The Yacas Book of Algorithms

by the YACAS team [1]

YACAS version: 1.3.6
generated on December 27, 2014

This book is a detailed description of the algorithms used in the Yacas system for exact symbolic and arbitrary-precision numerical computations. Very few of these algorithms are new, and most are well-known. The goal of this book is to become a compendium of all relevant issues of design and implementation of these algorithms.

[1]This text is part of the YACAS software package. Copyright 2000–2002. Principal documentation authors: Ayal Zwi Pinkus, Serge Winitzki, Jitse Niesen. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Contents

1 Symbolic algebra algorithms **3**
- 1.1 Sparse representations . . . 3
- 1.2 Implementation of multivariate polynomials . . . 4
- 1.3 Integration . . . 5
- 1.4 Transforms . . . 6
- 1.5 Finding real roots of polynomials . . . 7

2 Number theory algorithms **10**
- 2.1 Euclidean GCD algorithms . . . 10
- 2.2 Prime numbers: the Miller-Rabin test and its improvements . . . 10
- 2.3 Factorization of integers . . . 11
- 2.4 The Jacobi symbol . . . 12
- 2.5 Integer partitions . . . 12
- 2.6 Miscellaneous functions . . . 13
- 2.7 Gaussian integers . . . 13

3 A simple factorization algorithm for univariate polynomials **15**
- 3.1 Modular arithmetic . . . 15
- 3.2 Factoring using modular arithmetic . . . 16
- 3.3 Preparing the polynomial for factorization . . . 16
- 3.4 Definition of division of polynomials . . . 16
- 3.5 Determining possible factors modulo 2 . . . 16
- 3.6 Determining factors modulo 2^n given a factorization modulo 2 . . . 17
- 3.7 Efficiently deciding if a polynomial divides another . . . 17
- 3.8 Extending the algorithm . . . 17
- 3.9 Newton iteration . . . 18

4 Numerical algorithms I: basic methods **20**
- 4.1 Adaptive function plotting . . . 20
- 4.2 Surface plotting . . . 21
- 4.3 Parametric plots . . . 22
- 4.4 The cost of arbitrary-precision computations . . . 22
- 4.5 Estimating convergence of a series . . . 23
- 4.6 Estimating the round-off error . . . 23
- 4.7 Basic arbitrary-precision arithmetic . . . 24
- 4.8 How many digits of $\sin \exp (\exp (1000))$ do we need? . . . 25
- 4.9 Continued fractions . . . 26
- 4.10 Estimating convergence of continued fractions . . . 29
- 4.11 Newton's method and its improvements . . . 32
- 4.12 Fast evaluation of Taylor series . . . 35
- 4.13 Using asymptotic series for calculations . . . 37
- 4.14 The AGM sequence algorithms . . . 37
- 4.15 The binary splitting method . . . 38

5 Numerical algorithms II: elementary functions **40**
5.1 Powers . . . 40
5.2 Roots . . . 41
5.3 Logarithm . . . 43
5.4 Exponential . . . 46
5.5 Calculation of π . . . 47
5.6 Trigonometric functions . . . 49
5.7 Inverse trigonometric functions . . . 49
5.8 Factorials and binomial coefficients . . . 50
5.9 Classical orthogonal polynomials: general case . . . 51
5.10 Classical orthogonal polynomials: special cases . . . 52
5.11 Series of orthogonal polynomials . . . 53

6 Numerical algorithms III: special functions **54**
6.1 Euler's Gamma function . . . 54
6.2 Euler's constant γ . . . 56
6.3 Catalan's constant G . . . 57
6.4 Riemann's Zeta function . . . 58
6.5 Lambert's W function . . . 60
6.6 Bessel functions . . . 60
6.7 Bernoulli numbers and polynomials . . . 61
6.8 Error function $\operatorname{erf} x$ and related functions . . . 63

7 References **65**

8 GNU Free Documentation License **66**

Chapter 1

Symbolic algebra algorithms

1.1 Sparse representations

The sparse tree data structure

Yacas has a sparse tree object for use as a storage for storing (key,value) pairs for which the following properties hold:

- (key, value1) + (key, value2) = (key, value1+value2)

In addition, for multiplication the following rule is obeyed:

- (key1, value1) * (key2, value2) = (key1+key2, value1*value2)

The last is optional. For multivariate polynomials (described elsewhere) both hold, but for matrices, only the addition property holds. The function `MultiplyAddSparseTrees` (described below) should not be used in these cases.

Internal structure

A key is defined to be a list of integer numbers (n_1, ..., n_m). Thus for a two-dimensional key, one item in the sparse tree database could be reflected as the (key,value) pair `{{1,2},3}`, which states that element `(1,2)` has value `3`. (Note: this is not the way it is stored in the database!).

The storage is recursive. The sparse tree begins with a list of objects `{n1,tree1}` for values of `n1` for the first item in the key. The `tree1` part then contains a sub-tree for all the items in the database for which the value of the first item in the key is `n1`.

The above single element could be created with

```
In> r:=CreateSparseTree({1,2},3)
Out> {{1,{{2,3}}}};
```

`CreateSparseTree` makes a database with exactly one item. Items can now be obtained from the sparse tree with `SparseTreeGet`.

```
In> SparseTreeGet({1,2},r)
Out> 3;
In> SparseTreeGet({1,3},r)
Out> 0;
```

And values can also be set or changed:

```
In> SparseTreeSet({1,2},r,Current+5)
Out> 8;
In> r
Out> {{1,{{2,8}}}};
In> SparseTreeSet({1,3},r,Current+5)
Out> 5;
In> r
Out> {{1,{{3,5},{2,8}}}};
```

The variable `Current` represents the current value, and can be used to determine the new value. `SparseTreeSet` destructively modifies the original, and returns the new value. If the key pair was not found, it is added to the tree.

The sparse tree can be traversed, one element at a time, with `SparseTreeScan`:

```
In> SparseTreeScan(Hold({{k,v},Echo({k,v})}),2,r)
{1,3} 5
{1,2} 8
```

An example of the use of this function could be multiplying a sparse matrix with a sparse vector, where the entire matrix can be scanned with `SparseTreeScan`, and each non-zero matrix element $A_{(i,j)}$ can then be multiplied with a vector element v_j, and the result added to a sparse vector w_i, using the `SparseTreeGet` and `SparseTreeSet` functions. Multiplying two sparse matrices would require two nested calls to `SparseTreeScan` to multiply every item from one matrix with an element from the other, and add it to the appropriate element in the resulting sparse matrix.

When the matrix elements $A_{(i,j)}$ are defined by a function $f(i,j)$ (which can be considered a dense representation), and it needs to be multiplied with a sparse vector v_j, it is better to iterate over the sparse vector v_j. Representation defines the most efficient algorithm to use in this case.

The API to sparse trees is:

- `CreateSparseTree(coefs,fact)` - Create a sparse tree with one monomial, where 'coefs' is the key, and 'fact' the value. 'coefs' should be a list of integers.
- `SparseTreeMap(op,depth,tree)` - Walk over the sparse tree, one element at a time, and apply the function "op" on the arguments (key,value). The 'value' in the tree is replaced by the value returned by the `op` function. 'depth' signifies the dimension of the tree (number of indices in the key).
- `SparseTreeScan(op,depth,tree)` - Same as SparseTreeMap, but without changing elements.
- `AddSparseTrees(depth,x,y)`, `MultiplyAddSparseTrees(depth,x,y,coefs,fact)` - Add sparse tree 'y' to sparse tree 'x', destructively. in the `MultiplyAdd` case, the monomials are treated as if they were multiplied by a monomial with coefficients with the (key,value) pair (coefs,fact). 'depth' signifies the dimension of the tree (number of indices in the key).
- `SparseTreeGet(key,tree)` - return value stored for key in the tree.

- `SparseTreeSet(key,tree,newvalue)` - change the value stored for the key to newvalue. If the key was not found then `newvalue` is stored as a new item. The variable `Current` is set to the old value (or zero if the key didn't exist in the tree) before evaluating `newvalue`.

1.2 Implementation of multivariate polynomials

This section describes the implementation of multivariate polynomials in Yacas.

Concepts and ideas are taken from the books [Davenport *et al.* 1989] and [von zur Gathen *et al.* 1999].

Definitions

The following definitions define multivariate polynomials, and the functions defined on them that are of interest for using such multivariates.

A *term* is an object which can be written as

$$cx_1^{n_1}x_2^{n_2}...x_m^{n_m}$$

for m variables (x_1, ..., x_m). The numbers n_m are integers. c is called a *coefficient*, and $x_1^{n_1}x_2^{n_2}...x_m^{n_m}$ a *monomial*.

A *multivariate polynomial* is taken to be a sum over terms.

We write $c_a x^a$ for a term, where a is a list of powers for the monomial, and c_a the *coefficient* of the term.

It is useful to define an ordering of monomials, to be able to determine a canonical form of a multivariate.

For the currently implemented code the *lexicographic order* has been chosen:

- first an ordering of variables is chosen, (x_1, ..., x_m)
- for the exponents of a monomial, $a = (a_1, ..., a_m)$ the lexicographic order first looks at the first exponent, a_1, to determine which of the two monomials comes first in the multivariate. If the two exponents are the same, the next exponent is considered.

This method is called *lexicographic* because it is similar to the way words are ordered in a usual dictionary.

For all algorithms (including division) there is some freedom in the ordering of monomials. One interesting advantage of the lexicographic order is that it can be implemented with a recursive data structure, where the first variable, x_1 can be treated as the main variable, thus presenting it as a univariate polynomial in x_1 with all its terms grouped together.

Other orderings can be used, by re-implementing a part of the code dealing with multivariate polynomials, and then selecting the new code to be used as a driver, as will be described later on.

Given the above ordering, the following definitions can be stated:

For a non-zero *multivariate polynomial*

$$f = \sum_{a=a_{\max}}^{a_{\min}} c_a x^a$$

with a monomial order:

1. $c_a x^a$ is a *term* of the multivariate.
2. the *multidegree* of f is $\mathrm{mdeg}(f) \equiv a_{\max}$.
3. the *leading coefficient* of f is $lc(f) \equiv c_{\mathrm{mdeg}(f)}$, for the first term with non-zero coefficient.
4. the *leading monomial* of f is $lm(f) \equiv x^{\mathrm{mdeg}(f)}$.
5. the *leading term* of f is $lt(f) \equiv lc(f)\,lm(f)$.

The above define access to the leading monomial, which is used for divisions, gcd calculations and the like. Thus an implementation needs be able to determine `{mdeg(f),lc(f)}` . Note the similarity with the (key,value) pairs described in the sparse tree section. $\mathrm{mdeg}(f)$ can be thought of as a 'key', and $lc(f)$ as a 'value'.

The *multicontent*, $\mathrm{multicont}(f)$, is defined to be a term that divides all the terms in f, and is the term described by $(\min(a), \mathrm{Gcd}(c))$, with $\mathrm{Gcd}(c)$ the GCD of all the coefficients, and $\min(a)$ the lowest exponents for each variable, occurring in f for which c is non-zero.

The *multiprimitive part* is then defined as $pp(f) \equiv f/\mathrm{multicont}(f)$.

For a multivariate polynomial, the obvious addition and (distributive) multiplication rules hold:

```
(a+b) + (c+d) := a+b+c+d

a*(b+c) := (a*b)+(a*c)
```

These are supported in the Yacas system through a multiply-add operation:

$$\mathrm{muadd}(f,t,g) \equiv f + tg.$$

This allows for both adding two polynomials ($t \equiv 1$), or multiplication of two polynomials by scanning one polynomial, and multiplying each term of the scanned polynomial with the other polynomial, and adding the result to the polynomial that will be returned. Thus there should be an efficient `muadd` operation in the system.

Representation

For the representation of polynomials, on computers it is natural to do this in an array: (a_1, a_2, ..., a_n) for a univariate polynomial, and the equivalent for multivariates. This is called a *dense* representation, because all the coefficients are stored, even if they are zero. Computers are efficient at dealing with arrays. However, in the case of multivariate polynomials, arrays can become rather large, requiring a lot of storage and processing power even to add two such polynomials. For instance, $x^{200}y^{100}z^{300} + 1$ could take 6000000 places in an array for the coefficients. Of course variables could be substituted for the single factors, $p \equiv x^{200}$ etc., but it requires an additional ad hoc step.

An alternative is to store only the terms for which the coefficients are non-zero. This adds a little overhead to polynomials that could efficiently be stored in a dense representation, but it is still little memory, whereas large sparse polynomials are stored in acceptable memory too. It is of importance to still be able to add, multiply divide and get the leading term of a multivariate polynomial, when the polynomial is stored in a sparse representation.

For the representation, the data structure containing the `(exponents,coefficient)` pair can be viewed as a database holding `(key,value)` pairs, where the list of exponents is the key, and the coefficient of the term is the value stored for that key. Thus, for a variable set `{x,y}` the list `{{1,2},3}` represents $3xy^2$.

Yacas stores multivariates internally as `MultiNomial (vars, terms)`, where `vars` is the ordered list of variables, and terms some object storing all the `(key, value)` pairs representing the terms. Note we keep the storage vague: the `terms` placeholder is implemented by other code, as a database of terms. The specific

representation can be configured at startup (this is described in more detail below).

For the current version, Yacas uses the 'sparse tree' representation, which is a recursive sparse representation. For example, for a variable set {x,y,z}, the 'terms' object contains a list of objects of form {deg,terms}, one for each degree deg for the variable 'x' occurring in the polynomial. The 'terms' part of this object is then a sub-sparse tree for the variables {y,z}.

An explicit example:

```
In> MM(3*x^2+y)
Out> MultiNomial({x,y},{{2,{{0,3}}},{0,{{1,1},
  {0,0}}}});
```

The first item in the main list is {2,{{0,3}}}, which states that there is a term of the form $x^2y^0 \cdot 3$. The second item states that there are two terms, $x^0y^1 \cdot 1$ and $x^0y^0 \cdot 0 = 0$.

This representation is sparse:

```
In> r:=MM(x^1000+x)
Out> MultiNomial({x},{{1000,1},{1,1}});
```

and allows for easy multiplication:

```
In> r*r
Out> MultiNomial({x},{{2000,1},{1001,2},
  {2,1},{0,0}});
In> NormalForm(%)
Out> x^2000+2*x^1001+x^2;
```

Internal code organization

The implementation of multivariates can be divided in three levels.

At the top level are the routines callable by the user or the rest of the system: MultiDegree, MultiDivide, MultiGcd, Groebner, etc. In general, this is the level implementing the operations actually desired.

The middle level does the book-keeping of the `MultiNomial(vars,terms)` expressions, using the functionality offered by the lowest level.

For the current system, the middle level is in `multivar.rep/sparsenomial.ys`, and it uses the sparse tree representation implemented in `sparsetree.ys`.

The middle level is called the 'driver', and can be changed, or re-implemented if necessary. For instance, in case calculations need to be done for which dense representations are actually acceptable, one could write C++ implementing above-mentioned database structure, and then write a middle-level driver using the code. The driver can then be selected at startup. In the file 'yacasinit.ys' the default driver is chosen, but this can be overridden in the `.yacasrc` file or some file that is loaded, or at the command line, as long as it is done before the multivariates module is loaded (which loads the selected driver). Driver selection is as simple as setting a global variable to contain a file name of the file implementing the driver:

```
Set(MultiNomialDriver,
  "multivar.rep/sparsenomial.ys");
```

where "multivar.rep/sparsenomial.ys" is the file implementing the driver (this is also the default driver, so the above command would not change any thing).

The choice was made for static configuration of the driver before the system starts up because it is expected that there will in general be one best way of doing it, given a certain system with a certain set of libraries installed on the operating system, and for a specific version of Yacas. The best version can then be selected at start up, as a configuration step. The advantage of static selection is that no overhead is imposed: there is no performance penalty for the abstraction layers between the three levels.

Driver interface

The driver should implement the following interface:

- `CreateTerm(vars,{exp,coef})` - create a multivariate polynomial with one term, in the variables defined in 'var', with the (key,value) pair (coefs,fact)
- `MultiNomialAdd(multi1, multi2)` - add two multivars, and (possibly) destructively modify multi1 to contain the result: [multi1 := multi1 + multi2; multi1;];
- `MultiNomialMultiplyAdd(multi1, multi2,exp,coef)` - add two multivars, and (possibly) destructively modify multi1 to contain the result. multi2 is considered multiplied by a term represented by the (key,value) pair (exp,coef). [multi1 := multi1 + term * multi2; multi1;];
- `MultiNomialNegate(multi)` - negate a multivar, returning -multi, and destructively changing the original. [multi := - multi; multi1;];
- `MultiNomialMultiply(multi1,multi2)` - Multiply two multivars, and (possibly) destructively modify multi1 to contain the result, returning the result: [multi1 := multi1 * multi2; multi1;];
- `NormalForm(multi)` - convert MultiNomial to normal form (as would be typed in be the user). This is part of the driver because the driver might be able to do this more efficiently than code above it which can use ScanMultiNomial.
- `MultiLeadingTerm(multi)` - return the (key,value) pair (mdeg(f),lc(f)) representing the leading term. This is all the information needed about the leading term, and thus the leading coefficient and multidegree can be extracted from it.
- `MultiDropLeadingZeroes(multi)` - remove leading terms with zero factors.
- `MultiTermLess(x,y)` - for two (key,value) pairs, return `True` if $x < y$, where the operation `<` is the one used for the representation, and `False` otherwise.
- `ScanMultiNomial(op,multi)` - traverse all the terms of the multivariate, applying the function 'op' to each (key,value) pair (exp,coef). The monomials are traversed in the ordering defined by MultiTermLess. 'op' should be a function accepting two arguments.
- `MultiZero(multi)` - return `True` if the multivariate is zero (all coefficients are zero), `False` otherwise.

1.3 Integration

Integration can be performed by the function `Integrate`, which has two calling conventions:

- Integrate(variable) expression
- Integrate(variable, from, to) expression

Integrate can have its own set of rules for specific integrals, which might return a correct answer immediately. Alternatively, it calls the function AntiDeriv, to see if the anti-derivative can be determined for the integral requested. If this is the case, the anti-derivative is used to compose the output.

If the integration algorithm cannot perform the integral, the expression is returned unsimplified.

The integration algorithm

This section describes the steps taken in doing integration.

General structure

The integration starts at the function `Integrate`, but the task is delegated to other functions, one after the other. Each function can deem the integral unsolvable, and thus return the integral unevaluated. These different functions offer hooks for adding new types of integrals to be handled.

Expression clean-up

Integration starts by first cleaning up the expression, by calling `TrigSimpCombine` to simplify expressions containing multiplications of trigonometric functions into additions of trigonometric functions (for which the integration rules are trivial), and then passing the result to `Simplify`.

Generalized integration rules

For the function `AntiDeriv`, which is responsible for finding the anti-derivative of a function, the code splits up expressions according to the additive properties of integration, eg. integration of $a + b$ is the same as integrating a + integrating b.

- Polynomials which can be expressed as univariate polynomials in the variable to be integrated over are handled by one integration rule.
- Expressions of the form $pf(x)$, where p represents a univariate polynomial, and $f(x)$ an integrable function, are handled by a special integration rule. This transformation rule has to be designed carefully not to invoke infinite recursion.
- Rational functions, $\frac{f(x)}{g(x)}$ with both $f(x)$ and $g(x)$ univariate polynomials, is handled separately also, using partial fraction expansion to reduce rational function to a sum of simpler expressions.

Integration tables

For elementary functions, Yacas uses integration tables. For instance, the fact that the anti-derivative of $\cos x$ is $\sin x$ is declared in an integration table.

For the purpose of setting up the integration table, a few declaration functions have been defined, which use some generalized pattern matchers to be more flexible in recognizing expressions that are integrable.

Integrating simple functions of a variable

For functions like $\sin x$ the anti-derivative can be declared with the function `IntFunc`.

The calling sequence for `IntFunc` is

```
IntFunc(variable,pattern,antiderivative)
```

For instance, for the function $\cos x$ there is a declaration:

```
IntFunc(x,Cos(_x),Sin(x));
```

The fact that the second argument is a pattern means that each occurrence of the variable to be matched should be referred to as `_x`, as in the example above.

IntFunc generalizes the integration implicitly, in that it will set up the system to actually recognize expressions of the form $\cos(ax + b)$, and return $\frac{\sin(ax+b)}{a}$ automatically. This means that the variables a and b are reserved, and can not be used in the pattern. Also, the variable used (in this case, `_x` is actually matched to the expression passed in to the function, and the variable `var` is the real variable being integrated over. To clarify: suppose the user wants to integrate $\cos(cy + d)$ over y, then the following variables are set:

- `a` $= c$
- `b` $= d$
- `x` $= ay + b$
- `var` $= x$

When functions are multiplied by constants, that situation is handled by the integration rule that can deal with univariate polynomials multiplied by functions, as a constant is a polynomial of degree zero.

Integrating functions containing expressions of the form $ax^2 + b$

There are numerous expressions containing sub-expressions of the form $ax^2 + b$ which can easily be integrated.

The general form for declaring anti-derivatives for such expressions is:

```
IntPureSquare(variable, pattern, sign2, sign0,
  antiderivative)
```

Here `IntPureSquare` uses `MatchPureSquared` to match the expression.

The expression is searched for the pattern, where the variable can match to a sub-expression of the form ax^2+b, and for which both a and b are numbers and $a$sign2 > 0 and $b$sign0 > 0.

As an example:

```
IntPureSquare(x,num_IsFreeOf(var)/(_x),
  1,1,(num/(a*Sqrt(b/a)))*
  ArcTan(var/Sqrt(b/a)));
```

declares that the anti-derivative of $\frac{c}{ax^2+b}$ is

$$\frac{c}{a\sqrt{\frac{b}{a}}} \arctan \frac{x}{\sqrt{\frac{b}{a}}},$$

if both a and b are positive numbers.

1.4 Transforms

Currently the only tranform defined is `LaplaceTransform`, which has the calling convention:

- LaplaceTransform(var1,var2,func)

It has been setup much like the integration algorithm. If the transformation algorithm cannot perform the transform, the expression (in theory) is returned unsimplified. Some cases may still erroneously return `Undefined` or `Infinity`.

The `LaplaceTransform` algorithm

This section describes the steps taken in doing a Laplace transform.

General structure

`LaplaceTransform` is immediately handed off to `LapTran`. This is done because if the last `LapTran` rule is met, the Laplace transform couldn't be found and it can then return `LaplaceTransform` unevaluated.

Operational Properties

The first rules that are matched against utilize the various operational properties of `LaplaceTransform`, such as:

- Linearity Properties
- Shift properties, i.e. multiplying the function by an exponential
- $yx^n = (-1)^n \left(\frac{\partial^n}{\partial x^n} \text{LaplaceTransform}\,(x, x_2, y)\right)$
- $\frac{y}{x} = \int_{x_2}^{\infty} \text{LapTran}\,(x, x_2, y)\,dx_2$

The last operational property dealing with integration is not yet fully bug-tested, it sometimes returns `Undefined` or `Infinity` if the integral returns such.

Transform tables

For elementary functions, Yacas uses transform tables. For instance, the fact that the Laplace transform of $\cos t$ is $\frac{s}{s^2+1}$ is declared in a transform table.

For the purpose of setting up the transform table, a few declaration functions have been defined, which use some generalized pattern matchers to be more flexible in recognizing expressions that are transformable.

Transforming simple functions

For functions like $\sin t$ the transform can be declared with the function `LapTranDef`.

The calling sequence for `LapTranDef` is

```
LapTranDef( in, out )
```

Currently `in` must be a variable of `_t` and `out` must be a function of `s`. For instance, for the function $\cos t$ there is a declaration:

```
LapTranDef( Cos(_t),                    s/(s^2+1) );
```

The fact that the first argument is a pattern means that each occurrence of the variable to be matched should be referred to as `_t`, as in the example above.

`LapTranDef` generalizes the transform implicitly, in that it will set up the system to actually recognize expressions of the form $\cos at$ and $\cos \frac{t}{a}$, and return the appropriate answer. The way this is done is by three separate rules for case of `t` itself, `a*t` and `t/a`. This is similar to the `MatchLinear` function that `Integrate` uses, except `LaplaceTransforms` must have `b=0`.

Further Directions

Currenlty $\sin t \cos t$ cannot be transformed, because it requires a convolution integral. This will be implemented soon. The inverse laplace transform will be implement soon also.

1.5 Finding real roots of polynomials

This section deals with finding roots of polynomials in the field of real numbers.

Without loss of generality, the coefficients a_i of a polynomial

$$p = a_n x^n + ... + a_0$$

can be considered to be rational numbers, as real-valued numbers are truncated in practice, when doing calculations on a computer.

Assuming that the leading coefficient $a_n = 1$, the polynomial p can also be written as

$$p = p_1^{n_1} ... p_m^{n_m},$$

where p_i are the m distinct irreducible monic factors of the form $p_i = x - x_i$, and n_i are multiplicities of the factors. Here the roots are x_i and some of them may be complex. However, complex roots of a polynomial with real coefficients always come in conjugate pairs, so the corresponding irreducible factors should be taken as $p_i = x^2 + c_i x + d_i$. In this case, there will be less than m irreducible factors, and all coefficients will be real.

To find roots, it is useful to first remove the multiplicities, i.e. to convert the polynomial to one with multiplicity 1 for all irreducible factors, i.e. find the polynomial $p_1 ... p_m$. This is called the "square-free part" of the original polynomial p.

The square-free part of the polynomial p can be easily found using the polynomial GCD algorithm. The derivative of a polynomial p can be written as:

$$p' = \sum_{i=1}^{m} p_1^{n_1} ... n_i p_i^{n_i - 1} \left(\frac{\partial}{\partial x} p_i\right) ... p_m^{n_m}.$$

Not simplified

The g.c.d. of p and p' equals

$$\text{Gcd}\left(p, p'\right) = \prod_{i=1}^{m} p_i^{n_i - 1}.$$

So if we divide p by $\text{Gcd}\,(p, p')$, we get the square-free part of the polynomial:

$$\text{SquareFree}\,(p) \equiv \text{Div}\left(p, \text{Gcd}\left(p, p'\right)\right) = p_1 ... p_m.$$

In what follows we shall assume that all polynomials are square-free with rational coefficients. Given any polynomial, we can apply the functions `SquareFree` and `Rationalize` and reduce it to this form. The function `Rationalize` converts all numbers in an expression to rational numbers. The function `SquareFree` returns the square-free part of a polynomial. For example:

```
In> Expand((x+1.5)^5)
Out> x^5+7.5*x^4+22.5*x^3+33.75*x^2+25.3125*x
+7.59375;
In> SquareFree(Rationalize(%))
Out> x/5+3/10;
In> Simplify(%*5)
Out> (2*x+3)/2;
In> Expand(%)
Out> x+3/2;
```

Sturm sequences

For a polynomial $p\,(x)$ of degree n, the Sturm sequence p_0, p_1, ... p_n is defined by the following equations (following the book [Davenport *et al.* 1989]):

$$p_0 = p\,(x)\,,$$
$$p_1 = p'\,(x)\,,$$
$$p_i = -\text{remainder}\,(p_{i-2}, p_{i-1})\,,$$

where $\text{remainder}\,(p, q)$ is the remainder of division of polynomials p, q.

The polynomial p can be assumed to have no multiple factors, and thus p and p' are relatively prime. The sequence of polynomials in the Sturm sequence are (up to a minus sign) the

consecutive polynomials generated by Euclid's algorithm for the calculation of a greatest common divisor for p and p', so the last polynomial p_n will be a constant.

In Yacas, the function `SturmSequence(p)` returns the Sturm sequence of p, assuming p is a univariate polynomial in x, $p = p(x)$.

Given a Sturm sequence $S = \text{SturmSequence}(p)$ of a polynomial p, the *variation* in the Sturm sequence $V(S, y)$ is the number of sign changes in the sequence p_0, p_1 , ... , p_n, evaluated at point y, and disregarding zeroes in the sequence.

Sturm's theorem states that if a and b are two real numbers which are not roots of p, and $a < b$, then the number of roots between a and b is $V(S, a) - V(S, b)$. A proof can be found in Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*.

For a and b, the values $-\infty$ and ∞ can also be used. In these cases, $V(S, \infty)$ is the number of sign changes between the leading coefficients of the elements of the Sturm sequence, and $V(S, -\infty)$ the same, but with a minus sign for the leading coefficients for which the degree is odd.

Thus, the number of real roots of a polynomial is $V(S, -\infty) - V(S, \infty)$. The function `NumRealRoots(p)` returns the number of real roots of p.

The function `SturmVariations(S,y)` returns the number of sign changes between the elements in the Sturm sequence S, at point $x = y$:

```
In> p:=x^2-1
Out> x^2-1;
In> S:=SturmSequence(p)
Out> {x^2-1,2*x,1};
In> SturmVariations(S,-Infinity)- \
SturmVariations(S,Infinity)
Out> 2;
In> NumRealRoots(p)
Out> 2;
In> p:=x^2+1
Out> x^2+1;
In> S:=SturmSequence(p)
Out> {x^2+1,2*x,-1};
In> SturmVariations(S,-Infinity)- \
SturmVariations(S,Infinity)
Out> 0;
In> NumRealRoots(p)
Out> 0;
```

Finding bounds on real roots

Armed with the variations in the Sturm sequence given in the previous section, we can now find the number of real roots in a range (a,b), for $a < b$. We can thus bound all the roots by subdividing ranges until there is only one root in each range. To be able to start this process, we first need some upper bounds of the roots, or an interval that contains all roots. Davenport gives limits on the roots of a polynomial given the coefficients of the polynomial, as

$$|a| \leq \max\left(\left|\frac{a_{n-1}}{a_n}\right|, \sqrt{\left|\frac{a_{n-2}}{a_n}\right|}, ..., \sqrt[n]{\left|\frac{a_0}{a_n}\right|}\right),$$

for all real roots a of p. This gives the upper bound on the absolute value of the roots of the polynomial in question. if $p(0) \neq 0$, the minimum bound can be obtained also by considering the upper bound of $p\left(\frac{1}{x}\right)x^n$, and taking $\frac{1}{\text{bound}}$.

We thus know that given

$$a_{\max} = \text{MaximumBound}(p)$$

and

$$a_{\min} = \text{MinimumBound}(p)$$

for all roots a of polynomial, either

$$-a_{\max} \leq a \leq -a_{\min}$$

or

$$a_{\min} \leq a \leq a_{\max}.$$

Now we can start the search for the bounds on all roots. The search starts with initial upper and lower bounds on ranges, subdividing ranges until a range contains only one root, and adding that range to the resulting list of bounds. If, when dividing a range, the middle of the range lands on a root, care must be taken, because the bounds should not be on a root themselves. This can be solved by observing that if c is a root, p contains a factor $x - c$, and thus taking $p(x + c)$ results in a polynomial with all the roots shifted by a constant $-c$, and the root c moved to zero, e.g. $p(x + c)$ contains a factor x. Thus a new ranges to the left and right of c can be determined by first calculating the minimum bound M of $\frac{p(x+c)}{x}$. When the original range was (a,b), and $c = \frac{a+b}{2}$ is a root, the new ranges should become $(a,c - M)$ and $(c + M,b)$.

In Yacas, `MimimumBound(p)` returns the lower bound described above, and `MaximumBound(p)` returns the upper bound on the roots in p. These bounds are returned as rational numbers. `BoundRealRoots(p)` returns a list with sublists with the bounds on the roots of a polynomial:

```
In> p:=(x+20)*(x+10)
Out> (x+20)*(x+10);
In> MinimumBound(p)
Out> 10/3;
In> MaximumBound(p)
Out> 60;
In> BoundRealRoots(p)
Out> {{-95/3,-35/2},{-35/2,-10/3}};
In> N(%)
Out> {{-31.6666666666,-17.5},
  {-17.5,-3.3333333333}};
```

It should be noted that since all calculations are done with rational numbers, the algorithm for bounding the roots is very robust. This is important, as the roots can be very unstable for small variations in the coefficients of the polynomial in question (see Davenport).

Finding real roots given the bounds on the roots

Given the bounds on the real roots as determined in the previous section, two methods for finding roots are available: the secant method or the Newton method, where the function is locally approximated by a line, and extrapolated to find a new estimate for a root. This method converges quickly when "sufficiently" near a root, but can easily fail otherwise. The secant method can easily send the search to infinity.

The bisection method is more robust, but slower. It works by taking the middle of the range, and checking signs of the polynomial to select the half-range where the root is. As there is only one root in the range (a,b), in general it will be true that $p(a)\,p(b) < 0$, which is assumed by this method.

Yacas finds the roots by first trying the secant method, starting in the middle of the range, $c = \frac{a+b}{2}$. If this fails the bisection method is tried.

The function call to find the real roots of a polynomial p in variable x is `FindRealRoots(p)`, for example:

```
In> p:=Expand((x+3.1)*(x-6.23))
Out> x^2-3.13*x-19.313;
In> FindRealRoots(p)
Out> {-3.1,6.23};
In> p:=Expand((x+3.1)^3*(x-6.23))
Out> x^4+3.07*x^3-29.109*x^2-149.8199\
In> *x-185.59793;
In> p:=SquareFree(Rationalize( \
In> Expand((x+3.1)^3*(x-6.23))))
Out> (-160000*x^2+500800*x+3090080)/2611467;
In> FindRealRoots(p)
Out> {-3.1,6.23};
```

Chapter 2

Number theory algorithms

This chapter describes the algorithms used for computing various number-theoretic functions. We call "number-theoretic" any function that takes integer arguments, produces integer values, and is of interest to number theory.

2.1 Euclidean GCD algorithms

The main algorithm for the calculation of the GCD of two integers is the binary Euclidean algorithm. It is based on the following identities: $\mathrm{Gcd}\,(a,b) = \mathrm{Gcd}\,(b,a)$, $\mathrm{Gcd}\,(a,b) = \mathrm{Gcd}\,(a-b,b)$, and for odd b, $\mathrm{Gcd}\,(2a,b) = \mathrm{Gcd}\,(a,b)$. Thus we can produce a sequence of pairs with the same GCD as the original two numbers, and each pair will be at most half the size of the previous pair. The number of steps is logarithmic in the number of digits in a, b. The only operations needed for this algorithm are binary shifts and subtractions (no modular division is necessary). The low-level function for this is `MathGcd`.

