How to compute π to 10^{12} digits

A crash course in high precision arithmetics

Part I: Fast multiplication

Part II: Iteration schemes, the AGM, binary splitting

Jörg Arndt <arndt@jjj.de>

"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

Multiplication is convolution

Multiplication of two numbers is essentially a convolution of the sequences of their digits. The *convolution* of the two sequences $a_k, b_k, k = 0 \ldots N-1$ is defined as the sequence c where

$$c_k \;:=\; \sum_{i+j=k} a_i b_j \qquad k = 0 \ldots 2N-2 \tag{1}$$

A (n-digit) number written in radix R as

$$a_{n-1}\; a_{n-2}\; \ldots\; a_2\; a_1\; a_0 \tag{2}$$

denotes a quantity of

$$\sum_{i=0}^{n-1} a_i \cdot R^i \;=\; a_{n-1} \cdot R^{n-1} + a_{n-2} \cdot R^{n-2} + \;\ldots\; + a_1 \cdot R + a_0 \tag{3}$$

For example, with decimal numbers one has $R = 10$, and the number 578 equals $5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0$.

The product of two numbers is almost the polynomial product

$$\sum_{k=0}^{2N-2} c_k\, R^k \;:=\; \sum_{i=0}^{N-1} a_i\, R^i \cdot \sum_{j=0}^{N-1} b_j\, R^j \tag{4}$$

The c_k are found by comparing coefficients: they must satisfy the convolution equation 1.

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 $c'_k = c_k \bmod R$ and add $(c_k - c'_k)/R$ to its left neighbor.

Multiplication is convolution [cont.]

An example: usually one would multiply the numbers 82 and 34 as follows:

$$\begin{array}{rrrr}
 & 82 & \times & 34 \\ \hline
 & 3 & {}^{3}2 & 8 \\
2 & {}^{2}4 & 6 & \\ \hline\hline
= 2 & 7 & 8 & 8
\end{array}$$

We have seen that the carries can be delayed to the end of the computation:

$$\begin{array}{rrrr}
 & 82 & \times & 34 \\ \hline
 & & 32 & 8 \\
 & 24 & 6 & \\ \hline\hline
 & 24 & 38 & 8 \\ \hline
= 2 & {}^{2}\,7 & {}^{3}\,8 & 8
\end{array}$$

. . . which is really polynomial multiplication (which in turn is a convolution of the coefficients):

$$\begin{array}{rrrr}
 & (8\,x+2) & \times & (3\,x+4) \\ \hline
 & & 32\,x & 8 \\
 & 24\,x^2 & 6\,x & \\ \hline\hline
= & 24\,x^2 & +38\,x & +8
\end{array}$$

The value of the polynomial $24\,x^2 + 38\,x + 8$ for $x = 10$ is 2788.

2-way splitting (Karatsuba multiplication)

Split the numbers A and B (assumed to have approximately the same length) into two pieces

$$\begin{aligned} A &= x\,A_1 + A_0 \\ B &= x\,B_1 + B_0 \end{aligned} \tag{1}$$

where x is a power of the radix (for decimal numbers the radix is 10) close to the half length of A and B. The usual multiplication scheme needs 4 multiplications with half precision for one multiplication with full precision:

$$A\,B = A_0 \cdot B_0 + x\,(A_0 \cdot B_1 + B_0 \cdot A_1) + x^2\,A_1 \cdot B_1 \tag{2}$$

Only the multiplications $A_i \cdot B_j$ need to be considered. The multiplications by x, a power of the radix are only shifts. If we use the relation

$$A\,B = (1+x)\,A_0 \cdot B_0 + x\,(A_1 - A_0) \cdot (B_0 - B_1) + (x + x^2)\,A_1 \cdot B_1 \tag{3}$$

we need 3 multiplications with half precision for one multiplication with full precision. Applying the scheme recursively until the numbers to multiply are of machine size we obtain an algorithm whose asymptotic cost is $\sim N^{\log_2(3)} \approx N^{1.585}$.

For squaring use

$$A^2 = (1+x)\,A_0^2 - x\,(A_1 - A_0)^2 + (x + x^2)\,A_1^2 \tag{4}$$

We compute $8231^2 = 67749361$:

```
8231^2 ==
 ==  (100*82+31)^2
 ==  (1+100)*31^2  - 100*(82-31)^2 + (100+100^2)*82^2
 ==  (1+100)*[961] - 100*[2601]    + (100+100^2)*[6724]
 ==  961 + 96100   - 260100        + 672400 + 67240000
 ==  67749361
```

And, yes, the scheme also works for polynomials.

3-way splitting (Toom-Cook multiplication)

The 3-way multiplication scheme of Bodrato and Zanoni:

```
A = a2*x^2 + a1*x + a0
B = b2*x^2 + b1*x + b0

S0 = a0 * b0
S1 = (a2+a1+a0) * (b2+b1+b0)
S2 = (4*a2+2*a1+a0) * (4*b2+2*b1+b0)
S3 = (a2-a1+a0) * (b2-b1+b0)
S4 = a2 * b2

S2 = (S2 - S3)/3  \\ division by 3
S3 = (S1 - S3)/2
S1 = S1 - S0
S2 = (S2 - S1)/2
S1 = S1 - S3 - S4
S2 = S2 - 2*S4
S3 = S3 - S2

P = S4*x^4+ S2*x^3+ S1*x^2+ S3*x + S0

P - A*B \\ == zero
```

