.1.1..111111.1
c= 111..11  srs=.....11.111..11...111.1.111111.11.1...1....1.11..1.1.1..1..1111
```

These are *not* all possible SRS for $n = 6$, there are much more of them. 67,108,864 to be exact.

An exhaustive search for all SRS of given length $L = 2^n$ is possible only for tiny n. [FXT: file `demo/allsrs-demo.cc`] finds all SRS for $n = 3, 4, 5$. Its output with $n = 4$ is

```
n = 4  nn = 16
   ....1111.11..1.1
   ....1111.1.11..1
   ....1111.1..1.11
   ....1111..1.11.1
   ....11.1111..1.1
   ....11.1.111.1..1
   ....11.1..1.1111
   ....11..1.1111.1
   ....1.1111.1...11
   ....1.1111..11.1
   ....1.11.1..1111
   ....1.11..1111.1
   ....1.1...1111.11
   ....1.1..11.1111
   ....1..1111.1.11
   ....1..11.1.1111
total # of SRS found = 16
```

The last SRS is the one that is produced by the algorithm given by Knuth ([12], 2A: "Generating all n-tuples") which is implemented in [FXT: file `comb/binarydebruijn.h`]. Similarly for $n = 5$, which gives

```
n = 5  nn = 32
   .....11111.111..11.1.11...1.1..1
   .....11111.111..11.1.11...1..1.1
   .....11111.111..11.1.1...1.11...1
   .....11111.111..11.1.1...1...1.11
   .....11111.111..11.1.1...1.1.11
 [-snip-]
   .....1...11..1.1.11.1..111.11111
   .....1...11..1.1..11111.11.1.111
   .....1...11..1.1..11111.1.11.111
   .....1...11..1.1..111.11.1.11111
   .....1...11..1.1..111.1.11.11111
total # of SRS found = 2048
```

For the search only the cyclic minima of the sequences are considered. The bit-combinations are generated with the functions for bit-reversed combination in lexicographic order given on page 169. The run for $n = 5$ completes in a few seconds, with the algorithm used $\binom{25}{14} = 4,457,400$ bit-patterns are tested. For $n = 6$ the number of bit-combinations to check would already equal $\binom{56}{30} = 6,646,448,384,109,072$.

The total number of SRSs is $S_n = 2^x$ where $x = 2^{n-1} - n$:

n	$L_n = 2^n$	x	$S_n = 2^x$
1	2	0	1
2	4	0	1
3	8	1	2
4	16	4	16
5	32	11	2048
6	64	26	67108864
7	128	57	144115188075855872
8	256	120	1329227995784915872903807060280344576

The second column gives the length of the SRSs. One has $S_{n+1} = S_n^2 L_{n-1}$, equivalently $x_{n+1} = 2\,x_n + n - 1$.

The two SRSs for $n = 3$ are `[...111.1]` and `[...1.111]`, reversed sequences are considered different with the above formula. The general formula for the number of base-m SRSs is $S_n = m!^{m^{n-1}}/m^n$, as given in [12]. A graph theoretical proof for the case $m = 2$ can be found in [14], p.56.

12.2.1 Searching primitive polynomials

While computer algebra systems and number-theoretic libraries usually have a built-in function for testing irreducibility a routine for checking primality is most often lacking. The methods sometimes given such as checking that $mod(x^k, c) \equiv 1$ for all divisors of the maximal order that are greater than n get impractical quickly as n grows. Already with $n = 144$ one has $m = 2^{144} - 1$ (which has 262144 divisors of which 262112 have to be tested) so the computation gets expensive.

A better solution is a modification of the algorithm to determine the oder in a finite field given on page 72. The implementation given here uses the pari/gp (entry [75] in the bibliography) language:

```
nn = 0;  /* max order  =  2^n-1 */
np = 0;  /* number of primes in factorization */
vp = []; /* vector of primes */
vf = []; /* vector of factors (prime powers) */
vx = []; /* vector of exponents */

polorder(p) =
/* order of the polynomial p */
{
  local(g, g1);
  local(te);
  local(tp, tf, tx);

  g = x;
  te = nn;
  for(i=1, np,
    tf = vf[i];  tp = vp[i];  tx = vx[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
    );
  );

  return( te );
}
```

As given, the algorithm will do n_p exponentiations modulo p where n_p is the number of different primes

in the factorization in m. For $n = 144$ one has $n_p = 17$. Note that for prime $m = 2^n - 1$ (that is, m a Mersenne prime) irreducibility suffices for primality[1].

A shortcut that makes the algorithm terminate as soon as the computed order drops below maximum is

```
polmaxorder_q(p) =
/* early-out variant */
{
  local(g, g1);
  local(te);
  local(tp, tf, tx);
  local(ct);

  g = x;
  te = nn;
  for(i=1, np,
    tf = vf[i];  tp = vp[i];  tx = vx[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    ct = 0;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
      ct = ct + 1;
    );
    if ( ct<tx,  return(0) );
  );

  return(1);
}
```

Using `polmaxorder_q()` and pari's built-in `polisirreducible()` the search for the lowest-bit PPs up to degree $n = 400$ was a matter of 90 minutes on a 1,333MHz machine (AMD Athlon). Note that a table [FXT: file `data/mersenne-factorizations-400.txt`] of precomputed factorizations taken from [26] was used in order to save computation time.

The data in [FXT: file `data/minweight-primpoly.txt`] (and the corresponding [FXT: `ulong minweight_primpoly` in `auxbit/minweightprimpoly.h`]) lists minimal weight PPs where the coefficients/bits (apart from the constant and the leading term) are as close to the low end as possible.

For $n \le 400$ the entries where the second coefficient is bigger than in all prior lines are:

```
2,1,0
5,2,0
8,4,3,2,0
12,6,4,1,0
18,7,0
33,13,0
55,24,0
73,25,0
89,38,0
134,57,0
178,87,0
212,105,0
284,119,0
316,135,0
370,139,0
```

Primitive polynomials where the *weight* (that is, the number of nonzero coefficients is minimal) are given in [FXT: file `data/minweight-primpoly.txt`]. For many n there exist PPs with just three non-zero coefficients, these are called *trinomials*. Choosing those PPs where the highest nonzero coefficient is as low as possible one gets the list in [FXT: file `data/lowbit-primpoly.txt`].

```
2,1,0
3,1,0
4,1,0
5,2,0
6,1,0
```

[1] Thereby the one-liner `x=127;for(z=1,x-1,if(polisirreducible(Mod(1,2)+t^z+t^x),print1(" ",z)))` finds all primitive trinomials for certain exponents of Mersenne primes in no time: 89: 38 51 127: 1 7 15 30 63 64 97 112 120 126 521: 32 48 158 168 353 363 473 489 607: 105 147 273 334 460 502.

```
    7,1,0
    8,4,3,2,0
    9,4,0
    10,3,0
    11,2,0
    12,6,4,1,0
  [-snip-]
    31,3,0
    32,7,5,3,2,1,0
    33,6,4,1,0
    34,7,6,5,2,1,0
  [-snip-]
    397,8,7,6,5,1,0
    398,10,8,6,4,1,0
    399,9,6,4,2,1,0
    400,5,3,2,0
```

The corresponding data actually used for the computations can be found in [FXT: file `auxbit/lowbitprimpoly.h`]:

```
extern const ulong lowbit_primpoly[]=
{
//  hex_val,  // ==dec_val  (deg)  [weight]
 0x1,         // 1  (0)  [1]
 0x3,         // 3  (1)  [2]
 0x7UL,   // ==7  (2)  [3]
 0xbUL,   // ==11  (3)  [3]
 0x13UL,   // ==19  (4)  [3]
 0x25UL,   // ==37  (5)  [3]
 0x43UL,   // ==67  (6)  [3]
 0x83UL,   // ==131  (7)  [3]
 0x11dUL,   // ==285  (8)  [5]
 0x211UL,   // ==529  (9)  [3]
 0x409UL,   // ==1033  (10)  [3]
 0x805UL,   // ==2053  (11)  [3]
 0x1053UL,   // ==4179  (12)  [5]
[-snip-]
 0x80000009UL,   // ==2147483657  (31)  [3]
 0x1000000afUL,   // ==4294967471  (32)  [7]
 0x200000053UL,   // ==8589934675  (33)  [5]
 0x4000000e7UL,   // ==17179869415  (34)  [7]
[-snip-]
```

In fact the highest $k \neq n$ so that $c_k \neq 0$ grows slowly with n. For $n \leq 400$ the entries where the second coefficient is bigger than in all prior lines are:

```
    2,1,0
    5,2,0
    8,4,3,2,0
    12,6,4,1,0
    32,7,5,3,2,1,0
    46,8,5,3,2,1,0
    104,9,8,6,5,4,3,2,0
    108,10,9,7,6,5,4,3,2,1,0
    360,12,9,8,6,4,3,1,0
```

Thereby one can store the list in compact format even if it extends to high degrees.

Note that the 'reflected' form of a PP is again a PP: if $n, a, b, c, 0$ is primitive, then $n-a, n-b, n-c, 0$ is also primitive.

A list of all PPs for degree $n = 256$ with the second-highest order ≤ 15 is given in [FXT: file `data/lowbit256-primpoly.txt`]

```
    256,10,5,2,0
    256,10,8,5,4,1,0
    256,10,9,8,7,4,2,1,0
    256,11,8,4,3,2,0
    256,11,8,6,4,3,0
    256,11,10,9,4,2,0
    256,11,10,9,7,4,0
    256,12,7,5,4,2,0
    256,12,8,7,6,3,0
  [-snip-]
```

```
  256,15,14,13,12,10,9,8,7,6,5,4,3,2,0
  256,15,14,13,12,11,9,6,0
  256,15,14,13,12,11,9,8,6,4,3,2,0
  256,15,14,13,12,11,10,7,5,4,3,2,0
  256,15,14,13,12,11,10,8,6,3,2,1,0
[-end- (a few PP with second degree 16 follow)]
  256,16,3,1,0
  256,16,11,9,7,2,0
```

Similar tables for degrees 127, 128 and 521 can be found under the obvious names.

The total number of PPs of degree n is $P_n = \phi(2^n - 1)/n$:

n	P_n	n	P_n	n	P_n
1	1	11	176	21	84672
2	1	12	144	22	120032
3	2	13	630	23	356960
4	2	14	756	24	276480
5	6	15	1800	25	1296000
6	6	16	2048	26	1719900
7	18	17	7710	27	4202496
8	16	18	7776	28	4741632
9	48	19	27594	29	18407808
10	60	20	24000	30	17820000

If n is the exponent of a Mersenne prime we have $P_n = \frac{2^n-2}{n}$, this is almost the number of length-n necklaces: $P_n = N_n - 2$. For n a power of two the $P_n = 2^x$ where $x = 2^{n-2} - n - 1$, this is the number of length-$2n$ SRS.

For degree n up to 8 the complete list of PPs is (see [FXT: file `data/all-primpoly.txt`] which extends to degree 11):

```
2,1,0          5,2,0          7,1,0          8,4,3,2,0
               5,3,0          7,3,0          8,5,3,1,0
3,1,0          5,3,2,1,0      7,3,2,1,0      8,5,3,2,0
3,2,0          5,4,2,1,0      7,4,0          8,6,3,2,0
               5,4,3,1,0      7,4,3,2,0      8,6,4,3,2,1,0
4,1,0          5,4,3,2,0      7,5,2,1,0      8,6,5,1,0
4,3,0                         7,5,3,1,0      8,6,5,2,0
               6,1,0          7,5,4,3,0      8,6,5,3,0
               6,4,3,1,0      7,5,4,3,2,1,0  8,6,5,4,0
               6,5,0          7,6,0          8,7,2,1,0
               6,5,2,1,0      7,6,3,1,0      8,7,3,2,0
               6,5,3,2,0      7,6,4,1,0      8,7,5,3,0
               6,5,4,1,0      7,6,4,2,0      8,7,6,1,0
                              7,6,5,2,0      8,7,6,3,2,1,0
                              7,6,5,3,2,1,0  8,7,6,5,2,1,0
                              7,6,5,4,0      8,7,6,5,4,2,0
                              7,6,5,4,2,1,0
                              7,6,5,4,3,2,0
```

A list a primitive trinomials is given in [FXT: file `data/all-trinomial-primpoly.txt`]. Primitive trinomials of the most simple form $n, 1, 0 \sim x^n + x + 1$ exist for $n \leq 400$ and $n \in 2, 3, 4, 6, 7, 15, 22, 60, 63, 127, 153$. In fact these numbers are the sequence A073639 in [64], where one finds in addition $471, 532, 865, 900, 1366$ with the next candidate being 4495.

PPs with exactly five nonzero coefficients are given in [FXT: file `data/pentanomial-primpoly.txt`]. No primitive pentanomial exists for $n < 5$ but for all higher degrees one seems to exist (to my knowledge this has not been proven so far). Entries of the form $x^n+x^3+x^2+x+1$ are there for $n \in 5, 7, 17, 25, 31, 41, 151$ For $n \leq 400$ the entries where the second coefficient is bigger than in all prior lines are:

```
5,3,2,1,0
6,4,3,1,0
12,6,4,1,0
32,7,6,2,0
34,8,4,3,0
48,9,7,4,0
72,10,9,3,0
```

```
88,11,9,8,0
108,12,11,5,0
141,13,6,1,0
168,16,9,6,0
322,17,2,1,0
360,26,25,1,0
```

A list of PPs of the special form $x^n + \sum_{j=0}^{k} x^j$ is given in [FXT: file `data/lowblock-primpoly.txt`]. There an entry like $n, k =$ `16,5` should be read as `16,5,4,3,2,1,0` $= x^{16} + x^5 + x^4 + x^3 + x^2 + x + 1$. For $n \leq 400$ the entries where the second coefficient is bigger than in all prior lines are:

```
2,1
5,3
10,7
13,11
33,19
34,25
36,29
55,39
68,53
76,59
81,61
85,83
116,111
164,147
228,223
311,285
365,363
```

12.2.2 Irreducible polynomials of certain forms *

This section consists of some tables of irreducible polynomials of special forms.

Low-block and full polynomials

The list [FXT: file `data/all-lowblock-irredpoly.txt`] contains all irreducible polynomials of the form $x^s + \sum_{k=0}^{q} x^q$ where $q < s$ and $s \leq 400$. Note that the reversed polynomials are also irreducible. A subset of the list are those polynomials where all coefficients are one: The 'full' polynomial $p = \sum_{k=0}^{d-1} x^k = 1 + x + x^2 + \ldots + x^{d-1}$ can be irreducible only if d $(= s + 1)$ is prime. The first examples of such polynomials that are irreducible are

```
  d:   (irred. poly.)
  2:   x + 1
  3:   x^2 + x + 1
  5:   x^4 + x^3 + x^2 + x + 1
 11:   x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

In a way these are the 'Mersenne primes' among the polynomials over $GF(2)$. Note that, with the exception of $x^2 + x + 1$, none of them is primitive. The list of those primes up to $d = 2000$ is:

```
2, 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, 83, 101, 107, 131, 139,
149, 163, 173, 179, 181, 197, 211, 227, 269, 293, 317, 347, 349, 373,
379, 389, 419, 421, 443, 461, 467, 491, 509, 523, 541, 547, 557, 563,
587, 613, 619, 653, 659, 661, 677, 701, 709, 757, 773, 787, 797, 821,
827, 829, 853, 859, 877, 883, 907, 941, 947, 1019, 1061, 1091, 1109,
1117, 1123, 1171, 1187, 1213, 1229, 1237, 1259, 1277, 1283, 1291,
1301, 1307, 1373, 1381, 1427, 1451, 1453, 1483, 1493, 1499, 1523,
1531, 1549, 1571, 1619, 1621, 1637, 1667, 1669, 1693, 1733, 1741,
1747, 1787, 1861, 1867, 1877, 1901, 1907, 1931, 1949, 1973, 1979,
1987, 1997
```

it can be generated with the pari/gp one-liner:

```
forprime(d=2,N,s=sum(x=0,d-1,t^x);if(polisirreducible(Mod(1,2)*s),print1(", ",d)))
```

where `N` is the search limit.

Alternating polynomials

The 'alternating' polynomial $1 + \sum_{k=0}^{d} x^{2\,k+1} = 1 + x + x^3 + x^5 \ldots + x^{2d+1}$ can be irreducible only if d is odd:

```
d:   (irred. poly.)
1:    x^3 + x + 1
3:    x^7 + x^5 + x^3 + x + 1
5:    x^11 + x^9 + x^7 + x^5 + x^3 + x + 1
```

The list up to $d = 500$

```
1, 3, 5, 7, 9, 13, 23, 27, 31, 37, 63, 69, 117, 119, 173, 219, 223,
247, 307, 363, 383, 495
```

can be obtained via

```
for(d=1,N,p=(1+sum(t=0,d,x^(2*t+1)));if(polisirreducible(Mod(1,2)*p),print1(d,", ")))
```

Special trinomials I: $\mathbf{1 + x^d + x^{k\,d}}$

The polynomial $p = 1 + x^d + x^{2d}$ is irreducible whenever d is a power of three:

```
  1:  x^2 +  x +  1
  3:  x^6 +  x^3 +  1
  9:  x^18 +  x^9 +  1
 27:  x^54 +  x^27 +  1
 81:  x^162 +  x^81 +  1
243:  x^486 +  x^243 +  1
...
```

Similarly, $p = 1 + x^d + x^{3d}$ is irreducible whenever d is a power of seven:

```
  1:  x^3 +  x +  1
  7:  x^21 +  x^7 +  1
 49:  x^147 +  x^49 +  1
343:  x^1029 +  x^343 +  1
```

The test for the irreducibility of $p = 1 + x^d + x^{4d}$ can be done via

```
for(d=1,N,(p=(1+x^d+x^(4*d))); if(polisirreducible(Mod(1,2)*p),print(d,":  ",p)))
```

One gets

```
  1:  x^4 + x + 1
  3:  x^12 + x^3 + 1
  5:  x^20 + x^5 + 1
  9:  x^36 + x^9 + 1
 15:  x^60 + x^15 + 1
 25:  x^100 + x^25 + 1
 27:  x^108 + x^27 + 1
 45:  x^180 + x^45 + 1
 75:  x^300 + x^75 + 1
 81:  x^324 + x^81 + 1
125:  x^500 + x^125 + 1
135:  x^540 + x^135 + 1
225:  x^900 + x^225 + 1
243:  x^972 + x^243 + 1
375:  x^1500 + x^375 + 1
405:  x^1620 + x^405 + 1
...
```

The list of reversed polynomials

```
  1:  x^4 + x^3 + 1
  3:  x^12 + x^9 + 1
  5:  x^20 + x^15 + 1
  9:  x^36 + x^27 + 1
 15:  x^60 + x^45 + 1
 25:  x^100 + x^75 + 1
 27:  x^108 + x^81 + 1
 45:  x^180 + x^135 + 1
 75:  x^300 + x^225 + 1
 81:  x^324 + x^243 + 1
125:  x^500 + x^375 + 1
135:  x^540 + x^405 + 1
225:  x^900 + x^675 + 1
243:  x^972 + x^729 + 1
...
```

Might help to see that d always factors as $d = 3^i\,5^j, \quad i,j \in \mathbb{N}$, that is the list can be reproduced (and enhanced) using

```
fordiv((3*5)^7,i,s=(i);print1(" ",4*s))
```

```
  4, 12, 20, 36, 60, 100, 108, 180, 300, 324, 500, 540, 900, 972,
  1500, 1620, 2500, 2700, 2916, 4500, 4860, 7500, 8100, 8748, 12500,
  13500, 14580, 22500, 24300, 37500, 40500, 43740, 62500, 67500,
  72900, 112500, 121500, 187500, 202500, 218700, 312500, 337500,
  364500, 562500, 607500, 937500, 1012500, 1093500, 1687500, 1822500,
  2812500, 3037500, 5062500, 5467500, 8437500, 9112500, 15187500,
  25312500, 27337500, 45562500, 75937500, 136687500, 227812500,
  683437500, ...
```

Similar regularities can be observed for related forms, like

```
for(d=1,N,(p=(1+x^(5*d)+x^(6*d)));if(polisirreducible(Mod(1,2)*p),print(d,":  ",p)))
```

```
   1:  x^6 + x^5 + 1
   3:  x^18 + x^15 + 1
   7:  x^42 + x^35 + 1
   9:  x^54 + x^45 + 1
  21:  x^126 + x^105 + 1
  27:  x^162 + x^135 + 1
  49:  x^294 + x^245 + 1
  63:  x^378 + x^315 + 1
  81:  x^486 + x^405 + 1
 147:  x^882 + x^735 + 1
 189:  x^1134 + x^945 + 1
```

Special trinomials II: $\mathbf{1 + x^{d-k} + x^d}$

Irreducible polynomials of the form $p = 1 + x^{d-1} + x^d$. The first few are

```
  2:  x^2 +  x +  1
  3:  x^3 +  x^2 +  1
  4:  x^4 +  x^3 +  1
  6:  x^6 +  x^5 +  1
  7:  x^7 +  x^6 +  1
 ...
```

The list for $d \le 1000$:

```
    1, 2, 3, 4, 6, 7, 9, 15, 22, 28, 30, 46, 60, 63,
    127, 153, 172, 303, 471, 532, 865, 900, ...
```

The equivalent list for $p = 1 + x^{d-2} + x^d$ and $d \le 1000$:

```
    3, 5, 11, 21, 29, 35, 93, 123, 333, 845, ...
```

And for $p = 1 + x^{d-3} + x^d$ and $d \le 1000$:

```
    4, 5, 6, 7, 10, 12, 17, 18, 20, 25, 28, 31, 41, 52, 66,
    130, 151, 180, 196, 503, 650, 761, 986, ...
```

$p = 1 + x^{d-4} + x^d$ is irreducible for only few $d \le 1000$:

```
7, 9, 15, 39, 57, 81, 105, ...
```

$p = 1 + x^{d-5} + x^d$ and $d \le 1000$:

```
    6, 9, 12, 14, 17, 20, 23, 44, 47, 63, 84,
    129, 236, 278, 279, 297, 300, 647, 726, 737,
```

12.3 Computations with binary polynomials

Functions that operate on binary polynomials (see section 12.2) can be found in [FXT: file `auxbit/bitpolmodmult.h`]. To represent a polynomial over $Z/2Z$ as binary word one simply has to set a bit where the coefficient is one. We stick to the convention that the constant term goes to the lowest bit.

12.3.1 Basic operations

The following routines can be found in [FXT: file `auxbit/bitpol.h`].

Multiplication of two polynomials is identical to the usual (binary algorithm for) multiplication, except that no carry occurs:

```
inline ulong bitpol_mult(ulong a, ulong b)
// Return  A * B
// b=2 corresponds to multiplication with 'x'
// Note that the result silently overflows
//  if deg(A)+deg(B) > BITS_PER_LONG
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        a <<= 1;
    }
    return  t;
}
```

With a multiplication function at hand, it is straight forward to implement the algorithm for binary exponentiation: (Note that overflow will occur even for moderate exponents)

```
inline ulong bitpol_power(ulong a, ulong x)
// Return A ** x
{
    if ( 0==x )  return 1;

    ulong s = a;
    while ( 0==(x&1) )
    {
        s = bitpol_square(s);
        x >>= 1;
    }

    a = s;
    while ( 0!=(x>>=1) )
    {
        s = bitpol_square(s);
        if ( x & 1 )  a = bitpol_mult(a, s);
    }
    return  a;
}
```

The remainder with polynomial division can be implemented as

```
inline ulong bitpol_rem(ulong a, ulong b)
// Return  R = A % B = A - (A/B)*B
{
    while ( b <= a )
    {
        ulong t = b;
        while ( (a^t) > t )  t <<= 1;
        // =^= while ( highest_bit(a) > highest_bit(t) )  t <<= 1;
        a ^= t;
    }
}
```

Polynomial division

```
inline ulong bitpol_div(ulong a, ulong b)
// Return  R = A / B
{
    if ( b <= a )
    {
        ulong t = b;
        ulong qb = 1;
        while ( (a^t) > t )  { t <<= 1;  qb <<= 1; }
        ulong h = highest_bit(t);
        ulong q = 0;
        do
```

```
        {
            a ^= t;
            q ^= qb;
            do
            {
                t >>= 1;
                h >>= 1;
                qb >>= 1;
            }
            while ( h > a );
        }
        while ( t>=b );
        return  q;
    }
    else  return  0;
}
```

The polynomial greatest common divisor:

```
inline ulong bitpol_gcd(ulong a, ulong b)
// Return  polynomial gcd(A, B)
{
    if ( 0==a )  return 0;
    if ( 0==b )  return 0;
    while ( 0!=b )
    {
        ulong c = bitpol_rem(a, b);
        a = b;
        b = c;
    }
    return  a;
}
```

12.3.2 Computations modulo a polynomial

Here we consider arithmetic modulo a binary polynomial. The functions can be found in [FXT: file `auxbit/bitpolmodmult.h`].

Multiplication by x modulo (a polynomial) C is achieved by shifting left and subtracting (that is: XOR) C if the coefficient shifted out is one:

```
static inline ulong bitpolmod_times_x(ulong a, ulong c, ulong h)
// Return  (A * x) mod C
// where A and C represent polynomials over Z/2Z:
//  W = pol(w) =: \sum_k{ [bit_k(w)] * x^k}
//
// h needs to be a mask with one bit set:
//  h == highest_bit(c) >> 1  == 1UL << (degree(C)-1)
//
// If C is a primitive polynomial of degree n
// successive calls will cycle through all 2**n-1
// n-bit words and the sequence of bits
// (any fixed position) of a constitutes
// a shift register sequence (SRS).
// Start with a=2 to get a SRS that starts with
// n-1 consecutive zeroes (use bit 0 of a)
{
    ulong s = a & h;
    a <<= 1;
    if ( s )  a ^= c;
    return  a;
}
```

Multiplication also needs to take care of the modulus:

```
inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
// Return  (A * B) mod C
//
// With b=2 (== 'x') the result is identical to
//   bitpolmod_times_x(a, c)
{
```

```
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        ulong s = a & h;
        a <<= 1;
        if ( s )  a ^= c;
    }
    return  t;
}
```

Now exponentiation is identical to `bitpol_power()` except for the call to the modulo-multiplication functions:

```
inline ulong bitpolmod_power(ulong a, ulong x, ulong c, ulong h)
// Return (A ** x)  mod C
//
// With primitive C the inverse of A can be obtained via
// i = bitpolmod_power(a, c, r1, h)
//  where r1 = (h<<1)-2 = max_order - 1 = 2^degree(C) - 2
// Then  1 == bitpolmod_mult(a, c, i, h)
{
    if ( 0==x )  return 1;
    ulong s = a;
    while ( 0==(x&1) )
    {
        s = bitpolmod_square(s, c, h);
        x >>= 1;
    }
    a = s;
    while ( 0!=(x>>=1) )
    {
        s = bitpolmod_square(s, c, h);
        if ( x & 1 )  a = bitpolmod_mult(a, s, c, h);
    }
    return  a;
}
```

With primitive C the inverse of a polynomial A can be obtained by raising A to a power that equals the maximal order minus one:

```
    r1 = (h<<1)-2;  // = max_order - 1 = 2^degree(C) - 2
    i = bitpolmod_power(a, c, r1, h);
    // here:  1==bitpolmod_mult(a, i, c, h);
```

12.3.3 Testing for irreducibility

The following functions can be found in [FXT: file `auxbit/bitpolirred.h`].

The derivative of a binary polynomial can be computed like

```
inline ulong bitpol_deriv(ulong x)
// Return  derived polynomial
{
#if  BITS_PER_LONG >= 64
    x &= 0xaaaaaaaaaaaaaaaaUL;
#else
    x &= 0xaaaaaaaaUL;
#endif
    return  (x>>1);
}
```

The coefficients at the even powers have to be deleted because derivation multiplies them with an even factor (which is zero modulo two).

