Programs to Calculate some Mathematical Constants to Large Precision

Document Version 1.50

Prepared Using LaTeX

by

Stefan Spännare
E-mail: stefan@spaennare.se
September 19, 2007

Contents

1 Introduction

Note, this document is under development. Please look back for updated versions.

This documents describes algorithms to calculate some mathematical constants used by some of the C-programs in the program package *SSPROG*. Some timings, benchmarks and references are also given. The programs are not so fast (much faster programs exist especially for π) but they demonstrates the principle of multi precision calculations. See also the header of each C-program source code for more information.

At present algorithms used by the programs ***sspi, sseln2, ssgamma, ssgam134, sscatal, sszeta3, sszeta*** and ***sspieln2*** (in the NUMBERS directory) are described in this document.

The home page of the author and the web page of the program package and this document are found here:

http://www.spaennare.se/index.html

http://www.spaennare.se/ssprog.html

2 Disclaimer

For all the programs in this package the following statement is valid:

I make no warranties that this program is (1) free of errors, (2) consistent with any standard merchantability, or (3) meeting the requirements of a particular application. This software shall not, partly or as a whole, participate in a process, whose outcome can result in injury to a person or loss of property. It is solely designed for analytical work. Permission to use, copy, and distribute is hereby granted without fee, providing that the header above including this notice appears in all copies.

Please report comments and bugs in both the programs and this document to:

E-mail: stefan@spaennare.se

3 General information

The mathematical constants are calculated to desired precision (i.e. n decimal digits). Functions for multi precision calculations are included in the programs in this package. Especially fast FFT based multiplication must be used for large numbers. This presentation of the algorithms is quite brief (only the formulas). To put the constants in a wider context see for example reference [6]. This is the default reference throughout this document.

The natural logarithm is sometimes denoted $\log(x)$ and sometimes $\ln(x)$. The author prefers $\ln(x)$ which is used in this document.

A comment regarding the FEE (Fast E-function Evaluation) method for very fast evaluation of transcendental functions (series). The method was invented 1991 by Ekatherina A. Karatsuba, Russia. Another method called "divide and conquer" was invented earlier by Anatoly Karatsuba. This method was then called "binary splitting" by some people. However, by later years it seems as if "binary splitting" has been erroneously taken as a name also for the FEE method by some people preferably in the western countries. The correct name FEE is used throughout this document. See also references [1], [7] and [8].

4 Some numerical values

Here are presented numerical values of some mathematical constants to 75 decimal places.

$\pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640628 6\ldots$

$e = 2.71828182845904523536028747135266249775724709369995957496696762772407663035 3\ldots$

$\ln(2) = 0.69314718055994530941723212145817656807550013436025525412068000949339362196 9\ldots$

$\gamma = 0.57721566490153286060651209008240243104215933593992359880576723488486772677 7\ldots$

$\Gamma(1/3) = 2.67893853470774763365569294097467764412868937795730110095042832759041761016 7\ldots$

$\Gamma(1/4) = 3.62560990822190831193068515586767200299516768288006546743337799956991924353 8\ldots$

$G = 0.91596559417721901505460351493238411077414937428167213426649811962176301977 6\ldots$

$\zeta(2) = 1.64493406684822643647241516664602518921894990120679843773555822937000747040 3\ldots$

$\zeta(3) = 1.20205690315959428539973816151144999076498629234049888179227155534183820578 6\ldots$

$\zeta(5) = 1.03692775514336992633136548645703416805708091950191281197419267790380358978 6\ldots$

$\zeta(7) = 1.00834927738192282683979754984979675959986356056523870641728313657160147831 7\ldots$

$\zeta(10) = 1.00099457512781808533714595890031901700601953156447751725778899463629146515 1\ldots$

$\zeta(100) = 1.00000000000000000000000000000078886090522101180735205378276604136878962534 3\ldots$

$\sqrt{2} = 1.41421356237309504880168872420969807856967187537694807317667973799073247846 2\ldots$

$\sqrt{3} = 1.73205080756887729352744634150587236694280525381038062805580697945193301690 8\ldots$

$\sqrt{5} = 2.23606797749978969640917366873127623544061835961152572427089724541052092563 7\ldots$

$\sqrt{7} = 2.64575131106459059050161575363926042571025918308245018036833445920106882323 0\ldots$

$\frac{\sqrt{5}+1}{2} = 1.61803398874989484820458683436563811772030917980576286213544862270526046281 8\ldots$

5 Algorithms for π

5.1 π, Borwein's 4-th order

This algorithm gives four times more digits per iteration.