To speed up the calculation when one of the numbers is much larger than another, one could use the property $\mathrm{Gcd}\,(a,b) = \mathrm{Gcd}\,(a, a \bmod b)$. This will introduce an additional modular division into the algorithm; this is a slow operation when the numbers are large.

2.2 Prime numbers: the Miller-Rabin test and its improvements

Small prime numbers $p <= 65537$ are simply stored in a precomputed table as an array of bits; the bits corresponding to prime numbers are set to 1. This makes primality testing on small numbers very quick. This is implemented by the function `FastIsPrime`.

Primality of larger numbers is tested by the function `IsPrime` that uses the Miller-Rabin algorithm. [1] This algorithm is deterministic (guaranteed correct within a certain running time) for "small" numbers $n < 3.4 \cdot 10^{13}$ and probabilistic (correct with high probability but not guaranteed) for larger numbers. In other words, the Miller-Rabin test could sometimes flag a large number n as prime when in fact n is composite; but the probability for this to happen can be made extremely small. The basic reference is [Rabin 1980]. We also implemented some of the improvements suggested in [Davenport 1992].

The idea of the Miller-Rabin algorithm is to improve the Fermat primality test. If n is prime, then for any x we have $\mathrm{Gcd}\,(n,x) = 1$. Then by Fermat's "little theorem", $x^{n-1} \equiv 1 \bmod n$. (This is really a simple statement; if n is prime, then $n-1$ nonzero remainders modulo n: 1, 2, ..., $n-1$ form a cyclic multiplicative group.) Therefore we pick some "base" integer x and compute $x^{n-1} \bmod n$; this is a quick computation even if n is large. If this value is not equal to 1 for some base x, then n is definitely not prime. However, we cannot test *every* base $x < n$; instead we test only some x, so it may happen that we miss the right values of x that would expose the non-primality of n. So Fermat's test sometimes fails, i.e. says that n is a prime when n is in fact not a prime. Also there are infinitely many integers called "Carmichael numbers" which are not prime but pass the Fermat test for every base.

The Miller-Rabin algorithm improves on this by using the property that for prime n there are no nontrivial square roots of unity in the ring of integers modulo n (this is Lagrange's theorem). In other words, if $x^2 \equiv 1 \bmod n$ for some x, then x must be equal to 1 or -1 modulo n. (Since $n-1$ is equal to -1 modulo n, we have $n-1$ as a trivial square root of unity modulo n. Note that even if n is prime there may be nontrivial divisors of 1, for example, $2 \cdot 49 \equiv 1 \bmod 97$.)

We can check that n is odd before applying any primality test. (A test $n^2 \equiv 1 \bmod 24$ guarantees that n is not divisible by 2 or 3. For large n it is faster to first compute $n \bmod 24$ rather than n^2, or test n directly.) Then we note that in Fermat's test the number $n-1$ is certainly a composite number because $n-1$ is even. So if we first find the largest power of 2 in $n-1$ and decompose $n-1 = 2^r q$ with q odd, then $x^{n-1} \equiv a^{2^r} \bmod n$ where $a \equiv x^q \bmod n$. (Here $r \geq 1$ since n is odd.) In other words, the number $x^{n-1} \bmod n$ is obtained by repeated squaring of the number a. We get a sequence of r repeated squares: a, a^2, ..., a^{2^r}. The last element of this sequence must be 1 if n passes the Fermat test. (If it does not pass, n is definitely a composite number.) If n passes the Fermat test, the last-but-one element $a^{2^{r-1}}$ of the sequence of squares is a square root of unity modulo n. We can check whether this square root is non-trivial (i.e. not equal to 1 or -1 modulo n). If it is non-trivial, then n definitely cannot be a prime. If it is trivial and equal to 1, we can check the preceding element, and so on. If an element is equal to -1, we cannot say anything, i.e. the test passes (n is "probably a prime").

This procedure can be summarized like this:

1. Find the largest power of 2 in $n-1$ and an odd number q such that $n-1 = 2^r q$.
2. Select the "base number" $x < n$. Compute the sequence $a \equiv x^q \bmod n$, a^2, a^4, ..., a^{2^r} by repeated squaring modulo n. This sequence contains at least two elements since $r \geq 1$.
3. If $a = 1$ or $a = n-1$, the test passes on the base number x. Otherwise, the test passes if at least one of the elements of the sequence is equal to $n-1$ and fails if none of them are equal to $n-1$. This simplified procedure works because the first element that is equal to 1 *must* be preceded by a

[1] Initial implementation and documentation was supplied by Christian Obrecht.

-1, or else we would find a nontrivial root of unity.

Here is a more formal definition. An odd integer n is called *strongly-probably-prime* for base b if $b^q \equiv 1 \bmod n$ or $b^{q \cdot 2^i} \equiv (n-1) \bmod n$ for some i such that $0 \leq i < r$, where q and r are such that q is odd and $n - 1 = q \cdot 2^r$.

A practical application of this procedure needs to select particular base numbers. It is advantageous (according to [Pomerance *et al.* 1980]) to choose *prime* numbers b as bases, because for a composite base $b = pq$, if n is a strong pseudoprime for both p and q, then it is very probable that n is a strong pseudoprime also for b, so composite bases rarely give new information.

An additional check suggested by [Davenport 1992] is activated if $r > 2$ (i.e. if $n \equiv 1 \bmod 8$ which is true for only $1/4$ of all odd numbers). If $i \geq 1$ is found such that $b^{q \cdot 2^i} \equiv (n-1) \bmod n$, then $b^{q \cdot 2^{i-1}}$ is a square root of -1 modulo n. If n is prime, there may be only two different square roots of -1. Therefore we should store the set of found values of roots of -1; if there are more than two such roots, then we woill find some roots s_1, s_2 of -1 such that $s_1 + s_2 \neq 0 \bmod n$. But $s_1^2 - s_2^2 \equiv 0 \bmod n$. Therefore n is definitely composite, e.g. $\mathrm{Gcd}(s_1 + s_2, n) > 1$. This check costs very little computational effort but guards against some strong pseudoprimes.

Yet another small improvement comes from [Damgard *et al.* 1993]. They found that the strong primality test sometimes (rarely) passes on composite numbers n for more than $\frac{1}{8}$ of all bases $x < n$ if n is such that either $3n + 1$ or $8n + 1$ is a perfect square, or if n is a Carmichael number. Checking Carmichael numbers is slow, but it is easy to show that if n is a large enough prime number, then neither $3n + 1$, nor $8n + 1$, nor any $sn + 1$ with small integer s can be a perfect square. [If $sn + 1 = r^2$, then $sn = (r-1)(r+1)$.] Testing for a perfect square is quick and does not slow down the algorithm. This is however not implemented in Yacas because it seems that perfect squares are too rare for this improvement to be significant.

If an integer is not "strongly-probably-prime" for a given base b, then it is a composite number. However, the converse statement is false, i.e. "strongly-probably-prime" numbers can actually be composite. Composite strongly-probably-prime numbers for base b are called *strong pseudoprimes* for base b. There is a theorem that if n is composite, then among all numbers b such that $1 < b < n$, at most one fourth are such that n is a strong pseudoprime for base b. Therefore if n is strongly-probably-prime for many bases, then the probability for n to be composite is very small.

For numbers less than $B = 34155071728321$, exhaustive [2] computations have shown that there are no strong pseudoprimes simultaneously for bases 2, 3, 5, 7, 11, 13 and 17. This gives a simple and reliable primality test for integers below B. If $n \geq B$, the Rabin-Miller method consists in checking if n is strongly-probably-prime for k base numbers b. The base numbers are chosen to be consecutive "weak pseudoprimes" that are easy to generate (see below the function `NextPseudoPrime`).

In the implemented routine `RabinMiller`, the number of bases k is chosen to make the probability of erroneously passing the test $p < 10^{-25}$. (Note that this is *not* the same as the probability to give an incorrect answer, because all numbers that do not pass the test are definitely composite.) The probability for the test to pass mistakenly on a given number is found as follows. Suppose the number of bases k is fixed. Then the probability for a given composite number to pass the test is less than $p_f = 4^{-k}$. The probability for a given number n to be prime is roughly $p_p = \frac{1}{\ln n}$ and to be composite $p_c = 1 - \frac{1}{\ln n}$. Prime numbers never fail the test. Therefore, the probability for the test to pass is $p_f p_c + p_p$ and the probability to pass erroneously is

$$p = \frac{p_f p_c}{p_f p_c + p_p} < \ln n \cdot 4^{-k}.$$

To make $p < \epsilon$, it is enough to select $k = \frac{1}{\ln 4}(\ln n - \ln \epsilon)$.

Before calling `MillerRabin`, the function `IsPrime` performs two quick checks: first, for $n \geq 4$ it checks that n is not divisible by 2 or 3 (all primes larger than 4 must satisfy this); second, for $n > 257$, it checks that n does not contain small prime factors $p \leq 257$. This is checked by evaluating the GCD of n with the precomputed product of all primes up to 257. The computation of the GCD is quick and saves time in case a small prime factor is present.

There is also a function `NextPrime(n)` that returns the smallest prime number larger than `n`. This function uses a sequence 5,7,11,13,... generated by the function `NextPseudoPrime`. This sequence contains numbers not divisible by 2 or 3 (but perhaps divisible by 5,7,...). The function `NextPseudoPrime` is very fast because it does not perform a full primality test.

The function `NextPrime` however does check each of these pseudoprimes using `IsPrime` and finds the first prime number.

2.3 Factorization of integers

When we find from the primality test that an integer n is composite, we usually do not obtain any factors of n. Factorization is implemented by functions `Factor` and `Factors`. Both functions use the same algorithms to find all prime factors of a given integer n. (Before doing this, the primality checking algorithm is used to detect whether n is a prime number.) Factorization consists of repeatedly finding a factor, i.e. an integer f such that $n \bmod f = 0$, and dividing n by f. (Of course, each fastor f needs to be factorized too.)

First we determine whether the number n contains "small" prime factors $p \leq 257$. A quick test is to find the GCD of n and the product of all primes up to 257: if the GCD is greater than 1, then n has at least one small prime factor. (The product of primes is precomputed.) If this is the case, the trial division algorithm is used: n is divided by all prime numbers $p \leq 257$ until a factor is found. `NextPseudoPrime` is used to generate the sequence of candidate divisors p.

After separating small prime factors, we test whether the number n is an integer power of a prime number, i.e. whether $n = p^s$ for some prime number p and an integer $s \geq 1$. This is tested by the following algorithm. We already know that n is not prime and that n does not contain any small prime factors up to 257. Therefore if $n = p^s$, then $p > 257$ and $2 \leq s < s_0 = \frac{\ln n}{\ln 257}$. In other words, we only need to look for powers not greater than s_0. This number can be approximated by the "integer logarithm" of n in base 257 (routine `IntLog(n, 257)`).

Now we need to check whether n is of the form p^s for $s = 2$, 3, ..., s_0. Note that if for example $n = p^{24}$ for some p, then the square root of n will already be an integer, $\sqrt{n} = p^{12}$. Therefore it is enough to test whether $\sqrt[s]{n}$ is an integer for all *prime* values of s up to s_0, and then we will definitely discover whether n is a power of some other integer. The testing is performed using the integer n-th root function `IntNthRoot` which quickly computes the integer part of n-th root of an integer number. If we discover that n has an integer root p of order s, we have to check that p itself is a prime power (we use the same algorithm recursively). The number n is a prime power if and only if p is itself a prime power. If we find no integer roots of orders $s \leq s_0$, then n is not a prime power.

If the number n is not a prime power, the Pollard "rho" algorithm is applied [Pollard 1978]. The Pollard "rho" algorithm

[2] And surely exhausting.

takes an irreducible polynomial, e.g. $p(x) = x^2 + 1$ and builds a sequence of integers $x_{k+1} \equiv p(x_k) \bmod n$, starting from $x_0 = 2$. For each k, the value $x_{2k} - x_k$ is attempted as possibly containing a common factor with n. The GCD of $x_{2k} - x_k$ with n is computed, and if $\mathrm{Gcd}(x_{2k} - x_k, n) > 1$, then that GCD value divides n.

The idea behind the "rho" algorithm is to generate an effectively random sequence of trial numbers t_k that may have a common factor with n. The efficiency of this algorithm is determined by the size of the smallest factor p of n. Suppose p is the smallest prime factor of n and suppose we generate a random sequence of integers t_k such that $1 \leq t_k < n$. It is clear that, on the average, a fraction $\frac{1}{p}$ of these integers will be divisible by p. Therefore (if t_k are truly random) we should need on the average p tries until we find t_k which is accidentally divisible by p. In practice, of course, we do not use a truly random sequence and the number of tries before we find a factor p may be significantly different from p. The quadratic polynomial seems to help reduce the number of tries in most cases.

But the Pollard "rho" algorithm may actually enter an infinite loop when the sequence x_k repeats itself without giving any factors of n. For example, the unmodified "rho" algorithm starting from $x_0 = 2$ loops on the number 703. The loop is detected by comparing x_{2k} and x_k. When these two quantities become equal to each other for the first time, the loop may not yet have occurred so the value of GCD is set to 1 and the sequence is continued. But when the equality of x_{2k} and x_k occurs many times, it indicates that the algorithm has entered a loop. A solution is to randomly choose a different starting number x_0 when a loop occurs and try factoring again, and keep trying new random starting numbers between 1 and n until a non-looping sequence is found. The current implementation stops after 100 restart attempts and prints an error message, "failed to factorize number".

A better (and faster) integer factoring algorithm needs to be implemented in Yacas.

Modern factoring algorithms are all probabilistic (i.e. they do not guarantee a particular finishing time) and fall into three categories:

1. Methods that work well (i.e. quickly) if there is a relatively small factor p of n (even if n itself is large). Pollard's "rho" algorithm belongs to this category. The fastest in this category is Lenstra's elliptic curves method (ECM).
2. Methods that work equally quickly regardless of the size of factors (but slower with larger n). These are the continued fractions method and the various "sieve" methods. The current best is the "General Number Field Sieve" (GNFS) but it is quite a complicated algorithm requiring operations with high-order algebraic numbers. The next best one is the "Multiple Polynomial Quadratic Sieve" (MPQS).
3. Methods that are suitable only for numbers of special "interesting" form, e.g. Fermat numbers $2^{2^k} - 1$ or generally numbers of the form $r^s + a$ where s is large but r and a are very small integers. The best method seems to be the "Special Number Field Sieve" which is a faster variant of the GNFS adapted to the problem.

There is ample literature describing these algorithms.

2.4 The Jacobi symbol

A number m is a "quadratic residue modulo n" if there exists a number k such that $k^2 \equiv m \bmod n$.

The Legendre symbol (m/n) is defined as $+1$ if m is a quadratic residue modulo n and -1 if it is a non-residue. The Legendre symbol is equal to 0 if $\frac{m}{n}$ is an integer.

The Jacobi symbol $\left(\frac{m}{n}\right)$ is defined as the product of the Legendre symbols of the prime factors f_i of $n = f_1^{p_1} ... f_s^{p_s}$,

$$\left(\frac{m}{n}\right) \equiv \left(\frac{m}{f_1}\right)^{p_1} ... \left(\frac{m}{f_s}\right)^{p_s}.$$

(Here we used the same notation $\left(\frac{a}{b}\right)$ for the Legendre and the Jacobi symbols; this is confusing but seems to be the current practice.) The Jacobi symbol is equal to 0 if m, n are not mutually prime (have a common factor). The Jacobi symbol and the Legendre symbol have values $+1$, -1 or 0.

The Jacobi symbol can be efficiently computed without knowing the full factorization of the number n. The currently used method is based on the following four identities for the Jacobi symbol:

1. $\left(\frac{a}{1}\right) = 1$.
2. $\left(\frac{2}{b}\right) = (-1)^{\frac{b^2-1}{8}}$.
3. $\left(\frac{ab}{c}\right) = \left(\frac{a}{c}\right)\left(\frac{b}{c}\right)$.
4. If $a \equiv b \bmod c$, then $\left(\frac{a}{c}\right) = \left(\frac{b}{c}\right)$.
5. If a, b are both odd, then $\left(\frac{a}{b}\right) = \left(\frac{b}{a}\right)(-1)^{(a-1)\frac{b-1}{4}}$.

Using these identities, we can recursively reduce the computation of the Jacobi symbol $\left(\frac{a}{b}\right)$ to the computation of the Jacobi symbol for numbers that are on the average half as large. This is similar to the fast "binary" Euclidean algorithm for the computation of the GCD. The number of levels of recursion is logarithmic in the arguments a, b.

More formally, Jacobi symbol $\left(\frac{a}{b}\right)$ is computed by the following algorithm. (The number b must be an odd positive integer, otherwise the result is undefined.)

1. If $b = 1$, return 1 and stop. If $a = 0$, return 0 and stop. Otherwise, replace $\left(\frac{a}{b}\right)$ by $\left(\frac{a \bmod b}{b}\right)$ (identity 4).
2. Find the largest power of 2 that divides a. Say, $a = 2^s c$ where c is odd. Replace $\left(\frac{a}{b}\right)$ by $\left(\frac{c}{b}\right)(-1)^{s\frac{b^2-1}{8}}$ (identities 2 and 3).
3. Now that $c < b$, replace $\left(\frac{c}{b}\right)$ by $\left(\frac{b}{c}\right)(-1)^{(b-1)\frac{c-1}{4}}$ (identity 5).
4. Continue to step 1.

Note that the arguments a, b may be very large integers and we should avoid performing multiplications of these numbers. We can compute $(-1)^{(b-1)\frac{c-1}{4}}$ without multiplications. This expression is equal to 1 if either b or c is equal to 1 mod 4; it is equal to -1 only if both b and c are equal to 3 mod 4. Also, $(-1)^{\frac{b^2-1}{8}}$ is equal to 1 if either $b \equiv 1$ or $b \equiv 7 \bmod 8$, and it is equal to -1 if $b \equiv 3$ or $b \equiv 5 \bmod 8$. Of course, if s is even, none of this needs to be computed.

2.5 Integer partitions

A partition of an integer n is a way of writing n as the sum of positive integers, where the order of these integers is unimportant. For example, there are 3 ways to write the number 3 in this way: $3 = 1 + 1 + 1$, $3 = 1 + 2$, $3 = 3$. The function `PartitionsP` counts the number of such partitions.

Large n

The first algorithm used to compute this function uses the Rademacher-Hardy-Ramanujan (RHR) theorem and is efficient for large n. (See for example [Ahlgren *et al.* 2001].) The number of partitions $P(n)$ is equal to an infinite sum:

$$P(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} \sqrt{k} A(k,n) S(k,n),$$

where the functions A and S are defined as follows:

$$S(k,n) \equiv \frac{\partial}{\partial n} \frac{\sinh \frac{\pi}{k}\sqrt{\frac{2}{3}\left(n-\frac{1}{24}\right)}}{\sqrt{n-\frac{1}{24}}}$$

Not simplified

$$A(k,n) \equiv \sum_{l=1}^{k} \delta(\mathrm{Gcd}(l,k),1) \exp\left(-2\pi\imath\frac{ln}{k} + \pi\imath B(k,l)\right),$$

where $\delta(x,y)$ is the Kronecker delta function (so that the summation goes only over integers l which are mutually prime with k) and B is defined by

$$B(k,l) \equiv \sum_{j=1}^{k-1} \frac{j}{k}\left(l\frac{j}{k} - \left\lfloor l\frac{j}{k}\right\rfloor - \frac{1}{2}\right).$$

The first term of the series gives, at large n, the Hardy-Ramanujan asymptotic estimate,

$$P(n) \sim P_0(n) \equiv \frac{1}{4n\sqrt{3}} \exp\left(\pi\sqrt{\frac{2n}{3}}\right).$$

The absolute value of each term decays quickly, so after $O\left(\sqrt{n}\right)$ terms the series gives an answer that is very close to the integer result.

There exist estimates of the error of this series, but they are complicated. The series is sufficiently well-behaved and it is easier to determine the truncation point heuristically. Each term of the series is either 0 (when all terms in $A(k,n)$ happen to cancel) or has a magnitude which is not very much larger than the magnitude of the previous nonzero term. (But the series is not actually monotonic.) In the current implementation, the series is truncated when $\left|A(k,n)S(n)\sqrt{k}\right|$ becomes smaller than 0.1 for the first time; in any case, the maximum number of calculated terms is $5+\frac{\sqrt{n}}{2}$. One can show that asymptotically for large n, the required number of terms is less than $\frac{\mu}{\ln\mu}$, where $\mu \equiv \pi\sqrt{\frac{2n}{3}}$.

[Ahlgren *et al.* 2001] mention that there exist explicit constants B_1 and B_2 such that

$$\left|P(n) - \sum_{k=1}^{B_1\sqrt{n}} A(k,n)\right| < B_2 n^{-\frac{1}{4}}.$$

The floating-point precision necessary to obtain the integer result must be at least the number of digits in the first term $P_0(n)$, i.e.

$$\mathrm{Prec} > \frac{\pi\sqrt{\frac{2}{3}n} - \ln 4n\sqrt{3}}{\ln 10}.$$

However, Yacas currently uses the fixed-point precision model. Therefore, the current implementation divides the series by $P_0(n)$ and computes all terms to Prec digits.

The RHR algorithm requires $O\left(\left(\frac{n}{\ln n}\right)^{\frac{3}{2}}\right)$ operations, of which $O\left(\frac{n}{\ln n}\right)$ are long multiplications at precision Prec $\sim O\left(\sqrt{n}\right)$ digits. The computational cost is therefore $O\left(\frac{n}{\ln n}M\left(\sqrt{n}\right)\right)$.

Small n

The second, simpler algorithm involves a recurrence relation

$$P_n = \sum_{k=1}^{n} (-1)^{k+1}\left(P_{n-k\frac{3k-1}{2}} + P_{n-k\frac{3k+1}{2}}\right).$$

The sum can be written out as

$$P(n-1) + P(n-2) - P(n-5) - P(n-7) + ...,$$

where 1, 2, 5, 7, ... is the "generalized pentagonal sequence" generated by the pairs $k\frac{3k-1}{2}$, $k\frac{3k+1}{2}$ for $k = 1, 2, ...$ The recurrence starts from $P(0) = 1$, $P(1) = 1$. (This is implemented as `PartitionsP'recur`.)

The sum is actually not over all k up to n but is truncated when the pentagonal sequence grows above n. Therefore, it contains only $O\left(\sqrt{n}\right)$ terms. However, computing $P(n)$ using the recurrence relation requires computing and storing $P(k)$ for all $1 \le k \le n$. No long multiplications are necessary, but the number of long additions of numbers with Prec $\sim O\left(\sqrt{n}\right)$ digits is $O\left(n^{\frac{3}{2}}\right)$. Therefore the computational cost is $O\left(n^2\right)$. This is asymptotically slower than the RHR algorithm even if a slow $O\left(n^2\right)$ multiplication is used. With internal Yacas math, the recurrence relation is faster for $n < 300$ or so, and for larger n the RHR algorithm is faster.

2.6 Miscellaneous functions

The function `Divisors` currently returns the number of divisors of integer, while `DivisorsSum` returns the sum of these divisors. (The current algorithms need to factor the number.) The following theorem is used:

Let $p_1^{k_1}...p_r^{k_r}$ be the prime factorization of n, where r is the number of prime factors and k_r is the multiplicity of the r-th factor. Then

$$\mathrm{Divisors}(n) = (k_1+1)...(k_r+1),$$

$$\mathrm{DivisorsSum}(n) = \frac{p_1^{k_1+1}-1}{p_1-1}...\frac{p_r^{k_r+1}-1}{p_r-1}.$$

The functions `ProperDivisors` and `ProperDivisorsSum` are functions that do the same as the above functions, except they do not consider the number n as a divisor for itself. These functions are defined by:

ProperDivisors (n) = Divisors $(n) - 1$,

ProperDivisorsSum (n) = DivisorsSum $(n) - n$.

Another number-theoretic function is `Moebius`, defined as follows: Moebius $(n) = (-1)^r$ if no factors of n are repeated, Moebius $(n) = 0$ if some factors are repeated, and Moebius $(n) = 1$ if $n = 1$. This again requires to factor the number n completely and investigate the properties of its prime factors. From the definition, it can be seen that if n is prime, then Moebius $(n) = -1$. The predicate `IsSquareFree(n)` then reduces to Moebius $(n) \neq 0$, which means that no factors of n are repeated.

2.7 Gaussian integers

A "Gaussian integer" is a complex number of the form $z = a+bi$, where a and b are ordinary (rational) integers. [3] The ring of

[3] To distinguish ordinary integers from Gaussian integers, the ordinary integers (with no imaginary part) are called "rational integers".

Gaussian integers is usually denoted by $Z[\imath]$ in the mathematical literature. It is an example of a ring of algebraic integers.

The function `GaussianNorm` computes the norm $N(z) = a^2 + b^2$ of z. The norm plays a fundamental role in the arithmetic of Gaussian integers, since it has the multiplicative property:

$$N(z.w) = N(z).N(w).$$

A unit of a ring is an element that divides any other element of the ring. There are four units in the Gaussian integers: 1, -1, $\imath$, $-\imath$. They are exactly the Gaussian integers whose norm is 1. The predicate `IsGaussianUnit` tests for a Gaussian unit.

Two Gaussian integers z and w are "associated" is $\frac{z}{w}$ is a unit. For example, $2 + \imath$ and $-1 + 2\imath$ are associated.

A Gaussian integer is called prime if it is only divisible by the units and by its associates. It can be shown that the primes in the ring of Gaussian integers are:

1. $1 + i$ and its associates.
2. The rational (ordinary) primes of the form $4n + 3$.
3. The factors $a+b\imath$ of rational primes p of the form $p = 4n+1$, whose norm is $p = a^2 + b^2$.

For example, 7 is prime as a Gaussian integer, while 5 is not, since $5 = (2 + \imath)(2 - \imath)$. Here $2 + \imath$ is a Gaussian prime.

The ring of Gaussian integers is an example of an Euclidean ring, i.e. a ring where there is a division algorithm. This makes it possible to compute the greatest common divisor using Euclid's algorithm. This is what the function `GaussianGcd` computes.

As a consequence, one can prove a version of the fundamental theorem of arithmetic for this ring: The expression of a Gaussian integer as a product of primes is unique, apart from the order of primes, the presence of units, and the ambiguities between associated primes.

The function `GaussianFactors` finds this expression of a Gaussian integer z as the product of Gaussian primes, and returns the result as a list of pairs `{p,e}`, where p is a Gaussian prime and e is the corresponding exponent. To do that, an auxiliary function called `GaussianFactorPrime` is used. This function finds a factor of a rational prime of the form $4n + 1$. We compute $a \equiv (2n)!$ (mod p). By Wilson's theorem a^2 is congruent to -1 (mod p), and it follows that p divides $(a + \imath)(a - \imath) = a^2 + 1$ in the Gaussian integers. The desired factor is then the `GaussianGcd` of $a + \imath$ and p. If the result is $a + b\imath$, then $p = a^2 + b^2$.

If z is a rational (i.e. real) integer, we factor z in the Gaussian integers by first factoring it in the rational integers, and after that by factoring each of the integer prime factors in the Gaussian integers.

If z is not a rational integer, we find its possible Gaussian prime factors by first factoring its norm $N(z)$ and then computing the exponent of each of the factors of $N(z)$ in the decomposition of z.

References for Gaussian integers

1. G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers.* Oxford University Press (1945).
2. H. Pollard, *The theory of Algebraic Numbers.* Wiley, New York (1965).

Chapter 3

A simple factorization algorithm for univariate polynomials

This section discusses factoring polynomials using arithmetic modulo prime numbers. Information was used from D. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms* and J.H. Davenport et. al., *Computer Algebra, SYSTEMS AND ALGORITHMS FOR ALGEBRAIC COMPUTATION.*

A simple factorization algorithm is developed for univariate polynomials. This algorithm is implemented as the function `BinaryFactors`. The algorithm was named the binary factoring algorithm since it determines factors to a polynomial modulo 2^n for successive values of n, effectively adding one binary digit to the solution in each iteration. No reference to this algorithm has been found so far in literature.

Berlekamp showed that polynomials can be efficiently factored when arithmetic is done modulo a prime. The Berlekamp algorithm is only efficient for small primes, but after that Hensel lifting can be used to determine the factors modulo larger numbers.

The algorithm presented here is similar in approach to applying the Berlekamp algorithm to factor modulo a small prime, and then factoring modulo powers of this prime (using the solutions found modulo the small prime by the Berlekamp algorithm) by applying Hensel lifting. However it is simpler in set up. It factors modulo 2, by trying all possible factors modulo 2 (two possibilities, if the polynomial is monic). This performs the same action usually left to the Berlekamp step. After that, given a solution modulo 2^n, it will test for a solution f_i modulo 2^n if f_i or $f_i + 2^n$ are a solution modulo 2^{n+1}.

This scheme raises the precision of the solution with one digit in binary representation. This is similar to the linear Hensel lifting algorithm, which factors modulo p^n for some prime p, where n increases by one after each iteration. There is also a quadratic version of Hensel lifting which factors modulo p^{2^n}, in effect doubling the number of digits (in p-adic expansion) of the solution after each iteration. However, according to "Davenport", the quadratic algorithm is not necessarily faster.

The algorithm here thus should be equivalent in complexity to Hensel lifting linear version. This has not been verified yet.

3.1 Modular arithmetic

This section copies some definitions and rules from *The Art of Computer Programming, Volume 1, Fundamental Algorithms* regarding arithmetic modulo an integer.

Arithmetic modulo an integer p requires performing the arithmetic operation and afterwards determining that integer modulo p. A number x can be written as

$$x = qp + r$$

where q is called the quotient, and r remainder. There is some liberty in the range one chooses r to be in. If r is an integer in the range `0,1, ... ,(p-1)` then it is the *modulo*, $r = x \bmod p$.

When $x \bmod p = y \bmod p$, the notation $(x = y) \bmod p$ is used. All arithmetic calculations are done modulo an integer p in that case.

For calculations modulo some p the following rules hold:

- If $(a = b) \bmod p$ and $(x = y) \bmod p$, then $(ax = by) \bmod p$, $(a + x = b + y) \bmod p$, and $(a - x = b - y) \bmod p$. This means that for instance also $x^n \bmod p = (x \bmod p)^n \bmod p$
- Two numbers x and y are *relatively prime* if they don't share a common factor, that is, if their greatest common denominator is one, $\mathrm{Gcd}(x, y) = 1$.
- If $(ax = by) \bmod p$ and if $(a = b) \bmod p$, and if a and p are relatively prime, then $(x = y) \bmod p$. This is useful for dividing out common factors.
- $(a = b) \bmod p$ if and only if $(an = bn) \bmod (np)$ when $n \neq 0$. Also, if r and s are relatively prime, then $(a = b) \bmod (rs)$ only if $(a = b) \bmod r$ and $(a = b) \bmod s$. These rules are useful when the modulus is changed.

For polynomials $v_1(x)$ and $v_2(x)$ it further holds that

$$((v_1(x) + v_2(x))^p = v_1(x)^p + v_2(x)^p) \bmod p$$

This follows by writing out the expression, noting that the binomial coefficients that result are multiples of p, and thus their value modulo p is zero (p divides these coefficients), so only the two terms on the right hand side remain.

Some corollaries

One corollary of the rules for calculations modulo an integer is *Fermat's theorem, 1640* : if p is a prime number then

$$(a^p = a) \bmod p$$

for all integers a (for a proof, see Knuth).

An interesting corollary to this is that, for some prime integer p:

$$(v(x)^p = v(x^p)) \bmod p.$$

This follows from writing it out and using Fermat's theorem to replace a^p with a where appropriate (the coefficients to the polynomial when written out, on the left hand side).

3.2 Factoring using modular arithmetic

The task is to factor a polynomial

$$p(x) = a_n x^n + ... + a_0$$

into a form

$$p(x) = Cg(x) f_1(x)^{p_1} f_2(x)^{p_2} ... f_m(x)^{p_m}$$

Where $f_i(x)$ are irreducible polynomials of the form:

$$f_i(x) = x + c_i$$

The part that could not be factorized is returned as $g(x)$, with a possible constant factor C.

The factors $f_i(x)$ and $g(x)$ are determined uniquely by requiring them to be monic. The constant C accounts for a common factor.

The c_i constants in the resulting solutions $f_i(x)$ can be rational numbers (or even complex numbers, if Gaussian integers are used).

3.3 Preparing the polynomial for factorization

The final factoring algorithm needs the input polynomial to be monic with integer coefficients (a polynomial is monic if its leading coefficient is one). Given a non-monic polynomial with rational coefficients, the following steps are performed:

Convert polynomial with rational coefficients to polynomial with integer coefficients

First the least common multiple lcm of the denominators of the coefficients $p(x)$ has to be found, and the polynomial is multiplied by this number. Afterwards, the C constant in the result should have a factor $\frac{1}{\text{lcm}}$.

The polynomial now only has integer coefficients.

Convert polynomial to a monic polynomial

The next step is to convert the polynomial to one where the leading coefficient is one. In order to do so, following "Davenport", the following steps have to be taken:

1. Multiply the polynomial by a_n^{n-1}
2. Perform the substitution $x = \frac{y}{a_n}$

The polynomial is now a monic polynomial in y.

After factoring, the irreducible factors of $p(x)$ can be obtained by multiplying C with $\frac{1}{a_n^{n-1}}$, and replacing y with $a_n x$. The irreducible solutions $a_n x + c_i$ can be replaced by $x + \frac{c_i}{a_i}$ after multiplying C by a_n, converting the factors to monic factors.

After the steps described here the polynomial is now monic with integer coefficients, and the factorization of this polynomial can be used to determine the factors of the original polynomial $p(x)$.

3.4 Definition of division of polynomials

To factor a polynomial a division operation for polynomials modulo some integer is needed. This algorithm needs to return a quotient $q(x)$ and remainder $r(x)$ such that:

$$(p(x) = q(r) d(x) + r(x)) \bmod p$$

for some polymomial $d(x)$ to be divided by, modulo some integer p. $d(x)$ is said to divide $p(x)$ (modulo p) if $r(x)$ is zero. It is then a factor modulo p.