Now extraction of squared factors can be achieved via

```
inline ulong bitpol_test_squarefree(ulong x)
```

```
// Return  0 if polynomial is square-free
// else return square factor != 0
{
    ulong d = bitpol_deriv(x);
    if ( 0==d )  return  (1==x ? 0 : x);
    ulong g = bitpol_gcd(x, d);
    return  (1==g ? 0 : g);
}
```

Testing for irreducibility (that is, whether the polynomial has no non-trivial factors) uses the fact that the polynomial $x^{2^k}+x$ has all irreducible polynomials of degree k as a factor. For example,

$$\begin{aligned} x^{2^5}+x &= x^{32}+x \\ &= x\cdot(x+1)\cdot \\ &\quad \cdot\left(x^5+x^2+1\right)\cdot\left(x^5+x^3+1\right)\cdot\left(x^5+x^3+x^2+x+1\right)\cdot \\ &\quad \cdot\left(x^5+x^4+x^2+x+1\right)\cdot\left(x^5+x^4+x^3+x+1\right)\cdot\left(x^5+x^4+x^3+x^2+1\right) \end{aligned} \tag{12.1}$$

For a degree-d polynomial C one can compute $u_k = x^{2^k}$ (modulo C) for each $k < d$ by successive squarings and test whether $\gcd(C, u_k + x) \neq 1$ for all k. But as a factor of degree f implies another one of degree $d-f$ it suffices to do the first $\lfloor d/2 \rfloor$ of the tests.

```
inline ulong bitpol_irreducible_q(ulong c, ulong h)
// Return zero if C is reducible
// else (i.e. C is irreducible) return value != 0
// C must not be zero.
//
// h needs to be a mask with one bit set:
//  h == highest_bit(c) >> 1  == 1UL << (degree(C)-1)
{
    // if ( 0==(1&c) )  return (c==2 ? 1 : 0); // x is a factor
    // if ( 0==(c & 0xaa..a ) )  return 0; // at least one odd degree term
    // if ( 0==parity(c) )  return 0; // need odd number of nonzero coeff.
    // if ( 0!=bitpol_test_squarefree(c) )  return 0; // must be square free
    ulong d = c;
    ulong u = 2;  // =^= x
    while ( 0 != (d>>=2) )  // floor( degree/2 ) times
    {
        // Square r-times for coefficients of c in GF(2^r).
        // We have r==1
        u = bitpolmod_mult(u, u, c, h);
        ulong upx = u ^ 2;  // =^= u+x
        ulong g = bitpol_gcd(upx, c);
        if ( 1!=g )  return  0;   // reducible
    }
    return  1;  // irreducible
}
```

Let

$$z = x^{2^d}+x \tag{12.2}$$

$$s = 1+\sum_{k=0}^{d-1} x^{2^k} \tag{12.3}$$

$$t = s-1 \qquad = z/s \tag{12.4}$$

Then s is has all degree-d irreducible polynomials of trace 1 as factors and t those with trace 0. As an example consider the case $d=7$. We use the notation $[a,b,c,\ldots] := x^a + x^b + x^c + \ldots$:

```
z = [128,1]
s = [64,32,16,8,4,2,1,0] =
    [1,0] *
    [7,6,0] *
    [7,6,3,1,0] *
```

```
      [7,6,4,1,0] *
      [7,6,4,2,0] *
      [7,6,5,2,0] *
      [7,6,5,3,2,1,0] *
      [7,6,5,4,0] *
      [7,6,5,4,2,1,0] *
      [7,6,5,4,3,2,0]
  t = [64,32,16,8,4,2,1] =
      [1] *
      [7,1,0] *
      [7,3,0] *
      [7,3,2,1,0] *
      [7,4,0] *
      [7,4,3,2,0] *
      [7,5,2,1,0] *
      [7,5,3,1,0] *
      [7,5,4,3,0] *
      [7,5,4,3,2,1,0]
```

```
inline ulong bitpol_compose_xp1(ulong c)
// Return C(x+1)
// self-inverse
{
    ulong z = 1;
    ulong r = 0;
    while ( c )
    {
        if ( c & 1 )  r ^= z;
        c >>= 1;
        z ^= (z<<1);
    }
    return  r;
}
```

A version that avoids any branches and finishes in time $\log_2(b)$ (where $b =$ bits per word) is

```
inline ulong bitpol_compose_xp1(ulong c)
// Return C(x+1)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        c ^= ( (c&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  c;
}
```

Which is exactly the `blue_code()` from section 8.23.

When a polynomial is irreducible then the composition with $x+1$ is also irreducible. Similar, the reversed word corresponds to another irreducible polynomial. The two statements remain true if 'irreducible' is replaced by 'primitive' (see section 12.2.1).

```
inline ulong bitpol_recip(ulong c)
// Return x^deg(C) * C(1/x)  (the reciprocal polynomial)
{
    ulong t = 0;
    while ( c )
    {
        t <<= 1;
        t |= (c & 1);
        c >>= 1;
    }
    return  t;
}
```

In general the sequence of successive 'compose' and 'reverse' operations leads to 6 different polynomials:

```
C= [11, 10, 4, 3, 0]
```

```
[11, 10, 4, 3, 0]  -- recip (C=bitpol_recip(C)) -->
[11,  8, 7, 1, 0]   -- compose (C=bitpol_compose_xp1(C)) -->
[11, 10, 9, 7, 6, 5, 4, 1, 0]  -- recip -->
[11, 10, 7, 6, 5, 4, 2, 1, 0]  -- compose -->
[11,  9, 7, 2, 0]  -- recip -->
[11,  9, 4, 2, 0]  -- compose -->
[11, 10, 4, 3, 0]  == initial value
```

TBD: *factorization*

12.3.4 Modulo multiplication with reversed polynomials *

For the generation of shift register sequences (see section 12.2) a cycle-saving variant of the multiplication by x can be found in [FXT: file `auxbit/bitpolmodmultrev.h`]:

```
static inline ulong bitpolmod_times_x_rev(ulong a, ulong c)
// Return  (A * reverse(x)) mod C
// where A and C represent polynomials over Z/2Z:
//   W = pol(w) =: \sum_k{ [bit_k(w)] * x^k}
//
// If c is a primitive polynomial of degree n
// successive calls will cycle through all 2**n-1
// n-bit words and the sequence of bits
// (any fixed position) of a constitutes
// a shift register sequence (SRS).
// Start with a=(1UL<<(n-1)) to get a SRS
// that starts with n-1 consecutive zeroes
{
    if ( a & 1 )  a ^= c;
    a >>= 1;
    return  a;
}
```

For primitive polynomials see section 12.2.1.

Similarly, a multiplication by the a reversed polynomial can be implemented as:

```
inline ulong bitpolmod_mult_rev(ulong a, ulong b, ulong c)
// Return  (A * reverse(B)) mod C
// where A, B and C represent polynomials over Z/2Z:
//   W = pol(w) =: \sum_k{ [bit_k(w)] * x^k}
//
// With b=2 (== 'x') the result is identical to
//   bitpolmod_times_x_rev(a, c)
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        if ( a & 1 )  a ^= c;
        a >>= 1;
    }
    return  t;
}
```

12.4 Feedback carry shift register (FCSR)

There is a nice analogue of the LFSR in the modulo world, the *feedback carry shift register* (FCSR). With the LFSR we needed an irreducible ('prime') polynomial C where x has maximal order. The powers of x modulo C did run through all different (non-zero) words. Now take a prime c where 2 has maximal order (that is, 2 is s primitive root modulo c). Then the powers of 2 modulo c run through all non-zero values less than c.

The crucial part of the implementation is

```
ulong next()
{
    a_ <<= 1;
    if ( a_ > c_ )  a_ -= c_;

    w_ <<= 1;
    ulong s = a_ & 1;
    w_ |= s;

    w_ &= mask_;
    a_ &= mask_;
    return w_;
}
```

[FXT: `class fcsr` in `auxbit/fcsr.h`] The routine is much simpler than the approach usually found in the literature. It corresponds to the so-called Galois setup of the shift register. For a comparison of Galois- and Fibonacci- setup see [68].

A demo can be found in [FXT: file `demo/fcsr-demo.cc`], using $c = 37$ one gets

```
c = 1..1.1  = 37
 0 :   a= ....,1 =    1    w= ,1..11 =   19
 1 :   a= ...,i. =    2    w= i..ii. =   38
 2 :   a= ..,i.. =    4    w= .,ii.. =   12
 3 :   a= .,i... =    8    w= ,ii... =   24
 4 :   a= ,i.... =   16    w= ii.... =   48
 5 :   a= i..... =   32    w= i....., =   32
 6 :   a= .ii.ii =   27    w= ....,i =    1
 7 :   a= ,i..,1 =   17    w= ...,ii =    3
 8 :   a= i...i. =   34    w= ..ii., =    6
 9 :   a= .iiiii =   31    w= .,ii,i =   13
10 :   a= .ii..1 =   25    w= ,ii,ii =   27
11 :   a= ..ii,1 =   13    w= ii,iii =   55
12 :   a= .ii.i. =   26    w= i,iii, =   46
13 :   a= ..iiii =   15    w= ,iii,i =   29
14 :   a= .iiii, =   30    w= iii,i, =   58
15 :   a= .i.iii =   23    w= ii,i,i =   53
16 :   a= ..i..1 =    9    w= i,i,ii =   43
17 :   a= ,i..i. =   18    w= ,i,ii. =   22
18 :   a= i..i,, =   36    w= i,ii,, =   44
19 :   a= i...ii =   35    w= ,ii..i =   25
20 :   a= i....1 =   33    w= ii..ii =   51
21 :   a= .iii.i =   29    w= i..iii =   39
22 :   a= .i.i.i =   21    w= ,.iiii =   15
23 :   a= ..i.i. =    5    w= ,iiiii =   31
24 :   a= ,.i,i. =   10    w= iiiii. =   62
25 :   a= .i.i,, =   20    w= iiii,, =   60
26 :   a= ...,ii =    3    w= iii..i =   57
27 :   a= ...ii. =    6    w= ii..i. =   50
28 :   a= ..ii.. =   12    w= i..i.. =   36
29 :   a= .ii,, =   24    w= ,.i... =    8
30 :   a= .i.,ii =   11    w= ,i...i =   17
31 :   a= .i.ii, =   22    w= i...,i =   34
32 :   a= ..,iii =    7    w= ..i.,i =    5
33 :   a= ,.iii. =   14    w= ,.i,i. =   10
34 :   a= .iii,, =   28    w= ,i,i,, =   20
35 :   a= .i..ii =   19    w= i.i..i =   41

36 :   a= ....,1 =    1    w= ,1..11 =   19
37 :   a= ...,i. =    2    w= i..ii. =   38
38 :   a= ..,i.. =    4    w= .,ii.. =   12
39 :   a= .,i... =    8    w= ,ii... =   24
40 :   a= ,i.... =   16    w= ii.... =   48
41 :   a= i..... =   32    w= i....., =   32
42 :   a= .ii.ii =   27    w= ....,i =    1
```

Note that the w do not run through all values $< c$.

The primitive polynomials are replaced by the 'primitive' primes (those primes where 2 is a primitive root). The following list is complete for $c < 2048$:

```
 x:  p prime with 2 a primitive root, 2**x < p < 2**(x+1)
 1:  3
 2:  5
 3:  11 13
 4:  19 29
 5:  37 53 59 61
 6:  67 83 101 107
 7:  131 139 149 163 173 179 181 197 211 227
 8:  269 293 317 347 349 373 379 389 419 421 443 461 467 491 509
 9:  523 541 547 557 563 587 613 619 653 659 661 677 701 709 757
     773 787 797 821 827 829 853 859 877 883 907 941 947 1019
10:  1061 1091 1109 1117 1123 1171 1187 1213 1229 1237 1259 1277
     1283 1291 1301 1307 1373 1381 1427 1451 1453 1483 1493 1499
     1523 1531 1549 1571 1619 1621 1637 1667 1669 1693 1733 1741
     1747 1787 1861 1867 1877 1901 1907 1931 1949 1973 1979 1987
     1997 2027 2029
```

For the correspondence between LFSR and CFSR see [67].

12.5 Linear hybrid cellular automata (LHCA)

Consider 1-dimensional linear cellular automata (with 0 and 1 the only possible states) where two different rules are applied dependent[2] of the position:

```
inline ulong lhca_next(ulong x, ulong r, ulong m)
// return next state (after x) of the
// lhcr with rule defined by r, length defined by m:
// Rule 150 is applied for cells where r is one, rule 90 else.
// Rule 150 :=  next(x) = x + leftbit(x) + rightbit(x)
// Rule 90  :=  next(x) = leftbit(x) + rightbit(x)
// m has to be a burst of the n lowest bits (n: length of automaton)
{
    r &= x;
    ulong t = (x>>1) ^ (x<<1);
    t ^= r;
    t &= m;
    return  t;
}
```

[FXT: `lhca_next` in `auxbit/lhca.h`]

The naming convention for the rules is as follows: draw a table of the eight possible states of a cell together with its neighbors then draw the new states below:

```
  XXX  XX0  X0X  X00  0XX  0X0  00X  000
   0    X    0    X    X    0    X    0
```

Now read the lower row as a binary number, the result is 90, so this is rule 90. Rule 150 is

```
  XXX  XX0  X0X  X00  0XX  0X0  00X  000
   X    0    0    X    0    X    X    0
```

It turns out that for certain r the successive states x have the maximal period $m = 2^n - 1$, all non-zero values occur. This is demonstrated in [FXT: file `demo/lhca-demo.cc`], which for $n = 5$ (and rule $r = 1$) gives:

```
rule = ....1  == 0x1  length=5
   1   ...1
   2   ..11
   3   ..11.
   4   .1111
   5   11...
   6   11.1.
   7   1.11.
   8   .1111
   9   .11..
  10   1111.
  11   1..11
  12   .111.
  13   11.11
  14   11.1.
  15   11...1
  16   11111
  17   1....
  18   .1...
  19   1.1..
  20   ...1.
  21   .1.1.
  22   1...1
  23   1.111
  24   ..1..
  25   .1.1.
  26   1...1
  27   .1.11
  28   1...1.
  29   .11.1
  30   111.1
  31   1.1.1
```

There is an intimate connection between linear hybrid cellular automata (LHCA) and the linear feedback shift registers (LFSR, see also section 12.2 on page 254). This is nicely pointed out in the very readable paper [65] which is recommended for further studies.

Rule sets (with minimal weight) that lead to maximal period can be found in [66]. The list [FXT: `ulong minweight_lhca_rule` in `auxbit/minweightlhcarule.h`] was generated from that source:

[2] therefore the 'hybrid'.

```
#define R1(n,s1)     (1UL<<s1)
#define R2(n,s1,s2)  (1UL<<s1) | (1UL<<s2)
extern const ulong minweight_lhca_rule[]=
// LHCA rules of minimum weight that lead to maximal period.
{
      0,  // (empty)
 R1(  1,  0),
 R1(  2,  0),
 R1(  3,  0),
 R2(  4,  0, 2),
 R1(  5,  0),
 R1(  6,  0),
 R1(  7,  2),
 R2(  8,  1, 2),
 R1(  9,  0),
 R2( 10,  1, 6),
 R1( 11,  0),
 R2( 12,  2, 6),
 R1( 13,  4),
...
};
```

Up to $n = 500$ there is always a rule with weight at most 2. Quite pretty patterns emerge with LHCA, the following is the beginning part of the run for $n = 27$:

```
rule = .......1..................1  == 0x80001  length=27
   1   ...........................1
   2   ..........................11
   3   .........................11.
   4   ........................1111
   5   .......................11...
   6   ......................1111..
   7   .....................11..11.
   8   ....................11111111
   9   ...................11.......
  10   ..................1111......
  11   .................11..11.....
  12   ................11111111....
  13   ...............11......11...
  14   ..............1111....1111..
  15   .............11..11..11..11.
  16   ...........1111111111111111
  17   ..........11................
  18   ........1111................
  19   .......11..11...............
  20   ......11111111..............
  21   .....1.....11...............
  22   ....1.1....1111.............
  23   ...1..11...11..11...........
  24   ..1.11.11.11111111..........
  25   .1...11.11.1.....11.........
  26   1.1111.11..1...1111.........
  27   ...1...1.1111.1..11..11.....
  28   ..11.11..1..1..1111111111...
  29   111.1111.11.111.......11....
  30   1.1.1...11.1.11.....1111....
  31   .....1..111...111...11..11..
  32   ....1.111.11.11.11.11111111
  33   ...1..111.11.11.11.1........
  34   ..1.11111.11.11.11..1.......
  35   .1..1..11.11.11.1111.1.....
  36   1.11.11.1.11.11.1...1..1....
  37   ..11.11...11.11..11.11.1...
  38   .111.111.111.111111.11...1..
  39   11.1.1...1.1.1....1.1111.1.
  40   11...1.1.....1...1...1...1..1
  41   111...1...1...1.11.11.11.111
  42   1.111.1.1.1.1..11.11.11.1...
  43   ..1.1........1111.11.11..1.
```

12.6 Necklaces

A sequence that is minimal among all its cyclic rotations is called a necklace. When there are k possible values for each element one talks about an n-bead, k-color (or k-ary length-n) necklaces. We restrict our attention to the case were only two sorts of beads are allowed and represent them by 0 and 1.

Scanning all binary words of length n as to whether they are necklaces can easily be achieved by testing whether `x==bit_cyclic_min(x,n)` (see section 8.14). For $n = 7$ one gets the sequence of binary necklaces of length n (see [FXT: file `demo/necklace-demo.cc`]):

```
.......  = 0
......i  = 1
```

```
.....11  = 3
....1.1  = 5
....111  = 7
...1..1  = 9
...1.11  = 11
...11.1  = 13
...1111  = 15
..1..11  = 19
..1.1.1  = 21
..1.111  = 23
..11.11  = 27
..1111.  = 29
..11111  = 31
.1.1.11  = 43
.1.1111  = 47
.11.111  = 55
.111111  = 63
1111111  = 127
 n = 7:  20 necklaces
```

The number of binary necklaces of length n is

$$N_n \quad = \quad \frac{1}{n}\sum_{d\backslash n}\varphi(d)\,2^{n/d} \quad = \quad \frac{1}{n}\sum_{j=1}^{n}2^{\gcd(j,n)} \tag{12.5}$$

Replace 2 by r to get the number for r different sorts of beads (possible values at each digit), see [13], pp.139-141.

n	N_n	n	N_n	n	N_n
1	2	11	188	21	99880
2	3	12	352	22	190746
3	4	13	632	23	364724
4	6	14	1182	24	699252
5	8	15	2192	25	1342184
6	14	16	4116	26	2581428
7	20	17	7712	27	4971068
8	36	18	14602	28	9587580
9	60	19	27596	29	18512792
10	108	20	52488	30	35792568

For prime $n = p$ we have

$$N_p \quad = \quad 2 + \frac{2^p - 2}{p} \quad = 2 + \frac{1}{p}\sum_{k=2}^{p-1}\binom{p}{k} \tag{12.6}$$

The latter form transpires that there are exactly $\binom{p}{k}/p$ necklaces with k elements for $2 \le k \le p-1$ plus the two elements that consist of all zeros or ones.
Not all necklaces are created equal: all can be considered periodic with a period that is a divisor of the length. The necklaces of length $n = 6$ together with their periods are

```
......   1
.....1   6
....11   6
...1.1   6
...111   6
..1..1   3
..1.11   6
..11.1   6
..1111   6
.1.1.1   2
.1.111   6
.11.11   3
.11111   6
111111   1
```

For n prime the only periodic necklaces are those two that contain all ones or zeros. Aperiodic (or eqivalently, period equals length) necklaces are also called *Lyndon words*. The number of aperiodic necklaces is

$$M_n \quad = \quad \frac{1}{n}\sum_{d\backslash n}\mu(d)\,2^{n/d} \quad = \frac{1}{n}\sum_{d\backslash n}\mu(n/d)\,2^{d} \tag{12.7}$$

n	M_n	n	M_n	n	M_n
1	2	11	186	21	99858
2	1	12	335	22	190557
3	2	13	630	23	364722
4	3	14	1161	24	698870
5	6	15	2182	25	1342176
6	9	16	4080	26	2580795
7	18	17	7710	27	4971008
8	30	18	14532	28	9586395
9	56	19	27594	29	18512790
10	99	20	52377	30	35790267

The number of of degree-n irreducible polynomials with coefficient modulo two is also M_n. The exception is $n = 1$ where there is just one polynomial. The entry for $n = 1$ is special anyway because both '0' and '1' are of the maximal period. For the equivalence between necklaces and irreducible polynomials over GF(2) see [71]. The same source gives a constant amortized time (CAT) algorithm to generate all k-ary length-m necklaces:

```
long *a; // data in a[1..m],  a[0] = 0
long m; // length
long k; // k-ary

void crsms_print(long p)
{
    long mm = m;
    // no condition: pre-necklace
//    if ( p!=m )  return;  // Lyndon words
    if ( 0!=(m%p) )  return;  // necklaces
//    if ( 0!=(m%p) )  return;  else mm = p; // de Bruijn seq

    for (long j=1; j<=mm; ++j)  cout<< " " << a[j];
    cout << endl;
}

void crsms_gen(long n, long p)
{
    if ( n > m )  crsms_print(p);
    else
    {
        a[n] = a[n-p];
        crsms_gen(n+1, p);

        for (long j=a[n-p]+1; j<k; ++j)
        {
            a[n] = j;
            crsms_gen(n+1, n);
        }
    }
}

int main(int argc, char **argv)
{
    m = 4; // length
    if ( argc>1 )  m = atol(argv[1]);

    k = 2;  // k-ary
    if ( argc>2 )  k = atol(argv[2]);

    long aa[m+1];
    a = aa;
    a[0] = 0;

    crsms_gen(1, 1);
    return 0;
}
```

Depending the function `crsms_print` one can also generate pre-necklaces, Lyndon words or a de Bruijn sequence.

The binary necklaces of length n can be used as cycle leaders in the length-2^n zip-permutation (and its inverse) that is presented in section 7.5.

The length-n necklaces for n an exponent of a Mersenne prime $m = 2^n - 1$ can be generated efficiently by simply going through the powers of a primitive root r of m until the second word with just one bit is reached. With $n = 7$, $m = 127$ and the primitive root $r = 3$ we get

```
  0 :   a= .....1 =    1    =^= .....1
  1 :   a= ...ii1 =    3    =^= ...ii1
  2 :   a= ..i..1 =    9    =^= ..i..1
  3 :   a= ..ii.1 =   27    =^= ..ii.1
  4 :   a= i.i...1 =   81    =^= ...ii.1
  5 :   a= iii.i.. =  116    =^= ..iii.1
  6 :   a= i.iii. =   94    =^= .i.iii1
  7 :   a= ..iii.. =   28    =^= ...iii1
  8 :   a= i.i.i.. =   84    =^= ..iii.1
  9 :   a= iiiii.1 =  125    =^= .iiiii1
 10 :   a= iiii..1 =  121    =^= ..iiii1
 11 :   a= ii.ii.1 =  109    =^= .ii.ii1
 12 :   a= i..i..1 =   73    =^= ..i..ii
 13 :   a= i.iii.. =   92    =^= ..i.iii
 14 :   a= ..i.ii. =   22    =^= ...i.ii
 15 :   a= i...i.. =   66    =^= ...i.i1
 16 :   a= i..iii1 =   71    =^= ..iii1
 17 :   a= i.i.ii. =   86    =^= .i.i.ii

 18 :   a= ...i.. =    4    =^= .....1
 19 :   a= ..ii.. =   12    =^= ....ii
 20 :   a= .i..i.. =   36    =^= ..i..1
 21 :   a= ii.ii.. =  108    =^= ..ii.ii
 22 :   a= i..ii. =   70    =^= ..ii.1
 23 :   a= i.i..ii =   83    =^= .iii.1
 24 :   a= iiii.i. =  122    =^= .i.iii
 25 :   a= iii.... =  112    =^= ....iii
 [...]
```

The two words of all ones/zeros have to be added.

12.7 Necklaces and Gray codes *

There are 8 binary necklaces of length 5:

```
....i
...ii
..i1i
..i.1
.iii1
.i.ii
.iiii
iiiii
```

Discard the first and last necklace (those with all zeros/ones) and create a Gray code of the remaining words by hand. The first and last words have to match:

```
...i1
.i.ii
.iiii
..iii
..i.1
```

A Gray code can now be obtained by appending rotated versions of the block like this (at each stage, the block is left-rotated by two):

```
...i1
.i.ii
.iiii
..iii
..i.1

.i1..
.ii..
.ii.i
iii.1
iii..
i.i..

i...i
i.i.1
i.iii
i..ii
i..i.

..i1.
.ii..
i.ii.
iiii.
.iii.
```

```
.1.1.

.1...
11...
11.1.
11.11
11..1
.1..1
```

By construction the number of zero-one transitions is equal for each track. The given sequence omits the two words we discarded above. The zero could of course be prepended, the all-ones word, however, does not fit in. One exception is $n = 3$ were a monotonic Gray code is obtained:

```
..1
.11
.1.
11.
1..
1.1
111
```

A complementary (the second half of the code is the inverted first half) 4-bit code is easily obtained:

```
...1
..11
.1..
.11.
1...
1.11
1111
111.
11..
11.1
1..1
1.11
1.1.
1...
```

The construction works only if the word length is prime, only then it is guaranteed that the n cyclic shifts of the block contain all different words.
Here is how it fails for composite n: the construction with necklaces for $n = 4$ gives

```
...1    ..1.    .1..    1...
.1.1    1.1.    .1.1(!) 1.1.(!)
.111    111.    11.1    1.11
..11    .11.    11..    1..1
```

Note all hope is lost. Deleting the elements where they appear for the second time, reordering and inserting the zero and the word of ones we get

```
...1  0
..11  1
.1.1  2
.111  3
..11  2
..1.  1
1.1.  2
1111  3
.11.  2
.1..  1
11..  2
11.1  3
1111  4
1.11  3
1..1  2
1...  1
```

Transition counts are equal for each track. The numbers of the right column are bit counts.

For $n = 7$ one can use one of these (handwoven) blocks of length 18

```
.....1    1.....
...1.1    11....
..1.1.1   11..1.
..111.1   11.11.
...11.1   11.1..
...1..1   1111..
..1.11    11111.
.1.1.11   111111
.1.1111   1111.1
..1111    .111.1
..1.111   .111.11
..1.111   .111..1
.11.111   .111...
.111111   .1.1...
..11111   .1.1.1.
..11.11   11.1.1.
..1..11   11...1.
.....11   .1...1.
```

and six consecutive copies that are rotated left by one (right by one for the right block).

A degree of freedom that can be use is that the words in the basic block can be rotated at will. For example, for $n = 5$ one could use either of

```
 ...1      ...1     ...1
i..i1     ..ii1    ..ii1
ii..1     .iii1    .i.11
1ii.1     .1111    .1.1.
1.1.1     .1.11    .111.
..1.1     .1..1    1111.
```

Now use the elements of the third basic block in pairs that match to get a[3] monotonic Gray code:

```
....i    0  ....i
...i1    1  ...i1
...11    2  ...11
..1..    3
..11.    4
.1...    5
.11..    6
.1...    7
11...    8
1....    9
1...i   10
1.1.1   11  .1.11
1.1..   12  ..1.1
1.11.   13
1..1.   14
11.1.   15
.1.1.   16
.1.1i   17
.1..1   18
.11.1   19
..1.1   20
..111   21  ..111
.1111   22  .1111
 111.   23
1111.   24
111..   25
111.1   26
11..1   27
11.11   28
1..11   29
1.111   30
11111   31  11111
```

The first appearance of the necklaces is indicated on the right. Reordering slightly (not separating the runs of necklaces) we get a Gray code that is monotonic in two ways. The code on the right side is the 'unwrapped' version:

```
 0  .....    0  ....i     0  .....
 1  i....    9  ...i1     9  i....
 2  11...    8  ...11     8  11...
 3  .1...    7             7  .1...
 4  .11..    6             6  .11..
 5  ..1..    5             5  ..1..
 6  ..11.    4             4  ..11.
 7  ...1.    3             3  ...1.
 8  ...11    2             2  ...11
 9  ....1    1             1  ....1
10  1...1   10            10   ...11
11  1.1.1   11  .1.11      11     .11.1
12  1.1..   12  ..1.1      12      ..1.1
13  1.11.   13              13       .1.11
14  1..1.   14              14       .1..1
15  11.1.   15              15       .11.1
16  .1.1.   16              16       ..1.1
17  .1.1i   17               .1.11
18  .1..1   18               .1..1
19  .11.1   19               .11.1
20  ..1.1   20               ..1.1
21  ..111   21  ..111        ..111
22  .1111   30  .1111         .1111
23  1.111   29               ..1111
24  11.11   28                .1111
25  11..1   27                ..111
26  111.1   26                 .1111
27  111..   25                 ..111
28  1111.   24                  .1111
29  .111.   23                  ..111
30  .1111   22                   .1111
31  11111   31  11111           11111
```

The negated sequence in reversed order is (if bit positions are taken as [01234]) identical to the original when the bits are reordered as [04321] (that is, permuted as if indices multiplied with minus one modulo five).

