A Java Math.BigDecimal Implementation of Core Mathematical Functions

Richard J. Mathar*
Leiden Observatory, Leiden University, P.O. Box 9513, 2300 RA Leiden, The Netherlands
(Dated: August 20, 2009)

The mathematical functions $\log(x)$, $\exp(x)$, $\sqrt[n]{x}$, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arctan(x)$, x^y, $\sinh(x)$, $\cosh(x)$, $\tanh(x)$ and $\Gamma(x)$ have been implemented for arguments x in the real domain in a native Java library on top of the multi-precision `BigDecimal` representation of floating point numbers. This supports scientific applications where more than the double precision accuracy of the library of the Standard Edition is desired. The full source code is made available.

PACS numbers: 02.30.Gp, 02.30.Mv, 02.60.Gf
Keywords: Java library, BigDecimal, mathematical functions, logarithm, exponential, trigonometric

I. OVERVIEW

A. Aim

Whereas many Java applications can use the Java Native Interface to bind to (C-based) multi-precision programs on a *host* platform when a higher precision than the 64-bit standard is needed [5], others may observe that there is only rudimentary support on the *native* platform if standard mathematical functions are needed.

The aim of this script is to provide a base implementation of the core trigonometric and algebraic functions on top of the native `BigDecimal` class with infinite-precision capability. Demand originates from scientific and perhaps engineering computations, where accumulation of rounding errors (loss of digits) might pose a problem.

B. Design Choices

A characteristic feature of the implementation suggested here is that the floating point variables which are arguments to mathematical functions define by their number of digits which precision is achieved in the result. The estimate of the accuracy of the result is generally derived from the first order Taylor approximation of the function in question based on the accuracy of the input variable. The number of digits of the values returned will be larger than the number of digits of the variable where the function is flat—for example the $\arctan(x)$ where $x \gg 1$—, smaller where it is steep. This is a deliberate difference to most CAS systems. It provides some semi-automate detection of loss-of-precision through cancellation of digits, and it reduces some burden to the application programmer to decide on a `MathContext` interface prior to each individual call.

This is backed by some `xxxRound` functions where `xxx` are the fundamental `add`, `subtract`, `multiply`, `divide` operations, etc., which internally calculate estimators of the precision of the result based on the precisions of their arguments. The mathematical functions are typically some power series expansions, and they make heavy use of these to keep the error accumulation of the individual terms under control, that is, to chop off digits early to keep execution times short where intermediate results are known to be dominated by noise in the parameters.

As a side effect, the number of digits returned may be even smaller than the characteritic 6 digits of a single-precision calculation. In addition, the results depend on the number of trailing zeros of the inputs. (A function `scalePrec` is provided to boost the apparent accuracy of numbers by appending zeros.)

C. Known Limitations

As presented, the implementation is known to have jitters of 1 or 2 in the least significant digits in some values returned.

The algorithms have been chosen for reliability and simplicity, and may be slower than alternatives which have not been investigated.

Classes of important special functions (polynomials, Bessel functions, Elliptic Integrals, ...) and complex arithmetics are absent.

II. IMPLEMENTATION STRATEGIES

A. Constants

The heavy-duty constants π, e, $\ln 2$ and γ are tabulated to high precision which presumably suffices for most purposes in engineering and sciences. The backup implementations for applications in some areas of mathematics are:

- π is evaluated by Broadhurst's equation (18) [2]. I once read the argument that the ratio of the radius of the known universe, 14 billion light years, over the Planck length is approximately 8×10^{60}, so π could not be required to a precision of more than 61 digits anywhere in the natural sciences.

*URL: `http://www.strw.leidenuniv.nl/~mathar`; Electronic address: mathar@strw.leidenuniv.nl

- $\log 2$ is evaluated by Broadhurst's equation (21) [2] finalized by pulling a square root.
- γ is generated by the series [4, (3.9)]

$$\gamma = 1 - \log\frac{3}{2} - \sum_{n\ge 1}\frac{\zeta(2n+1)-1}{4^n(2n+1)}. \tag{1}$$

- e is forwarded to the generic evaluation of $\exp(1)$, Section II C.

B. Roots

The roots $y = \sqrt[n]{x}$ for positive integer n, including the special case of square roots $n = 2$, are computed by iterative updates with the first order Newton method [1, (3.96)]

$$y \to y - \frac{1}{n}\left(y - \frac{x}{y^{n-1}}\right). \tag{2}$$

The initial estimates of y are set in double precision by a call to the `Math.pow`.

The `hypot` function computes

$$z = \sqrt{x^2+y^2} \tag{3}$$

from two arguments x and y. A derived case has been implemented taking an integer value x, because this implies that the precision of the result z is determined from the precision in y alone.

C. Exponential

If x is close to zero, the standard Taylor series [1, (4.2.1)]

$$\exp(x) = \sum_{k\ge 0}\frac{x^k}{k!} \tag{4}$$

is employed. For larger x, x is scaled by powers of 10

$$e^x = \left(e^{10^{-t}x}\right)^{10^t} \tag{5}$$

such that the value in parenthesis can be evaluated by recourse to (4). Scaling by powers of 10 is a cheap operation in the `BigDecimal` library because it only involves a diminuation of the scale. Powers with integer exponents are also more efficient than one might naïvely expect. The 10th power needs 4 multiplications, for example; see [7] and sequence A003313 in the OEIS [8].

The general powers are forwarded to a mixed call of the `log` and `exp` functions,

$$x^y = \exp(y\log x). \tag{6}$$

D. Logarithm

For arguments close to 1, the standard Taylor expansion [1, (4.1.24)]

$$\log(1+x) = \sum_{k\ge 1}(-1)^{j+1}\frac{x^k}{k} \tag{7}$$

is used. For larger x, adaptation to that range is achieved by scaling with some integer r

$$\log x = r\log\sqrt[r]{x} \tag{8}$$

with an auxiliary call to the `root` function of Section II B. The variable r is obtained by a call to the `Math.log` of the native library.

For some integer arguments, dedicated routines are implemented assuming that $\ln 2$ is instantly available,

$$12\ln 3 = 19\ln 2 + \sum_{k\ge 1}\frac{(-1)^{k+1}}{k}\left(\frac{7153}{524288}\right)^k, \tag{9}$$

$$6\ln 5 = 14\ln 2 - \sum_{k\ge 1}\frac{1}{k}\left(\frac{759}{16384}\right)^k, \tag{10}$$

$$\ln 7 = 3\ln 2 - \sum_{k\ge 1}\frac{1}{k8^k}. \tag{11}$$

These have practically no speed advantage compared to the alternative of adding zeros and handling them with the generic procedure described above.

E. Trigonometric

The arguments of sin, cos and tan are reduced to the fundamental domain modulo 2π or modulo π, then folded with standard shifting equations, Table 4.3.44 in the Handbook [1] into regions where the basic Taylor series converge well. These are in particular

$$\sin x = \sum_{k\ge 0}(-1)^k\frac{x^{2k+1}}{(2k+1)!}, \tag{12}$$

$$\cos x = \sum_{k\ge 0}(-1)^k\frac{x^{2k}}{(2k)!} \tag{13}$$

for $x < \pi/4$, and

$$\tan x = \sum_{k\ge 1}(-1)^{k+1}\frac{4^k(4^k-1)}{(2k)!}B_{2k}x^{2k-1} \tag{14}$$

for $x < 0.8$ [1, (4.3.67)]. The $\tan x$ is forwarded to a similar expansion of $\cot x$ [1, (4.3.70)] if $x > 0.8$.

F. Inverse Trigonometric

The arcsin is implemented as [1, (4.4.40)]

$$\arcsin x = \sum_{k\ge 0} \frac{(2k-1)!!}{(2k)!!(2k+1)} x^{2k+1} \tag{15}$$

where $x < 0.7$, and as the complementary [1, (4.4.41)] where $0.7 < x < 1$. The arctan is implemented by the standard Taylor expansions

$$\arctan x = \sum_{k\ge 0} (-1)^k \frac{x^{2k+1}}{2k+1}, \quad x < 0.7 \tag{16}$$

$$\arctan x = \frac{\pi}{2} - \sum_{k\ge 0} (-1)^k \frac{1}{(2k+1)x^{2k+1}}, \quad x > 3. \tag{17}$$

The intermediate cases are mapped to the region $x < 0.7$ by reverse application of [1, (4.4.34)]

$$2\arctan x = \arctan \frac{2x}{1-x^2} \tag{18}$$

at the cost of one additional square root.

G. Hyperbolic

The hyperbolic functions sinh and cosh are evaluated by their power series [1, (4.5.62),(4.5.63)] if the argument is close to zero, otherwise transformed by the multi-angle formulas. The tanh is implemented as

$$\tanh x = \frac{1-\exp(-2x)}{1+\exp(-2x)}. \tag{19}$$

The inverse hyperbolic functions are mapped to their logarithmic representations.

H. Gamma Function

The Γ function is reduced to the region near $x = 1$ by its functional equation

$$x\Gamma(x) = \Gamma(x+1), \tag{20}$$

and then expanded with [1, (6.1.33)]

$$\ln\Gamma(1+x) = -\ln(1+x) + x(1-\gamma) + \sum_{k\ge 2} (-1)^k \frac{\zeta(k)-1}{k} x^k. \tag{21}$$

This bypasses difficulties of regulating the errors in the Stirling formula [9, 10], but needs a rather costly evaluation of the ζ function. For even indices we implement [1, (23.2.16)]

$$\zeta(2n) = \frac{(2\pi)^{2n}}{2(2n)!} |B_{2n}|, \tag{22}$$

for indices 3 or 5 the Broadhurst expansions [2], and for odd arguments ≥ 7 [3]

$$\zeta(n) = \frac{(2\pi)^n}{n-1} \sum_{k=0}^{(n+1)/2} (-1)^k (1-2k) \frac{B_{2k}B_{n+1-2k}}{(2k)!(n+1-2k)!} - 2\sum_{k\ge 1} \frac{1}{k^n(e^{2\pi k}-1)} \left(1 + \frac{2\pi k \epsilon_n}{1-e^{-2\pi k}}\right), \tag{23}$$

where

$$\epsilon_n = \begin{cases} 0, & n \equiv 3 \pmod 4, \\ 2/(n-1), & n \equiv 1 \pmod 4. \end{cases} \tag{24}$$

III. AUXILIARY CLASSES

The aim to delay rounding of rational numbers leads to the auxiliary implementation of a `Rational` data type which consists of a signed numerator and an unsigned denominator, both of the `BigInteger` type. The basic operations of multiplication, division, addition and raising to an integer power are all exact in that class, and also some integer roots if numerator and denominator are perfect powers.

The class `Bernoulli` creates a special instance of these rational numbers, the Bernoulli numbers which are helpful in (14) and (22). From a short initial table at small indices [1, (Tab 23.2)], values at larger indices are generated by a double sum [6, (1)]:

$$B_n = \sum_{k=0}^{n} \frac{1}{k} \sum_{j=0}^{k} (-1)^j j^n \binom{k}{j}. \tag{25}$$

IV. SUMMARY

The most frequently used mathematical function with arguments and return values of the multi-precision `BigDecimal` type are presented in a Java library. Control over the variable requirements in precision is basically achieved by recourse to simple algorithms that allow semi-analytical estimations of the propagation of errors.