1. Initial value settings:

$$a_0 = 6 - 4\sqrt{2}$$

$$y_0 = \sqrt{2} - 1$$

$$s_0 = 8$$

2. Iterate the following to desired accuracy:

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{\frac{1}{4}}}{1 + (1 - y_k^4)^{\frac{1}{4}}}$$

$$a_{k+1} = a_k(1 + y_{k+1})^4 - s_k y_{k+1}(1 + y_{k+1} + y_{k+1}^2)$$

$$s_{k+1} = 4s_k$$

3. π is approximated after n iterations as:

$$\pi \approx \frac{1}{a_n}$$

5.2 π, Borwein's 2-th order

This algorithm gives two times more digits per iteration.

1. Initial value settings:

$$x_0 = \sqrt{2}$$

$$\pi_0 = 2 + \sqrt{2}$$

$$y_0 = 2^{\frac{1}{4}}$$

2. Iterate the following to desired accuracy:

$$x_{i+1} = \frac{1}{2}(\sqrt{x_i} + \frac{1}{\sqrt{x_i}})$$

$$\pi_{i+1} = \pi_i \frac{x_{i+1} + 1}{y_i + 1}$$

$$y_{i+1} = \frac{y_i \sqrt{x_{i+1}} + \frac{1}{\sqrt{x_{i+1}}}}{y_i + 1}$$

3. π is approximated after n iterations as:

$$\pi \approx \pi_n$$

5.3 π, Gauss-Legendre, AGM

This algorithm of calculating π is based on AGM (Arithmetic Geometric Mean).

This algorithm gives two times more digits per iteration.

1. Initial value settings:

$$a_0 = 1$$

$$b_0 = \frac{1}{\sqrt{2}}$$

$$t_0 = \frac{1}{4}$$

$$s_0 = 1$$

2. Iterate the following to desired accuracy:

$$a_{k+1} = \frac{a_k + b_k}{2} \quad \text{arithmetic mean}$$

$$b_{k+1} = \sqrt{a_k b_k} \quad \text{geometric mean}$$

$$t_{k+1} = t_k - s_k(a_k - a_{k+1})^2$$

$$s_{k+1} = 2s_k$$

3. π is approximated after n iterations as:

$$\pi \approx \frac{(a_n + b_n)^2}{4t_n}$$

5.4 π, Ramanujan series

In 1914 the mathematician S. Ramanujan from India published the following famous series for π:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!\,(1103 + 26390n)}{(n!)^4\,(396)^{4n}}$$

Each term of the series adds about 8 digits to π. The series can be computed very fast to desired precision using FEE.

5.5 π, Chudnovsky series

In the 1990s David and Gregory Chudnovsky found the following series (of Ramanujan type) for π:

$$\frac{1}{\pi} = \frac{12}{C\sqrt{C}} \sum_{n=0}^{\infty} \frac{(-1)^n\,(6n)!\,(A + Bn)}{(3n)!\,(n!)^3\,(C)^{3n}}$$

Here $A = 13591409$, $B = 545140134$ and $C = 640320$.

Each term of the series adds about 14 digits to π, which is remarkable. The series can be computed very fast to desired precision using FEE.

6 Algorithms for $e^x = \exp(x)$ and $e = \exp(1)$

6.1 $e^x = \exp(x)$, Newton's Iteration

$\exp(x)$ can be calculated using Newton's iteration:

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)}$$

If $f(y) = \ln(y) - x$ Newton's iteration for $\exp(x)$ is defined by:

$$y_{k+1} = y_k \cdot (1 + x - \ln(y_k))$$

Here y_0 should be a good starting value for $\exp(x)$. This algorithm gives two times more digits per iteration.

6.2 $e = \exp(1)$, using the series $\sum 1/k!$

$e = \exp(1)$ can be calculated by the series:

$$e = \exp(1) = \sum_{k=0}^{\infty} \frac{1}{k!}$$

By rearranging $\sum 1/k!$, $e = exp(1)$ can be calculated by the following algorithm (C-like notation):

```
a=1;
b=0;

for (i=n; i>=1; i--) {

    a=a*i;
    b=b+a;

} /* for i */

e=b/a;
```

Here $n!$ is the factorial that fits into the digits of the desired precision. The operations $a = a * i$, $b = b + a$ and $e = b/a$ are calculated in full precision. The numbers $i \leq n$ are quite small, which makes the computation of $a = a * i$ possible in $O(n)$ time as well as $b = b + a$. Calculation of $e = b/a$ requires Newton's iteration and FFT multiplications. This direct way of calculating $e = \exp(1)$ is now only of historical interest because of the slow $O(n^2)$ convergence.