4-way splitting multiplication

The 4-way multiplication scheme of Bodrato and Zanoni:

```
A = a3*x^3 + a2*x^2 + a1*x + a0
B = b3*x^3 + b2*x^2 + b1*x + b0

S1 = a3*b3
S2 = (8*a3+4*a2+2*a1+a0)*(8*b3+4*b2+2*b1+b0)
S3 = (+a3+a2+a1+a0)*(+b3+b2+b1+b0)
S4 = (-a3+a2-a1+a0)*(-b3+b2-b1+b0)
S5 = (+8*a0+4*a1+2*a2+a3)*(+8*b0+4*b1+2*b2+b3);
S6 = (-8*a0+4*a1-2*a2+a3)*(-8*b0+4*b1-2*b2+b3)
S7 = a0*b0


S2 += S5
S4 -= S3
S6 -= S5
S4 /= 2  \\
S5 -= S1
S5 -= (64*S7)
S3 += S4
S5 *= 2;  S5 += S6

S2 -= (65*S3)
S3 -= S1
S3 -= S7
S4 = -S4 \\
S6 = -S6 \\
S2 += (45*S3)
S5 -= (8*S3)
S5 /= 24  \\ division by 24

S6 -= S2
S2 -= (16*S4)
S2 /= 18  \\ division by 18
S3 -= S5
S4 -= S2
S6 += (30*S2)
S6 /= 60   \\ division by 60
S2 -= S6

P = S1*x^6 + S2*x^5 + S3*x^4 + S4*x^3 + S5*x^2 + S6*x + S7;

P - A*B \\ == zero
```

FFT-multiplication

Convolution can be done efficiently using the Fast Fourier Transform (FFT): Convolution is a simple (element-wise) multiplication in Fourier space. The FFT itself takes $\sim N \cdot \log N$ operations. Instead of the direct convolution ($\sim N^2$) one proceeds as follows:

- Compute the FFTs of both factors.
- Multiply the transformed sequences element-wise.
- Compute inverse transform of the product.

Note that (1) the multiplication of two polynomials can be achieved by the (more complicated) scheme:

- evaluate both polynomials at sufficiently many points
- element-wise multiply the values found
- find the polynomial corresponding to those (product-)values

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.

Re-launching our example ($82 \cdot 34 = 2788$), 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 as follows: 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.

The radix-2 DIF FFT algorithm

Splitting of the Fourier sum into a left and right half leads to the *decimation in frequency* (DIF) FFT algorithm.
For even values of n the k-th element of the Fourier transform is

$$\mathcal{F}\left[\,a\,\right]_k = \sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n} a_x\, z^{x\,k} \tag{1a}$$

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

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

where $z = e^{\sigma\, i\, 2\,\pi/n}$, $\sigma = \pm 1$ is the sign of the transform 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 $j \in \{0, 2, \ldots, \frac{n}{2} - 1\}$ and $\delta \in \{0, 1\}$:

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

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

$z^{(2\,j+\delta)\,n/2} = e^{\pm\pi\, i\,\delta}$ is equal to plus or minus one for $\delta = 0$ or $\delta = 1$ corresponding to k even or odd. The last two equations are, more compactly written, the key to the *radix-2 DIF FFT step*:

$$\mathcal{F}\left[\,a\,\right]^{(even)} \stackrel{n/2}{=} \mathcal{F}\left[\,a^{(left)} + a^{(right)}\,\right] \tag{3a}$$

$$\mathcal{F}\left[\,a\,\right]^{(odd)} \stackrel{n/2}{=} \mathcal{F}\left[\,\mathcal{S}^{1/2}\left(a^{(left)} - a^{(right)}\right)\,\right] \tag{3b}$$

The radix-2 DIT FFT algorithm

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

$$\mathcal{F}\left[\,a\,\right]_k \;=\; \sum_{x=0}^{n-1} a_x\, z^{x\,k} \;=\; \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{(2\,x+1)\,k} \qquad \text{(1a)}$$

$$= \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{2\,x\,k} \qquad \text{(1b)}$$

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

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

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

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

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

Observing that z^2 is just the root of unity that appears in a length-$n/2$ transform one can rewrite the last two equations to obtain the *radix-2 DIT FFT step*:

$$\mathcal{F}\left[\,a\,\right]^{(left)} \stackrel{n/2}{=} \mathcal{F}\left[\,a^{(even)}\,\right] + \mathcal{S}^{1/2}\mathcal{F}\left[\,a^{(odd)}\,\right] \qquad \text{(3a)}$$

$$\mathcal{F}\left[\,a\,\right]^{(right)} \stackrel{n/2}{=} \mathcal{F}\left[\,a^{(even)}\,\right] - \mathcal{S}^{1/2}\mathcal{F}\left[\,a^{(odd)}\,\right] \qquad \text{(3b)}$$

Radix-4 FFT algorithms

The radix-2 DIF step in the new notation:

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

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

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

The radix-2 DIT step in the new notation:

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

Note that $\mathcal{S}^{0/2} = \mathcal{S}^0$ is the identity operator.
The *radix-4 DIT FFT step*:

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

In contrast to the radix-2 step that happens to be identical for forward and backward transform the sign of the transform $\sigma = \pm 1$ appears here.

Split radix FFT algorithms

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 we use the first, the one for the even indices. For the odd indices we use the radix-4 splitting: The radix-4 decimation in frequency (DIF) step for the split radix FFT:

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

Now we have expressed the length-$N = 2^n$ FFT as one length-$N/2$ and two length-$N/4$ FFTs.