For binary factoring algorithm it is important that if some monic $d(x)$ divides $p(x)$, then it also divides $p(x)$ modulo some integer p.

Define $\deg(f(x))$ to be the degree of $f(x)$ and $lc(f(x))$ to be the leading coefficient of $f(x)$. Then, if $\deg(p(x)) \geq \deg(d(x))$, one can compute an integer s such that

$$(lc(d(x)) s = lc(p(x))) \bmod p$$

If p is prime, then

$$s = \left(lc(p(x)) lc(d(x))^{p-2}\right) \bmod p$$

Because $\left(a^{p-1} = 1\right) \bmod p$ for any a. If p is not prime but $d(x)$ is monic (and thus $lc(d(x)) = 1$),

$$s = lc(p(x))$$

This identity can also be used when dividing in general (not modulo some integer), since the divisor is monic.

The quotient can then be updated by adding a term:
$\text{term} = sx^{\deg(p(x))-\deg(d(x))}$
and updating the polynomial to be divided, $p(x)$, by subtracting $d(x)$ term. The resulting polynomial to be divided now has a degree one smaller than the previous.

When the degree of $p(x)$ is less than the degree of $d(x)$ it is returned as the remainder.

A full division algorithm for arbitrary integer $p > 1$ with $lc(d(x)) = 1$ would thus look like:

```
divide(p(x),d(x),p)
   q(x) = 0
   r(x) = p(x)
   while (deg(r(x)) >= deg(d(x)))
      s = lc(r(x))
      term = s*x^(deg(r(x))-deg(d(x)))
      q(x) = q(x) + term
      r(x) = r(x) - term*d(x) mod p
   return {q(x),r(x)}
```

The reason we can get away with factoring modulo 2^n as opposed to factoring modulo some prime p in later sections is that the divisor $d(x)$ is monic. Its leading coefficient is one and thus $q(x)$ and $r(x)$ can be uniquely determined. If p is not prime and $lc(d(x))$ is not equal to one, there might be multiple combinations for which $p(x) = q(x) d(x) + r(x)$, and we are interested in the combinations where $r(x)$ is zero. This can be costly to determine unless `q(x),r(x)` is unique. This is the case here because we are factoring a monic polynomial, and are thus only interested in cases where $lc(d(x)) = 1$.

3.5 Determining possible factors modulo 2

We start with a polynomial $p(x)$ which is monic and has integer coefficients.

It will be factored into a form:

$$p(x) = g(x) f_1(x)^{p_1} f_2(x)^{p_2} ... f_m(x)^{p_m}$$

where all factors $f_i(x)$ are monic also.

The algorithm starts by setting up a test polynomial, $p_{\text{test}}(x)$ which divides $p(x)$, but has the property that

$$p_{\text{test}}(x) = g(x) f_1(x) f_2(x) ... f_m(x)$$

Such a polynomial is said to be *square-free*. It has the same factors as the original polynomial, but the original might have multiple of each factor, where $p_{\text{test}}(x)$ does not.

The square-free part of a polynomial can be obtained as follows:

$$p_{\text{test}}(x) = \frac{p(x)}{\text{Gcd}\left(p(x), \frac{d}{dx}p(x)\right)}$$

It can be seen by simply writing this out that $p(x)$ and $\frac{d}{dx}p(x)$ will have factors $f_i(x)^{p_i-1}$ in common. these can thus be divided out.

It is not a requirement of the algorithm that the algorithm being worked with is square-free, but it speeds up computations to work with the square-free part of the polynomial if the only thing sought after is the set of factors. The multiplicity of the factors can be determined using the original $p(x)$.

Binary factoring then proceeds by trying to find potential solutions modulo $p = 2$ first. There can only be two such solutions: $x + 0$ and $x + 1$.

A list of possible solutions L is set up with potential solutions.

3.6 Determining factors modulo 2^n given a factorization modulo 2

At this point there is a list L with solutions modulo 2^n for some n. The solutions will be of the form: $x + a$. The first step is to determine if any of the elements in L divides $p(x)$ (not modulo any integer). Since $x + a$ divides $p_{\text{test}}(x)$ modulo 2^n, both $x + a$ and $x + a - 2^n$ have to be checked.

If an element in L divides $p_{\text{test}}(x)$, $p_{\text{test}}(x)$ is divided by it, and a loop is entered to test how often it divides $p(x)$ to determine the multiplicity p_i of the factor. The found factor $f_i(x) = x + c_i$ is added as a combination $(x + c_i, p_i)$. $p(x)$ is divided by $f_i(x)^{p_i}$.

At this point there is a list L of factors that divide $p_{\text{test}}(x)$ modulo 2^n. This implies that for each of the elements u in L, either u or $u + 2^n$ should divide $p_{\text{test}}(x)$ modulo 2^{n+1}. The following step is thus to set up a new list with new elements that divide $p_{\text{test}}(x)$ modulo 2^{n+1}.

The loop is re-entered, this time doing the calculation modulo 2^{n+1} instead of modulo 2^n.

The loop is terminated if the number of factors found equals $\deg(p_{\text{test}}(x))$, or if 2^n is larger than the smallest non-zero coefficient of $p_{\text{test}}(x)$ as this smallest non-zero coefficient is the product of all the smallest non-zero coefficients of the factors, or if the list of potential factors is zero.

The polynomial $p(x)$ can not be factored any further, and is added as a factor $(p(x), 1)$.

The function `BinaryFactors`, when implemented, yields the following interaction in Yacas:

```
In> BinaryFactors((x+1)^4*(x-3)^2)
Out> {{x-3,2},{x+1,4}}
In> BinaryFactors((x-1/5)*(2*x+1/3))
Out> {{2,1},{x-1/5,1},{x+1/6,1}}
In> BinaryFactors((x-1123125)*(2*x+123233))
Out> {{2,1},{x-1123125,1},{x+123233/2,1}}
```

The binary factoring algorithm starts with a factorization modulo 2, and then each time tries to guess the next bit of the solution, maintaining a list of potential solutions. This list can grow exponentially in certain instances. For instance, factoring $(x - a)(x - 2a)(x - 3a)...$ implies a that the roots have common factors. There are inputs where the number of potential solutions (almost) doubles with each iteration. For these inputs the algorithm becomes exponential. The worst-case performance is therefore exponential. The list of potential solutions while iterating will contain a lot of false roots in that case.

3.7 Efficiently deciding if a polynomial divides another

Given the polynomial $p(x)$, and a potential divisor

$$f_i(x) = x - p$$

modulo some $q = 2^n$ an expression for the remainder after division is

$$\text{rem}(p) = \sum_{i=0}^{n} a_i p^i$$

For the initial solutions modulo 2, where the possible solutions are x and $x - 1$. For $p = 0$, $\text{rem}(0) = a_0$. For $p = 1$, $\text{rem}(1) = \sum_{i=0}^{n} a_i$.

Given a solution $x-p$ modulo $q = 2^n$, we consider the possible solutions $(x - p) \bmod 2^{n+1}$ and $(x - (p + 2^n)) \bmod (2^n + 1)$.

$x - p$ is a possible solution if $\text{rem}(p) \bmod 2^{n+1} = 0$.

$x - (p + q)$ is a possible solution if $\text{rem}(p + q) \bmod 2^{n+1} = 0$. Expanding $\text{rem}(p + q) \bmod (2q)$ yields:

$$\text{rem}(p + q) \bmod (2q) = (\text{rem}(p) + \text{extra}(p, q)) \bmod (2q)$$

When expanding this expression, some terms grouped under $\text{extra}(p, q)$ have factors like $2q$ or q^2. Since $q = 2^n$, these terms vanish if the calculation is done modulo 2^{n+1}.

The expression for $\text{extra}(p, q)$ then becomes

$$\text{extra}(p, q) = q \sum_{i=1}^{\frac{n}{2}} (2i - 1) a(2i) p^{2i-2}$$

An efficient approach to determining if $x - p$ or $x - (p + q)$ divides $p(x)$ modulo 2^{n+1} is then to first calculate $\text{rem}(p) \bmod (2q)$. If this is zero, $x - p$ divides $p(x)$. In addition, if $(\text{rem}(p) + \text{extra}(p, q)) \bmod (2q)$ is zero, $x-(p + q)$ is a potential candidate.

Other efficiencies are derived from the fact that the operations are done in binary. Eg. if $q = 2^n$, then $q_{\text{next}} = 2^{n+1} = 2q = q << 1$ is used in the next iteration. Also, calculations modulo 2^n are equivalent to performing a bitwise and with $2^n - 1$. These operations can in general be performed efficiently on todays hardware which is based on binary representations.

3.8 Extending the algorithm

Only univariate polynomials with rational coefficients have been considered so far. This could be extended to allow for roots that are complex numbers $a + ib$ where both a and b are rational numbers.

For this to work the division algorithm would have to be extended to handle complex numbers with integer a and b modulo some integer, and the initial setup of the potential solutions would have to be extended to try $x+1+\imath$ and $x+\imath$ also. The step where new potential solutions modulo 2^{n+1} are determined should then also test for $x+\imath\cdot 2^n$ and $x+2^n+\imath\cdot 2^n$.

The same extension could be made for multivariate polynomials, although setting up the initial irreducible polynomials that divide $p_{\text{test}}(x)$ modulo 2 might become expensive if done on a polynomial with many variables (2^{2^m-1} trials for m variables).

Lastly, polynomials with real-valued coefficients *could* be factored, if the coefficients were first converted to rational numbers. However, for real-valued coefficients there exist other methods (Sturm sequences).

3.9 Newton iteration

What the `BinaryFactor` algorithm effectively does is finding a set of potential solutions modulo 2^{n+1} when given a set of potential solutions modulo 2^n. There is a better algorithm that does something similar: Hensel lifting. Hensel lifting is a generalized form of Newton iteration, where given a factorization modulo p, each iteration returns a factorization modulo p^2.

Newton iteration is based on the following idea: when one takes a Taylor series expansion of a function:

$$f(x_0+dx) \equiv f(x_0) + \left(\frac{d}{dx}f(x_0)\right)dx + ...$$

Newton iteration then proceeds by taking only the first two terms in this series, the constant plus the constant times dx. Given some good initial value x_0, the function will is assumed to be close to a root, and the function is assumed to be almost linear, hence this approximation. Under these assumptions, if we want $f(x_0+dx)$ to be zero,

$$f(x_0+dx) = f(x_0) + \left(\frac{d}{dx}f(x_0)\right)dx = 0$$

This yields:

$$dx \equiv -\frac{f(x_0)}{\frac{d}{dx}f(x_0)} = 0$$

And thus a next, better, approximation for the root is $x_1 \equiv x_0 - \frac{f(x_0)}{\frac{d}{dx}f(x_0)}$, or more general:

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{\partial}{\partial x}f(x_n)}$$

If the root has multiplicity one, a Newton iteration can converge *quadratically*, meaning the number of decimals precision for each iteration doubles.

As an example, we can try to find a root of $\sin x$ near 3, which should converge to π.

Setting precision to 30 digits,

```
In> Builtin'Precision'Set(30)
Out> True;
```

We first set up a function $dx(x)$:

```
In> dx(x):=Eval(-Sin(x)/(D(x)Sin(x)))
Out> True;
```

And we start with a good initial approximation to π, namely 3. Note we should set `x` *after* we set dx(x), as the right hand side of the function definition is evaluated. We could also have used a different parameter name for the definition of the function $dx(x)$.

```
In> x:=3
Out> 3;
```

We can now start the iteration:

```
In> x:=N(x+dx(x))
Out> 3.14254654307427780529563541053 4;
In> x:=N(x+dx(x))
Out> 3.14159265330047681544988577172;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328 7;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328;
```

As shown, in this example the iteration converges quite quickly.

Finding roots of multiple equations in multiple variables using Newton iteration

One generalization, mentioned in W.H. Press et al., *NUMERICAL RECIPES in C, The Art of Scientific computing* is finding roots for multiple functions in multiple variables.

Given N functions in N variables, we want to solve

$$f_i(x_1,...,x_N) = 0$$

for $i = 1..N$. If de denote by X the vector

$$X := \texttt{x[1],x[2],...,x[N]}$$

and by dX the delta vector, then one can write

$$f_i(\mathrm{X}+dX) \equiv f_i(\mathrm{X}) + \sum_{j=1}^{N}\left(\frac{\partial}{\partial x_j}f_i(\mathrm{X})\right)dx_j$$

Setting $f_i(\mathrm{X}+dX)$ to zero, one obtains

$$\sum_{j=1}^{N} a_{(i,j)}dx_j = b_i$$

where

$$a_{(i,j)} \equiv \frac{\partial}{\partial x_j}f_i(\mathrm{X})$$

and

$$b_i \equiv -f_i(\mathrm{X})$$

So the generalization is to first initialize X to a good initial value, calculate the matrix elements $a_{(i,j)}$ and the vector b_i, and then to proceed to calculate dX by solving the matrix equation, and calculating

$$\mathrm{X}_{i+1} \equiv \mathrm{X}_i + dX_i$$

In the case of one function with one variable, the summation reduces to one term, so this linear set of equations was a lot simpler in that case. In this case we will have to solve this set of linear equations in each iteration.

As an example, suppose we want to find the zeroes for the following two functions:

$$f_1(a,x) \equiv \sin ax$$

and

$$f_2(a,x) \equiv a-2$$

It is clear that the solution to this is $a = 2$ and $x \equiv N\frac{\pi}{2}$ for any integer value N.

We will do calculations with precision 30:

```
In> Builtin'Precision'Set(30)
Out> True;
```

And set up a vector of functions `f_1(X),f_2(X)` where $X :=$ `a,x`

```
In> f(a,x):={Sin(a*x),a-2}
Out> True;
```

Now we set up a function `matrix(a,x)` which returns the matrix $a_{(i,j)}$:

```
In> matrix(a,x):=Eval({D(a)f(a,x),D(x)f(a,x)})
Out> True;
```

We now set up some initial values:

```
In> {a,x}:={1.5,1.5}
Out> {1.5,1.5};
```

The iteration converges a lot slower for this example, so we will loop 100 times:

```
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
   N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,0.059667311457823162437151576236};
```

The value for a has already been found. Iterating a few more times:

```
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
   N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,-0.042792753588155918852832259721};
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
   N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,0.035119151349413516969586788023};
```

the value for x converges a lot slower this time, and to the uninteresting value of zero (a rather trivial zero of this set of functions). In fact for all integer values N the value $N\frac{\pi}{2}$ is a solution. Trying various initial values will find them.

Newton iteration on polynomials

von zur Gathen et al., *Modern Computer algebra* discusses taking the inverse of a polynomial using Newton iteration. The task is, given a polynomial $f(x)$, to find a polynomial $g(x)$ such that $f(x) = \frac{1}{g(x)}$, modulo some power in x. This implies that we want to find a polynom g for which:

$$h(g) = \frac{1}{g} - f = 0$$

Applying a Newton iteration step $g_{i+1} = g_i - \frac{h(g_i)}{\frac{\partial}{\partial g}h(g_i)}$ to this expression yields:

$$g_{i+1} = 2g_i - fg_i^2$$

von zur Gathen then proves by induction that for $f(x)$ monic, and thus $f(0) = 1$, given initial value $g_0(x) = 1$, that

$$(fg_i = 1) \bmod x^{2^i}$$

Example:

suppose we want to find the polynomial $g(x)$ up to the 7th degree for which $(f(x)\,g(x) = 1) \bmod x^8$, for the function

$$f(x) \equiv 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$$

First we define the function f:

```
In> f:=1+x+x^2/2+x^3/6+x^4/24
Out> x+x^2/2+x^3/6+x^4/24+1;
```

And initialize g and i.

```
In> g:=1
Out> 1;
In> i:=0
Out> 0;
```

Now we iterate, increasing i, and replacing g with the new value for g:

```
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> 1-x;
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> x^2/2-x^3/6-x+1;
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> x^7/72-x^6/72+x^4/24-x^3/6+x^2/2-x+1;
```

The resulting expression must thus be:

$$g(x) \equiv \frac{x^7}{72} - \frac{x^6}{72} + \frac{x^4}{24} - \frac{x^3}{6} + \frac{x^2}{2} - x + 1$$

We can easily verify this:

```
In> Expand(f*g)
Out> x^11/1728+x^10/576+x^9/216+(5*x^8)/576+1;
```

This expression is 1 modulo x^8, as can easily be shown:

```
In> BigOh(%,x,8)
Out> 1;
```

Chapter 4

Numerical algorithms I: basic methods

This and subsequent chapters document the numerical algorithms used in Yacas for exact integer calculations as well as for multiple precision floating-point calculations. We give self-contained descriptions of the non-trivial algorithms and estimate their computational cost. Most of the algorithms were taken from referenced literature; the remaining algorithms were developed by us.

4.1 Adaptive function plotting

Here we consider plotting of functions $y = f(x)$.

There are two tasks related to preparation of plots of functions: first, to produce the numbers required for a plot, and second, to draw a plot with axes, symbols, a legend, perhaps additional illustrations and so on. Here we only concern ourselves with the first task, that of preparation of the numerical data for a plot. There are many plotting programs that can read a file with numbers and plot it in any desired manner.

Generating data for plots of functions generally does not require high-precision calculations. However, we need an algorithm that can be adjusted to produce data to different levels of precision. In some particularly ill-behaved cases, a precise plot will not be possible and we would not want to waste time producing data that is too accurate for what it is worth.

A simple approach to plotting would be to divide the interval into many equal subintervals and to evaluate the function on the resulting grid. Precision of the plot can be adjusted by choosing a larger or a smaller number of points.

However, this approach is not optimal. Sometimes a function changes rapidly near one point but slowly everywhere else. For example, $f(x) = \frac{1}{x}$ changes very quickly at small x. Suppose we need to plot this function between 0 and 100. It would be wasteful to use the same subdivision interval everywhere: a finer grid is only required over a small portion of the plotting range near $x = 0$.

The adaptive plotting routine `Plot2D'adaptive` uses a simple algorithm to select the optimal grid to approximate a function of one argument $f(x)$. The algorithm repeatedly subdivides the grid intervals near points where the existing grid does not represent the function well enough. A similar algorithm for adaptive grid refinement could be used for numerical integration. The idea is that plotting and numerical integration require the same kind of detailed knowledge about the behavior of the function.

The algorithm first splits the interval into a specified initial number of equal subintervals, and then repeatedly splits each subinterval in half until the function is well enough approximated by the resulting grid. The integer parameter `depth` gives the maximum number of binary splittings for a given initial interval; thus, at most 2^{depth} additional grid points will be generated. The function `Plot2D'adaptive` should return a list of pairs of points `{{x1,y1}, {x2,y2}, ...}` to be used directly for plotting.

The adaptive plotting algorithm works like this:

- 1. Given an interval $(a,\ c)$, we split it in half, $b \equiv \frac{a+c}{2}$ and first compute $f(x)$ at five grid points a, $a_1 \equiv \frac{a+b}{2}$, b, $b_1 \equiv \frac{b+c}{2}$, c.
- 2. If currently depth ≤ 0, return this list of five points and values because we cannot refine the grid any more.
- 3. Otherwise, check that the function does not oscillate too rapidly on the interval $[a,\ c]$. The formal criterion is that the five values are all finite and do not make a "zigzag" pattern such as (1,3,2,3,1). More formally, we use the following procedure: For each three consecutive values, write "1" if the middle value is larger than the other two, or if it is smaller than the other two, or if one of them is not a number (e.g. `Infinity` or `Undefined`). If we have at most two ones now, then we consider the change of values to be "slow enough". Otherwise it is not "slow enough". In this case we need to refine the grid; go to step 5. Otherwise, go to step 4.
- 4. Check that the function values are smooth enough through the interval. Smoothness is controlled by a parameter ϵ. The meaning of the parameter ϵ is the (relative) error of the numerical approximation of the integral of $f(x)$ by the grid. A good heuristic value of ϵ is 1/(the number of pixels on the screen) because it means that no pixels will be missing in the area under the graph. For this to work we need to make sure that we are actually computing the area *under* the graph; so we define $g(x) \equiv f(x) - f_0$ where f_0 is the minimum of the values of $f(x)$ on the five grid points a, a_1, b, b_1, and c; the function $g(x)$ is nonnegative and has the minimum value 0. Then we compute two different Newton-Cotes quadratures for $\int_b^{b_1} g(x)\,dx$ using these five points. (Asymmetric quadratures are chosen to avoid running into an accidental symmetry of the function; the first quadrature uses points a, a_1, b, b_1 and the second quadrature uses b, b_1, c.) If the absolute value of the difference between these quadratures is less than ϵ * (value of the second quadrature), then we are done and we return the list of these five points and values.
- 5. Otherwise, we need to refine the grid. We compute `Plot2D'adaptive` recursively for the two halves of the interval, that is, for $(a,\ b)$ and $(b,\ c)$. We also decrease `depth` by 1 and multiply ϵ by 2 because we need to maintain a constant *absolute* precision and this means that the relative error for the two subintervals can be twice as large. The resulting two lists for the two subintervals are concatenated (excluding the double value at point b) and returned.

This algorithm works well if the initial number of points and the `depth` parameter are large enough. These parameters can be adjusted to balance the available computing time and the desired level of detail in the resulting plot.

Singularities in the function are handled by the step 3. Namely, the change in the sequence a, a_1, b, b_1, c is always considered to be "too rapid" if one of these values is a non-number (e.g. `Infinity` or `Undefined`). Thus, the interval immediately adjacent to a singularity will be plotted at the highest allowed refinement level. When preparing the plotting data, the singular points are simply not printed to the data file, so that a plotting programs does not encounter any problems.

Newton-Cotes quadratures

The meaning of Newton-Cotes quadrature coefficients is that an integral of a function $f(x)$ is approximated by a sum,

$$\int_{a_0}^{a_n} f(x)\,dx \approx h\sum_{k=0}^{n} c_k f(a_k),$$

where a_k are the grid points, $h \equiv a_1 - a_0$ is the grid step, and c_k are the quadrature coefficients. It may seem surprising, but these coefficients c_k are independent of the function $f(x)$ and can be precomputed in advance for a given grid a_k. [The quadrature coefficients do depend on the relative separations of the grid. Here we assume a uniform grid with a constant step $h = a_k - a_{k-1}$. Quadrature coefficients can also be found for non-uniform grids.]

The coefficients c_k for grids with a constant step h can be found, for example, by solving the following system of equations,

$$\sum_{k=0}^{n} c_k k^p = \frac{n^{p+1}}{p+1}$$

for $p = 0, 1, ..., n$. This system of equations means that the quadrature correctly gives the integrals of $p+1$ functions $f(x) = x^p$, $p = 0, 1, ..., n$, over the interval $(0, n)$. The solution of this system always exists and gives quadrature coefficients as rational numbers. For example, the well-known Simpson quadrature $c_0 = \frac{1}{3}$, $c_1 = \frac{4}{3}$, $c_2 = \frac{1}{3}$ is obtained with $n = 2$. An example of using this quadrature is the approximation

$$\int_0^2 f(x)\,dx \approx \frac{f(0)+f(2)}{3} + \frac{4}{3}f(1).$$

In the same way it is possible to find quadratures for the integral over a subinterval rather than over the whole interval of x. In the current implementation of the adaptive plotting algorithm, two quadratures are used: the 3-point quadrature ($n = 2$) and the 4-point quadrature ($n = 3$) for the integral over the first subinterval, $\int_{a_0}^{a_1} f(x)\,dx$. Their coefficients are ($\frac{5}{12}$, $\frac{2}{3}$, $-\frac{1}{12}$) and ($\frac{3}{8}$, $\frac{19}{24}$, $-\frac{5}{24}$, $\frac{1}{24}$). An example of using the first of these subinterval quadratures would be the approximation

$$\int_0^1 f(x)\,dx \approx \frac{5}{12}f(0) + \frac{2}{3}f(1) - \frac{1}{12}f(2).$$

These quadratures are intentionally chosen to be asymmetric to avoid an accidental cancellation when the function $f(x)$ itself is symmetric. (Otherwise the error estimate could accidentally become exactly zero.)

4.2 Surface plotting

Here we consider plotting of functions $z = f(x, y)$.

The task of surface plotting is to obtain a picture of a two-dimensional surface as if it were a solid object in three dimensions. A graphical representation of a surface is a complicated task. Sometimes it is required to use particular coordinates or projections, to colorize the surface, to remove hidden lines and so on. We shall only be concerned with the task of obtaining the data for a plot from a given function of two variables $f(x, y)$. Specialized programs can take a text file with the data and let the user interactively produce a variety of surface plots.

The currently implemented algorithm in the function `Plot3DS` is very similar to the adaptive plotting algorithm for two-dimensional plots. A given rectangular plotting region $a_1 \le x \le a_2$, $b_1 \le y \le b_2$ is subdivided to produce an equally spaced rectangular grid of points. This is the initial grid which will be adaptively refined where necessary. The refinement algorithm will divide a given rectangle in four quarters if the available function values indicate that the function does not change smoothly enough on that rectangle.

The criterion of a "smooth enough" change is very similar to the procedure outlined in the previous section. The change is "smooth enough" if all points are finite, nonsingular values, and if the integral of the function over the rectangle is sufficiently well approximated by a certain low-order "cubature" formula.

The two-dimensional integral of the function is estimated using the following 5-point Newton-Cotes cubature:

```
1/12   0   1/12

 0    2/3   0

1/12   0   1/12
```

An example of using this cubature would be the approximation

$$\int_0^1\int_0^1 f(x,y)\,dxdy \approx \frac{f(0,0)+f(0,1)+f(1,0)+f(1,1)}{12} + \frac{2}{3}f\left(\frac{1}{2},\frac{1}{2}\right).$$

Similarly, an 8-point cubature with zero sum is used to estimate the error:

```
-1/3   2/3   1/6

-1/6  -2/3  -1/2

 1/2    0    1/3
```

This set of coefficients was intentionally chosen to be asymmetric to avoid possible exact cancellations when the function itself is symmetric.

One minor problem with adaptive surface plotting is that the resulting set of points may not correspond to a rectangular grid in the parameter space (x,y). This is because some rectangles from the initial grid will need to be bisected more times than others. So, unless adaptive refinement is disabled, the function `Plot3DS` produces a somewhat disordered set of points. However, most surface plotting programs require that the set of data points be a rectangular grid in the parameter space. So a smoothing and interpolation procedure is necessary to convert a non-gridded set of data points ("scattered" data) to a gridded set.

4.3 Parametric plots

Currently, parametric plots are not directly implemented in Yacas. However, it is possible to use Yacas to obtain numerical data for such plots. One can then use external programs to produce actual graphics.

A two-dimensional parametric plot is a line in a two-dimensional space, defined by two equations such as $x = f(t)$, $y = g(t)$. Two functions f, g and a range of the independent variable t, for example, $t_1 \leq t \leq t_2$, need to be specified.

Parametric plots can be used to represent plots of functions in non-Euclidean coordinates. For example, to plot the function $\rho = (\cos 4\phi)^2$ in polar coordinates (ρ,ϕ), one can rewrite the Euclidean coordinates through the polar coordinates, $x = \rho\cos\phi$, $y = \rho\sin\phi$, and use the equivalent parametric plot with ϕ as the parameter: $x = (\cos 4\phi)^2 \cos\phi$, $y = (\cos 4\phi)^2 \sin\phi$.

Sometimes higher-dimensional parametric plots are required. A line plot in three dimensions is defined by three functions of one variable, for example, $x = f(t)$, $y = g(t)$, $z = h(t)$, and a range of the parameter t. A surface plot in three dimensions is defined by three functions of two variables each, for example, $x = f(u,v)$, $y = g(u,v)$, $z = h(u,v)$, and a rectangular domain in the (u,v) space.

The data for parametric plots can be generated separately using the same adaptive plotting algorithms as for ordinary function plots, as if all functions such as $f(t)$ or $g(u,v)$ were unrelated functions. The result would be several separate data sets for the x, y, ... coordinates. These data sets could then be combined using an interactive plotting program.

4.4 The cost of arbitrary-precision computations

A computer algebra system absolutely needs to be able to perform computations with very large *integer* numbers. Without this capability, many symbolic computations (such as exact GCD of polynomials or exact solution of polynomial equations) would be impossible.

A different question is whether a CAS really needs to be able to evaluate, say, 10,000 digits of the value of a Bessel function of some 10,000-digit complex argument. It seems likely that no applied problem of natural sciences would need floating-point computations of special functions with such a high precision. However, arbitrary-precision computations are certainly useful in some mathematical applications; e.g. some mathematical identities can be first guessed by a floating-point computation with many digits and then proved.

Very high precision computations of special functions *might* be useful in the future. But it is already quite clear that computations with moderately high precision (say, 50 or 100 decimal digits) are useful for applied problems. For example, to obtain the leading asymptotic of an analytic function, we could expand it in series and take the first term. But we need to check that the coefficient at what we think is the leading term of the series does not vanish. This coefficient could be a certain "exact" number such as $(\cos 355 + 1)^2$. This number is "exact" in the sense that it is made of integers and elementary functions. But we cannot say *a priori* that this number is nonzero. The problem of "zero determination" (finding out whether a certain "exact" number is zero) is known to be algorithmically unsolvable if we allow transcendental functions. The only practical general approach seems to be to compute the number in question with many digits. Usually a few digits are enough, but occasionally several hundred digits are needed.

Implementing an efficient algorithm that computes 100 digits of $\sin\frac{3}{7}$ already involves many of the issues that would also be relevant for a 10,000 digit computation. Modern algorithms allow evaluations of all elementary functions in time that is asymptotically logarithmic in the number of digits P and linear in the cost of long multiplication (usually denoted $M(P)$). Almost all special functions can be evaluated in time that is asymptotically linear in P and in $M(P)$. (However, this asymptotic cost sometimes applies only to very high precision, e.g., $P > 1000$, and different algorithms need to be implemented for calculations in lower precision.)

In Yacas we strive to implement all numerical functions to arbitrary precision. All integer or rational functions return exact results, and all floating-point functions return their value with P correct decimal digits (assuming sufficient precision of the arguments). The current value of P is accessed as `Builtin'Precision'Get()` and may be changed by `Builtin'Precision'Set(...)`.

Implementing an arbitrary-precision floating-point computation of a function $f(x)$, such as $f(x) = \exp(x)$, typically needs the following:

- An algorithm that will compute $f(x)$ for a given value x to a user-specified precision of P (decimal) digits. Often, several algorithms must be implemented for different subdomains of the (x,P) space.
- An estimate of the computational cost of the algorithm(s), as a function of x and P. This is needed to select the best algorithm for given x, P.
- An estimate of the round-off error. This is needed to select the "working precision" which will typically be somewhat higher than the precision of the final result.

In calculations with machine precision where the number of digits is fixed, the problem of round-off errors is quite prominent. Every arithmetic operation causes a small loss of precision; as a result, a few last digits of the final value are usually incorrect. But if we have an arbitrary precision capability, we can always increase precision by a few more digits during intermediate computations and thus eliminate all round-off error in the final result. We should, of course, take care not to increase the working precision unnecessarily, because any increase of precision means slower calculations. Taking twice as many digits as needed and hoping that the result is precise is not a good solution.

Selecting algorithms for computations is the most non-trivial part of the implementation. We want to achieve arbitrarily high precision, so we need to find either a series, or a continued fraction, or a sequence given by explicit formula, that converges to the function in a controlled way. It is not enough to use a table of precomputed values or a fixed approximation formula that has a limited precision.

In the last 30 years, the interest in arbitrary-precision computations grew and many efficient algorithms for elementary and special functions were published. Most algorithms are iterative. Almost always it is very important to know in advance how many iterations are needed for given x, P. This knowledge allows to estimate the computational cost, in terms of the required precision P and of the cost of long multiplication $M(P)$, and choose the best algorithm.

Typically all operations will fall into one of the following categories (sorted by the increasing cost):

- addition, subtraction: linear in P;
- multiplication, division, integer power, integer root: linear in $M(P)$;

- elementary functions: $\exp(x)$, $\ln x$, $\sin x$, $\arctan x$ etc.: $M(P)\ln P$ or slower by some powers of $\ln P$;
- transcendental functions: $\operatorname{erf} x$, $\Gamma(x)$ etc.: typically $PM(P)$ or slower.

The cost of long multiplication $M(P)$ is between $O\left(P^2\right)$ for low precision and $O(P\ln P)$ for very high precision. In some cases, a different algorithm should be chosen if the precision is high enough to allow $M(P)$ faster than $O\left(P^2\right)$.

Some algorithms also need storage space (e.g. an efficient algorithm for summation of the Taylor series uses $O(\ln P)$ temporary P-digit numbers).

Below we shall normally denote by P the required number of decimal digits. The formulae frequently contain conspicuous factors of $\ln 10$, so it will be clear how to obtain analogous expressions for another base. (Most implementations use a binary base rather than a decimal base since it is more convenient for many calculations.)

4.5 Estimating convergence of a series

Analyzing convergence of a power series is usually not difficult. Here is a worked-out example of how we could estimate the required number of terms in the power series

$$\exp(x) = 1 + x + \frac{x^2}{2!} + ... + \frac{x^n}{n!} + O\left(x^{n+1}\right)$$

if we need P decimal digits of precision in the result. To be specific, assume that $|x| < 1$. (A similar calculation can be done for any other bound on x.)

Suppose we truncate the series after n-th term and the series converges "well enough" after that term. Then the error will be approximately equal to the first term we dropped. (This is what we really mean by "converges well enough" and this will generally be the case in all applications, because we would not want to use a series that does not converge well enough.)

The term we dropped is $\frac{x^{n+1}}{(n+1)!}$. To estimate $n!$ for large n, one can use the inequality

$$e^{e-1}\left(\frac{n}{e}\right)^n < n! < \left(\frac{n}{e}\right)^{n+1}$$

(valid for all $n \geq 47$) which provides tight bounds for the growth of the factorial, or a weaker inequality which is somewhat easier to use,

$$\left(\frac{n}{e}\right)^n < n! < \left(\frac{n+1}{e}\right)^{n+1}$$

(valid for all $n \geq 6$). The latter inequality is sufficient for most purposes.