6.3 $e = \exp(1)$, $\sum 1/k!$ using FEE

Here is a short description of the very fast FEE (Fast E-function Evaluation) method for calculating $e = \exp(1)$.

Assume we want to calculate:

$$e = \exp(1) = \sum_{k=0}^{\infty} \frac{1}{k!}$$

If we define:

$$Q(a,b) = (a+1)(a+2)\cdot \ldots \cdot b$$

and

$$P(a,b) = b(b-1)\cdot \ldots \cdot (a+2) + b(b-1)\cdot \ldots \cdot (a+3) + \ldots + (b-1)b + b + 1$$

$P(a,b)$ and $Q(a,b)$ are integers which satisfy:

$$\frac{P(a,b)}{Q(a,b)} = \frac{1}{a+1} + \frac{1}{(a+1)(a+2)} + \ldots + \frac{1}{(a+1)(a+2)\cdot \ldots \cdot b}$$

If $K!$ is the factorial that fits in to the desired number of digits, $1 + P(0,K)/Q(0,K)$ are the first K terms of the series $\sum 1/k!$.

To compute $P(0,K)$ and $Q(0,K)$ using binary splitting calculate the integer part of $(a+b)/2$:

$$m = (\text{int})(\frac{a+b}{2})$$

$P(a,b)$ and $Q(a,b)$ are then computed recursively (multi precision integer operations):

$$P(a,b) = P(a,m)Q(m,b) + P(m,b)$$

and

$$Q(a,b) = Q(a,m)Q(m,b)$$

These operations can be computed in $O(n\ \log(n)^3)$ time if FFT multiplication is used. The final (multi precision floating point) division $P(0,K)/Q(0,K)$ can be computed in $O(n\ \log(n)^2)$ time using Newton's Iteration and FFT multiplication.

7 Algorithms for $\ln(x)$ and $\ln(2)$

7.1 $\ln(x)$, AGM

This algorithm of calculating $\ln(x)$ is based on AGM (Arithmetic Geometric Mean). The algorithm gives N digits of $\ln(x)$.

This algorithm gives two times more digits per iteration.

Define the iterative procedure $R(y)$ as:

1. Initial value settings:

$$a_0 = 1$$

$$b_0 = \frac{1}{y} 10^{-N/2}$$

$$t_0 = 0$$

$$s_0 = 1$$

2. Iterate the following to desired accuracy:

$$a_{k+1} = \frac{a_k + b_k}{2} \quad \text{arithmetic mean}$$

$$b_{k+1} = \sqrt{a_k b_k} \quad \text{geometric mean}$$

$$t_{k+1} = t_k + s_k(a_{k+1}^2 - b_{k+1}^2)$$

$$s_{k+1} = 2s_k$$

3. $R(y)$ is approximated after n iterations as:

$$R(y) = \frac{1}{\frac{1}{2} - t_n}$$

4. Finally $\ln(x)$ is approximated as:

$$\ln(x) \approx R(x) - R(1)$$

especially:

$$\ln(2) \approx R(2) - R(1)$$

7.2 $\ln(2)$, $\operatorname{arctanh}(x)$ based methods

The constant $\ln(2)$ can be calculated by many $\operatorname{arctanh}(x)$ based series.

The simplest is:

$$\ln(2) = 2\operatorname{arctanh}\left(\frac{1}{3}\right)$$

or one with two terms:

$$\ln(2) = 2\operatorname{arctanh}\left(\frac{1}{5}\right) + 2\operatorname{arctanh}\left(\frac{1}{7}\right)$$

For numerical calculations the following series are faster:

$$\ln(2) = 18\operatorname{arctanh}\left(\frac{1}{26}\right) - 2\operatorname{arctanh}\left(\frac{1}{4801}\right) + 8\operatorname{arctanh}\left(\frac{1}{8749}\right)$$

$$\ln(2) = 144\operatorname{arctanh}\left(\frac{1}{251}\right) + 54\operatorname{arctanh}\left(\frac{1}{449}\right) - 38\operatorname{arctanh}\left(\frac{1}{4801}\right) + 62\operatorname{arctanh}\left(\frac{1}{8749}\right)$$

$$\ln(2) = 72\operatorname{arctanh}\left(\frac{1}{127}\right) + 54\operatorname{arctanh}\left(\frac{1}{449}\right) + 34\operatorname{arctanh}\left(\frac{1}{4801}\right) - 10\operatorname{arctanh}\left(\frac{1}{8749}\right)$$

Each iteration of these series adds about 2.8, 4.8 and 4.2 digits to $\ln(2)$ respectively.