The operation count of the split radix FFT is actually lower than that of the radix-4 FFT.

Using the introduced notation it is almost trivial to write down the DIT version of the algorithm: The radix-4 decimation in time (DIT) step for the split radix FFT:

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

Cyclic vs. linear convolution

The *cyclic convolution* (or *circular convolution*) of two length-n sequences $A = [a_0, a_1, \dots, a_{n-1}]$ and $B = [b_0, b1, \dots, b_{n-1}]$ is defined as the length-n sequence C with elements C_τ as:

$$C = A \circledast B \tag{1a}$$

$$C_\tau := \sum_{x+y \equiv \tau \pmod{n}} a_x \, b_y \tag{1b}$$

The last equation may be rewritten as

$$C_\tau := \sum_{x=0}^{n-1} a_x \, b_{(\tau - x) \mod n} \tag{2}$$

That is, indices $\tau - x$ wrap around, it is a cyclic convolution.
The FFT scheme

$$A \circledast B = \mathcal{F}^{-1} \left[\mathcal{F}\left[A \right] \mathcal{F}\left[B \right] \right] \tag{3}$$

computes the cyclic convolution (the polynomial product is computed modulo z^n).
But we want the *linear* convolution:

$$C_\tau := \sum_{x=0}^{n-1} a_x \, b_{(\tau - x) \mod \text{nothing}} \tag{4}$$

Solution: We (roughly) double the length of the input sequences by appending zeros to the end. The polynomial product is computed modulo z^{2n}, but $2n > \deg AB$.

Radix and precision with FFT multiplication

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: The cumulative sums have to be represented precisely enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a will appear in the middle of the product and when multiplicand and multiplier consist of 'nines' (that is $R-1$) only. For radix R and a precision of N `LIMB`s Let the maximal possible value be C, then

$$C = N\,(R-1)^2 \tag{1}$$

The number of bits to represent C exactly is the integer greater or equal to

$$\log_2(N\,(R-1)^2) = \log_2 N + 2\,\log_2(R-1) \tag{2}$$

Due to numerical errors there must be a few more bits for safety. If computations are made using double-precision floating point numbers (C-type `double`) one typically has a mantissa of 53 bits. then we need to have

$$M \geq \log_2 N + 2\,\log_2(R-1) + S \tag{3}$$

where M :=mantissa-bits and S :=safety-bits. Using $\log_2(R-1) < \log_2(R)$:

$$N_{max}(R) = 2^{M-S-2\,\log_2(R)} \tag{4}$$

Radix R	max # `LIMB`s	max # hex digits	max # bits
$2^{10} = 1024$	$1048,576\,k$	$2621,440\,k$	$10240\,M$
$2^{11} = 2048$	$262,144\,k$	$720,896\,k$	$2816\,M$
$2^{12} = 4096$	$65,536\,k$	$196,608\,k$	$768\,M$
$2^{13} = 8192$	$16384\,k$	$53,248\,k$	$208\,M$
$2^{14} = 16384$	$4096\,k$	$14,336\,k$	$56\,M$
$2^{15} = 32768$	$1024\,k$	$3840\,k$	$15\,M$
$2^{16} = 65536$	$256\,k$	$1024\,k$	$4\,M$

For decimal numbers:

Radix R	max # `LIMB`s	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

Do the sum of digits test!

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 implement 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: for a given modulus m primitive n-th 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:

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

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 to m (i.e. $\gcd(n, m) = 1$), the second condition for NTTs:

$$n \perp m \iff \exists\, n^{-1} \text{ in } \mathbb{Z}/m\mathbb{Z} \tag{2}$$

Prime modulus

If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = GF(p)$: 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 $n < p$.

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). Primes of that form are not 'exceptional'. 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:

```
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 !
}
```

The algorithm is a simple search and might seem ineffective. In practice the root is found after only several tries.

An element of order n in $\mathbb{F}_p$ is returned by this function:

```
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
}
```

Division: Inversion

The ordinary division algorithm is far too expensive for numbers of extreme precision. Instead one replaces the division $\frac{a}{d}$ by the multiplication of a with the inverse of d. The inverse of d is computed by finding a starting approximation $x_0 \approx \frac{1}{d}$ and then iterating

$$x_{k+1} = x_k + x_k\,(1 - d\,x_k) \tag{1}$$

until the desired precision is reached. The convergence is quadratic (second order), which means that the number of correct digits is doubled with each step: if $x_k = \frac{1}{d}(1+e)$ then $x_{k+1} = \frac{1}{d}\left(1 - e^2\right)$.

Moreover, each only requires 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 of the current 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[1] 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 of an inverse starting from a two-digit initial approximation:

$$\begin{aligned}
d &:= 3.1415926 && (2)\\
x_0 &= 0.31 \qquad \text{initial 2 digit approximation for } 1/d && (3)\\[1ex]
d \cdot x_0 &= 3.141 \cdot 0.3100 = 0.9737 && (4)\\
y_0 &:= 1.000 - d \cdot x_0 = 0.02629 && (5)\\
x_0 \cdot y_0 &= 0.3100 \cdot 0.02629 = 0.0081(49) && (6)\\
x_1 &:= x_0 + x_0 \cdot y_0 = 0.3100 + 0.0081 = 0.3181 && (7)\\[1ex]
d \cdot x_1 &= 3.1415926 \cdot 0.31810000 = 0.9993406 && (8)\\
y_1 &:= 1.0000000 - d \cdot x_0 = 0.0006594 && (9)\\
x_1 \cdot y_1 &= 0.31810000 \cdot 0.0006594 = 0.0002097(5500) && (10)\\
x_2 &:= x_1 + x_1 \cdot y_1 = 0.31810000 + 0.0002097 = 0.31830975 && (11)
\end{aligned}$$

[1] 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.