An analogue basic block for a 7-bit monotone Gray code would be:

[3]This actually is the Savage-Winkler Gray code for $n = 5$, I constructed it by hand using my editor.

```
.:.....  0  .......
.i.....  1  ......i
ii.....  2  .....ii
ii..i..  3  ..i..ii
i...i..  2  ...i..i
i...i.i  3  ...i.ii
...i.i.  2  ...i.i.
..ii.i.  3  ..ii.i.
..iii.i  4  ..iii.i
..i.i.i  3  ..i.i.i
..i.iii  4  ..i.iii
...iiii  3  ...iiii
.i.iiii  4  .i.iiii
.i.i.ii  5  .i.i.ii
.iii.ii  5  .ii.iii
..ii.ii  4  ..ii.ii
..iiiii  5  ..iiiii
.iiiiii  6  .iiiiii
iiiiiii  7  iiiiiii
```

A slight reordering gives the (reversed-negated) symmetry we have seen for $n = 5$:

```
.:.....  0  .......
.i.....  1  ......i
ii.....  2  .....ii
ii..i..  3  ..i..ii
i...i..  2  ...i..i
i...i.i  3  ...i.ii
....i.i  2  ....i.i
...iii.  3  ...iii.
..i.iii  4  ..i.iii
.i.i.i.  3  .i.i.i.
.i.i.ii  4  .i.i.ii
i...i.i  3  ...ii.i
iii...i  4  ...iiii
iii.i.i  5  .i.iiii
iii.i..  4  ..iii.i
iii.ii.  5  .ii.iii
ii..ii.  4  ..ii.ii
ii.iiii  5  .iiiiii
ii.iiii  6  .iiiiii
iiiiiii  7  iiiiiii
```

The right column gives the cyclic minimum (necklace). However, it is not possible to build a monotonic Gray code for $n > 5$ where pairs of necklaces appear interlaced in all their cyclic shifts. The reason is, that the pairs have to match in *two* ways. Take, as an example the two necklaces `..1.1` and `.1.11` and $n = 5$:

```
1.1.1     .11.1
i.i.      ..i.i
i.ii.      .i.ii
i..i.      .i..i
ii.i.      .ii.i
.i.i.      ..i.i
.i.ii      .i.ii
i..i.      .i..i
ii.i.      .ii.i
..i.i      ..i.i
```

The same two necklaces match in just one way with $n = 7$.

When we sort the necklaces by their bit-count and tabulate the ways necklace k matches with necklace j we get the following matrix for $n = 5$:

```
      01234567 < j
      01223345 < bit_count(necklace[j])       p= period
0:  0 .*......  0    0                .....   p= 1
1:  1 *.22....  1    2                ....1   p= 5
2:  2 .2..12..  2    2                ...11   p= 5
3:  2 .2..21..  3    2                ..1.1   p= 5
4:  3 ..12..2.  4    2                11.1.   p= 5
5:  3 ..21..2.  5    2                111..   p= 5
6:  4 ....22.*  6    2                1111.   p= 5
7:  5 ......*.  7    0                11111   p= 1
^   ^           ^    ^                ^
k   bitcount    k    number of        necklace
                     twos in row
```

Here a dot is used for zero and an asterisk is used when all cyclic shifts match. The matrix for $n = 7$ is

```
    0         1
    01234567890123456789    (!)
    01222333344444455567     v
0:0 .*................  0  0  .......   p= 1
1:1 *.222.............  1  3  ......1   p= 7
2:2 .2...2111.........  2  2  .....11   p= 7
```

```
 3:2 .2...111.2.........  3  2  ....1.1   p= 7
 4:2 .2....1121.........  4  2  ...1..1   p= 7
 5:3 ..21........112.....  5  2  ....111   p= 7
 6:3 ..111.....111.1.....  6  0  ...1.11   p= 7
 7:3 ..111.....11.11.....  7  0  ...11.1   p= 7
 8:3 ..1.2......211......  8  2  ..1..11   p= 7
 9:3 ...21.....2.11......  9  2  ..1.1.1   p= 7
10:4 ......11.2.....12... 10  2  11.1.1.   p= 7
11:4 ......112......2.1.. 11  2  11.11..   p= 7
12:4 .....11.11.....111.. 12  0  111..1.   p= 7
13:4 .....1.111.....111.. 13  0  111.1..   p= 7
14:4 .....211........12.. 14  2  1111...   p= 7
15:5 ..........1211....2. 15  2  111.11.   p= 7
16:5 ..........2.111...2. 16  2  1111.1.   p= 7
17:5 ...........1112...2. 17  2  11111..   p= 7
18:6 ..............222.* 18  3  111111.   p= 7
```

There are four necklaces that have no 'double-match' with, the non-terminal entries that have a zero in the column marked with `(!)`. These stay alone for all $n > 7$ as no possible matches enter.

Still, some interesting structures can be found. Choosing the 3-element sequences from the basic block for $n = 7$ gives a minimal-change sequence of 3-combinations out of 7:

```
..1.11  .1.11.  1.11..  .11..1  11..1.  1..1.1
.1..11  1..11.  ..11..  1..1..  1..1..  ..1..1
..1.1.  .1.1.1  1.1.1.  .1.1..  .1..1.  1.1.1.
...11.1  ..11.1  .11.1..  11.1...  1.1...1  .1...11  1...11.
....111  ...111.  ..111..  .111...  111....  11....1  1....11
```

Chapter 13

Arithmetical algorithms

13.1 Asymptotics of algorithms

An important feature of an algorithm is the number of operations that must be performed for the completion of a task of a certain size N. The quantity N should be some reasonable quantity that grows strictly with the size of the task. For high precision computations one will take the length of the numbers counted in decimal digits or bits. For computations with square matrices one may take for N the number of rows. An operation is typically a (machine word) multiplication plus an addition, one could also simply count machine instructions.

An algorithm is said to have some asymptotics $f(N)$ if it needs proportional $f(N)$ operations for a task of size N.

Examples:

- Addition of an N-digit number needs proportional N operations (here: machine word addition plus some carry operation).
- Ordinary multiplication needs $\sim N^2$ operations.
- The Fast Fourier Transform (FFT) needs $\sim N \log(N)$ operations (a straight forward implementation of the Fourier Transform, i.e. computing N sums each of length N would be $\sim N^2$).
- Matrix multiplication (by the obvious algorithm) is $\sim N^3$ (N^2 sums each of N products).

The algorithm with the 'best' asymptotics wins for some, possibly huge, N. For smaller N another algorithm will be superior. For the exact break-even point the constants omitted elsewhere are of course important.

Example: Let the algorithm `mult1` take $1.0 \cdot N^2$ operations, `mult2` take $8.0 \cdot N \log_2(N)$ operations. Then, for $N < 64$ `mult1` is faster and for $N > 64$ `mult2` is faster. Completely different algorithms may be optimal for the same task at different problem sizes.

13.2 Multiplication of large numbers

Ordinary multiplication is $\sim N^2$. Computing the product of two million-digit numbers would require $\approx 10^{12}$ operations, taking about 1 day on a machine that does 10 million operations per second (MIPS). But there are better ways ...

13.2.1 The Karatsuba algorithm

Split the numbers U and V (assumed to have approximately the same length/precision) in two pieces

$$\begin{aligned} U &= U_0 + U_1 B \\ V &= V_0 + V_1 B \end{aligned} \tag{13.1}$$

Where B is a power of the radix[1] (or base) close to the half length of U and V.

Instead of the straight forward multiplication that needs 4 multiplications with half precision for one multiplication with full precision

$$UV = U_0V_0 + B(U_0V_1 + V_0U_1) + B^2U_1V_1 \tag{13.2}$$

use the relation

$$UV = (1+B)U_0V_0 + B(U_1 - U_0)(V_0 - V_1) + (B + B^2)U_1V_1 \tag{13.3}$$

which needs 3 multiplications with half precision for one multiplication with full precision.

Apply the scheme recursively until the numbers to multiply are of machine size. The asymptotics of the algorithm is $\sim N^{\log_2(3)} \approx N^{1.585}$.

For squaring use

$$U^2 = (1+B)U_0^2 - B(U_1 - U_0)^2 + (B + B^2)U_1^2 \tag{13.4}$$

or

$$U^2 = (1-B)U_0^2 + B(U_1 + U_0)^2 + (-B + B^2)U_1^2 \tag{13.5}$$

One can extend the above idea by splitting U and V into more than two pieces each, the resulting algorithm is called Toom Cook algorithm.

Computing the product of two million-digit numbers would require $\approx (10^6)^{1.585} \approx 3200 \cdot 10^6$ operations, taking about 5 minutes on the 10 MIPS machine.

See [11], chapter 4.3.3 ('How fast can we multiply?').

13.2.2 Fast multiplication via FFT

Multiplication of two numbers is essentially a convolution of the sequences of their digits. The (linear) convolution of the two sequences $a_k, b_k, k = 0 \ldots N-1$ is defined as the sequence c where

$$c_k := \sum_{i,j=0;\ i+j=k}^{N-1} a_i b_j \qquad k = 0 \ldots 2N-2 \tag{13.6}$$

A number written in radix r as

$$a_P \quad a_{P-1} \quad \ldots \quad a_2 \quad a_1 \quad a_0 \quad . \quad a_{-1} \quad a_{-2} \quad \ldots \quad a_{-p+1} \quad a_{-p} \tag{13.7}$$

denotes a quantity of

$$\sum_{i=-p}^{P} a_i \cdot r^i = a_P \cdot r^P + a_{P-1} \cdot r^{P-1} + \cdots + a_{-p} \cdot r^{-p}. \tag{13.8}$$

[1] For decimal numbers the radix is 10.

That means, the digits can be considered as coefficients of a polynomial in r. For example, with decimal numbers one has $r = 10$ and $123.4 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1}$. The product of two numbers is almost the polynomial product

$$\sum_{k=0}^{2N-2} c_k r^k \quad := \quad \sum_{i=0}^{N-1} a_i r^i \cdot \sum_{j=0}^{N-1} b_j r^j \tag{13.9}$$

The c_k are found by comparing coefficients. One easily checks that the c_k must satisfy the convolution equation 13.6.

As the c_k can be greater than 'nine' (that is, $r - 1$), the result has to be 'fixed' using *carry* operations: Go from right to left, replace c_k by $c_k\%r$ and add $(c_k - c_k\%r)/r$ to its left neighbor.

An example: usually one would multiply the numbers 82 and 34 as follows:

	82	×	34
	3	^{3}2	8
2	^{2}4	6	
= 2	7	8	8

We just said that the carries can be delayed to the end of the computation:

	82	×	34
		32	8
	24	6	
	24	38	8
= 2	^{2} 7	^{3} 8	8

... which is really polynomial multiplication (which in turn is a convolution of the coefficients):

	$(8\,x + 2)$	×	$(3\,x + 4)$
		$32\,x$	8
	$24\,x^2$	$6\,x$	
=	$24\,x^2$	$+38\,x$	$+8$

Convolution can be done efficiently using the Fast Fourier Transform (FFT): Convolution is a simple (element wise array) multiplication in Fourier space. The FFT itself takes $N \cdot \log N$ operations. Instead of the direct convolution ($\sim N^2$) one proceeds like this:

- compute the FFTs of multiplicand and multiplicator
- multiply the transformed sequences elementwise
- compute inverse transform of the product

To understand why this actually works note that (1) the multiplication of two polynomials can be achieved by the (more complicated) scheme:

- evaluate both polynomials at sufficiently many[2] points
- pointwise multiply the values found
- find the polynomial corresponding to those (product-)values

[2] At least one more point than the degree of the product polynomial c: $\deg c = \deg a + \deg b$

and (2) that *the FFT is an algorithm for the parallel evaluation of a given polynomial at many points, namely the roots of unity.* (3) the inverse FFT is an algorithm to find (the coefficients of) a polynomial whose values are given at the roots of unity.

You might be surprised if you always thought of the FFT as an algorithm for the 'decomposition into frequencies'. There is no problem with either of these notions.

Relaunching our example we use the fourth roots of unity ± 1 and $\pm i$:

	$a = (8\,x + 2)$	$\times$	$b = (3\,x + 4)$	$c = a\,b$
$+1$	$+10$		$+7$	$+70$
$+i$	$+8i + 2$		$+3i + 4$	$+38i - 16$
-1	-6		$+1$	-6
$-i$	$-8i + 2$		$-3i + 4$	$-38i - 16$
				$c = (24\,x^2 + 38\,x + 8)$

This table has to be read like this: first the given polynomials a and b are evaluated at the points given in the left column, thereby the columns below a and b are filled. Then the values are multiplied to fill the column below c, giving the values of c at the points. Finally, the actual polynomial c is found from those values, resulting in the lower right entry. You may find it instructive to verify that a 4-point FFT really evaluates a, b by transforming the sequences 0, 0, 8, 2 and 0, 0, 3, 4 by hand. The backward transform of 70, $38i - 16$, -6, $-38i - 16$ should produce the final result given for c.

The operation count is dominated by that of the FFTs (the elementwise multiplication is of course $\sim N$), so the whole fast convolution algorithm takes $\sim N \cdot \log N$ operations. The following carry operation is also $\sim N$ and can therefore be neglected when counting operations.

Multiplying our million-digit numbers will now take only $10^6 \log_2(10^6) \approx 10^6 \cdot 20$ operations, taking approximately 2 seconds on a 10 Mips machine.

Strictly speaking $N \cdot \log N$ is not really the truth: it has to be $N \cdot \log N \cdot \log\log N$. This is because the sums in the convolutions have to be represented as exact integers. The biggest term C that can possibly occur is approximately NR^2 for a number with N digits (see next section). Therefore, working with some fixed radix R one has to do FFTs with $\log N$ bits precision, leading to an operation count of $N \cdot \log N \cdot \log N$. The slightly better $N \cdot \log N \cdot \log\log N$ is obtained by recursive use of FFT multiplies. For realistic applications (where the sums in the convolution all fit into the machine type floating point numbers) it is safe to think of FFT multiplication being proportional $N \cdot \log N$.

For a survey of multiplication methods, some mathematical background and further references see [37]. How far the idea 'polynomials for numbers' can be carried and where it fails see [38].

13.2.3 Radix/precision considerations with FFT multiplication

This section describes the dependencies between the radix of the number and the achievable precision when using FFT multiplication. In what follows it is assumed that the 'super-digits', called `LIMB`s occupy a 16 bit word in memory. Thereby the radix of the numbers can be in the range $2 \ldots 65536 (= 2^{16})$. Further restrictions are due to the fact that the components of the convolution must be representable as integer numbers with the data type used for the FFTs (here: `doubles`): The cumulative sums c_k have to be represented precisely enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a c_k will appear in the middle of the product and when multiplicand and multiplier consist of 'nines' (that is $R-1$) only. It must not jump to $c_m \pm 1$ due to numerical errors. For radix R and a precision of N `LIMB`s Let the maximal possible value be C, then

$$C \quad = \quad N\,(R-1)^2 \tag{13.10}$$

The number of bits to represent C exactly is the integer greater or equal to

$$\log_2(N\,(R-1)^2) \quad = \quad \log_2 N + 2\,\log_2(R-1) \tag{13.11}$$

Due to numerical errors there must be a few more bits for safety. If computations are made using `doubles` one typically has a mantissa of 53 bits[3] then we need to have

$$M \quad \geq \quad \log_2 N + 2\,\log_2(R-1) + S \tag{13.12}$$

where M :=mantissa-bits and S :=safety-bits. Using $\log_2(R-1) < \log_2(R)$:

$$N_{max}(R) \quad = \quad 2^{M-S-2\,\log_2(R)} \tag{13.13}$$

Suppose we have $M = 53$ mantissa-bits and require $S = 3$ safety-bits. With base 2 numbers one could use radix $R = 2^{16}$ for precisions up to a length of $N_{max} = 2^{53-3-2\cdot 16} = 256k$ `LIMBs`. Corresponding are 4096 kilo bits and = 1024 kilo hex digits. For greater lengths smaller radices have to be used according to the following table (extra horizontal line at the 16 bit limit for `LIMBs`):

Radix R	max # `LIMBs`	max # hex digits	max # bits
$2^{10} = 1024$	$1048,576\,k$	$2621,440\,k$	$10240\,M$
$2^{11} = 2048$	$262,144\,k$	$720,896\,k$	$2816\,M$
$2^{12} = 4096$	$65,536\,k$	$196,608\,k$	$768\,M$
$2^{13} = 8192$	$16384\,k$	$53,248\,k$	$208\,M$
$2^{14} = 16384$	$4096\,k$	$14,336\,k$	$56\,M$
$2^{15} = 32768$	$1024\,k$	$3840\,k$	$15\,M$
$2^{16} = 65536$	$256\,k$	$1024\,k$	$4\,M$
$2^{17} = 128\,k$	$64\,k$	$272\,k$	$1062\,k$
$2^{18} = 256\,k$	$16\,k$	$72\,k$	$281\,k$
$2^{19} = 512\,k$	$4\,k$	$19\,k$	$74\,k$
$2^{20} = 1\,M$	$1\,k$	$5\,k$	$19\,k$
$2^{21} = 2\,M$	256	1300	5120

For decimal numbers:

Radix R	max # `LIMBs`	max # digits	max # bits
10^2	$110\,G$	$220\,G$	$730\,G$
10^3	$1100\,M$	$3300\,M$	$11\,G$
10^4	$11\,M$	$44\,M$	$146\,M$
10^5	$110\,k$	$550\,k$	$1826\,k$
10^6	$1\,k$	$6,597$	$22\,k$
10^7	11	77	255

Summarizing:

- For decimal digits and precisions up to 11 million `LIMBs` use radix 10,000. (corresponding to more about 44 million decimal digits), for even greater precisions choose radix 1,000.
- For hexadecimal digits and precisions up to 256,000 `LIMBs` use radix 65,536 (corresponding to more than 1 million hexadecimal digits), for even greater precisions choose radix 4,096.

13.3 Division, square root and cube root

13.3.1 Division

The ordinary division algorithm is useless for numbers of extreme precision. Instead one replaces the division $\frac{a}{b}$ by the multiplication of a with the inverse of b. The inverse of $b = \frac{1}{b}$ is computed by finding a starting approximation $x_0 \approx \frac{1}{b}$ and then iterating

$$x_{k+1} \quad = \quad x_k + x_k(1 - b\,x_k) \tag{13.14}$$

[3]Of which only the 52 least significant bits are physically present, the most significant bit is implied to be always set.

until the desired precision is reached. The convergence is quadratic (2nd order), which means that the number of correct digits is doubled with each step: if $x_k = \frac{1}{b}(1+e)$ then

$$x_{k+1} = \frac{1}{b}(1+e) + \frac{1}{b}(1+e)\left(1 - b\frac{1}{b}(1+e)\right) \tag{13.15}$$

$$= \frac{1}{b}\left(1-e^2\right) \tag{13.16}$$

Moreover, each step needs only computations with twice the number of digits that were correct at its beginning. Still better: the multiplication $x_k(\dots)$ needs only to be done with half precision as it computes the 'correcting' digits (which alter only the less significant half of the digits). Thus, at each step we have 1.5 multiplications of the 'current' precision. The total work[4] amounts to

$$1.5 \cdot \sum_{n=0}^{N} \frac{1}{2^n}$$

which is less than 3 full precision multiplications. Together with the final multiplication a division costs as much as 4 multiplications. Another nice feature of the algorithm is that it is self-correcting. The following numerical example shows the first two steps of the computation[5] of an inverse starting from a two-digit initial approximation:

$$b := 3.1415926 \tag{13.17}$$

$$x_0 = 0.31 \quad \text{initial 2 digit approximation for } 1/b \tag{13.18}$$

$$b \cdot x_0 = 3.141 \cdot 0.3100 = 0.9737 \tag{13.19}$$

$$y_0 := 1.000 - b \cdot x_0 = 0.02629 \tag{13.20}$$

$$x_0 \cdot y_0 = 0.3100 \cdot 0.02629 = 0.0081(49) \tag{13.21}$$

$$x_1 := x_0 + x_0 \cdot y_0 = 0.3100 + 0.0081 = 0.3181 \tag{13.22}$$

$$b \cdot x_1 = 3.1415926 \cdot 0.31810000 = 0.9993406 \tag{13.23}$$

$$y_1 := 1.0000000 - b \cdot x_0 = 0.0006594 \tag{13.24}$$

$$x_1 \cdot y_1 = 0.31810000 \cdot 0.0006594 = 0.0002097(5500) \tag{13.25}$$

$$x_2 := x_1 + x_1 \cdot y_1 = 0.31810000 + 0.0002097 = 0.31830975 \tag{13.26}$$

13.3.2 Square root extraction

Computing square roots is quite similar to division: first compute $\frac{1}{\sqrt{d}}$ then a final multiply with d gives $\sqrt{d}$. Find a starting approximation $x_0 \approx \frac{1}{\sqrt{b}}$ then iterate

$$x_{k+1} = x_k + x_k\frac{(1 - d\,x_k^2)}{2} \tag{13.27}$$

until the desired precision is reached. Convergence is again 2nd order: if $x_k = \frac{1}{\sqrt{b}}(1+e)$ then

$$x_{k+1} = \frac{1}{\sqrt{b}}\left(1 - \frac{3}{2}e^2 - \frac{1}{2}e^3\right) \tag{13.28}$$

[4] The asymptotics of the multiplication is set to $\sim N$ (instead of $N\log(N)$) for the estimates made here, this gives a realistic picture for large N.

[5] using a second order iteration

Similar considerations as above (with squaring considered as expensive as multiplication[6]) give an operation count of 4 multiplications for $\frac{1}{\sqrt{d}}$ or 5 for $\sqrt{d}$.

Note that this algorithm is considerably better than the one where $x_{k+1} := \frac{1}{2}(x_k + \frac{d}{x_k})$ is used as iteration, because no long divisions are involved.

In hfloat, when the achieved precision is below a certain limit a third order correction is used to assure maximum precision at the last step:

$$x_{k+1} = x_k + x_k \frac{(1 - d\,x_k^2)}{2} + x_k \frac{3\,(1 - d\,x_k^2)^2}{8} \tag{13.29}$$

An improved version

Actually, the 'simple' version of the square root iteration can be used for practical purposes when rewritten as a *coupled iteration* for both $\sqrt{d}$ and its inverse. Using for $\sqrt{d}$ the iteration

$$x_{k+1} = x_k - \frac{(x_k^2 - d)}{2\,x_k} \tag{13.30}$$

$$= x_k - v_{k+1} \frac{(x_k^2 - d)}{2} \quad \text{where} \quad v \approx 1/x \tag{13.31}$$

and for the auxiliary $v \approx 1/\sqrt{d}$ the iteration

$$v_{k+1} = v_k + v_k\,(1 - x_k\,v_k) \tag{13.32}$$

where one starts with approximations

$$x_0 \approx \sqrt{d} \tag{13.33}$$

$$v_0 \approx 1/x_0 \tag{13.34}$$

and the v-iteration step precedes that for x. When carefully implemented this method turns out to be significantly more efficient than the preceding version. [hfloat: `src/hf/itsqrt.cc`]

TBD: *details & analysis* TBD: *last step versions for sqrt and inv*

13.3.3 Cube root extraction

Use $d^{1/3} = d\,(d^2)^{-1/3}$, i.e. compute the inverse third root of d^2 using the iteration

$$x_{k+1} = x_k + x_k \frac{(1 - d^2\,x_k^3)}{3} \tag{13.35}$$

finally multiply with d.

Convergence is 2nd order: if $x_k = \frac{1}{\sqrt[3]{b}}(1 + e)$ then

$$x_{k+1} = \frac{1}{\sqrt[3]{b}} \left(1 - 2e^2 - \frac{4}{3}e^3 - \frac{1}{3}e^4\right) \tag{13.36}$$

13.4 Square root extraction for rationals

For rational $x = \frac{p}{q}$ the well known iteration for the square root is

$$\Phi_2(x) = \frac{x^2 + d}{2\,x} = \frac{p^2 + d\,q^2}{2\,p\,q} \tag{13.37}$$

[6] Indeed it costs about $\frac{2}{3}$ of a multiplication.

A general formula for an k-th order ($k \geq 2$) iteration toward $\sqrt{d}$ is

$$\Phi_k(x) = \sqrt{d}\,\frac{\left(x+\sqrt{d}\right)^k + \left(x-\sqrt{d}\right)^k}{\left(x+\sqrt{d}\right)^k - \left(x-\sqrt{d}\right)^k} = \sqrt{d}\,\frac{\left(p+q\sqrt{d}\right)^k + \left(p-q\sqrt{d}\right)^k}{\left(p+q\sqrt{d}\right)^k - \left(p-q\sqrt{d}\right)^k} \tag{13.38}$$

Obviously, we have:

$$\Phi_m(\Phi_n(x)) = \Phi_{mn}(x) \tag{13.39}$$

All $\sqrt{d}$ vanish when expanded, e.g. the third and fifth order versions are

$$\Phi_3(x) = x\,\frac{x^2+3d}{3x^2+d} = \frac{p}{q}\,\frac{p^2+3d\,q^2}{3p^2+d\,q^2} \tag{13.40}$$

$$\Phi_5(x) = x\,\frac{x^4+10dx^2+5d^2}{5x^4+10dx^2+d^2} \tag{13.41}$$

There is a nice expression for the error behavior of the k-th order iteration:

$$\Phi_k(\sqrt{d}\cdot\frac{1+e}{1-e}) = \sqrt{d}\cdot\frac{1+e^k}{1-e^k} \tag{13.42}$$

An equivalent form of 13.38 comes from the theory of continued fractions:

$$\Phi_k(x) = \sqrt{d}\,\cot\left(k\,\operatorname{arccot}\frac{x}{\sqrt{d}}\right) \tag{13.43}$$

The iterations can also be obtained using Padé-approximants. Let $P_{[i,j]}(z)$ be the Padé-expansion of $\sqrt{z}$ around $z=1$ of order $[i,j]$. An iteration of order $i+j+1$ is given by $x\,P_{[i,j]}(\frac{d}{x^2})$. For $i=j$ one gets the iterations of odd orders, for $i=j+1$ the even orders are obtained. Different combinations of i and j result in alternative iterations:

$$[i,j] \mapsto x\,P_{[i,j]}(\frac{d}{x^2}) \tag{13.44a}$$

$$[1,0] \mapsto \frac{x^2+d}{2x} \tag{13.44b}$$

$$[0,1] \mapsto \frac{2x^3}{3x^2-d} \tag{13.44c}$$

$$[1,1] \mapsto x\,\frac{x^2+3d}{3x^2+d} \tag{13.44d}$$

$$[2,0] \mapsto \frac{3x^4+6dx^2-3d^2}{8x^3} \tag{13.44e}$$

$$[0,2] \mapsto \frac{8x^5}{15x^4-10dx^2+3d^2} \tag{13.44f}$$

Still other forms are obtained by using $\frac{d}{x}\,P_{[i,j]}(\frac{x^2}{d})$:

$$[i,j] \quad \mapsto \quad \frac{d}{x}\,P_{[i,j]}(\frac{x^2}{d}) \tag{13.45a}$$

$$[1,0] \quad \mapsto \quad \frac{x^2+d}{2x} \tag{13.45b}$$

$$[0,1] \quad \mapsto \quad \frac{2d^2}{3dx-x^3} \tag{13.45c}$$

$$[1,1] \quad \mapsto \quad \frac{d\,(d+3x^3)}{x\,(3d+x^2)} \tag{13.45d}$$

$$[2,0] \quad \mapsto \quad \frac{-x^4+6dx^2+3d^2}{8xd} \tag{13.45e}$$

$$[0,2] \quad \mapsto \quad \frac{8d^3}{3x^4-10dx^2+15d^2} \tag{13.45f}$$

Using the expansion of $1/\sqrt{x}$ and $x\,P_{[i,j]}(x^2d)$ we get:

$$[i,j] \quad \mapsto \quad x\,P_{[i,j]}(x^2d) \tag{13.46a}$$

$$[1,0] \quad \mapsto \quad \frac{x\,(3-d\,x^2)}{2} \tag{13.46b}$$

$$[0,1] \quad \mapsto \quad \frac{2x}{dx^2-1} \tag{13.46c}$$

$$[1,1] \quad \mapsto \quad x\,\frac{dx^2+3}{3dx^2+1} \tag{13.46d}$$

$$[2,0] \quad \mapsto \quad \frac{x\,(3d^2x^4-10dx+15)}{8} \tag{13.46e}$$

$$[0,2] \quad \mapsto \quad \frac{8x}{-d^2x^4+6dx^2+3} \tag{13.46f}$$

Extraction of higher roots for rationals

The Padé idea can be adapted for higher roots: use the expansion of $\sqrt[a]{z}$ around $z=1$ then $x\,P_{[i,j]}(\frac{d}{x^a})$ produces an order $i+j+1$ iteration for $\sqrt[a]{z}$. A second order iteration is given by

$$\Phi_2(x) \quad = \quad x+\frac{d-x^a}{a\,x^{a-1}} = \frac{(a-1)\,x^a+d}{a\,x^{a-1}} = \frac{1}{a}\,\left((a-1)\,x+\frac{d}{x^{a-1}}\right) \tag{13.47}$$

A third order iteration for $\sqrt[a]{d}$ is

$$\Phi_3(x) \quad = \quad x\cdot\frac{\alpha\,x^a+\beta\,d}{\beta\,x^a+\alpha\,d} = \frac{p}{q}\cdot\frac{\alpha\,p^a+\beta\,q^a d}{\beta\,p^a+\alpha\,q^a d} \tag{13.48}$$

where $\alpha=a-1, \beta=a+1$ for a even, $\alpha=(a-1)/2, \beta=(a+1)/2$ for a odd.

With $1/\sqrt[a]{x}$ and $x\,P_{[i,j]}(x^a d)$ division-free iterations for the inverse a-th root of d are obtained, see section 13.5. If you suspect a general principle behind the Padé idea, yes there is one: read on until section 13.8.4.