By a "brute force" computer search the author found the following quite fast series:

$$\ln(2) = 6\operatorname{arctanh}\left(\frac{1}{26}\right) + 8\operatorname{arctanh}\left(\frac{1}{31}\right) + 10\operatorname{arctanh}\left(\frac{1}{49}\right)$$

$$\ln(2) = 14\operatorname{arctanh}\left(\frac{1}{31}\right) + 10\operatorname{arctanh}\left(\frac{1}{49}\right) + 6\operatorname{arctanh}\left(\frac{1}{161}\right)$$

Each iteration of these series adds about 2.8 and 3.0 digits to $\ln(2)$ respectively.

In the series above $\operatorname{arctanh}(x)$ is given by:

$$\operatorname{arctanh}(x) = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right) = \sum_{k=0}^{\infty}\frac{x^{2k+1}}{2k+1}$$

The series can be computed very fast to desired precision using FEE (for rational x).

8 Algorithms for Euler C, γ

Euler C, γ is defined as:

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln(n)\right)$$

However a direct computation is very slow.

The constant can be calculated much faster by a method based on modified Bessel functions:

$$\gamma = \frac{A_N}{B_N} - \ln(N) + O(e^{-4N})$$

here:

$$A_N = \sum_{n=0}^{\alpha N} \left(\frac{N^n}{n!}\right)^2 H_n \quad ; \quad B_N = \sum_{n=0}^{\alpha N} \left(\frac{N^n}{n!}\right)^2 \quad ; \quad H_n = \sum_{k=1}^{n} \frac{1}{k}$$

with $\alpha = 3.5911...$ satisfies $\alpha(\ln(\alpha) - 1) = 1$. The series can be computed very fast to desired precision using FEE.

By taking into account the error term C_N the following (even faster) method can be used:

$$\gamma = \frac{A_N}{B_N} - \frac{C_N}{B_N^2} - \ln(N) + O(e^{-8N})$$

here:

$$C_N = \frac{1}{4N} \sum_{n=0}^{N} \frac{[(2n)!]^3}{(n!)^4 \, (16N)^{2n}}$$

This time the summation of A_N and B_N should go up to βN where $\beta = 4.9706...$ satisfies $\beta(\ln(\beta) - 1) = 3$. The series can be computed very fast to desired precision using FEE.

9 Algorithms for $\Gamma(1/3)$ and $\Gamma(1/4)$

9.1 $\Gamma(1/3)$

$\Gamma(1/3)$ can be calculated by the formula:

$$\Gamma(1/3) = \frac{\pi^{\frac{2}{3}} \, 2^{\frac{4}{9}} \, 3^{\frac{3}{4}}}{3 \, \mathrm{agm}(1, v)^{\frac{1}{3}}}$$

$$v = \frac{\sqrt{3} - 1}{2\sqrt{2}}$$

9.2 $\Gamma(1/4)$

$\Gamma(1/4)$ can be calculated by the formula:

$$\Gamma(1/4) = \frac{(2\pi)^{\frac{3}{4}}}{\mathrm{agm}(1, v)^{\frac{1}{2}}}$$

$$v = \sqrt{2}$$

9.3 AGM

Here $\mathrm{agm}(a, b)$ means AGM (Arithmetic Geometric Mean) of a and b and is defined by the following iteration to desired precision. This algorithm gives two times more digits per iteration.

$$a_{k+1} = \frac{a_k + b_k}{2} \quad \text{arithmetic mean}$$

$$b_{k+1} = \sqrt{a_k b_k} \quad \text{geometric mean}$$

10 Algorithms for Catalan's constant G

Catalan's constant G is defined as:

$$G = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}$$

However a direct computation is very slow.

The constant can be calculated much faster by accelerated methods. Lupas, see reference [5]:

$$G = \frac{1}{64} \sum_{n=1}^{\infty} \frac{(-1)^{n-1}\, 2^{8n}\, (40n^2 - 24n + 3)\, [(2n)!]^3\, (n!)^2}{n^3\, (2n-1)\, [(4n)!]^2}$$

or, see reference [12]:

$$G = 3 \sum_{n=0}^{\infty} \frac{1}{2^{4n}} \left(-\frac{1}{2(8n+2)^2} + \frac{1}{2^2(8n+3)^2} - \frac{1}{2^3(8n+5)^2} + \frac{1}{2^3(8n+6)^2} - \frac{1}{2^4(8n+7)^2} + \frac{1}{2(8n+1)^2} \right) -$$

$$2 \sum_{n=0}^{\infty} \frac{1}{2^{12n}} \left(\frac{1}{2^4(8n+2)^2} + \frac{1}{2^6(8n+3)^2} - \frac{1}{2^9(8n+5)^2} - \frac{1}{2^{10}(8n+6)^2} - \frac{1}{2^{12}(8n+7)^2} + \frac{1}{2^3(8n+1)^2} \right)$$

Each iteration of these series adds about 0.60 and 1.20 digits to G respectively. The series can be computed very fast to desired precision using FEE.