If we use the upper bound on $n!$ from this estimate, we find that the term we dropped is bounded by

$$\frac{x^{n+1}}{(n+1)!} < \left(\frac{e}{n+2}\right)^{n+2}.$$

We need this number to be smaller than 10^{-P}. This leads to an inequality

$$\left(\frac{e}{n+2}\right)^{n+2} < 10^{-P},$$

which we now need to solve for n. The left hand side decreases with growing n. So it is clear that the inequality will hold for large enough n, say for $n \geq n_0$ where n_0 is an unknown (integer) value. We can take a logarithm of both sides, replace n with n_0 and obtain the following equation for n_0:

$$(n_0+2)\ln\frac{n_0+2}{e} = P\ln 10.$$

This equation cannot be solved exactly in terms of elementary functions; this is a typical situation in such estimates. However, we do not really need a very precise solution for n_0; all we need is an estimate of its integer part. This is also a typical situation. It is acceptable if our approximate value of n_0 comes out a couple of units higher than necessary, because a couple of extra terms of the Taylor series will not significantly slow down the algorithm (but it is important that we do not underestimate n_0). Finally, we are mostly interested in having a good enough answer for large values of P.

We can try to guess the result. The largest term on the LHS grows as $n_0\ln n_0$ and it should be approximately equal to $P\ln 10$; but $\ln n_0$ grows very slowly, so this gives us a hint that n_0 is proportional to $P\ln 10$. As a first try, we set $n_0 = P\ln 10 - 2$ and compare the RHS with the LHS; we find that we have overshot by a factor $\ln P - 1 + \ln\ln 10$, which is not a large factor. We can now compensate and divide n_0 by this factor, so our second try is

$$n_0 = \frac{P\ln 10}{\ln P - 1 + \ln\ln 10} - 2.$$

(This approximation procedure is equivalent to solving the equation

$$x = \frac{P\ln 10}{\ln x - 1}$$

by direct iteration, starting from $x = P\ln 10$.) If we substitute our second try for n_0 into the equation, we shall find that we undershot a little bit (i.e. the LHS is a little smaller than the RHS), but our n_0 is now smaller than it should be by a quantity that is smaller than 1 for large enough P. So we should stop at this point and simply add 1 to this approximate answer. We should also replace $\ln\ln 10 - 1$ by 0 for simplicity (this is safe because it will slightly increase n_0.)

Our final result is that it is enough to take

$$n = \frac{P\ln 10}{\ln P} - 1$$

terms in the Taylor series to compute $\exp(x)$ for $|x| < 1$ to P decimal digits. (Of course, if x is much smaller than 1, many fewer terms will suffice.)

4.6 Estimating the round-off error

Unavoidable round-off errors

As the required precision P grows, an arbitrary-precision algorithm will need more iterations or more terms of the series. So the round-off error introduced by every floating-point operation will increase. When doing arbitrary-precision computations, we can always perform all calculations with a few more digits and compensate for round-off error. It is however imperative to know in advance how many more digits we need to take for our "working precision". We should also take that increase into account when estimating the total cost of the method. (In most cases this increase is small.)

Here is a simple estimate of the normal round-off error in a computation of n terms of a power series. Suppose that the sum of the series is of order 1, that the terms monotonically decrease in magnitude, and that adding one term requires two multiplications and one addition. If all calculations are performed with

absolute precision $\epsilon = 10^{-P}$, then the total accumulated round-off error is $3n\epsilon$. If the relative error is $3n\epsilon$, it means that our answer is something like $a\,(1 + 3n\epsilon)$ where a is the correct answer. We can see that out of the total P digits of this answer, only the first k decimal digits are correct, where $k = -\frac{\ln 3n\epsilon}{\ln 10}$. In other words, we have lost

$$P - k = \frac{\ln 3n}{\ln 10}$$

digits because of accumulated round-off error. So we found that we need $\frac{\ln 3n}{\ln 10}$ extra decimal digits to compensate for this round-off error.

This estimate assumes several things about the series (basically, that the series is "well-behaved"). These assumptions must be verified in each particular case. For example, if the series begins with some large terms but converges to a very small value, this estimate is wrong (see the next subsection).

In the previous exercise we found the number of terms n for $\exp(x)$. So now we know how many extra digits of working precision we need for this particular case.

Below we shall have to perform similar estimates of the required number of terms and of the accumulated round-off error in our analysis of the algorithms.

Catastrophic round-off error

Sometimes the round-off error of a particular method of computation becomes so large that the method becomes highly inefficient.

Consider the computation of $\sin x$ by the truncated Taylor series

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

when x is large. We know that this series converges for all x, no matter how large. Assume that $x = 10^M$ with $M \geq 1$, and that we need P decimal digits of precision in the result.

First, we determine the necessary number of terms N. The magnitude of the sum is never larger than 1. Therefore we need the N-th term of the series to be smaller than 10^{-P}. The inequality is $(2N+1)! > 10^{P+M(2N+1)}$. We obtain that $2N + 2 > e \cdot 10^M$ is a necessary condition, and if P is large, we find approximately

$$2N + 2 \approx \frac{(P - M)\ln 10}{\ln(P - M) - 1 - M\ln 10}.$$

However, taking enough terms does not yet guarantee a good result. The terms of the series grow at first and then start to decrease. The sum of these terms is, however, small. Therefore there is some cancellation and we need to increase the working precision to avoid the round-off. Let us estimate the required working precision.

We need to find the magnitude of the largest term of the series. The ratio of the next term to the previous term is $\frac{x}{2k(2k+1)}$ and therefore the maximum will be when this ratio becomes equal to 1, i.e. for $2k \approx \sqrt{x}$. Therefore the largest term is of order $\frac{x^{\sqrt{x}}}{\sqrt{x}!}$ and so we need about $\frac{M}{2}\sqrt{x}$ decimal digits before the decimal point to represent this term. But we also need to keep at least P digits after the decimal point, or else the round-off error will erase the significant digits of the result. In addition, we will have unavoidable round-off error due to $O(P)$ arithmetic operations. So we should increase precision again by $P + \frac{\ln P}{\ln 10}$ digits plus a few guard digits.

As an example, to compute $\sin 10$ to $P = 50$ decimal digits with this method, we need a working precision of about 60 digits, while to compute $\sin 10000$ we need to work with about 260 digits. This shows how inefficient the Taylor series for $\sin x$ becomes for large arguments x. A simple transformation $x = 2\pi n + x'$ would have reduced x to at most 7, and the unnecessary computations with 260 digits would be avoided. The main cause of this inefficiency is that we have to add and subtract extremely large numbers to get a relatively small result of order 1.

We find that the method of Taylor series for $\sin x$ at large x is highly inefficient because of round-off error and should be complemented by other methods. This situation seems to be typical for Taylor series.

4.7 Basic arbitrary-precision arithmetic

Yacas uses an internal math library (the `yacasnumbers` library) which comes with the source code. This reduces the dependencies of the Yacas system and improves portability. The internal math library is simple and does not necessarily use the most optimal algorithms.

If P is the number of digits of precision, then multiplication and division take $M(P) = O\left(P^2\right)$ operations in the internal math. (Of course, multiplication and division by a short integer takes time linear in P.) Much faster algorithms (Karatsuba, Toom-Cook, FFT multiplication, Newton-Raphson division etc.) are implemented in `gmp`, `CLN` and some other libraries. The asymptotic cost of multiplication for very large precision is $M(P) \approx O\left(P^{1.6}\right)$ for the Karatsuba method and $M(P) = O(P \ln P \ln\ln P)$ for the FFT method. In the estimates of computation cost in this book we shall assume that $M(P)$ is at least linear in P and maybe a bit slower.

The costs of multiplication may be different in various arbitrary-precision arithmetic libraries and on different computer platforms. As a rough guide, one can assume that the straightforward $O\left(P^2\right)$ multiplication is good until 100-200 decimal digits, the asymptotically fastest method of FFT multiplication is good at the precision of about 5,000 or more decimal digits, and the Karatsuba multiplication is best in the middle range.

Warning: calculations with internal Yacas math using precision exceeding 10,000 digits are currently impractically slow.

In some algorithms it is necessary to compute the integer parts of expressions such as $a\frac{\ln b}{\ln 10}$ or $a\frac{\ln 10}{\ln 2}$, where a, b are short integers of order $O(P)$. Such expressions are frequently needed to estimate the number of terms in the Taylor series or similar parameters of the algorithms. In these cases, it is important that the result is not underestimated. However, it would be wasteful to compute $1000\frac{\ln 10}{\ln 2}$ in great precision only to discard most of that information by taking the integer part of that number. It is more efficient to approximate such constants from above by short rational numbers, for example, $\frac{\ln 10}{\ln 2} < \frac{28738}{8651}$ and $\ln 2 < \frac{7050}{10171}$. The error of such an approximation will be small enough for practical purposes. The function `BracketRational` can be used to find optimal rational approximations.

The function `IntLog` (see below) efficiently computes the integer part of a logarithm (for an integer base, not a natural logarithm). If more precision is desired in calculating $\frac{\ln a}{\ln b}$ for integer a, b, one can compute $\text{IntLog}\left(a^k, b\right)$ for some integer k and then divide by k.

4.8 How many digits of $\sin\exp(\exp(1000))$ do we need?

Arbitrary-precision math is not omnipotent against overflow. Consider the problem of representing very large numbers such as $x = \exp(\exp(1000))$. Suppose we need a floating-point representation of the number x with P decimal digits of precision. In other words, we need to express $x \approx M \cdot 10^E$, where the mantissa $1 < M < 10$ is a floating-point number and the exponent E is an integer, chosen so that the relative precision is 10^{-P}. How much effort is needed to find M and E?

The exponent E is easy to obtain:

$$E = \left\lfloor \frac{\ln x}{\ln 10} \right\rfloor = \left\lfloor \frac{\exp(1000)}{\ln 10} \right\rfloor \approx 8.55 \cdot 10^{433}.$$

To compute the integer part $\lfloor y \rfloor$ of a number y exactly, we need to approximate y with at least $\frac{\ln y}{\ln 10}$ floating-point digits. In our example, we find that we need 434 decimal digits to represent E.

Once we found E, we can write $x = 10^{E+m}$ where $m = \frac{\exp(1000)}{\ln 10} - E$ is a floating-point number, $0 < m < 1$. Then $M = 10^m$. To find M with P (decimal) digits, we need m with also at least P digits. Therefore, we actually need to evaluate $\frac{\exp(1000)}{\ln 10}$ with $434 + P$ decimal digits before we can find P digits of the mantissa of x. We ran into a perhaps surprising situation: one needs a high-precision calculation even to find the first digit of x, because it is necessary to find the exponent E exactly as an integer, and E is a rather large integer. A normal double-precision numerical calculation would give an overflow error at this point.

Suppose we have found the number $x = \exp(\exp(1000))$ with some precision. What about finding $\sin x$? Now, this is extremely difficult, because to find even the first digit of $\sin x$ we have to evaluate x with *absolute* error of at most 0.5. We know, however, that the number x has approximately 10^{434} digits *before* the decimal point. Therefore, we would need to calculate x with at least that many digits. Computations with 10^{434} digits is clearly far beyond the capability of modern computers. It seems unlikely that even the sign of $\sin\exp(\exp(1000))$ will be obtained in the near future. [1]

Suppose that N is the largest integer that our arbitrary-precision facility can reasonably handle. (For Yacas internal math library, N is about 10^{10000}.) Then it follows that numbers x of order 10^N can be calculated with at most one (1) digit of floating-point precision, while larger numbers cannot be calculated with any precision at all.

It seems that very large numbers can be obtained in practice only through exponentiation or powers. It is unlikely that such numbers will arise from sums or products of reasonably-sized numbers in some formula. [2] For example, suppose a program operates with numbers x of size N or smaller; a number such as 10^N can be obtained only by multiplying $O(N)$ numbers x together. But since N is the largest representable number, it is certainly not feasible to perform $O(N)$ sequential operations on a computer. However, it is feasible to obtain N-th power of a small number, since it requires only $O(\ln N)$ operations.

If numbers larger than 10^N are created only by exponentiation operations, then special exponential notation could be used to represent them. For example, a very large number z could be stored and manipulated as an unevaluated exponential $z = \exp\left(M \cdot 10^E\right)$ where $M \geq 1$ is a floating-point number with P digits of mantissa and E is an integer, $\ln N < E < N$. Let us call such objects "exponentially large numbers" or "exp-numbers" for short.

In practice, we should decide on a threshold value N and promote a number to an exp-number when its logarithm exceeds N.

Note that an exp-number z might be positive or negative, e.g. $z = -\exp\left(M \cdot 10^E\right)$.

Arithmetic operations can be applied to the exp-numbers. However, exp-large arithmetic is of limited use because of an almost certainly critical loss of precision. The power and logarithm operations can be meaningfully performed on exp-numbers z. For example, if $z = \exp\left(M \cdot 10^E\right)$ and p is a normal floating-point number, then $z^p = \exp\left(pM \cdot 10^E\right)$ and $\ln z = M \cdot 10^E$. We can also multiply or divide two exp-numbers. But it makes no sense to multiply an exp-number z by a normal number because we cannot represent the difference between z and say $2.52z$. Similarly, adding z to anything else would result in a total underflow, since we do not actually know a single digit of the decimal representation of z. So if z_1 and z_2 are exp-numbers, then $z_1 + z_2$ is simply equal to either z_1 or z_2 depending on which of them is larger.

We find that an exp-number z acts as an effective "infinity" compared with normal numbers. But exp-numbers cannot be used as a device for computing limits: the unavoidable underflow will almost certainly produce wrong results. For example, trying to verify

$$\lim_{x \to 0} \frac{\exp(x) - 1}{x} = 1$$

by substituting $x = \frac{1}{z}$ with some exp-number z gives 0 instead of 1.

Taking a logarithm of an exp-number brings it back to the realm of normal, representable numbers. However, taking an exponential of an exp-number results in a number which is not representable even as an exp-number. This is because an exp-number z needs to have its exponent E represented exactly as an integer, but $\exp(z)$ has an exponent of order $O(z)$ which is not a representable number. The monstrous number $\exp(z)$ could be only written as $\exp\left(\exp\left(M \cdot 10^E\right)\right)$, i.e. as a "doubly exponentially large" number, or "2-exp-number" for short. Thus we obtain a hierarchy of iterated exp-numbers. Each layer is "unrepresentably larger" than the previous one.

The same considerations apply to very small numbers of the order 10^{-N} or smaller. Such numbers can be manipulated as "exponentially small numbers", i.e. expressions of the form $\exp\left(-M \cdot 10^E\right)$ with floating-point mantissa $M \geq 1$ and integer E satisfying $\ln N < E < N$. Exponentially small numbers act as an effective zero compared with normal numbers.

Taking a logarithm of an exp-small number makes it again a normal representable number. However, taking an exponential of an exp-small number produces 1 because of underflow. To obtain a "doubly exponentially small" number, we need to take a reciprocal of a doubly exponentially large number, or take the exponent of an exponentially large negative power. In other words, $\exp\left(-M \cdot 10^E\right)$ is exp-small, while $\exp\left(-\exp\left(M \cdot 10^E\right)\right)$ is 2-exp-small.

The practical significance of exp-numbers is rather limited. We cannot obtain even a single significant digit of an exp-number. A "computation" with exp-numbers is essentially a floating-point computation with logarithms of these exp-numbers. A practical problem that needs numbers of this magnitude can probably be restated in terms of more manageable logarithms of such numbers. In practice, exp-numbers could

[1] It seems even less likely that the sign of $\sin\exp(\exp(1000))$ would be of any use to anybody even if it could be computed.

[2] A factorial function can produce rapidly growing results, but exact factorials $n!$ for large n are well represented by the Stirling formula which involves powers and exponentials.

be useful not as a means to get a numerical answer, but as a warning sign of critical overflow or underflow. [3]

4.9 Continued fractions

A "continued fraction" is an expression of the form

$$a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \dots}}}.$$

The coefficients a_i, b_i are called the "terms" of the fraction. (Usually one has $a_i \neq 0$, $b_i \neq 0$.) The above continued fraction is sometimes written as

$$a_0 + \frac{b_0}{a_1 + \dots} \frac{b_1}{a_2 + \dots} \frac{b_2}{a_3 + \dots} \dots$$

Usually one considers infinite continued fractions, i.e. the sequences a_i, b_i are infinite. The value of an infinite continued fraction is defined as the limit of the fraction truncated after a very large number of terms. (A continued traction can be truncated after n-th term if one replaces b_n by 0.)

An infinite continued fraction does not always converge. Convergence depends on the values of the terms.

The representation of a number via a continued fraction is not unique because we could, for example, multiply the numerator and the denominator of any simple fraction inside it by any number. Therefore one may consider some normalized representations. A continued fraction is called "regular" if $b_k = 1$ for all k, all a_k are integer and $a_k > 0$ for $k \geq 1$. Regular continued fractions always converge.

Approximation of numbers by continued fractions

The function `ContFrac` converts a (real) number r into a regular continued fraction with integer terms,

$$r = n_0 + \cfrac{1}{n_1 + \cfrac{1}{n_2 + \dots}}.$$

Here all numbers n_i are integers and all except n_0 are positive. This representation of a real number r is unique. We may write this representation as $r = (n_0, n_1, n_2, ...)$. If r is a rational number, the corresponding regular continued fraction is finite, terminating at some n_N. Otherwise the continued fraction will be infinite. It is known that the truncated fractions will be in some sense "optimal" rational representations of the real number r.

The algorithm for converting a rational number $r = \frac{n}{m}$ into a regular continued fraction is simple. First, we determine the integer part of r, which is $\mathrm{Div}\,(n, m)$. If it is negative, we need to subtract one, so that $r = n_0 + x$ and the remainder x is nonnegative and less than 1. The remainder $x = \frac{n \bmod m}{m}$ is then inverted, $r_1 \equiv \frac{1}{x} = \frac{m}{n \bmod m}$ and so we have completed the first step in the decomposition, $r = n_0 + \frac{1}{r_1}$; now n_0 is integer but r_1 is perhaps not integer. We repeat the same procedure on r_1, obtain the next integer term n_1 and the remainder r_2 and so on, until such n that r_n is an integer and there is no more work to do. This process will always terminate.

If r is a real number which is known by its floating-point representation at some precision, then we can use the same algorithm because all floating-point values are actually rational numbers.

[3] Yacas currently does not implement exp-numbers or any other guards against overflow and underflow. If a decimal exponential becomes too large, an incorrect answer may result.

Real numbers known by their exact representations can sometimes be expressed as infinite continued fractions, for example

$$\sqrt{11} = (3, 3, 6, 3, 6, 3, 6, ...)\,;$$

$$\exp\left(\frac{1}{p}\right) = (1, p-1, 1, 1, 3p-1, 1, 1, 5p-1, ...)\,.$$

The functions `GuessRational` and `NearRational` take a real number x and use continued fractions to find rational approximations $r = \frac{p}{q} \approx x$ with "optimal" (small) numerators and denominators p, q.

Suppose we know that a certain number x is rational but we have only a floating-point representation of x with a limited precision, for example, $x \approx 1.5662650602409638$. We would like to guess a rational form for x (in this example $x = \frac{130}{83}$). The function `GuessRational` solves this problem.

Consider the following example. The number $\frac{17}{3}$ has a continued fraction expansion `{5,1,2}`. Evaluated as a floating point number with limited precision, it may become something like $\frac{17}{3} + 0.00001$, where the small number represents a round-off error. The continued fraction expansion of this number is `{5, 1, 2, 11110, 1, 5, 1, 3, 2777, 2}`. The presence of an unnaturally large term 11110 clearly signifies the place where the floating-point error was introduced; all terms following it should be discarded to recover the continued fraction `{5,1,2}` and from it the initial number $\frac{17}{3}$.

If a continued fraction for a number x is cut right before an unusually large term and evaluated, the resulting rational number has a small denominator and is very close to x. This works because partial continued fractions provide "optimal" rational approximations for the final (irrational) number, and because the magnitude of the terms of the partial fraction is related to the magnitude of the denominator of the resulting rational approximation.

`GuessRational(x, prec)` needs to choose the place where it should cut the continued fraction. The algorithm for this is somewhat heuristic but it works well enough. The idea is to cut the continued fraction when adding one more term would change the result by less than the specified precision. To realize this in practice, we need an estimate of how much a continued fraction changes when we add one term.

The routine `GuessRational` uses a (somewhat weak) upper bound for the difference of continued fractions that differ only by an additional last term:

$$|\delta| \equiv \left| \cfrac{1}{a_1 + \cfrac{1}{\dots + \frac{1}{a_n}}} - \cfrac{1}{a_1 + \cfrac{1}{\dots + \frac{1}{a_{n+1}}}} \right| < \frac{1}{(a_1 \dots a_n)^2\, a_{n+1}}.$$

(The derivation of this inequality is given below.) Thus we should compute the product of successive terms a_i of the continued fraction and stop at a_n at which this product exceeds the maximum number of digits. The routine `GuessRational` has a second parameter `prec` which is by default 1/2 times the number of decimal digits of current precision; it stops at a_n at which the product $a_1 \dots a_n$ exceeds 10^{prec}.

The above estimate for δ hinges on the inequality

$$\cfrac{1}{a + \cfrac{1}{b + \dots}} < \frac{1}{a}$$

and is suboptimal if some terms $a_i = 1$, because the product of a_i does not increase when one of the terms is equal to 1, whereas in fact these terms do make δ smaller. A somewhat better estimate would be obtained if we use the inequality

$$\cfrac{1}{a + \cfrac{1}{b + \cfrac{1}{c + \dots}}} < \cfrac{1}{a + \cfrac{1}{b + \frac{1}{c}}}.$$

(See the next section for more explanations of precision of continued fraction approximations.) This does not lead to a significant improvement if $a > 1$ but makes a difference when $a = 1$. In the product $a_1 ... a_n$, the terms a_i which are equal to 1 should be replaced by

$$a_i + \frac{1}{a_{i+1} + \frac{1}{a_{i+2}}}.$$

Since the comparison of $a_1 ... a_n$ with 10^{prec} is qualitative, it it enough to perform calculations of $a_1 ... a_n$ with limited precision.

This algorithm works well if x is computed with enough precision; namely, it must be computed to at least as many digits as there are in the numerator and the denominator of the fraction combined. Also, the parameter `prec` should not be too large (or else the algorithm will find another rational number with a larger denominator that approximates x "better" than the precision to which you know x).

The related function `NearRational(x, prec)` works somewhat differently. The goal is to find an "optimal" rational number, i.e. with smallest numerator and denominator, that is within the distance $10^{-\text{prec}}$ of a given value x. The function `NearRational` does not always give the same answer as `GuessRational`.

The algorithm for `NearRational` comes from the HAKMEM [Beeler *et al.* 1972], Item 101C. Their description is terse but clear:

```
Problem: Given an interval, find in it the
rational number with the smallest numerator and
denominator.
Solution: Express the endpoints as continued
fractions.  Find the first term where they differ
and add 1 to the lesser term, unless it's last.
Discard the terms to the right.  What's left is
the continued fraction for the "smallest"
rational in the interval.  (If one fraction
terminates but matches the other as far as it
goes, append an infinity and proceed as above.)
```

The HAKMEM text [Beeler *et al.* 1972] contains several interesting insights relevant to continued fractions and other numerical algorithms.

Accurate computation of continued fractions

Sometimes an analytic function $f(x)$ can be approximated using a continued fraction that contains x in its terms. Examples include the inverse tangent $\arctan x$, the error function $\operatorname{erf} x$, and the incomplete gamma function $\Gamma(a, x)$ (see below for details). For these functions, continued fractions provide a method of numerical calculation that works when the Taylor series converges slowly or not at all, or is not easily available. However, continued fractions may converge quickly for one value of x but slowly for another. Also, it is not as easy to obtain an analytic error bound for a continued fraction approximation as it is for power series.

In this section we describe some methods for computing general continued fractions and for estimating the number of terms needed to achieve a given precision.

Let us introduce some notation. A continued fraction

$$a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \ldots}}$$

is specified by a set of terms (a_i, b_i). [If continued fractions are used to approximate analytic functions such as $\arctan x$, then (a_i, b_i) will depend on x.] Let us denote by $F_{(m,n)}$ the truncated fraction containing only the terms from m to n,

$$F_{(m,n)} \equiv a_m + \frac{b_m}{a_{m+1} + \frac{b_{m+1}}{\ldots + \frac{b_n}{a_n}}}.$$

In this notation, the continued fraction that we need to compute is $F_{(0,n)}$. Our task is first, to select a large enough n so that $F_{(0,n)}$ gives enough precision, and second, to compute that value.

Method 1: bottom-up with straightforward division

All "bottom-up" methods need to know the number of terms n in advance. The simplest method is to start evaluating the fraction from the bottom upwards. As the initial approximation we take $F_{(n,n)} = a_n$. Then we use the obvious relation of backward recurrence,

$$F_{(m,n)} = a_m + \frac{b_m}{F_{(m+1,n)}},$$

to obtain successively $F_{(n-1,n)}$, ..., $F_{(0,n)}$.

This method requires one long division at each step. There may be significant round-off error if a_m and b_m have opposite signs, but otherwise the round-off error is very small because a convergent continued fraction is not sensitive to small changes in its terms.

Method 2: bottom-up with two recurrences

An alternative implementation may be faster in some cases. The idea is to obtain the numerator and the denominator of $F_{(0,n)}$ separately as two simultaneous backward recurrences. If $F_{(m+1,n)} = \frac{p_{m+1}}{q_{m+1}}$, then $p_m = a_m p_{m+1} + b_m q_{m+1}$ and $q_m = p_{m+1}$. The recurrences start with $p_n = a_n$, $q_n = 1$. The method requires two long multiplications at each step; the only division will be performed at the very end. Sometimes this method reduces the round-off error as well.

Method 3: bottom-up with estimated remainders

There is an improvement over the bottom-up method that can sometimes increase the achieved precision without computing more terms. This trick is suggested in [Tsimring 1988], sec. 2.4, where it is also claimed that the remainder estimates improve convergence.

The idea is that the starting value of the backward recurrence should be chosen not as a_n but as another number that more closely approximates the infinite remainder of the fraction. The infinite remainder, which we can symbolically write as $F_{(n,\infty)}$, can be sometimes estimated analytically (obviously, we are unable to compute the remainder exactly). In simple cases, $F_{(n,\infty)}$ changes very slowly at large n (warning: this is not always true and needs to be verified in each particular case!). Suppose that $F_{(n,\infty)}$ is approximately constant; then it must be approximately equal to $F_{(n+1,\infty)}$. Therefore, if we solve the (quadratic) equation

$$x = a_n + \frac{b_n}{x},$$

we shall obtain the (positive) value x which may be a much better approximation for $F_{(n,\infty)}$ than a_n. But this depends on the assumption of the way the continued fraction converges. It may happen, for example, that for large n the value $F_{(n,\infty)}$ is almost

the same as $F_{(n+2,\infty)}$ but is significantly different from $F_{(n+1,\infty)}$. Then we should instead solve the (quadratic) equation

$$x = a_n + \frac{b_n}{a_{n+1} + \frac{b_{n+1}}{x}}$$

and take the positive solution x as $F_{(n,\infty)}$.

We may use more terms of the original continued fraction starting from a_n and obtain a more precise estimate of the remainder. In each case we will only have to solve one quadratic equation.

Method 4: top-down computation

The "bottom-up" method obviously requires to know the number of terms n in advance; calculations have to be started all over again if more terms are needed. Also, the bottom-up method provides no error estimate.

The "top-down" method is slower but provides an automatic error estimate and can be used to evaluate a continued fraction with more and more terms until the desired precision is achieved. The idea [4] is to replace the continued fraction $F_{(0,n)}$ with a sum of a certain series,

$$a_0 + \frac{b_0}{a_1 + \frac{b_1}{\ldots + \frac{b_{n-1}}{a_n}}} = \sum_{k=0}^{n} f_k.$$

Here

$$f_k \equiv F_{(0,k)} - F_{(0,k-1)}$$

($k \geq 1$) is a sequence that will be calculated in the forward direction, starting from $k = 1$. If we manage to find a formula for this sequence, then adding one more term f_k will be equivalent to recalculating the continued fraction with k terms instead of $k - 1$ terms. This will automatically give an error estimate and allow to compute with more precision if necessary without having to repeat the calculation from the beginning. (The transformation of the continued fraction into a series is exact, not merely an approximation.)

The formula for f_k is the following. First the auxiliary sequence P_k, Q_k for $k \geq 1$ needs to be defined by $P_1 = 0$, Q_1=1, and $P_{k+1} \equiv b_k Q_k$, $Q_{k+1} \equiv P_k + a_k Q_k$. Then define $f_0 \equiv a_0$ and

$$f_k \equiv \frac{(-1)^k \, b_0 \ldots b_{k-1}}{Q_k Q_{k+1}}$$

for $k \geq 1$. The "top-down" method consists of computing f_n sequentially and adding them together, until n is large enough so that $\frac{f_n}{f_0}$ is less than the required precision.

Evaluating the next element f_k requires four long multiplications and one division. This is significantly slower, compared with just one long division or two long multiplications in the bottom-up method. Therefore it is desirable to have an a priori estimate of the convergence rate and to be able to choose the number of terms before the computation. Below we shall consider some examples when the formula for f_k allows to estimate the required number of terms analytically.

Method 5: top-down with two steps at once

If all coefficients a_i, b_i are positive, then the series we obtained in the top-down method will have terms f_k with alternating signs. This leads to a somewhat larger round-off error. We can convert the alternating series into a monotonic series by adding together two adjacent elements, say $f_{2k} + f_{2k+1}$. [5] The relevant formulae can be derived from the definition of f_k using the recurrence relations for P_k, Q_k:

$$f_{2k-1} + f_{2k} = -\frac{b_0 \ldots b_{2k-2} a_{2k}}{Q_{2k-1} Q_{2k+1}},$$

$$f_{2k} + f_{2k+1} = \frac{b_0 \ldots b_{2k-1} a_{2k+1}}{Q_{2k} Q_{2k+2}}.$$

Now in the series $f_0 + (f_1 + f_2) + (f_3 + f_4) + \ldots$ the first term is positive and all subsequent terms will be negative.

Which method to use

We have just described the following methods of computing a continued fraction:

1. Bottom-up, straight division.
2. Bottom-up, separate recurrences for numerators and denominators.
3. Bottom-up, with an estimate of the remainder.
4. Top-down, with ordinary step.
5. Top-down, with two steps at once.

The bottom-up methods are simpler and faster than the top-down methods but require to know the number of terms in advance. In many cases the required number of terms can be estimated analytically, and then the bottom-up methods are always preferable. But in some cases the convergence analysis is very complicated.

The plain bottom-up method requires one long division at each step, while the bottom-up method with two recurrences requires two long multiplications at each step. Since the time needed for a long division is usually about four times that for a long multiplication (e.g. when the division is computed by Newton's method), the second variation of the bottom-up method is normally faster.

The estimate of the remainder improves the convergence of the bottom-up method and should always be used if available.

If an estimate of the number of terms is not possible, the top-down methods should be used, looping until the running error estimate shows enough precision. This incurs a performance penalty of at least 100% and at most 300%. The top-down method with two steps at once should be used only when the formula for f_k results in alternating signs.

Usefulness of continued fractions

Many mathematical functions have a representation as a continued fraction. Some systems of "exact arithmetic" use continued fractions as a primary internal representation of real numbers. This has its advantages (no round-off errors, lazy "exact" computations) and disadvantages (it is slow, especially with some operations). Here we consider the use of continued fractions with a traditional implementation of arithmetic (floating-point numbers with variable precision).

Usually, a continued fraction representation of a function will converge geometrically or slower, i.e. at least $O(P)$ terms are needed for a precision of P digits. If a geometrically convergent Taylor series representation is also available, the continued fraction method will be slower because it requires at least as many or more long multiplications per term. Also, in most cases the Taylor series can be computed much more efficiently using the

[4] This is a well-known result in the theory of continued fractions. We give an elementary derivation below.

[5] This method is used by [Thacher 1963], who refers to a suggestion by Hans Maehly.

rectangular scheme. (See, e.g., the section on $\arctan x$ for a more detailed consideration.)

However, there are some functions for which a Taylor series is not easily computable or does not converge but a continued fraction is available. For example, the incomplete Gamma function and related functions can be computed using continued fractions in some domains of their arguments.

Derivation of the formula for f_k

Here is a straightforward derivation of the formula for f_k in the top-down method. We need to compute the difference between successive approximations $F_{(0,n)}$ and $F_{(0,n+1)}$. The recurrence relation we shall use is

$$F_{(m,n+1)} - F_{(m,n)} = -\frac{b_m \left(F_{(m+1,n+1)} - F_{(m+1,n)}\right)}{F_{(m+1,n+1)} F_{(m+1,n)}}.$$

This can be shown by manipulating the two fractions, or by using the recurrence relation for $F_{(m,n)}$.

So far we have reduced the difference between $F_{(m,n+1)}$ and $F_{(m,n)}$ to a similar difference on the next level $m+1$ instead of m; i.e. we can increment m but keep n fixed. We can apply this formula to $F_{(0,n+1)} - F_{(0,n)}$, i.e. for $m = 0$, and continue applying the same recurrence relation until m reaches n. The result is

$$F_{(0,n+1)} - F_{(0,n)} = \frac{(-1)^n \, b_0 ... b_n}{F_{(1,n+1)} ... F_{(n+1,n+1)} F_{(1,n)} ... F_{(n,n)}}.$$

Now the problem is to simplify the two long products in the denominator. We notice that $F_{(1,n)}$ has $F_{(2,n)}$ in the denominator, and therefore $F_{(1,n)} F_{(2,n)} = F_{(2,n)} a_1 + b_1$. The next product is $F_{(1,n)} F_{(2,n)} F_{(3,n)}$ and it simplifies to a linear function of $F_{(3,n)}$, namely $F_{(1,n)} F_{(2,n)} F_{(3,n)} = (b_1 + a_1 a_2) F_{(3,n)} + b_1 a_2$. So we can see that there is a general formula

$$F_{(1,n)} ... F_{(k,n)} = P_k + Q_k F_{(k,n)}$$

with some coefficients P_k, Q_k which do not actually depend on n but only on the terms of the partial fraction up to k. In other words, these coefficients can be computed starting with $P_1 = 0$, $Q_1 = 1$ in the forward direction. The recurrence relations for P_k, Q_k that we have seen above in the definition of f_k follow from the identity $\left(P_k + Q_k F_{(k,n)}\right) F_{(k+1,n)} = P_{k+1} + Q_{k+1} F_{(k+1,n)}$.