Root extraction

Computation of square roots can be done using a similar scheme: first compute $\frac{1}{\sqrt{d}}$ then a final multiply with d gives $\sqrt{d}$. Find a starting approximation $x_0 \approx \frac{1}{\sqrt{d}}$ then iterate

$$x_{k+1} = x_k + x_k \frac{(1 - d\, x_k^2)}{2} \tag{1}$$

until the desired precision is reached. Convergence is again 2nd order: if $x_k = \frac{1}{\sqrt{d}}(1+e)$ then

$$x_{k+1} = \frac{1}{\sqrt{d}} \left(1 - \frac{3}{2}e^2 - \frac{1}{2}e^3\right) \tag{2}$$

Similar considerations as above (with squaring considered as expensive as multiplication[2]) 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:

$$x_{k+1} = x_k + x_k \frac{(1 - d\, x_k^2)}{2} + x_k \frac{3\,(1 - d\, x_k^2)^2}{8} \tag{3}$$

[2] Indeed it costs about $\frac{2}{3}$ of a multiplication.

Inverse a-th root, a general expression

There is a nice general formula that allows to build iterations with arbitrary order of convergence for $1/\sqrt[a]{d} = d^{-1/a}$ that involve no long division.

One uses the identity

$$
\begin{aligned}
d^{-1/a} &= x\,(1-(1-x^a\,d))^{-1/a} && (1)\\
&= x\,(1-y)^{-1/a} \quad \text{where} \quad y := (1-x^a\,d) && (2)
\end{aligned}
$$

Taylor expansion gives

$$
d^{-1/a} = x \sum_{k=0}^{\infty} (1/a)^{\bar{k}}\, y^k \tag{3}
$$

where $z^{\bar{k}} := z(z+1)(z+2)\dots(z+k-1)$ (and $z^{\bar{0}} = 1$). Written out:

$$
\begin{aligned}
d^{-1/a} = x\,\frac{1}{\sqrt[a]{1-y}} = x\,\Bigg(1 + \frac{y}{a} + \frac{(1+a)\,y^2}{2\,a^2} + \frac{(1+a)(1+2a)\,y^3}{6\,a^3} + \quad (4)\\
+ \frac{(1+a)(1+2a)(1+3a)\,y^4}{24\,a^4} + \dots + \frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\,a^n}\,y^n + \dots\Bigg)
\end{aligned}
$$

A n-th order iteration for $d^{-1/a}$ is obtained by truncating the above series after the $(n-1)$-th term:

$$
\begin{aligned}
\Phi_n(x) &:= x \sum_{k=0}^{n-1} (1/a)^{\bar{k}}\, y^k && (5)\\
x_{k+1} &= \Phi_n(x_k) && (6)
\end{aligned}
$$

Convergence is n-th order:

$$
\Phi_n(d^{-1/a}(1+e)) = d^{-1/a}(1+O(e^n)) \tag{7}
$$

Iterations for the inversion of a function

An *iteration* for a zero r (or root, $f(r) = 0$) of a function $f(x)$ are themselves functions $\Phi(x)$ that, when 'used' like

$$x_{k+1} = \Phi(x_k) \tag{1}$$

will make x_k converge towards the root: $x_\infty = r$. Convergence is subject to the condition that x_0 was chosen not too far away from r. The function $\Phi(x)$ must (and can) be constructed so that it has an attracting fixed point where $f(x)$ has a zero:

$$\Phi(r) = r \qquad \text{(fixed point)} \tag{2}$$

$$|\Phi'(r)| < 1 \qquad \text{(attracting)} \tag{3}$$

This type of iteration is a so-called *one-point iteration.* There are also *multi-point iterations*, these are of the form $x_{k+1} = \Phi(x_k, x_{k-1}, \ldots, x_{k-j}), j \geq 1$. The best known example is the two-point iteration

$$x_{k+1} = \Phi(x_k, x_{k-1}) = x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{4}$$

We are mainly concerned with one-point iterations in what follows.

Order of convergence: linear vs. *super-linear.*

The number of correct digits grows exponentially (to the base n) at each step. Iterations of second order ($n = 2$) are often called *quadratic* (or *quadratically convergent*), those of third order *cubic* iterations. Fourth, fifth and sixth order iterations are called *quartic*, *quintic* and *sextic* and so on.

For $n \geq 2$ the function Φ has a *super-attracting fixed point* at r: $\Phi'(r) = 0$. For an iteration of order n one has

$$\Phi'(r) = 0, \quad \Phi''(r) = 0, \quad \ldots, \quad \Phi^{(n-1)}(r) = 0 \tag{5}$$

There seems to be no standard term for emphasizing the number of derivatives vanishing at the fixed point: *super-attracting of order n* might be appropriate.