11 Algorithms for Apery's constant $\zeta(3)$

Apery's constant $\zeta(3)$ is defined as a special case of the Riemann Zeta function for $s = 3$:

$$\zeta(3) = \sum_{n=1}^{\infty} \frac{1}{n^3} \quad ; \quad \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

However a direct computation is very slow.

The constant can be calculated much faster by accelerated methods.

$$\zeta(3) = \frac{1}{64} \sum_{n=0}^{\infty} \frac{(-1)^n\, (205n^2 + 250n + 77)\, (n!)^{10}}{[(2n+1)!]^5}$$

or:

$$\zeta(3) = \frac{1}{24} \sum_{n=0}^{\infty} \frac{(-1)^n\, A(n)\, [(2n+1)!\, (2n)!\, n!]^3}{(3n+2)!\, [(4n+3)!]^3}$$

here:

$$A(n) = 126392n^5 + 412708n^4 + 531578n^3 + 336367n^2 + 104000n + 12463$$

Each term of these series adds about 3.0 and 5.0 digits to $\zeta(3)$ respectively. The series can be computed very fast to desired precision using FEE.

12 Algorithms for $\zeta(s)$

The Riemann Zeta function $\zeta(s)$ is defined by:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \frac{2^{s-1}}{2^{s-1}-1} \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n^s}$$

Below are described two accelerated methods to calculate $\zeta(s)$ (for integer $s \geq 2$).

Method 1 use the alternating series with Cohen-Villegas-Zaiger convergence acceleration and FEE. The algorithm is outlined below without FEE (C-like notation). The calculations are performed with full precision (i.e. `dec` digits). See references [4] and [9] for more details.

```
N=(int)(1.5728794*dec)+1;

fterm=2*N*N;
fsum=fterm;
gterm=fterm;
gsum=fterm;

for (n=1; n < N; n++) {

   temp=(2*(N-n)*(N+n)) / ((2*n+1)*(n+1));

   fterm=fterm*temp;

   fsum=fsum+fterm;

   gterm=gterm*temp;

   gterm=gterm+fterm / (((-1)^(n-1))*((n+1)^s));

   gsum=gsum+gterm;

} /* for n */

gsum=gsum / (fsum+1);

gsum=(2^(s-1))*gsum / (2^(s-1)-1);

zeta=gsum;
```

Method 2 use the alternating series with acceleration through $\exp(x)$ and FEE. The algorithm is outlined below without FEE (C-like notation). The calculations are performed with full precision (i.e. `dec` digits). See references [4] and [9] for more details.

```
x=(int)(2.772592*dec)+1;
N=(int)(exp(1)*x);

fterm=1;
fsum=fterm;
gterm=fterm;
gsum=fterm;

for (n=1; n < N; n++) {

   temp=x / n;

   fterm=fterm*temp;

   fsum=fsum+fterm;

   gterm=gterm*temp;

   gterm=gterm+fterm / (((-1)^(n-1))*((n+1)^s));

   gsum=gsum+gterm;

} /* for n */

gsum=gsum / (fsum+1);

gsum=(2^(s-1))*gsum / (2^(s-1)-1);

zeta=gsum;
```

13 Algorithms for inversion, division, square and cube roots

This section describes iterative high order algorithms for calculating $1/v$, $1/\sqrt{v}$, $1/v^{\frac{1}{3}}$ and $1/v^{\frac{1}{4}}$. The iteration should always have a good starting value.

13.1 Inversion $1/v$ and division

Algorithm 1, (Newton's iteration). Requires two full precision multiplications. This algorithm gives two times more digits per iteration.

$$h = 1 - vx_k$$

$$x_{k+1} = x_k + x_k h$$

Algorithm 2, (cubic convergence). Requires three full precision multiplications. This algorithm gives three times more digits per iteration.

$$h = 1 - vx_k$$

$$x_{k+1} = x_k + x_k(h + h^2)$$

Algorithm 3, (quartic convergence). Used for the programs in this package. Requires four full precision multiplications. This algorithm gives four times more digits per iteration.