Having found the coefficients P_k, Q_k, we can now rewrite the long products in the denominator, e.g.

$$F_{(1,n)} ... F_{(n,n)} = P_n + Q_n F_{(n,n)} = Q_{n+1}.$$

(We have used the recurrence relation for Q_{n+1}.) Now it follows that

$$f_{n+1} \equiv F_{(0,n+1)} - F_{(0,n)} = \frac{(-1)^n \, b_0 ... b_n}{Q_{n+1} Q_{n+2}}.$$

Thus we have converted the continued fraction into a series, i.e. $F_{(0,n)} = \sum_{k=0}^{n} f_k$ with f_k defined above.

Examples of continued fraction representations

We have already mentioned that continued fractions give a computational advantage only when other methods are not available. There exist continued fraction representations of almost all functions, but in most cases the usual methods (Taylor series, identity transformations, Newton's method and so on) are superior.

For example, the continued fraction

$$\arctan x = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{(2x)^2}{5 + \cfrac{(3x)^2}{7 + \cdots}}}}$$

converges geometrically at all x. However, the Taylor series also converges geometrically and can be computed much faster than the continued fraction.

There are some cases when a continued fraction representation is efficient. The complementary error function $\operatorname{erfc} x$ can be computed using the continued fraction due to Laplace (e.g. [Thacher 1963]),

$$\frac{\sqrt{\pi}}{2} x \exp\left(x^2\right) \operatorname{erfc} x = \cfrac{1}{1 + \cfrac{v}{1 + \cfrac{2v}{1 + \cfrac{3v}{1 + \cdots}}}},$$

where $v \equiv \left(2x^2\right)^{-1}$. This continued fraction converges for all (complex) x except pure imaginary x.

The error function is a particular case of the incomplete Gamma function

$$\Gamma(a, z) \equiv \int_z^{+\infty} x^{a-1} \exp(-x) \, dx.$$

There exists an analogous continued fraction representation due to Legendre (e.g. [Abramowitz *et al.*], 6.5.31),

$$\Gamma(a, z) = \frac{z^{a-1} \exp(-z)}{1 + \cfrac{(1-a)v}{1 + \cfrac{v}{1 + \cfrac{(2-a)v}{1 + \cfrac{2v}{1 + \cdots}}}}},$$

where $v \equiv z^{-1}$.

4.10 Estimating convergence of continued fractions

Elsewhere in this book we have used elementary considerations to find the required number of terms in a power series. It is much more difficult to estimate the convergence rate of a continued fraction. In many cases this can be done using the theory of complex variable. Below we shall consider some cases when this computation is analytically tractable.

Suppose we are given the terms a_k, b_k that define an infinite continued fraction, and we need to estimate its convergence rate. We have to find the number of terms n for which the error of approximation is less than a given ϵ. In our notation, we need to solve $|f_{n+1}| < \epsilon$ for n.

The formula that we derived for f_{n+1} gives an error estimate for the continued fraction truncated at the n-th term. But this formula contains the numbers Q_n in the denominator. The main problem is to find how quickly the sequence Q_n grows. The recurrence relation for this sequence can be rewritten as

$$Q_{n+2} = a_{n+1} Q_{n+1} + b_n Q_n,$$

for $k \geq 0$, with initial values $Q_0 = 0$ and $Q_1 = 1$. It is not always easy to get a handle on this sequence, because in most cases there is no closed-form expression for Q_n.

Simple bound on Q_n

A simple lower bound on the growth of Q_n can be obtained from the recurrence relation for Q_n. Assume that $a_k > 0$, $b_k > 0$. It

is clear that all Q_n are positive, so $Q_{n+1} \geq a_n Q_n$. Therefore Q_n grows at least as the product of all a_n:

$$Q_{n+1} \geq \prod_{i=1}^{n} a_n.$$

This result gives the following upper bound on the precision,

$$|f_{n+1}| \leq \frac{b_0 ... b_n}{(a_1 ... a_n)^2 a_{n+1}}.$$

We have used this bound to estimate the relative error of the continued fraction expansion for $\arctan x$ at small x (elsewhere in this book). However, we found that at large x this bound becomes greater than 1. This does not mean that the continued fraction does not converge and cannot be used to compute $\arctan x$ when $x > 1$, but merely indicates that the "simple bound" is too weak. The sequence Q_n actually grows faster than the product of all a_k and we need a tighter bound on this growth. In many cases such a bound can be obtained by the method of generating functions.

The method of generating functions

The idea is to find a generating function $G(s)$ of the sequence Q_n and then use an explicit form of $G(s)$ to obtain an asymptotic estimate of Q_n at large k.

The asymptotic growth of the sequence Q_n can be estimated by the method of steepest descent, also known as Laplace's method. (See, e.g., [Olver 1974], ch. 3, sec. 7.5.) This method is somewhat complicated but quite powerful. The method requires that we find an integral representation for Q_n (usually a contour integral in the complex plane). Then we can convert the integral into an asymptotic series in k^{-1}.

Along with the general presentation of the method, we shall consider an example when the convergence rate can be obtained analytically. The example is the representation of the complementary error function $\operatorname{erfc} x$,

$$\frac{\sqrt{\pi}}{2} x \exp\left(x^2\right) \operatorname{erfc} x = \frac{1}{1 + \frac{v}{1 + \frac{2v}{1 + \frac{3v}{1+\ldots}}}},$$

where $v \equiv \left(2x^2\right)^{-1}$. We shall assume that $|v| < \frac{1}{2}$ since the continued fraction representation will not be used for small x (where the Taylor series is efficient). The terms of this continued fraction are: $a_k = 1$, $b_k = kv$, for $k \geq 1$, and $a_0 = 0$, $b_0 = 1$.

The "simple bound" would give $|f_{n+1}| \leq v^n n!$ and this expression grows with n. But we know that the above continued fraction actually converges for any v, so f_{n+1} must tend to zero for large n. It seems that the "simple bound" is not strong enough for any v and we need a better bound.

An integral representation for Q_n can be obtained using the method of generating functions. Consider a function $G(s)$ defined by the infinite series

$$G(s) = \sum_{n=0}^{\infty} Q_{n+1} \frac{s^n}{n!}.$$

$G(s)$ is usually called the "generating function" of a sequence. We shifted the index to $n+1$ for convenience, since $Q_0 = 0$, so now $G(0) = 1$.

Note that the above series for the function $G(s)$ may or may not converge for any given s; we shall manipulate $G(s)$ as a formal power series until we obtain an explicit representation. What we really need is an analytic continuation of $G(s)$ to the complex s.

It is generally the case that if we know a simple linear recurrence relation for a sequence, then we can also easily find its generating function. The generating function will satisfy a linear differential equation. To guess this equation, we write down the series for $G(s)$ and its derivative $G'(s)$ and try to find their linear combination which is identically zero because of the recurrence relation. (There is, of course, a computer algebra algorithm for doing this automatically.)

Taking the derivative $G'(s)$ produces the forward-shifted series

$$G'(s) = \sum_{n=0}^{\infty} Q_{n+2} \frac{s^n}{n!}.$$

Multiplying $G(s)$ by s produces a back-shifted series with each term multiplied by n:

$$sG(s) = \sum_{n=0}^{\infty} n Q_n \frac{s^n}{n!}.$$

If the recurrence relation for Q_n contains only constants and polynomials in n, then we can easily convert that relation into a differential equation for $G(s)$. We only need to find the right combination of back- and forward-shifts and multiplications by n.

In the case of our sequence Q_n above, the recurrence relation is

$$Q_{n+2} = Q_{n+1} + vnQ_n.$$

This is equivalent to the differential equation

$$G'(s) = (1 + vs) G(s).$$

The solution with the obvious initial condition $G(0) = 1$ is

$$G(s) = \exp\left(s + \frac{vs^2}{2}\right).$$

The second step is to obtain an integral representation for Q_n, so that we could use the method of steepest descents and find its asymptotic at large n.

In our notation Q_{n+1} is equal to the n-th derivative of the generating function at $s = 0$:

$$Q_{n+1} = \frac{d^n}{ds^n} G(s = 0),$$

but it is generally not easy to estimate the growth of this derivative at large n.

There are two ways to proceed. One is to obtain an integral representation for $G(s)$, for instance

$$G(s) = \int_{-\infty}^{\infty} F(t, s)\, dt,$$

where $F(t, s)$ is some known function. Then an integral representation for Q_n will be found by differentiation. But it may be difficult to find such $F(t, s)$.

The second possibility is to express Q_n as a contour integral in the complex plane around $s = 0$ in the counter-clockwise direction:

$$Q_n = \frac{(n-1)!}{2\pi\imath} \int G(s)\, s^{-n} ds.$$

If we know the singularities and of $G(s)$, we may transform the contour of the integration into a path that is more convenient for the method of the steepest descent. This procedure is more general but requires a detailed understanding of the behavior of $G(s)$ in the complex plane.

In the particular case of the continued fraction for $\operatorname{erfc} x$, the calculations are somewhat easier if $Re(v) > 0$ (where $v \equiv \frac{1}{2x^2}$).

Full details are given in a separate section. The result for $Re\,(v) > 0$ is

$$Q_n \approx \frac{(vn)^{\frac{n}{2}}}{\sqrt{2nv}} \exp\left(\sqrt{\frac{n}{v}} - \frac{1}{4v} - \frac{n}{2}\right).$$

This, together with Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n,$$

allows to estimate the error of the continued fraction approximation:

$$f_{n+1} \approx 2\,(-1)^{n+1} \sqrt{\frac{2\pi}{v}} \exp\left(-2\sqrt{\frac{n}{v}} + \frac{1}{2v}\right).$$

Note that this is not merely a bound but an actual asymptotic estimate of f_{n+1}. (Stirling's formula can also be derived using the method of steepest descent from an integral representation of the Gamma function, in a similar way.)

Defined as above, the value of f_{n+1} is in general a complex number. The absolute value of f_{n+1} can be found using the formula

$$Re\left(\sqrt{\frac{n}{v}}\right) = \sqrt{\frac{n}{2}}\sqrt{1 + \frac{Re\,(v)}{|v|}}.$$

We obtain

$$|f_{n+1}| \approx 2\sqrt{\frac{2\pi}{|v|}} \exp\left(-\sqrt{2n}\sqrt{1 + \frac{Re\,(v)}{|v|}} + \frac{Re\,(v)}{2\,|v|^2}\right).$$

When $Re\,(v) \leq 0$, the same formula can be used (this can be shown by a more careful consideration of the branches of the square roots). The continued fraction does not converge when $Re\,(v) < 0$ and $Im\,(v) = 0$ (i.e. for pure imaginary x). This can be seen from the above formula: in that case $Re\,(v) = -|v|$ and $|f_{n+1}|$ does not decrease when n grows.

These estimates show that the error of the continued fraction approximation to $\operatorname{erfc} x$ (when it converges) decreases with n slower than in a geometric progression. This means that we need to take $O\left(P^2\right)$ terms to get P digits of precision.

Derivations for the example with $\operatorname{erfc} x$

Here we give a detailed calculation of the convergence rate of the continued fraction for $\operatorname{erfc} x$ using the method of generating functions.

A simpler approach

In our case, $G\,(s)$ is a function with a known Fourier transform and we can obtain a straightforward representation valid when $Re\,(v) > 0$,

$$Q_{n+1} = \frac{1}{\sqrt{2\pi v}} \int_{-\infty}^{\infty} (1+t)^n \exp\left(-\frac{t^2}{2v}\right) dt.$$

We shall first apply the method of steepest descent to this integral (assuming real $v > 0$ for simplicity) and then consider the more general procedure with the contour integration.

To use the method of steepest descent, we represent the integrand as an exponential of some function $g\,(t, n)$ and find "stationary points" where this function has local maxima:

$$Q_{n+1} = \frac{1}{\sqrt{2\pi v}} \int_{-\infty}^{\infty} \exp\left(g\,(t, n)\right) dt,$$

$$g\,(t, n) \equiv -\frac{t^2}{2v} + n \ln\left(1 + t\right).$$

(Note that the logarithm here must acquire an imaginary part $\imath\pi$ for $t < -1$, and we should take the real part which is equal to $\ln|1 + t|$. We shall see that the integral over negative t is negligible.) We expect that when n is large, $Re\,(g\,(t, n))$ will have a peak or several peaks at certain values of t. When t is not close to the peaks, the value of $Re\,(g\,(t, n))$ is smaller and, since g is in the exponential, the integral is dominated by the contribution of the peaks. This is the essence of the method of steepest descent on the real line.

We only need to consider very large values of n, so we can neglect terms of order $O\left(\frac{1}{\sqrt{n}}\right)$ or smaller. We find that, in our case, two peaks of $Re\,(g)$ occur at approximately $t_1 \approx -\frac{1}{2} + \sqrt{nv}$ and $t_2 \approx -\frac{1}{2} - \sqrt{nv}$. We assume that n is large enough so that $nv > \frac{1}{2}$. Then the first peak is at a positive t and the second peak is at a negative t. The contribution of the peaks can be computed from the Taylor approximation of $g\,(t, n)$ near the peaks. We can expand, for example,

$$g\,(t, n) \approx g\,(t_1, n) + \left(\frac{\partial^2}{\partial t^2} g\,(t_1, n)\right) \frac{(t - t_1)^2}{2}$$

near $t = t_1$. The values $g\,(t_1, n)$ and $\frac{\partial^2}{\partial t^2} g\,(t_1, n)$, and likewise for t_2, are constants that we already know since we know t_1 and t_2. Then the integral of $\exp(g)$ will be approximated by the integral

$$\int_{-\infty}^{\infty} \exp\left(g\,(t_1, n) + \left(\frac{\partial^2}{\partial t^2} g\,(t_1, n)\right) \frac{(t - t_1)^2}{2}\right) dt.$$

(Note that $\frac{\partial^2}{\partial t^2} g\,(t_1, n)$ is negative.) This is a Gaussian integral that can be easily evaluated, with the result

$$\exp\left(g\,(t_1, n)\right) \sqrt{-\frac{2\pi}{\frac{\partial^2}{\partial t^2} g\,(t_1, n)}}.$$

This is the leading term of the contribution of the peak at t_1; there will be a similar expression for the contribution of t_2. We find that the peak at t_1 gives a larger contribution, by the factor $\exp\left(2\sqrt{\frac{n}{v}}\right)$. This factor is never small since $n > 1$ and $v < \frac{1}{2}$. So it is safe to ignore the peak at t_2 for the purposes of our analysis.

Then we obtain the estimate

$$Q_{n+1} \approx \frac{1}{\sqrt{2}} \exp\left(\sqrt{\frac{n}{v}} - \frac{1}{4v} - \frac{n}{2}\right) (vn)^{\frac{n}{2}}.$$

The contour integral approach

In many cases it is impossible to compute the Fourier transform of the generating function $G\,(s)$. Then one can use the contour integral approach. One should represent the integrand as

$$G\,(s)\, s^{-n} = \exp\left(g\,(s)\right)$$

where

$$g\,(s) \equiv \ln G\,(s) - n \ln s,$$

and look for stationary points of $g\,(s)$ in the complex plane ($0 = 0$). The original contour is around the pole $s = 0$ in the counterclockwise direction. We need to deform that contour so that the new contour passes through the stationary points. The contour should cross each stationary point in a certain direction in the complex plane. The direction is chosen to make the stationary point the sharpest local maximum of $Re\,(g\,(s))$ on the contour.

Usually one of the stationary points has the largest value of $Re\,(g\,(s))$; this is the dominant stationary point. If s_0 is the dominant stationary point and $g_2 = \frac{d^2}{ds^2} g\,(s_0)$ is the second

derivative of g at that point, then the asymptotic of the integral is

$$\frac{1}{2\pi}\int \exp\left(g\left(s\right)\right)ds = \frac{1}{\sqrt{2\pi\left|g_2\right|}}\exp\left(g\left(s_0\right)\right).$$

(The integral will have a negative sign if the contour crosses the point s_0 in the negative imaginary direction.)

We have to choose a new contour and check the convergence of the resulting integral separately. In each case we may need to isolate the singularities of $G\left(s\right)$ or to find the regions of infinity where $G\left(s\right)$ quickly decays (so that the infinite parts of the contour might be moved there). There is no prescription that works for all functions $G\left(s\right)$.

Let us return to our example. For $G\left(s\right) = \exp\left(s + \frac{vs^2}{2}\right)$, the function $g\left(s\right)$ has no singularities except the pole at $s = 0$. There are two stationary points located at the (complex) roots s_1, s_2 of the quadratic equation $vs^2 + s - n = 0$. Note that v is an arbitrary (nonzero) complex number. We now need to find which of the two stationary points gives the dominant contribution. By comparing $Re\left(g\left(s_1\right)\right)$ and $Re\left(g\left(s_2\right)\right)$ we find that the point s with the largest real part gives the dominant contribution. However, if $Re\left(s_1\right) = Re\left(s_2\right)$ (this happens only if v is real and $v < 0$, i.e. if x is pure imaginary), then both stationary points contribute equally. Barring that possibility, we find (with the usual definition of the complex square root) that the dominant contribution for large n is from the stationary point at

$$s_1 = \frac{\sqrt{4nv+1}-1}{2v}.$$

The second derivative of $g\left(s\right)$ at the stationary point is approximately $2v$. The contour of integration can be deformed into a line passing through the dominant stationary point in the positive imaginary direction. Then the leading asymptotic is found using the Gaussian approximation (assuming $Re\left(v\right) > 0$):

$$Q_n = \frac{(n-1)!v^{\frac{n}{2}}}{\sqrt{4\pi v}}\exp\left(\frac{n\left(1-\ln n\right)}{2} + \sqrt{\frac{n}{v}} - \frac{1}{4v}\right).$$

This formula agrees with the asymptotic for Q_{n+1} found above for real $v > 0$, when we use Stirling's formula for $(n-1)!$:

$$(n-1)! = \sqrt{2\pi}e^{-n}n^{n-\frac{1}{2}}.$$

The treatment for $Re\left(v\right) < 0$ is similar.

4.11 Newton's method and its improvements

Newton's method (also called the Newton-Raphson method) of numerical solution of algebraic equations and its generalizations can be used to obtain multiple-precision values of several elementary functions.

Newton's method

The basic formula is widely known: If $f\left(x\right) = 0$ must be solved, one starts with a value of x that is close to some root and iterates

$$x' = x - f\left(x\right)\left(\frac{d}{dx}f\left(x\right)\right)^{-1}.$$

This formula is based on the approximation of the function $f\left(x\right)$ by a tangent line at some point x. A Taylor expansion in the neighborhood of the root shows that (for an initial value x_0 sufficiently close to the root) each iteration gives at least twice as many correct digits of the root as the previous one ("quadratic convergence"). Therefore the complexity of this algorithm is proportional to a logarithm of the required precision and to the time it takes to evaluate the function and its derivative. Generalizations of this method require computation of higher derivatives of the function $f\left(x\right)$ but successive approximations to the root converge several times faster (the complexity is still logarithmic).

Newton's method sometimes suffers from a sensitivity to the initial guess. If the initial value x_0 is not chosen sufficiently close to the root, the iterations may converge very slowly or not converge at all. To remedy this, one can combine Newton's iteration with simple bisection. Once the root is bracketed inside an interval $(a,\ b)$, one checks whether $\frac{a+b}{2}$ is a better approximation for the root than that obtained from Newton's iteration. This guarantees at least linear convergence in the worst case.

For some equations $f\left(x\right) = 0$, Newton's method converges faster than quadratically. For example, solving $\sin x = 0$ in the neighborhood of $x = 3.14159$ gives "cubic" convergence, i.e. the number of correct digits is tripled at each step. This happens because $\sin x$ near its root $x = \pi$ has a vanishing second derivative and thus the function is particularly well approximated by a straight line.

Halley's method

Halley's method is an improvement over Newton's method that makes each equation well approximated by a straight line near the root. Edmund Halley computed fractional powers, $x = \sqrt[n]{a}$, by the iteration

$$x' = x\frac{n\left(a+x^n\right)+a-x^n}{n\left(a+x^n\right)-\left(a-x^n\right)}.$$

This formula is equivalent to Newton's method applied to the equation $x^{n-q} = ax^{-q}$ with $q = \frac{n-1}{2}$. This iteration has a cubic convergence rate. This is the fastest method to compute n-th roots ($n \geq 3$) with multiple precision. Iterations with higher order of convergence, for example, the method with quintic convergence rate

$$x' = x\frac{\frac{n-1}{n+1}\frac{2n-1}{2n+1}x^{2n} + 2\frac{2n-1}{n+1}x^n a + a^2}{x^{2n} + 2\frac{2n-1}{n+1}x^n a + \frac{n-1}{n+1}\frac{2n-1}{2n+1}a^2},$$

require more arithmetic operations per step and are in fact less efficient at high precision.

Halley's method can be generalized to any function $f\left(x\right)$. A cubically convergent iteration is always obtained if we replace the equation $f\left(x\right) = 0$ by an equivalent equation

$$g\left(x\right) \equiv \frac{f\left(x\right)}{\sqrt{\left|\frac{d}{dx}f\left(x\right)\right|}} = 0$$

and use the standard Newton's method on it. Here the function $g\left(x\right)$ is chosen so that its second derivative vanishes ($\frac{d^2}{dx^2}g\left(x\right) = 0$) at the root of the equation $f\left(x\right) = 0$, independently of where this root is. For example, the equation $\exp\left(x\right) = a$ is transformed into $g\left(x\right) \equiv \exp\left(\frac{x}{2}\right) - a\exp\left(-\frac{x}{2}\right) = 0$. (There is no unique choice of the function $g\left(x\right)$ and sometimes another choice will make the iteration more quickly computable.)

The Halley iteration for the equation $f\left(x\right) = 0$ can be written as

$$x' = x - \frac{2f\left(x\right)\left(\frac{d}{dx}f\left(x\right)\right)}{2\left(\frac{d}{dx}f\left(x\right)\right)^2 - f\left(x\right)\left(\frac{d^2}{dx^2}f\left(x\right)\right)}.$$

Halley's iteration, despite its faster convergence rate, may be more cumbersome to evaluate than Newton's iteration and so it may not provide a more efficient numerical method for a given

function. Only in some special cases is Halley's iteration just as simple to compute as Newton's iteration.

Halley's method is sometimes less sensitive to the choice of the initial point x_0. An extreme example of sensitivity to the initial point is the equation $x^{-2} = 12$ for which Newton's iteration $x' = \frac{3}{2}x - 6x^3$ converges to the root only from initial points $0 < x_0 < 0.5$ and wildly diverges otherwise, while Halley's iteration converges to the root from any $x_0 > 0$.

It is at any rate not true that Halley's method always converges better than Newton's method. For instance, it diverges on the equation $2\cos x = x$ unless started at x_0 within the interval $(-\frac{1}{6}\pi, \frac{7}{6}\pi)$. Another example is the equation $\ln x = a$. This equation allows to compute $x = \exp(a)$ if a fast method for computing $\ln x$ is available (e.g. the AGM-based method). For this equation, Newton's iteration

$$x' = x\left(1 + a - \ln x\right)$$

converges for any $0 < x < \exp(a+1)$, while Halley's iteration converges only if $\exp(a-2) < x < \exp(a+2)$.

When it converges, Halley's iteration can still converge very slowly for certain functions $f(x)$, for example, for $f(x) = x^n - a$ if $n^n > a$. For such functions that have very large and rapidly changing derivatives, no general method can converge faster than linearly. In other words, a simple bisection will generally do just as well as any sophisticated iteration, until the root is approximated very precisely. Halley's iteration combined with bisection seems to be a good choice for such problems.

When to use Halley's method

Despite its faster convergence, Halley's iteration frequently gives no advantage over Newton's method. To obtain P digits of the result, we need about $\frac{\ln P}{\ln 2}$ iterations of a quadratically convergent method and about $\frac{\ln P}{\ln 3}$ iterations of a cubically convergent method. So the cubically convergent iteration is faster only if the time taken by cubic one iteration is less than about $\frac{\ln 3}{\ln 2} \approx 1.6$ of the time of one quadratic iteration.

Higher-order schemes

Sometimes it is easy to generalize Newton's iteration to higher-order schemes. There are general formulae such as Shroeder's and Householder's iterations. We shall give some examples where the construction is very straightforward. In all examples x is the initial approximation and the next approximation is obtained by truncating the given series.

1. Inverse $\frac{1}{a}$. Set $y = 1 - ax$, then

$$\frac{1}{a} = \frac{x}{1-y} = x\left(1 + y + y^2 + ...\right).$$

2. Square root $\sqrt{a}$. Set $y = 1 - ax^2$, then

$$\sqrt{a} = \frac{\sqrt{1-y}}{x} = \frac{1}{x}\left(1 - \frac{1}{2}y - \frac{1}{8}y^2 - ...\right).$$

3. Inverse square root $\frac{1}{\sqrt{a}}$. Set $y = 1 - ax^2$, then

$$\frac{1}{\sqrt{a}} = \frac{x}{\sqrt{1-y}} = x\left(1 + \frac{1}{2}y + \frac{3}{8}y^2 + ...\right).$$

4. n-th root $\sqrt[n]{a}$. Set $y = 1 - ax^n$, then

$$\sqrt[n]{a} = \frac{\sqrt[n]{1-y}}{x} = \frac{1}{x}\left(1 - \frac{1}{n}y - \frac{n-1}{2n^2}y^2 - ...\right).$$

5. Exponential $\exp(a)$. Set $y = a - \ln x$, then

$$\exp(a) = x\exp(y) = x\left(1 + y + \frac{y^2}{2!} + \frac{y^3}{3!} + ...\right).$$

6. Logarithm $\ln a$. Set $y = 1 - a\exp(-x)$, then

$$\ln a = x + \ln(1-y) = x - y - \frac{y^2}{2} - \frac{y^3}{3} - ....$$

In the above examples, y is a small quantity and the series represents corrections to the initial value x, therefore the order of convergence is equal to the first discarded order of y in the series.

These simple constructions are possible because the functions satisfy simple identities, such as $\exp(a+b) = \exp(a)\exp(b)$ or $\sqrt{ab} = \sqrt{a}\sqrt{b}$. For other functions the formulae quickly become very complicated and unsuitable for practical computations.

Precision control

Newton's method and its generalizations are particularly convenient for multiple precision calculations because of their insensitivity to accumulated errors: if at some iteration x_k is found with a small error, the error will be corrected at the next iteration. Therefore it is not necessary to compute all iterations with the full required precision; each iteration needs to be performed at the precision of the root expected from that iteration (plus a few more digits to allow for round-off error). For example, if we know that the initial approximation is accurate to 3 digits, then (assuming quadratic convergence) [6] it is enough to perform the first iteration to 6 digits, the second iteration to 12 digits and so on. In this way, multiple-precision calculations are enormously speeded up.

For practical evaluation, iterations must be supplemented with "quality control". For example, if x_0 and x_1 are two consecutive approximations that are already very close, we can quickly compute the achieved (relative) precision by finding the number of leading zeros in the number

$$\frac{|x_0 - x_1|}{\max(x_0, x_1)}.$$

This is easily done using the bit count function. After performing a small number of initial iterations at low precision, we can make sure that x_1 has at least a certain number of correct digits of the root. Then we know which precision to use for the next iteration (e.g. triple precision if we are using a cubically convergent scheme). It is important to perform each iteration at the precision of the root which it will give and not at a higher precision; this saves a great deal of time since all calculations are very slow at high precision.

Fine-tuning the working precision

To reduce the computation time, it is important to write the iteration formula with explicit separation of higher-order quantities. For example, Newton's iteration for the inverse square root $\frac{1}{\sqrt{a}}$ can be written either as

$$x' = x\frac{3 - ax^2}{2}$$

[6] This disregards the possibility that the convergence might be slightly slower. For example, when the precision at one iteration is n digits, it might be $2n-10$ digits at the next iteration. In these (fringe) cases, the initial approximation must be already precise enough (e.g. to at least 10 digits in this example).

or equivalently as

$$x' = x + x\frac{1-ax^2}{2}.$$

At first sight the first formula seems simpler because it saves one long addition. However, the second formula can be computed significantly faster than the first one, if we are willing to exercise a somewhat more fine-grained control of the working precision.

Suppose x is an approximation that is correct to P digits; then we expect the quantity x' to be correct to $2P$ digits. Therefore we should perform calculations in the first formula with $2P$ digits; this means three long multiplications, $3M(2P)$. Now consider the calculation in the second formula. First, the quantity $y \equiv 1-ax^2$ is computed using two $2P$-digit multiplications. [7] Now, the number y is small and has only P nonzero digits. Therefore the third multiplication xy costs only $M(P)$, not $M(2P)$. This is a significant time savings, especially with slower multiplication. The total cost is now $2M(2P) + M(P)$.

The advantage is even greater with higher-order methods. For example, a fourth-order iteration for the inverse square root can be written as

$$x' = x + \frac{1}{2}xy + \frac{3}{8}xy^2 + \frac{5}{16}xy^3,$$

where $y \equiv 1-ax^2$. Suppose x is an approximation that is correct to P digits; we expect $4P$ correct digits in x'. We need two long multiplications in precision $4P$ to compute y, then $M(3P)$ to compute xy, $M(2P)$ to compute xy^2, and $M(P)$ to compute xy^3. The total cost is $2M(4P) + M(3P) + M(2P) + M(P)$.

The asymptotic cost of finding the root x of the equation $f(x) = 0$ with P digits of precision is usually the same as the cost of computing $f(x)$ [Brent 1975]. The main argument can be summarized by the following simple example. To get the result to P digits, we need $O(\ln P)$ Newton's iterations. At each iteration we shall have to compute the function $f(x)$ to a certain number of digits. Suppose that we start with one correct digit and that each iteration costs us $cM(2P)$ operations where c is a given constant, while the number of correct digits grows from P to $2P$. Then the total cost of k iterations is

$$cM(2) + cM(4) + cM(8) + ... + cM\left(2^k\right).$$

If the function $M(P)$ grows linearly with $P = 2^k$, then we can estimate this sum roughly as $2cM(P)$; if $M(P) = O\left(P^2\right)$ then the result is about $\frac{4}{3}cM(P)$. It is easy to see that when $M(P)$ is some power of P that grows faster than linear, the sum is not larger than a small multiple of $M(P)$.

Thus, if we have a fast method of computing, say, $\arctan x$, then we immediately obtain a method of computing $\tan x$ which is asymptotically as fast (up to a constant).

Choosing the optimal order

Suppose we need to compute a function by Newton's method to precision P. We can sometimes find iterations of any order of convergence. For example, a k-th order iteration for the reciprocal $\frac{1}{a}$ is

$$x' = x + xy + xy^2 + ... + xy^{k-1},$$

where $y \equiv 1 - ax$. The cost of one iteration with final precision P is

$$C(k,P) \equiv M\left(\frac{P}{k}\right) + M\left(\frac{2P}{k}\right) + M\left(\frac{3P}{k}\right) + ... + cM(P).$$

[7] In fact, both multiplications are a little shorter, since x is a number with only P correct digits; we can compute ax and then ax^2 as products of a $2P$-digit number and a P-digit number, with a $2P$-digit result. We ignore this small difference.

(Here the constant $c \equiv 1$ is introduced for later convenience. It denotes the number of multiplications needed to compute y.)

Increasing the order by 1 costs us comparatively little, and we may change the order k at any time. Is there a particular order k that gives the smallest computational cost and should be used for all iterations, or the order needs to be adjusted during the computation? A natural question is to find the optimal computational strategy.

It is difficult to fully analyze this question, but it seems that choosing a particular order k for all iterations is close to the optimal strategy.

A general "strategy" is a set of orders $S(P, P_0)$=(k_1, k_2, ..., k_n) to be chosen at the first, second, ..., n-th iteration, given the initial precision P_0 and the required final precision P. At each iteration, the precision will be multiplied by the factor k_i. The optimal strategy $S(P, P_0)$ is a certain function of P_0 and P such that the required precision is reached, i.e.

$$P_0 k_1 ... k_n = P,$$

and the cost

$$C(k_1, P_0 k_1) + C(k_2, P_0 k_1 k_2) + ... + C(k_n, P)$$

is minimized.

If we assume that the cost of multiplication $M(P)$ is proportional to some power of P, for instance $M(P) = P^\mu$, then the cost of each iteration and the total cost are homogeneous functions of P and P_0. Therefore the optimal strategy is a function only of the ratio $\frac{P}{P_0}$. We can multiply both P_0 and P by a constant factor and the optimal strategy will remain the same. We can denote the optimal strategy $S\left(\frac{P}{P_0}\right)$.

We can check whether it is better to use several iterations at smaller orders instead of one iteration at a large order. Suppose that $M(P) = P^\mu$, the initial precision is 1 digit, and the final precision $P = k^n$. We can use either n iterations of the order k or 1 iteration of the order P. The cost of one iteration of order P at target precision P is $C(P, P)$, whereas the total cost of n iterations of order k is

$$C(k,k) + C\left(k,k^2\right) + ... + C\left(k,k^n\right).$$

With $C(k, P)$ defined above, we can take approximately

$$C(k,p) \approx p^\mu\left(c - 1 + \frac{k}{\mu+1}\right).$$

Then the cost of one P-th order iteration is

$$P^\mu\left(c - 1 + \frac{P}{\mu+1}\right),$$

while the cost of n iterations of the order k is clearly smaller since $k < P$,

$$P^\mu\left(c - 1 + \frac{k}{\mu+1}\right)\frac{k^\mu}{k^\mu - 1}.$$

At fixed P, the best value of k is found by minimizing this function. For $c = 1$ (reciprocal) we find $k = \sqrt[\mu]{1+\mu}$ which is never above 2. This suggests that $k = 2$ is the best value for finding the reciprocal $\frac{1}{a}$. However, larger values of c can give larger values of k. The equation for the optimal value of k is

$$\frac{k^{\mu+1}}{\mu+1} - k = \mu(c-1).$$

So far we have only considered strategies that use the same order k for all iterations, and we have not yet shown that such strategies are the best ones. We now give a plausible argument (not quite a rigorous proof) to justify this claim.