APPENDIX A: RATIONAL.JAVA

```
package org.nevec.rjm ;

import java.util.* ;
import java.math.* ;

/** Fractions (rational numbers).
* They are divisions of two BigInteger variables, reduced to greatest
* common divisors of 1.
*/
class Rational implements Cloneable
{
        /** numerator
        */
        BigInteger a ;

        /** denominator
        */
        BigInteger b ;

        /* The maximum and minimum value of a standard Java integer, 2^31.
        */
        static BigInteger MAX_INT = new BigInteger("2147483647") ;
        static BigInteger MIN_INT = new BigInteger("-2147483648") ;
        static Rational ONE = new Rational(1,1) ;
        static Rational ZERO = new Rational() ;

        /** Default ctor, which represents the zero.
        */
        public Rational()
        {
                a = BigInteger.ZERO ;
                b = BigInteger.ONE ;
        }

        /** ctor from a numerator and denominator.
        * @param a the numerator.
        * @param b the denominator.
        */
        public Rational(BigInteger a, BigInteger b)
        {
                this.a = a ;
                this.b = b ;
                normalize() ;
        }

        /** ctor from a numerator.
        * @param a the BigInteger.
        */
        public Rational(BigInteger a)
        {
                this.a = a ;
                b = new BigInteger("1") ;
        }

        /** ctor from a numerator and denominator.
        * @param a the numerator.
        * @param b the denominator.
        */
        public Rational(int a, int b)
        {
                this(new BigInteger(""+a),new BigInteger(""+b)) ;
        }

        /** ctor from a string representation.
        * @param str the string.
        *   This either has a slash in it, separating two integers, or, if there is no slash,
        *   is representing the numerator with implicit denominator equal to 1.
        * @warning this does not yet test for a denominator equal to zero
        */
        public Rational(String str) throws NumberFormatException
        {
                this(str,10) ;
        }

        /** ctor from a string representation in a specified base.
        * @param str the string.
        *   This either has a slash in it, separating two integers, or, if there is no slash,
        *   is just representing the numerator.
        * @param radix the number base for numerator and denominator
        * @warning this does not yet test for a denominator equal to zero
```

```
*/
public Rational(String str, int radix) throws NumberFormatException
{
        int hasslah = str.indexOf("/") ;
        if ( hasslah == -1 )
        {
                a = new BigInteger(str,radix) ;
                b = new BigInteger("1",radix) ;
                /* no normalization necessary here */
        }
        else
        {
                /* create numerator and denominator separately
                */
                a = new BigInteger(str.substring(0,hasslah),radix) ;
                b = new BigInteger(str.substring(hasslah+1),radix) ;
                normalize() ;
        }
}

/** Create a copy.
*/
public Rational clone()
{
        /* protected access means this does not work
        * return new Rational(a.clone(), b.clone()) ;
        */
        BigInteger aclon = new BigInteger(""+a) ;
        BigInteger bclon = new BigInteger(""+b) ;
        return new Rational(aclon,bclon) ;
} /* Rational.clone */

/** Multiply by another fraction.
* @param val a second rational number.
* @return the product of this with the val.
*/
public Rational multiply(final Rational val)
{
        BigInteger num = a.multiply(val.a) ;
        BigInteger deno = b.multiply(val.b) ;
        /* Normalization to an coprime format will be done inside
        * the ctor() and is not duplicated here.
        */
        return ( new Rational(num,deno) ) ;
} /* Rational.multiply */

/** Multiply by a BigInteger.
* @param val a second number.
* @return the product of this with the value.
*/
public Rational multiply(final BigInteger val)
{
        Rational val2 = new Rational(val,BigInteger.ONE) ;
        return ( multiply(val2) ) ;
} /* Rational.multiply */

/** Multiply by an integer.
* @param val a second number.
* @return the product of this with the value.
*/
public Rational multiply(final int val)
{
        BigInteger tmp = new BigInteger(""+val) ;
        return multiply(tmp) ;
} /* Rational.multiply */

/** Power to an integer.
* @param exponent the exponent.
* @return this value raised to the power given by the exponent.
*  If the exponent is 0, the value 1 is returned.
*/
public Rational pow(int exponent)
{
        if ( exponent == 0 )
                return new Rational(1,1) ;

        BigInteger num = a.pow(Math.abs(exponent)) ;
        BigInteger deno = b.pow(Math.abs(exponent)) ;
        if ( exponent > 0 )
                return ( new Rational(num,deno) ) ;
        else
                return ( new Rational(deno,num) ) ;
} /* Rational.pow */
```

```
/** Power to an integer.
* @param exponent the exponent.
* @return this value raised to the power given by the exponent.
*  If the exponent is 0, the value 1 is returned.
*/
public Rational pow(BigInteger exponent) throws NumberFormatException
{
        /* test for overflow */
        if ( exponent.compareTo(MAX_INT) == 1 )
                throw new NumberFormatException("Exponent "+exponent.toString()+" too large.") ;
        if ( exponent.compareTo(MIN_INT) == -1 )
                throw new NumberFormatException("Exponent "+exponent.toString()+" too small.") ;

        /* promote to the simpler interface above */
        return pow( exponent.intValue() ) ;
} /* Rational.pow */


/** Divide by another fraction.
* @param val A second rational number.
* @return The value of this/val
*/
public Rational divide(final Rational val)
{
        BigInteger num = a.multiply(val.b) ;
        BigInteger deno = b.multiply(val.a) ;
        /* Reduction to a coprime format is done inside the ctor,
        * and not repeated here.
        */
        return ( new Rational(num,deno) ) ;
} /* Rational.divide */

/** Divide by an integer.
* @param val a second number.
* @return the value of this/val
*/
public Rational divide(BigInteger val)
{
        Rational val2 = new Rational(val,BigInteger.ONE) ;
        return ( divide(val2)) ;
} /* Rational.divide */

/** Divide by an integer.
* @param val A second number.
* @return The value of this/val
*/
public Rational divide(int val)
{
        Rational val2 = new Rational(val,1) ;
        return ( divide(val2)) ;
} /* Rational.divide */

/** Add another fraction.
* @param val The number to be added
* @return this+val.
*/
public Rational add(Rational val)
{
        BigInteger num = a.multiply(val.b).add(b.multiply(val.a)) ;
        BigInteger deno = b.multiply(val.b) ;
        return ( new Rational(num,deno) ) ;
} /* Rational.add */

/** Add another integer.
* @param val The number to be added
* @return this+val.
*/
public Rational add(BigInteger val)
{
        Rational val2 = new Rational(val,BigInteger.ONE) ;
        return ( add(val2) ) ;
} /* Rational.add */

/** Compute the negative.
* @return -this.
*/
public Rational negate()
{
        return ( new Rational(a.negate(),b) ) ;
} /* Rational.negate */

/** Subtract another fraction.
```

```
* @param val the number to be subtracted from this
* @return this - val.
*/
public Rational subtract(Rational val)
{
        Rational val2 = val.negate() ;
        return ( add(val2) ) ;
} /* Rational.subtract */

/** Subtract an integer.
* @param val the number to be subtracted from this
* @return this - val.
*/
public Rational subtract(BigInteger val)
{
        Rational val2 = new Rational(val,BigInteger.ONE) ;
        return ( subtract(val2) ) ;
} /* Rational.subtract */

/** Subtract an integer.
* @param val the number to be subtracted from this
* @return this - val.
*/
public Rational subtract(int val)
{
        Rational val2 = new Rational(val,1) ;
        return ( subtract(val2) ) ;
} /* Rational.subtract */

/** binomial (n choose m).
* @param n the numerator. Equals the size of the set to choose from.
* @param m the denominator. Equals the number of elements to select.
* @return the binomial coefficient.
*/
public static Rational binomial(Rational n, BigInteger m)
{
        if ( m.compareTo(BigInteger.ZERO) == 0 )
                return Rational.ONE ;
        Rational bin = n ;
        for(BigInteger i=new BigInteger("2") ; i.compareTo(m) != 1 ; i = i.add(BigInteger.ONE) )
        {
                bin = bin.multiply(n.subtract(i.subtract(BigInteger.ONE))).divide(i) ;
        }
        return bin ;
} /* Rational.binomial */

/** binomial (n choose m).
* @param n the numerator. Equals the size of the set to choose from.
* @param m the denominator. Equals the number of elements to select.
* @return the binomial coefficient.
*/
public static Rational binomial(Rational n, int m)
{
        if ( m == 0 )
                return Rational.ONE ;
        Rational bin = n ;
        for( int i=2 ; i <= m ; i++ )
        {
                bin = bin.multiply(n.subtract(i-1)).divide(i) ;
        }
        return bin ;
} /* Rational.binomial */

/** Get the numerator.
* @return The numerator of the reduced fraction.
*/
public BigInteger numer()
{
        return a ;
}

/** Get the denominator.
* @return The denominator of the reduced fraction.
*/
public BigInteger denom()
{
        return b ;
}

/** Absolute value.
* @return The absolute (non-negative) value of this.
*/
public Rational abs()
```

```
{
        return( new Rational(a.abs(),b.abs())) ;
}

/** floor(): the nearest integer not greater than this.
* @return The integer rounded towards negative infinity.
*/
public BigInteger floor()
{
        /* is already integer: return the numerator
        */
        if ( b.compareTo(BigInteger.ONE) == 0 )
                return a;
        else if ( a.compareTo(BigInteger.ZERO) > 0 )
                return a.divide(b);
        else
                return a.divide(b).subtract(BigInteger.ONE) ;
} /* Rational.floor */

/** Remove the fractional part.
* @return The integer rounded towards zero.
*/
public BigInteger trunc()
{
        /* is already integer: return the numerator
        */
        if ( b.compareTo(BigInteger.ONE) == 0 )
                return a;
        else
                return a.divide(b);
} /* Rational.trunc */

/** Compares the value of this with another constant.
* @param val the other constant to compare with
* @return -1, 0 or 1 if this number is numerically less than, equal to,
*    or greater than val.
*/
public int compareTo(final Rational val)
{
        /* Since we have always kept the denominators positive,
        * simple cross-multiplying works without changing the sign.
        */
        final BigInteger left = a.multiply(val.b) ;
        final BigInteger right = val.a.multiply(b) ;
        return left.compareTo(right) ;
} /* Rational.compareTo */

/** Compares the value of this with another constant.
* @param val the other constant to compare with
* @return -1, 0 or 1 if this number is numerically less than, equal to,
*    or greater than val.
*/
public int compareTo(final BigInteger val)
{
        final Rational val2 = new Rational(val,BigInteger.ONE) ;
        return ( compareTo(val2) ) ;
} /* Rational.compareTo */

/** Return a string in the format number/denom.
* If the denominator equals 1, print just the numerator without a slash.
* @return the human-readable version in base 10
*/
public String toString()
{
        if ( b.compareTo(BigInteger.ONE) != 0)
                return( a.toString()+"/"+b.toString() ) ;
        else
                return a.toString() ;
} /* Rational.toString */

/** Return a double value representation.
* @return The value with double precision.
*/
public double doubleValue()
{
        /* To meet the risk of individual overflows of the exponents of
        * a separate invocation a.doubleValue() or b.doubleValue(), we divide first
        * in a BigDecimal environment and converst the result.
        */
        BigDecimal adivb  = (new BigDecimal(a)).divide(new BigDecimal(b), MathContext.DECIMAL128) ;
        return adivb.doubleValue() ;
} /* Rational.doubleValue */
```

```
/** Return a float value representation.
* @return The value with single precision.
*/
public float floatValue()
{
        BigDecimal adivb  = (new BigDecimal(a)).divide(new BigDecimal(b), MathContext.DECIMAL128) ;
        return adivb.floatValue() ;
} /* Rational.floatValue */

/** Return a representation as BigDecimal.
* @param mc the mathematical context which determines precision, rounding mode etc
* @return A representation as a BigDecimal floating point number.
*/
public BigDecimal BigDecimalValue(MathContext mc)
{
        /* numerator and denominator individually rephrased
        */
        BigDecimal n = new BigDecimal(a) ;
        BigDecimal d = new BigDecimal(b) ;
        return n.divide(d,mc) ;
} /* Rational.BigDecimnalValue */

/** Return a string in floating point format.
* @param digits The precision (number of digits)
* @return The human-readable version in base 10.
*/
public String toFString(int digits)
{
        if ( b.compareTo(BigInteger.ONE) != 0)
        {
                MathContext mc = new MathContext(digits,RoundingMode.DOWN) ;
                BigDecimal f = (new BigDecimal(a)).divide(new BigDecimal(b),mc) ;
                return( f.toString() ) ;
        }
        else
                return a.toString() ;
} /* Rational.toFString */

/** Compares the value of this with another constant.
* @param val The other constant to compare with
* @return The arithmetic maximum of this and val.
*/
public Rational max(final Rational val)
{
        if ( compareTo(val) > 0 )
                return this;
        else
                return val;
} /* Rational.max */

/** Compares the value of this with another constant.
* @param val The other constant to compare with
* @return The arithmetic minimum of this and val.
*/
public Rational min(final Rational val)
{
        if ( compareTo(val) < 0 )
                return this;
        else
                return val;
} /* Rational.min */

/** Compute Pochhammer's symbol (this)_n.
* @param n The number of product terms in the evaluation.
* @return Gamma(this+n)/Gamma(this) = this*(this+1)*...*(this+n-1).
*/
public Rational Pochhammer(final BigInteger n)
{
        if ( n.compareTo(BigInteger.ZERO) < 0 )
                return null;
        else if ( n.compareTo(BigInteger.ZERO) == 0 )
                return Rational.ONE ;
        else
        {
                /* initialize results with the current value
                */
                Rational res = new Rational(a,b) ;
                BigInteger i = BigInteger.ONE ;
                for( ; i.compareTo(n) < 0 ; i=i.add(BigInteger.ONE) )
                        res = res.multiply( add(i) ) ;
                return res;
        }
} /* Rational.pochhammer */
```

```
        /** Compute pochhammer's symbol (this)_n.
        * @param n The number of product terms in the evaluation.
        * @return Gamma(this+n)/GAMMA(this).
        */
        public Rational Pochhammer(int n)
        {
                return Pochhammer(new BigInteger(""+n)) ;
        } /* Rational.pochhammer */

        /** Normalize to coprime numerator and denominator.
        * Also copy a negative sign of the denominator to the numerator.
        */
        protected void normalize()
        {
                /* compute greatest common divisor of numerator and denominator
                */
                final BigInteger g = a.gcd(b) ;
                if ( g.compareTo(BigInteger.ONE) > 0 )
                {
                        a = a.divide(g) ;
                        b = b.divide(g);
                }
                if ( b.compareTo(BigInteger.ZERO) == -1 )
                {
                        a = a.negate() ;
                        b = b.negate() ;
                }
        } /* Rational.normalize */
} /* Rational */
```

APPENDIX B: FACTORIAL.JAVA

```
package org.nevec.rjm ;

import java.util.* ;
import java.math.* ;


/** Factorials.
*/
class Factorial
{
        /** The list of all factorials as a vector.
        */
        static Vector<BigInteger> a = new Vector<BigInteger>() ;

        /** ctor().
        * Initialize the vector of the factorials with 0!=1 and 1!=1.
        */
        public Factorial()
        {
                if ( a.size() == 0 )
                {
                        a.add(BigInteger.ONE) ;
                        a.add(BigInteger.ONE) ;
                }
        }

        /** Compute the factorial of the non-negative integer.
        * @param n the argument to the factorial, non-negative.
        * @return the factorial of n.
        */
        public BigInteger at(int n)
        {
                while ( a.size() <=n )
                {
                        final int lastn = a.size()-1 ;
                        final BigInteger nextn = new BigInteger(""+(lastn+1)) ;
                        a.add(a.elementAt(lastn).multiply(nextn) ) ;
                }
                return a.elementAt(n) ;
        }

} /* Factorial */
```

APPENDIX C: BERNOULLI.JAVA

```
package org.nevec.rjm ;

import java.util.* ;
import java.math.* ;


/** Bernoulli numbers.
*/
class Bernoulli
{
        /*
        * The list of all Bernoulli numbers as a vector, n=0,2,4,....
        */
        static Vector<Rational> a = new Vector<Rational>() ;

        public Bernoulli()
        {
                if ( a.size() == 0 )
                {
                        a.add(Rational.ONE) ;
                        a.add(new Rational(1,6)) ;
                }
        }

        /** Set a coefficient in the internal table.
        * @param n the zero-based index of the coefficient. n=0 for the constant term.
        * @param value the new value of the coefficient.
        */
        protected void set(final int n, final Rational value)
        {
                final int nindx = n /2 ;
                if ( nindx < a.size())
                        a.set(nindx,value) ;
                else
                {
                        while ( a.size() < nindx )
                                a.add( Rational.ZERO ) ;
                        a.add(value) ;
                }
        }

        /** The Bernoulli number at the index provided.
        * @param n the index, non-negative.
        * @return the B_0=1 for n=0, B_1=-1/2 for n=1, B_2=1/6 for n=2 etc
        */
        public Rational at(int n)
        {
                if ( n == 1)
                        return(new Rational(-1,2)) ;
                else if ( n % 2 != 0 )
                        return Rational.ZERO ;
                else
                {
                        final int nindx = n /2 ;
                        if( a.size() <= nindx )
                        {
                                        for(int i= 2*a.size() ; i <= n; i+= 2)
                                                set(i, doubleSum(i) ) ;
                        }
                        return a.elementAt(nindx) ;
                }
        }

        /* Generate a new B_n by a standard double sum.
        * @param n The index of the Bernoulli number.
        * @return The Bernoulli number at n.
        */
        private Rational doubleSum(int n)
        {
                Rational resul = Rational.ZERO ;
                for(int k=0 ; k <= n ; k++)
                {
                        Rational jsum = Rational.ZERO ;
                        BigInteger bin = BigInteger.ONE ;
                        for(int j=0 ; j <= k ; j++)
                        {
                                BigInteger jpown = (new BigInteger(""+j)).pow(n);
                                if ( j % 2 == 0)
                                        jsum = jsum.add(bin.multiply(jpown)) ;
                                else
```

```
                                        jsum = jsum.subtract(bin.multiply(jpown)) ;

                                /* update binomial(k,j) recursively
                                */
                                bin = bin.multiply( new BigInteger(""+(k-j))). divide( new BigInteger(""+(j+1)) ) ;
                        }
                        resul = resul.add(jsum.divide(new BigInteger(""+(k+1)))) ;
                }
                return resul ;
        }


} /* Bernoulli */
```