13.5 A general procedure for the inverse n-th root

There is a nice general formula that allows to build iterations with arbitrary order of convergence for $d^{-1/a}$ that involve no long division.

One uses the identity

$$d^{-1/a} = x\,(1-(1-x^a\,d))^{-1/a} \tag{13.49}$$
$$= x\,(1-y)^{-1/a} \quad \text{where} \quad y := (1-x^a\,d) \tag{13.50}$$

Taylor expansion gives

$$d^{-1/a} = x\sum_{k=0}^{\infty}(1/a)^{\bar{k}}\,y^k \tag{13.51}$$

where $z^{\bar{k}} := z(z+1)(z+2)\ldots(z+k-1)$. Written out:

$$d^{-1/a} = x\left(1+\frac{y}{a}+\frac{(1+a)\,y^2}{2\,a^2}+\frac{(1+a)(1+2a)\,y^3}{6\,a^3}+\right. \tag{13.52}$$
$$\left.+\frac{(1+a)(1+2a)(1+3a)\,y^4}{24\,a^4}+\cdots+\frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\,a^n}\,y^n+\ldots\right)$$

A n-th order iteration for $d^{-1/a}$ is obtained by truncating the above series after the $(n-1)$-th term,

$$\Phi_n(a,x) := x\sum_{k=0}^{n-1}(1/a)^{\bar{k}}\,y^k \tag{13.53}$$
$$x_{k+1} = \Phi_n(a,x_k) \tag{13.54}$$

Convergence is n-th order:

$$\Phi_n(d^{-1/a}(1+e)) = d^{-1/a}(1+O(e^n)) \tag{13.55}$$

Second order is:

$$\Phi_2(a,x) := x+x\frac{(1-dx^a)}{a} \tag{13.56}$$

Convergence: if $x=\frac{1}{\sqrt[a]{b}}(1+e)$ then

$$\Phi_2(a,x) = \frac{1}{\sqrt[a]{b}}\left((1+e)\left[(1+e)^a-(a+1)\right]\right) \tag{13.57}$$
$$= \frac{1}{\sqrt[a]{b}}\left(1-\frac{a+1}{2}e^2-O(e^3)\right) \tag{13.58}$$

Example 1: $a=1$ **(computation of the inverse of** d**):**

$$\frac{1}{d} = x\,\frac{1}{1-y} \tag{13.59}$$
$$\Phi(1,x) = x\,\left(1+y+y^2+y^3+y^4+\ldots\right) \tag{13.60}$$

$\Phi_2(1,x)=x\,(1+y)$ is the iteration 13.14 on page 283.

Convergence:

$$\Phi_k(1,\frac{1}{d}(1+e)) = \frac{1}{d}\left(1-(-e)^k\right) \tag{13.61}$$

Composition:

$$\Phi_{n\,m} = \Phi_n(\Phi_m) \tag{13.62}$$

There are simple closed forms for this iteration

$$\Phi_k = \frac{1-y^k}{d} = x\,\frac{1-y^k}{1-y} \tag{13.63}$$

$$\Phi_k = x\,(1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots \tag{13.64}$$

Example 2: $a=2$ **(computation of the inverse square root of** d**):**

$$\frac{1}{\sqrt{d}} = x\,\frac{1}{\sqrt{1-y}} \tag{13.65}$$

$$= x\left(1+\frac{y}{2}+\frac{3\,y^2}{8}+\frac{5\,y^3}{16}+\frac{35\,y^4}{128}+\cdots+\frac{\binom{2k}{k}\,y^k}{4^k}+\ldots\right) \tag{13.66}$$

$\Phi_2(2,x) = x\,(1+y/2)$ is the iteration 13.27 on page 284.

An expression for the error behavior of the n-th order iteration similar to formula 13.42 is

$$\Phi_n(d^{-1/a}\frac{1+e}{1-e}) = d^{-1/a}\,\frac{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k - \sum_{k=n+1}^{2n+1}\binom{2n-1}{k}(-e)^k}{(1-e)^{2n-1}} \tag{13.67}$$

$$= \frac{1+c}{1-c} \quad \text{where} \quad c = e^n\,\frac{\sum_{k=0}^{n}\binom{2n-1}{n-k}(-e)^k}{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k} \tag{13.68}$$

e.g. for $k=4$ the fraction on the right hand side is

$$\Phi_4(d^{-1/a}\frac{1+e}{1-e})/d^{-1/a} =: F = \frac{1-7\,e+21\,e^2-35\,e^3-35\,e^4+21\,e^5-7\,e^6+e^7}{1-7\,e+21\,e^2-35\,e^3+35\,e^4-21\,e^5+7\,e^6-e^7} \tag{13.69}$$

$$c = e^4\,\frac{e^3-7\,e^2+21\,e-35}{1-7\,e+21\,e^2-35\,e^3} \tag{13.70}$$

$$F = 1-70e^4-448e^5-1680e^6-4800e^7-\ldots \tag{13.71}$$

The coefficients of the Taylor expansion are always integers. There is always a partial fraction decomposition like

$$F = 1-\frac{70}{(e-1)^4}-\frac{168}{(e-1)^5}-\frac{140}{(e-1)^6}-\frac{40}{(e-1)^7} \tag{13.72}$$

Composition is not as trivial as for the inverse, e.g.:

$$\Phi_4-\Phi_2(\Phi_2) = -\frac{1}{16}\,x\,(y)^4 \tag{13.73}$$

In general, one has

$$\Phi_{n\,m}-\Phi_n(\Phi_m) = x\,P(y)\,y^{n\,m} \tag{13.74}$$

where P is a polynomial in $y=1-d\,x^2$. Also, in general $\Phi_n(\Phi_m)\neq\Phi_m(\Phi_n)$ for $n\neq m$, e.g.:

$$\Phi_3(\Phi_2)-\Phi_2(\Phi_3) = \frac{15}{1024}x\,(x^2d)\,y^6 = \frac{15}{1024}x\,(1-y)\,y^6 \tag{13.75}$$

Product forms for compositions of the second-order iteration for $1/\sqrt{d}$:

$$\Phi_2(x) = x\left(1+\frac{1}{2}\,y\right) \quad \text{where} \quad y=1-d\,x^2 \tag{13.76}$$

$$\Phi_2(\Phi_2(x)) = x\left(1+\frac{1}{2}\,y\right)\left(1+\frac{1}{8}\,y^2\,(3+y)\right) \tag{13.77}$$

$$= \Phi_2(x)\left(1+\frac{1}{8}\,y^2\,(3+y)\right) \tag{13.78}$$

$$\Phi_2(\Phi_2(\Phi_2(x))) = \Phi_2(\Phi_2(x))\left(1+\frac{1}{512}\,y^4\,(3+y)^2\;\left(12+y^2\,(3+y)\right)\right) \tag{13.79}$$

13.6 Re-orthogonalization of matrices

A task from graphics applications: a rotation matrix A that deviates from being orthogonal[7] shall be transformed to the closest orthogonal matrix E. It is well known that (see e.g. [59])

$$E = A\,(A^T A)^{-\frac{1}{2}} \tag{13.80}$$

With the division-free iteration for the inverse square root

$$\Phi(x) = x\left(1+\frac{1}{2}\,(1-dx^2)+\frac{3}{8}\,(1-dx^2)^2+\frac{5}{16}\,(1-dx^2)^3+\ldots\right) \tag{13.81}$$

at hand the given task is pretty easy: As $A^T A$ is close to unity (the identity matrix) we can use the (second order) iteration with $d = A^T A$ and $x = 1$

$$(A^T A)^{-\frac{1}{2}} \approx \left(1+\frac{1-A^T A}{2}\right) \tag{13.82}$$

and multiply by A to get a 'closer-to-orthogonal' matrix A_+:

$$A_+ = A\left(1+\frac{1-A^T A}{2}\right) \approx E \tag{13.83}$$

The step can be repeated with A_+ (or higher orders can be used) if necessary. Note the identical equation would be obtained when trying to compute the inverse square root of 1:

$$x_+ = x\left(1+\frac{1-x^2}{2}\right) \to 1 \tag{13.84}$$

It is instructive to write things down in the SVD[8]-representation

$$A = U\,\Omega\,V^T \tag{13.85}$$

where U and V are orthogonal and Ω is a diagonal matrix with non-negative entries (cf. [61]). The SVD is the unique decomposition of the action of the matrix as: rotation – element wise stretching – rotation.

Now

$$A^T A = \left(V\Omega U^T\right)\,\left(U\Omega V^T\right) = V\Omega^2 V^T \tag{13.86}$$

and positive exponents of A are 'absorbed' as powers of Ω

$$\left(U\Omega V^T\right)^n = U\Omega^n V^T \tag{13.87}$$

while for negative exponents

$$\left(U\Omega V^T\right)^{-n} = V\Omega^{-n} U^T \tag{13.88}$$

Thereby

$$(A^T A)^{-\frac{1}{2}} = \left(\left(V\Omega U^T\right)\left(U\Omega V^T\right)\right)^{-\frac{1}{2}} = \left(V\Omega^2 V^T\right)^{-\frac{1}{2}} = V\Omega^{-1}V^T \tag{13.89}$$

and we have

$$A\,(A^T A)^{-\frac{1}{2}} = \left(U\Omega V^T\right)\,\left(V\Omega^{-1}V^T\right) = U\,V^T \tag{13.90}$$

[7] typically due to cumulative errors from multiplications with many incremental rotations

[8] singular value decomposition

that is, the 'stretching part' was removed.

Observe that

$$E \;=\; A \cdot \left(1 + \frac{1 - A^T A}{2}\right) \cdot \left(1 + \frac{1 - A_+^T A_+}{2}\right) \cdot \ldots =: A\,P \tag{13.91}$$

i.e. P is the accumulated product of the expressions in parenthesis of the right hand side of the iteration 13.83. One has $P = V\Omega^{-1}V^T$ and $A = P^{-1}E$. This resembles The so called *polar decomposition*

$$A \;=\; WH \quad = \left(A(A^T A)^{-1/2}\right)\,\left((A^T A)^{1/2}\right) \tag{13.92}$$

where H is the (unique, positive semidefinite) square root $H = (A^T A)^{1/2}$ and $W = A(A^T A)^{-1/2}$, W is orthogonal and $H = H^T$ (cf. [62]). One has $W = E$ and $H = W^{-1}A = W^T A$. Compute the polar decomposition as

$$P_0 \;=\; 1 \qquad E_0 = A \tag{13.93a}$$

$$P_{k+1} \;=\; P_k \left(1 + \frac{1 - E_k^T\,E_k}{2}\right) \quad \to P = H \tag{13.93b}$$

$$E_{k+1} \;=\; E_k\,P_{k+1} \quad \to E = W \tag{13.93c}$$

higher orders can be added in the computation of P_{k+1}. The polar decomposition can be seen as an analogue to $z = r\,e^{i\,\phi}$ for $z \in \mathbb{C}$, identify $H \sim r$, $W \sim e^{i\,\phi}$.

Similarly, [62] defines a *sign decomposition*

$$A \;=\; S\,N \quad = \left(A(A^2)^{-1/2}\right)\,\left((A^2)^{1/2}\right) \tag{13.94}$$

where $N = (A^2)^{-1/2}$ and $S = A\,(A^2)^{-1/2} = A\,N$ (A, S and N commute pairwise). The square root has to be chosen such that all its eigenvalues all have positive real parts. The sign decomposition is undefined if A has eigenvalues on the imaginary axis. S is its own inverse (its eigenvalues are ± 1). Use

$$N_0 \;=\; 1 \qquad S_0 = A \tag{13.95a}$$

$$N_{k+1} \;=\; N_k \left(1 + \frac{1 - S_k^2}{2}\right) \quad \to N \tag{13.95b}$$

$$S_{k+1} \;=\; S_k\,N_{k+1} \quad \to S \tag{13.95c}$$

While we are at it: Define a matrix A^+ as

$$A^+ \;:=\; (AA^T)^{-1}A^T = \left(V\Omega^{-2}V^T\right)\,\left(V\Omega U^T\right) = V\Omega^{-1}U^T \tag{13.96}$$

This looks suspiciously like the inverse of A. In fact, this is the *pseudo-inverse* of A:

$$A^+\,A \;=\; \left(V\,\Omega^{-1}\,U^T\right)\,\left(U\,\Omega\,V^T\right) = 1 \quad \text{but wait} \tag{13.97}$$

A^+ has the nice property to exist even if A^{-1} does not. If A^{-1} exists, it is identical to A^+. If not, $A^+\,A \neq 1$ but A^+ will give the best possible (in a least-square sense) solution $x^+ = A^+ b$ of the equation $A\,x = b$ (see [21], p.770ff). To find $(AA^T)^{-1}$ use the iteration for the inverse:

$$\Phi(x) = x\,\left(1 + (1 - dx) + (1 - dx)^2 + \ldots\right) \tag{13.98}$$

with $d = A\,A^T$ and the start value $x_0 = 2 - n\,(A\,A^T)/\,||A\,A^T||^2$ where n is the dimension of A.

13.7 n-th root by Goldschmidt's algorithm

TBD: *show derivation (as root of 1)* TBD: *give numerical example* TBD: *parallel feature*

The so-called Goldschmidt algorithm to approximate the a-th root of d can be stated as follows:

set

$$x_0 := d \qquad E_0 := d^{a-1} \tag{13.99a}$$

then iterate:

$$r_k := 1 + \frac{1 - E_k}{a} \quad \to 1 \tag{13.99b}$$

$$x_{k+1} := x_k \cdot r_k \tag{13.99c}$$

$$E_{k+1} := E_k \cdot r_k^a \qquad \to 1 \tag{13.99d}$$

until x close enough to

$$x_\infty = d^{\frac{1}{a}}. \tag{13.100}$$

The invariant quantity is $\frac{(x_k \cdot r)^a}{(E_k \cdot r^a)}$. Clearly

$$\frac{x_{k+1}^a}{E_{k+1}} = \frac{(x_k \cdot r)^a}{(E_k \cdot r^a)} = \frac{x_k^a}{E_k} \tag{13.101}$$

With $\frac{x_0^a}{E_0} = \frac{d^a}{d^{a-1}} = d$ and $E_\infty = 1$, therefore $x_\infty^a = d$. Convergence is quadratic.

A variant for inverse roots is as follows:

set

$$x_0 := 1 \qquad E_0 := d \tag{13.102}$$

then iterate as in formulas 13.99b..13.99d

For $a = 1$ we get:

$$\frac{1}{d} = \prod_{k=0}^{\infty} (2 - E_k) \tag{13.103}$$

$$\tag{13.104}$$

where $E_{k+1} := E_k\,(2 - E_k)$.

For $a = 2$ we get a iteration for the inverse square root:

$$\frac{1}{\sqrt{d}} = \prod_{k=0}^{\infty} \frac{3 - E_k}{2} \tag{13.105}$$

$$\tag{13.106}$$

where $E_{k+1} := E_k\,(\frac{3-E_k}{2})^2$. Cf. [55].

Higher order iterations are obtained by appending higher terms to the expression $\left(1 + \frac{1-E_k}{a}\right)$ in the definitions of r_{k+1} as suggested by equation 13.52 (and the identification $y = 1 - E$):

$$\begin{aligned}
\Bigg(1 + \frac{1 - E_k}{a} &+ && \\
\text{[third order:]} &+ \frac{(1+a)\,(1-E_k)^2}{2\,a^2} \\
\text{[fourth order:]} &+ \frac{(1+a)\,(2+a)\,(1-E_k)^3}{6\,a^3} \\
&+ \quad \ldots \quad + \\
\text{[}(n+1)\text{-th order:]} &+ \frac{(1+a)\,(1+2w)\ldots(1+n\,a)\,(1-E_k)^n}{n!\,a^n}\Bigg)
\end{aligned} \tag{13.107}$$

For those fond of products: for $d > 0, d \neq 1$

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty} \left(1 + \frac{1}{q_k}\right) \quad \text{where} \quad q_0 = \frac{d+1}{d-1}, \quad q_{k+1} = 2q_k^2 - 1 \tag{13.108}$$

(convergence is quadratic) and

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty} \left(1 + \frac{2}{h_k}\right) \quad \text{where} \quad h_0 = \frac{d+3}{d-1}, \quad h_{k+1} = (h_k + 2)^2 (h_k - 1) + 1 \tag{13.109}$$

(convergence is cubic). These are given in [56], the first is ascribed to Friedrich Engel. The paper gives $h_{k+1} = \frac{4d}{d-1} \prod_{i=0}^{k} h_i^2 - 3$. Note that for 13.108

$$q_k \quad = \quad T_{2^k}(q_0) \tag{13.110}$$

$$\frac{1}{q_k} \quad = \quad \frac{(d-1)^N}{\sum_{i=0}^{N} \binom{2N}{2i} d^i} \quad \text{where} \quad N = 2^k \tag{13.111}$$

where T_n is the n-th Chebychev polynomial of the first kind. One finds

$$q_k \quad = \quad T_{2^k}(1/c) \quad \text{where} \quad d = \frac{1-c}{1+c}, \quad c < 1 \tag{13.112}$$

and

$$\sqrt{\frac{1-c}{1+c}} \quad \approx \quad \frac{1-c}{c} \frac{U_{2^k-1}(1/c)}{T_{2^k}(1/c)} \tag{13.113}$$

which can be expressed in $d = \frac{1-c}{1+c}$ as

$$\sqrt{d} \quad \approx \quad \frac{2\,d}{1-d} \frac{U_{2^k-1}(\frac{1+d}{1-d})}{T_{2^k}(\frac{1+d}{1-d})} \quad \text{where} \quad d > 1 \tag{13.114}$$

where U_n is the n-th Chebychev polynomial of the second kind. Note that $U_{2^k-1}(x) = 2^k \prod_{i=0}^{k-1} T_{2^i}(x)$. A computation might successively compute $T_{2^i} = 2\,T_{2^{i-1}}^2 - 1$, accumulate the product $U_{2^i-1} = 2\,U_{2^{i-1}-1}\,T_{2^{i-1}}$ until U_{2^k-1} and T_{2^k} are obtained. Alternatively one might use $U_k(x) = \frac{1}{k+1}\,\partial_z T_{k+1}(x)$ and the recursion for the coefficients of T (cf. section 13.14).

Similarly, approximations for $\sqrt{a^2+1}$ are given by $R_k(a) = \frac{T_k(i\,a)/i^k}{U_{k-1}(i\,a)/i^{k-1}} = -i\,\frac{T_k(i\,a)}{U_{k-1}(i\,a)}$ where $i = \sqrt{-1}$. The composition law $R_{m\,n}(x) = R_m(R_n(x))$ holds in analogy to $T_{m\,n}(x) = T_m(T_n(x))$

k	$R_k(1)$	$R_k(a)$
1	$\frac{1}{1}$	$\frac{a}{1}$
2	$\frac{3}{2}$	$\frac{1+2\,a^2}{2\,a}$
3	$\frac{7}{5}$	$\frac{3\,a+4\,a^3}{1+4\,a^2}$
4	$\frac{17}{12}$	$\frac{1+8\,a^2+8\,a^4}{4\,a+8\,a^3}$
5	$\frac{41}{29}$	$\frac{5\,a+20\,a^3+16\,a^5}{1+12\,a^2+16\,a^4}$

(13.115)

13.8 Iterations for the inversion of a function

In this section we will look at general forms of *iterations* for zeros[9] $x = r$ of a function $f(x)$. Iterations are themselves functions $\Phi(x)$ that, when 'used' as

$$x_{k+1} \quad = \quad \Phi(x_k) \tag{13.116}$$

will make x converge towards $x_\infty = r$ if x_0 was chosen not too far away from r.

The functions $\Phi(x)$ must be constructed so that they have an attracting fixed point where $f(x)$ has a zero: $\Phi(r) = r$ (fixed point) and $|\Phi'(r)| < 1$ (attracting).

The order of convergence (or simply *order*) of a given iteration can be defined as follows: let $x = r \cdot (1+e)$ with $|e| \ll 1$ and $\Phi(x) = r \cdot (1+\alpha e^n + O(e^{n+1})$, then the iteration Φ is called *linear* (or first order) if $n = 1$ (and $|\alpha| < 1$) and super-linear if $n > 1$. Iterations of second order ($n = 2$) are often called *quadratically-*, those of third order *cubically* convergent. A linear iteration improves the result by (roughly) adding a constant amount of correct digits with every step, a super-linear iteration of order n will multiply the number of correct digits by n.

For $n \geq 2$ the function Φ has a super-attracting fixed point at r: $\Phi'(r) = 0$. Moreover, an iteration of order $n \geq 2$ has

$$\Phi'(r) = 0, \quad \Phi''(r) = 0, \quad \ldots, \quad \Phi^{(n-1)}(r) = 0 \tag{13.117}$$

There seems to be no standard term for this in terms of fixed points, attracting of order n might be appropriate.

To any iteration of order n for a function f one can add a term $f(x_k)^{n+1} \cdot \varphi(x)$ (where φ is an arbitrary function that is analytic in a neighborhood of the root) without changing the order of convergence. It is assumed to be zero in what follows.

Any two iterations of (the same) order n differ in a term $(x-r)^n \, \nu(x)$ where $\nu(x)$ is a function that is finite at r (cf. [10], p. 174, ex.3).

Two general expressions, Householder's formula and Schröder's formula, can be found in the literature. Both allow the construction of iterations for a given function $f(x)$ that converge at arbitrary order. A simple construction that contains both of them as special cases is given.

TBD: *p-adic iterations*

13.8.1 Householder's formula

Let $n \geq 2$, then

$$\Phi_n(x_k) \quad := \quad x_k + (n-1) \, \frac{\left(\frac{g(x_k)}{f(x_k)}\right)^{(n-2)}}{\left(\frac{g(x_k)}{f(x_k)}\right)^{(n-1)}} + f(x_k)^{n+1} \, \varphi(x) \tag{13.118}$$

gives a $n-$th order iteration for a (simple) root r of f. $g(x)$ must be a function that is analytic near the root and is set to 1 in what follows (cf. [10] p.169).

For $n = 2$ we get Newton's formula:

$$\Phi_2(x) \quad = \quad x - \frac{f}{f'} \tag{13.119}$$

For $n = 3$ we get Halley's formula:

$$\Phi_3(x) \quad = \quad x - \frac{2ff'}{2f'^2 - ff''} \tag{13.120}$$

[9] or roots of the function: r so that $f(r) = 0$

$n = 4$ and $n = 5$ result in:

$$\Phi_4(x) = x - \frac{3f(ff'' - 2f'^2)}{6ff'f'' - 6f'^3 - ff'''} \tag{13.121}$$

$$\Phi_5(x) = x + \frac{4f\left(6f'^3 - 6ff'f'' + f^2f'''\right)}{\left(f^3f'''' - 24f'^4 + 36ff'^2f'' - 8f^2f'f''' - 6f^2f''^2\right)} \tag{13.122}$$

Second order 13.118 with $f(x) := \frac{1}{x^a} - d$ gives formula 13.56, but for higher orders one gets iterations that require long divisions.

Kalantari and Gerlach [57] give the iteration

$$B_m(x) = x - f(x)\frac{D_{m-2}(x)}{D_{m-1}(x)} \tag{13.123}$$

where $m \geq 2$ and

$$D_m(x) = \det\begin{pmatrix} f'(x) & \frac{f''(x)}{2!} & \dots & \frac{f^{(m-1)}(x)}{(m-1)!} & \frac{f^{(m)}(x)}{m!} \\ f(x) & f'(x) & \ddots & \ddots & \frac{f^{(m-1)}(x)}{(m-1)!} \\ 0 & f(x) & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \frac{f''(x)}{2!} \\ 0 & 0 & \ddots & f(x) & f'(x) \end{pmatrix} \tag{13.124}$$

(and $D_0 = 1$). The iteration turns out to be identical to the one of Householder. A recursive definition for $D_m(x)$ is given by

$$D_m(x) = \sum_{i=1}^{m} (-1)^{i-1} f(x)^{i-1} \frac{f^{(i)}(x)}{i!} D_{m-i}(x) \tag{13.125}$$

Similar, the well-known derivation of Halley's formula by applying Newton's formula to $f/\sqrt{f'}$ can be generalized to produce m-order iterations as follows: Let $F_1(x) = f(x)$ and for $m \geq 2$ let

$$F_m(x) = \frac{F_{m-1}(x)}{F'_{m-1}(x)^{1/m}} \tag{13.126}$$

$$G_m(x) = x - \frac{F_{m-1}(x)}{F'_{m-1}(x)} \tag{13.127}$$

Then $G_m(x) = D_m(x)$ as shown in [57].

13.8.2 Schröder's formula

Let $n \geq 2$, and φ be an arbitrary (analytic near the root) function that is set to zero in what follows, then the expression

$$\Phi_n(x_k) := \sum_{t=0}^{n} (-1)^t \frac{f(x_k)^t}{t!} \left(\frac{1}{f'(x_k)}\partial\right)^{t-1} \frac{1}{f'(x_k)} + f(x_k)^{n+1}\varphi(x) \tag{13.128}$$

gives a $n-$th order iteration for a (simple) root r of f (cf. [9] p.13). This is, explicitly,

$$\begin{aligned} \Phi_n = x &- \frac{f}{1!\,f'} - \frac{f^2}{2!\,f'^3}\cdot f'' - \frac{f^3}{3!\,f'^5}\cdot\left(3f''^2 - f'f'''\right) \\ &- \frac{f^4}{4!\,f'^7}\cdot\left(15f''^3 - 10f'f''f''' + f'^2f''''\right) \\ &- \frac{f^5}{5!\,f'^9}\cdot\left(105f''^4 - 105f'f''^2f''' + 10f'^2f'''^2 + 15f'^2f''f'''' - f'^3f'''''\right) - \dots \end{aligned} \tag{13.129}$$

The second order iteration is the same as the corresponding iteration from 13.118 while all higher order iterations are different. The third order iteration obtained upon truncation after the third term on the right hand side, written as

$$\Phi_3 \quad = \quad x - \frac{f}{f'}\left(1 - \frac{ff''}{2f'^2}\right) \tag{13.130}$$

is sometimes referred to as 'Householder's method'.

Cite from [9], (p.16, translation has a typo in the first formula):

> If we denote the general term by
>
> $$-\frac{f^a}{a!}\frac{\chi_a}{f'^{2a-1}} \tag{13.131}$$
>
> the numbers χ_a can be easily computed by the recurrence
>
> $$\chi_{a+1} \quad = \quad (2a-1)f''\chi_a - f'\partial\chi_a \tag{13.132}$$
>
> .

Formula 13.128 with $f(x) := 1/x^a - d$ gives the 'division-free' iteration 13.53 for arbitrary order.

For $f(x) := \log(x) - d$ one gets the iteration 13.9.3.

For $f(x) := x^2 - d$ one gets

$$\Phi(x) \quad = \quad x - \left(\frac{x^2-d}{2x} + \frac{\left(x^2-d\right)^2}{8x^3} + \frac{\left(x^2-d\right)^3}{16x^5} + \frac{5\left(x^2-d\right)^4}{128x^7} + \dots\right) \tag{13.133}$$

$$= \quad x - \left(y + \frac{1}{2x}\cdot y^2 + \frac{2}{(2x)^2}\cdot y^3 + \frac{5}{(2x)^3}\cdot y^4 + \dots\right) \quad \text{where} \quad y := \frac{x^2-d}{2x} \tag{13.134}$$

$$= \quad x - 2x\cdot\left(Y + Y^2 + 2\,Y^3 + 5\,Y^4 + 14\,Y^5 + 42\,Y^6 + \dots\right) \quad \text{where} \quad Y := \frac{x^2-d}{(2x)^2} \tag{13.135}$$

The connection between Householder's and Schröder's iterations is that the Taylor series of the k-th order Householder iteration around $f = 0$ up to order $k-1$ gives the k-th order Schröder iteration.

13.8.3 Dealing with multiple roots

The iterations given so far will not converge at the stated order if f has a multiple root at r. As an example consider the (for simple roots second order) iteration $\Phi(x) = x - f/f'$ for $f(x) = (x^2-d)^p, p \in \mathbb{N}, p \geq 2$: $\Phi_?(x) = x - \frac{x^2-d}{p2x}$. Its convergence is only linear: $\Phi(\sqrt{d}(1+e)) = \sqrt{d}(1 + \frac{p-1}{p}e + O(e^2))$

Householder ([10] p.161 ex.6) gives

$$\Phi_2(x) = x - p\cdot\frac{f}{f'} \tag{13.136}$$

as a second order iteration for functions f known *a priori* to have roots of multiplicity p.

A general approach is to use the general[10] expressions with $F := f/f'$ instead of f. Both F and f have the same set of roots, but the multiple roots of f are simple roots of F. To illustrate this let f have a root of multiplicity p at r: $f(x) = (x-r)^p\,h(x)$ with $h(r) \neq 0$. Then

$$f'(x) \quad = \quad p\,(x-r)^{p-1}\,h(x) + (x-r)^p\,h'(x) \tag{13.137}$$

$$= \quad (x-r)^{p-1}\left(p\,h(x) + (x-r)\,h'(x)\right) \tag{13.138}$$

[10] This word intentionally used twice.

and

$$F(x) = f(x)/f'(x) \quad = \quad (x-r)\,\frac{h(x)}{p\,h(x)+(x-r)\,h'(x)} \tag{13.139}$$

The fraction on the right hand side does not vanish at the root r.