Consider the optimal strategy $S\left(P^2\right)$ for the initial precision 1 and the final precision P^2, when P is very large. Since it is better to use several iterations at lower orders, we may assume that the strategy $S\left(P^2\right)$ contains many iterations and that one of these iterations reaches precision P. Then the strategy $S\left(P^2\right)$ is equivalent to a sequence of the two optimal strategies to go from 1 to P and from P to P^2. However, both strategies must be the same because the optimal strategy only depends on the ratio of precisions. Therefore, the optimal strategy $S\left(P^2\right)$ is a sequence of two identical strategies $(S(P), S(P))$.

Suppose that k_1 is the first element of $S(P)$. The optimal strategy to go from precision k_1 to precision Pk_1 is also $S(P)$. Therefore the second element of $S(P)$ is also equal to k_1, and by extension all elements of $S(P)$ are the same.

A similar consideration gives the optimal strategy for other iterations that compute inverses of analytic functions, such as Newton's iteration for the inverse square root or for higher roots. The difference is that the value of c should be chosen as the equivalent number of multiplications needed to compute the function. For instance, $c = 1$ for division and $c = 2$ for the inverse square root iteration.

The conclusion is that in each case we should compute the optimal order k in advance and use this order for all iterations.

4.12 Fast evaluation of Taylor series

Taylor series for analytic functions is a common method of evaluation.

Method 1: simple summation

If we do not know the required number of terms in advance, we cannot do any better than just evaluate each term and check if it is already small enough. Take, for example, the series for $\exp(x)$. To straightforwardly evaluate

$$\exp(x) \approx \sum_{k=0}^{N-1} \frac{x^k}{k!}$$

with P decimal digits of precision and $x < 2$, one would need about $N \approx P\frac{\ln 10}{\ln P}$ terms of the series.

Divisions by large integers $k!$ and separate evaluations of powers x^k are avoided if we store the previous term. The next term can be obtained by a short division of the previous term by k and a long multiplication by x. Then we only need $O(N)$ long multiplications to evaluate the series. Usually the required number of terms $N = O(P)$, so the total cost is $O(PM(P))$.

There is no accumulation of round-off error in this method if x is small enough (in the case of $\exp(x)$, a sufficient condition is $|x| < \frac{1}{2}$). To see this, suppose that x is known to P digits (with relative error 10^{-P}). Since $|x| < \frac{1}{2}$, the n-th term $\frac{x^n}{n!} < 4^{-n}$ (this is a rough estimate but it is enough). Since each multiplication by x results in adding 1 significant bit of relative round-off error, the relative error of $\frac{x^n}{n!}$ is about 2^n times the relative error of x, i.e. $2^n \cdot 10^{-P}$. So the absolute round-off error of $\frac{x^n}{n!}$ is not larger than

$$\Delta < 4^{-n} \cdot 2^n \cdot 10^{-P} = 2^{-n} \cdot 10^{-P}.$$

Therefore all terms with $n > 1$ contribute less than 10^{-P} of absolute round-off error, i.e. less than was originally present in x.

In practice, one could truncate the precision of $\frac{x^n}{n!}$ as n grows, leaving a few guard bits each time to keep the round-off error negligibly small and yet to gain some computation speed. This however does not change the asymptotic complexity of the method—it remains $O(PM(P))$.

However, if x is a small rational number, then the multiplication by x is short and takes $O(P)$ operations. In that case, the total complexity of the method is $O\left(P^2\right)$ which is always faster than $O(PM(P))$.

Method 2: Horner's scheme

Horner's scheme is widely known and consists of the following rearrangement,

$$\sum_{k=0}^{N-1} a_k x^k = a_0 + x\left(a_1 + x\left(a_2 + x\left(\ldots + xa_{N-1}\right)\right)\right)$$

The calculation is started from the last coefficient a_{N-1} toward the first. If x is small, then the round-off error generated during the summation is constantly being multiplied by a small number x and thus is always insignificant. Even if x is not small or if the coefficients of the series are not small, Horner's scheme usually results in a smaller round-off error than the simple summation.

If the coefficients a_k are related by a simple ratio, then Horner's scheme may be modified to simplify the calculations. For example, the Horner scheme for the Taylor series for $\exp(x)$ may be written as

$$\sum_{k=0}^{N-1} \frac{x^k}{k!} = 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(\ldots + \frac{x}{N-1}\right)\right)\right).$$

This avoids computing the factorial function.

Similarly to the simple summation method, the working precision for Horner's scheme may be adjusted to reduce the computation time: for example, xa_{N-1} needs to be computed with just a few significant digits if x is small. This does not change the asymptotic complexity of the method: it requires $O(N) = O(P)$ long multiplications by x, so for general real x the complexity is again $O(PM(P))$. However, if x is a small rational number, then the multiplication by x is short and takes $O(P)$ operations. In that case, the total complexity of the method is $O\left(P^2\right)$.

Method 3: "rectangular" or "baby step/giant step"

We can organize the calculation much more efficiently if we are able to estimate the necessary number of terms and to afford some storage (see [Smith 1989]).

The "rectangular" algorithm uses $2\sqrt{N}$ long multiplications (assuming that the coefficients of the series are short rational numbers) and $\sqrt{N}$ units of storage. For high-precision floating-point x, this method provides a significant advantage over Horner's scheme.

Suppose we need to evaluate $\sum_{k=0}^{N} a_k x^k$ and we know the number of terms N in advance. Suppose also that the coefficients a_k are rational numbers with small numerators and denominators, so a multiplication $a_k x$ is not a long multiplication (usually, either a_k or the ratio a_k/a_{k-1} is a short rational number). Then we can organize the calculation in a rectangular array with c columns and r rows like this,

$$a_0 + a_r x^r + \ldots + a_{(c-1)r} x^{(c-1)r} +$$

$$x\left(a_1 + a_{r+1} x^r + \ldots + a_{(c-1)r+1} x^{(c-1)r}\right) +$$

$$...+$$

$$x^{r-1}\left(a_{r-1}+a_{2r+1}x^r+...\right).$$

To evaluate this rectangle, we first compute x^r (which, if done by the fast binary algorithm, requires $O(\ln r)$ long multiplications). Then we compute the $c-1$ successive powers of x^r, namely x^{2r}, x^{3r}, ..., $x^{(c-1)r}$ in $c-1$ long multiplications. The partial sums in the r rows are evaluated column by column as more powers of x^r become available. This requires storage of r intermediate results but no more long multiplications by x. If a simple formula relating the coefficients a_k and a_{k-1} is available, then a whole column can be computed and added to the accumulated row values using only short operations, e.g. $a_{r+1}x^r$ can be computed from $a_r x^r$ (note that each column contains some consecutive terms of the series). Otherwise, we would need to multiply each coefficient a_k separately by the power of x; if the coefficients a_k are short numbers, this is also a short operation. After this, we need $r-1$ more multiplications for the vertical summation of rows (using the Horner scheme). We have potentially saved time because we do not need to evaluate powers such as x^{r+1} separately, so we do not have to multiply x by itself quite so many times.

The total required number of long multiplications is $r+c+\ln r-2$. The minimum number of multiplications, given that $rc \geq N$, is around $2\sqrt{N}$ at $r \approx \sqrt{N}-\frac{1}{2}$. Therefore, by arranging the Taylor series in a rectangle with sides r and c, we obtain an algorithm which costs $O\left(\sqrt{N}\right)$ instead of $O(N)$ long multiplications and requires $\sqrt{N}$ units of storage.

One might wonder if we should not try to arrange the Taylor series in a cube or another multidimensional matrix instead of a rectangle. However, calculations show that this does not save time: the optimal arrangement is the two-dimensional rectangle.

The rectangular method saves the number of long multiplications by x but increases the number of short multiplications and additions. If x is a small integer or a small rational number, multiplications by x are fast and it does not make sense to use the rectangular method. Direct evaluation schemes are more efficient in that case.

Truncating the working precision

At the k-th step of the rectangular method, we are evaluating the k-th column with terms containing x^{rk}. Since a power series in x is normally used at small x, the number x^{rk} is typically much smaller than 1. This number is to be multiplied by some a_i and added to the previously computed part of each row, which is not small. Therefore we do not need all P floating-point digits of the number x^{rk}, and the precision with which we obtain it can be gradually decreased from column to column. For example, if $x^r < 10^{-M}$, then we only need $P-kM$ decimal digits of x^{rk} when evaluating the k-th column. (This assumes that the coefficients a_i do not grow, which is the case for most of the practically useful series.)

Reducing the working precision saves some computation time. (We also need to estimate M but this can usually be done quickly by bit counting.) Instead of $O\left(\sqrt{P}\right)$ long multiplications at precision P, we now need one long multiplication at precision P, another long multiplication at precision $P-M$, and so on. This technique will not change the asymptotic complexity which remains $O\left(\sqrt{P}M(P)\right)$, but it will reduce the constant factor in front of the O.

Like the previous two methods, there is no accumulated round-off error if x is small.

Which method to use

There are two cases: first, the argument x is a small integer or rational number with very few digits and the result needs to be found as a floating-point number with P digits; second, the argument x itself is a floating-point number with P digits.

In the first case, it is better to use either Horner's scheme (for small P, slow multiplication) or the binary splitting technique (for large P, fast multiplication). The rectangular method is actually *slower* than Horner's scheme if x and the coefficients a_k are small rational numbers. In the second case (when x is a floating-point number), it is better to use the "rectangular" algorithm.

In both cases we need to know the number of terms in advance, as we will have to repeat the whole calculation if a few more terms are needed. The simple summation method rarely gives an advantage over Horner's scheme, because it is almost always the case that one can easily compute the number of terms required for any target precision.

Note that if the argument x is not small, round-off error will become significant and needs to be considered separately for a given series.

Speed-up for some functions

An additional speed-up is possible if the function allows a transformation that reduces x and makes the Taylor series converge faster. For example, $\ln x = 2\ln\sqrt{x}$, $\cos 2x = 2(\cos x)^2 - 1$ (bisection), and $\sin 3x = 3\sin x - 4(\sin x)^3$ (trisection) are such transformations. It may be worthwhile to perform a number of such transformations before evaluating the Taylor series, if the time saved by its quicker convergence is more than the time needed to perform the transformations. The optimal number of transformations can be estimated. Using this technique in principle reduces the cost of Taylor series from $O\left(\sqrt{N}\right)$ to $O\left(\sqrt[3]{N}\right)$ long multiplications. However, additional round-off error may be introduced by this procedure for some x.

For example, consider the Taylor series for $\sin x$,

$$\sin x \approx \sum_{k=0}^{N-1} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

It is sufficient to be able to evaluate $\sin x$ for $0 < x < \frac{\pi}{2}$. Suppose we perform l steps of the trisection and then use the Taylor series with the rectangular method. Each step of the trisection needs two long multiplications. The value of x after l trisection steps becomes much smaller, $x' = x \cdot 3^{-l}$. For this x', the required number of terms in the Taylor series for P decimal digits of precision is

$$N \approx \frac{P\ln 10}{2(\ln P - \ln x')} - 1.$$

The number of long multiplications in the rectangular method is $2\sqrt{N}$. The total number of long multiplications, as a function of l, has its minimum at

$$l \approx \sqrt[3]{32\frac{\ln 10}{\ln 3}P} - \frac{\ln P - \ln x}{\ln 3},$$

where it has a value roughly proportional to $\sqrt[3]{P}$. Therefore we shall minimize the total number of long multiplications if we first perform l steps of trisection and then use the rectangular method to compute N terms of the Taylor series.

4.13 Using asymptotic series for calculations

Several important analytic functions have asymptotic series expansions. For example, the complementary error function $\operatorname{erfc} x$ and Euler's Gamma function $\Gamma(x)$ have the following asymptotic expansions at large (positive) x:

$$\operatorname{erfc} x = \frac{e^{-x^2}}{x\sqrt{\pi}}\left(1 - \frac{1}{2x^2} + ... + \frac{(2n-1)!!}{(-2x^2)^n} + ...\right),$$

$$\ln\Gamma(x) = \left(x - \frac{1}{2}\right)\ln x - x + \frac{\ln 2\pi}{2} + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n(2n-1)x^{2n-1}}$$

(here B_k are Bernoulli numbers).

The above series expansions are asymptotic in the following sense: if we truncate the series and then take the limit of very large x, then the difference between the two sides of the equation goes to zero.

It is important that the series be first truncated and then the limit of large x be taken. Usually, an asymptotic series, if taken as an infinite series, does not actually converge for any finite x. This can be seen in the examples above. For instance, in the asymptotic series for $\operatorname{erfc} x$ the n-th term has $(2n-1)!!$ in the numerator which grows faster than the n-th power of any number. The terms of the series decrease at first but then eventually start to grow, even if we select a large value of x.

The way to use an asymptotic series for a numerical calculation is to truncate the series *well before* the terms start to grow.

Error estimates of the asymptotic series are sometimes difficult, but the rule of thumb seems to be that the error of the approximation is usually not greater than the first discarded term of the series. This can be understood intuitively as follows. Suppose we truncate the asymptotic series at a point where the terms still decrease, safely before they start to grow. For example, let the terms around the 100-th term be A_{100}, A_{101}, A_{102}, ..., each of these numbers being significantly smaller than the previous one, and suppose we retain A_{100} but drop the terms after it. Then our approximation would have been a lot better if we retained A_{101} as well. (This step of the argument is really an assumption about the behavior of the series; it seems that this assumption is correct in many practically important cases.) Therefore the error of the approximation is approximately equal to A_{101}.

The inherent limitation of the method of asymptotic series is that for any given x, there will be a certain place in the series where the term has the minimum absolute value (after that, the series is unusable), and the error of the approximation cannot be smaller than that term.

For example, take the above asymptotic series for $\operatorname{erfc} x$. The logarithm of the absolute value of the n-th term can be estimated using Stirling's formula for the factorial as

$$\ln\frac{(2n-1)!!}{(2x^2)^n} \approx n(\ln n - 1 - 2\ln x).$$

This function of n has its minimum at $n = x^2$ where it is equal to $-x^2$. Therefore the best we can do with this series is to truncate it before this term. The resulting approximation to $\operatorname{erfc} x$ will have relative precision of order $\exp\left(-x^2\right)$. Suppose that x is large and we need to compute $\operatorname{erfc} x$ with P decimal digits of floating point. Then it follows that we can use the asymptotic series only if $x > \sqrt{P\ln 10}$.

We find that for a given finite x, no matter how large, there is a maximum precision that can be achieved with the asymptotic series; if we need more precision, we have to use a different method.

However, sometimes the function we are evaluating allows identity transformations that relate $f(x)$ to $f(y)$ with $y > x$. For example, the Gamma function satisfies $x\Gamma(x) = \Gamma(x+1)$. In this case we can transform the function so that we would need to evaluate it at large enough x for the asymptotic series to give us enough precision.

4.14 The AGM sequence algorithms

Several algorithms are based on the arithmetic-geometric mean (AGM) sequence. If one takes two numbers a, b and computes their arithmetic mean $\frac{a+b}{2}$ and their geometric mean $\sqrt{ab}$, then one finds that the two means are generally much closer to each other than the original numbers. Repeating this process creates a rapidly converging sequence of pairs.

More formally, one can define the function of two arguments $\operatorname{AGM}(x, y)$ as the limit of the sequence a_k where $a_{k+1} = \frac{1}{2}(a_k + b_k)$, $b_{k+1} = \sqrt{a_k b_k}$, and the initial values are $a_0 = x$, $b_0 = y$. (The limit of the sequence b_k is the same.) This function is obviously linear, $\operatorname{AGM}(cx, cy) = c\operatorname{AGM}(x, y)$, so in principle it is enough to compute $\operatorname{AGM}(1, x)$ or arbitrarily select c for convenience.

Gauss and Legendre knew that the limit of the AGM sequence is related to the complete elliptic integral,

$$\frac{\pi}{2}\frac{1}{\operatorname{AGM}\left(a, \sqrt{a^2 - b^2}\right)} = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{a^2 - b^2(\sin x)^2}} dx.$$

(Here $0 < b < a$.) This integral can be rearranged to provide some other useful functions. For example, with suitable parameters a and b, this integral is equal to π. Thus, one obtains a fast method of computing π (the Brent-Salamin method).

The AGM sequence is also defined for complex values a, b. One needs to take a square root $\sqrt{ab}$, which requires a branch cut to be well-defined. Selecting the natural cut along the negative real semiaxis ($Re(x) < 0$, $Im(x) = 0$), we obtain an AGM sequence that converges for any initial values x, y with positive real part.

Let us estimate the convergence rate of the AGM sequence starting from x, y, following the paper [Brent 1975]. Clearly the worst case is when the numbers x and y are very different (one is much larger than another). In this case the numbers a_k, b_k become approximately equal after about $k = \frac{1}{\ln 2}\ln\left|\ln\frac{x}{y}\right|$ iterations (note: Brent's paper online mistypes this as $\frac{1}{\ln 2}\left|\ln\frac{x}{y}\right|$). This is easy to see: if x is much larger than y, then at each step the ratio $r \equiv \frac{x}{y}$ is transformed into $r' = \frac{1}{2}\sqrt{r}$. When the two numbers become roughly equal to each other, one needs about $\frac{\ln n}{\ln 2}$ more iterations to make the first n (decimal) digits of a_k and b_k coincide, because the relative error $\epsilon = 1 - \frac{b}{a}$ decays approximately as $\epsilon_k \approx \frac{1}{8}\exp\left(-2^k\right)$.

Unlike Newton's iteration, the AGM sequence does not correct errors, so all numbers need to be computed with full precision. Actually, slightly more precision is needed to compensate for accumulated round-off error. Brent (in [Brent 1975]) says that $O(\ln\ln n)$ bits of accuracy are lost to round-off error if there are total of n iterations.

The AGM sequence can be used for fast computations of π, $\ln x$ and $\arctan x$. However, currently the limitations of Yacas

internal math make these methods less efficient than simpler methods based on Taylor series and Newton iterations.

4.15 The binary splitting method

The method of binary splitting is well explained in [Haible *et al.* 1998]. Some examples are also given in [Gourdon *et al.* 2001]. This method applies to power series of rational numbers and to hypergeometric series. Most series for transcendental functions belong to this category.

If we need to take $O(P)$ terms of the series to obtain P digits of precision, then ordinary methods would require $O\left(P^2\right)$ arithmetic operations. (Each term needs $O(P)$ operations because all coefficients are rational numbers with $O(P)$ digits and we need to perform a few short multiplications or divisions.) The binary splitting method requires $O(M(P\ln P)\ln P)$ operations instead of the $O\left(P^2\right)$ operations. In other words, we need to perform long multiplications of integers of size $O(P\ln P)$ digits, but we need only $O(\ln P)$ such multiplications. The binary splitting method performs better than the straightforward summation method if the cost of multiplication is lower than $\frac{O\left(P^2\right)}{\ln P}$. This is usually true only for large enough precision (at least a thousand digits).

Thus there are two main limitations of the binary splitting method:

- As a rule, we can only compute functions of small integer or rational arguments. For instance, the method works for the calculation of a Bessel function $J_0\left(\frac{1}{3}\right)$ but not for $J_0(\pi)$. (As an exception, certain elementary functions *can* be computed by the binary splitting method for general floating-point arguments, with some clever tricks.)
- The method is fast only at high enough precision, when advanced multiplication methods become more efficient than simple $O\left(P^2\right)$ methods. The binary splitting method is actually *slower* than the simple summation when the long integer multiplication is $M(P) = O\left(P^2\right)$.

The main advantages of the method are:

- The method is asymptotically fast and, when applicable, outperforms most other methods at very high precision. The best applications of this method are for computing various constants.
- There is no accumulated round-off error since the method uses only exact integer arithmetic.
- The sum of a long series can be split into many independent partial sums which can be computed in parallel. One can store exact intermediate results of a partial summation (a few long integers), which provides straightforward checkpointing: a failed partial summation can be repeated without repeating all other parts. One can also resume the summation later to get more precision and reuse the old results, instead of starting all over again.

Description of the method

We follow [Haible *et al.* 1998]. The method applies to any series of rational numbers of the form

$$S = \sum_{n=0}^{N-1} \frac{A(n)}{B(n)},$$

where A, B are integer coefficients with $O(n\ln n)$ bits. Usually the series is of the particular form

$$S(0,N) \equiv \sum_{n=0}^{N-1} \frac{a(n)}{b(n)} \frac{p(0)\ldots p(n)}{q(0)\ldots q(n)},$$

where a, b, p, q are polynomials in n with small integer coefficients and values that fit into $O(\ln n)$ bits.

For example, the Taylor series for $\arcsin x$ (when x is a short rational number) is of this form:

$$\arcsin x = x + \frac{1}{2}\frac{x^3}{3} + \frac{1\cdot 3}{2\cdot 4}\frac{x^5}{5} + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\frac{x^7}{7} + \ldots$$

This example is of the above form with the definitions $a = 1$, $b(n) = 2n+1$, $p(n) = x^2(2n-1)$, $q(n) = 2n$ for $n \geq 1$ and $p(0) = x$, $q(0) = 1$. (The method will apply only if x is a rational number with $O(\ln N)$ bits in the numerator and the denominator.) The Taylor series for the hypergeometric function is also of this form.

The goal is to compute the sum $S(0,N)$ with a chosen number of terms N. Instead of computing the rational number S directly, the binary splitting method propose to compute the following four integers P, Q, B, and T:

$$P(0,N) \equiv p(0)\ldots p(N-1),$$
$$Q(0,N) \equiv q(0)\ldots q(N-1),$$
$$B(0,N) \equiv b(0)\ldots b(N-1),$$

and

$$T(0,N) \equiv B(0,N)\,Q(0,N)\,S(0,N).$$

At first sight it seems difficult to compute T, but the computation is organized recursively. These four integers are computed for the left (l) half and for the right (r) half of the range [0, N) and then combined using the obvious recurrence relations $P = P_l P_r$, $Q = Q_l Q_r$, $B = B_l B_r$, and the slightly less obvious relation

$$T = B_r Q_r T_l + B_l P_l T_r.$$

Here we used the shorthand $P_l \equiv P\left(0, \frac{N}{2}-1\right)$, $P_r \equiv P\left(\frac{N}{2}, N-1\right)$ and so on.

Thus the range [0, N) is split in half on each step. At the base of recursion the four integers P, Q, B, and T are computed directly. At the end of the calculation (top level of recursion), one floating-point division is performed to recover $S = \frac{T}{BQ}$. It is clear that the four integers carry the full information needed to continue the calculation with more terms. So this algorithm is easy to checkpoint and parallelize.

The integers P, Q, B, and T grow during the calculation to $O(N\ln N)$ bits, and we need to multiply these large integers. However, there are only $O(\ln N)$ steps of recursion and therefore $O(\ln N)$ long multiplications are needed. If the series converges linearly, we need $N = O(P)$ terms to obtain P digits of precision. Therefore, the total asymptotic cost of the method is $O(M(P\ln P)\ln P)$ operations.

A more general form of the binary splitting technique is also given in [Haible *et al.* 1998]. The generalization applies to series for the form

$$\sum_{n=0}^{N-1} \frac{a(n)}{b(n)} \frac{p(0)\ldots p(n)}{q(0)\ldots q(n)} \left(\frac{c(0)}{d(0)} + \ldots + \frac{c(n)}{d(n)}\right),$$

Here $a(n)$, $b(n)$, $c(n)$, $d(n)$, $p(n)$, $q(n)$ are integer-valued functions with "short" values of size $O(\ln n)$ bits. For example, the Ramanujan series for Catalan's constant is of this form.

The binary splitting technique can also be used for series with complex integer coefficients, or more generally for coefficients in

any finite algebraic extension of integers, e.q. $Z[\sqrt{2}]$ (the ring of numbers of the form $p + q\sqrt{2}$ where p, q are integers). Thus we may compute the Bessel function $J_0\left(\sqrt{3}\right)$ using the binary splitting method and obtain exact intermediate results of the form $p + q\sqrt{3}$. But this will still not help compute $J_0(\pi)$. This is a genuine limitation of the binary splitting method.

Chapter 5

Numerical algorithms II: elementary functions

5.1 Powers

There are two computational tasks: to compute the power x^n where n is an integer (but x may be a real or a complex number), and compute x^y for arbitrary (real or complex) x, y. We assume that x, y, n are "big" numbers with P significant digits.

We also assume that the power is positive, or else we need to perform an additional division to obtain $x^{-y} = \frac{1}{x^y}$.

If $x \neq 0$ is known to a relative precision ϵ, then x^y has the relative precision ϵy. This means a loss of precision if $|y| > 1$ and an improvement of precision otherwise.

Integer powers

Integer powers x^n with integer n are computed by a fast algorithm of "repeated squaring". This algorithm is well known (see, for example, the famous book, *The art of computer programming* [Knuth 1973]).

The algorithm is based on the following trick: if n is even, say $n = 2k$, then $x^n = \left(x^k\right)^2$; and if n is odd, $n = 2k + 1$, then $x^n = x\left(x^k\right)^2$. Thus we can reduce the calculation of x^n to the calculation of x^k with $k \leq \frac{n}{2}$, using at most two long multiplications. This reduction is one step of the algorithm; at each step n is reduced to at most half. This algorithm stops when n becomes 1, which happens after m steps where m is the number of bits in n. So the total number of long multiplications is at most $2m = \frac{2\ln n}{\ln 2}$. More precisely, it is equal to m plus the number of nonzero bits in the binary representation of n. On the average, we shall have $\frac{3}{2}\frac{\ln n}{\ln 2}$ long multiplications. The computational cost of the algorithm is therefore $O(M(P)\ln n)$. This should be compared with e.g. the cost of the best method for $\ln x$ which is $O(PM(P))$.

The outlined procedure is most easily implemented using recursive calls. The depth of recursion is of order $\ln n$ and should be manageable for most real-life applications. The Yacas code would look like this:

```
10# power(_x,1)<--x;
20# power(_x,n_IsEven)<-- power(x,n>>1)^2;
30# power(_x,n_IsOdd)<--x*power(x,n>>1)^2;
```

The function `power(m,n)` calculates the result of m^n for $n > 0$, $m > 0$, integer n and integer m. The bit shifts and the check for an odd number are very fast operations if the internal representation of big numbers uses base 2.

If we wanted to avoid recursion with its overhead, we would have to obtain the bits of the number n in reverse order. This is possible but is somewhat cumbersome unless we store the bits in an array.

It is easier to implement the non-recursive version of the squaring algorithm in a slightly different form. Suppose we obtain the bits b_i of the number n in the usual order, so that $n = b_0 + 2b_1 + ... + b_m \cdot 2^m$. Then we can express the power x^n as

$$x^n = x^{b_0}\left(x^2\right)^{b_1} ... \left(x^{2^m}\right)^{b_m}.$$

In other words, we evaluate x^2, x^4, ... by repeated squaring, select those x^{2^k} for which the k-th bit b_k of the number n is nonzero, and multiply all selected powers together.

In the Yacas script form, the algorithm looks like this:

```
power(x_IsPositiveInteger,n_IsPositiveInteger)<--
[
  Local(result, p);
  result:=1;
  p := x;
  While(n != 0)
  [ // at step k, p = x^(2^k)
    if (IsOdd(n))
      result := result*p;
    p := p*p;
    n := n>>1;
  ];
  result;
];
```

The same algorithm can be used to obtain a power of an integer modulo another integer, $x^n \bmod M$, if we replace the multiplication `p*p` by a modular multiplication, such as `p:=Mod(p*p,M)`. Since the remainder modulo m would be computed at each step, the results do not grow beyond M. This allows to efficiently compute even extremely large modular powers of integers.

Matrix multiplication, or, more generally, multiplication in any given ring, can be substituted into the algorithm instead of the normal multiplication. The function `IntPowerNum` encapsulates the computation of the n-th power of an expression using the binary squaring algorithm.

The squaring algorithm can be improved a little bit if we are willing to use recursion or to obtain the bits of n in the reverse order. (This was suggested in the exercise 4.21 in the book [von zur Gathen *et al.* 1999].) Let us represent the power n in base 4 instead of base 2. If q_k are the digits of n in base 4, then we can express

$$x^n = x^{q_0}\left(x^4\right)^{q_1} ... \left(x^{4^m}\right)^{q_m}.$$

We shall compute this expression from right to left: first we compute x^{q_m}. This is a small power because q_m is a digit in base 4, an integer between 0 and 3. Then we raise it to the 4th

power and multiply by $x^{q_{m-1}}$. We repeat this process until we reach the 0th digit of n. At each step we would need to multiply at most three times. Since each of the q_k is between 0 and 3, we would need to precompute x^2 and x^3 which requires one extra multiplication (x^2 would be computed anyway). Therefore the total number of long multiplications is in the worst case $3\frac{\ln n}{\ln 4}+1$. This is about 25% better than the previous worst-case result, $2\frac{\ln n}{\ln 2}$. However, the average-case improvement is only about 8% because the average number of multiplications in the base-4 method is $\frac{11}{4}\frac{\ln n}{\ln 4}$.

We might then use the base 8 instead of 4 and obtain a further small improvement. (Using bases other than powers of 2 is less efficient.) But the small gain in speed probably does not justify the increased complexity of the algorithm.

Real powers

The squaring algorithm can be used to obtain integer powers x^n in any ring—as long as n is an integer, x can be anything from a complex number to a matrix. But for a general real number n, there is no such trick and the power x^n has to be computed through the logarithm and the exponential function, $x^n = \exp(n \ln x)$. (This also covers the case when x is negative and the result is a complex number.)

An exceptional case is when n is a rational number with a very small numerator and denominator, for example, $n = \frac{2}{3}$. In this case it is faster to take the square of the cubic root of x. (See the section on the computation of roots below.) Then the case of negative x should be handled separately. This speedup is not implemented in Yacas.

Note that the relative precision changes when taking powers. If x is known to relative precision ϵ, i.e. x represents a real number that could be $x(1+\epsilon)$, then $x^2 \approx x(1+2\epsilon)$ has relative precision 2ϵ, while $\sqrt{x}$ has relative precision $\frac{\epsilon}{2}$. So if we square a number x, we lose one significant bit of x, and when we take a square root of x, we gain one significant bit.

5.2 Roots

Computation of roots $r = \sqrt[n]{x}$ is efficient when n is a small integer. The basic approach is to numerically solve the equation $r^n = x$.

Note that the relative precision is improved after taking a root with $n > 1$.

Method 1: bisection

The square root can be computed by using the bisection method, which works well for integers (if only the integer part of the square root is needed). The algorithm is described in [Johnson 1987]. The general approach is to scan each bit of the input number and to see if a certain bit should be set in the resulting integer. The time is linear in the number of decimals, or logarithmic in the input number. The method is very similar in approach to the repeated squaring method described above for raising numbers to a power.

For integer N, the following steps are performed:

1. Find the highest bit set, l_2, in the number N.
2. $1 << \left(\frac{l_2}{2}\right)$ is definitely a bit that is set in the result. Start by setting that bit in the result, $u = 1 << l_2$. It is also the highest bit set in the result.
3. Now, traverse all the lower bits, one by one. For each lower bit, starting at lnext $= l_2 - 1$, set $v = 1 <<$ lnext. Now, $(u+v)^2 = u^2 + 2uv + v^2$. If $(u+v)^2 \leq N$, then the bit set in v should also be set in the result, u, otherwise that bit should be cleared.
4. Set lnext = lnext $-$ 1, and repeat until all bits are tested, and lnext $= 0$ and return the result found.

The intermediate results, u^2, v^2 and $2uv$ can be maintained easily too, due to the nature of the numbers involved (v having only one bit set, and it being known which bit that is).

For floating point numbers, first the required number of decimals p after the decimal point is determined. Then the input number N is multiplied by a power of 10 until it has $2p$ decimal. Then the integer square root calculation is performed, and the resulting number has p digits of precision.

Below is some YACAS script code to perform the calculation for integers.

```
//sqrt(1) = 1, sqrt(0) = 0
10 # BisectSqrt(0) <-- 0;
10 # BisectSqrt(1) <-- 1;

20 # BisectSqrt(N_IsPositiveInteger) <--
[
  Local(l2,u,v,u2,v2,uv2,n);

  // Find highest set bit, l2
  u  := N;
  l2 := 0;
  While (u!=0)
  [
    u:=u>>1;
    l2++;
  ];
  l2--;

  // 1<<(l2/2) now would be a good under estimate
  // for the square root. 1<<(l2/2) is definitely
  // set in the result. Also it is the highest
  // set bit.
  l2 := l2>>1;

  // initialize u and u2 (u2==u^2).
  u  := 1 << l2;
  u2 := u << l2;

  // Now for each lower bit:
  While( l2 != 0 )
  [
l2--;
     // Get that bit in v, and v2 == v^2.
      v  := 1<<l2;
      v2 := v<<l2;

      // uv2 == 2*u*v, where 2==1<<1, and
      // v==1<<l2, thus 2*u*v ==
      // (1<<1)*u*(1<<l2) == u<<(l2+1)
      uv2 := u<<(l2 + 1);

      // n = (u+v)^2  = u^2 + 2*u*v + v^2
      //   = u2+uv2+v2
      n := u2 + uv2 + v2;

      // if n (possible new best estimate for
      // sqrt(N)^2 is smaller than N, then the
      // bit l2 is set in the result, and
      // add v to u.
      if( n <= N )
```

```
        [
          u  := u+v;  // u <- u+v
          u2 := n;    // u^2 <- u^2 + 2*u*v + v^2
        ];
        l2--;
      ];
      u; // return result, accumulated in u.
  ];
```

`BisectSqrt(N)` computes the integer part of $\sqrt{N}$ for integer N. (If we need to obtain more digits, we should first multiply N by a suitable power of 2.) The algorithm works for floats as well as for integers.

The bisection algorithm uses only additions and bit shifting operations. Suppose the integer N has P decimal digits, then it has $n = P\frac{\ln 10}{\ln 2}$ bits. For each bit, the number of additions is about 4. Since the cost of an addition is linear in the number of bits, the total complexity of the bisection method is roughly $4n^2 = O\left(P^2\right)$.