Schröder's formula

Let $n \geq 2$ then the expression

$$S_n(x) \;:=\; x + \sum_{t=1}^{n-1} (-1)^t \, \frac{f(x)^t}{t!} \left(\frac{1}{f'(x)} \, \partial \right)^{t-1} \frac{1}{f'(x)} \tag{1}$$

gives a $n-$th order iteration for a (simple) root r of f. This is, explicitly,

$$\begin{aligned} S \;=\; x \quad & - \frac{f}{1!\,f'} \quad - \frac{f^2}{2!\,f'^3} \cdot f'' \quad - \frac{f^3}{3!\,f'^5} \cdot \left(3f''^2 - f'f'''\right) \\ & - \frac{f^4}{4!\,f'^7} \cdot \left(15f''^3 - 10f'f''f''' + f'^2 f''''\right) \\ & - \frac{f^5}{5!\,f'^9} \cdot \left(105f''^4 - 105f'f''^2f''' + 10f'^2f'''^2 + 15f'^2f''f'''' - f'^3f'''''\right) - \dots \end{aligned} \tag{2}$$

The third order iteration obtained upon truncation after the third term on the right hand side, written as

$$S_3 \;=\; x - \frac{f}{f'} \left(1 + \frac{f f''}{2f'^2} \right) \tag{3}$$

is sometimes referred to as 'Householder's method'. Approximating the second term on the rhs. as $\frac{f}{f'}\left(1 - \frac{ff''}{2f'^2}\right)^{-1}$ gives Halley's formula.

Write

$$S \;=\; x - U_1 \frac{f}{1!\,f'} - U_2 \frac{f^2}{2!\,f'^3} - U_3 \frac{f^3}{3!\,f'^5} - \dots - U_n \frac{f^n}{n!\,f'^{2n-1}} - \dots \tag{4}$$

then $U_1 = 1$, $U_2 = f''$, $U_3 = 3f''^2 - f'f'''$, and we have the recursion

$$U_n \;=\; (2n-3)f''U_{n-1} - f'U'_{n-1} \tag{5}$$

Alternatively write

$$S \;=\; x - Y_1 \left(\frac{f}{f'}\right) - Y_2 \left(\frac{f}{f'}\right)^2 - Y_3 \left(\frac{f}{f'}\right)^3 - \dots - Y_t \left(\frac{f}{f'}\right)^t - \dots \tag{6}$$

then $Y_1 = 1$ and

$$Y_t \;=\; \frac{1}{t} \left(2\,(t-1)\,\frac{f''}{2f'}\,Y_{t-1} - Y'_{t-1} \right) \tag{7}$$

A simple derivation of Schröder's formula

The starting point is the Taylor series of a function f around x_0:

$$f(x) \;=\; f(x_0) + f'(x_0)\,(x - x_0) + \frac{1}{2} f''(x_0)\,(x - x_0)^2 + \dots \qquad (1)$$

Now let $f(x_0) = y_0$ and r be the zero of f (that is, $f(r) = 0$). We expand the inverse $g = f^{-1}$ around y_0:

$$g(0) \;=\; g(y_0) + g'(y_0)\,(0 - y_0) + \frac{1}{2} g''(y_0)\,(0 - y_0)^2 + \dots \qquad (2)$$

Using $x_0 = g(y_0)$ and $g(0) = r$ we obtain

$$r \;=\; x_0 - g'(y_0)\,f(x_0) + \frac{1}{2} g''(y_0)\,f(x_0)^2 - \frac{1}{6} g'''(y_0)\,f(x_0)^3 + \dots \qquad (3)$$

Remains to express the derivatives of the inverse g in terms of (derivatives of) f. Set

$$f \circ g \;=\; \mathrm{id}\,, \qquad \text{that is:} \quad f(g(x)) \;=\; x \qquad (4)$$

and derive the equation (chain rule) to obtain $g'(f(x))\,f'(x) = 1$, so $g'(y) = \frac{1}{f'(x)}$. Derive $f(g(x)) - x$ multiple times to obtain (arguments y of g and x of f are omitted for readability):

$$1 \;=\; f'g' \qquad (5a)$$
$$0 \;=\; g'f'' + f'^2 g'' \qquad (5b)$$
$$0 \;=\; g'f''' + 3f'f''g'' + f'^3 g''' \qquad (5c)$$
$$0 \;=\; g'f'''' + 4f'g''f''' + 3f''^2 g'' + 6f'^2 f''g''' + f'^4 g'''' \qquad (5d)$$

This system of linear equations in the derivatives of g can be solved successively for g', g'', g''', ...:

$$g' \;=\; \frac{1}{f'} \qquad (6a)$$
$$g'' \;=\; -\frac{f''}{f'^3} \qquad (6b)$$
$$g''' \;=\; \frac{1}{f'^5}\left(3f''^2 - f'f'''\right) \qquad (6c)$$

Thereby equation 3 can be written as

$$r \;=\; x - \frac{1}{f'}\,f + \frac{1}{2}\left(-\frac{f''}{f'^3}\right) f^2 - \frac{1}{6}\left(\frac{1}{f'^5}\left(3f''^2 - f'f'''\right)\right) f^3 + \dots \qquad (7)$$

which is Schröder's iteration.

Householder's formula

For $n \geq 2$ the expression

$$H_n(x) \;:=\; x + (n-1)\,\frac{\left(\frac{1}{f(x)}\right)^{(n-2)}}{\left(\frac{1}{f(x)}\right)^{(n-1)}} \tag{1}$$

gives a $n-$th order iteration for a (simple) root r of f.

$$H_2(x) \;=\; x - \frac{f}{f'} \tag{2a}$$

$$H_3(x) \;=\; x - \frac{2ff'}{2f'^2 - ff''} \tag{2b}$$

$$H_4(x) \;=\; x - \frac{3f(ff'' - 2f'^2)}{6ff'f'' - 6f'^3 - f^2f'''} \tag{2c}$$

$$H_5(x) \;=\; x + \frac{4f\left(6f'^3 - 6ff'f'' + f^2f'''\right)}{f^3f'''' - 24f'^4 + 36ff'^2f'' - 8f^2f'f''' - 6f^2f''^2} \tag{2d}$$

The second order variant is Newton's formula, the third order iteration is called Halley's formula.