APPENDIX D: BIGDECIMALMATH.JAVA

```
package org.nevec.rjm ;

import java.security.* ;
import java.util.* ;
import java.math.* ;


/** BigDecimal special functions.
*/
class BigDecimalMath
{

        /** The base of the natural logarithm in a predefined accuracy.
        * \protect\vrule width0pt\protect\href{http://www.cs.arizona.edu/icon/oddsends/e.htm}{http://www.cs.arizona.edu/icon/oddsends/e.htm}
        * The precision of the predefined constant is one less than
        * the string's length, taking into account the decimal dot.
        * static int E_PRECISION = E.length()-1 ;
        */
        static BigDecimal E = new BigDecimal("2.71828182845904523536028747135266249775724709369995957496696762772407663035354"+
"759457138217852516642742746639193200305992181741359662904357290033429526059563"+
"073813232862794349076323382988075319525101901157383418793070215408914993488416"+
"750924476146066808226480016847741185374234544243710753907774499206955170276183"+
"860626133138458300075204493382656029760673711320070932870912744374704723069697"+
"720931014169283681902551510865746377211125238978442505695369677078544996996794"+
"686445490598793163688923009879312773617821542499922957635148220826989519366803"+
"318252886939849646510582093923982948879332036250944311730123819706841614039701"+
"983767932068328237646480429531180232878250981945581530175671736133206981125099"+
"618188159304169035159888851934580727386673858942287922849989208680582574927961"+
"048419844436346324496848756023362482704197862320900216099023530436994184914631"+
"409343173814364054625315209618369088870701676839642437814059271456354906130310"+
"720851038375051011574770417189861068739696552126715468895703503540212340784981"+
"933432106817012100562788023519303322474501585390473041995777709350366041699732"+
"972508868769664035557071622684471625607988265178713419512466520103059212366771"+
"943252786753985589448969709640975459185695638023637016211204774272283648961342"+
"251644507818244235294863637214174023889344124796357437026375529444833799801612"+
"549227850925778256209262264832627793338656648162772516401910590049164499828931") ;

        /** Euler's constant Pi.
        * \protect\vrule width0pt\protect\href{http://www.cs.arizona.edu/icon/oddsends/pi.htm}{http://www.cs.arizona.edu/icon/oddsends/pi.htm}
        */
        static BigDecimal PI = new BigDecimal("3.14159265358979323846264338327950288419716939937510582097494459230781640628620"+
"899862803482534211706798214808651328230664709384460955058223172535940812848111"+
"745028410270193852110555964462294895493038196442881097566593344612847564823378"+
"678316527120190914564856692346034861045432664821339360726024914127372458700660"+
"631558817488152092096282925409171536436789259036001133053054882046652138414695"+
"194151160943305727036575959195309218611738193261179310511854807446237996274956"+
"735188575272489122793818301194912983367336244065664308602139494639522473719070"+
"217986094370277053921717629317675238467481846766940513200056812714526356082778"+
"577134275778960917363717872146844090122495343014654958537105079227968925892354"+
"201995611212902196086403441815981362977477130996051870721134999999837297804995"+
"105973173281609631859502445945534690830264252230825334468503526193118817101000"+
"313783875288658753320838142061717766914730359825349042875546873115956286388235"+
"378759375195778185778053217122680661300192787661119590921642019893809525720106"+
"548586327886593615338182796823030195203530185296899577362259941389124972177528"+
"347913151557485724245415069595082953311686172785588907509838175463746493931925"+
"506040092770167113900984882401285836160356370766010471018194295559619894676783"+
"744944825537977472684710404753464620804668425906949129331367702898915210475216"+
"205696602405803815019351125338243003558764024749647326391419927260426992279678"+
"235478163600934172164121992458631503028618297455570674983850549458858692699569"+
"092721079750930295532116534498720275596023648066549911988183479775356636980742"+
"654252786255181841757467289097777279380008164706001614524919217321721477235014") ;

        /** Euler-Mascheroni constant lower-case gamma.
```

```
        * \protect\vrule width0pt\protect\href{http://www.worldwideschool.org/library/books/sci/math/MiscellaneousMathematicalConstants/chap35.html}{ht
        */
        static BigDecimal GAMMA = new BigDecimal("0.5772156649015328606065120900824024310"+
"0421593359399235988057672348848677267776646709369470632917467495146314472498070"+
"8248096050401448654283622417399764492353625350033374293733773767394279259525824"+
"7094916008735203948165670853233151776611528621199501507984793745085705740029921"+
"3547861466940296043254215190587755352673313992540129674205137541395491116851028"+
"0798423487758720503843109399736137255306088933126760017247953783675927135157722"+
"6102734929139407984301034177717780881549570661075010161916633401522789358679654"+
"9725203621287922655595366962817638879272680132431010476505963703947394957638906"+
"5729679296010090151251959509222435014093498712282479497471956469763185066761290"+
"6381105182419744486783638086174945516989279230187739107294578155431600500218284"+
"4096053772434203285478367015177394398700302370339518328690001558193988042707411"+
"5422278197165230110735658339673487176504919418123000406546931429992977795693031"+
"0050308630341856980323108369164002589297089098548682577736428825395492587362959"+
"6133298574739302373438847070370284412920166417850248733379080562754998434590761"+
"6431671031467107223700218107450444186647591348036690255324586254422253451813879"+
"1243457350136129778227828814894590986384600629316947188714958752549236649352047"+
"3243641097268276160877595088095126208404544477992299157248292516251278427659657"+
"0832146102982146179519579590959227042089896279712553632179488737642106606070659"+
"8256199010288075612519913751167821764361905705844078357350158005607745793421314"+
"49885007864151716151945");

        /** Natural logarithm of 2.
        * \protect\vrule width0pt\protect\href{http://www.worldwideschool.org/library/books/sci/math/MiscellaneousMathematicalConstants/chap58.html}{ht
        */
        static BigDecimal LOG2 = new BigDecimal("0.693147180559945309417232121458176568075"+
"50013436025525412068000949339362196969471560586332699641868754200148102057068573"+
"368552023575813055703267075163507596193072757082837143519030703862389167347112335"+
"011536449795523912047517268157493206515552473413952588295045300709532636664265410"+
"423915781495204374043038550080194417064167151864471283996817178454695702627163106"+
"454615025720740248163777338963855069526066834113727387372292895649354702576265209"+
"885969320196505855476470330679365443254763274495125040606943814710468994650622016"+
"772042452452961268794654619316517468139267250410380254625965686914419287160829380"+
"317271436778265487756648508567407764845146443994046142260319309673540257444607030"+
"809608504748663852313818167675143866747664789088143714198549423151997354880375165"+
"861275352916610007105355824987941472950929311389715599820565439287170007218085761"+
"025236889213244971389320378439353088774825970171559107088236836275898425891853530"+
"243634214367061189236789192372314672321720534016492568727477823445353476481149418"+
"642386776774406069562657379600867076257199184734022651462837904883062033061144630"+
"073719489002743643965002580936519443041191150608094879306786515887090060520346842"+
"973619384128965255653968602219412292420757432175748909770675268711581705113700915"+
"894266547859596489065305846025866838294002283300538207400567705304678700184162404"+
"418833232798386349001563121889560650553151272199398332030751408426091479001265168"+
"243443893572472788205486271552741877243002489794540196187233980860831664811490930"
"667519339312890431641370681397776498176974868903887789991296503619270710889264105"+
"230924783917373501229842420499568935992206602204654941510613");


        /** Euler's constant.
        * @param mc The required precision of the result.
        * @return 3.14159...
        */
        static public BigDecimal pi(final MathContext mc)
        {
                /* look it up if possible */
                if ( mc.getPrecision() < PI.precision() )
                        return PI.round(mc) ;
                else
                {
                        /* Broadhurst \protect\vrule width0pt\protect\href{http://arxiv.org/abs/math/9803067}{arXiv:math/9803067}
                        */
                        int[] a = {1,0,0,-1,-1,-1,0,0} ;
                        BigDecimal S = broadhurstBBP(1,1,a,mc) ;
                        return multiplyRound(S,8) ;
                }
        } /* BigDecimalMath.pi */

        /** Euler-Mascheroni constant.
        * @param mc The required precision of the result.
        * @return 0.577...
        */
        static public BigDecimal gamma(MathContext mc)
        {
                /* look it up if possible */
                if ( mc.getPrecision() < GAMMA.precision() )
                        return GAMMA.round(mc) ;
                else
                {
                        double eps = prec2err(0.577, mc.getPrecision() ) ;


                        /* Euler-Stieltjes as shown in Dilcher, Aequat Math 48 (1) (1994) 55-85
```

```
                */
                MathContext mcloc =  new MathContext(2+mc.getPrecision()) ;
                BigDecimal resul =  BigDecimal.ONE ;
                resul =  resul.add( log(2,mcloc) ) ;
                resul =  resul.subtract( log(3,mcloc) ) ;

                /* how many terms: zeta-1 falls as 1/2^(2n+1), so the
                * terms drop faster than 1/2^(4n+2). Set 1/2^(4kmax+2) < eps.
                * Leading term zeta(3)/(4^1*3) is 0.017. Leading zeta(3) is 1.2. Log(2) is 0.7
                */
                int kmax = (int)((Math.log(eps/0.7)-2.)/4.) ;
                mcloc =  new MathContext( 1+err2prec(1.2,eps/kmax) ) ;
                for(int n=1; ; n++)
                {
                        /* zeta is close to 1. Division of zeta-1 through
                        * 4^n*(2n+1) means divion through roughly 2^(2n+1)
                        */
                        BigDecimal c = zeta(2*n+1,mcloc).subtract(BigDecimal.ONE) ;
                        BigInteger fourn = new BigInteger(""+(2*n+1)) ;
                        fourn = fourn.shiftLeft(2*n) ;
                        c = divideRound(c, fourn) ;
                        resul = resul.subtract(c) ;
                        if ( c.doubleValue() < 0.1*eps)
                                break;
                }
                return resul.round(mc) ;
        }

} /* BigDecimalMath.gamma */



/** The square root.
* @param x the non-negative argument.
* @return the square root of the BigDecimal rounded to the precision implied by x.
*/
static public BigDecimal sqrt(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0 )
                throw new ArithmeticException("negative argument "+x.toString()+ " of square root") ;

        return root(2,x) ;
} /* BigDecimalMath.sqrt */

/** The cube root.
* @param x The argument.
* @return The cubic root of the BigDecimal rounded to the precision implied by x.
* The sign of the result is the sign of the argument.
*/
static public BigDecimal cbrt(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0 )
                return root(3,x.negate()).negate() ;
        else
                return root(3,x) ;
} /* BigDecimalMath.cbrt */

/** The integer root.
* @param n the positive argument.
* @param x the non-negative argument.
* @return The n-th root of the BigDecimal rounded to the precision implied by x, x^(1/n).
*/
static public BigDecimal root(final int n, final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0 )
                throw new ArithmeticException("negative argument "+x.toString()+ " of root") ;
        if ( n<= 0 )
                throw new ArithmeticException("negative power "+ n + " of root") ;

        if ( n == 1 )
                return x ;

        /* start the computation from a double precision estimate */
        BigDecimal s = new BigDecimal( Math.pow(x.doubleValue(),1.0/n) ) ;

        /* this creates nth with nominal precision of 1 digit
        */
        final BigDecimal nth = new BigDecimal(n) ;

        /* Specify an internal accuracy within the loop which is
        * slightly larger than what is demanded by 'eps' below.
        */
        final BigDecimal xhighpr = scalePrec(x,2) ;
```

```
        MathContext mc = new MathContext( 2+x.precision() ) ;

        /* Relative accuracy of the result is eps.
        */
        final double eps = x.ulp().doubleValue()/(2*n*x.doubleValue()) ;
        for (;;)
        {
                /* s = s -(s/n-x/n/s^(n-1)) = s-(s-x/s^(n-1))/n; test correction s/n-x/s for being
                * smaller than the precision requested. The relative correction is (1-x/s^n)/n,
                */
                BigDecimal c = xhighpr.divide( s.pow(n-1),mc ) ;
                c = s.subtract(c) ;
                MathContext locmc = new MathContext( c.precision() ) ;
                c = c.divide(nth,locmc) ;
                s = s. subtract(c) ;
                if ( Math.abs( c.doubleValue()/s.doubleValue()) < eps)
                        break ;
        }
        return s.round(new MathContext( err2prec(eps)) ) ;
} /* BigDecimalMath.root */

/** The hypotenuse.
* @param x the first argument.
* @param y the second argument.
* @return the square root of the sum of the squares of the two arguments, sqrt(x^2+y^2).
*/
static public BigDecimal hypot(final BigDecimal x, final BigDecimal y)
{
        /* compute x^2+y^2
        */
        BigDecimal z = x.pow(2).add(y.pow(2)) ;

        /* truncate to the precision set by x and y. Absolute error = 2*x*xerr+2*y*yerr,
        * where the two errors are 1/2 of the ulp's.  Two intermediate protectio digits.
        */
        BigDecimal zerr = x.abs().multiply(x.ulp()).add(y.abs().multiply(y.ulp())) ;
        MathContext mc = new MathContext(  2+err2prec(z,zerr) ) ;

        /* Pull square root */
        z = sqrt(z.round(mc)) ;

        /* Final rounding. Absolute error in the square root is (y*yerr+x*xerr)/z, where zerr holds 2*(x*xerr+y*yerr).
        */
        mc = new MathContext(  err2prec(z.doubleValue() ,0.5*zerr.doubleValue() /z.doubleValue() ) ) ;
        return z.round(mc) ;
} /* BigDecimalMath.hypot */

/** The hypotenuse.
* @param n the first argument.
* @param x the second argument.
* @return the square root of the sum of the squares of the two arguments, sqrt(n^2+x^2).
*/
static public BigDecimal hypot(final int n, final BigDecimal x)
{
        /* compute n^2+x^2 in infinite precision
        */
        BigDecimal z = (new BigDecimal(n)).pow(2).add(x.pow(2)) ;

        /* Truncate to the precision set by x. Absolute error = in z (square of the result) is |2*x*xerr|,
        * where the error is 1/2 of the ulp. Two intermediate protection digits.
        * zerr is a signed value, but used only in conjunction with err2prec(), so this feature does not harm.
        */
        double zerr = x.doubleValue()*x.ulp().doubleValue() ;
        MathContext mc = new MathContext(  2+err2prec(z.doubleValue(),zerr) ) ;

        /* Pull square root */
        z = sqrt(z.round(mc)) ;

        /* Final rounding. Absolute error in the square root is x*xerr/z, where zerr holds 2*x*xerr.
        */
        mc = new MathContext(  err2prec(z.doubleValue(),0.5*zerr/z.doubleValue() ) ) ;
        return z.round(mc) ;
} /* BigDecimalMath.hypot */


/** A suggestion for the maximum numter of terms in the Taylor expansion of the exponential.
*/
static private int TAYLOR_NTERM = 8 ;

/** The exponential function.
* @param x the argument.
* @return exp(x).
* The precision of the result is implicitly defined by the precision in the argument.
```

```
* In particular this means that "Invalid Operation" errors are thrown if catastrophic
* cancellation of digits causes the result to have no valid digits left.