Method 2: Newton's iteration

An efficient method for computing the square root is found by using Newton's iteration for the equation $r^2 - x = 0$. The initial value of r can be obtained by bit counting and shifting, as in the bisection method. The iteration formula is

$$r' = \frac{r}{2} + \frac{x}{2r}.$$

The convergence is quadratic, so we double the number of correct digits at each step. Therefore, if the initial guess is accurate to one bit, the number of steps n needed to obtain P decimal digits is

$$n = \frac{\ln P\frac{\ln 10}{\ln 2}}{\ln 2} = O\left(\ln P\right).$$

We need to perform one long division at each step; a long division costs $O\left(M\left(P\right)\right)$. Therefore the total complexity of this algorithm is $O\left(M\left(P\right)\ln P\right)$. This is better than the $O\left(P^2\right)$ algorithm if the cost of multiplication is below $O\left(P^2\right)$.

In most implementations of arbitrary-precision arithmetic, the time to perform a long division is several times that of a long multiplication. Therefore it makes sense to use a method that avoids divisions. One variant of Newton's method is to solve the equation $\frac{1}{r^2} = x$. The solution of this equation $r = \frac{1}{\sqrt{x}}$ is the limit of the iteration

$$r' = r + r\frac{1 - r^2x}{2}$$

that does not require any divisions (but instead requires three multiplications). The final multiplication rx completes the calculation of the square root.

As usual with Newton's method, all errors are automatically corrected, so the working precision can be gradually increased until the last iteration. The full precision of P digits is used only at the last iteration; the last-but-one iteration uses $\frac{P}{2}$ digits and so on.

An optimization trick is to combine the multiplication by x with the last iteration. Then computations can be organized in a special way to avoid the last full-precision multiplication. (This is described in [Karp *et al.* 1997] where the same trick is also applied to Newton's iteration for division.)

The idea is the following: let r be the P-digit approximation to $\frac{1}{\sqrt{x}}$ at the beginning of the last iteration. (In this notation, $2P$ is the precision of the final result, so x is also known to about $2P$ digits.) The unmodified procedure would have run as follows:

$$r' = r + r\frac{1 - r^2x}{2},$$

$$s = xr'.$$

Then s would have been the final result, $\sqrt{x}$ to $2P$ digits. We would need one multiplication $M\left(P\right)$ with $2P$-digit result to compute r^2, then one $M\left(2P\right)$ to compute r^2x (the product of a P-digit r^2 and a $2P$-digit x). Then we subtract this from 1 and lose P digits since r was already a P-digit approximation to $\frac{1}{\sqrt{x}}$. The value $y \equiv 1 - r^2x$ is of order 10^{-P} and has P significant digits. So the third multiplication, ry, is only $M\left(P\right)$. The fourth multiplication sx is again $M\left(2P\right)$. The total cost is then $2M\left(P\right) + 2M\left(2P\right)$.

Now consider Newton's iteration for $s \approx \sqrt{x}$,

$$s' = s + \frac{1}{s}\frac{1 - s^2x}{2}.$$

The only reason we are trying to avoid it is the division by s. However, after all but the last iterations for r we already have a P-digit approximation for $\frac{1}{s}$, which is r. Therefore we can simply define $s = rx$ and perform the last iteration for s, taking $\frac{1}{s} \approx r$. This is slightly inexact, but the error is higher-order than the precision of the final result, because Newton's method erases any accumulated errors. So this will give us $2P$ digits of s without divisions, and lower the total computational cost.

Consider the cost of the last iteration of this combined method. First, we compute $s = xr$, but since we only need P correct digits of s, we can use only P digits of x, so this costs us $M\left(P\right)$. Then we compute s^2x which, as before, costs $M\left(P\right) + M\left(2P\right)$, and then we compute $r\left(1 - s^2x\right)$ which costs only $M\left(P\right)$. The total cost is therefore $3M\left(P\right) + M\left(2P\right)$, so we have traded one multiplication with $2P$ digits for one multiplication with P digits. Since the time of the last iteration dominates the total computing time, this is a significant cost savings. For example, if the multiplication is quadratic, $M\left(P\right) = O\left(P^2\right)$, then this saves about 30% of total execution time; for linear multiplication, the savings is about 16.67%.

These optimizations do not change the asymptotic complexity of the method, although they do reduce the constant in front of $O\left(\right)$.

Method 3: argument reduction and interpolation

Before using the bisection or Newton's method, we might apply some argument reduction to speed up the convergence of the iterations and to simplify finding the first approximation.

Suppose we need to find $\sqrt{x}$. Choose an integer n such that $\frac{1}{4} < x' \equiv 4^{-n}x \leq 1$. The value of n is easily found from bit counting: if b is the bit count of x, then

$$n = \left\lfloor \frac{b+1}{2} \right\rfloor.$$

We find

$$\sqrt{x} = 2^n\sqrt{x'}.$$

The precision of x' is the same as that of x since 2^n is an exact number.

To compute $\sqrt{x'}$, we use Newton's method with the initial value x'_0 obtained by interpolation of the function $\sqrt{x}$ on the interval $[\frac{1}{4},1]$. A suitable interpolation function might be taken as simply $\frac{2x+1}{3}$ or more precisely

$$\sqrt{x} \approx \frac{1}{90}\left(-28x^2 + 95x + 23\right).$$

By using a particular interpolation function, we can guarantee a certain number of precise bits at every iteration.

This may save a few iterations, at the small expense of evaluating the interpolation function once at the beginning. However, in computing with high precision the initial iterations are very fast and this argument reduction does not give a significant speed gain. But the gain may be important at low precisions, and this technique is sometimes used in microprocessors.

Method 4: Halley's iteration

A separate function `IntNthRoot` is provided to compute the integer part of $\sqrt[s]{n}$ for integer n and s. For a given s, it evaluates the integer part of $\sqrt[s]{n}$ using only integer arithmetic with integers of size $n^{1+\frac{1}{s}}$. This can be done by Halley's iteration method, solving the equation $x^s = n$. For this function, the Halley iteration sequence is monotonic. The initial guess is $x_0 = 2^{\frac{b(n)}{s}}$ where $b(n)$ is the number of bits in n obtained by bit counting or using the integer logarithm function. It is clear that the initial guess is accurate to within a factor of 2. Since the relative error is squared at every iteration, we need as many iteration steps as bits in $\sqrt[s]{n}$.

Since we only need the integer part of the root, it is enough to use integer division in the Halley iteration. The sequence x_k will monotonically approximate the number $\sqrt[s]{n}$ from below if we start from an initial guess that is less than the exact value. (We start from below so that we have to deal with smaller integers rather than with larger integers.) If $n = p^s$, then after enough iterations the floating-point value of x_k would be slightly less than p; our value is the integer part of x_k. Therefore, at each step we check whether $1 + x_k$ is a solution of $x^s = n$, in which case we are done; and we also check whether $(1 + x_k)^s > n$, in which case the integer part of the root is x_k. To speed up the Halley iteration in the worst case when $s^s > n$, it is combined with bisection. The root bracket interval $x_1 < x < x_2$ is maintained and the next iteration x_{k+1} is assigned to the midpoint of the interval if Halley's formula does not give sufficiently rapid convergence. The initial root bracket interval can be taken as x_0, $2x_0$.

If s is very large ($s^s > n$), the convergence of both Newton's and Halley's iterations is almost linear until the final few iterations. Therefore it is faster to evaluate the floating-point power for large b using the exponential and the logarithm.

Method 5: higher-order iterations

A higher-order generalization of Newton's iteration for inverse square root $\frac{1}{\sqrt{x}}$ is:

$$r' = r + \frac{r}{2}\left(1 - r^2 x\right) + 3\frac{r}{8}\left(1 - r^2 x\right)^2 + ...$$

The more terms of the series we add, the higher is the convergence rate. This is the Taylor series for $(1-y)^{-\frac{1}{2}}$ where $y \equiv 1 - r^2 x$. If we take the terms up to y^{n-1}, the precision at the next iteration will be multiplied by n. The usual second-order iteration (our "method 2") corresponds to $n = 2$.

The trick of combining the last iteration with the final multiplication by x can be also used with all higher-order schemes.

Consider the cost of one iteration of n-th order. Let the initial precision of r be P; then the final precision is kP and we use up to nP digits of x. First we compute $y \equiv 1 - r^2 x$ to $P(n-1)$ digits, this costs $M(P)$ for r^2 and then $M(Pn)$ for $r^2 x$. The value of y is of order 10^{-P} and it has $P(n-1)$ digits, so we only need to use that many digits to multiply it by r, and ry now costs us $M(P(n-1))$. To compute y^k (here $2 \le k \le n-1$), we need $M(P(n-k))$ digits of y; since we need all consecutive powers of y, it is best to compute the powers one after another, lowering the precision on the way. The cost of computing $ry^k y$ after having computed ry^k is therefore $M(P(n-k-1))$. The total cost of the iteration comes to

$$2M(P) + M(2P) + ... + M((n-1)P) + M(nP).$$

From the general considerations in the previous chapter (see the section on Newton's method) it follows that the optimal order is $n = 2$ and that higher-order schemes are slower in this case.

Which method to use

The bisection method (1) for square roots is probably the fastest for small integers or low-precision floats. Argument reduction and/or interpolation (3) can be used to simplify the iterative algorithm or to make it more robust.

Newton's method (2) is best for all other cases: large precision and/or roots other than square roots.

5.3 Logarithm

The basic computational task is to obtain the logarithm of a real number. However, sometimes only the integer part of the logarithm is needed and the logarithm is taken with respect to an integer base. For example, we might need to evaluate the integer part of $\frac{\ln n}{\ln 2}$ where n is a large integer, to find how many bits are needed to hold n. Computing this "integer logarithm" is a much easier task than computing the logarithm in floating-point.

Logarithms of complex numbers can be reduced to elementary functions of real numbers, for example:

$$\ln(a + \imath b) = \frac{1}{2}\ln\left(a^2 + b^2\right) + \imath \arctan\frac{b}{a}.$$

For a negative real number $x < 0$, we have

$$\ln x = \ln|x| + \imath\pi.$$

This assumes, of course, an appropriate branch cut for the complex logarithm. A natural choice is to cut along the negative real semiaxis, $Im(z) = 0$, $Re(z) < 0$.

Integer logarithm

The "integer logarithm", defined as the integer part of $\frac{\ln x}{\ln b}$, where x and b are integers, is computed using a special routine `IntLog(x,b)` with purely integer math. When both arguments are integers and only the integer part of the logarithm is needed, the integer logarithm is much faster than evaluating the full floating-point logarithm and truncating the result.

The basic algorithm consists of (integer-) dividing x by b repeatedly until x becomes 0 and counting the necessary number of divisions. If x has P digits and b and P are small numbers, then division is linear in P and the total number of divisions is $O(P)$. Therefore this algorithm costs $O\left(P^2\right)$ operations.

A speed-up for large x is achieved by first comparing x with b, then with b^2, b^4, etc., without performing any divisions. We perform n such steps until the factor b^{2^n} is larger than x. At this point, x is divided by the previous power of b and the remaining value is iteratively compared with and divided by successively smaller powers of b. The number of squarings needed to compute b^{2^n} is logarithmic in P. However, the last few of these multiplications are long multiplications with numbers of length $\frac{P}{4}$, $\frac{P}{2}$, P digits. These multiplications take the time $O(M(P))$. Then we need to perform another long division and a series of

progressively shorter divisions. The total cost is still $O(M(P))$. For large P, the cost of multiplication $M(P)$ is less than $O\left(P^2\right)$ and therefore this method is preferable.

There is one special case, the binary (base 2) logarithm. Since the internal representation of floating-point numbers is usually in binary, the integer part of the binary logarithm can be usually implemented as a constant-time operation.

Real logarithms

There are many methods to compute the logarithm of a real number. Here we collect these methods and analyze them.

The logarithm satisfies $\ln \frac{1}{x} = -\ln x$. Therefore we need to consider only $x > 1$, or alternatively, only $0 < x < 1$.

Note that the relative precision for x translates into *absolute* precision for $\ln x$. This is because $\ln x(1+\epsilon) \approx \ln x + \epsilon$ for small ϵ. Therefore, the relative precision of the result is at best $\frac{\epsilon}{\ln x}$. So to obtain P decimal digits of $\ln x$, we need to know $P - \frac{\ln|\ln x|}{\ln 10}$ digits of x. This is better than the relative precision of x if $x > e$ but worse if $x \approx 1$.

Method 1: Taylor series

The logarithm function $\ln x$ for general (real or complex) x such that $|x-1| < 1$ can be computed using the Taylor series,

$$\ln(1+z) = z - \frac{z^2}{2} + \frac{z^3}{3} - ...$$

The series converges quite slowly unless $|x|$ is small. For real $x < 1$, the series is monotonic,

$$\ln(1-z) = -z - \frac{z^2}{2} - \frac{z^3}{3} - ...,$$

and the round-off error is somewhat smaller in that case (but not very much smaller, because the Taylor series method is normally used only for very small x).

If $x > 1$, then we can compute $-\ln \frac{1}{x}$ instead of $\ln x$. However, the series converges very slowly if x is close to 0 or to 2.

Here is an estimate of the necessary number of terms to achieve a (relative) precision of P decimal digits when computing $\ln(1+x)$ for small real x. Suppose that x is of order 10^{-N}, where $N \geq 1$. The error after keeping n terms is not greater than the first discarded term, $\frac{x^{n+1}}{n+1}$. The magnitude of the sum is approximately x, so the relative error is $\frac{x^n}{n+1}$ and this should be smaller than 10^{-P}. We obtain a sufficient condition $n > \frac{P}{N}$.

All calculations need to be performed with P digits of precision. The "rectangular" scheme for evaluating n terms of the Taylor series needs about $2\sqrt{n}$ long multiplications. Therefore the cost of this calculation is $2\sqrt{\frac{P}{N}}M(P)$.

When P is very large (so that a fast multiplication can be used) and x is a small rational number, then the binary splitting technique can be used to compute the Taylor series. In this case the cost is $O(M(P)\ln P)$.

Note that we need to know $P+N$ digits of $1+x$ to be able to extract P digits of $\ln(1+x)$. The N extra digits will be lost when we subtract 1 from $1+x$.

Method 2: square roots + Taylor series

The method of the Taylor series allows to compute $\ln x$ efficiently when $x - 1 = 10^{-N}$ is very close to 1 (i.e. for large N). For other values of x the series converges very slowly. We can transform the argument to improve the performance of the Taylor series.

One way is to take several square roots, reducing x to $x^{2^{-k}}$ until x becomes close to 1. Then we can compute $\ln x^{2^{-k}}$ using the Taylor series and use the identity $\ln x = 2^k \ln x^{2^{-k}}$.

The number of times to take the square root can be chosen to minimize the total computational cost. Each square root operation takes the time equivalent to a fixed number c of long multiplications. (According to the estimate of [Brent 1975], $c \approx \frac{13}{2}$.) Suppose x is initially of order 10^L where $L > 0$. Then we can take the square root k_1 times and reduce x to about 1.33. Here we can take $k_1 \approx \frac{\ln L}{\ln 2} + 3$. After that, we can take the square root k_2 times and reduce x to $1 + 10^{-N}$ with $N \geq 1$. For this we need $k_2 \approx 1 + N\frac{\ln 10}{\ln 2}$ square roots. The cost of all square roots is $c(k_1 + k_2)$ long multiplications. Now we can use the Taylor series and obtain $\ln x^{2^{-k_1-k_2}}$ in $2\sqrt{\frac{P}{N}}$ multiplications. We can choose N to minimize the total cost for a given L.

Method 3: inverse exponential

The method is to solve the equation $\exp(x) - a = 0$ to find $x = \ln a$. We can use either the quadratically convergent Newton iteration,

$$x' = x - 1 + \frac{a}{\exp(x)},$$

or the cubically convergent Halley iteration,

$$x' = x - 2\frac{\exp(x) - a}{\exp(x) + a}.$$

Each iteration requires one evaluation of $\exp(x)$ and one long division. Newton's iteration can be rewritten through $\exp(-x)$ but this does not really avoid a long division: $\exp(-x)$ for positive x is usually computed as $\frac{1}{\exp(x)}$ because other methods are much less efficient. Therefore the Halley iteration is preferable.

The initial value for x can be found by bit counting on the number a. If m is the "bit count" of a, i.e. m is an integer such that $\frac{1}{2} \leq a \cdot 2^{-m} < 1$, then the first approximation to $\ln a$ is $m \ln 2$. (Here we can use a very rough approximation to $\ln 2$, for example, $\frac{2}{3}$.)

The initial value found in this fashion will be correct to about one bit. The number of digits triples at each Halley iteration, so the result will have about $3k$ correct bits after k iterations (this disregards round-off error). Therefore the required number of iterations for P decimal digits is $\frac{1}{\ln 3}\ln P\frac{\ln 2}{\ln 10}$.

This method is currently faster than other methods (with internal math) and so it is implemented in the routine `Internal'LnNum`.

This method can be generalized to higher orders. Let $y \equiv 1 - a\exp(-x_0)$, where x_0 is a good approximation to $\ln a$ so y is small. Then $\ln a = x_0 + \ln(1-y)$ and we can expand in y to obtain

$$\ln a = x_0 - y - \frac{y^2}{2} - \frac{y^3}{3} - ...$$

By truncating this sum after k-th term we obtain a $(k-1)$-th order method that multiplies the number of correct digits by $k+1$ after each iteration.

The optimal number of terms to take depends on the speed of the implementation of $\exp(x)$.

Method 4: AGM

A fast algorithm based on the AGM sequence was given by Salamin (see [Brent 1975]). The formula is based on an asymptotic relation,

$$\ln x = \pi x \frac{1 + 4x^{-2}\left(1 - \frac{1}{\ln x}\right) + O\left(x^{-4}\right)}{2\mathrm{AGM}(x, 4)}.$$

If x is large enough, the numerator can be replaced by 1. "Large enough" for a desired precision of P decimal digits means that $4x^{-2} < 10^{-P}$. The AGM algorithm gives P digits only for such large values of x, unlike the Taylor series which is only good for x close to 1.

The required number of AGM iterations is approximately $2\frac{\ln P}{\ln 2}$. For smaller values of x (but $x > 1$), one can either raise x to a large integer power r and then compute $\frac{1}{r}\ln x^r$ (this is quick only if x is itself an integer or a rational), or multiply x by a large integer power of 2 and compute $\ln 2^s x - s\ln 2$ (this is better for floating-point x). Here the required powers are

$$r = \frac{\ln 10^P \cdot 4}{2\ln x},$$

$$s = P\frac{\ln 10}{2\ln 2} + 1 - \frac{\ln x}{\ln 2}.$$

The values of these parameters can be found quickly by using the integer logarithm procedure `IntLog`, while constant values such as $\frac{\ln 10}{\ln 2}$ can be simply approximated by rational numbers because r and s do not need to be very precise (but they do need to be large enough). For the second calculation, $\ln 2^s x - s\ln 2$, we must precompute $\ln 2$ to the same precision of P digits. Also, the subtraction of a large number $s\ln 2$ leads to a certain loss of precision, namely, about $\frac{\ln s}{\ln 10}$ decimal digits are lost, therefore the operating precision must be increased by this number of digits. (The quantity $\frac{\ln s}{\ln 10}$ is computed, of course, by the integer logarithm procedure.)

If $x < 1$, then $(-\ln\frac{1}{x})$ is computed.

Finally, there is a special case when x is very close to 1, where the Taylor series converges quickly but the AGM algorithm requires to multiply x by a large power of 2 and then subtract two almost equal numbers, leading to a great waste of precision. Suppose $1 < x < 1 + 10^{-M}$, where M is large (say of order P). The Taylor series for $\ln(1+\epsilon)$ needs about $N = -P\frac{\ln 10}{\ln\epsilon} = \frac{P}{M}$ terms. If we evaluate the Taylor series using the rectangular scheme, we need $2\sqrt{N}$ multiplications and $\sqrt{N}$ units of storage. On the other hand, the main slow operation for the AGM sequence is the geometric mean $\sqrt{ab}$. If $\sqrt{ab}$ takes an equivalent of c multiplications (Brent's estimate is $c = \frac{13}{2}$ but it may be more in practice), then the AGM sequence requires $2c\frac{\ln P}{\ln 2}$ multiplications. Therefore the Taylor series method is more efficient for

$$M > \frac{1}{c^2}P\left(\frac{\ln 2}{\ln P}\right)^2.$$

In this case it requires at most $c\frac{\ln P}{\ln 2}$ units of storage and $2c\frac{\ln P}{\ln 2}$ multiplications.

For larger $x > 1+10^{-M}$, the AGM method is more efficient. It is necessary to increase the working precision to $P + M\frac{\ln 2}{\ln 10}$ but this does not decrease the asymptotic speed of the algorithm. To compute $\ln x$ with P digits of precision for any x, only $O(\ln P)$ long multiplications are required.

Method 5: argument reduction + Taylor series

Here is a straightforward method that reduces $\ln x$ for large $x > 2$ to $\ln(1+\delta)$ with a small δ; now the logarithm can be quickly computed using the Taylor series.

The simplest version is this: for integer m, we have the identity $\ln x = m + \ln xe^{-m}$. Assuming that $e \equiv \exp(1)$ is precomputed, we can find the smallest integer m for which $x \le e^m$ by computing the integer powers of e and comparing with x. (If x is large, we do not really have to go through all integer m: instead we can estimate m by bit counting on x and start from e^m.) Once we found m, we can use the Taylor series on $1 - \delta \equiv xe^{-m}$ since we have found the smallest possible m, so $0 \le \delta < 1 - \frac{1}{e}$.

A refinement of this method requires to precompute $b = \exp\left(2^{-k}\right)$ for some fixed integer $k \ge 1$. (This can be done efficiently using the squaring trick for the exponentials.) First we find the smallest power m of b which is above x. To do this, we compute successive powers of b and find the first integer m such that $x \le b^m = \exp\left(m \cdot 2^{-k}\right)$. When we find such m, we define $1 - \delta \equiv xb^{-m}$ and then δ will be small, because $0 < \delta < 1 - \frac{1}{b} \approx 2^{-k}$ (the latter approximation is good if k is large). We compute $\ln(1-\delta)$ using the Taylor series and finally find $\ln x = m \cdot 2^k + \ln(1-\delta)$.

For smaller δ, the Taylor series of $\ln(1-\delta)$ is more efficient. Therefore, we have a trade-off between having to perform more multiplications to find m, and having a faster convergence of the Taylor series.

Method 6: transformed Taylor series

We can use an alternative Taylor series for the logarithm that converges for all x,

$$\ln(a+z) = \ln a + 2\sum_{k=0}^{\infty}\frac{1}{2k+1}\left(\frac{z}{2a+z}\right)^{2k+1}.$$

This series is obtained from the series for $\operatorname{arctanh} x$ and the identity

$$2\operatorname{arctanh} x = \ln\frac{1+x}{1-x}.$$

This series converges for all z such that $Re(a+z) > 0$ if $a > 0$. The convergence rate is, however, the same as for the original Taylor series. In other words, it converges slowly unless $\frac{z}{2a+z}$ is small. The parameter a can be chosen to optimize the convergence; however, $\ln a$ should be either precomputed or easily computable for this method to be efficient.

For instance, if $x > 1$, we can choose $a = 2^k$ for an integer $k \ge 1$, such that $2^{k-1} \le x < 2^k = a$. (In other words, k is the bit count of x.) In that case, we represent $x = a - z$ and we find that the expansion parameter $\frac{z}{2a-z} < \frac{1}{3}$. So a certain rate of convergence is guaranteed, and it is enough to take a fixed number of terms, about $P\frac{\ln 10}{\ln 3}$, to obtain P decimal digits of $\ln x$ for any x. (We should also precompute $\ln 2$ for this scheme to work.)

If $0 < x < 1$, we can compute $-\ln\frac{1}{x}$.

This method works robustly but is slower than the Taylor series with some kind of argument reduction. With the "rectangular" method of summation, the total cost is $O\left(\sqrt{P}M(P)\right)$.

Method 7: binary reduction

This method is based on the binary splitting technique and is described in [Haible *et al.* 1998] with a reference to [Brent 1976].

The method shall compute $\ln(1+x)$ for real x such that $|x| < \frac{1}{2}$. For other x, some sort of argument reduction needs to be applied. (So this method is a replacement for the Taylor series that is asymptotically faster at very high precision.)

The main idea is to use the property

$$\ln\left(1 + z\cdot 2^{-k}\right) = z\cdot 2^{-k} + O\left(2^{-2k}\right)$$

for integer $k \ge 1$ and real z such that $|z| \le 1$. This property allows to find the first $2k$ binary digits of $\ln\left(1 + z\cdot 2^{-k}\right)$ by inspection: these digits are the first k nonzero digits of z. Then we can perform a very quick computation of $\exp\left(-m\cdot 2^{-k}\right)$ for integer k, m (evaluated using the binary splitting of the Taylor series) and reduce z by at least the factor 2^k.

More formally, we can write the method as a loop over k, starting with $k = 1$ and stopping when $2^{-k} < 10^{-P}$ is below the required precision. At the beginning of the loop we have $y = 0$, $z = x$, $k = 1$ and $|z| < \frac{1}{2}$. The loop invariants are $(1 + z)\exp(y)$ which is always equal to the original number $1+x$, and the condition $|z| < 2^{-k}$. If we construct this loop, then it is clear that at the end of the loop $1 + z$ will become 1 to required precision and therefore y will be equal to $\ln(1 + x)$.

The body of the loop consists of the following steps:

1. Separate the first k significant digits of z:

$$f = 2^{-2k}\left\lfloor 2^{2k} z \right\rfloor.$$

   Now f is a good approximation for $\ln(1 + z)$.

2. Compute $\exp(-f)$ using the binary splitting technique (f is a rational number with the denominator 2^{2k} and numerator at most 2^k). It is in fact sufficient to compute $1 - \exp(-f)$ which does not need all digits.

3. Set $y = y + f$ and $z = (1 + z)\exp(-f) - 1$.

The total number of steps in the loop is at most $\frac{\ln P \frac{\ln 10}{\ln 2}}{\ln 2}$. Each step requires $O(M(P)\ln P)$ operations because the exponential $\exp(-f)$ is taken at a rational arguments f and can be computed using the binary splitting technique. (Toward the end of the loop, the number of significant digits of f grows, but the number of digits we need to obtain is decreased. At the last iteration, f contains about half of the digits of x but computing $\exp(-f)$ requires only one term of the Taylor series.) Therefore the total cost is $O\left(M(P)(\ln P)^2\right)$.

Essentially the same method can be used to evaluate a complex logarithm, $\ln(a + \imath b)$. It is slower but the asymptotic cost is the same.

Method 8: continued fraction

There is a continued fraction representation of the logarithm:

$$\ln(1 + x) = \cfrac{x}{1 + \cfrac{x}{2 + \cfrac{x}{3 + \cfrac{4x}{4 + \cfrac{4x}{5 + \cfrac{9x}{6 + \ldots}}}}}}.$$

This fraction converges for all x, although the speed of convergence varies with the magnitude of x.

This method does not seem to provide a computational advantage compared with the other methods.

Method 9: bisection

A simple bisection algorithm for $\frac{\ln x}{\ln 2}$ (the base 2 logarithm) with real x is described in [Johnson 1987].

First, we need to divide x by a certain power of 2 to reduce x to y in the interval $1 \le y < 2$. We can use the bit count $m = \text{BitCount}(x)$ to find an integer m such that $\frac{1}{2} \le x \cdot 2^{-m} < 1$ and take $y = x \cdot 2^{1-m}$. Then $\frac{\ln x}{\ln 2} = \frac{\ln y}{\ln 2} + m - 1$.

Now we shall find the bits in the binary representation of $\frac{\ln y}{\ln 2}$, one by one. Given a real y such that $1 \le y < 2$, the value $\frac{\ln y}{\ln 2}$ is between 0 and 1. Now,

$$\frac{\ln y}{\ln 2} = 2^{-1}\frac{\ln y^2}{\ln 2}.$$

The leading bit of this value is 1 if $y^2 \ge 2$ and 0 otherwise. Therefore we need to compute $y' = y^2$ using a long P-digit multiplication and compare it with 2. If $y' \ge 2$ we set $y = \frac{y'}{2}$, otherwise we set $y = y'$; then we obtain $1 \le y < 2$ again and repeat the process to extract the next bit of $\frac{\ln y}{\ln 2}$.

The process is finished either when the required number of bits of $\frac{\ln y}{\ln 2}$ is found, or when the precision of the argument is exhausted, whichever occurs first. Note that each iteration requires a long multiplication (squaring) of a number, and each squaring loses 1 bit of relative precision, so after k iterations the number of precise bits of y would be $P - k$. Therefore we cannot have more iterations than P (the number of precise bits in the original value of x). The total cost is $O(PM(P))$.

The squaring at each iteration needs to be performed not with all digits, but with the number of precise digits left in the current value of y. This does not reduce the asymptotic complexity; it remains $O(PM(P))$.

Comparing this method with the Taylor series, we find that the only advantage of this method is simplicity. The Taylor series requires about P terms, with one long multiplication and one short division per term, while the bisection method does not need any short divisions. However, the rectangular method of Taylor summation cuts the time down to $O\left(\sqrt{P}\right)$ long multiplications, at a cost of some storage and bookkeeping overhead. Therefore, the bisection method may give an advantage only at very low precisions. (This is why it is sometimes used in microprocessors.) The similar method for the exponential function requires a square root at every iteration and is never competitive with the Taylor series.

Which method to use

This remains to be seen.

5.4 Exponential

There are many methods to compute the exponential of a real number. Here we collect these methods and analyze them.

The exponential function satisfies $\exp(-x) = \frac{1}{\exp(x)}$. Therefore we need to consider only $x > 0$.

Note that the *absolute* precision for x translates into *relative* precision for $\exp(x)$. This is because $\exp(x + \epsilon) \approx \exp(x)(1 + \epsilon)$ for small ϵ. Therefore, to obtain P decimal digits of $\exp(x)$ we need to know x with absolute precision of at least 10^{-P}, that is, we need to know $P + \frac{\ln|x|}{\ln 10}$ digits of x. Thus, the relative precision becomes worse after taking the exponential if $x > 1$ but improves if x is very small.

Method 1: Taylor series

The exponential function is computed using its Taylor series,

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \ldots$$

This series converges for all (complex) x, but if $|x|$ is large, or if x is negative, then the series converges slowly and/or gives a large round-off error. So one should use this Taylor series only when x is small.

If x is sufficiently small, e.g. $|x| < 10^{-M}$ and $M > \frac{\ln P}{\ln 10}$, then it is enough to take about $\frac{P}{M}$ terms in the Taylor series. If x is of order 1, one needs about $P\frac{\ln 10}{\ln P}$ terms.

If $x = \frac{p}{q}$ is a small rational number, and if a fast multiplication is available, then the binary splitting technique should be used to evaluate the Taylor series. The computational cost of that is $O(M(P\ln P)\ln P)$.

Method 2: squaring + Taylor series

A speed-up trick used for large x is to divide the argument by some power of 2 and then square the result several times, i.e.

$$\exp(x) = \left(\exp\left(2^{-k}x\right)\right)^{2^k},$$

where k is chosen sufficiently large so that the Taylor series converges quickly at $2^{-k}x$ [Smith 1985]. The threshold value for x can be obtained through `MathExpThreshold()`, and set through `SetMathExpThreshold(threshold)` in `stdfuncs`.

A modification of the squaring reduction allows to significantly reduce the round-off error [Brent 1978]. Instead of $\exp(x) = \left(\exp\left(\frac{x}{2}\right)\right)^2$, we use the identity

$$\exp(x) - 1 = \left(\exp\left(\frac{x}{2}\right) - 1\right)\left(\exp\left(\frac{x}{2}\right) + 1\right)$$

and reduce $\exp(x)-1$ directly to $\exp\left(\frac{x}{2}\right)-1$. If $y = \exp\left(\frac{x}{2}\right)-1$, then $\exp(x) - 1 = 2y + y^2$.

Method 3: inverse logarithm

An alternative way to compute $x = \exp(a)$ if a fast logarithm routine is available would be to solve the equation $\ln x = a$. (This might be better at very large precision where the AGM method for the logarithm is asymptotically the fastest.)

Newton's method gives the iteration

$$x' = x(a + 1 - \ln x).$$

The iteration converges quadratically to $\exp(a)$ if the initial value of x is $0 < x < \exp(a + 1)$.

A cubically convergent formula is obtained if we replace $\ln x = a$ by an equivalent equation

$$\frac{\ln x - a}{\ln x - a - 2} = 0.$$

For this equation, Newton's method gives the iteration

$$x' = x\frac{1 + (a + 1 - \ln x)^2}{2}.$$

This iteration converges for initial values $0 < x < \exp(a + 2)$ with a cubic convergence rate and requires only one more multiplication, compared with Newton's method for $\ln x = a$. A good initial guess can be found by raising 2 to the integer part of $\frac{a}{\ln 2}$ (the value $\ln 2$ can be approximated from above by a suitable rational number, e.g. $\frac{7050}{10171}$).

This cubically convergent iteration seems to follow from a good equivalent equation that we guessed. But it turns out that it can be generalized to higher orders. Let $y \equiv a - \ln x_0$ where x_0 is an approximation to $\exp(a)$; if it is a good approximation, then y is small. Then $\exp(a) = x_0 \exp(y)$. Expanding in y, we obtain

$$\exp(a) = x_0\left(1 + y + \frac{y^2}{2!} + \frac{y^3}{3!} + ...\right),$$

and if we truncate the series after k-th term, the new approximation will have k times the number of correct digits in x_0. It is easy to see that the above cubic iteration is a particular case with $k = 3$.

The optimal number of terms to take depends on the speed of the implementation of $\ln x$.

Method 4: linear reduction + Taylor series

In this method we reduce the argument x by subtracting an integer. Suppose $x > 1$, then take $n = \lfloor x \rfloor$ where n is an integer, so that $0 \le x - n < 1$. Then we can compute $\exp(x) = \exp(n)\exp(x - n)$ by using the Taylor series on the small number $x - n$. The integer power e^n is found from a precomputed value of e.

A refinement of this method is to subtract not only the integer part of x, but also the first few binary digits. We fix an integer $k \ge 1$ and precompute $b \equiv \exp\left(2^{-k}\right)$. Then we find the integer m such that $0 \le x - m \cdot 2^{-k} < 2^{-k}$. (The rational number $m \cdot 2^{-k}$ contains the integer part of x and the first k bits of x after the binary point.) Then we compute $\exp\left(x - m \cdot 2^{-k}\right)$ using the Taylor series and $\exp\left(m \cdot 2^{-k}\right) = b^m$ by the integer powering algorithm from the precomputed value of b.