With Householder's formula (13.118) we get (iterations for F denoted by $\Phi_k^{\%}$):

$$\Phi_2(x) \quad = \quad x - \frac{f}{f'} \tag{13.140a}$$

$$\Phi_2^{\%}(x) \quad = \quad x - \frac{ff'}{f'^2 - ff''} \tag{13.140b}$$

$$\Phi_3(x) \quad = \quad x - \frac{2ff'}{2f'^2 - ff''} \tag{13.140c}$$

$$\Phi_3^{\%}(x) \quad = \quad x + \frac{2f^2f'' - 2ff'^2}{2f'^3 - 3ff'f'' + f^2f'''} \tag{13.140d}$$

$$\Phi_4(x) \quad = \quad x + \frac{3f^2f'' - 6ff'^2}{6f'^3 - 6ff'f'' + f^2f'''} \tag{13.140e}$$

$$\Phi_4^{\%}(x) \quad = \quad x + \frac{6ff'^3 + 3f^3f''' - 9f^2f'f''}{f^3f'''' - 6f'^4 + 12ff'^2f'' - 4f^2f'f''' - 3f^2f''^2} \tag{13.140f}$$

$$\Phi_5(x) \quad = \quad x + \frac{24ff'^3 + 4f^3f''' - 24f^2f'f''}{f^3f'''' - 24f'^4 + 36ff'^2f'' - 8f^2f'f''' - 6f^2f''^2} \tag{13.140g}$$

The terms in the numerators and denominators of $\Phi_k^{\%}$ and Φ_{k+1} are identical up to the integral constants.

Schröder's formula (13.128), when inserting f/f', becomes:

$$\begin{aligned}\Phi^{\%}(x) \quad = \quad & x + \frac{ff'}{(ff''-f'^2)} - \frac{f^2f'\left(ff'f''' - 2ff''^2 + f'^2f''\right)}{2\,(ff''-f'^2)^3} - \\ & - \frac{f^3f'\left(2ff'^3f''f''' \pm \ldots - 3f^2f'^2f'''^2\right)}{6\,(ff''-f'^2)^5} - \frac{f^4f'\left(3f'^8f'''' \pm \ldots - 36f^3f'^2f''^2f'''^2\right)}{24\,(ff''-f'^2)^7} - \\ & - \ldots - \frac{f^kf'(\ldots)}{k!\,(ff''-f'^2)^{2k-1}}\end{aligned} \tag{13.141}$$

Checking convergence with the above example: the iteration is:

$$\Phi_2^{\%}(x) \quad = \quad x + x\frac{x^2-d}{x^2+d} = \frac{d\,x}{x^2+d} \tag{13.142}$$

Convergence is second order (independent of p): $\Phi_2^{\%}(\sqrt{d}(1+\epsilon)) = \sqrt{d}(1-\epsilon^2/2+O(\epsilon^3))$.

Similar to formula 13.42 we have for the error

$$\Phi_k^{\%}(\sqrt{d}\cdot\frac{1-e}{1+e}) \quad = \quad \sqrt{d}\cdot\frac{1-e^k}{1+e^k} \tag{13.143}$$

if Householder's iteration is used (one gets d divided by the k-th iteration obtained from f).

Using the Schröder's 3rd order formula for f/f' with f as above we get a nice 4th (!) order iteration for $\sqrt{d}$:

$$\Phi_3^{\%}(x) \quad = \quad x + x\,\frac{d-x^2}{d+x^2} + x\,d\,\frac{(d-x^2)^2}{(d+x^2)^3} \tag{13.144}$$

$$\Phi_3^{\%}(\sqrt{d}\,\frac{1-e}{1+e}) \quad = \quad \sqrt{d}\,\frac{1+3e^2-3e^4-e^6}{1+3e^2+3e^4+e^6} \tag{13.145}$$

$$= \quad \sqrt{d}\,\frac{1-c}{1+c} \quad \text{where} \quad c = e^4\,\frac{e^2+3}{3e^2+1} \tag{13.146}$$

In general, the $(1 + a\,k)$-th order Schröder iteration for $1/\sqrt[a]{d}$ obtained through f/f' has an order of convergence that exceeds the expected one by one.

For $f(x) = 1 - d\,x^2$ one gets (3rd order Schröder):

$$\Phi_3^{\%}(x) \quad = \quad x + x\,\frac{1 - dx^2}{1 + dx^2} + x\,\frac{(1 - dx^2)^2}{(1 + dx^2)^3} \tag{13.147}$$

which is even slightly nicer, also has 4th order convergence and the error expression $\Phi_3^{\%}(\frac{1}{\sqrt{d}}\,\frac{1-e}{1+e})$ is as in 13.145.

13.8.4 A general scheme

Starting point is the Taylor series of a function f around x_0:

$$f(x) \quad = \quad \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x_0)\,(x - x_0)^k \tag{13.148}$$

$$= \quad f(x_0) + f'(x_0)\,(x - x_0) + \frac{1}{2} f''(x_0)\,(x - x_0)^2 + \frac{1}{6} f'''(x_0)\,(x - x_0)^3 + \ldots \tag{13.149}$$

Now let $f(x_0) = y_0$ and r be the zero ($f(0) = r$). We then happily expand the inverse $g = f^{-1}$ around y_0

$$g(0) \quad = \quad \sum_{k=0}^{\infty} \frac{1}{k!} g^{(k)}(y_0)\,(0 - y_0)^k \tag{13.150}$$

$$= \quad g(y_0) + g'(y_0)\,(0 - y_0) + \frac{1}{2} g''(y_0)\,(0 - y_0)^2 + \frac{1}{6} g'''(y_0)\,(0 - y_0)^3 + \ldots \tag{13.151}$$

Using $x_0 = g(y_0)$ and $r = g(0)$ we get

$$r \quad = \quad x_0 - g'(y_0)\,f(x_0) + \frac{1}{2} g''(y_0)\,f(x_0)^2 - \frac{1}{6} g'''(y_0)\,f(x_0)^3 + \ldots \tag{13.152}$$

Remains to express the derivatives of the inverse g in terms of (derivatives of) f. Not a difficult task, note that

$$f \circ g = \mathrm{id} \qquad \text{that is:} \quad f(g(x)) = x \tag{13.153}$$

and derive (chain rule) to get $g'(f(x))\,f'(x) = 1$, so $g'(y) = \frac{1}{f'(x)}$. Derive $f(g(x)) - x$ multiple times and set the expressions to zero (arguments y of g and x of f are omitted for readability):

$$1 \quad = \quad f'g' \tag{13.154a}$$

$$0 \quad = \quad g'f'' + f'^2 g'' \tag{13.154b}$$

$$0 \quad = \quad g'f''' + 3f'f''g'' + f'^3 g''' \tag{13.154c}$$

$$0 \quad = \quad g'f'''' + 4f'g''f''' + 3f''^2 g'' + 6f'^2 f''g''' + f'^4 g'''' \tag{13.154d}$$

This system of linear equations in the derivatives of g is trivially solved because it is already triangular. We obtain:

$$g' \quad = \quad \frac{1}{f'} \tag{13.155a}$$

$$g'' \quad = \quad -\frac{f''}{f'^3} \tag{13.155b}$$

$$g''' \quad = \quad \frac{1}{f'^5}\left(3f''^2 - f'f'''\right) \tag{13.155c}$$

$$g'''' \quad = \quad \frac{1}{f'^7}\left(10f'f''f''' - 15f''^3 - f'^2 f''''\right) \tag{13.155d}$$

$$g''''' \quad = \quad \frac{1}{f'^9}\left(105f''^4 - f'^3 f''''' - 105f'f''^2 f''' + 15f'^2 f''f'''' + 10f'^2 f'''^2\right) \tag{13.155e}$$

Thereby equation 13.152 can be written as (omitting arguments x of f everywhere)

$$\begin{aligned} r &= x - \frac{1}{f'} f + \frac{1}{2}\left(-\frac{f''}{f'^3}\right) f^2 - \frac{1}{6}\left(\frac{1}{f'^5}\left(3f''^2 - f'f'''\right)\right) f^3 + \ldots \\ &= x - \frac{f}{1!\,f'} - \frac{f^2}{2!\,f'^3}\cdot f'' - \frac{f^3}{3!\,f'^5}\cdot\left(3f''^2 - f'f'''\right) - \ldots \end{aligned} \tag{13.156}$$

which is Schröder's iteration (equation 13.129).

The $[i,j]$-th Padé approximant of r in f gives an iteration of order $i+j+1$. Write $P_{[i,j]}$ for an iteration (of order $i+j+1$) that is obtained using the Padé approximant $[i,j]$.

$$P_{[0,1]}(x) = x^2\,\frac{f'}{f + x\,f'} = x - \frac{x\,f}{f + x\,f'} = x - \frac{x\,f}{(x\,f)'} = x\left(1 + \frac{f}{x\,f'}\right)^{-1} \tag{13.157}$$

The iterations obtained in general involve powers of x in the 'increment' $P - x$:

$$\begin{aligned} P_{[0,2]}(x) &= \frac{2x^3 f'^3}{2f^2f' + 2xff'^2 + xf^2f'' + 2x^2f'^3} = x - \frac{xf\left(2ff' + xff'' + 2xf'^2\right)}{\left(2f^2f' + 2xff'^2 + xf^2f'' + 2x^2f'^3\right)} \\ &= x\left(1 + \frac{f}{xf'} + \frac{f^2}{x^2f'^2} + \frac{f^2f''}{2xf'^3}\right)^{-1} = x\left(1 + \frac{f}{(x\,f')} + \frac{f^2(x^2\,f'')}{2(x\,f')^3} + \frac{f^2}{(x\,f')^2}\right)^{-1} \end{aligned} \tag{13.158}$$

The $[i,j]$-th Padé approximant of $r - x$ in f gives an iteration of order $i+j+2$ where the increment consists of products of powers $f, f', f'', \ldots$. Write $\Phi_{[i,j]}$ for an iteration that is obtained using the Padé approximant $[i,j]$:

$$\Phi_{[0,1]}(x) = x - \frac{2ff'}{2f'^2 - ff''} \tag{13.159}$$

happens to be Householder's third order iteration.

Fourth order iterations are

$$\Phi_{[1,1]}(x) = x - \frac{f\left(2\,f\,f'\,f''' - 3\,f\,f''^2 + 6\,f'^2\,f''\right)}{f'\left(2\,f\,f'\,f''' - 6\,f\,f''^2 + 6\,f'^2\,f''\right)} \tag{13.160}$$

$$\begin{aligned} \Phi_{[0,2]}(x) &= x - \frac{12\,f\,f'^3}{\left(12\,f'^4 - 6\,f\,f'^2\,f'' + 2\,f^2\,f'\,f''' - 3\,f^2\,f''^2\right)} \\ &= x - \left(\frac{f'}{f} - \frac{f''}{2\,f'} - \frac{f\left(3\,f''^2 - 2\,f'\,f'''\right)}{12\,f'^3}\right)^{-1} \end{aligned} \tag{13.161}$$

$$\begin{aligned} \Phi_{[2,0]}(x) &= x - \frac{f\left(6\,f'^4 + 3\,f\,f'^2\,f'' - f^2\,f'\,f''' + 3\,f^2\,f''^2\right)}{6\,f'^5} \\ &= x - \frac{f}{f'} - \frac{f^2}{2\,f'^3}\cdot f'' - \frac{f^3}{6\,f'^5}\cdot\left(3f''^2 - f'f'''\right) \\ &= x - \frac{f^2}{f'^2}\left(\frac{f'}{f} + \frac{f''}{2\,f'} + \frac{f\left(3\,f''^2 - f'\,f'''\right)}{6\,f'^3}\right) \end{aligned} \tag{13.162}$$

The last one is of course just Schröder's fourth order iteration.

In general one gets $n-2$ alternative forms of iterations using the approximants $[0,n-2]$, $[1,n-3]$, $\ldots[n-3,1]$. It is not difficult to find alternative Padé based iterations like the fourth order

$$\Phi_4 = x - \frac{f\left(6\,f\,f'\,f''\,f''' - \ 9\,f\,f''^3 + 2\,f^2\,f'''^2 + 18\,f'^2\,f''^2\right)}{f'\left(6\,f\,f'\,f''\,f''' - 18\,f\,f''^3 + 2\,f^2\,f'''^2 + 18\,f'^2\,f''^2\right)} \tag{13.163}$$

that can be obtained by the $[2,2]$-approximant in f/f' and neglecting terms containing the fourth derivative of f.

The iterations obtained are fractions with numerator and denominator consisting only of terms that are products of integral constants and $f, f', f'', \ldots, f^{(n-1)}$. There are obviously other forms of iterations, e.g. the third order iteration

$$\Phi_3(x) \quad = \quad x - \frac{1}{f''}\left(f' \pm \sqrt{f'^2 - 2ff''}\right) = x - \frac{f'}{f''}\left(1 \pm \sqrt{1 - 2\,\frac{f\,f''}{f'^2}}\right) \tag{13.164}$$

that stems from directly solving the truncated Taylor expansion of $f(r) = 0$ around x.

$$f(r) \quad = \quad f(x) + f'(x)\,(r-x) + \frac{1}{2}f''(x)\,(r-x)^2 \tag{13.165}$$

(For $f(x) = ax^2 + bx + c$ it gives the two solutions of the quadratic equation $f(x) = 0$; for other functions one gets an iterated square root expression for the roots.)

Alternative rational forms can also be obtained in a way that generalizes the the method used for multiple roots: if we emphasize the so far notationally omitted dependency from the function f as $\Phi\{f\}$. The iteration $\Phi\{f\}$ has fixed points where f has a root r, so $x - \Phi\{f\}(x)$ again has a root at r. Hence we can build more iterations that will converge to those roots as $\Phi\{x - \Phi\{f\}\}(x)$. For dealing with multiple roots we used $\Phi\{x - \Phi\{f\}_2\}_k = \Phi\{f/f'\}$. An iteration $\Phi\left\{x - \Phi\{f\}_j\right\}_k$ can only be expected to have a kth order convergence.

Similarly, one can derive alternative iterations of given order by using functions that have roots where f has them[11]. For example

$$g(x) \quad := \quad 1 - \frac{1}{1 - \alpha f(x)} \quad \text{where} \quad \alpha \in \mathbb{C}, \alpha \neq 0 \tag{13.166}$$

leads to the second order iteration

$$\Phi\{g\}_2 \quad = \quad x - \frac{f(x)(1 + \alpha f(x))}{f'(x)} \tag{13.167}$$

Using $g := xf(x)$ leads to the alternative second order iteration.

Moreover, one could use a function g and its inverse $\bar{g} := g^{-1}$ and the corresponding iteration for $f(g(x))$ and finally apply g to get the root: (Let r' be the zero of $f(g(x))$: $f(g(r')) = 0$ if $g(r') = r$. r' is what we get from $\Phi\{f \circ g\}$.) A simple example is $g(x) = \bar{g}(x) = 1/x$, with $f = x^a - d$ and Schröder's formula one gets the division-less iterations for the (inverse) a-th root of d. g subject to reasonable conditions: it must be invertible near the root r of f.

13.8.5 Improvements by the delta squared process

Given a sequence of partial sums x_k the so called *delta squared process* computes a new sequence x_k^* of extrapolated sums:

$$x_k^* \quad = \quad x_{k+2} - \frac{(x_{k+2} - x_{k+1})^2}{x_{k+2} - 2\,x_{k+1} + x_k} \tag{13.168}$$

The method is due to Aitken. The name delta squared is due to the fact that the formula can be written symbolically as

$$x^* = x - \frac{(\Delta x)^2}{(\Delta^2 x)} \tag{13.169}$$

[11] It does not do any harm if g has additional roots.

where Δ is the difference operator.

Note that the mathematically equivalent form

$$x_k^* = \frac{x_k\, x_{k+2} - x_{k+1}^2}{x_{k+2} - 2\, x_{k+1} + x_k} \tag{13.170}$$

sometimes given should be avoided with numerical computations due to possible cancellation.

If $x_k = \sum_{i=0}^{k} a_i$ and the ratio of consecutive summands a_i is approximately constant (that is, a is close to a geometric series) then x^* converges significantly faster to x_∞ than x. Let us partly rewrite the formula using $x_k - x_{k-1} = a_k$:

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} \tag{13.171}$$

Then for a geometric series with $a_{k+1}/a_k = q$

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} = x_{k+2} - \frac{\left(a_0\, q^{k+2}\right)^2}{a_0\,(q^{k+2} - q^{k+1})} \tag{13.172a}$$

$$= a_0\, \frac{1 - q^{k+3}}{1 - q} + a_0\, q^{k+2} \cdot \frac{q^{k+2}}{q^{k+1} - q^{k+2}} = \frac{a_0}{1 - q}\, \left(1 - q^{k+3} + q^{k+3}\right) \tag{13.172b}$$

$$= \frac{a_0}{1 - q} \tag{13.172c}$$

which is the exact sum.

Why do we meet the delta squared here? Consider the sequence

$$x_0, \quad x_1 = \Phi(x_0), \quad x_2 = \Phi(x_1) = \Phi(\Phi(x_0)), \quad \ldots \tag{13.173}$$

of better and better approximations to some root r of a function f. Think of the x_k as partial sums of a series whose sum is the root r. Apply the idea to define an improved iteration Φ^* from a given Φ:

$$\Phi^*(x) = \Phi(\Phi(x)) - \frac{(\Phi(\Phi(x)) - \Phi(x))^2}{\Phi(\Phi(x)) - 2\,\Phi(x) + x} \tag{13.174}$$

The good news is that $\Phi*$ will give quadratic convergence even if Φ only gives linear convergence. As an example let us take $f(x) = (x^2 - d)^2$, forget that its root $\sqrt{d}$ is a double root and happily define $\Phi(x) = x - f(x)/f'(x) = x - (x^2 - d)/(4x)$. Convergence is only linear:

$$\Phi(\sqrt{d} \cdot (1 + e)) = \sqrt{d} \cdot \left(1 + \frac{e}{2} + \frac{e^2}{4} + O(e^3)\right) \tag{13.175}$$

Then try

$$\Phi^*(x) = \frac{d\,(7\,x^2 + d)}{x\,(3\,x^2 + 5\,d)} \tag{13.176}$$

and find that it offers quadratic convergence

$$\Phi(\sqrt{d} \cdot (1 + e)) = \sqrt{d} \cdot \left(1 - \frac{e^2}{4} + \frac{e^3}{16} + O(e^4)\right) \tag{13.177}$$

In general, if Φ has convergence of order $n > 1$ then Φ^* will be of order $2n - 1$, linear convergence ($n = 1$) is turned into second order. (See [10]).

13.8.6 Improvements of the delta squared process *

Let us rewrite the partial sums of equation 13.168 as values of a function F: $F(k) := x_k$. The delta squared process extrapolated the function F:

$$F(\infty) \approx F(k) - \frac{F'(k)^2}{F''(k)} \tag{13.178}$$

if we approximate the n-th derivatives using the backward difference operator Δ as

$$\begin{aligned}
F'(k) &\approx \Delta\ F(k) = F(k) - F(k-1) && (13.179a)\\
F''(k) &\approx \Delta^2 F(k) = F(k) - 2\,F(k-1) + F(k-2) && (13.179b)\\
F^{(3)}(k) &\approx \Delta^3 F(k) = F(k) - 3\,F(k-1) + 3\,F(k-2) - F(k-3) && (13.179c)\\
F^{(n)} &\approx \Delta^n\,F(k) = \sum_{i=0}^{k} (-1)^i \binom{k}{i} F(k-i) && (13.179d)
\end{aligned}$$

and identify

$$\begin{aligned}
F(0) &= x_0 && (13.180a)\\
F(1) &= f(x) = x_1 && (13.180b)\\
F(2) &= f(f(x)) = x_2 && (13.180c)\\
F(3) &= f(f(f(x))) = x_3 && (13.180d)\\
F(k) &= f^{\circ k}(x) = x_k && (13.180e)
\end{aligned}$$

where $f^{\circ k}(.)$ denotes the k-fold composition of $f(.)$. We had $f = \Phi$ before. Write the Taylor expansion as

$$F(k+D) = F(k) + F'(k)\,D + \frac{F''(k)}{2}\,D^2 + \frac{F'''(k)}{6}\,D^3 + \ldots \tag{13.181}$$

Use the second order iteration for an extremum of F

$$\Phi\left\{F'\right\}_2(k) = X - \frac{F'(k)}{F''(k)} \tag{13.182}$$

set $X := F(k)$

$$\Phi\left\{F'\right\}_2(k) = F(k) - \frac{F'(k)}{F''(k)} = F(k) + D \tag{13.183}$$

and insert $D = \Phi - F$ into equation 13.181 up to first order:

$$F(k+D) = F(k) + F'(k)\,D = F(k) + F'(k)\,\frac{F'(k)}{F''(k)} \tag{13.184}$$

This is Aitken's delta squared method.

Note that for the numerical process it is not necessary to know F algebraically, one just needs the values of $F(k)$, the partial sums.

There were two degrees of freedom in our derivation: the order a after which the Taylor series in D was truncated and the order b of the iteration used for $D = \Phi\left\{F'\right\} - F$. Define an extrapolation operator

$$\begin{aligned}
T_{[a,b]}(F) &:= F(k) + F'(k)\,D + \frac{F''(k)}{2}\,D^2 + \ldots + \frac{F^{(a)}}{a!}\,D^a && (13.185)\\
&\text{where} \quad D := \Phi\left\{F'\right\}_b - F && (13.186)
\end{aligned}$$

Φ_b can be any iteration of order b, we'll use Householder's variant in what follows. $T_{[1,2]}$ is the delta squared operator which 'improves' given partial sums (or sequence of functional iterates) of order n to order $2n-1$.

$T_{[2,2]}$ is a bit of a surprise:

$$T_{[2,2]}(F) = F + F' \left(-\frac{F'}{F''}\right) + \frac{F''}{2} \left(-\frac{F'}{F''}\right)^2 = F - \frac{1}{2}\frac{F'^2}{F''} \tag{13.187}$$

and gets order $2n^2 - n$ out of n. A linearly convergent iteration will not be made super-linear by $T_{[2,2]}$.

In general, $T_{[a,2]}$

$$T_{[a,2]}(F) = F + \sum_{i=1}^{a} \frac{F^{(i)}}{i!} \left(-\frac{F'}{F''}\right)^i \tag{13.188}$$

$$= F - \frac{1}{2}\frac{F'^2}{F''} - \frac{F''' F'^3}{6 F''^3} + \frac{F'''' F'^4}{24 F''^4} \pm \ldots \tag{13.189}$$

takes order n to $n^{a-1}(2n-1) = 2n^a - n^{a-1}$.

$T_{[a,b]}$ also (only) takes order n to $n^{a-1}(2n-1) = 2n^a - n^{a-1}$. Linear convergence is turned into second order for $a = b - 1$.

Here is what we get:

	$n=2$	$n=3$	$n=4$	n
$a=1$	3	5	7	$2n-1$
$a=2$	6	15	28	$2n^2-n$
$a=3$	12	45	112	$2n^3-n^2$
$a=4$	24	135	448	$2n^4-n^3$
$a=5$	48	1215	1792	$2n^5-n^4$
a	$2^{a+1}-2^{a-1}$	$2\,3^a - 3^{a-1}$	$2\,4^a - 4^{a-1}$	$2n^a - n^{a-1}$

Using a Padé approximant for the expansion in D and the general Padé type iteration $\Phi_{[]}$ one can define the extrapolation operator

$$P_{[a,a'][b,b']}(F) := \text{Padé}_{[a,a']}\Big(F(x+D)\Big) \tag{13.190}$$

$$\text{where} \quad D := \Phi\left\{F'\right\}_{[b,b']} - F \tag{13.191}$$

$P_{[1,0][2,0]}$ is Aitken's delta squared process. For $a=1$, $a'=1$, $b=2$, $b'=1$ one gets

$$P_{[1,1][2,1]}(F) = F - \frac{2F'^2 D}{F'' D - 2F'} = \quad \text{with} \quad D = \frac{2F'F''}{F'F''' - 2F''^2} \tag{13.192}$$

$$= F - \frac{2F'^2 F''}{3F''^2 - F' F'''} \tag{13.193}$$

which makes linear convergence quadratic and takes super-linear convergence to $2n^2 - n$.

13.9 Transcendental functions & the AGM

13.9.1 The AGM

The AGM (arithmetic geometric mean) plays a central role in the high precision computation of logarithms and π.

The $AGM(a,b)$ is defined as the limit of the iteration AGM iteration, cf.13.194a :

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{13.194a}$$
$$b_{k+1} = \sqrt{a_k\, b_k} \tag{13.194b}$$

starting with $a_0 = a$ and $b_0 = b$. Both of the values converge quadratically to a common limit. The related quantity c_k (used in many AGM based computations) is defined as

$$c_k^2 = a_k^2 - b_k^2 \tag{13.195}$$
$$= (a_{k-1} - a_k)^2 \tag{13.196}$$

One further defines (cf. sections 13.9.5 and 13.9.6)

$$R'(k) := \left[1 - \frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2\right]^{-1} \tag{13.197}$$

where $c_n^2 := a_n^2 - b_n^2$ corresponding to $AGM(1,k)$.
An alternative way for the computation for the AGM iteration is

$$c_{k+1} = \frac{a_k - b_k}{2} \tag{13.198a}$$
$$a_{k+1} = \frac{a_k + b_k}{2} \tag{13.198b}$$
$$b_{k+1} = \sqrt{a_{k+1}^2 - c_{k+1}^2} \tag{13.198c}$$

Schönhage gives the most economic variant of the AGM, which, apart from the square root, only needs one squaring per step:

$$A_0 = a_0^2 \tag{13.199a}$$
$$B_0 = b_0^2 \tag{13.199b}$$
$$t_0 = 1 - (A_0 - B_0) \tag{13.199c}$$
$$S_k = \frac{A_k + B_k}{4} \tag{13.199d}$$
$$b_k = \sqrt{B_k} \quad \text{square root computation} \tag{13.199e}$$
$$a_{k+1} = \frac{a_k + b_k}{2} \tag{13.199f}$$
$$A_{k+1} = a_{k+1}^2 \quad \text{squaring} \tag{13.199g}$$
$$= \left(\frac{\sqrt{A_k} + \sqrt{B_k}}{2}\right)^2 = \frac{A_k + B_k}{4} + \frac{\sqrt{A_k\, B_k}}{2} \tag{13.199h}$$
$$B_{k+1} = 2\,(A_{k+1} - S_K) \quad = b_{k+1}^2 \tag{13.199i}$$
$$c_{k+1}^2 = A_{k+1} - B_{k+1} \tag{13.199j}$$
$$t_{k+1} = t_k - 2^{k+1}\, c_{k+1}^2 \tag{13.199k}$$

Starting with $a_0 = A_0 = 1$, $B_0 = 1/2$ one has $\pi \approx (2\, a_n^2)/t_n$.

Combining two steps of the AGM iteration leads to the 4th order AGM iteration:

$$\begin{aligned}
\alpha_0 &= \sqrt{a_0} && (13.200a)\\
\beta_0 &= \sqrt{b_0} && (13.200b)\\
\alpha_{k+1} &= \frac{\alpha_k + \beta_k}{2} && (13.200c)\\
\beta_{k+1} &= \left(\frac{\alpha_k\,\beta_k\,(\alpha_k^2+\beta_k^2)}{2}\right)^{1/4} && (13.200d)\\
\gamma_k^4 &= \alpha_k^4 - \beta_k^4 \qquad = c_{k/2}^2 && (13.200e)
\end{aligned}$$

(Note that $\alpha_k = \sqrt{a_{2k}}$ and $\beta_k = \sqrt{b_{2k}}$.) and

$$R'(k) = \left[1 - \sum_{n=0}^{\infty} 4^n \left(\alpha_n^4 - \left(\frac{\alpha_n^2+\beta_n^2}{2}\right)^2\right)\right]^{-1} \qquad (13.201)$$

corresponding to $AGM4(1,\sqrt{k})$ (cf. [8] p.17).

An alternative formulation of the 4th order AGM iteration is:

$$\begin{aligned}
\gamma_{k+1} &= \frac{\alpha_k - \beta_k}{2} && (13.202a)\\
\alpha_{k+1} &= \frac{\alpha_k + \beta_k}{2} && (13.202b)\\
\beta_{k+1} &= \left(\alpha_{k+1}^4 - \gamma_{k+1}^4\right)^{1/4} && (13.202c)\\
c_{k/2}^2 + 2\,c_{k/2+1}^2 &= \alpha_{k-1}^4 - \left(\alpha_k^2 - \gamma_k^2\right)^2 && (13.202d)
\end{aligned}$$

13.9.2 log

The (natural) logarithm can be computed using the following relation (cf. [8] p.221)

$$\begin{aligned}
\left|\log(x) - R'(10^{-n}) + R'(10^{-n}\,x)\right| &\le \frac{n}{10^{2(n-1)}} && (13.203)\\
\log(x) &\approx R'(10^{-n}) - R'(10^{-n}\,x) && (13.204)
\end{aligned}$$

that holds for $n \ge 3$ and $x \in]\frac{1}{2}, 1[$. Note that the first term on the rhs. is constant and might be stored for subsequent log-computations. See also section 13.10.