$$h = 1 - vx_k$$

$$x_{k+1} = x_k + x_k(h + h^2 + h^3)$$

Division $q = w/v$ can be calculated as:

$$q = \frac{1}{v}w$$

13.2 $1/\sqrt{v}$ and $\sqrt{v}$

Algorithm 1, (Newton's iteration). Requires three full precision multiplications. This algorithm gives two times more digits per iteration.

$$h = 1 - vx_k^2$$

$$x_{k+1} = x_k + \frac{x_k}{2}h$$

Algorithm 2, (cubic convergence). Requires four full precision multiplications. This algorithm gives three times more digits per iteration.

$$h = vx_k^2$$

$$x_{k+1} = \frac{x_k}{8}(15 + h(-10 + 3h))$$

Algorithm 3, (quartic convergence). Requires five full precision multiplications. This algorithm gives four times more digits per iteration.

$$h = vx_k^2$$

$$x_{k+1} = \frac{x_k}{16}(35 + h(-35 + h(21 - 5h)))$$

A more efficient way to write this is (used for the programs in this package):

$$h = 1 - vx_k^2$$

$$x_{k+1} = x_k + \frac{x_k}{16}(8h + 6h^2 + 5h^3)$$

$\sqrt{v}$ can then be calculated as:

$$\sqrt{v} \approx vx_n$$

13.3 $1/v^{\frac{1}{3}}$ and $v^{\frac{1}{3}}$

Algorithm 1, (Newton's iteration). Requires four full precision multiplications. This algorithm gives two times more digits per iteration.

$$h = 1 - vx_k^3$$

$$x_{k+1} = x_k + \frac{x_k}{3}h$$

Algorithm 2, (quartic convergence). Used for for the programs in this package. Requires six full precision multiplications. This algorithm gives four times more digits per iteration.

$$h = 1 - vx_k^3$$

$$x_{k+1} = x_k + \frac{x_k}{162}(54h + 36h^2 + 28h^3)$$

$v^{\frac{1}{3}}$ can then be calculated as:

$$v^{\frac{1}{3}} \approx vx_n^2$$

13.4 $1/v^{\frac{1}{4}}$ and $v^{\frac{1}{4}}$

Algorithm 1, (Newton's iteration). Requires four full precision multiplications. This algorithm gives two times more digits per iteration.

$$h = 1 - vx_k^4$$

$$x_{k+1} = x_k + \frac{x_k}{4}h$$

Algorithm 2, (quartic convergence). Requires six full precision multiplications. This algorithm gives four times more digits per iteration.

$$h = vx_k^4$$

$$x_{k+1} = \frac{x_k}{128}(195 + h(-117 + h(65 - 15h)))$$

A more efficient way to write this is (used for the programs in this package):

$$h = 1 - vx_k^4$$

$$x_{k+1} = x_k + \frac{x_k}{128}(32h + 20h^2 + 15h^3)$$

$v^{\frac{1}{4}}$ can then be calculated as:

$$v^{\frac{1}{4}} \approx vx_n^3$$

14 Time complexity of algorithms

Full precision calculations with n digits can be performed for some different algorithms with the following time complexity. Perhaps some factors $\text{loglog}(n)$ are missing. NI means Newton's Iteration (or higher order methods). FEE means Fast E-function Evaluation. Small multiplication means a multiplication between a full precision number and a small number that can be stored in a normal (for example integer) variable.

Algorithm	Time complexity
Addition	$O(n)$
Subtraction	$O(n)$
Small multiplication	$O(n)$
FFT multiplication	$O(n\ \log(n))$
Inversion, NI	$O(n\ \log(n)^2)$
Division, NI	$O(n\ \log(n)^2)$
$\sqrt{x}$, NI	$O(n\ \log(n)^2)$
$x^{\frac{1}{3}}$, NI	$O(n\ \log(n)^2)$
$x^{\frac{1}{4}}$, NI	$O(n\ \log(n)^2)$
π Borwein's 4-th order	$O(n\ \log(n)^3)$
π Borwein's 2-th order	$O(n\ \log(n)^3)$
π Gauss Legendre, AGM	$O(n\ \log(n)^3)$
π Chudnovsky series, FEE	$O(n\ \log(n)^3)$
π Ramanujan series, FEE	$O(n\ \log(n)^3)$
$\exp(1)$, $e = \sum 1/k!$, FEE	$O(n\ \log(n)^3)$
$\exp(1)$, $1/e = \sum(-1)^k/k!$, FEE	$O(n\ \log(n)^3)$
$\exp(x)$, NI (using $\ln(x)$ AGM)	$O(n\ \log(n)^4)$
$\exp(1)$, $e = \sum 1/k!$, direct computation	$O(n^2)$
$\ln(x)$, AGM	$O(n\ \log(n)^3)$
$\ln(2)$, $\text{arctanh}(x)$ based methods, FEE	$O(n\ \log(n)^3)$
Euler C, γ, FEE	$O(n\ \log(n)^3)$
$\Gamma(1/3)$, AGM	$O(n\ \log(n)^3)$
$\Gamma(1/4)$, AGM	$O(n\ \log(n)^3)$
Catalan, G, FEE	$O(n\ \log(n)^3)$
Apery's, $\zeta(3)$, FEE	$O(n\ \log(n)^3)$
$\zeta(s)$, FEE	$O(n\ \log(n)^3)$