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

$$F_m(x) \;=\; \frac{F_{m-1}(x)}{F'_{m-1}(x)^{1/m}} \tag{3a}$$

$$H_m(x) \;=\; x - \frac{F_{m-1}(x)}{F'_{m-1}(x)} \tag{3b}$$

An alternative recursive formulation:

$$Q_2(x) \;=\; 1 \tag{4a}$$

$$Q_{m+1} \;=\; f'(x)\,Q_m(x) - \frac{1}{m-1}\,f(x)\,Q'_m(x) \tag{4b}$$

$$H_m \;=\; x - f(x)\,\frac{Q_m(x)}{Q_{m-1}(x)} \tag{4c}$$

For multiple roots use iterations for f/f', for example (Schröder):

$$S^{\%} \;=\; x - \frac{ff'}{(f'^2 - ff'')} - \frac{f^2f'\left(ff'f''' - 2ff''^2 + f'^2f''\right)}{2\left(ff'' - f'^2\right)^3} - \ldots \tag{5}$$

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

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{1a}$$

$$b_{k+1} = \sqrt{a_k\, b_k} \tag{1b}$$

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

$$c_k^2 = a_k^2 - b_k^2 \tag{2}$$

$$= (a_{k-1} - a_k)^2 \tag{3}$$

One further defines

$$R'(k) := 1 - \frac{1}{2}\sum_{n=0}^{\infty} 2^n c_n^2 \tag{4}$$

corresponding to $AGM(1, k)$, that is, $a_0 = 1, b_0 = k, c_0 = \sqrt{1 - k^2}$.

It can be shown that

$$F\left(\left.\begin{matrix}\frac{1}{2}, \frac{1}{2} \\ 1\end{matrix}\right| 1 - \frac{b^2}{a^2}\right) = \frac{a}{AGM(a, b)} == \frac{1}{AGM(1, b/a)} \tag{5}$$

$$F\left(\left.\begin{matrix}\frac{1}{2}, \frac{1}{2} \\ 1\end{matrix}\right| x\right) = \frac{1}{AGM(1, \sqrt{1 - x})} \tag{6}$$

An alternative way for the computation for the AGM iteration is

$$c_{k+1} = \frac{a_k - b_k}{2} \tag{7a}$$

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{7b}$$

$$b_{k+1} = \sqrt{a_{k+1}^2 - c_{k+1}^2} \tag{7c}$$

The AGM, Schönhage's variant

Schönhage gives the most economic variant of the AGM, which, apart from the square root, only needs one squaring per step:

$$
\begin{aligned}
A_0 &= a_0^2 && \text{(1a)}\\
B_0 &= b_0^2 && \text{(1b)}\\
t_0 &= 1 - (A_0 - B_0) && \text{(1c)}\\
S_k &= \frac{A_k + B_k}{4} && \text{(1d)}\\
b_k &= \sqrt{B_k} \qquad \text{[square root]} && \text{(1e)}\\
a_{k+1} &= \frac{a_k + b_k}{2} && \text{(1f)}\\
A_{k+1} &= a_{k+1}^2 \qquad \text{[squaring]} && \text{(1g)}\\
&= \left(\frac{\sqrt{A_k} + \sqrt{B_k}}{2}\right)^2 = \frac{A_k + B_k}{4} + \frac{\sqrt{A_k\, B_k}}{2} && \text{(1h)}\\
B_{k+1} &= 2\,(A_{k+1} - S_k) \;=\; b_{k+1}^2 && \text{(1i)}\\
c_{k+1}^2 &= A_{k+1} - B_{k+1} \;=\; a_{k+1}^2 - b_{k+1}^2 && \text{(1j)}\\
t_{k+1} &= t_k - 2^{k+1}\, c_{k+1}^2 && \text{(1k)}
\end{aligned}
$$

Starting with $a_0 = A_0 = 1$, $B_0 = 1/2$ one has

$$\pi \;\approx\; \frac{2\,a_n^2}{t_n} \tag{2}$$

This is a special case of Legendre's relation (a relation between complete elliptic integrals) and was discovered (independently) 1976 by Salamin and Brent. However, the relations has been given 200 years earlier by Gauss.

Superlinear iterations for π

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[3] 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):

$$\begin{aligned} a_0 &= 1 && (1a) \\ b_0 &= \frac{1}{\sqrt{2}} && (1b) \\ p_n &= \frac{2\, a_{n+1}^2}{1 - \sum_{k=0}^{n} 2^k\, c_k^2} \quad \to \pi && (1c) \\ \pi - p_n &= \frac{\pi^2\, 2^{n+4}\, e^{-\pi\, 2^{n+1}}}{AGM^2(a_0, b_0)} && (1d) \end{aligned}$$

Borwein's quartic (fourth order) iteration, variant $r = 4$ as in [hfloat: src/pi/pi4th.cc], #FPM=170.5:

$$\begin{aligned} y_0 &= \sqrt{2} - 1 && (2a) \\ a_0 &= 6 - 4\sqrt{2} && (2b) \\ y_{k+1} &= \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}} \quad \to 0+ && (2c) \\ &= \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} && (2d) \\ a_{k+1} &= a_k\, (1 + y_{k+1})^4 - 2^{2k+3}\, y_{k+1}\, (1 + y_{k+1} + y_{k+1}^2) \quad \to \frac{1}{\pi} && (2e) \\ &= a_k\, ((1 + y_{k+1})^2)^2 - 2^{2k+3}\, y_{k+1}\, ((1 + y_{k+1})^2 - y_{k+1}) && (2f) \\ 0 &< a_k - \pi^{-1} \le 16 \cdot 4^n\, 2\, e^{-4^n\, 2\, \pi} && (2g) \end{aligned}$$

Identities 2d and 2f show how to save operations.