*/
static public BigDecimal exp(BigDecimal x)
{
        /* To calculate the value if x is negative, use exp(-x) = 1/exp(x)
        */
        if ( x.compareTo(BigDecimal.ZERO) < 0 )
        {
                final BigDecimal invx = exp(x.negate() ) ;
                /* Relative error in inverse of invx is the same as the relative errror in invx.
                * This is used to define the precision of the result.
                */
                MathContext mc = new MathContext( invx.precision() ) ;
                return BigDecimal.ONE.divide( invx, mc ) ;
        }
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
        {
                /* recover the valid number of digits from x.ulp(), if x hits the
                * zero. The x.precision() is 1 then, and does not provide this information.
                */
                return scalePrec(BigDecimal.ONE, -(int)(Math.log10( x.ulp().doubleValue() )) ) ;
        }
        else
        {
                /* Push the number in the Taylor expansion down to a small
                * value where TAYLOR_NTERM terms will do. If x<1, the n-th term is of the order
                * x^n/n!, and equal to both the absolute and relative error of the result
                * since the result is close to 1. The x.ulp() sets the relative and absolute error
                * of the result, as estimated from the first Taylor term.
                * We want x^TAYLOR_NTERM/TAYLOR_NTERM! < x.ulp, which is guaranteed if
                * x^TAYLOR_NTERM < TAYLOR_NTERM*(TAYLOR_NTERM-1)*...*x.ulp.
                */
                final double xDbl = x.doubleValue() ;
                final double xUlpDbl = x.ulp().doubleValue() ;
                if ( Math.pow(xDbl,TAYLOR_NTERM) < TAYLOR_NTERM*(TAYLOR_NTERM-1.0)*(TAYLOR_NTERM-2.0)*xUlpDbl )
                {
                        /* Add TAYLOR_NTERM terms of the Taylor expansion (Euler's sum formula)
                        */
                        BigDecimal resul = BigDecimal.ONE ;

                        /* x^i */
                        BigDecimal xpowi = BigDecimal.ONE ;

                        /* i factorial */
                        BigInteger ifac = BigInteger.ONE ;

                        /* TAYLOR_NTERM terms to be added means we move x.ulp() to the right
                        * for each power of 10 in TAYLOR_NTERM, so the addition won't add noise beyond
                        * what's already in x.
                        */
                        MathContext mcTay = new MathContext( err2prec(1.,xUlpDbl/TAYLOR_NTERM) ) ;
                        for(int i=1 ; i <= TAYLOR_NTERM ; i++)
                        {
                                ifac = ifac.multiply(new BigInteger(""+i) ) ;
                                xpowi = xpowi.multiply(x) ;
                                final BigDecimal c= xpowi.divide(new BigDecimal(ifac),mcTay)  ;
                                resul = resul.add(c) ;
                                if ( Math.abs(xpowi.doubleValue()) < i && Math.abs(c.doubleValue()) < 0.5* xUlpDbl )
                                        break;
                        }
                        /* exp(x+deltax) = exp(x)(1+deltax) if deltax is <<1. So the relative error
                        * in the result equals the absolute error in the argument.
                        */
                        MathContext mc = new MathContext( err2prec(xUlpDbl/2.) ) ;
                        return resul.round(mc) ;
                }
                else
                {
                        /* Compute exp(x) = (exp(0.1*x))^10. Division by 10 does not lead
                        * to loss of accuracy.
                        */
                        int exSc = (int) ( 1.0-Math.log10( TAYLOR_NTERM*(TAYLOR_NTERM-1.0)*(TAYLOR_NTERM-2.0)*xUlpDbl
                                                /Math.pow(xDbl,TAYLOR_NTERM) ) / ( TAYLOR_NTERM-1.0) ) ;
                        BigDecimal xby10 = x.scaleByPowerOfTen(-exSc) ;
                        BigDecimal expxby10 = exp(xby10) ;

                        /* Final powering by 10 means that the relative error of the result
                        * is 10 times the relative error of the base (First order binomial expansion).
                        * This looses one digit.
                        */
                        MathContext mc = new MathContext( expxby10.precision()-exSc ) ;
                        /* Rescaling the powers of 10 is done in chunks of a maximum of 8 to avoid an invalid operation
```

```
			* response by the BigDecimal.pow library or integer overflow.
			*/
			while ( exSc > 0 )
			{
				int exsub = Math.min(8,exSc) ;
				exSc -= exsub ;
				MathContext mctmp = new MathContext( expxby10.precision()-exsub+2 ) ;
				int pex = 1 ;
				while ( exsub-- > 0 )
					pex *= 10 ;
				expxby10 = expxby10.pow(pex,mctmp) ;
			}
			return expxby10.round(mc) ;
		}
	}
} /* BigDecimalMath.exp */

/** The base of the natural logarithm.
* @param mc the required precision of the result
* @return exp(1) = 2.71828....
*/
static public BigDecimal exp(final MathContext mc)
{
	/* look it up if possible */
	if ( mc.getPrecision() < E.precision() )
		return E.round(mc) ;
	else
	{
		/* Instantiate a 1.0 with the requested pseudo-accuracy
		* and delegate the computation to the public method above.
		*/
		BigDecimal uni = scalePrec(BigDecimal.ONE, mc.getPrecision() ) ;
		return exp(uni) ;
	}
} /* BigDecimalMath.exp */

/** The natural logarithm.
* @param x the argument.
* @return ln(x).
* The precision of the result is implicitly defined by the precision in the argument.
*/
static public BigDecimal log(BigDecimal x)
{
	/* the value is undefined if x is negative.
	*/
	if ( x.compareTo(BigDecimal.ZERO) < 0 )
		throw new ArithmeticException("Cannot take log of negative "+ x.toString() ) ;
	else if ( x.compareTo(BigDecimal.ONE) == 0 )
	{
		/* log 1. = 0. */
		return scalePrec(BigDecimal.ZERO, x.precision()-1) ;
	}
	else if ( Math.abs(x.doubleValue()-1.0) <= 0.3 )
	{
		/* The standard Taylor series around x=1, z=0, z=x-1. Abramowitz-Stegun 4.124.
		* The absolute error is err(z)/(1+z) = err(x)/x.
		*/
		BigDecimal z = scalePrec(x.subtract(BigDecimal.ONE),2) ;
		BigDecimal zpown = z ;
		double eps = 0.5*x.ulp().doubleValue()/Math.abs(x.doubleValue()) ;
		BigDecimal resul = z ;
		for(int k= 2;; k++)
		{
			zpown = multiplyRound(zpown,z) ;
			BigDecimal c = divideRound(zpown,k) ;
			if ( k % 2 == 0)
				resul = resul.subtract(c) ;
			else
				resul = resul.add(c) ;
			if ( Math.abs(c.doubleValue()) < eps)
				break;
		}
		MathContext mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
		return resul.round(mc) ;
	}
	else
	{
		final double xDbl = x.doubleValue() ;
		final double xUlpDbl = x.ulp().doubleValue() ;

		/* Map log(x) = log root[r](x)^r = r*log( root[r](x)) with the aim
		* to move roor[r](x) near to 1.2 (that is, below the 0.3 appearing above), where log(1.2) is roughly 0.2.
		*/
```

```
                int r = (int) (Math.log(xDbl)/0.2) ;

                /* Since the actual requirement is a function of the value 0.3 appearing above,
                * we avoid the hypothetical case of endless recurrence by ensuring that r >= 2.
                */
                r = Math.max(2,r) ;

                /* Compute r-th root with 2 additional digits of precision
                */
                BigDecimal xhighpr = scalePrec(x,2) ;
                BigDecimal resul = root(r,xhighpr) ;
                resul = log(resul).multiply(new BigDecimal(r)) ;

                /* error propagation: log(x+errx) = log(x)+errx/x, so the absolute error
                * in the result equals the relative error in the input, xUlpDbl/xDbl .
                */
                MathContext mc = new MathContext( err2prec(resul.doubleValue(),xUlpDbl/xDbl) ) ;
                return resul.round(mc) ;
        }
} /* BigDecimalMath.log */

/** The natural logarithm.
* @param n The main argument, a strictly positive integer.
* @param mc The requirements on the precision.
* @return ln(n).
*/
static public BigDecimal log(int n, final MathContext mc)
{
        /* the value is undefined if x is negative.
        */
        if ( n <= 0 )
                throw new ArithmeticException("Cannot take log of negative "+ n ) ;
        else if ( n == 1)
                return BigDecimal.ZERO ;
        else if ( n == 2)
        {
                if ( mc.getPrecision() < LOG2.precision() )
                        return LOG2.round(mc) ;
                else
                {
                        /* Broadhurst \protect\vrule width0pt\protect\href{http://arxiv.org/abs/math/9803067}{arXiv:math/9803067}
                        * Error propagation: the error in log(2) is twice the error in S(2,-5,...).
                        */
                        int[] a = {2,-5,-2,-7,-2,-5,2,-3} ;
                        BigDecimal S = broadhurstBBP(2,1,a, new MathContext(1+mc.getPrecision()) ) ;
                        S = S.multiply(new BigDecimal(8)) ;
                        S = sqrt(divideRound(S,3)) ;
                        return S.round(mc) ;
                }
        }
        else if ( n == 3)
        {
                /* summation of a series roughly proportional to (7/500)^k. Estimate count
                * of terms to estimate the precision (drop the favorable additional
                * 1/k here): 0.013^k <= 10^(-precision), so k*log10(0.013) <= -precision
                * so k>= precision/1.87.
                */
                int kmax = (int)(mc.getPrecision()/1.87) ;
                MathContext mcloc = new MathContext( mc.getPrecision()+ 1+(int)(Math.log10(kmax*0.693/1.098)) ) ;
                BigDecimal log3 = multiplyRound( log(2,mcloc),19 ) ;

                /* log3 is roughly 1, so absolute and relative error are the same. The
                * result will be divided by 12, so a conservative error is the one
                * already found in mc
                */
                double eps = prec2err(1.098,mc.getPrecision() )/kmax ;
                Rational r = new Rational(7153,524288) ;
                Rational pk = new Rational(7153,524288) ;
                for(int k=1; ; k++)
                {
                        Rational tmp = pk.divide(k) ;
                        if ( tmp.doubleValue() < eps)
                                break ;

                        /* how many digits of tmp do we need in the sum?
                        */
                        mcloc = new MathContext( err2prec(tmp.doubleValue(),eps) ) ;
                        BigDecimal c = pk.divide(k).BigDecimalValue(mcloc) ;
                        if ( k % 2 != 0)
                                log3 = log3.add(c) ;
                        else
                                log3 = log3.subtract(c) ;
                        pk = pk.multiply(r) ;
```

```
        }
        log3 = divideRound( log3,12 ) ;
        return log3.round(mc) ;
}
else if ( n == 5)
{
        /* summation of a series roughly proportional to (7/160)^k. Estimate count
        * of terms to estimate the precision (drop the favorable additional
        * 1/k here): 0.046^k <= 10^(-precision), so k*log10(0.046) <= -precision
        * so k>= precision/1.33.
        */
        int kmax = (int)(mc.getPrecision()/1.33) ;
        MathContext mcloc = new MathContext( mc.getPrecision()+ 1+(int)(Math.log10(kmax*0.693/1.609)) ) ;
        BigDecimal log5 = multiplyRound( log(2,mcloc),14 ) ;

        /* log5 is roughly 1.6, so absolute and relative error are the same. The
        * result will be divided by 6, so a conservative error is the one
        * already found in mc
        */
        double eps = prec2err(1.6,mc.getPrecision() )/kmax ;
        Rational r = new Rational(759,16384) ;
        Rational pk = new Rational(759,16384) ;
        for(int k=1; ; k++)
        {
                Rational tmp = pk.divide(k) ;
                if ( tmp.doubleValue() < eps)
                        break ;

                /* how many digits of tmp do we need in the sum?
                */
                mcloc = new MathContext( err2prec(tmp.doubleValue(),eps) ) ;
                BigDecimal c = pk.divide(k).BigDecimalValue(mcloc) ;
                log5 = log5.subtract(c) ;
                pk = pk.multiply(r) ;
        }
        log5 = divideRound( log5,6 ) ;
        return log5.round(mc) ;
}
else if ( n == 7)
{
        /* summation of a series roughly proportional to (1/8)^k. Estimate count
        * of terms to estimate the precision (drop the favorable additional
        * 1/k here): 0.125^k <= 10^(-precision), so k*log10(0.125) <= -precision
        * so k>= precision/0.903.
        */
        int kmax = (int)(mc.getPrecision()/0.903) ;
        MathContext mcloc = new MathContext( mc.getPrecision()+ 1+(int)(Math.log10(kmax*3*0.693/1.098)) ) ;
        BigDecimal log7 = multiplyRound( log(2,mcloc),3 ) ;

        /* log7 is roughly 1.9, so absolute and relative error are the same.
        */
        double eps = prec2err(1.9,mc.getPrecision() )/kmax ;
        Rational r = new Rational(1,8) ;
        Rational pk = new Rational(1,8) ;
        for(int k=1; ; k++)
        {
                Rational tmp = pk.divide(k) ;
                if ( tmp.doubleValue() < eps)
                        break ;

                /* how many digits of tmp do we need in the sum?
                */
                mcloc = new MathContext( err2prec(tmp.doubleValue(),eps) ) ;
                BigDecimal c = pk.divide(k).BigDecimalValue(mcloc) ;
                log7 = log7.subtract(c) ;
                pk = pk.multiply(r) ;
        }
        return log7.round(mc) ;

}

else
{
        /* At this point one could either forward to the log(BigDecimal) signature (implemented)
        * or decompose n into Ifactors and use an implemenation of all the prime bases.
        * Estimate of the result; convert the mc argument to an  absolute error eps
        * log(n+errn) = log(n)+errn/n = log(n)+eps
        */
        double res = Math.log((double)n) ;
        double eps = prec2err(res,mc.getPrecision() ) ;
        /* errn = eps*n, convert absolute error in result to requirement on absolute error in input
        */
        eps *= n ;
```

```
                /* Convert this absolute requirement of error in n to a relative error in n
                */
                final MathContext mcloc = new MathContext( 1+err2prec((double)n,eps ) ) ;
                /* Padd n with a number of zeros to trigger the required accuracy in
                * the standard signature method
                */
                BigDecimal nb = scalePrec(new BigDecimal(n),mcloc) ;
                return log(nb) ;
        }
} /* log */

/** The natural logarithm.
* @param r The main argument, a strictly positive value.
* @param mc The requirements on the precision.
* @return ln(r).
*/
static public BigDecimal log(final Rational r, final MathContext mc)
{
        /* the value is undefined if x is negative.
        */
        if ( r.compareTo(Rational.ZERO) <= 0 )
                throw new ArithmeticException("Cannot take log of negative "+ r.toString() ) ;
        else if ( r.compareTo(Rational.ONE) == 0)
                return BigDecimal.ZERO ;
        else
        {

                /* log(r+epsr) = log(r)+epsr/r. Convert the precision to an absolute error in the result.
                * eps contains the required absolute error of the result, epsr/r.
                */
                double eps = prec2err( Math.log(r.doubleValue()), mc.getPrecision()) ;

                /* Convert this further into a requirement of the relative precision in r, given that
                * epsr/r is also the relative precision of r. Add one safety digit.
                */
                MathContext mcloc = new MathContext( 1+err2prec(eps)  ) ;

                final BigDecimal resul = log( r.BigDecimalValue(mcloc) );

                return resul.round(mc) ;
        }