[hfloat: `src/tz/log.cc`]

If one has some efficient algorithm for exp() one can compute log() from exp() using

$$\begin{aligned}
y &:= 1 - d\,e^{-x} && (13.205)\\
\log(d) &= x + \log(1-y) && (13.206)\\
&= x + \log\left(1 - (1 - d\,e^{-x})\right) = x + \log\left(e^{-x} d\right) = x + (-x + \log(d)) && (13.207)
\end{aligned}$$

Then

$$\log(d) = x + \log(1-y) = x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \dots\right) \qquad (13.208)$$

Truncation of the series after the n-th power of y gives an iteration of order $n+1$:

$$x_{k+1} = \Phi_n(x_k) := x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \dots + \frac{y^{n-1}}{n-1}\right) \qquad (13.209)$$

Padé series $P_{[i,j]}(z)$ of $\log(1-z)$ at $z=0$ produce (order $i+j+2$) iterations. For $i=j$ we get

$$[i,j] \mapsto x + P_{[i,j]}(z = 1 - d\,e^{-x}) \tag{13.210a}$$
$$[0,0] \mapsto x - z \tag{13.210b}$$
$$[1,1] \mapsto x - z \cdot \frac{6-z}{6-4z} \tag{13.210c}$$
$$[2,2] \mapsto x - z \cdot \frac{30-21z+z^2}{30-36z+9z^2} \tag{13.210d}$$
$$[4,4] \mapsto x - z \cdot \frac{3780-6510z+3360z^2-505z^3+6z^4}{3780-8400z+6300z^2-1800z^3+150z^4} \tag{13.210e}$$

Compared to the power series based iteration one needs one additional long division but saves half of the exponentiations. This can be a substantial saving for high order iterations.

13.9.3 exp

The exponential function can be computed using the iteration that is obtained as follows:

$$\exp(d) = x\,\exp(d-\log(x)) \tag{13.211}$$
$$= x\,\exp(y) \quad \text{where} \quad y := d-\log(x) \tag{13.212}$$
$$= x\left(1+y+\frac{y^2}{2}+\frac{y^3}{3!}+\dots\right) \tag{13.213}$$

The corresponding n-th oder iteration is

$$x_{k+1} = \Phi_n(x_k) := x_k\left(1+y+\frac{y^2}{2}+\frac{y^3}{3!}+\dots\frac{y^{n-1}}{(n-1)!}\right) \tag{13.214}$$

As the computation of logarithms is expensive one should use a higher (e.g. 8th) order iteration.

[hfloat: `src/tz/itexp.cc`]

Padé series $P_{[i,j]}(z)$ of $\exp(z)$ at $z=0$ produce (order $i+j+1$) iterations. For $i=j$ we get

$$[i,j] \mapsto x\,P_{[i,j]}(z = d-\log x) \tag{13.215a}$$
$$[1,1] \mapsto x \cdot \frac{z+2}{z-2} \tag{13.215b}$$
$$[2,2] \mapsto x \cdot \frac{12+6z+z^2}{12-6z+z^2} \tag{13.215c}$$
$$[4,4] \mapsto x \cdot \frac{1680+840z+180z^2+20z^3+z^4}{1680-840z+180z^2-20z^3+z^4} \tag{13.215d}$$

The $[i,j]$-th Padé approximant of $\exp(z)$ is

$$P_{[i,j]}(z) = \left\{\sum_{k=0}^{i}\frac{\binom{i}{k}}{\binom{i+j}{k}}\frac{z^k}{k!}\right\} \Big/ \left\{\sum_{k=0}^{j}\frac{\binom{j}{k}}{\binom{i+j}{k}}\frac{(-z)^k}{k!}\right\} \tag{13.216}$$

The numerator for $i=j$ (multiplied by $(2i)!/i!$ in order to avoid rational coefficients) is

$$= \frac{(2i)!}{i!}\cdot\sum_{k=0}^{i}\frac{\binom{i}{k}}{\binom{2\,i}{k}}\frac{z^k}{k!} \tag{13.217}$$

13.9.4 sin, cos and tan

For arcsin, arccos and arctan use the complex analogue of the AGM. For sin, cos and tan use the exp iteration above think complex.

13.9.5 Elliptic K

The function K can be defined as

$$K(k) = \int_0^{\pi/2} \frac{d\,\vartheta}{\sqrt{1-k^2\,\sin^2\vartheta}} = \int_0^1 \frac{d\,t}{\sqrt{(1-t^2)\,(1-k^2t^2)}} \tag{13.218}$$

One has

$$K(k) = \frac{\pi}{2}\,F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\Big|\,k^2\right) \tag{13.219}$$

$$= \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2 k^{2i} \tag{13.220}$$

$$= \frac{\pi}{2}\left(1+\left(\frac{1}{2}\right)^2 k^2+\left(\frac{1\cdot 3}{2\cdot 4}\right)^2 k^4+\left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2 k^6+\ldots\right) \tag{13.221}$$

$$= \frac{\pi}{2}\left(1+\frac{1}{4}\,k^2+\frac{9}{64}\,k^4+\frac{25}{256}\,k^6+\frac{1225}{16384}\,k^8+\frac{3969}{65536}\,k^{10}+\ldots\right) \tag{13.222}$$

A special value is

$$K(0) = \frac{\pi}{2} \tag{13.223}$$

The computational interesting form is

$$K(k) = \frac{\pi}{2\,AGM(1,k')} = \frac{\pi}{2\,AGM(1,1-k^2)} \tag{13.224}$$

One defines $k' = 1-k^2$ and K' as

$$K'(k) := K(k') = K(1-k^2) = \frac{\pi}{2\,AGM(1,k)} \tag{13.225}$$

[hfloat: `src/tz/elliptick.cc`]

Product forms for K and K' that are also candidates for fast computations are

$$\frac{2}{\pi}\,K'(k_0) = \prod_{n=0}^{\infty}\frac{2}{1+k_n} = \prod_{n=1}^{\infty} 1+k'_n \tag{13.226}$$

$$\text{where}\quad k_{n+1} := \frac{2\sqrt{k_n}}{1+k_n},\quad 0<k_0\le 1$$

$$\frac{2}{\pi}\,K'(k_0) = \prod_{n=0}^{\infty}\frac{1}{\sqrt{k_n}} \tag{13.227}$$

$$\text{where}\quad k_{n+1} := \frac{1+k_n}{2\sqrt{k_n}},\quad 0<k_0\le 1$$

These follow directly from 13.225 (and $AGM(a,b) = a\,AGM(1,b/a)$):

$$\frac{1}{AGM(1,k)} = \left(AGM(\frac{1+k}{2},\sqrt{k})\right)^{-1} \tag{13.228}$$

$$= \left(\frac{1+k}{2}\,AGM(1,\frac{2\sqrt{k}}{1+k})\right)^{-1} \qquad \text{(first form)}$$

$$= \left(\sqrt{k}\,AGM(\frac{1+k}{2\sqrt{k}},1)\right)^{-1} \qquad \text{(second form)}$$

The second form seems computationally more effective. Similarly,

$$\frac{2}{\pi}\,K(k_0) \;=\; \prod_{n=0}^{\infty}\frac{2}{1+k'_n} \;=\; \prod_{n=1}^{\infty} 1+k_n \tag{13.229}$$

$$\text{where}\quad k_{n+1} := \frac{1-k'_n}{1+k'_n} = \frac{1-\sqrt{1-k_n^2}}{1+\sqrt{1-k_n^2}},\quad 0<k_0\le 1$$

With an efficient algorithm for K the logarithm can be computed using

$$\left|K'(k)-\log\left(\frac{4}{k}\right)\right| \;\le\; 4k^2\;(8+|\log k|)\quad\text{where}\quad 0<k\le 1 \tag{13.230}$$

13.9.6 Elliptic E

The function E can be defined as

$$E(k) \;=\; \int_0^{\pi/2}\sqrt{1-k^2\,\sin^2\vartheta}\,d\,\vartheta = \int_0^1 \frac{\sqrt{1-k^2t^2}}{\sqrt{1-t^2}}\,d\,t \tag{13.231}$$

One has

$$E(k) \;=\; \frac{\pi}{2}\,F\left(\begin{matrix}-\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{13.232}$$

$$\;=\; \frac{\pi}{2}\left(1-\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2\frac{k^{2i}}{2i-1}\right) \tag{13.233}$$

$$\;=\; \frac{\pi}{2}\left(1-\left(\frac{1}{2}\right)^2k^2-\left(\frac{1\cdot 3}{2\cdot 4}\right)^2\frac{k^4}{3}-\left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2\frac{k^6}{5}-\ldots\right) \tag{13.234}$$

$$\;=\; \frac{\pi}{2}\left(1-\frac{1}{4}\,k^2-\frac{3}{64}\,k^4-\frac{5}{256}\,k^6-\frac{175}{16384}\,k^8-\frac{441}{65536}\,k^{10}-\ldots\right) \tag{13.235}$$

Special values are

$$E(0) \;=\; \frac{\pi}{2}\qquad E(1)=1 \tag{13.236}$$

(The latter leads to a series for $2/\pi$). The key to fast computations is

$$1-\frac{E}{K} \;=\; 1-R = \frac{1}{2}\sum_{n=0}^{\infty}2^n c'^2_n \tag{13.237}$$

(see [7] p.8) and so

$$R'(k) \;:=\; \frac{K'(k)}{E'(k)} \;=\; \left[1-\frac{1}{2}\sum_{n=0}^{\infty}2^n c_n^2\right]^{-1} \tag{13.238}$$

$$E(k) \;=\; R'(k)\,K(k) = \frac{\pi}{2\,AGM(1,1-k^2)\cdot\left(1-\sum_{n=0}^{\infty}2^{n-1}c_n^2\right)} \tag{13.239}$$

Similar as for K' one defines

$$E'(k) \;:=\; E(k') = E(1-k^2) \tag{13.240}$$

An ellipse with major axis a and minor axis b has the circumference (or arclength) $L = 4\,a\,E'(b/a) = 4\,a\,E(e)$. The quantity $e = \sqrt{1 - a^2/b^2}$ is the eccentricity of the ellipse.

Legendre's relation between K and E is (arguments omitted for readability, choose your favorite form):

$$\frac{E}{K} + \frac{E'}{K'} - 1 = \frac{\pi}{2\,K\,K'} \tag{13.241}$$

$$E\,K' + E'\,K - K\,K' = \frac{\pi}{2} \tag{13.242}$$

Equivalently,

$$AGM(1,k) = \frac{E/K}{(1 - E'/K')} = \frac{R}{(1 - R')} \tag{13.243}$$

For $k = \frac{1}{\sqrt{2}} =: s$ we have $k = k'$, thereby $K = K'$ and $E = E'$, so

$$\frac{K(s)}{\pi}\left(\frac{2\,E(s)}{\pi} - \frac{K(s)}{\pi}\right) = \frac{1}{2\,\pi} \tag{13.244}$$

As expressions 13.224 and 13.239 provide a fast AGM based computation of $\frac{K}{\pi}$ and $\frac{E}{\pi}$ the above formula can be used to compute π (cf. [8]).

Using $E - K = k\,k'\,\frac{dK}{dk} - k^2\,K$ one can express the derivative of K in terms of E and K and thereby compute that quantity fast:

$$\frac{dK}{dk} = \frac{E - k'^2\,K}{k\,k'^2} \tag{13.245}$$

13.10 Computation of $\pi/\log(q)$

For the computation of the natural logarithm one can use the relation

$$\log(m\,r^x) = \log(m) + x\,\log(r) \tag{13.246}$$

where m is the mantissa and r the radix of the floating point numbers.

There is a nice way to compute the value of $\log(r)$ if the value of π has been precomputed. One defines

$$\theta_2(q) = \sum_{n=-\infty}^{\infty} q^{(n+1/2)^2} \tag{13.247}$$

$$\theta_3(q) = \sum_{n=-\infty}^{\infty} q^{n^2} \tag{13.248}$$

$$\theta_4(q) = \sum_{n=-\infty}^{\infty} (-1)^n\,q^{n^2} \tag{13.249}$$

Then

$$\log\frac{1}{q} = -\log q = \pi\,\frac{K'}{K} \tag{13.250}$$

(so $q = \exp(-\pi\,K'/K)$) and (cf. [8] p.225)

$$\frac{\pi}{\log(1/q)} = -\frac{\pi}{\log(q)} = AGM\left(\theta_3(q)^2, \theta_2(q)^2\right) \tag{13.251}$$

Computing $\theta_3(q)$ is easy when $q = 1/r$:

$$\theta_3(q) \;=\; 1 + 2\sum_{n=1}^{\infty} q^{n^2} \;=\; 2\,(1 + \sum_{n=1}^{\infty} q^{n^2}) - 1 \tag{13.252}$$

However, the computation of $\theta_2(q)$ suggests to choose $q = 1/r^4 =: b^4$:

$$\theta_2(q) \;=\; 0 + 2\sum_{n=0}^{\infty} q^{(n+1/2)^2} \;=\; 2\sum_{n=0}^{\infty} b^{4n^2+4n+1} \quad \text{where} \quad q = b^4 \tag{13.253}$$

$$\;=\; 2\,b\sum_{n=0}^{\infty} q^{n^2+n} \;=\; 2\,b\,(1 + \sum_{n=1}^{\infty} q^{n^2+n}) \tag{13.254}$$

[hfloat: `src/tz/pilogq.cc`]
One has (see [22])

$$\Theta_2^2(q) \;=\; \frac{2\,k\,K}{\pi} \qquad \Theta_3^2(q) \;=\; \frac{2\,K}{\pi} \qquad \Theta_4^2(q) \;=\; \frac{2\,k'\,K}{\pi} \tag{13.255}$$

$$k \;=\; \frac{\Theta_2^2(q)}{\Theta_3^2(q)} \qquad\qquad k' \;=\; \frac{\Theta_4^2(q)}{\Theta_3^2(q)} \tag{13.256}$$

$$\frac{\pi}{\log(q)} \;=\; -\frac{AGM(1,k)}{AGM(1,k')} \tag{13.257}$$

$$1 \;=\; AGM\left(\Theta_3^2(q), \Theta_4^2(q)\right) \tag{13.258}$$

13.11 Computation of $q = \exp(-\pi\,K'/K)$

We use (cf. [7] p.35/36)

$$q \;=\; \exp\left(-\pi\,\frac{K'}{K}\right) \tag{13.259}$$

Following the usual convention, we write

$$\frac{K'}{K} \;=\; \frac{AGM(1,k')}{AGM(1,k)} \;=\; \frac{AGM(1,b_0)}{AGM(1,b'_0)} \tag{13.260}$$

where $k' = b_0$ and $k = b'_0 = \sqrt{1-b_0^2}$ and use ([7] p.38, note the missing '4' there)

$$\frac{\pi}{2}\,\frac{AGM(1,b_0)}{AGM(1,b'_0)} \;=\; \lim_{n\to\infty} \frac{1}{2^n} \log \frac{4\,a_n}{c_n} \tag{13.261}$$

thereby

$$q \;=\; \exp\left(-2\,\lim_{n\to\infty} \frac{1}{2^n}\,\log\frac{4\,a_n}{c_n}\right) \;=\; \lim_{n\to\infty} \exp\left(-\frac{1}{2^{n-1}}\,\log\frac{4\,a_n}{c_n}\right) \tag{13.262}$$

$$\;=\; \lim_{n\to\infty} \left(\exp\log\frac{4\,a_n}{c_n}\right)^{-1/2^{n-1}} \;=\; \lim_{n\to\infty} \left(\frac{4\,a_n}{c_n}\right)^{-1/(2^{n-1})} \tag{13.263}$$

so

$$q \;=\; \lim_{n\to\infty} \left(\frac{c_n}{4\,a_n}\right)^{1/(2^{n-1})} \tag{13.264}$$

One obtains an algorithm for $\exp(-x)$ by first solving for k, k' so that $x = \pi\, K'/K$ (precomputed π) and applying the last relation that implies the computation of a 2^{n-1}-th root. Note that the quantity c should be computed via $c_{n+1} = \frac{c_n^2}{4\, a_{n+1}}$ throughout the AGM computation in order to preserve its accuracy. (cf. also [8] p.227)

For $k = 1/\sqrt{2} =: s$ one has $k = k'$ and so $q = \exp(-\pi)$. Thus the calculation of $\exp(-\pi) = 0.0432139182637\ldots$ can directly be done via a single AGM computation as $(c_n/(4a_n))^N$ where $N = 1/2^{(n-1)}$. The quantity $i^i = \exp(-\pi/2) = 0.2078795763507\ldots$ can be obtained using $N = 1/2^n$. (cf. also [7] p.13)

13.12 Iterations for high precision computations of π

In this section various iterations for computing π with at least second order convergence are given.

The number of full precision multiplications (FPM) are an indication of the efficiency of the algorithm. The approximate number of FPMs that were counted with a computation of π to 4 million decimal digits[12] is indicated like this: #FPM=123.4.

AGM as in [hfloat: `src/pi/piagm.cc`], #FPM=98.4 (#FPM=149.3 for the quartic variant):

$$a_0 = 1 \tag{13.265a}$$

$$b_0 = \frac{1}{\sqrt{2}} \tag{13.265b}$$

$$p_n = \frac{2\, a_{n+1}^2}{1 - \sum_{k=0}^{n} 2^k\, c_k^2} \quad \to \pi \tag{13.265c}$$

$$\pi - p_n = \frac{\pi^2\, 2^{n+4}\, e^{-\pi\, 2^{n+1}}}{AGM^2(a_0, b_0)} \tag{13.265d}$$

A fourth order version uses 13.200a, cf. also [hfloat: `src/pi/piagm.cc`].

AGM variant as in [hfloat: `src/pi/piagm3.cc`], #FPM=99.5 (#FPM=155.3 for the quartic variant):

$$a_0 = 1 \tag{13.266a}$$

$$b_0 = \frac{\sqrt{6} + \sqrt{2}}{4} \tag{13.266b}$$

$$p_n = \frac{2\, a_{n+1}^2}{\sqrt{3}\,(1 - \sum_{k=0}^{n} 2^k\, c_k^2) - 1} \quad \to \pi \tag{13.266c}$$

$$\pi - p_n < \frac{\sqrt{3}\,\pi^2\, 2^{n+4}\, e^{-\sqrt{3}\,\pi\, 2^{n+1}}}{AGM^2(a_0, b_0)} \tag{13.266d}$$

AGM variant as in [hfloat: `src/pi/piagm3.cc`], #FPM=108.2 (#FPM=169.5 for the quartic variant):

$$a_0 = 1 \tag{13.267a}$$

$$b_0 = \frac{\sqrt{6} - \sqrt{2}}{4} \tag{13.267b}$$

$$p_n = \frac{6\, a_{n+1}^2}{\sqrt{3}\,(1 - \sum_{k=0}^{n} 2^k\, c_k^2) + 1} \quad \to \pi \tag{13.267c}$$

$$\pi - p_n < \frac{\frac{1}{\sqrt{3}}\,\pi^2\, 2^{n+4}\, e^{-\frac{1}{\sqrt{3}}\,\pi\, 2^{n+1}}}{AGM(a_0, b_0)^2} \tag{13.267d}$$

[12] using radix $10,000$ and 1 million LIMBs.

Borwein's quartic (fourth order) iteration, variant $r = 4$ as in [hfloat: `src/pi/pi4th.cc`], #FPM=170.5:

$$\begin{aligned}
y_0 &= \sqrt{2} - 1 && (13.268a)\\
a_0 &= 6 - 4\sqrt{2} && (13.268b)\\
y_{k+1} &= \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}} \quad \to 0+ && (13.268c)\\
&= \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} && (13.268d)\\
a_{k+1} &= a_k\,(1 + y_{k+1})^4 - 2^{2k+3}\,y_{k+1}\,(1 + y_{k+1} + y_{k+1}^2) \quad \to \frac{1}{\pi} && (13.268e)\\
&= a_k\,((1 + y_{k+1})^2)^2 - 2^{2k+3}\,y_{k+1}\,((1 + y_{k+1})^2 - y_{k+1}) && (13.268f)\\
0 &< a_k - \pi^{-1} \le 16 \cdot 4^n\,2\,e^{-4^n\,2\,\pi} && (13.268g)
\end{aligned}$$

Identities 13.268d and 13.268f show how to save operations.

Borwein's quartic (fourth order) iteration, variant $r = 16$ as in [hfloat: `src/pi/pi4th.cc`], #FPM=164.4:

$$\begin{aligned}
y_0 &= \frac{1 - 2^{-1/4}}{1 + 2^{-1/4}} && (13.269a)\\
a_0 &= \frac{8/\sqrt{2} - 2}{(2^{-1/4} + 1)^4} && (13.269b)\\
y_{k+1} &= \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} \quad \to 0+ && (13.269c)\\
a_{k+1} &= a_k\,(1 + y_{k+1})^4 - 2^{2k+4}\,y_{k+1}\,(1 + y_{k+1} + y_{k+1}^2) \quad \to \frac{1}{\pi} && (13.269d)\\
0 &< a_k - \pi^{-1} \le 16 \cdot 4^n\,4\,e^{-4^n\,4\,\pi} && (13.269e)
\end{aligned}$$

Same operation count as before, but this variant gives approximately twice as much precision after the same number of steps.

The general form of the quartic iterations (13.268a and 13.269a) is

$$\begin{aligned}
y_0 &= \sqrt{\lambda^*(r)} && (13.270a)\\
a_0 &= \alpha(r) && (13.270b)\\
y_{k+1} &= \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} \quad \to 0+ && (13.270c)\\
a_{k+1} &= a_k\,(1 + y_{k+1})^4 - 2^{2k+2}\,\sqrt{r}\,y_k\,(1 + y_{k+1} + y_{k+1}^2) \quad \to \frac{1}{\pi} && (13.270d)\\
0 &< a_k - \pi^{-1} \le \quad 16 \cdot 4^n\,\sqrt{r}\,e^{-4^n\,\sqrt{r}\,\pi} && (13.270e)
\end{aligned}$$

Cf. [8], p.170f.

Derived AGM iteration (second order) as in [hfloat: `src/pi/pideriv.cc`], #FPM=276.2:

$$x_0 = \sqrt{2} \tag{13.271a}$$
$$p_0 = 2+\sqrt{2} \tag{13.271b}$$
$$y_1 = 2^{1/4} \tag{13.271c}$$
$$x_{k+1} = \frac{1}{2}\left(\sqrt{x_k}+\frac{1}{\sqrt{x_k}}\right) \quad (k\geq 0) \quad \to 1+ \tag{13.271d}$$
$$y_{k+1} = \frac{y_k\sqrt{x_k}+\frac{1}{\sqrt{x_k}}}{y_k+1} \quad (k\geq 1) \quad \to 1+ \tag{13.271e}$$
$$p_{k+1} = p_k\frac{x_k+1}{y_k+1} \quad (k\geq 1) \quad \to \pi+ \tag{13.271f}$$
$$p_k-\pi = 10^{-2^{k+1}} \tag{13.271g}$$

Cubic AGM from [46], as in [hfloat: `src/pi/picubagm.cc`], #FPM=182.7:

$$a_0 = 1 \tag{13.272a}$$
$$b_0 = \frac{\sqrt{3}-1}{2} \tag{13.272b}$$
$$a_{n+1} = \frac{a_n+2\,b_n}{3} \tag{13.272c}$$
$$b_{n+1} = \sqrt[3]{\frac{b_n\,(a_n^2+a_n\,b_n+b_n^2)}{3}} \tag{13.272d}$$
$$p_n = \frac{3\,a_n^2}{1-\sum_{k=0}^{n}3^k\,(a_k^2-a_{k+1}^2)} \tag{13.272e}$$

Second order iteration, as in [hfloat: `src/pi/pi2nd.cc`], #FPM=255.7:

$$y_0 = \frac{1}{\sqrt{2}} \tag{13.273a}$$
$$a_0 = \frac{1}{2} \tag{13.273b}$$
$$y_{k+1} = \frac{1-(1-y_k^2)^{1/2}}{1+(1-y_k^2)^{1/2}} \quad \to 0+ \tag{13.273c}$$
$$= \frac{(1-y_k^2)^{-1/2}-1}{(1-y_k^2)^{-1/2}+1} \tag{13.273d}$$
$$a_{k+1} = a_k\,(1+y_{k+1})^2-2^{k+1}\,y_{k+1} \quad \to \frac{1}{\pi} \tag{13.273e}$$
$$a_k-\pi^{-1} \leq 16\cdot 2^{k+1}\,e^{-2^{k+1}\,\pi} \tag{13.273f}$$

13.273d shows how to save 1 multiplication per step (cf. section 13.3).

Quintic (5th order) iteration from the article [43], as in [hfloat: `src/pi/pi5th.cc`], #FPM=353.2:

$$s_0 = 5(\sqrt{5}-2) \tag{13.274a}$$
$$a_0 = \frac{1}{2} \tag{13.274b}$$
$$s_{n+1} = \frac{25}{s_n(z+x/z+1)^2} \quad \to 1 \tag{13.274c}$$
$$\text{where } x = \frac{5}{s_n} - 1 \quad \to 4 \tag{13.274d}$$
$$\text{and } y = (x-1)^2 + 7 \quad \to 16 \tag{13.274e}$$
$$\text{and } z = \left(\frac{x}{2}\left(y + \sqrt{y^2 - 4x^3}\right)\right)^{1/5} \quad \to 2 \tag{13.274f}$$
$$a_{n+1} = s_n^2 a_n - 5^n \left(\frac{s_n^2 - 5}{2} + \sqrt{s_n\left(s_n^2 - 2s_n + 5\right)}\right) \quad \to \frac{1}{\pi} \tag{13.274g}$$
$$a_n - \frac{1}{\pi} < 16 \cdot 5^n \, e^{-\pi \, 5^n} \tag{13.274h}$$

Cubic (third order) iteration from [44], as in [hfloat: `src/pi/pi3rd.cc`], #FPM=200.3:

$$a_0 = \frac{1}{3} \tag{13.275a}$$
$$s_0 = \frac{\sqrt{3}-1}{2} \tag{13.275b}$$
$$r_{k+1} = \frac{3}{1 + 2\,(1 - s_k^3)^{1/3}} \tag{13.275c}$$
$$s_{k+1} = \frac{r_{k+1} - 1}{2} \tag{13.275d}$$
$$a_{k+1} = r_{k+1}^2 \, a_k - 3^k \left(r_{k+1}^2 - 1\right) \quad \to \frac{1}{\pi} \tag{13.275e}$$

Nonic (9th order) iteration from [44], as in [hfloat: `src/pi/pi9th.cc`], #FPM=273.7:

$$a_0 = \frac{1}{3} \tag{13.276a}$$
$$r_0 = \frac{\sqrt{3}-1}{2} \tag{13.276b}$$
$$s_0 = (1 - r_0^3)^{1/3} \tag{13.276c}$$
$$t = 1 + 2\,r_k \tag{13.276d}$$
$$u = \left(9\,r_k\,(1 + r_k + r_k^2)\right)^{1/3} \tag{13.276e}$$
$$v = t^2 + t\,u + u^2 \tag{13.276f}$$
$$m = \frac{27\,(1 + s_k + s_k^2)}{v} \tag{13.276g}$$
$$a_{k+1} = m\,a_k + 3^{2\,k-1}\,(1 - m) \quad \to \frac{1}{\pi} \tag{13.276h}$$
$$s_{k+1} = \frac{(1 - r_k)^3}{(t + 2\,u)\,v} \tag{13.276i}$$
$$r_{k+1} = (1 - s_k^3)^{1/3} \tag{13.276j}$$

Summary of operation count vs. algorithms:

```
    #FPM  -  algorithm name in hfloat

 78.424   -   pi_agm_sch()
 98.424   -   pi_agm()
 99.510   -   pi_agm3(fast variant)
108.241   -   pi_agm3(slow variant)
149.324   -   pi_agm(quartic)
155.265   -   pi_agm3(quartic, fast variant)
164.359   -   pi_4th_order(r=16 variant)
169.544   -   pi_agm3(quartic, slow variant)
170.519   -   pi_4th_order(r=4 variant)
182.710   -   pi_cubic_agm()
200.261   -   pi_3rd_order()
255.699   -   pi_2nd_order()
273.763   -   pi_9th_order()
276.221   -   pi_derived_agm()
353.202   -   pi_5th_order()
```

TBD: *notes: discontin.*

TBD: *slow quartic, slow quart.AGM*

TBD: *other quant: num of variables*

More iterations for π

These are not (yet) implemented in hfloat.