[3]using radix $10,000$ and 1 million LIMBs.

More iterations for π

Derived AGM iteration (second order) as in [hfloat: src/pi/pideriv.cc], #FPM=276.2:

$$\begin{aligned}
x_0 &= \sqrt{2} && \text{(1a)}\\
p_0 &= 2+\sqrt{2} && \text{(1b)}\\
y_1 &= 2^{1/4} && \text{(1c)}\\
x_{k+1} &= \frac{1}{2}\left(\sqrt{x_k}+\frac{1}{\sqrt{x_k}}\right) \quad (k \geq 0) \quad \to 1+ && \text{(1d)}\\
y_{k+1} &= \frac{y_k\sqrt{x_k}+\frac{1}{\sqrt{x_k}}}{y_k+1} \quad (k \geq 1) \quad \to 1+ && \text{(1e)}\\
p_{k+1} &= p_k\,\frac{x_k+1}{y_k+1} \quad (k \geq 1) \quad \to \pi + && \text{(1f)}\\
p_k - \pi &= 10^{-2^{k+1}} && \text{(1g)}
\end{aligned}$$

Cubic AGM as in [hfloat: src/pi/picubagm.cc], #FPM=182.7:

$$\begin{aligned}
a_0 &= 1 && \text{(2a)}\\
b_0 &= \frac{\sqrt{3}-1}{2} && \text{(2b)}\\
a_{n+1} &= \frac{a_n+2\,b_n}{3} && \text{(2c)}\\
b_{n+1} &= \sqrt[3]{\frac{b_n\,(a_n^2+a_n\,b_n+b_n^2)}{3}} && \text{(2d)}\\
p_n &= \frac{3\,a_n^2}{1-\sum_{k=0}^{n} 3^k\,(a_k^2-a_{k+1}^2)} && \text{(2e)}
\end{aligned}$$

Quintic (5th order) iteration as in [hfloat: src/pi/pi5th.cc], #FPM=353.2:

$$\begin{aligned}
s_0 &= 5(\sqrt{5}-2) && \text{(3a)}\\
a_0 &= \frac{1}{2} && \text{(3b)}\\
s_{n+1} &= \frac{25}{s_n(z+x/z+1)^2} \quad \to 1 && \text{(3c)}\\
\text{where } x &= \frac{5}{s_n}-1 \quad \to 4 && \text{(3d)}\\
\text{and } y &= (x-1)^2+7 \quad \to 16 && \text{(3e)}\\
\text{and } z &= \left(\frac{x}{2}\left(y+\sqrt{y^2-4x^3}\right)\right)^{1/5} \quad \to 2 && \text{(3f)}\\
a_{n+1} &= s_n^2 a_n - 5^n\left(\frac{s_n^2-5}{2}+\sqrt{s_n\,(s_n^2-2s_n+5)}\right) \quad \to \frac{1}{\pi} && \text{(3g)}\\
a_n - \frac{1}{\pi} &< 16\cdot 5^n\,e^{-\pi\,5^n} && \text{(3h)}
\end{aligned}$$

High order = fast?

Nonic (9th order) iteration as in [hfloat: src/pi/pi9th.cc], #FPM=273.7:

$$a_0 = \frac{1}{3} \tag{1a}$$

$$r_0 = \frac{\sqrt{3}-1}{2} \tag{1b}$$

$$s_0 = (1 - r_0^3)^{1/3} \tag{1c}$$

$$t = 1 + 2\,r_k \tag{1d}$$

$$u = \left(9\,r_k\,(1 + r_k + r_k^2)\right)^{1/3} \tag{1e}$$

$$v = t^2 + t\,u + u^2 \tag{1f}$$

$$m = \frac{27\,(1 + s_k + s_k^2)}{v} \tag{1g}$$

$$a_{k+1} = m\,a_k + 3^{2\,k-1}\,(1-m) \quad \to \frac{1}{\pi} \tag{1h}$$

$$s_{k+1} = \frac{(1-r_k)^3}{(t + 2\,u)\,v} \tag{1i}$$

$$r_{k+1} = (1 - s_k^3)^{1/3} \tag{1j}$$

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()
```

The binary splitting algorithm (for products)

We motivate the binsplit algorithm by giving the analogue for the fast computation of the factorial. Define $f_{m,n} := m\cdot(m+1)\cdot(m+2)\cdots(n-1)\cdot n$, then $n! = f_{1,n}$. We compute $n!$ by recursively using the relation $f_{m,n} = f_{m,x} \cdot f_{x+1,n}$ where $x = \lfloor (m+n)/2 \rfloor$:

```
indent(i)=for(k=1,8*i,print1(" "));  \\ aux: print 8*i spaces

F(m, n, i=0)=
{ /* Factorial, self documenting */
    local(x, ret);
    indent(i);  print( "F(", m, ", ", n, ")");
    if ( m==n, /* then: */
        ret = m;  \\ == F(m,m)
    , /* else: */
        x = floor( (m+n)/2 );
        ret = F(m, x, i+1) * F(x+1, n, i+1);
    );
    indent(i);  print( "^== ", ret);
    return( ret );
}
```

The function prints the intermediate values occuring in the computation. The additional parameter `i` keeps track of the calling depth, used with the auxiliary function `indent()`.

The intermediate quantities with the computation of $8!$ =`F(1,8)` are

```
F(1, 8)
        F(1, 4)
                F(1, 2)
                        F(1, 1)
                        ^== 1
                        F(2, 2)
                        ^== 2
                ^== 2
                F(3, 4)
                        F(3, 3)
                        ^== 3
                        F(4, 4)
                        ^== 4
                ^== 12
        ^== 24
        F(5, 8)
                F(5, 6)
                        F(5, 5)
                        ^== 5
                        F(6, 6)
                        ^== 6
                ^== 30
                F(7, 8)
                        F(7, 7)
                        ^== 7
                        F(8, 8)
                        ^== 8
                ^== 56
        ^== 1680
^== 40320
```

A fragment like

```
                F(5, 6)
                        F(5, 5)
                        ^== 5
                        F(6, 6)
                        ^== 6
                ^== 30
```

says "`F(5,6)` called `F(5,5)` [which returned 5], then called `F(6,6)` [which returned 6]. Then `F(5,6)` returned 30." For the computation of other products modify the line `ret=m;` as indicated in the code.

The binary splitting algorithm (for sums)

For the evaluation of a sum $\sum_{k=0}^{N-1} a_k$ we use the ratios r_k of consecutive terms:

$$r_k \;:=\; \frac{a_k}{a_{k-1}} \tag{1}$$

Set $a_{-1} := 1$ to avoid a special case for $k = 0$. One has

$$\sum_{k=0}^{N-1} a_k \;=:\; r_0\,(1 + r_1\,(1 + r_2\,(1 + r_3\,(1 + \ldots (1 + r_{N-1}) \ldots)))) \tag{2}$$

Now define

$$r_{m,n} \;:=\; r_m\,(1 + r_{m+1}\,(\ldots (1 + r_n) \ldots)) \quad \text{where} \quad m < n \tag{3a}$$

$$r_{m,m} \;:=\; r_m \tag{3b}$$

then

$$r_{m,n} \;=\; \frac{1}{a_{m-1}} \sum_{k=m}^{n} a_k \tag{4}$$

and especially

$$r_{0,n} \;=\; \sum_{k=0}^{n} a_k \tag{5}$$

We have

$$\begin{aligned} r_{m,n} \;&=\; r_m + r_m \cdot r_{m+1} + r_m \cdot r_{m+1} \cdot r_{m+2} + \ldots && \text{(6a)} \\ &\quad\;\cdots + r_m \cdot \cdots \cdot r_x + r_m \cdot \cdots \cdot r_x \cdot [r_{x+1} + \cdots + r_{x+1} \cdot \cdots \cdot r_n] \\ &=\; r_{m,x} + \prod_{k=m}^{x} r_k \cdot r_{x+1,n} && \text{(6b)} \end{aligned}$$

The product telescopes, one gets (for $m \le x < n$)

$$r_{m,n} \;=\; r_{m,x} + \frac{a_x}{a_{m-1}} \cdot r_{x+1,n} \tag{7}$$

The binary splitting algorithm (for sums) [cont.]

```
R(m, n)=
{ /* Rational binsplit */
    local(x, ret);
    if ( m==n, /* then: */
        ret = A(m)/A(m-1);
    , /* else: */
        x = floor( (m+n)/2 );
        ret = R(m, x) + A(x) / A(m-1) * R(x+1, n);
    );
    return( ret );
}
```

The intermediate values with the computation of $\sum_{k=0}^{6} 2^{-(k+1)}$ are

```
        R(0, 6)
                R(0, 3)
                        R(0, 1)
                                R(0, 0)
                                ^== 1/2
                                R(1, 1)
                                ^== 1/2
                        ^== 3/4
                        R(2, 3)
                                R(2, 2)
                                ^== 1/2
                                R(3, 3)
                                ^== 1/2
                        ^== 3/4
                ^== 15/16
                R(4, 6)
                        R(4, 5)
                                R(4, 4)
                                ^== 1/2
                                R(5, 5)
                                ^== 1/2
                        ^== 3/4
                        R(6, 6)
                        ^== 1/2
                ^== 7/8
        ^== 127/128
```

Reducing the intermediate fractions to lowest terms is necessary with certain sums. For example, computing arctan(1/10) without reduction shows the explosive growth of intermediate quantities:

```
    Q(0, 6)
            Q(0, 3)
                    Q(0, 1)
                            Q(0, 0)
                            ^== [1, 10]
                            Q(1, 1)
                            ^== [-10, 3000]
                    ^== [29900, 300000]
                    Q(2, 3)
                            Q(2, 2)
                            ^== [3000, -500000]
                            Q(3, 3)
                            ^== [-500000, 70000000]
                    ^== [-1042500000000000000, 175000000000000000000]
            ^== [15697812750000000000000000000, 157500000000000000000000000000]
     [--snip--]
```

More details are given in the online draft of my book:

`http://www.jjj.de/fxt/#fxtbook`

Thanks for your feedback!

Jörg Arndt `<arndt@jjj.de>`