} /* log */

/** Power function.
* @param x Base of the power.
* @param y Exponent of the power.
* @return x^y.
*  The estimation of the relative error in the result is |log(x)*err(y)|+|y*err(x)/x|
*/
static public BigDecimal pow(final BigDecimal x, final BigDecimal y)
{
        if( x.compareTo(BigDecimal.ZERO) < 0 )
                throw new ArithmeticException("Cannot power negative "+ x.toString()) ;
        else if( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                /* return x^y = exp(y*log(x)) ;
                */
                BigDecimal logx = log(x) ;
                BigDecimal ylogx = y.multiply(logx) ;
                BigDecimal resul = exp(ylogx) ;

                /* The estimation of the relative error in the result is |log(x)*err(y)|+|y*err(x)/x|
                */
                double errR = Math.abs(logx.doubleValue()*y.ulp().doubleValue()/2.)
                        + Math.abs(y.doubleValue()*x.ulp().doubleValue()/2./x.doubleValue()) ;
                MathContext mcR = new MathContext( err2prec(1.0,errR) ) ;
                return resul.round(mcR) ;
        }
} /* BigDecimalMath.pow */

/** Raise to an integer power and round.
* @param x The base.
* @param n The exponent.
* @return x^n.
*/
static public BigDecimal powRound(final BigDecimal x, final int n)
{
        /* The relative error in the result is n times the relative error in the input.
        * The estimation is slightly optimistic due to the integer rounding of the logarithm.
        */
        MathContext mc = new MathContext( x.precision() - (int)Math.log10((double)(Math.abs(n))) ) ;
```

```
        return x.pow(n,mc) ;
} /* BigDecimalMath.powRound */

/** Trigonometric sine.
* @param x The argument in radians.
* @return sin(x) in the range -1 to 1.
*/
static public BigDecimal sin(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0)
                return sin(x.negate()).negate() ;
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                /* reduce modulo 2pi
                */
                BigDecimal res = mod2pi(x) ;
                double errpi = 0.5*Math.abs(x.ulp().doubleValue()) ;
                MathContext mc = new MathContext( 2+err2prec(3.14159,errpi) ) ;
                BigDecimal p= pi(mc) ;
                mc = new MathContext( x.precision() ) ;
                if ( res.compareTo(p) > 0 )
                {
                        /* pi<x<=2pi: sin(x)= - sin(x-pi)
                        */
                        return sin(subtractRound(res,p)) .negate() ;
                }
                else if ( res.multiply(new BigDecimal("2")).compareTo(p) > 0 )
                {
                        /* pi/2<x<=pi: sin(x)= sin(pi-x)
                        */
                        return sin(subtractRound(p,res)) ;
                }
                else
                {
                        /* for the range 0<=x<Pi/2 one could use sin(2x)=2sin(x)cos(x)
                        * to split this further. Here, use the sine up to pi/4 and the cosine higher up.
                        */
                        if ( res.multiply(new BigDecimal("4")).compareTo(p) > 0 )
                        {
                                /* x>pi/4: sin(x) = cos(pi/2-x)
                                */
                                return cos( subtractRound(p.divide(new BigDecimal("2")),res) ) ;
                        }
                        else
                        {
                                /* Simple Taylor expansion, sum_{i=1..infinity} (-1)^(..)res^(2i+1)/(2i+1)! */
                                BigDecimal resul = res ;

                                /* x^i */
                                BigDecimal xpowi = res ;

                                /* 2i+1 factorial */
                                BigInteger ifac = BigInteger.ONE ;

                                /* The error in the result is set by the error in x itself.
                                */
                                double xUlpDbl = res.ulp().doubleValue() ;

                                /* The error in the result is set by the error in x itself.
                                * We need at most k terms to squeeze x^(2k+1)/(2k+1)! below this value.
                                * x^(2k+1) < x.ulp; (2k+1)*log10(x) < -x.precision; 2k*log10(x)< -x.precision;
                                * 2k*(-log10(x)) > x.precision; 2k*log10(1/x) > x.precision
                                */
                                int k = (int)(res.precision()/Math.log10(1.0/res.doubleValue()))/2 ;
                                MathContext mcTay = new MathContext( err2prec(res.doubleValue(),xUlpDbl/k) ) ;
                                for(int i=1 ; ; i++)
                                {
                                        /* TBD: at which precision will 2*i or 2*i+1 overflow?
                                        */
                                        ifac = ifac.multiply(new BigInteger(""+(2*i) ) ) ;
                                        ifac = ifac.multiply( new BigInteger(""+(2*i+1)) ) ;
                                        xpowi = xpowi.multiply(res).multiply(res).negate() ;
                                        BigDecimal corr = xpowi.divide(new BigDecimal(ifac),mcTay) ;
                                        resul = resul.add( corr ) ;
                                        if ( corr.abs().doubleValue() < 0.5*xUlpDbl )
                                                break ;
                                }
                                /* The error in the result is set by the error in x itself.
                                */
                                mc = new MathContext(res.precision() ) ;
                                return resul.round(mc) ;
```

```
                        }
                }
        }
} /* sin */

/** Trigonometric cosine.
* @param x The argument in radians.
* @return cos(x) in the range -1 to 1.
*/
static public BigDecimal cos(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0)
                return cos(x.negate());
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ONE ;
        else
        {
                /* reduce modulo 2pi
                */
                BigDecimal res = mod2pi(x) ;
                double errpi = 0.5*Math.abs(x.ulp().doubleValue()) ;
                MathContext mc = new MathContext( 2+err2prec(3.14159,errpi) ) ;
                BigDecimal p= pi(mc) ;
                mc = new MathContext( x.precision() ) ;
                if ( res.compareTo(p) > 0 )
                {
                        /* pi<x<=2pi: cos(x)= - cos(x-pi)
                        */
                        return cos( subtractRound(res,p)) .negate() ;
                }
                else if ( res.multiply(new BigDecimal("2")).compareTo(p) > 0 )
                {
                        /* pi/2<x<=pi: cos(x)= -cos(pi-x)
                        */
                        return cos( subtractRound(p,res)).negate() ;
                }
                else
                {
                        /* for the range 0<=x<Pi/2 one could use cos(2x)= 1-2*sin^2(x)
                        * to split this further, or use the cos up to pi/4 and the sine higher up.
                                throw new ProviderException("Unimplemented cosine ") ;
                        */
                        if ( res.multiply(new BigDecimal("4")).compareTo(p) > 0 )
                        {
                                /* x>pi/4: cos(x) = sin(pi/2-x)
                                */
                                return sin( subtractRound(p.divide(new BigDecimal("2")),res) ) ;
                        }
                        else
                        {
                                /* Simple Taylor expansion, sum_{i=0..infinity} (-1)^(..)res^(2i)/(2i)! */
                                BigDecimal resul = BigDecimal.ONE ;

                                /* x^i */
                                BigDecimal xpowi = BigDecimal.ONE ;

                                /* 2i factorial */
                                BigInteger ifac = BigInteger.ONE ;

                                /* The absolute error in the result is the error in x^2/2 which is x times the error in x.
                                */
                                double xUlpDbl = 0.5*res.ulp().doubleValue()*res.doubleValue() ;

                                /* The error in the result is set by the error in x^2/2 itself, xUlpDbl.
                                * We need at most k terms to push x^(2k+1)/(2k+1)! below this value.
                                * x^(2k) < xUlpDbl; (2k)*log(x) < log(xUlpDbl);
                                */
                                int k = (int)(Math.log(xUlpDbl)/Math.log(res.doubleValue()) )/2 ;
                                MathContext mcTay = new MathContext( err2prec(1.,xUlpDbl/k) ) ;
                                for(int i=1 ; ; i++)
                                {
                                        /* TBD: at which precision will 2*i-1 or 2*i overflow?
                                        */
                                        ifac = ifac.multiply(new BigInteger(""+(2*i-1) ) ) ;
                                        ifac = ifac.multiply( new BigInteger(""+(2*i)) ) ;
                                        xpowi = xpowi.multiply(res).multiply(res).negate() ;
                                        BigDecimal corr = xpowi.divide(new BigDecimal(ifac),mcTay) ;
                                        resul = resul.add( corr ) ;
                                        if ( corr.abs().doubleValue() < 0.5*xUlpDbl )
                                                break ;
                                }
                                /* The error in the result is governed by the error in x itself.
                                */
```

```
                                mc = new MathContext( err2prec(resul.doubleValue(),xUlpDbl) ) ;
                                return resul.round(mc) ;
                        }
                }
        }
} /* BigDecimalMath.cos */

/** The trigonometric tangent.
* @param x the argument in radians.
* @return the tan(x)
*/
static public BigDecimal tan(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else if ( x.compareTo(BigDecimal.ZERO) < 0 )
        {
                return tan(x.negate()).negate() ;
        }
        else
        {
                /* reduce modulo pi
                */
                BigDecimal res = modpi(x) ;

                /* absolute error in the result is err(x)/cos^2(x) to lowest order
                */
                final double xDbl = res.doubleValue() ;
                final double xUlpDbl = x.ulp().doubleValue()/2. ;
                final double eps = xUlpDbl/2./Math.pow(Math.cos(xDbl),2.) ;

                if ( xDbl > 0.8)
                {
                        /* tan(x) = 1/cot(x) */
                        BigDecimal co = cot(x) ;
                        MathContext mc = new MathContext( err2prec(1./co.doubleValue(),eps) ) ;
                        return BigDecimal.ONE.divide(co,mc) ;
                }
                else
                {
                        final BigDecimal xhighpr = scalePrec(res,2) ;
                        final BigDecimal xhighprSq = multiplyRound(xhighpr,xhighpr) ;

                        BigDecimal resul = xhighpr.plus() ;

                        /* x^(2i+1) */
                        BigDecimal xpowi = xhighpr ;

                        Bernoulli b = new Bernoulli() ;

                        /* 2^(2i) */
                        BigInteger fourn = new BigInteger("4") ;
                        /* (2i)! */
                        BigInteger fac = new BigInteger("2") ;

                        for(int i= 2 ; ; i++)
                        {
                                Rational f = b.at(2*i).abs() ;
                                fourn = fourn.shiftLeft(2) ;
                                fac = fac.multiply(new BigInteger(""+(2*i))).multiply(new BigInteger(""+(2*i-1))) ;
                                f = f.multiply(fourn).multiply(fourn.subtract(BigInteger.ONE)).divide(fac) ;
                                xpowi = multiplyRound(xpowi,xhighprSq) ;
                                BigDecimal c = multiplyRound(xpowi,f) ;
                                resul = resul.add(c) ;
                                if ( Math.abs(c.doubleValue()) < 0.1*eps)
                                        break;
                        }
                        MathContext mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
                        return resul.round(mc) ;
                }
        }
} /* BigDecimalMath.tan */

/** The trigonometric co-tangent.
* @param x the argument in radians.
* @return the cot(x)
*/
static public BigDecimal cot(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) == 0 )
        {
                throw new ArithmeticException("Cannot take cot of zero "+ x.toString() ) ;
        }
```

```
        else if ( x.compareTo(BigDecimal.ZERO) < 0 )
        {
                return cot(x.negate()).negate() ;
        }
        else
        {
                /* reduce modulo pi
                */
                BigDecimal res = modpi(x) ;

                /* absolute error in the result is err(x)/sin^2(x) to lowest order
                */
                final double xDbl = res.doubleValue() ;
                final double xUlpDbl = x.ulp().doubleValue()/2. ;
                final double eps = xUlpDbl/2./Math.pow(Math.sin(xDbl),2.) ;

                final BigDecimal xhighpr = scalePrec(res,2) ;
                final BigDecimal xhighprSq = multiplyRound(xhighpr,xhighpr) ;

                MathContext mc = new MathContext( err2prec(xhighpr.doubleValue(),eps) ) ;
                BigDecimal resul = BigDecimal.ONE.divide(xhighpr,mc) ;

                /* x^(2i-1) */
                BigDecimal xpowi = xhighpr ;

                Bernoulli b = new Bernoulli() ;

                /* 2^(2i) */
                BigInteger fourn = new BigInteger("4") ;
                /* (2i)! */
                BigInteger fac = BigInteger.ONE ;

                for(int i= 1 ; ; i++)
                {
                        Rational f = b.at(2*i) ;
                        fac = fac.multiply(new BigInteger(""+(2*i))).multiply(new BigInteger(""+(2*i-1))) ;
                        f = f.multiply(fourn).divide(fac) ;
                        BigDecimal c = multiplyRound(xpowi,f) ;
                        if ( i % 2 == 0 )
                                resul = resul.add(c) ;
                        else
                                resul = resul.subtract(c) ;
                        if ( Math.abs(c.doubleValue()) < 0.1*eps)
                                break;

                        fourn = fourn.shiftLeft(2) ;
                        xpowi = multiplyRound(xpowi,xhighprSq) ;
                }
                mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
                return resul.round(mc) ;
        }
} /* BigDecimalMath.cot */

/** The inverse trigonometric sine.
* @param x the argument.
* @return the arcsin(x) in radians.
*/
static public BigDecimal asin(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ONE) > 0 || x.compareTo(BigDecimal.ONE.negate()) < 0 )
        {
                throw new ArithmeticException("Out of range argument "+ x.toString() + " of asin") ;
        }
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else if ( x.compareTo(BigDecimal.ONE) == 0 )
        {
                /* arcsin(1) = pi/2
                */
                double errpi = Math.sqrt(x.ulp().doubleValue()) ;
                MathContext mc = new MathContext( err2prec(3.14159,errpi) ) ;
                return pi(mc).divide(new BigDecimal(2)) ;
        }
        else if ( x.compareTo(BigDecimal.ZERO) < 0 )
        {
                return asin(x.negate()).negate() ;