15 Accuracy and benchmarks

15.1 Accuracy of calculations

The programs in this package calculate the mathematical constants below to many decimal places. All printed digits are supposed to be correct. The programs use a very fast FFT (fftsg_h.c) by Takuya Ooura (see reference [10]).

Warning! If you want to calculate more than 2^{24} (about 16 million) digits set the constant mulversion = 2 in the file "mulver.txt" to avoid errors in the FFT multiplication. At least this must be done if "FFT max error" > 0.25. This makes the programs two times slower and requires more memory.

15.2 Benchmarks

Some other good web-pages about benchmarks of mathematical constants are found in references [13] and [14].

In these benchmarks the FFT multiplication variable (mulversion) was set to 1 (i.e. fastest method).

15.2.1 Information about the computer used for benchmarks

The CPU-times are given for an Intel Celeron computer at 1400 MHz measured with the "time" function in Linux. The computer has 512 Mbyte SDRAM memory and 100 MHz memory bus. The cache memory is 256 kbyte (Advanced Transfer) at full CPU speed. The operating system was Red Hat 9 Linux with the gcc 3.2.2 C-compiler. The CPU load was on average 1.00 (i.e. no other programs running). The programs were compiled using the line:

>gcc -O3 -o program program.c fftsg_h.c -lm

15.2.2 Benchmarks with the program *sspi*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	CPU-time 2^{24} digits	Max memory per 2^{20} digits
π, Chudnovsky	2^{24}	3.78 s	57.26 s	1533.77 s	13.4 Mbyte
π, Ramanujan	2^{24}	5.81 s	72.26 s	2449.46 s	13.4 Mbyte

15.2.3 Benchmarks with the program *sseln2*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
$e = \exp(1)$, method 1	2^{20}	1.99 s	21.04 s	12.8 Mbyte
$e = \exp(1)$, method 2	2^{20}	2.38 s	26.45 s	12.8 Mbyte
$\ln(2)$, method 3	2^{20}	19.17 s	253.59 s	16.0 Mbyte
$\ln(2)$, method 4	2^{20}	21.44 s	276.08 s	16.0 Mbyte
$\ln(2)$, method 5	2^{20}	23.51 s	309.69 s	16.0 Mbyte
$\ln(2)$, method 6	2^{20}	19.54 s	244.00 s	16.0 Mbyte
$\ln(2)$, method 7	2^{20}	19.61 s	259.85 s	16.0 Mbyte

15.2.4 Benchmarks with the program *ssgamma*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
Euler C, γ, method 1	2^{20}	119.19 s	1607.36 s	36.4 Mbyte
Euler C, γ, method 2	2^{20}	104.01 s	1561.67 s	36.4 Mbyte

15.2.5 Benchmarks with the program *ssgam134*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
$\Gamma(1/3)$	2^{20}	34.06 s	433.33 s	14.9 Mbyte
$\Gamma(1/4)$	2^{20}	26.44 s	347.41 s	14.4 Mbyte

15.2.6 Benchmarks with the program *sscatal*

Note, this program was run under Fedora Core 1 Linux and the gcc 3.3.2 compiler.

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
Catalan G, method 1	2^{20}	188.43 s	2948.54 s	29.6 Mbyte
Catalan G, method 2	2^{20}	241.99 s	3431.92 s	19.7 Mbyte

15.2.7 Benchmarks with the program *sszeta3*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
Apery's $\zeta(3)$, method 1	2^{20}	21.97 s	386.66 s	19.7 Mbyte
Apery's $\zeta(3)$, method 2	2^{20}	17.29 s	341.97 s	21.8 Mbyte