A third order algorithm from [45]:

$$v_0 = 2^{-1/8} \tag{13.277a}$$

$$v_1 = 2^{-7/8}\left(\left(1-3^{1/2}\right)2^{-1/2}+3^{1/4}\right) \tag{13.277b}$$

$$w_0 = 1 \tag{13.277c}$$

$$\alpha_0 = 1 \tag{13.277d}$$

$$\beta_0 = 0 \tag{13.277e}$$

$$v_{n+1} = v_n^3 - \left\{v_n^6 + \left[4v_n^2(1-v_n^8)\right]^{1/3}\right\}^{1/2} + v_{n-1} \tag{13.277f}$$

$$w_{n+1} = \frac{2v_n^3 + v_{n+1}\left(3v_{n+1}^2v_n^2-1\right)}{2v_{n+1}^3 - v_n\left(3v_{n+1}^2v_n^2-1\right)}w_n \tag{13.277g}$$

$$\alpha_{n+1} = \left(\frac{2v_{n+1}^3}{v_n}+1\right)\alpha_n \tag{13.277h}$$

$$\beta_{n+1} = \left(\frac{2v_{n+1}^3}{v_n}+1\right)\beta_n + \left(6w_{n+1}v_n - 2v_{n+1}w_n\right)\frac{v_{n+1}^2\alpha_n}{v_n^2} \tag{13.277i}$$

$$\pi_n = \frac{8\cdot 2^{1/8}}{\alpha_n\,\beta_n} \quad \to \pi \tag{13.277j}$$

A second order algorithm from [47]:

$$\begin{aligned}
\alpha_0 &= 1/3 && (13.278a)\\
m_0 &= 2 && (13.278b)\\
m_{n+1} &= \frac{4}{1+\sqrt{(4-m_n)\,(2+m_n)}} && (13.278c)\\
\alpha_{n+1} &= m_n\,\alpha_n + \frac{2^n}{3}(1-m_n) \quad \to \frac{1}{\pi} && (13.278d)
\end{aligned}$$

Another second order algorithm from [47]:

$$\begin{aligned}
\alpha_0 &= 1/3 && (13.279a)\\
s_1 &= 1/3 && (13.279b)\\
(s_n)^2 + (s_n^*)^2 &= 1 && (13.279c)\\
(1+3\,s_{n+1})\,(1+3\,s_n^*) &= 4 && (13.279d)\\
\alpha_{n+1} &= (1+3\,s_{n+1})\alpha_n - 2^n\,s_{n+1} \quad \to \frac{1}{\pi} && (13.279e)
\end{aligned}$$

A fourth order algorithm from [47]:

$$\begin{aligned}
\alpha_0 &= 1/3 && (13.280a)\\
s_1 &= \sqrt{2}-1 && (13.280b)\\
(s_n)^4 + (s_n^*)^4 &= 1 && (13.280c)\\
(1+3\,s_{n+1})\,(1+3\,s_n^*) &= 2 && (13.280d)\\
\alpha_{n+1} &= (1+s_{n+1})^4\alpha_n + \frac{4^{n+1}}{3}\,(1-(1+s_{n+1})^4) \quad \to \frac{1}{\pi} && (13.280e)
\end{aligned}$$

13.13 The binary splitting algorithm for rational series

The straight forward computation of a series for which each term adds a constant amount of precision[13] to a precision of N digits involves the summation of proportional N terms. To get N bits of precision one has to add proportional N terms of the sum, each term involves one (length-N) short division (and one addition). Therefore the total work is proportional N^2, which makes it impossible to compute billions of digits from linearly convergent series even if they are as 'good' as Chudnovsky's famous series for π:

$$\begin{aligned}
\frac{1}{\pi} &= \frac{6541681608}{\sqrt{640320}^3}\sum_{k=0}^{\infty}\left(\frac{13591409}{545140134}+k\right)\left(\frac{(6k)!}{(k!)^3\,(3k)!}\,\frac{(-1)^k}{640320^{3k}}\right) && (13.281)\\
&= \frac{12}{\sqrt{640320}^3}\sum_{k=0}^{\infty}(-1)^k\,\frac{(6k)!}{(k!)^3\,(3k)!}\,\frac{13591409+k\,545140134}{(640320)^{3k}} && (13.282)
\end{aligned}$$

Here is an alternative way to evaluate a sum $\sum_{k=0}^{N-1} a_k$ of rational summands: One looks at the ratios r_k of consecutive terms:

$$r_k := \frac{a_k}{a_{k-1}} \qquad (13.283)$$

(set $a_{-1} := 1$ to avoid a special case for $k = 0$)

That is

$$\sum_{k=0}^{N-1} a_k =: r_0\,(1+r_1\,(1+r_2\,(1+r_3\,(1+\ldots(1+r_{N-1})\ldots)))) \qquad (13.284)$$

[13] e.g. arc-cotangent series with arguments > 1

Now define

$$r_{m,n} := r_m\,(1 + r_{m+1}\,(\ldots(1 + r_n)\ldots)) \quad \text{where} \quad m < n \tag{13.285}$$
$$r_{m,m} := r_m \tag{13.286}$$

then

$$r_{m,n} = \frac{1}{a_{m-1}} \sum_{k=m}^{n} a_k \tag{13.287}$$

and especially

$$r_{0,n} = \sum_{k=0}^{n} a_k \tag{13.288}$$

With

$$r_{m,n} = r_m + r_m \cdot r_{m+1} + r_m \cdot r_{m+1} \cdot r_{m+2} + \ldots \tag{13.289}$$
$$\cdots + r_m \cdot \cdots \cdot r_x + r_m \cdot \cdots \cdot r_x \cdot [r_{x+1} + \cdots + r_{x+1} \cdot \cdots \cdot r_n]$$
$$= r_{m,x} + \prod_{k=m}^{x} r_k \cdot r_{x+1,n} \tag{13.290}$$

The product telescopes, one gets

$$r_{m,n} = r_{m,x} + \frac{a_x}{a_{m-1}} \cdot r_{x+1,n} \tag{13.291}$$

(where $m \le x < n$).

Now we can formulate the binary splitting algorithm by giving a binsplit function `r`:

```
function r(function a, int m, int n)
{
    rational ret;
    if m==n then
    {
        ret := a(m)/a(m-1)
    }
    else
    {
        x := floor( (m+n)/2 )
        ret := r(a,m,x) + a(x) / a(m-1) * r(a,x+1,n)
    }
    print( "r:", m, n, "=", ret )
    return ret
}
```

Here `a(k)` must be a function that returns the `k`-th term of the series we wish to compute, in addition one must have `a(-1)`=1. A trivial example: to compute arctan(1/10) one would use

```
function a(int k)
{
    if k<0 then  return 1
    else         return (-1)^k/((2*k+1)*10^(2*k+1))
}
```

Calling `r(a,0,N)` returns $\sum_{k=0}^{N} a_k$.

In case the programming language used does not provide rational numbers one needs to rewrite formula 13.291 in separate parts for denominator and numerator. With $a_i = \frac{p_i}{q_i}$, $p_{-1} = q_{-1} = 1$ and $r_{m,n} =: \frac{U_{m,n}}{V_{m,n}}$ one gets

$$U_{m,n} = p_{m-1}\,q_x\,U_{m,x}\,V_{x+1,n} + p_x\,q_{m-1}\,U_{x+1,n}\,V_{m,x} \tag{13.292}$$
$$V_{m,n} = p_{m-1}\,q_x\,V_{m,x}\,V_{x+1,n} \tag{13.293}$$

The reason why binary splitting is better than the straight forward way is that the involved work is only $O((\log N)^2\, M(N))$, where $M(N)$ is the complexity of one N-bit multiplication (see [42]). This means that sums of linear but sufficient convergence are again candidates for high precision computations.

In addition, the ratio $r_{0,N-1}$ (i.e. the sum of the first N terms) can be reused if one wants to evaluate the sum to a higher precision than before. To get twice the precision use

$$r_{0,2\,N-1} \quad = \quad r_{0,N-1} + a_{N-1} \cdot r_{N,2\,N-1} \tag{13.294}$$

(this is formula 13.291 with $m = 0, x = N-1, n = 2N-1$). With explicit rational arithmetic:

$$U_{0,2N-1} \quad = \quad q_{N-1}\, U_{0,N-1}\, V_{N,2N-1} + p_{N-1}\, U_{N,2N-1}\, V_{0,N-1} \tag{13.295}$$

$$V_{0,2N-1} \quad = \quad q_{N-1}\, V_{0,N-1}\, V_{N,2N-1} \tag{13.296}$$

Thereby with the appearance of some new computer that can multiply two length $2 \cdot N$ numbers[14] one only needs to combine the two ratios $r_{0,N-1}$ and $r_{N,2N-1}$ that had been precomputed by the last generation of computers. This costs only a few full-size multiplications on your new and expensive supercomputer (instead of several *hundreds* for the iterative schemes), which means that one can improve on prior computations at low cost.

If one wants to stare at zillions of decimal digits of the floating point expansion then one division is also needed which costs not more than 4 multiplications (cf. section 13.3).

Note that this algorithm can trivially be extended (or rather simplified) to infinite products, e.g. matrix products as Bellard's

$$\prod_{k=0}^{\infty} \begin{bmatrix} \frac{2\,(k-\frac{1}{2})\,(k+2)}{27\,(k+\frac{2}{3})\,(k+\frac{4}{3})} & 10 \\ 0 & 1 \end{bmatrix} \quad = \quad \begin{bmatrix} 0 & \pi+6 \\ 0 & 1 \end{bmatrix} \tag{13.297}$$

Cf. [42] and [48].

13.14 The magic sumalt algorithm

The following algorithm is due to Cohen, Villegas and Zagier, see [49].

Pseudo code to compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first n summands. The x_k summands are expected in `x[0,1, ...,n-1]`.

```
function sumalt(x[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2

    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        c := c - b
        s := s + c * x[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }

    return s/d
}
```

With alternating sums the accuracy of the estimate will be $(3+\sqrt{8})^{-n} \approx 5.82^{-n}$.

As an example let us explicitely write down the estimate for the $4 \cdot \arctan(1)$ using the first 8 terms

$$\pi \quad \approx \quad 4 \cdot \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \right) = 3.017\ldots \tag{13.298}$$

[14] assuming one could multiply length-N numbers before

The sumalt-massaged estimate is

$$\begin{aligned}\pi &\approx 4\cdot\left(\frac{665856}{1}-\frac{665728}{3}+\frac{663040}{5}-\frac{641536}{7}+\right. \\ &\qquad \left.+\frac{557056}{9}-\frac{376832}{11}+\frac{163840}{13}-\frac{32768}{15}\right)/665857 \\ &= 4\cdot 3365266048/4284789795 = 3.141592665\ldots\end{aligned} \tag{13.299}$$

and already gives 7 correct digits of π. The linear but impressive growth of the successive sumalt estimates with n, the number of terms used, is illustrated by the following table:

n	sumalt(n)	sumalt(n)−π
1	2.666666666666666666	0.474925986923126571795
2	3.137254901960784313725	0.004337751629008924737
3	3.140740740740740740	0.000851912849052497721
4	3.141635718412148221507	-0.000043064822354983044
5	3.141586546403673968348	0.000006107186119270114
6	3.141593344215659403660	-0.000000690625866165197
7	3.141592564937540122015	0.000000088652253116447
8	3.141592665224315864017	-0.000000011634522625555
9	3.141592652008811951619	0.000000001580981286843
10	3.141592653809731569318	-0.000000000219938330856
11	3.141592653558578755513	0.000000000031214482948
12	3.141592653594296338470	-0.000000000004503100007
13	3.141592653589134580517	0.000000000000658657944
14	3.141592653589890718625	-0.000000000000097480163
15	3.141592653589778664375	0.000000000000014574087
16	3.141592653589795436775	-0.000000000000002198312
17	3.141592653589792904285	0.000000000000000334177
18	3.141592653589793289614	-0.000000000000000051151
19	3.141592653589793230584	0.000000000000000007877
20	3.141592653589793239682	-0.000000000000000001220

Therefore even slowly converging series like

$$\pi = 4\cdot\sum_{k=0}^{\infty}\frac{(-1)^k}{2\,k+1} = 4\cdot\arctan(1) \tag{13.300}$$

$$C = \sum_{k=0}^{\infty}\frac{(-1)^k}{(2\,k+1)^2} = 0.9159655941772190\ldots \tag{13.301}$$

$$\log(2) = \sum_{k=0}^{\infty}\frac{(-1)^k}{k+1} = 0.6931471805599453\ldots \tag{13.302}$$

$$\zeta(s) = \frac{1}{1-2^{1-s}}\sum_{k=1}^{\infty}\frac{(-1)^k}{k^s} \tag{13.303}$$

can be used to compute estimates that are correct up to thousands of digits. The algorithm scales like n^2 if the series terms in `x[]` are small rational values and like $n^3\cdot\log(n)$ if they are full precision (rational or float) values.

All values c_k and b_k occurring in the computation are integers. In fact, the b_k in the computation with n terms are the coefficients of the expanded n-th Chebychev polynomial (of the first kind) with argument $1+2x$:

k	b_k	c_k
0	1	665857
1	128	665856
2	2688	665728
3	21504	663040
4	84480	641536
5	180224	557056
6	212992	376832
7	131072	163840
8	32768	32768

$$\begin{aligned} T_8(1+2x) &= 1+128x+2688x^2+21504x^3+84480x^4+ \\ &\quad +180224x^5+212992x^6+131072x^7+32768x^8 = T_{16}(\sqrt{1+x}) \end{aligned} \tag{13.304}$$

$$\begin{aligned} T_{16}(x) &= 1-128x^2+2688x^4-21504x^6+84480x^8- \\ &\quad -180224x^{10}+212992x^{12}-131072x^{14}+32768x^{16} \end{aligned} \tag{13.305}$$

Now observe that one has always $c_n = b_n = 2^{2n-1}$ in a length-n sumalt computation. Obviously, 'going backwards' avoids the computation of $(3+\sqrt{8})^n$:

```
function sumalt(x[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * x[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }

    return s/c
}
```

The b_k and c_k occurring in a length-n sumalt computation can be given explicitely as

$$b_k = \frac{n}{n+k}\binom{n+i}{2k}2^{2k} \tag{13.306}$$

$$c_k = d_n - \sum_{i=0}^{k} c_k = \sum_{i=k+1}^{n} \frac{n}{n+i}\binom{n+i}{2i}2^{2i} \tag{13.307}$$

To compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first n partial sums use the following pseudo code (the partial sums $p_k = \sum_{j=0}^{k} x_j$ are expected in `p[0,1,...,n-1]`):

```
function sumalt_partial(p[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2

    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }

    return s/d
}
```

The backward variant is:

```
function sumalt_partial(p[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + b * p[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

Cf. [hfloat: src/hf/sumalt.cc]

For series of already geometrical rate of convergence [49] gives

```
function sumalt_partial(p[], n, e)
{
    d := ( 2*e + 1 + 2*sqrt(e*(e+1)) )^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1)) * e
    }
    return s/d
}
```

for series where $|a_k/a_{k+1}| \approx e$. Convergence is improved from $\sim e^{-n}$ to $\sim (2e+1+2\sqrt{e(e+1)})^{-n} \approx (4e+2)^{-n}$. This algorithm specializes to the original one for $e=1$.

13.15 Chebyshev polynomials *

The Chebychev polynomials of the first and second kind can be defined by the functions

$$T_n(x) = \cos(n \arccos(x)) \tag{13.308}$$

$$U_n(x) = \frac{\sin((n+1)\arccos(x))}{\sqrt{1-x^2}} \tag{13.309}$$

For integral n both of them are polynomials. We have

$$T_{-n}(x) = T_n(x) \tag{13.310}$$

$$T_{-1}(x) = x \tag{13.311}$$

$$T_0(x) = 1 \tag{13.312}$$

$$T_1(x) = x \tag{13.313}$$

$$T_2(x) = 2x^2 - 1 \tag{13.314}$$

$$T_3(x) = 4x^3 - 3x \tag{13.315}$$

$$T_4(x) = 8x^4 - 8x^2 + 1 \tag{13.316}$$

$$T_5(x) = 16x^5 - 20x^3 + 5x \tag{13.317}$$

$$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1 \tag{13.318}$$

$$T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x \tag{13.319}$$

$$T_n(x) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{(-1)^k (n-k-1)! (2x)^{n-2k}}{k! (n-2k)!} \tag{13.320}$$

$$= \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2\,k} (2x)^{n-2k} (x^2-1)^k \tag{13.321}$$

and

$$U_{-n}(x) = -U_{n-2}(x) \tag{13.322}$$
$$U_{-1}(x) = 0 \tag{13.323}$$
$$U_0(x) = 1 \tag{13.324}$$
$$U_1(x) = 2\,x \tag{13.325}$$
$$U_2(x) = 4\,x^2 - 1 \tag{13.326}$$
$$U_3(x) = 8\,x^3 - 4\,x \tag{13.327}$$
$$U_4(x) = 16\,x^4 - 12\,x^3 + 1 \tag{13.328}$$
$$U_5(x) = 32\,x^5 - 32\,x^3 + 6\,x \tag{13.329}$$
$$U_6(x) = 64\,x^6 - 80\,x^4 + 24\,x^2 - 1 \tag{13.330}$$
$$U_7(x) = 128\,x^7 - 192\,x^5 + 80\,x^3 - 8\,x \tag{13.331}$$

$$U_n(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{(-1)^k (n-k)! (2x)^{n-2k}}{k! (n-2k)!} \tag{13.332}$$

$$= \sum_{k=0}^{\lfloor n/2+1 \rfloor} \binom{n+1}{2\,k+1} x^{n-2k} (x^2-1)^k \tag{13.333}$$

Both obey the same recurrence (omitting argument x)

$$T_n = 2\,x\,T_{n-1} - T_{n-2} \tag{13.334}$$
$$U_n = 2\,x\,U_{n-1} - U_{n-2} \tag{13.335}$$

Their generating functions are

$$\frac{1 - x\,t}{t^2 - 2\,x\,t + 1} = \sum_{n=0}^{\infty} t^n\,T_n(x) \tag{13.336}$$

$$\frac{1}{t^2 - 2\,x\,t + 1} = \sum_{n=0}^{\infty} t^n\,U_n(x) \tag{13.337}$$

Composition is multiplication of indices:

$$T_n(T_m(x)) = T_{n\,m}(x) \tag{13.338}$$

For example,

$$T_{2n}(x) = T_2(T_n(x)) = 2\,T_n^2(x) - 1 \tag{13.339}$$

Some relations between T and U are

$$T_n = U_n - x\,U_{n-1} = x\,U_{n-1} - U_{n-2} = \frac{1}{2}(U_n - U_{n-2}) \tag{13.340}$$
$$T_{n+1} = x\,T_n - (1-x^2)\,U_{n-1} \tag{13.341}$$
$$U_{2n-1} = 2\,T_n\,U_{n-1} \tag{13.342}$$

and

$$T_{n+m} + T_{n-m} = 2\,T_n\,T_m \quad \text{where} \quad n \geq m \tag{13.343}$$
$$T_{n+m} - T_{n-m} = 2\,(x^2+1)\,U_{n-1}\,U_{m-1} \quad \text{where} \quad n \geq m \tag{13.344}$$
$$U_{n+m-1} + U_{n-m-1} = 2\,U_{n-1}\,T_m \quad \text{where} \quad n > m \tag{13.345}$$
$$U_{n+m-1} - U_{n-m-1} = 2\,T_n\,U_{m-1} \quad \text{where} \quad n > m \tag{13.346}$$

The key to the fast computation of both T_n and U_n via powering algorithms is given by

$$\begin{bmatrix} T_0 & T_1 \\ U_0 & U_1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 2\,x \end{bmatrix}^k = \begin{bmatrix} T_k & T_{k+1} \\ U_k & U_{k+1} \end{bmatrix} \tag{13.347}$$

It is instructive to do the case $k = 1$ by hand: The first column of the second matrix moves the second entries of the first matrix to the left, the second column does the recursion. The generalization for the general case is straightforward. For example, a recursion $a_n = \lambda_1\,a_{n-1} + \lambda_2\,a_{n-2} + \lambda_3\,a_{n-3}$ leads to

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \begin{bmatrix} 0 & 0 & \lambda_3 \\ 1 & 0 & \lambda_2 \\ 0 & 1 & \lambda_1 \end{bmatrix}^k = \begin{bmatrix} A_k & A_{k+1} & A_{k+2} \\ B_k & B_{k+1} & B_{k+2} \\ C_k & C_{k+1} & C_{k+2} \end{bmatrix} \tag{13.348}$$

See also [27] and [63].

13.16 Continued fractions *

Set

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cdots}}}} \tag{13.349}$$

For $k > 0$ let $\frac{p_k}{q_k}$ be the value of the above fraction if b_{k+1} is set to zero (set $\frac{p_{-1}}{q_{-1}} := \frac{1}{0}$ and $\frac{p_0}{q_0} := \frac{a_0}{1}$).

Then

$$p_k = a_k\,p_{k-1} + b_k\,p_{k-2} \tag{13.350a}$$
$$q_k = a_k\,q_{k-1} + b_k\,q_{k-2} \tag{13.350b}$$

(Simple continued fractions are those with $b_k = 1\ \forall k$).

Pseudo code for a procedure that computes the $p_k, q_k \quad k = -1 \ldots n$ of a continued fraction :

```
procedure ratios_from_contfrac(a[0..n], b[0..n], n, p[-1..n], q[-1..n])
{
    p[-1] := 1
    q[-1] := 0
    p[0] := a[0]
    q[0] := 1
    for k:=1 to n
    {
        p[k] := a[k] * p[k-1] + b[k] * p[k-2]
        q[k] := a[k] * q[k-1] + b[k] * q[k-2]
    }
}
```

Pseudo code for a procedure that fills the first `n` terms of the simple continued fraction of (the floating point number) `x` into the array `cf[]`:

```
procedure continued_fraction(x, n, cf[0..n-1])
{
    for k:=0 to n-1
    {
        xi := floor(x)
        cf[k] := xi
        x := 1 / (x-xi)
    }
}
```

Pseudo code for a function that computes the numerical value of a number `x` from (the leading `n` terms of) its simple continued fraction representation:

```
function number_from_contfrac(cf[0..n-1], n)
{
    x := cf[n-1]
    for k:=n-2 to 0 step -1
    {
        x := 1/x + cf[k]
    }
    return x
}
```

(cf. [24], [23], [13], [17]).

It is possible to rewrite a continued fraction with positive a_k, b_k as an alternating sum

$$\begin{aligned} x &= a_0 + \sum_{k=1}^{\infty} (-1)^{k+1} s_k \\ &= a_0 + \frac{b_1}{q_0\, q_1} - \frac{b_1\, b_2}{q_1\, q_2} + \frac{b_1\, b_2\, b_3}{q_2\, q_3} \pm \ldots + (-1)^{k+1} \frac{\prod_{i=1}^{k} b_i}{q_k\, q_{k+1}} \pm \ldots \end{aligned} \tag{13.351}$$

where $q_{-1} = 0$, $q_1 = 1$ and $q_n = a_n q_{n-1} + b_n q_{n-2}$, cf. [49].

Continued fractions with $b_k = 1 \, \forall k$

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}} \tag{13.352}$$

are called *simple continued fractions*

See [24] for an easy introduction and also [39].

13.17 Some hypergeometric identities *

13.17.1 Definition

Let $x^{\bar{k}} := x\,(x+1)\,(x+2)\,\ldots\,(x+k)$ then

$$F\left(\left.\begin{matrix} a, b \\ c \end{matrix}\right| z\right) := \sum_{k=0}^{\infty} \frac{a^{\bar{k}}\, b^{\bar{k}}}{c^{\bar{k}}} \frac{z^k}{k!} \tag{13.353}$$

Often the so called Pochhammer symbol $(x)_k$ is used instead of the "rising factorial power" notation $x^{\bar{k}}$ used here. The $k!$ in the denominators is there for historical reasons. You might want to have an additional 1 in the lower entries in mind:

$$F\left(\left.\begin{matrix} 2, 2 \\ 1 \end{matrix}\right| z\right) = "F\left(\left.\begin{matrix} 2, 2 \\ 1, 1 \end{matrix}\right| z\right)" \tag{13.354}$$

is a sum of perfect squares if z is a square.

Obviously,

$$F\left(\begin{matrix}a,b\\c\end{matrix}\middle|\,z\right) = 1+\frac{a}{1}\frac{b}{c}z\left(1+\frac{a+1}{2}\frac{b+1}{c+1}z\left(1+\frac{a+2}{3}\frac{b+2}{c+2}z\,(1+\ldots)\right)\right) \tag{13.355}$$

so by formula 13.355 hypergeometric functions with rational arguments can be computed with the binary splitting method described in section 13.13.

Hypergeometric functions can have any number of entries:

$$F\left(\begin{matrix}a_1,\,\ldots,\,a_m\\b_1,\,\ldots,\,b_n\end{matrix}\middle|\,z\right) = \sum_{k=0}^{\infty}\frac{a_1^{\bar{k}}\ldots a_m^{\bar{k}}}{b_1^{\bar{k}}\ldots b_n^{\bar{k}}}\frac{z^k}{k!} \tag{13.356}$$

Negative integer entries in the upper row lead to polynomials:

$$F\left(\begin{matrix}-3,\,3\\1\end{matrix}\middle|\,z\right) = 1-9\,z+18\,z^2-10\,z^3 \tag{13.357}$$

Negative integer entries in the lower row are verboten.

13.17.2 Transformations

As obvious from the definition, entries can be swapped (capitalized symbols for readability):

$$F\left(\begin{matrix}A,\,B,\,c\\e,\,f,\,g\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}B,\,A,\,c\\e,\,f,\,g\end{matrix}\middle|\,z\right) \tag{13.358}$$

Usually one writes the entries in ascending order. Identical elements in the lower and upper row can be canceled:

$$F\left(\begin{matrix}a,\,b,\,C\\e,\,f,\,C\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}a,\,b\\e,\,f\end{matrix}\middle|\,z\right) \tag{13.359}$$

These are true for any number of elements, the following transformations are only valid for the given structure.

Pfaff's reflection law and Euler's identity

(the latter is obtained by applying the reflection on both upper entries):

$$\frac{1}{(1-z)^a}F\left(\begin{matrix}a,\,b\\c\end{matrix}\middle|\,\frac{-z}{1-z}\right) = F\left(\begin{matrix}a,\,c-b\\c\end{matrix}\middle|\,z\right) \tag{13.360}$$

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\middle|\,z\right) = \frac{1}{(1-z)^a}F\left(\begin{matrix}a,\,c-b\\c\end{matrix}\middle|\,\frac{-z}{1-z}\right) \tag{13.361}$$

$$= \frac{1}{(1-z)^b}F\left(\begin{matrix}c-a,\,b\\c\end{matrix}\middle|\,\frac{-z}{1-z}\right) \tag{13.362}$$

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\middle|\,z\right) = (1-z)^{(c-a-b)}F\left(\begin{matrix}c-a,\,c-b\\c\end{matrix}\middle|\,z\right) \tag{13.363}$$

A transformation by C.F.Gauss

$$F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\middle|\,4z(1-z)\right) \quad \text{where} \quad |z|<\frac{1}{2} \tag{13.364}$$

$$F\left(\begin{matrix}a,\,b\\a+b+\frac{1}{2}\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}2a,\,2b\\a+b+\frac{1}{2}\end{matrix}\middle|\,\frac{1-\sqrt{1-z}}{2}\right) \tag{13.365}$$

Note that the right hand side of relation 13.364 does not change if z is replaced by $1-z$, so it seems that

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) = F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,1-z\right) \tag{13.366}$$

However, the relation is only true for terminating series, i.e. polynomials. Rewriting relation 13.364 for the argument $\frac{1-z}{2}$ we get

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-z}{2}\right) = F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,1-z^2\right) \tag{13.367}$$

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-\sqrt{1-z^2}}{2}\right) = F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z^2\right) \tag{13.368}$$

Whipple's identity

$$F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2},\,1-a-b-c\\ 1+a-b,\,1+a-c\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) = (1-z)^a\,F\left(\begin{matrix}a,\,b,\,c\\ 1+a-b,\,1+a-c\end{matrix}\Big|\,z\right) \tag{13.369}$$

Specializing 13.369 $c=(a+1)/2$ (note the symmetry between b an c so specializing for $c=(b+1)/2$ produces the identical relation)

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,z\right) = \frac{1}{(1-z)^a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{13.370}$$

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) = \left(\frac{2\left(1-\sqrt{1-z}\right)}{z}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{\left(1-\sqrt{1-z}\right)^2}{z}\right) \tag{13.371}$$

With $c:=a-b$ in 13.370 one gets:

$$F\left(\begin{matrix}a,\,a-c\\ 1+c\end{matrix}\Big|\,z\right) = \frac{1}{(1-z)^a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}-\frac{1}{2}a+c\\ 1+c\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{13.372}$$

If $b=a$ in relation 13.370, then

$$F\left(\begin{matrix}a,\,a\\ 1\end{matrix}\Big|\,z\right) = \frac{1}{(1-z)^a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}-\frac{1}{2}a\\ 1\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{13.373}$$

Observing that the hypergeometric function on the right side does not change when replacing a by $1-a$ one finds Euler's transformation for $a=b, c=1$.

Similarly as for the relations by Gauss, from 13.370 and 13.371:

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,-\frac{1-z}{1+z}\right) = \left(\frac{1+z}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,1-z^2\right) \tag{13.374}$$

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,-\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) = \left(\frac{1+\sqrt{1-z^2}}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,z^2\right) \tag{13.375}$$

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,1-z^2\right) = \left(\frac{2}{1+z}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{1-z}{1+z}\right) \tag{13.376}$$

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z^2\right) = \left(\frac{2}{1+\sqrt{1-z^2}}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) \tag{13.377}$$

The following is due to Ramanujan. Let

$$z' := \left(1-\left(\frac{1-z}{1+2z}\right)^3\right)^{1/3} \tag{13.378}$$

then

$$z = \frac{1-(1-z'^3)^{1/3}}{1+2\,(1-z'^3)^{1/3}} \tag{13.379}$$

$$F\left(\begin{matrix}\frac{1}{3},\,\frac{2}{3}\\ 1\end{matrix}\Big|\, z'^3\right) = (1+2\,z)\,F\left(\begin{matrix}\frac{1}{3},\,\frac{2}{3}\\ 1\end{matrix}\Big|\, z^3\right) \tag{13.380}$$

Clausen's product formulas

$$\left[F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right)\right]^2 = F\left(\begin{matrix}2a,\,a+b,\,2b\\ a+b+\frac{1}{2},\,2a+2b\end{matrix}\Big|\, z\right) \tag{13.381}$$

$$F\left(\begin{matrix}\frac{1}{4}+a,\,\frac{1}{4}+b\\ 1+a+b\end{matrix}\Big|\, z\right)\,F\left(\begin{matrix}\frac{1}{4}-a,\,\frac{1}{4}-b\\ 1-a-b\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}+a-b,\,\frac{1}{2}-a+b\\ 1+a+b,\,1-a-b\end{matrix}\Big|\, z\right) \tag{13.382}$$

If in 13.381 $a = b+\frac{1}{2}$ then (two entries on the right hand side cancel)

$$\left[F\left(\begin{matrix}b+\frac{1}{2},\,b\\ 2b+1\end{matrix}\Big|\, z\right)\right]^2 = F\left(\begin{matrix}2b+\frac{1}{2},\,2b\\ 4b+1\end{matrix}\Big|\, z\right) \tag{13.383}$$

and the right hand side again matches the structure on the left. The corresponding function can be identified (see. [8]) as $G_b(z) = \left(\frac{1+\sqrt{1-z}}{2}\right)^{-2b}$. One has $G_{n\,m}(z) = (G_n(z))^m$.

Specializing relation 13.382 for $b = -a$ we get

$$\left[F\left(\begin{matrix}\frac{1}{4}+a,\,\frac{1}{4}-a\\ 1\end{matrix}\Big|\, z\right)\right]^2 = F\left(\begin{matrix}\frac{1}{2}+2a,\,\frac{1}{2},\,\frac{1}{2}-2a\\ 1,\,1\end{matrix}\Big|\, z\right) \tag{13.384}$$

A complementary relation

A relation involving arguments z and $1-z$ (given in [22], p.291) is

$$F\left(\begin{matrix}a,\,b\\ c\end{matrix}\Big|\, z\right) = \alpha\,F\left(\begin{matrix}a,\,b\\ a+b-c+1\end{matrix}\Big|\, z\right) + \beta\,(1-z)^{c-a-b}\,F\left(\begin{matrix}c-a,\,c-b\\ c-a-b+1\end{matrix}\Big|\, 1-z\right) \tag{13.385}$$

where
$\alpha = \frac{1}{\gamma}\,\Gamma(a)\,\Gamma(b)\,\Gamma(c)\,\Gamma(c-a-b)$
$\beta = \frac{1}{\gamma}\,\Gamma(c)\,\Gamma(c-a)\,\Gamma(c-b)\,\Gamma(a+b-c)$ and
$\gamma = \Gamma(a)\,\Gamma(b)\,\Gamma(c-a)\,\Gamma(c-b)$.

An integral representation

For $z \in \mathbb{C} - [1,\infty)$ one has

$$F\left(\begin{matrix}a,\,b\\ c\end{matrix}\Big|\, z\right) = \frac{\Gamma(c)}{\Gamma(b)\,\Gamma(c-b)}\int_0^1 \frac{t^{b-1}\,(1-t)^{c-b-1}}{(1-t\,z)^a}dt \tag{13.386}$$

Differential equation

$f(z) = F\left(\begin{matrix}a,\,b\\ c\end{matrix}\Big|\, z\right)$ is a solution of the differential equation

$$z\,(1-z)\,\frac{d^2\,f}{dz^2} + [c-(1+a+b)\,z]\,\frac{d\,f}{dz} - a\,b\,f = 0 \tag{13.387}$$

A summation formula

See [13].

$$F\left(\begin{matrix}a,\ b\\ c\end{matrix}\middle|\ 1\right) \quad = \quad \frac{\Gamma(c)\,\Gamma(c-a-b)}{\Gamma(c-a)\,\Gamma(c-b)} \quad \text{if } \Re(c-a-b)>0 \quad \text{or } b\in\mathbb{N}, b<0 \tag{13.388}$$

See [27] chapter 15, [63] and [39].

13.17.3 Examples: elementary functions

As a warmup the 'well known' functions like $\exp, \log, \sin, \ldots$ are expressed as hypergeometric functions. In some cases a transform is applied to give an alternative series.

Simple series, exp and log

$$\frac{1}{(1-z)^a} \quad = \quad F\left(\begin{matrix}a\\ \end{matrix}\middle|\ z\right) \ = \ \sum_{k=0}^{\infty}\binom{a+k-1}{k} z^k \tag{13.389}$$

$$(1+z)^a \quad = \quad F\left(\begin{matrix}-a\\ \end{matrix}\middle|\ -z\right) \ = \ \sum_{k=0}^{a}\binom{a}{k} z^k \tag{13.390}$$

$$\exp(z) \quad = \quad F\left(\begin{matrix}\\ \end{matrix}\middle|\ z\right) \ = \ \sum_{k=0}^{\infty}\frac{z^k}{k!} \tag{13.391}$$

$$\log(1+z) \quad = \quad F\left(\begin{matrix}1,\ 1\\ 2\end{matrix}\middle|\ -z\right) \ = \ \sum_{k=0}^{\infty}\frac{(-1)^k\, z^{k+1}}{k+1} \tag{13.392}$$

$$\operatorname{erf}(z) \quad = \quad \frac{2\,z}{\sqrt{\pi}}\, F\left(\begin{matrix}\frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\ -z^2\right) \tag{13.393}$$

Trigonometric and hyperbolic functions

$$\sin(z) \quad = \quad z\, F\left(\begin{matrix}\\ \frac{3}{2}\end{matrix}\middle|\ \frac{-z^2}{4}\right) \ = \ \sum_{k=0}^{\infty}\frac{(-1)^k\, z^{2k+1}}{(2k+1)!} \tag{13.394}$$

$$\sinh(z) \quad = \quad z\, F\left(\begin{matrix}\\ \frac{3}{2}\end{matrix}\middle|\ \frac{z^2}{4}\right) \ = \ \sum_{k=0}^{\infty}\frac{z^{2k+1}}{(2k+1)!} \tag{13.395}$$

$$\quad = \quad z\, F\left(\begin{matrix}1,\ 1\\ \frac{3}{2}\end{matrix}\middle|\ \frac{1-\sqrt{1-z^2}}{2}\right) \quad \text{by 13.365} \tag{13.396}$$

$$\cos(z) \quad = \quad z\, F\left(\begin{matrix}\\ \frac{1}{2}\end{matrix}\middle|\ \frac{-z^2}{4}\right) \ = \ \sum_{k=0}^{\infty}\frac{(-1)^k\, z^{2k}}{(2k)!} \tag{13.397}$$

$$\cosh(z) \quad = \quad z\, F\left(\begin{matrix}\\ \frac{1}{2}\end{matrix}\middle|\ \frac{z^2}{4}\right) \ = \ \sum_{k=0}^{\infty}\frac{z^{2k}}{(2k)!} \tag{13.398}$$

$$\sin(a\,z) = F\left(\begin{matrix}\frac{1+a}{2},\ \frac{1-a}{2}\\ \frac{3}{2}\end{matrix}\middle|\ (\sin(z))^2\right) \tag{13.399}$$

$$\cos(a\,z) = F\left(\begin{matrix}+\frac{a}{2},\ -\frac{a}{2}\\ \frac{1}{2}\end{matrix}\middle|\ (\sin(z))^2\right) \tag{13.400}$$

$$\sinh(a\,z) = F\left(\begin{matrix}\frac{1+a}{2},\ \frac{1-a}{2}\\ \frac{3}{2}\end{matrix}\middle|\ (\sinh(z))^2\right) \tag{13.401}$$

$$\cosh(a\,z) = F\left(\begin{matrix}+\frac{a}{2},\ -\frac{a}{2}\\ \frac{1}{2}\end{matrix}\middle|\ (\sinh(z))^2\right) \tag{13.402}$$

Inverse trigonometric and hyperbolic functions

$$\arcsin(z) = z\,F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\ z^2\right) \tag{13.403}$$

$$(\arcsin(z))^2 = z\,F\left(\begin{matrix}1,\ 1,\ 1\\ \frac{3}{2},\ 2\end{matrix}\middle|\ z^2\right) \quad \text{by 13.381} \tag{13.404}$$

$$\operatorname{arcsinh}(z) = \log(z+\sqrt{1+z^2}) = z\,F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\ -z^2\right) \tag{13.405}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\ 1\\ \frac{3}{2}\end{matrix}\middle|\ \frac{z^2}{1+z^2}\right) \quad \text{by 13.361} \tag{13.406}$$

$$= z\,F\left(\begin{matrix}1,\ 1\\ \frac{3}{2}\end{matrix}\middle|\ \frac{1-\sqrt{1+z^2}}{2}\right) \quad \text{by 13.365} \tag{13.407}$$

$$\arctan(z) = z\,F\left(\begin{matrix}\frac{1}{2},\ 1\\ \frac{3}{2}\end{matrix}\middle|\ -z^2\right) = \sum_{k=0}^{\infty}\frac{(-1)^k\,z^{2k+1}}{2k+1} \tag{13.408}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\ \frac{z^2}{1+z^2}\right) \quad \text{by 13.361} \tag{13.409}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\ 1\\ \frac{3}{2}\end{matrix}\middle|\ \frac{z^2}{1+z^2}\right) \quad \text{by 13.361} \tag{13.410}$$

$$\operatorname{arctanh}(z) = z\,F\left(\begin{matrix}\frac{1}{2},\ 1\\ \frac{3}{2}\end{matrix}\middle|\ z^2\right) = \sum_{k=0}^{\infty}\frac{z^{2k+1}}{2k+1} \tag{13.411}$$

$$\operatorname{arccot}(z) = \frac{1}{z}\,F\left(\begin{matrix}\frac{1}{2},\ 1\\ \frac{3}{2}\end{matrix}\middle|\ -\frac{1}{z^2}\right) \tag{13.412}$$

$$= \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ \frac{3}{2}\end{matrix}\middle|\ \frac{1}{1+z^2}\right) \tag{13.413}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\ 1\\ \frac{3}{2}\end{matrix}\middle|\ \frac{1}{1+z^2}\right) \tag{13.414}$$

13.17.4 Elliptic K and E

In order to avoid the factor $\frac{\pi}{2}$ we let $\tilde{K} := \frac{2\,K}{\pi}$, $\tilde{E} := \frac{2\,E}{\pi}$.

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{13.415}$$

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{13.416}$$

We further set

$$\tilde{N}(k) = F\left(\begin{matrix}-\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{13.417}$$

Most of the following relations can be written in several ways by one of the identities

$$k' = \sqrt{1-k^2} \tag{13.418}$$

$$\frac{-4k^2}{(1-k^2)^2} = -\left(\frac{2k}{k'^2}\right)^2 = 1-\left(\frac{1+k^2}{1-k^2}\right)^2 \tag{13.419}$$

$$-\frac{k^2}{1-k^2} = -\left(\frac{k}{k'}\right)^2 = -\frac{1-k'^2}{k'^2} \tag{13.420}$$

$$-\left(\frac{k}{1-\sqrt{1-k^2}}\right)^2 = -\frac{1+k'}{1-k'} = -\frac{1-k'^2}{(1-k')^2} = -\frac{(1+k')^2}{1-k'^2} \tag{13.421}$$

$$\frac{2}{1+k'} = 1+\frac{1-k'}{1+k'} \tag{13.422}$$

Elliptic K

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{13.423}$$

$$= \frac{1}{k'}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,-\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 13.361} \tag{13.424}$$

$$\tilde{K}(k) = \frac{1}{1-k'}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,-\frac{1+k'}{1-k'}\right) \tag{13.425}$$

From relation 13.229 we get:

$$\tilde{K}(k) = \frac{2}{1+k'}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,\left(\frac{1-k'}{1+k'}\right)^2\right) \tag{13.426}$$

$$= \left(1+\frac{1-k'}{1+k'}\right)\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\,\middle|\,\left(\frac{1-k'}{1+k'}\right)^2\right) \tag{13.427}$$

With $z(k) := \frac{1-k'}{1+k'}$ (and $(z(k))^2 =: z^2(k)$) the last relation can be written as:

$$\tilde{K}(k) = (1+z(k))\;\tilde{K}\left(z^2(k)\right) \tag{13.428}$$

Elliptic E

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\middle| k^2\right) \tag{13.429}$$

$$\tilde{E}(k) = k'\,F\left(\begin{matrix}-\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\middle| -\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 13.361} \tag{13.430}$$

The following relation resembles relation 13.426:

$$\tilde{E}(k) = \frac{\left(1+\sqrt{1-k^2}\right)}{2}\,F\left(\begin{matrix}-\frac{1}{2},\ -\frac{1}{2}\\ 1\end{matrix}\middle| \left(\frac{1-\sqrt{1-k^2}}{1+\sqrt{1-k^2}}\right)^2\right) \tag{13.431}$$

$$= \frac{1+k'}{2}\,F\left(\begin{matrix}-\frac{1}{2},\ -\frac{1}{2}\\ 1\end{matrix}\middle| \left(\frac{1-k'}{1+k'}\right)^2\right) \tag{13.432}$$

$$= (1+z(k))^{-1}\ \tilde{N}\left(z^2(k)\right) \tag{13.433}$$

$$\tilde{K}(k) = \frac{1}{\sqrt{1-k^2}}\,F\left(\begin{matrix}\frac{1}{4},\ \frac{1}{4}\\ 1\end{matrix}\middle| \frac{-4k^2}{(1-k^2)^2}\right) \tag{13.434}$$

Euler's transform on 13.434 gives:

$$\tilde{K}(k) = \frac{1}{k'}\,\frac{1+k^2}{1-k^2}\,F\left(\begin{matrix}\frac{3}{4},\ \frac{3}{4}\\ 1\end{matrix}\middle| -\left(\frac{2k}{k'^2}\right)^2\right) \tag{13.435}$$

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{4},\ \frac{1}{4}\\ 1\end{matrix}\middle| (2\,k\,k')^2\right) \qquad \text{by 13.364} \tag{13.436}$$

$$\tilde{N}(k) = \sqrt{1-k^2}\,F\left(\begin{matrix}-\frac{1}{4},\ \frac{3}{4}\\ 1\end{matrix}\middle| \frac{-4\,k^2}{\left(1-k^2\right)^2}\right) \qquad \text{by 13.373} \tag{13.437}$$

$$= k'\,F\left(\begin{matrix}-\frac{1}{4},\ \frac{3}{4}\\ 1\end{matrix}\middle| -\left(\frac{2k}{k'^2}\right)^2\right) \tag{13.438}$$

$$= \sqrt{1+k^2}\,F\left(\begin{matrix}-\frac{1}{4},\ \frac{1}{4}\\ 1\end{matrix}\middle| \left(\frac{2k}{k^2+1}\right)^2\right) \qquad \text{by 13.361} \tag{13.439}$$

Appendix A

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
$\stackrel{m}{=}$	emphasize that the sequences to the left and right are all of length m
$\mathcal{F}[a] \quad (= c)$	(discrete) Fourier transform (FT) of a, $c_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{x\,k}$ where $z = e^{\pm 2\,\pi\,i/n}$
$\mathcal{F}^{-1}[a]$	inverse (discrete) Fourier transform (IFT) of a, $\mathcal{F}^{-1}[a]_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{-x\,k}$
$\mathcal{S}^k a$	a sequence c with elements $c_x := a_x \, e^{\pm\, k\, 2\,\pi\, i\, x/n}$
$\mathcal{H}[a]$	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: $a_A := a - \overline{a}$
$\mathcal{Z}[a]$	discrete z-transform (ZT) of a
$\mathcal{W}_v[a]$	discrete weighted transform of a, weight (sequence) v
$\mathcal{W}_v^{-1}[a]$	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 sequences
$a \circledast_{-} b$	negacyclic (or skew circular) convolution of sequences
$a \circledast_{\{v\}} b$	weighted convolution of sequences with weight v
$a \circledast_{\hat{}} b$	dyadic convolution of sequences
$a * b$	cyclic correlation TBD: *replace ugly symbol*
$a *_{ac} b$	acyclic (linear) correlation
$n \backslash N$	n divides N
$n \perp m$	$\gcd(n, m) = 1$
$a^{(j\%m)}$	sequence consisting of the elements of a with indices k: $k \equiv j \mod m$ e.g.
$a^{(even)}$, $a^{(odd)}$	$a^{(0\%2)}$, $a^{(1\%2)}$
$a^{(j/m)}$	sequence consisting of the elements of a with indices k: $j \cdot n/m < k < (j+1) \cdot n/m$ e.g.

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. [4]) or description by words (cf. [11] or [19]) 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 (avoiding the 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 access (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, xor, 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` $= 3.14159265\ldots$

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, XOR`

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`.

Bibliography

— Textbooks & Thesis —

[1] H.S.Wilf: **Algorithms and Complexity**, internet edition, 1994,
online at `ftp://ftp.cis.upenn.edu/pub/wilf/AlgComp.ps.Z`

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

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

[4] R.Tolimieri, M.An, C.Lu: **Algorithms for Discrete Fourier Transform and Convolution**, Springer 1997 (second edition)

[5] Hari Krishna Grag: **Digital Signal Processing Algorithms**, CRC Press, 1998

[6] Charles Van Loan: **Computational Frameworks for the Fast Fourier Transform**, SIAM, 1992

[7] Louis V.King: **On the Direct Numerical Calculation of Elliptic Functions and Integrals**, Cambridge University Press 1924

[8] J.M.Borwein, P.B.Borwein: **Pi and the AGM**, Wiley 1987

[9] E.Schröder: **On Infinitely Many Algorithms for Solving Equations** (translation by G.W.Stewart of: **Ueber unendlich viele Algorithmen zur Auflösung der Gleichungen**, which appeared 1870 in the 'Mathematische Annalen')
online at `ftp://thales.cs.umd.edu/pub/reports/`

[10] Householder: **The Numerical Treatment of a Single Nonlinear Equation**, McGraw-Hill 1970

[11] 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/`

[12] D.E.Knuth: **The Art of Computer Programming**, pre-fascicles for Volume 4,
online at `http://www-cs-staff.stanford.edu/~knuth/`

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

[14] J.H.van Lint, R.M.Wilson: **A Course in Combinatorics**, Cambridge University Press, 1992

[15] H.-D.Ebbinghaus, H.Hermes, F.Hirzebruch, M.Koecher, K.Mainzer, J.Neukirch, A.Prestel, R.Remmert: **Zahlen**, second edition, (english translation: **Numbers**) Springer, 1988

[16] W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery: **Numerical Recipes in C**, Cambridge University Press, 1988, 2nd Edition 1992
online at `http://nr.harvard.edu/nr/`

[17] 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)

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

[19] H.Cohen: **A Course in Computational Algebraic Number Theory**, Springer Verlag, Berlin Heidelberg, 1993

[20] William Stein: **An Explicit Approach to Elementary Number Theory**, Harvard University, Fall 2001

[21] Thomas H.Corman, Charles E.Leiserson, Ronald L.Rivest: **Introduction to Algorithms**, MIT Press, 1990 (twenty-first printing, 1998)

[22] E.T.Whittaker, G.N.Whatson: **A Course of Modern Analysis**, Cambridge University Press, Fourth Edition, 1927, reprinted 1990

[23] L.Lorentzen and H.Waadeland: **Continued Fractions and Applications**, North-Holland 1992

[24] C.D.Olds: **Continued Fractions**, The Mathematical Association of America, 1963

[25] Robert Sedgewick: **Algorithms in C**, Addison-Wesley, 1990

[26] J.Brillhart, D.H.Lehmer, J.L.Selfridge, B.Tuckerman, S.S.Wagstaff Jr.: **Factorizations of** $b^n \pm 1$ $b = 2, 3, 5, 6, 10, 11$ **up to high powers**, Contemporary Mathematics, Volume 22, Second Edition, American Mathematical Society, 1988
online at `http://www.ams.org/`

[27] Milton Abramowitz, Irene A.Stegun (Eds.): **Handbook of mathematical Functions**, National Bureau of Standards, 1964, third printing, 1965

— Papers —

[28] P.Duhamel, H.Hollmann: **Split radix FFT algorithm**, Electronis Letters 20 pp.14-16, 1984

[29] P.Duhamel: **Implementation of 'split-radix' FFT algorithms for complex, real and real-symmetric data**, IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-34 pp.285-295, 1986

[30] M.A.Thornton, D.M.Miller, R.Drechsler: **Transformations Amongst the Walsh, Haar, Arithmetic and Reed-Muller Spectral Domains**,

[31] Moon Ho Lee, B.Sundar Rajan, J.Y.Park: **A Generalized Reverse Jacket Transform**,

[32] H.Malvar: **Fast computation of the discrete cosine transform through fast Hartley transform**, Electronics Letters 22 pp.352-353, 1986

[33] H.Malvar: **Fast Computation of the discrete cosine transform and the discrete Hartley transform**, IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35 pp.1484-1485, 1987

[34] F.Arguello, E.L.Zapata: **Fast Cosine Transform on the Successive Doubling Method**,

[35] S.Gudvangen: **Practical Applications of Number Theoretic Transforms**,

[36] R.Crandall, B.Fagin: **Discrete Weighted Transforms and Large Integer Arithmetic**, Math. Comp. (62) 1994 pp.305-324

[37] Daniel J.Bernstein: **Multidigit Multiplication for Mathematicians**, 1998

[38] Susan Landau, Neil Immermann: **The Similarities (and Differences) between Polynomials and Integers**, 1996

[39] Peter John Potts: **Computable Real Arithmetic Using Linear Fractional Transformations**, 1996

[40] Zhong-De Wang: **New algorithm for the slant transform**, IEEE Transactions Pattern Anal. Mach. Intell. PAMI-4, No.5, pp.551-555, September 1982

[41] R.P.Brent: **Fast multiple-precision evaluation of elementary functions**, J. ACM (23) 1976 pp.242-251

[42] B.Haible, T.Papanikolaou: **Fast multiprecision evaluation of series of rational numbers**,

[43] J.M.Borwein, P.B.Borwein: **?title?**, Article in Scientific American, March 1988

[44] D.H.Bailey, J.M.Borwein, P.B.Borwein and S.Plouffe: **The Quest for Pi**, 1996, online at `http://www.cecm.sfu.ca/~pborwein/`

[45] J.M.Borwein, P.B.Borwein: **Cubic and higher order algorithms for π**, Canad.Math.Bull. Vol.27 (4), 1984, pp.436-443

[46] J.M.Borwein, P.B.Borwein, F.G.Garvan: **Some cubic modular identities of Ramanujan**, Transactions A.M.S. 343, 1994, pp.35-47

[47] J.M.Borwein, F.G.Garvan: **Approximations to π via the Dedekind eta function**, March 27, 1996

[48] D.V.Chudnovsky, G.V.Chudnovsky: **Classical constants and functions: computations and continued fraction expansions**, in Number Theory: New York seminar 1989-1990, Springer Verlag 1991

[49] H.Cohen, F.R.Villegas, D.Zagier: **Convergence acceleration of alternating series**, 1997

[50] D.H.Bailey: **FFTs in External or Hierarchical Memory**, 1989

[51] D.H.Bailey: **The Fractional Fourier Transform and Applications**, 1995 online at `http://citeseer.nj.nec.com/`

[52] D.H.Bailey: **On the Computation of FFT-based Linear Convolutions**, June 1996

[53] M.Beeler, R.W.Gosper, R.Schroeppel: **HAKMEM**, MIT AI Memo 239, Feb. 29, 1972, Retyped and converted to html by Henry Baker, April 1995, online at `ftp://ftp.netcom.com/pub/hb/hbaker/`

[54] Advanced Micro Devices (AMD) Inc.: **AMD Athlon Processor, x86 code optimization guide**, Publication #22007, Revision H, June 2000 online at `http://www.amd.com/`

[55] P.Soderquist, M.Leeser: **An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations**, Cornell School of Electrical Engineering online at `http://orac.ee.cornell.edu:80/unit1/pgs/#papers`

[56] F.L.Bauer: **An Infinite Product for Square-Rooting with Cubic Convergence**, The Mathematical Intelligencer, 1998

[57] Bahman Kalantari, Jürgen Gerlach: **Newton's Method and Generation of a Determinantal Family of Iteration Functions**, 1998

[58] Nicholas J.Higham: **Stable Iterations for the Matrix Square Root**, August 1997

[59] Eugene Salamin: **Application of Quaternions to Computation with Rotations**, Working Paper, Stanford AI Lab, 1979, Edited and TeX-formatted by Henry G.Baker, 1995 online at `ftp://ftp.netcom.com/pub/hb/hbaker/`

[60] Thomas D.Howell, Jean-Claude Lafon: **The Complexity of the Quaternion Product**, Department of Computer Science, Cornell University, Ithaca, NY, June 1975
online at `ftp://ftp.netcom.com/pub/hb/hbaker/`

[61] Paweł Zieliński, Krystyna Ziętak: **The Polar Decomposition – Properties, Applications and Algorithms**, Annals of the Polish Mathematical Society, 38, 1995
online at `http://citeseer.nj.nec.com/`

[62] Nicholas J.Higham: **The Matrix Sign Decomposition and its Relation to the Polar Decomposition**, Linear Algebra and Appl., 212/213, 1994

[63] Erik Weisstein: **MathWorld**,
online at `http://mathworld.wolfram.com/`

[64] N.J.A.Sloane: **The On-Line Encyclopedia of Integer Sequences**
online at `http://www.research.att.com/~njas/sequences/`

[65] K.Cattel, S.Zhang, X.Sun, M.Serra, J.C.Muzio, D.M.Miller: **One-Dimensional Linear Hybrid Cellular Automata: Their Synthesis, Properties, and Applications in VLSI Testing**,

[66] Kevin Cattel, Shujian Zhang: **Minimal Cost One-Dimensional Linear Hybrid Cellular Automata of Degree Through 500**, 1994

[67] Andrew Klapper, Mark Goresky: **Feedback Shift Registers, 2-Adic Span and Combiners With Memory**, 1996

[68] Mark Goresky, Andrew Klapper: **Fibonacci and Galois Representations of Feedback with Carry Shift Registers**, 2000

[69] Carla Savage: **A Survey of Combinatorial Gray Codes**,
online at `http://citeseer.nj.nec.com/`

[70] Nirmal R.Saxena, Edward J.McCluskey: **Degree-r Primitive Polynomial Generation-$\mathbf{O}(r^3) \sim \mathbf{O}(r^4)$ Algorithms**, Center for Reliable Computing, Stanford University

[71] Kevin Cattell, Frank Ruskey, Joe Sawada, C.Robert Miers, Micaela Serra: **Generating Unlabeled Necklaces and Irreducible Polynomials over GF(2)**, Department of Computer Science, University of Victoria, Canada, 1998

[72] Frank Ruskey: **Simple combinatorial Gray codes constructed by reversing sublists**, Department of Computer Science, University of Victoria, Canada, 1993

[73] T.A.Jenkyns: **Loopless Gray Code Algorithms**,

— Software —

[74] Mikko Tommila: **apfloat, A High Performance Arbitrary Precision Arithmetic Package**, 1996,
online at `http://www.jjj.de/mtommila/`

[75] The PARI Group (C.Batut, K.Belabas, D.Bernardi, H.Cohen, M.Olivier et.al.): **PARI/GP**,
online at `http://www.parigp-home.de/`

[76] The Free Software Foundation (FSF): **GCC, the GNU Compiler Collection**,
online at `http://www.gnu.org/gcc/`

Index

N

O

P

Q

R

S

T

U

W

X

Z