        }
        else if ( x.doubleValue() > 0.7)
        {
                final BigDecimal xCompl = BigDecimal.ONE.subtract(x) ;
                final double xDbl = x.doubleValue() ;
                final double xUlpDbl = x.ulp().doubleValue()/2. ;
                final double eps = xUlpDbl/2./Math.sqrt(1.-Math.pow(xDbl,2.)) ;
```

```
		final BigDecimal xhighpr = scalePrec(xCompl,3) ;
		final BigDecimal xhighprV = divideRound(xhighpr,4) ;

		BigDecimal resul = BigDecimal.ONE ;

		/* x^(2i+1) */
		BigDecimal xpowi = BigDecimal.ONE ;

		/* i factorial */
		BigInteger ifacN = BigInteger.ONE ;
		BigInteger ifacD = BigInteger.ONE ;

		for(int i=1 ; ; i++)
		{
			ifacN = ifacN.multiply(new BigInteger(""+(2*i-1)) ) ;
			ifacD = ifacD.multiply(new BigInteger(""+i) ) ;
			if ( i == 1)
				xpowi = xhighprV ;
			else
				xpowi = multiplyRound(xpowi,xhighprV) ;
			BigDecimal c = divideRound( multiplyRound(xpowi,ifacN),
						ifacD.multiply(new BigInteger(""+(2*i+1)) ) ) ;
			resul = resul.add(c) ;
			/* series started 1+x/12+... which yields an estimate of the sum's error
			*/
			if ( Math.abs(c.doubleValue()) < xUlpDbl/120.)
				break;
		}
		/* sqrt(2*z)*(1+...)
		*/
		xpowi = sqrt(xhighpr.multiply(new BigDecimal(2))) ;
		resul = multiplyRound(xpowi,resul) ;

		MathContext mc = new MathContext( resul.precision() ) ;
		BigDecimal pihalf = pi(mc).divide(new BigDecimal(2)) ;

		mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
		return pihalf.subtract(resul,mc) ;
	}
	else
	{
		/* absolute error in the result is err(x)/sqrt(1-x^2) to lowest order
		*/
		final double xDbl = x.doubleValue() ;
		final double xUlpDbl = x.ulp().doubleValue()/2. ;
		final double eps = xUlpDbl/2./Math.sqrt(1.-Math.pow(xDbl,2.)) ;

		final BigDecimal xhighpr = scalePrec(x,2) ;
		final BigDecimal xhighprSq = multiplyRound(xhighpr,xhighpr) ;

		BigDecimal resul = xhighpr.plus() ;

		/* x^(2i+1) */
		BigDecimal xpowi = xhighpr ;

		/* i factorial */
		BigInteger ifacN = BigInteger.ONE ;
		BigInteger ifacD = BigInteger.ONE ;

		for(int i=1 ; ; i++)
		{
			ifacN = ifacN.multiply(new BigInteger(""+(2*i-1)) ) ;
			ifacD = ifacD.multiply(new BigInteger(""+(2*i)) ) ;
			xpowi = multiplyRound(xpowi,xhighprSq) ;
			BigDecimal c = divideRound( multiplyRound(xpowi,ifacN),
						ifacD.multiply(new BigInteger(""+(2*i+1)) ) ) ;
			resul = resul.add(c) ;
			if ( Math.abs(c.doubleValue()) < 0.1*eps)
				break;
		}
		MathContext mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
		return resul.round(mc) ;
	}
} /* BigDecimalMath.asin */

/** The inverse trigonometric tangent.
* @param x the argument.
* @return the principal value of arctan(x) in radians in the range -pi/2 to +pi/2.
*/
static public BigDecimal atan(final BigDecimal x)
{
	if ( x.compareTo(BigDecimal.ZERO) < 0 )
```

```
        {
                return atan(x.negate()).negate() ;
        }
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else if ( x.doubleValue() >0.7 && x.doubleValue() <3.0)
        {
                /* Abramowitz-Stegun 4.4.34 convergence acceleration
                * 2*arctan(x) = arctan(2x/(1-x^2)) = arctan(y).  x=(sqrt(1+y^2)-1)/y
                * This maps 0<=y<=3 to 0<=x<=0.73 roughly. Temporarily with 2 protectionist digits.
                */
                BigDecimal y = scalePrec(x,2) ;
                BigDecimal newx = divideRound( hypot(1,y).subtract(BigDecimal.ONE) , y);

                /* intermediate result with too optimistic error estimate*/
                BigDecimal resul = multiplyRound( atan(newx), 2) ;

                /* absolute error in the result is errx/(1+x^2), where errx = half  of the ulp. */
                double eps = x.ulp().doubleValue()/( 2.0*Math.hypot(1.0,x.doubleValue()) ) ;
                MathContext mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
                return resul.round(mc) ;
        }
        else if ( x.doubleValue() < 0.71 )
        {
                /* Taylor expansion around x=0; Abramowitz-Stegun 4.4.42 */

                final BigDecimal xhighpr = scalePrec(x,2) ;
                final BigDecimal xhighprSq = multiplyRound(xhighpr,xhighpr).negate() ;

                BigDecimal resul = xhighpr.plus() ;

                /* signed x^(2i+1) */
                BigDecimal xpowi = xhighpr ;

                /* absolute error in the result is errx/(1+x^2), where errx = half  of the ulp.
                */
                double eps = x.ulp().doubleValue()/( 2.0*Math.hypot(1.0,x.doubleValue()) ) ;

                for(int i= 1 ; ; i++)
                {
                        xpowi = multiplyRound(xpowi,xhighprSq) ;
                        BigDecimal c = divideRound(xpowi,2*i+1) ;

                        resul = resul.add(c) ;
                        if ( Math.abs(c.doubleValue()) < 0.1*eps)
                                break;
                }
                MathContext mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
                return resul.round(mc) ;
        }
        else
        {
                /* Taylor expansion around x=infinity; Abramowitz-Stegun 4.4.42 */

                /* absolute error in the result is errx/(1+x^2), where errx = half  of the ulp.
                */
                double eps = x.ulp().doubleValue()/( 2.0*Math.hypot(1.0,x.doubleValue()) ) ;

                /* start with the term pi/2; gather its precision relative to the expected result
                */
                MathContext mc = new MathContext( 2+err2prec(3.1416,eps) ) ;
                BigDecimal onepi= pi(mc) ;
                BigDecimal resul = onepi.divide(new BigDecimal(2)) ;

                final BigDecimal xhighpr = divideRound(-1,scalePrec(x,2)) ;
                final BigDecimal xhighprSq = multiplyRound(xhighpr,xhighpr).negate() ;

                /* signed x^(2i+1) */
                BigDecimal xpowi = xhighpr ;

                for(int i= 0 ; ; i++)
                {
                        BigDecimal c = divideRound(xpowi,2*i+1) ;

                        resul = resul.add(c) ;
                        if ( Math.abs(c.doubleValue()) < 0.1*eps)
                                break;
                        xpowi = multiplyRound(xpowi,xhighprSq) ;
                }
                mc = new MathContext( err2prec(resul.doubleValue(),eps) ) ;
                return resul.round(mc) ;
        }
} /* BigDecimalMath.atan */
```

```
/** The hyperbolic cosine.
* @param x The argument.
* @return The cosh(x) = (exp(x)+exp(-x))/2 .
*/
static public BigDecimal cosh(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0)
                return cos(x.negate());
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ONE ;
        else
        {
                if ( x.doubleValue() > 1.5 )
                {
                        /* cosh^2(x) = 1+ sinh^2(x).
                        */
                        return hypot(1, sinh(x) ) ;
                }
                else
                {
                        BigDecimal xhighpr = scalePrec(x,2) ;
                        /* Simple Taylor expansion, sum_{0=1..infinity} x^(2i)/(2i)! */
                        BigDecimal resul = BigDecimal.ONE ;

                        /* x^i */
                        BigDecimal xpowi = BigDecimal.ONE ;

                        /* 2i factorial */
                        BigInteger ifac = BigInteger.ONE ;

                        /* The absolute error in the result is the error in x^2/2 which is x times the error in x.
                        */
                        double xUlpDbl = 0.5*x.ulp().doubleValue()*x.doubleValue() ;

                        /* The error in the result is set by the error in x^2/2 itself, xUlpDbl.
                        * We need at most k terms to push x^(2k)/(2k)! below this value.
                        * x^(2k) < xUlpDbl; (2k)*log(x) < log(xUlpDbl);
                        */
                        int k = (int)(Math.log(xUlpDbl)/Math.log(x.doubleValue()) )/2 ;

                        /* The individual terms are all smaller than 1, so an estimate of 1.0 for
                        * the absolute value will give a safe relative error estimate for the indivdual terms
                        */
                        MathContext mcTay = new MathContext( err2prec(1.,xUlpDbl/k) ) ;
                        for(int i=1 ; ; i++)
                        {
                                /* TBD: at which precision will 2*i-1 or 2*i overflow?
                                */
                                ifac = ifac.multiply(new BigInteger(""+(2*i-1) ) ) ;
                                ifac = ifac.multiply( new BigInteger(""+(2*i)) ) ;
                                xpowi = xpowi.multiply(xhighpr).multiply(xhighpr) ;
                                BigDecimal corr = xpowi.divide(new BigDecimal(ifac),mcTay) ;
                                resul = resul.add( corr ) ;
                                if ( corr.abs().doubleValue() < 0.5*xUlpDbl )
                                        break ;
                        }
                        /* The error in the result is governed by the error in x itself.
                        */
                        MathContext mc = new MathContext( err2prec(resul.doubleValue(),xUlpDbl) ) ;
                        return resul.round(mc) ;
                }
        }
} /* BigDecimalMath.cosh */

/** The hyperbolic sine.
* @param x the argument.
* @return the sinh(x) = (exp(x)-exp(-x))/2 .
*/
static public BigDecimal sinh(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0)
                return sinh(x.negate()).negate() ;
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                if ( x.doubleValue() > 2.4 )
                {
                        /* Move closer to zero with sinh(2x)= 2*sinh(x)*cosh(x).
                        */
                        BigDecimal two = new BigDecimal(2) ;
                        BigDecimal xhalf = x.divide(two) ;
```

```
                        BigDecimal resul =  sinh(xhalf).multiply(cosh(xhalf)).multiply(two) ;
                        /* The error in the result is set by the error in x itself.
                        * The first derivative of sinh(x) is cosh(x), so the absolute error
                        * in the result is cosh(x)*errx, and the relative error is coth(x)*errx = errx/tanh(x)
                        */
                        double eps =  Math.tanh(x.doubleValue()) ;
                        MathContext mc = new MathContext( err2prec(0.5*x.ulp().doubleValue()/eps) ) ;
                        return resul.round(mc) ;
                }
                else
                {
                        BigDecimal xhighpr = scalePrec(x,2) ;
                        /* Simple Taylor expansion, sum_{i=0..infinity} x^(2i+1)/(2i+1)! */
                        BigDecimal resul = xhighpr ;

                        /* x^i */
                        BigDecimal xpowi = xhighpr ;

                        /* 2i+1 factorial */
                        BigInteger ifac = BigInteger.ONE ;

                        /* The error in the result is set by the error in x itself.
                        */
                        double xUlpDbl = x.ulp().doubleValue() ;

                        /* The error in the result is set by the error in x itself.
                        * We need at most k terms to squeeze x^(2k+1)/(2k+1)! below this value.
                        * x^(2k+1) < x.ulp; (2k+1)*log10(x) < -x.precision; 2k*log10(x)< -x.precision;
                        * 2k*(-log10(x)) > x.precision; 2k*log10(1/x) > x.precision
                        */
                        int k = (int)(x.precision()/Math.log10(1.0/xhighpr.doubleValue()))/2 ;
                        MathContext mcTay = new MathContext( err2prec(x.doubleValue(),xUlpDbl/k) ) ;
                        for(int i=1 ; ; i++)
                        {
                                /* TBD: at which precision will 2*i or 2*i+1 overflow?
                                */
                                ifac = ifac.multiply(new BigInteger(""+(2*i) ) ) ;
                                ifac = ifac.multiply( new BigInteger(""+(2*i+1)) ) ;
                                xpowi = xpowi.multiply(xhighpr).multiply(xhighpr) ;
                                BigDecimal corr = xpowi.divide(new BigDecimal(ifac),mcTay) ;
                                resul = resul.add( corr ) ;
                                if ( corr.abs().doubleValue() < 0.5*xUlpDbl )
                                        break ;
                        }
                        /* The error in the result is set by the error in x itself.
                        */
                        MathContext mc = new MathContext(x.precision() ) ;
                        return resul.round(mc) ;
                }
        }
} /* BigDecimalMath.sinh */

/** The hyperbolic tangent.
* @param x The argument.
* @return The tanh(x) = sinh(x)/cosh(x).
*/
static public BigDecimal tanh(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ZERO) < 0)
                return tanh(x.negate()).negate() ;
        else if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                BigDecimal xhighpr = scalePrec(x,2) ;

                /* tanh(x) = (1-e^(-2x))/(1+e^(-2x)) .
                */
                BigDecimal exp2x = exp( xhighpr.multiply(new BigDecimal(-2)) ) ;

                /* The error in tanh x is err(x)/cosh^2(x).
                */
                double eps = 0.5*x.ulp().doubleValue()/Math.pow( Math.cosh(x.doubleValue()), 2.0 ) ;
                MathContext mc = new MathContext( err2prec(Math.tanh(x.doubleValue()),eps) ) ;
                return BigDecimal.ONE.subtract(exp2x).divide( BigDecimal.ONE.add(exp2x), mc) ;
        }
} /* BigDecimalMath.tanh */

/** The inverse hyperbolic sine.
* @param x The argument.
* @return The arcsinh(x) .
*/
static public BigDecimal asinh(final BigDecimal x)
```

```
{
        if ( x.compareTo(BigDecimal.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                BigDecimal xhighpr = scalePrec(x,2) ;

                /* arcsinh(x) = log(x+hypot(1,x))
                */
                BigDecimal logx = log(hypot(1,xhighpr).add(xhighpr)) ;

                /* The absolute error in arcsinh x is err(x)/sqrt(1+x^2)
                */
                double xDbl = x.doubleValue() ;
                double eps = 0.5*x.ulp().doubleValue()/Math.hypot(1.,xDbl ) ;
                MathContext mc = new MathContext( err2prec(logx.doubleValue(),eps) ) ;
                return logx.round(mc) ;
        }
} /* BigDecimalMath.asinh */

/** The inverse hyperbolic cosine.
* @param x The argument.
* @return The arccosh(x) .
*/
static public BigDecimal acosh(final BigDecimal x)
{
        if ( x.compareTo(BigDecimal.ONE) < 0 )
                throw new ArithmeticException("Out of range argument cosh "+x.toString() ) ;
        else if ( x.compareTo(BigDecimal.ONE) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                BigDecimal xhighpr = scalePrec(x,2) ;

                /* arccosh(x) = log(x+sqrt(x^2-1))
                */
                BigDecimal logx = log( sqrt(xhighpr.pow(2).subtract(BigDecimal.ONE) ) .add(xhighpr)) ;

                /* The absolute error in arcsinh x is err(x)/sqrt(x^2-1)
                */
                double xDbl = x.doubleValue() ;
                double eps = 0.5*x.ulp().doubleValue()/Math.sqrt(xDbl*xDbl-1.) ;
                MathContext mc = new MathContext( err2prec(logx.doubleValue(),eps) ) ;
                return logx.round(mc) ;
        }
} /* BigDecimalMath.acosh */

/** The Gamma function.
* @param x The argument.
* @return Gamma(x).
*/
static public BigDecimal Gamma(final BigDecimal x)
{
        /* reduce to interval near 1.0 with the functional relation, Abramowitz-Stegun 6.1.33
        */
        if ( x.compareTo(BigDecimal.ZERO) < 0 )
                return divideRound(Gamma( x.add(BigDecimal.ONE) ),x) ;
        else if ( x.doubleValue() > 1.5 )
        {