15.2.8 Benchmarks with the program *sszeta*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
$\zeta(3)$, method 1	2^{20}	121.19 s	1739.45 s	36.4 Mbyte
$\zeta(3)$, method 2	2^{20}	494.86 s	6825.30 s	36.4 Mbyte
$\zeta(10)$, method 1	2^{20}	227.76 s	3328.41 s	36.4 Mbyte
$\zeta(10)$, method 2	2^{20}	942.43 s	16713.07 s	36.4 Mbyte
$\zeta(100)$, method 1	2^{20}	1309.66 s	19457.07 s	36.4 Mbyte
$\zeta(100)$, method 2	2^{20}	8060.97 s	— s	36.4 Mbyte

15.2.9 Benchmarks with the program *sspieln2*

Intel Celeron 1400 MHz Algorithm	Digits verified	CPU-time 2^{17} digits	CPU-time 2^{20} digits	Max memory per 2^{20} digits
π, Borwein's 4-th order	2^{20}	26.89 s	382.24 s	13.4 Mbyte
π, Borwein's 2-th order	2^{20}	32.91 s	372.56 s	14.9 Mbyte
π, Gauss Legendre, AGM	2^{20}	22.92 s	289.99 s	13.4 Mbyte
$\exp(1)$, $e = \sum 1/k!$	2^{20}	24.87 s	1922.04 s	12.8 Mbyte
$\exp(1)$, Newton's Iteration	2^{20}	731.08 s	— s	33.8 Mbyte
$\ln(2)$, AGM	2^{20}	102.31 s	1287.43 s	30.8 Mbyte
Golden Section= $(\sqrt{5}+1)/2$	2^{20}	1.11 s	12.11 s	11.8 Mbyte
$\sqrt{2}$	2^{20}	0.99 s	9.45 s	11.3 Mbyte

16 References

16.1 Book and article references

[**1**] Ekatherina A. Karatsuba, "Fast evaluation of transcendental functions", *Problems of Information Transmission*, vol. 27, (1991), p. 339-360.

[**2**] "Pi and the AGM, *A Study in Analytic Number Theory and Computational Complexity*", Volume 4, by Jonathan M. Borwein and Peter B. Borwein, 1987. A Wiley-Interscience Publication, JOHN WILEY & SON INC.

[**3**] "Fast multiprecision evaluation of series of rational numbers", by Bruno Haible and Thomas Papanikolaou, 1997.

[**4**] "Convergence Acceleration of Alternating Series", by H. Cohen, F. R. Villegas and D. Zaiger, Bonn, (1991).

[**5**] "Formulae for some classical constants", (to appear in Proceedings of ROGER-2000), by Alexandru Lupas.

16.2 Internet references

The iterative (high order) formulas for calculating $1/\sqrt{v}$ and $1/v^{\frac{1}{4}}$ and the rearranged method to calculate $e = \exp(1) = \sum 1/k!$ were derived by the author of this document. The home page of the author and the web page of the program package and this document are found here:

http://www.spaennare.se/index.html

http://www.spaennare.se/ssprog.html

Most other information is found on Internet. Many people are calculating different mathematical constants (mostly π) to many decimal places just for fun or scientific purposes. Different algorithms for calculating the constants can be found on many places on Internet.

[**6**] "Mathematical constants and computation", by Xavier Gourdon and Pascal Sebah. Here is described almost everything you want to know about mathematical constants and how to compute them. One of the fastest programs for calculating π and other constants on a PC is also found here.

http://numbers.computation.free.fr/Constants/constants.html

[**7**] "Fast Algorithms and the FEE Method", by Ekatherina A. Karatsuba.

http://www.ccas.ru/personal/karatsuba/algen.htm

[**8**] "The Karatsuba Method 'Divide and Conquer'", by Ekatherina A. Karatsuba.

http://www.ccas.ru/personal/karatsuba/divcen.htm

[**9**] "CLN - Class Library for Numbers", by Bruno Haible et. al.

http://www.ginac.de/CLN/

[**10**] "Ooura's Mathematical Software Packages", by Takuya Ooura. Among other things many fast FFTs written in C and Fortran.

http://www.kurims.kyoto-u.ac.jp/~ooura/

[**11**] "Plouffe's Inverter", by Simon Plouffe. Some interesting digit records. Files containing different numbers to many decimal places can also be found here. These files can be used for verification purposes.

http://pi.lacim.uqam.ca/eng/

[**12**] "The Wolfram Functions Site". Created with Mathematica by Wolfram Research. Very good information about mathematics.

http://functions.wolfram.com

[**13**] "Dara's π Pages". Comparison of different π calculating programs etc.

http://www.myownlittleworld.com/miscellaneous/computers/pilargetable.html

[**14**] "Stu's Pi page". Record page for the fastest π programs for PC. The record holder can also be downloaded here.

http://home.istar.ca/~lyster/pi.html