                /* Gamma(x) = Gamma(xmin+n) = Gamma(xmin)*Pochhammer(xmin,n).
                */
                int n = (int) ( x.doubleValue()-0.5 );
                BigDecimal xmin1 = x.subtract(new BigDecimal(n)) ;
                return multiplyRound(Gamma(xmin1), pochhammer(xmin1,n) ) ;
        }
        else
        {
                /* apply Abramowitz-Stegun 6.1.33
                */
                BigDecimal z = x.subtract(BigDecimal.ONE) ;

                /* add intermediately 2 digits to the partial sum accumulation
                */
                z = scalePrec(z,2) ;
                MathContext mcloc = new MathContext(z.precision()) ;

                /* measure of the absolute error is the relative error in the first, logarithmic term
                */
                double eps = x.ulp().doubleValue()/x.doubleValue() ;

                BigDecimal resul = log( scalePrec(x,2)).negate() ;

                if ( x.compareTo(BigDecimal.ONE) != 0 )
```

```
		{
			BigDecimal gammCompl = BigDecimal.ONE.subtract(gamma(mcloc) ) ;
			resul = resul.add( multiplyRound(z,gammCompl) ) ;
			for(int n=2; ;n++)
			{
				/* multiplying z^n/n by zeta(n-1) means that the two relative errors add.
				* so the requirement in the relative error of zeta(n)-1 is that this is somewhat
				* smaller than the relative error in z^n/n (the absolute error of thelatter  is the
				* absolute error in z)
				*/
				BigDecimal c = divideRound(z.pow(n,mcloc),n) ;
				MathContext m = new MathContext( err2prec(n*z.ulp().doubleValue()/2./z.doubleValue()) ) ;
				c = c.round(m) ;

				/* At larger n, zeta(n)-1 is roughly 1/2^n. The product is c/2^n.
				* The relative error in c is c.ulp/2/c . The error in the product should be small versus eps/10.
				* Error from 1/2^n is c*err(sigma-1).
				* We need a relative error of zeta-1 of the order of c.ulp/50/c. This is an absolute
				* error in zeta-1 of c.ulp/50/c/2^n, and also the absolute error in zeta, because zeta is
				* of the order of 1.
				*/
				if ( eps/100./c.doubleValue() < 0.01 )
					m = new MathContext( err2prec(eps/100./c.doubleValue()) ) ;
				else
					m = new MathContext( 2) ;
				/* zeta(n) -1 */
				BigDecimal zetm1 = zeta(n,m).subtract(BigDecimal.ONE) ;
				c = multiplyRound(c,zetm1) ;

				if ( n % 2 == 0 )
					resul = resul.add(c) ;
				else
					resul = resul.subtract(c) ;

				/* alternating sum, so truncating as eps is reached suffices
				*/
				if ( Math.abs(c.doubleValue()) < eps)
					break;
			}
		}

		/* The relative error in the result is the absolute error in the
		* input variable times the digamma (psi) value at that point.
		*/
		double psi = 0.5772156649 ;
		double zdbl = z.doubleValue() ;
		for( int n=1 ; n < 5 ; n++)
			psi += zdbl/n/(n+zdbl) ;
		eps = psi* x.ulp().doubleValue()/2. ;
		mcloc = new MathContext( err2prec(eps) ) ;
		return exp(resul).round(mcloc) ;
	}
} /* BigDecimalMath.gamma */

/** Pochhammer's  function.
* @param x The main argument.
* @param n The non-negative index.
* @return (x)_n = x(x+1)(x+2)*...*(x+n-1).
*/
static public BigDecimal pochhammer(final BigDecimal x, final int n)
{
	/* reduce to interval near 1.0 with the functional relation, Abramowitz-Stegun 6.1.33
	*/
	if ( n < 0 )
		throw new ProviderException("Unimplemented pochhammer with negative index "+n) ;
	else if ( n == 0 )
		return BigDecimal.ONE ;
	else
	{
		/* internally two safety digits
		*/
		BigDecimal xhighpr = scalePrec(x,2) ;
		BigDecimal resul = xhighpr ;

		double xUlpDbl = x.ulp().doubleValue() ;
		double xDbl = x.doubleValue() ;
		/* relative error of the result is the sum of the relative errors of the factors
		*/
		double eps = 0.5*xUlpDbl/Math.abs(xDbl) ;
		for (int i =1 ; i < n ; i++)
		{
			eps += 0.5*xUlpDbl/Math.abs(xDbl+i) ;
```

```
                        resul = resul.multiply( xhighpr.add(new BigDecimal(i)) ) ;
                        final MathContext mcloc = new MathContext(4+ err2prec(eps) ) ;
                        resul = resul.round(mcloc) ;
                }
                return resul.round(new MathContext(err2prec(eps)) )  ;
        }
} /* BigDecimalMath.pochhammer */

/** Reduce value to the interval [0,2*Pi].
* @param x the original value
* @return the value modulo 2*pi in the interval from 0 to 2*pi.
*/
static public BigDecimal mod2pi(BigDecimal x)
{
        /* write x= 2*pi*k+r with the precision in r defined by the precision of x and not
        * compromised by the precision of 2*pi, so the ulp of 2*pi*k should match the ulp of x.
        * First get a guess of k to figure out how many digits of 2*pi are needed.
        */
        int k = (int)(0.5*x.doubleValue()/Math.PI) ;

        /* want to have err(2*pi*k)< err(x)=0.5*x.ulp, so err(pi) = err(x)/(4k) with two safety digits
        */
        double err2pi ;
        if ( k != 0 )
                err2pi = 0.25*Math.abs(x.ulp().doubleValue()/k) ;
        else
                err2pi = 0.5*Math.abs(x.ulp().doubleValue()) ;
        MathContext mc = new MathContext( 2+err2prec(6.283,err2pi) ) ;
        BigDecimal twopi= pi(mc).multiply(new BigDecimal(2)) ;

        /* Delegate the actual operation to the BigDecimal class, which may return
        * a negative value of x was negative .
        */
        BigDecimal res = x.remainder(twopi) ;
        if ( res.compareTo(BigDecimal.ZERO) < 0 )
                res =  res.add(twopi) ;

        /* The actual precision is set by the input value, its absolute value of x.ulp()/2.
        */
        mc = new MathContext( err2prec(res.doubleValue(),x.ulp().doubleValue()/2.) ) ;
        return res.round(mc) ;
} /* mod2pi */

/** Reduce value to the interval [-Pi/2,Pi/2].
* @param x The original value
* @return The value modulo pi, shifted to the interval from -Pi/2 to Pi/2.
*/
static public BigDecimal modpi(BigDecimal x)
{
        /* write x= pi*k+r with the precision in r defined by the precision of x and not
        * compromised by the precision of pi, so the ulp of pi*k should match the ulp of x.
        * First get a guess of k to figure out how many digits of pi are needed.
        */
        int k = (int)(x.doubleValue()/Math.PI) ;

        /* want to have err(pi*k)< err(x)=x.ulp/2, so err(pi) = err(x)/(2k) with two safety digits
        */
        double errpi ;
        if ( k != 0 )
                errpi = 0.5*Math.abs(x.ulp().doubleValue()/k) ;
        else
                errpi = 0.5*Math.abs(x.ulp().doubleValue()) ;
        MathContext mc = new MathContext( 2+err2prec(3.1416,errpi) ) ;
        BigDecimal onepi= pi(mc) ;
        BigDecimal pihalf = onepi.divide(new BigDecimal(2)) ;

        /* Delegate the actual operation to the BigDecimal class, which may return
        * a negative value of x was negative .
        */
        BigDecimal res = x.remainder(onepi) ;
        if ( res.compareTo(pihalf) > 0 )
                res =  res.subtract(onepi) ;
        else if ( res.compareTo(pihalf.negate()) < 0 )
                res =  res.add(onepi) ;

        /* The actual precision is set by the input value, its absolute value of x.ulp()/2.
        */
        mc = new MathContext( err2prec(res.doubleValue(),x.ulp().doubleValue()/2.) ) ;
        return res.round(mc) ;
} /* modpi */

/** Riemann zeta function.
* @param n The positive integer argument.
```

```
* @param mc Specification of the accuracy of the result.
* @return zeta(n).
*/
static public BigDecimal zeta(final int n, final MathContext mc)
{
        if( n <= 0 )
                throw new ProviderException("Unimplemented zeta at negative argument "+n) ;
        if( n == 1 )
                throw new ArithmeticException("Pole at zeta(1) ") ;

        if( n % 2 == 0 )
        {
                /* Even indices. Abramowitz-Stegun 23.2.16. Start with 2^(n-1)*B(n)/n!
                */
                Rational b = (new Bernoulli()).at(n).abs() ;
                b = b.divide((new Factorial()).at(n)) ;
                b = b.multiply( BigInteger.ONE.shiftLeft(n-1) );

                /* to be multiplied by pi^n. Absolute error in the result of pi^n is n times
                * error in pi times pi^(n-1). Relative error is n*error(pi)/pi, requested by mc.
                * Need one more digit in pi if n=10, two digits if n=100 etc, and add one extra digit.
                */
                MathContext mcpi = new MathContext( mc.getPrecision() + (int)(Math.log10(10.0*n)) ) ;
                final BigDecimal piton = pi(mcpi).pow(n,mc) ;
                return multiplyRound( piton, b) ;
        }
        else if ( n == 3)
        {
                /* Broadhurst BBP \protect\vrule width0pt\protect\href{http://arxiv.org/abs/math/9803067}{arXiv:math/9803067}
                * Error propagation: S31 is roughly 0.087, S33 roughly 0.131
                */
                int[] a31 = {1,-7,-1,10,-1,-7,1,0} ;
                int[] a33 = {1,1,-1,-2,-1,1,1,0} ;
                BigDecimal S31 = broadhurstBBP(3,1,a31,mc) ;
                BigDecimal S33 = broadhurstBBP(3,3,a33,mc) ;
                S31 = S31.multiply(new BigDecimal(48)) ;
                S33 = S33.multiply(new BigDecimal(32)) ;
                return S31.add(S33).divide(new BigDecimal(7),mc) ;
        }
        else if ( n == 5)
        {
                /* Broadhurst BBP \protect\vrule width0pt\protect\href{http://arxiv.org/abs/math/9803067}{arXiv:math/9803067}
                * Error propagation: S51 is roughly -11.15, S53 roughly 22.165, S55 is roughly 0.031
                * 9*2048*S51/6265 = -3.28. 7*2038*S53/61651= 5.07. 738*2048*S55/61651= 0.747.
                * The result is of the order 1.03, so we add 2 digits to S51 and S52 and one digit to S55.
                */
                int[] a51 = {31,-1614,-31,-6212,-31,-1614,31,74552} ;
                int[] a53 = {173,284,-173,-457,-173,284,173,-111} ;
                int[] a55 = {1,0,-1,-1,-1,0,1,1} ;
                BigDecimal S51 = broadhurstBBP(5,1,a51, new MathContext(2+mc.getPrecision()) ) ;
                BigDecimal S53 = broadhurstBBP(5,3,a53, new MathContext(2+mc.getPrecision()) ) ;
                BigDecimal S55 = broadhurstBBP(5,5,a55, new MathContext(1+mc.getPrecision()) ) ;
                S51 = S51.multiply(new BigDecimal(18432)) ;
                S53 = S53.multiply(new BigDecimal(14336)) ;
                S55 = S55.multiply(new BigDecimal(1511424)) ;
                return S51.add(S53).subtract(S55).divide(new BigDecimal(62651),mc) ;
        }
        else
        {
                /* Cohen et al Exp Math 1 (1) (1992) 25
                */
                Rational betsum = new Rational() ;
                Bernoulli bern = new Bernoulli() ;
                Factorial fact = new Factorial() ;
                for(int npr=0 ; npr <= (n+1)/2 ; npr++)
                {
                        Rational b = bern.at(2*npr).multiply(bern.at(n+1-2*npr)) ;
                        b = b.divide(fact.at(2*npr)).divide(fact.at(n+1-2*npr)) ;
                        b = b.multiply(1-2*npr) ;
                        if ( npr % 2 ==0 )
                                betsum = betsum.add(b) ;
                        else
                                betsum = betsum.subtract(b) ;
                }
                betsum = betsum.divide(n-1) ;
                /* The first term, including the facor (2pi)^n, is essentially most
                * of the result, near one. The second term below is roughly in the range 0.003 to 0.009.
                * So the precision here is matching the precisionn requested by mc, and the precision
                * requested for 2*pi is in absolute terms adjusted.
                */
                MathContext mcloc = new MathContext( 2+mc.getPrecision() + (int)(Math.log10((double)(n))) ) ;
                BigDecimal ftrm = pi(mcloc).multiply(new BigDecimal(2)) ;
                ftrm = ftrm.pow(n) ;
```

```
            ftrm = multiplyRound(ftrm, betsum.BigDecimalValue(mcloc) ) ;
            BigDecimal exps = new BigDecimal(0) ;

            /* the basic accuracy of the accumulated terms before multiplication with 2
            */
            double eps = Math.pow(10.,-mc.getPrecision()) ;

            if ( n % 4 == 3)
            {
                    /* since the argument n is at least 7 here, the drop
                    * of the terms is at rather constant pace at least 10^-3, for example
                    * 0.0018, 0.2e-7, 0.29e-11, 0.74e-15 etc for npr=1,2,3.... We want 2 times these terms
                    * fall below eps/10.
                    */
                    int kmax = mc.getPrecision()/3 ;
                    eps /= kmax ;
                    /* need an error of eps for 2/(exp(2pi)-1) = 0.0037
                    * The absolute error is 4*exp(2pi)*err(pi)/(exp(2pi)-1)^2=0.0075*err(pi)
                    */
                    BigDecimal exp2p = pi( new MathContext(3+err2prec(3.14, eps/0.0075)) ) ;
                    exp2p = exp(exp2p.multiply(new BigDecimal(2))) ;
                    BigDecimal c =  exp2p.subtract(BigDecimal.ONE) ;
                    exps = divideRound(1,c) ;
                    for(int npr=2 ; npr<= kmax ; npr++)
                    {
                            /* the error estimate above for npr=1 is the worst case of
                            * the absolute error created by an error in 2pi. So we can
                            * safely re-use the exp2p value computed above without
                            * reassessment of its error.
                            */
                            c =  powRound(exp2p,npr).subtract(BigDecimal.ONE) ;
                            c =  multiplyRound(c, (new BigInteger(""+npr)).pow(n) ) ;
                            c =  divideRound(1,c) ;
                            exps = exps.add(c) ;
                    }
            }
            else
            {
                    /* since the argument n is at least 9 here, the drop
                    * of the terms is at rather constant pace at least 10^-3, for example
                    * 0.0096, 0.5e-7, 0.3e-11, 0.6e-15 etc. We want these terms
                    * fall below eps/10.
                    */
                    int kmax = (1+mc.getPrecision())/3 ;
                    eps /= kmax ;
                    /* need an error of eps for 2/(exp(2pi)-1)*(1+4*Pi/8/(1-exp(-2pi)) = 0.0096
                    * at k=7 or = 0.00766 at k=13 for example.
                    * The absolute error is 0.017*err(pi) at k=9, 0.013*err(pi) at k=13, 0.012 at k=17
                    */
                    BigDecimal twop = pi( new MathContext(3+err2prec(3.14, eps/0.017)) ) ;
                    twop = twop.multiply(new BigDecimal(2)) ;
                    BigDecimal exp2p = exp(twop) ;
                    BigDecimal c =  exp2p.subtract(BigDecimal.ONE) ;
                    exps = divideRound(1,c) ;
                    c =  BigDecimal.ONE.subtract(divideRound(1,exp2p)) ;
                    c =  divideRound(twop,c).multiply(new BigDecimal(2)) ;
                    c =  divideRound(c,n-1).add(BigDecimal.ONE) ;
                    exps = multiplyRound(exps,c) ;
                    for(int npr=2 ; npr<= kmax ; npr++)
                    {
                            c =  powRound(exp2p,npr).subtract(BigDecimal.ONE) ;
                            c =  multiplyRound(c, (new BigInteger(""+npr)).pow(n) ) ;

                            BigDecimal d =  divideRound(1, exp2p.pow(npr) ) ;
                            d =  BigDecimal.ONE.subtract(d) ;
                            d =  divideRound(twop,d).multiply(new BigDecimal(2*npr)) ;
                            d =  divideRound(d,n-1).add(BigDecimal.ONE) ;

                            d = divideRound(d,c) ;

                            exps = exps.add(d) ;
                    }
            }
            exps = exps.multiply(new BigDecimal(2)) ;
            return ftrm.subtract(exps,mc) ;
    }
} /* zeta */

/** Riemann zeta function.
* @param n The positive integer argument.
* @return zeta(n)-1.
*/
static public double zeta1(final int n)
```

```
        {
                /* precomputed static table in double precision
                */
                final double[] zmin1 = {0.,0.,
6.44934066848226436472415166646e-01,
2.02056903159594285399738161511e-01,8.23232337111381915160036965e-02,
3.69277551433699263313654864e-02,1.73430619844491397145179297e-02,
8.34927738192282683979754985e-03,4.07735619794433937868523850e-03,
2.00839282608221441785276923e-03,9.94575127818085337145958900e-04,
4.94188604119464558702282526e-04,2.46086553308048298637998047e-04,
1.22713347578489146751836526e-04,6.12481350587048292585451051e-05,
3.05882363070204935517285106e-05,1.52822594086518717325714876e-05,
7.63719763789976227360029356e-06,3.81729326499983985646164462e-06,
1.90821271655393892565695779e-06,9.53962033872796113152038683e-07,
4.76932986787806463116719604e-07,2.38450502727732990003648186e-07,
1.19219925965311073067788718e-07,5.96081890512594796124402079e-08,
2.98035035146522801860637050e-08,1.49015548283650412346585066e-08,
7.45071178983542949198100417e-09,3.72533402478845705481920401e-09,
1.86265972351304900640390994e-09,9.31327432419668182871764735e-10,
4.65662906503378407298923325e-10,2.32831183367650549200145597e-10,
1.16415501727005197759297383e-10,5.82077208790270088924368598e-11,
2.91038504449709968692942522e-11,1.45519218910419842359296322e-11,
7.27595983505748101452086901e-12,3.63797954737865119023723635e-12,
1.81898965030706594758483210e-12,9.09494784026388928253311838e-13,
4.54747378304215402679911202e-13,2.27373684582465251522682157e-13,
1.13686840768022784934910483e-13,5.68434198762758560927718296e-14,
2.84217097688930185545507370e-14,1.42108548280316067698343071e-14,
7.10542739521085271287735448e-15,3.55271369133711367329846953e-15,
1.77635684357912032747334901e-15,8.88178421093081590309609138e-16,
4.44089210314381336419777094e-16,2.22044605079804198399932009e-16,
1.11022302514106613372054457e-16,5.55111512484548124372373659e-17,
2.77555756213612417258163245e-17,1.38777878097252327628390949e-17,
6.93889390454415369744608532e-18,3.46944695216592262474427149e-18,
1.73472347604757657204897297e-18,8.67361738011993372834205506e-19,
4.33680869002065048749702356e-19,2.16840434499721978501391016e-19,
1.08420217249424140630127111e-19,5.42101086245664541091870040e-20,
2.71050543122346883195462131e-20,1.35525271561010116458148523e-20,
6.77626357804518909799529874e-21,3.38813178902079681808570310e-21,
1.69406589450979916540649274e-21,8.47032947254699834824699260e-22,
4.23516473627283334786227048e-22,2.11758236813619473184420944e-22,
1.05879118406802338522650015e-22,5.29395592033987032381391230e-23,
2.64697796016985296113411668e-23,1.32348898008489908030945102e-23,
6.61744490042440406735524533e-24,3.30872245021217158894695638e-24,
1.65436122510607564622992367e-24,8.27180612553034440367110561e-25,
4.13590306276516092600938245e-25,2.06795153138257670439596791e-25,
1.03397576569128709932840955e-25,5.16987882845643132041013321e-26,
2.58493941422821426812776177e-26,1.29246970711410667003811261e-26,
6.46234853557053180343800216e-27,3.23117426778526538613481411e-27,
1.61558713389263252120601140e-27,8.07793566946316203315873818e-28,
4.03896783473158082562226281e-28,2.01948391736579034915876264e-28,
1.00974195868289515336192507e-28,5.04870979341447569608477117e-29,
2.52435489670723782446743419e-29,1.26217744835361890437539996e-29,
6.31088724176809449568260939e-30,3.15544362088404723910984122e-30,
1.57772181044202361664443278e-30,7.88860905221011807352053780e-31
                } ;
                if( n <= 0 )
                        throw new ProviderException("Unimplemented zeta at negative argument "+n) ;
                if( n == 1 )
                        throw new ArithmeticException("Pole at zeta(1) ") ;

                if( n < zmin1.length )
                        /* look it up if available */
                        return zmin1[n] ;
                else
                {
                        /* Result is roughly 2^(-n), desired accuracy 18 digits. If zeta(n) is computed, the equivalent accuracy
                        * in relative units is higher, because zeta is around 1.
                        */
                        double eps = 1.e-18*Math.pow(2.,(double)(-n) ) ;
                        MathContext mc = new MathContext( err2prec(eps) ) ;
                        return zeta(n,mc).subtract(BigDecimal.ONE).doubleValue() ;
                }
        } /* zeta */


        /** Broadhurst ladder sequence.
        * @param a The vector of 8 integer arguments
        * @param mc Specification of the accuracy of the result
        * @return S_(n,p)(a)
        * @see \protect\vrule width0pt\protect\href{http://arxiv.org/abs/math/9803067}{arXiv:math/9803067}
        */
        static protected BigDecimal broadhurstBBP(final int n, final int p, final int a[], MathContext mc)
```

```
{
        /* Explore the actual magnitude of the result first with a quick estimate.
        */
        double x = 0.0 ;
        for(int k=1; k < 10 ; k++)
                x += a[ (k-1) % 8]/Math.pow(2., p*(k+1)/2)/Math.pow((double)k,n) ;

        /* Convert the relative precision and estimate of the result into an absolute precision.
        */
        double eps = prec2err(x,mc.getPrecision()) ;

        /* Divide this through the number of terms in the sum to account for error accumulation
        * The divisor 2^(p(k+1)/2) means that on the average each 8th term in k has shrunk by
        * relative to the 8th predecessor by 1/2^(4p).  1/2^(4pc) = 10^(-precision) with c the 8term
        * cycles yields c=log_2( 10^precision)/4p = 3.3*precision/4p  with k=8c
        */
        int kmax= (int)(6.6*mc.getPrecision()/p) ;

        /* Now eps is the absolute error in each term */
        eps /= kmax ;
        BigDecimal res = BigDecimal.ZERO ;
        for(int c =0 ; ; c++)
        {
                Rational r = new Rational() ;
                for (int k=0; k < 8 ; k++)
                {
                        Rational tmp = new Rational(new BigInteger(""+a[k]),(new BigInteger(""+(1+8*c+k))).pow(n)) ;
                        /* floor( (pk+p)/2)
                        */
                        int pk1h = p*(2+8*c+k)/2 ;
                        tmp = tmp.divide( BigInteger.ONE.shiftLeft(pk1h) ) ;
                        r = r.add(tmp) ;
                }

                if ( Math.abs(r.doubleValue()) < eps)
                        break;
                MathContext mcloc = new MathContext( 1+err2prec(r.doubleValue(),eps) ) ;
                res = res.add( r.BigDecimalValue(mcloc) ) ;
        }
        return res.round(mc) ;
} /* broadhurstBBP */



/** Add and round according to the larger of the two ulp's.
* @param x The left summand
* @param y The right summand
* @return The sum x+y.
*/
static public BigDecimal addRound(final BigDecimal x, final BigDecimal y)
{
        BigDecimal resul = x.add(y) ;
        /* The estimation of the absolute error in the result is |err(y)|+|err(x)|
        */
        double errR = Math.abs( y.ulp().doubleValue()/2. ) + Math.abs( x.ulp().doubleValue()/2. ) ;
        MathContext mc = new MathContext( err2prec(resul.doubleValue(),errR) ) ;
        return resul.round(mc) ;
} /* addRound */

/** Subtract and round according to the larger of the two ulp's.
* @param x The left term.
* @param y The right term.
* @return The difference x-y.
*/
static public BigDecimal subtractRound(final BigDecimal x, final BigDecimal y)
{
        BigDecimal resul = x.subtract(y) ;
        /* The estimation of the absolute error in the result is |err(y)|+|err(x)|
        */
        double errR = Math.abs( y.ulp().doubleValue()/2. ) + Math.abs( x.ulp().doubleValue()/2. ) ;
        MathContext mc = new MathContext( err2prec(resul.doubleValue(),errR) ) ;
        return resul.round(mc) ;
} /* subtractRound */

/** Multiply and round.
* @param x The left factor.
* @param y The right factor.
* @return The product x*y.
*/
static public BigDecimal multiplyRound(final BigDecimal x, final BigDecimal y)
{
        BigDecimal resul = x.multiply(y) ;
```

```
        /* The estimation of the relative error in the result is the sum of the relative
        * errors |err(y)/y|+|err(x)/x|
        */
        MathContext mc = new MathContext( Math.min(x.precision(),y.precision()) ) ;
        return resul.round(mc) ;
} /* multiplyRound */

/** Multiply and round.
* @param x The left factor.
* @param f The right factor.
* @return The product x*f.
*/
static public BigDecimal multiplyRound(final BigDecimal x, final Rational f)
{
        if (  f.compareTo(BigInteger.ZERO) == 0 )
                return BigDecimal.ZERO ;
        else
        {
                /* Convert the rational value with two digits of extra precision
                */
                MathContext mc = new MathContext( 2+x.precision() ) ;
                BigDecimal fbd = f.BigDecimalValue(mc) ;

                /* and the precision of the product is then dominated by the precision in x
                */
                return multiplyRound(x,fbd) ;
        }
}

/** Multiply and round.
* @param x The left factor.
* @param n The right factor.
* @return The product x*n.
*/
static public BigDecimal multiplyRound(final BigDecimal x, final int n)
{
        BigDecimal resul = x.multiply(new BigDecimal(n)) ;
        /* The estimation of the absolute error in the result is |n*err(x)|
        */
        MathContext mc = new MathContext( n != 0 ? x.precision(): 0 ) ;
        return resul.round(mc) ;
}

/** Multiply and round.
* @param x The left factor.
* @param n The right factor.
* @return the product x*n
*/
static public BigDecimal multiplyRound(final BigDecimal x, final BigInteger n)
{
        BigDecimal resul = x.multiply(new BigDecimal(n)) ;
        /* The estimation of the absolute error in the result is |n*err(x)|
        */
        MathContext mc = new MathContext( n.compareTo(BigInteger.ZERO) != 0 ? x.precision(): 0 ) ;
        return resul.round(mc) ;
}

/** Divide and round.
* @param x The numerator
* @param y The denominator
* @return the divided x/y
*/
static public BigDecimal divideRound(final BigDecimal x, final BigDecimal y)
{
        /* The estimation of the relative error in the result is |err(y)/y|+|err(x)/x|
        */
        MathContext mc = new MathContext( Math.min(x.precision(),y.precision()) ) ;
        return x.divide(y,mc) ;
}

/** Divide and round.
* @param x The numerator
* @param n The denominator
* @return the divided x/n
*/
static public BigDecimal divideRound(final BigDecimal x, final int n)
{
        /* The estimation of the relative error in the result is |err(x)/x|
        */
        MathContext mc = new MathContext( x.precision() ) ;
        return x.divide(new BigDecimal(n),mc) ;
}
```

```
/** Divide and round.
* @param x The numerator
* @param n The denominator
* @return the divided x/n
*/
static public BigDecimal divideRound(final BigDecimal x, final BigInteger n)
{
        /* The estimation of the relative error in the result is |err(x)/x|
        */
        MathContext mc = new MathContext( x.precision() ) ;
        return x.divide(new BigDecimal(n),mc) ;
}

/** Divide and round.
* @param n The numerator
* @param x The denominator
* @return the divided n/x
*/
static public BigDecimal divideRound(final BigInteger n, final BigDecimal x)
{
        /* The estimation of the relative error in the result is |err(x)/x|
        */
        MathContext mc = new MathContext( x.precision() ) ;
        return new BigDecimal(n).divide(x,mc) ;
}

/** Divide and round.
* @param n The numerator.
* @param x The denominator.
* @return the divided n/x.
*/
static public BigDecimal divideRound(final int n, final BigDecimal x)
{
        /* The estimation of the relative error in the result is |err(x)/x|
        */
        MathContext mc = new MathContext( x.precision() ) ;
        return new BigDecimal(n).divide(x,mc) ;
}

/** Append decimal zeros to the value. This returns a value which appears to have
* a higher precision than the input.
* @param x The input value
* @param d The (positive) value of zeros to be added as least significant digits.
* @return The same value as the input but with increased (pseudo) precision.
*/
static public BigDecimal scalePrec(final BigDecimal x, int d)
{
        return x.setScale(d+x.scale()) ;
}

/** Boost the precision by appending decimal zeros to the value. This returns a value which appears to have
* a higher precision than the input.
* @param x The input value
* @param mc The requirement on the minimum precision on return.
* @return The same value as the input but with increased (pseudo) precision.
*/
static public BigDecimal scalePrec(final BigDecimal x, final MathContext mc)
{
        final int diffPr = mc.getPrecision() - x.precision() ;
        if ( diffPr > 0 )
                return scalePrec(x, diffPr) ;
        else
                return x ;
} /* BigDecimalMath.scalePrec */

/** Convert an absolute error to a precision.
* @param x The value of the variable
* @param xerr The absolute error in the variable
* @return The number of valid digits in x.
*    The value is rounded down, and on the pessimistic side for that reason.
*/
static public int err2prec(BigDecimal x, BigDecimal xerr)
{
        return err2prec( xerr.divide(x,MathContext.DECIMAL64).doubleValue() );
}

/** Convert an absolute error to a precision.
* @param x The value of the variable
*    The value returned depends only on the absolute value, not on the sign.
* @param xerr The absolute error in the variable
*    The value returned depends only on the absolute value, not on the sign.
* @return The number of valid digits in x.
*    Derived from the representation x+- xerr, as if the error was represented
```

```
	*    in a "half width" (half of the error bar) form.
	*    The value is rounded down, and on the pessimistic side for that reason.
	*/
	static public int err2prec(double x, double xerr)
	{
		/* Example: an error of xerr=+-0.5 at x=100 represents 100+-0.5 with
		* a precision = 3 (digits).
		*/
		return 1+(int)(Math.log10(Math.abs(0.5*x/xerr) ) );
	}

	/** Convert a relative error to a precision.
	* @param xerr The relative error in the variable.
	*    The value returned depends only on the absolute value, not on the sign.
	* @return The number of valid digits in x.
	*    The value is rounded down, and on the pessimistic side for that reason.
	*/
	static public int err2prec(double xerr)
	{
		/* Example: an error of xerr=+-0.5 a precision of 1 (digit), an error of
		* +-0.05 a precision of 2 (digits)
		*/
		return 1+(int)(Math.log10(Math.abs(0.5/xerr) ) );
	}

	/** Convert a precision (relative error) to an absolute error.
	*    The is the inverse functionality of err2prec().
	* @param x The value of the variable
	*    The value returned depends only on the absolute value, not on the sign.
	* @param prec The number of valid digits of the variable.
	* @return the absolute error in x.
	*    Derived from the an accuracy of one half of the ulp.
	*/
	static public double prec2err(final double x, final int prec)
	{
		return 5.*Math.abs(x)*Math.pow(10.,-prec) ;
	}

} /* BigDecimalMath */
```

---

[1] Abramowitz, M., and I. A. Stegun (eds.), 1972, *Handbook of Mathematical Functions* (Dover Publications, New York), 9th edition, ISBN 0-486-61272-4.
[2] Broadhurst, D. J., 1998, arXiv:math/9803067 [math.CA] .
[3] Cohen, H., L. Lewin, and D. Zagier, 1992, Exp. Math **1**(1), 25.
[4] Dilcher, K., 2005, Aequat. Math. **48**(1), 55.
[5] Fousse, L., G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, 2007, ACM Trans. Math. Softw. **33**(2), # 13.
[6] Gould, H. W., 1972, Amer. Math. Monthly **79**(1), 44.
[7] McCarthy, D. P., 1986, Math. Comp. **46**(174), 603.
[8] Sloane, N. J. A., 2003, Notices Am. Math. Soc. **50**(8), 912, http://www.research.att.com/˜njas/sequences/, URL `http://www.research.att.com/~njas/sequences/`.
[9] Spira, R., 1971, Math. Comp. **25**(114), 317.
[10] Wrench Jr., J. W., 1968, Math. Comput. **22**(103), 617.