The parameter k should be chosen to minimize the computational effort.

Method 5: binary rediction

This method is based on the binary splitting technique and is described in [Haible *et al.* 1998] with a reference to [Brent 1976]. The idea is to reduce the computation of $\exp(x)$ to the computation of $\exp(r_k)$ for some rational numbers r_0, r_1, r_2, ...

Take the binary decomposition of x of the following form,

$$x = x_0 + \sum_{k=0}^{N} u_k \cdot 2^{-2^k},$$

where x_0 and u_k are integers such that $|u_k| < 2^{2^{k-1}}$. Then define $r_k = u_k \cdot 2^{-2^k}$. Note that all r_k are rational numbers such that $|r_k| < 2^{-2^{k-1}}$. The exponentials $\exp(r_k)$ are computed using the binary splitting on the Taylor series. Finally,

$$\exp(x) = \exp(x_0)\exp(r_0)\exp(r_1)....$$

The cost of this method is $O(M(P \ln P)\ln P)$ operations.

Essentially the same method can be used to compute the complex exponential, $\exp(a + \imath b)$. This is slower but the asymptotic cost is the same.

Method 6: continued fraction

There is a continued fraction representation of the exponential function:

$$\exp(-x) = 1 - \cfrac{x}{1 + \cfrac{x}{2 - \cfrac{x}{3 + \cfrac{x}{2 - \cfrac{x}{5 + \cfrac{x}{2 - \ddots}}}}}}.$$

This fraction converges for all x, although the speed of convergence varies with the magnitude of x.

This method does not seem to provide a computational advantage compared with the other methods.

Which method to use

This remains to be seen.

5.5 Calculation of π

In Yacas, the constant π is computed by the library routine `Internal'Pi()` which uses the internal routine `MathPi` to compute the value to current precision `Builtin'Precision'Set()`. The result is stored in a global variable as a list of the form

{`precision, value`} where `precision` is the number of digits of π that have already been found and `value` is the multiple-precision value. This is done to avoid recalculating π if a precise enough value for it has already been found.

Efficient iterative algorithms for computing π with arbitrary precision have been recently developed by Brent, Salamin, Borwein and others. However, limitations of the current multiple-precision implementation in Yacas (compiled with the "internal" math option) make these advanced algorithms run slower because they require many more arbitrary-precision multiplications at each iteration.

The file `examples/pi.ys` implements several different algorithms that duplicate the functionality of `Internal'Pi()`. See [Gourdon *et al.* 2001] for more details of computations of π and generalizations of Newton-Raphson iteration.

Method 1: solve $\sin x = 0$

`PiMethod0()`, `PiMethod1()`, `PiMethod2()` are all based on a generalized Newton-Raphson method of solving equations.

Since π is a solution of $\sin x = 0$, one may start sufficiently close, e.g. at $x_0 = 3.14159265$ and iterate $x' = x - \tan x$. In fact it is faster to iterate $x' = x + \sin x$ which solves a different equation for π. `PiMethod0()` is the straightforward implementation of the latter iteration. A significant speed improvement is achieved by doing calculations at each iteration only with the precision of the root that we expect to get from that iteration. Any imprecision introduced by round-off will be automatically corrected at the next iteration.

If at some iteration $x = \pi + \epsilon$ for small ϵ, then from the Taylor expansion of $\sin x$ it follows that the value x' at the next iteration will differ from π by $O\left(\epsilon^3\right)$. Therefore, the number of correct digits triples at each iteration. If we know the number of correct digits of π in the initial approximation, we can decide in advance how many iterations to compute and what precision to use at each iteration.

The final speed-up in `PiMethod0()` is to avoid computing at unnecessarily high precision. This may happen if, for example, we need to evaluate 200 digits of π starting with 20 correct digits. After 2 iterations we would be calculating with 180 digits; the next iteration would have given us 540 digits but we only need 200, so the third iteration would be wasteful. This can be avoided by first computing π to just over 1/3 of the required precision, i.e. to 67 digits, and then executing the last iteration at full 200 digits. There is still a wasteful step when we would go from 60 digits to 67, but much less time would be wasted than in the calculation with 200 digits of precision.

Newton's method is based on approximating the function $f(x)$ by a straight line. One can achieve better approximation and therefore faster convergence to the root if one approximates the function with a polynomial curve of higher order. The routine `PiMethod1()` uses the iteration

$$x' = x + \sin x + \frac{1}{6}(\sin x)^3 + \frac{3}{40}(\sin x)^5 + \frac{5}{112}(\sin x)^7$$

which has a faster convergence, giving 9 times as many digits at every iteration. (The series is the Taylor series for $\arcsin y$ cut at $O\left(y^9\right)$.) The same speed-up tricks are used as in `PiMethod0()`. In addition, the last iteration, which must be done at full precision, is performed with the simpler iteration $x' = x + \sin x$ to reduce the number of high-precision multiplications.

Both `PiMethod0()` and `PiMethod1()` require a computation of $\sin x$ at every iteration. An industrial-strength arbitrary precision library such as `gmp` can multiply numbers much faster than it can evaluate a trigonometric function. Therefore, it would be good to have a method which does not require trigonometrics.

`PiMethod2()` is a simple attempt to remedy the problem. It computes the Taylor series for $\arctan x$,

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + ...,$$

for the value of x obtained as the tangent of the initial guess for π; in other words, if $x = \pi + \epsilon$ where ϵ is small, then $\tan x = \tan \epsilon$, therefore $\epsilon = \arctan \tan x$ and π is found as $\pi = x - \epsilon$. If the initial guess is good (i.e. ϵ is very small), then the Taylor series for $\arctan x$ converges very quickly (although linearly, i.e. it gives a fixed number of digits of π per term). Only a single full-precision evaluation of $\tan x$ is necessary at the beginning of the algorithm. The complexity of this algorithm is proportional to the number of digits and to the time of a long multiplication.

Method 2: Borwein's iteration

The routines `PiBrentSalamin()` and `PiBorwein()` are based on much more advanced mathematics. (See papers by P. Borwein for review and explanations of the methods.) These methods do not require evaluations of trigonometric functions, but they do require taking a few square roots at each iteration, and all calculations must be done using full precision. Using modern algorithms, one can compute a square root roughly in the same time as a division; but Yacas's internal math is not yet up to it. Therefore, these two routines perform poorly compared to the more simple-minded `PiMethod0()`.

Method 3: AGM sequence (Brent-Salamin)

The algorithm of Brent and Salamin uses the AGM sequence. The calculation can be summarized as follows:

$$\left(a = 1, b = \frac{1}{\sqrt{2}}, c = \frac{1}{2}, k = 1\right)$$

`While(Not enough precision) [`

$$(a, b) = \left(\frac{a+b}{2}, \sqrt{ab}\right)$$

$$c = c - 2^k\left(a^2 - b^2\right)$$

$$\pi = 2\frac{a^2}{c}$$

$$k = k + 1$$

`];`

At each iteration, the variable π will have twice as many correct digits as it had at the previous iteration.

Method 4: Ramanujan's series

Another method for fast computation of π is based on the following mysterious series,

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

where $A = 13591409$, $B = 545140134$, and $C = 640320$. This formula was found by the Chudnovsky brothers, but it traces back to Ramanujan's notebooks.

To obtain the value of π with P decimal digits, one needs to take

$$n \approx P \frac{\ln 10}{3 \ln \frac{C}{12}} < \frac{479}{6793} P$$

terms of the series.

If this series is evaluated using Horner's scheme (the routine `PiChudnovsky`), then about $\frac{\ln n}{\ln 10}$ extra digits are needed to compensate for round-off error while adding n terms. This method does not require any long multiplications and costs $O\left(P^2\right)$ operations.

A potentially much faster way to evaluate this series at high precision is by using the binary splitting technique. This would give the asymptotic cost $O\left(M\left(P\ln P\right)\ln P\right)$.

Which method to use

This remains to be seen.

5.6 Trigonometric functions

Trigonometric functions $\sin x$, $\cos x$ are computed by subtracting 2π from x until it is in the range $0 < x < 2\pi$ and then using the Taylor series. (The value of π is precomputed.)

Tangent is computed by dividing $\frac{\sin x}{\cos x}$ or from $\sin x$ using the identity

$$\tan x = \frac{\sin x}{\sqrt{1-(\sin x)^2}}.$$

Method 1: Taylor series

The Taylor series for the basic trigonometric functions are

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...,$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

These series converge for all x but are optimal for multiple-precision calculations only for small x. The convergence rate and possible optimizations are the same as those of the Taylor series for $\exp(x)$.

Method 2: argument reduction

Basic argument reduction requires a precomputed value for $\frac{\pi}{2}$. The identities $\sin\left(x+\frac{\pi}{2}\right) = \cos x$, $\cos\left(x+\frac{\pi}{2}\right) = -\sin x$ can be used to reduce the argument to the range between 0 and $\frac{\pi}{2}$. Then the bisection for $\cos x$ and the trisection for $\sin x$ are used.

For $\cos x$, the bisection identity can be used more efficiently if it is written as

$$1 - \cos 2x = 4\left(1-\cos x\right) - 2\left(1-\cos x\right)^2.$$

If $1-\cos x$ is very small, then this decomposition allows to use a shorter multiplication and reduces round-off error.

For $\sin x$, the trisection identity is

$$\sin 3x = 3\sin x - 4\left(\sin x\right)^3.$$

The optimal number of bisections or trisections should be estimated to reduce the total computational cost. The resulting number will depend on the magnitude of the argument x, on the required precision P, and on the speed of the available multiplication $M(P)$.

Method 3: inverse $\arctan x$

The function $\arctan x$ can be found from its Taylor series, or from the complex AGM method, or by another method. Then the function can be inverted by Newton's iteration to obtain $\tan x$ and from it also $\sin x$, $\cos x$ using the trigonometric identities.

Alternatively, $\arcsin x$ may be found from the Taylor series and inverted to obtain $\sin x$.

This method seems to be of marginal value since efficient direct methods for $\cos x$, $\sin x$ are available.

Which method to use

This remains to be seen.

5.7 Inverse trigonometric functions

Inverse trigonometric functions are computed by various methods. To compute $y = \arcsin x$, Newton's method is used for to invert $x = \sin y$. The inverse tangent $\arctan x$ can be computed by its Taylor series,

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - ...,$$

or by the continued fraction expansion,

$$\arctan x = \cfrac{x}{1+\cfrac{x^2}{3+\cfrac{(2x)^2}{5+\cfrac{(3x)^2}{7+\ldots}}}}.$$

The convergence of this expansion for large $|x|$ is improved by using the identities

$$\arctan x = \frac{\pi}{2}\mathrm{Sign}\left(x\right) - \arctan\frac{1}{x},$$

$$\arctan x = 2\arctan\frac{x}{1+\sqrt{1+x^2}}.$$

Thus, any value of x is reduced to $|x| < 0.42$. This is implemented in the standard library scripts.

By the identity $\arccos x \equiv \frac{\pi}{2} - \arcsin x$, the inverse cosine is reduced to the inverse sine. Newton's method for $\arcsin x$ consists of solving the equation $\sin y = x$ for y. Implementation is similar to the calculation of π in `PiMethod0()`.

For x close to 1, Newton's method for $\arcsin x$ converges very slowly. An identity

$$\arcsin x = \mathrm{Sign}\left(x\right)\left(\frac{\pi}{2} - \arcsin\sqrt{1-x^2}\right)$$

can be used in this case. Another potentially useful identity is

$$\arcsin x = 2\arcsin\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}.$$

Inverse tangent can also be related to inverse sine by

$$\arctan x = \arcsin\frac{x}{\sqrt{1+x^2}},$$

$$\arctan\frac{1}{x} = \arcsin\frac{1}{\sqrt{1+x^2}}.$$

Alternatively, the Taylor series can be used for the inverse sine:

$$\arcsin x = x + \frac{1}{2}\frac{x^3}{3} + \frac{1\cdot 3}{2\cdot 4}\frac{x^5}{5} + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\frac{x^7}{7} + ...$$

An everywhere convergent continued fraction can be used for the tangent:

$$\tan x = \cfrac{x}{1-\cfrac{x^2}{3-\cfrac{x^2}{5-\cfrac{x^2}{7-\ldots}}}}.$$

Hyperbolic and inverse hyperbolic functions are reduced to exponentials and logarithms: $\cosh x = \frac{1}{2}(\exp(x) + \exp(-x))$, $\sinh x = \frac{1}{2}(\exp(x) - \exp(-x))$, $\tanh x = \frac{\sinh x}{\cosh x}$,

$$\operatorname{arccosh} x = \ln\left(x + \sqrt{x^2 - 1}\right),$$

$$\operatorname{arcsinh} x = \ln\left(x + \sqrt{x^2 + 1}\right),$$

$$\operatorname{arctanh} x = \frac{1}{2}\ln\frac{1+x}{1-x}.$$

Continued fraction for $\arctan x$

The idea to use continued fraction expansions for `ArcTan` comes from the book [Crenshaw 2000]. In that book the author explains how he got the idea to use continued fraction expansions to approximate $\arctan x$, given that the Taylor series converges slowly, and having a hunch that in that case the continued fraction expansion then converges rapidly. He then proceeds to show that in the case of $\arctan x$, this advantage is very significant. However, it might not be true for all slowly converging series.

The convergence of the continued fraction expansion of $\arctan x$ is indeed better than convergence of the Taylor series. Namely, the Taylor series converges only for $|x| < 1$ while the continued fraction converges for all x. However, the speed of its convergence is not uniform in x; the larger the value of x, the slower the convergence. The necessary number of terms of the continued fraction is in any case proportional to the required number of digits of precision, but the constant of proportionality depends on x.

This can be understood by the following argument. The difference between two partial continued fractions that differ only by one extra last term can be estimated as

$$|\delta| \equiv \left| \frac{b_0}{a_1 + \frac{b_1}{\ldots + \frac{b_{n-1}}{a_n}}} - \frac{b_0}{a_1 + \frac{b_1}{\ldots + \frac{b_n}{a_{n+1}}}} \right| < \frac{b_0 b_1 \ldots b_n}{(a_1 \ldots a_n)^2 a_{n+1}}.$$

(This is a conservative estimate that could be improved with more careful analysis. See also the section on numerical continued fractions.) For the above continued fraction for $\arctan x$, this directly gives the following estimate,

$$|\delta| < \frac{x^{2n+1}(n!)^2}{(2n+1)((2n-1)!!)^2} \approx \pi\left(\frac{x}{2}\right)^{2n+1}.$$

This formula only gives a meaningful bound if $x < 2$, but it is clear that the precision generally becomes worse when x grows. If we need P digits of precision, then, for a given x, the number of terms n has to be large enough so that the relative precision is sufficient, i.e.

$$\frac{\delta}{\arctan x} < 10^{-P}.$$

This gives $n > \frac{P \ln 10}{\ln 4 - 2\ln x}$ and for $x = 1$, $n > \frac{3}{2}P$. This estimate is very close for small x and only slightly suboptimal for larger x: numerical experimentation shows that for $x \leq 1$, the required number of terms for P decimal digits is only about $\frac{4}{3}P$, and for $x \leq 0.42$, n must be about $\frac{3}{4}P$. If $x < 1$ is very small then one needs a much smaller number of terms $n > P\frac{\ln 10}{\ln 4 - 2\ln x}$. Round-off errors may actually make the result less precise if we use many more terms than needed.

If we compare the rate of convergence of the continued fraction for $\arctan x$ with the Taylor series

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

we find that the number of terms of the Taylor series needed for P digits is about $n > \frac{P \ln 10}{2\ln x}$. Since the range of x can be reduced to about $[0, 0.42]$ by trigonometric identities, the difference between this and $P\frac{\ln 10}{\ln 4 - 2\ln x}$ is never very large. At most twice as many terms n are needed in the Taylor series as in the continued fraction. However, a Taylor series can be evaluated efficiently using $O\left(\sqrt{n}\right)$ long multiplications, while a continued fraction with n terms always requires n divisions. Therefore, at high enough precision the continued fraction method will be much less efficient than the Taylor series.

Which method to use

This remains to be seen.

5.8 Factorials and binomial coefficients

The factorial is defined by $n! \equiv n(n-1)\ldots\cdot 1$ for integer $n \geq 1$ and the binomial coefficient is defined by

$$\binom{n}{m} \equiv \frac{n!}{m!(n-m)!}.$$

The "double factorial" $n!! \equiv n(n-2)(n-4)\ldots$ is also useful for some calculations. For convenience, one defines $0! \equiv 1$, $0!! \equiv 1$, and $(-1)!! \equiv 1$; with these definitions, the recurrence relations

$$n!(n+1) = (n+1)!,$$

$$n!!(n+2) = (n+2)!!$$

are valid also for $n = 0$ and $n = -1$.

There are two tasks related to the factorial: the exact integer calculation and an approximate calculation to some floating-point precision. Factorial of n has approximately $n\frac{\ln n}{\ln 10}$ decimal digits, so an exact calculation is practical only for relatively small n. In the current implementation, exact factorials for $n > 65535$ are not computed but print an error message advising the user to avoid exact computations of large factorials. For example, `Internal'LnGammaNum(n+1)` is able to compute $\ln n!$ for very large n to any desired floating-point precision.

Exact factorials

To compute factorials exactly, we use two direct methods. The first method is to multiply the numbers 1, 2, ..., n in a loop. This method requires n multiplications of short numbers with P-digit numbers, where $P = O(n \ln n)$ is the number of digits in $n!$. Therefore its complexity is $O\left(n^2 \ln n\right)$. This factorial routine is implemented in the Yacas core with a small speedup: consecutive pairs of integers are first multiplied together using platform math and then multiplied by the accumulator product.

A second method uses a binary tree arrangement of the numbers 1, 2, ..., n similar to the recursive sorting routine ("merge-sort"). If we denote by `a *** b` the "partial factorial" product $a(a+1)\ldots(b-1)b$, then the tree-factorial algorithm consists of replacing $n!$ by $1***n$ and recursively evaluating $(1***m)((m+1)***n)$ for some integer m near $\frac{n}{2}$. The partial factorials of nearby numbers such as $m***(m+2)$ are evaluated explicitly. The binary tree algorithm requires one multiplication of $\frac{P}{2}$ digit integers at the last step, two $\frac{P}{4}$ digit multiplications at the last-but-one step and so on. There are $O(\ln n)$ total steps of the recursion. If the cost of multiplication is $M(P) = P^{1+a}(\ln P)^b$, then one can show that the total cost of the binary tree algorithm is $O(M(P))$ if $a > 0$ and

$O(M(P)\ln n)$ if $a = 0$ (which is the best asymptotic multiplication algorithm).

Therefore, the tree method wins over the simple method if the cost of multiplication is lower than quadratic.

The tree method can also be used to compute "double factorials" ($n!!$). This is faster than to use the identities

$$(2n)!! = 2^n n!$$

and

$$(2n-1)!! = \frac{(2n)!}{2^n n!}.$$

Double factorials are used, for instance, in the exact calculation of the Gamma function of half-integer arguments.

Binomial coefficients $\binom{n}{m}$ are found by first selecting the smaller of m, $n-m$ and using the identity $\binom{n}{m} = \binom{n}{n-m}$. Then a partial factorial is used to compute

$$\binom{n}{m} = \frac{(n-m+1)***n}{m!}.$$

This is always much faster than computing the three factorials in the definition of $\binom{n}{m}$.

Approximate factorials

A floating-point computation of the factorial may proceed either via Euler's Gamma function, $n! = \Gamma(n+1)$, or by a direct method (multiplying the integers and maintaining a certain floating-point precision). If the required precision is much less than the number of digits in the exact factorial, then almost all multiplications will be truncated to the precision P and the tree method $O(nM(P))$ is always slower than the simple method $O(nP)$.

Which method to use

This remains to be seen.

5.9 Classical orthogonal polynomials: general case

A family of orthogonal polynomials is a sequence of polynomials $q_n(x)$, $n = 0, 1, ...$ that satisfy the orthogonality condition on some interval $[a, b]$ with respect to some weight function $\rho(x)$:

$$\int_a^b q_m(x)\, q_n(x)\, \rho(x)\, dx = 0$$

for $m \neq n$. The interval $[a, b]$ can be finite or infinite and the weight function must be real and non-negative on this interval.

In principle, one could choose any (non-negative) weight function $\rho(x)$ and any interval $[a, b]$ and construct the corresponding family of orthogonal polynomials $q_n(x)$. For example, take $q_0 = 1$, then take $q_1 = x + c$ with unknown c and find such c that q_0 and q_1 satisfy the orthogonality condition; this requires solving a linear equation. Then we can similarly find two unknown coefficients of q_2 and so on. (This is called the Gramm-Schmidt orthogonalization procedure.)

But of course not all weight functions $\rho(x)$ and not all intervals $[a, b]$ are equally interesting. There are several "classical" families of orthogonal polynomials that have been of use to theoretical and applied science. The "classical" polynomials are always solutions of a simple second-order differential equation and are always a specific case of some hypergeometric function.

The construction of "classical" polynomials can be described by the following scheme. The function $\rho(x)$ must satisfy the so-called Pearson's equation,

$$\frac{1}{\rho(x)}\left(\frac{d}{dx}\rho(x)\right) = \frac{\alpha(x)}{\beta(x)},$$

where the functions α, β are of the form

$$\alpha(x) = \alpha_0 + \alpha_1 x,$$

$$\beta(x) = \beta_0 + \beta_1 x + \beta_2 x^2.$$

Also, the following boundary conditions must be satisfied at both ends a, b of the interval,

$$\rho(a)\,\beta(a) = \rho(b)\,\beta(b) = 0.$$

If the function $\rho(x)$ and the interval $[a, b]$ are chosen in this way, then the corresponding orthogonal polynomials $q_n(x)$ are solutions of the differential equation

$$\frac{\partial}{\partial x}\left(\beta(x)\,\rho(x)\left(\frac{\partial}{\partial x}q_n(x)\right)\right) - n\left(\alpha_1 + (n+1)\,\beta_2\right) q_n = 0.$$

The polynomials $q_n(x)$ are also given by the Rodrigues formula,

$$q_n(x) = \frac{A_n}{\rho(x)}\left(\frac{\partial^n}{\partial x^n}\left(\rho(x)\,\beta(x)^n\right)\right),$$

where A_n is a normalization constant. It is usual to normalize the polynomials so that

$$\int_a^b q_n(x)^2\, \rho(x)\, dx = 1.$$

The normalization constant A_n must be chosen accordingly.

Finally, there is a formula for the generating function of the polynomials,

$$G(x, w) = \frac{1}{\rho(x)}\frac{\rho(t(x,w))}{|1 - w(\beta_1 + 2\beta_2 t(x,w))|},$$

where $t(x, w)$ is the root of $t - x - w\beta(t) = 0$ which is nearest to $t = x$ at small w. This function $G(x, w)$ gives the unnormalized polynomials,

$$G(x, w) = \sum_{n=0}^{\infty} \frac{q_n(x)}{n!} w^n,$$

where

$$q_n(x) = \frac{1}{\rho(x)}\left(\frac{\partial^n}{\partial x^n}\left(\rho(x)\,\beta(x)^n\right)\right).$$

The classical families of (normalized) orthogonal polynomials are obtained in this framework with the following definitions:

- The Legendre polynomials $P_n(x)$: $a = -1$, $b = 1$, $\rho(x) = 1$, $\alpha(x) = 0$, $\beta(x) = 1 - x^2$, $A_n = \frac{(-1)^n}{2^n n!}$.
- The Laguerre polynomials $(L^m)_n(x)$: $a = 0$, $b = +\infty$, $\rho(x) = x^m \exp(-x)$, (here $m > -1$ or else the weight function is not integrable on the interval), $\alpha(x) = m - x$, $\beta(x) = x$, $A_n = 1$.
- The Hermite polynomials $H_n(x)$: $a = -\infty$, $b = +\infty$, $\rho(x) = \exp\left(-x^2\right)$, $\alpha(x) = -2x$, $\beta(x) = 1$, $A_n = (-1)^n$.
- The Chebyshev polynomials of the first kind $T_n(x)$: $a = -1$, $b = 1$, $\rho(x) = \frac{1}{\sqrt{1-x^2}}$, $\alpha(x) = x$, $\beta(x) = 1 - x^2$, $A_n = \frac{(-1)^n}{(2n)!!}$.

The Rodrigues formula or the generating function are not efficient ways to calculate the polynomials. A better way is to use linear recurrence relations connecting q_{n+1} with q_n and q_{n-1}. (These recurrence relations can also be written out in full generality through $\alpha(x)$ and $\beta(x)$ but we shall save the space.)

There are three computational tasks related to orthogonal polynomials:

1. Compute all coefficients of a given polynomial $q_n(x)$ exactly. The coefficients are rational numbers, but their numerators and denominators usually grow exponentially quickly with n, so that at least some coefficients of the n-th polynomial are n-digit numbers. The array of coefficients can be obtained using recurrence relations n times. The required number of operations is proportional to n^2 (because we need n coefficients and we have n recurrence relations for each of them) and to the multiplication time of n-digit integers, i.e. $O\left(n^2 M(n)\right)$. Sometimes an exact formula for the coefficients is available (this is the case for the four "classical" families of polynomials above; see the next section). Then the computation time dramatically drops down to $O\left(n^2\right)$ because no recurrences are needed.
2. Compute the numerical value of a given polynomial $q_n(x)$ at given x, either exactly (at rational x) or approximately in floating point. This requires $O(nM(n))$ operations for exact computation and $O(nM(P))$ operations in P-digit floating point.
3. Compute a series of orthogonal polynomials with given coefficients f_n, i.e. $f(x) \equiv \sum_{n=0}^{N} f_n q_n(x)$, at a given x. This task does not actually require computing the polynomials first, if we use the so-called Clenshaw-Smith procedure which gives the value of $f(x)$ directly in n iterations. (See *below*, in *The Yacas book of algorithms, Chapter 4, Section 10* for more explanations.) The number of operations is again $O(nM(P))$.

In the next section we shall give some formulae that allow to calculate particular polynomials more efficiently.

5.10 Classical orthogonal polynomials: special cases

The fastest algorithm available is for Chebyshev (sometimes spelled Tschebyscheff) polynomials $T_n(x)$, $U_n(x)$. The following recurrence relations can be used:

$$T_{2n}(x) = 2\left(T_n(x)\right)^2 - 1,$$

$$T_{2n+1}(x) = 2T_{n+1}(x)\,T_n(x) - x,$$
$$U_{2n}(x) = 2T_n(x)\,U_n(x) - 1,$$
$$U_{2n+1}(x) = 2T_{n+1}(x)\,U_n(x).$$

This allows to compute $T_n(x)$ and $U_n(x)$ in time logarithmic in n.

There is a way to implement this method without recursion. The idea is to build the sequence of numbers n_1, n_2, ... that are needed to compute $T_n(x)$.

For example, to compute $T_{19}(x)$ using the second recurrence relation, we need $T_{10}(x)$ and $T_9(x)$. We can write this chain symbolically as $19 \sim c(9,10)$. For $T_{10}(x)$ we need only $T_5(x)$. This we can write as $10 \sim c(5)$. Similarly we find: $9 \sim c(4,5)$. Therefore, we can find both $T_9(x)$ and $T_{10}(x)$ if we know $T_4(x)$ and $T_5(x)$. Eventually we find the following chain of pairs:

$$19 \sim c(9,10) \sim c(4,5) \sim c(2,3) \sim c(1,2) \sim c(1).$$

Therefore, we find that $T_{19}(x)$ requires to compute $T_k(x)$ sequentially for all k that appear in this chain (1,2,3,4,5,9,10).

There are about $2\frac{\ln n}{\ln 2}$ elements in the chain that leads to the number n. We can generate this chain in a straightforward way by examining the bits in the binary representation of n. Therefore, we find that this method requires no storage and time logarithmic in n. A recursive routine would also take logarithmic time but require logarithmic storage space.

Note that using these recurrence relations we do not obtain any individual coefficients of the Chebyshev polynomials. This method does not seem very useful for symbolic calculations (with symbolic x), because the resulting expressions are rather complicated combinations of nested products. It is difficult to expand such an expression into powers of x or manipulate it in any other way, except compute a numerical value. However, these fast recurrences are numerically unstable, so numerical values need to be evaluated with extended working precision. Currently this method is not used in Yacas, despite its speed.

An alternative method for very large n would be to use the identities

$$T_n(x) = \cos n \arccos x,$$

$$U_n(x) = \frac{\sin(n+1)\arccos x}{\sqrt{1-x^2}}.$$

The computation will require an extended-precision evaluation of $\arccos x$.

Coefficients for Legendre, Hermite, Laguerre, Chebyshev polynomials can be obtained by explicit formulae. This is faster than using recurrences if we need the entire polynomial symbolically, but still slower than the recurrences for numerical calculations.

- Chebyshev polynomials of the first and of the second kind: If $k = \left\lfloor \frac{n}{2} \right\rfloor$, then

$$T_n(x) = \sum_{i=0}^{k} (-1)^i \frac{(2x)^{n-2i}}{n-i} \binom{n-i}{i},$$

$$U_n(x) = \sum_{i=0}^{k} (-1)^i (2x)^{n-2i} \binom{n-i}{i}.$$

  Here it is assumed that $n > 0$ (the case $n = 0$ must be done separately). The summation is over integer values of i such that $0 \le 2i \le n$, regardless of whether n is even or odd.

- Hermite polynomials: For even $n = 2k$ where $k \ge 0$,

$$H_n(x) = (-2)^k (n-1)!! \sum_{i=0}^{k} \frac{x^{2i}(-4)^i k!}{(2i)!\,(k-i)!},$$

  and for odd $n = 2k+1$ where $k \ge 0$,

$$H_n(x) = 2(-2)^k n!! \sum_{i=0}^{k} \frac{x^{2i+1}(-4)^i k!}{(2i+1)!\,(k-i)!}.$$

- Legendre polynomials: If $k = \left\lfloor \frac{n}{2} \right\rfloor$, then

$$P_n(x) = 2^{-n} \sum_{i=0}^{k} (-1)^i x^{n-2i} \binom{n}{i} \binom{2n-2i}{n}.$$

  The summation is over integer values of i such that $0 \le 2i \le n$, regardless of whether n is even or odd.

- Laguerre polynomials:

$$\mathrm{OrthoL}\left(n,a,x\right)=\sum_{i=0}^{n}\frac{\left(-x\right)^{i}}{i!}\binom{n+a}{n-i}.$$

Here the parameter a might be non-integer. So the binomial coefficient must be defined for non-integer a through the Gamma function instead of factorials, which gives

$$\binom{n+a}{n-i}=\frac{\left(n+a\right)...\left(i+1+a\right)}{\left(n-i\right)!}.$$

The result is a rational function of a.

In all formulae for the coefficients, there is no need to compute factorials every time: the next coefficient can be obtained from the previous one by a few short multiplications and divisions. Therefore this computation costs $O\left(n^{2}\right)$ short operations.

5.11 Series of orthogonal polynomials

If we need to compute a series of orthogonal polynomials with given coefficients f_n, i.e.

$$f\left(x\right)\equiv\sum_{n=0}^{N}f_{n}q_{n}\left(x\right)$$

at a given x, we do not need to compute the orthogonal polynomials separately. The Clenshaw-Smith recurrence procedure allows to compute the value of the sum directly.

Suppose a family of functions $q_n\left(x\right)$, $n=0$, 1, ... satisfies known recurrence relations of the form

$$q_{n}=A_{n}\left(x\right)q_{n-1}+B_{n}\left(x\right)q_{n-2},$$

where $A_n\left(x\right)$ and $B_n\left(x\right)$ are some known functions and $q_0\left(x\right)$ and $q_1\left(x\right)$ are known.

The procedure goes as follows [Luke 1975]. First, for convenience, we define $q_{-1}\equiv 0$ and the coefficient $A_1\left(x\right)$ so that $q_1=A_1q_0$. This allows us to use the above recurrence relation formally also at $n=1$. Then, we take the array of coefficients f_n and define a backward recurrence relation

$$\mathrm{X}_{N+1}=\mathrm{X}_{N+2}=0,$$

$$\mathrm{X}_{n}=A_{n+1}\mathrm{X}_{n+1}+B_{n+2}\mathrm{X}_{n+2}+f_{n},$$

where $n=N$, $N-1$, ..., 0. (Note that here we have used the artificially defined coefficient A_1.) Magically, the value we are looking for is given by

$$f\left(x\right)=\mathrm{X}_{0}q_{0}.$$

This happens because we can express

$$f_{n}=\mathrm{X}_{n}-A_{n+1}\mathrm{X}_{n+1}-B_{n+2}\mathrm{X}_{n+2},$$

for $n=0$, 1, ..., N, regroup the terms in the sum

$$f\left(x\right)\equiv\sum_{n=0}^{N}f_{n}q_{n}\left(x\right)$$

to collect X_n, obtain

$$f\left(x\right)=\sum_{n=0}^{N}\mathrm{X}_{n}q_{n}-\sum_{n=1}^{N}\mathrm{X}_{n}A_{n}q_{n-1}-\sum_{n=2}^{N}\mathrm{X}_{n}B_{n}q_{n-2},$$

and finally

$$f\left(x\right)=\mathrm{X}_{0}q_{0}+\mathrm{X}_{1}\left(q_{1}-A_{1}q_{1}\right)$$
$$+\sum_{n=2}^{N}\mathrm{X}_{n}\left(q_{n}-A_{n}q_{n-1}-B_{n}q_{n-2}\right)=\mathrm{X}_{0}q_{0}.$$

The book [Luke 1975] warns that the recurrence relation for X_n is not always numerically stable.

Note that in the book there seems to be some confusion as to how the coefficient A_1 is defined. (It is not defined explicitly there.) Our final formula differs from the formula in [Luke 1975] for this reason.

The Clenshaw-Smith procedure is analogous to the Horner scheme of calculating polynomials. This procedure can also be generalized for linear recurrence relations having more than two terms. The functions $q_0\left(x\right)$, $q_1\left(x\right)$, $A_n\left(x\right)$, and $B_n\left(x\right)$ do not actually have to be polynomials for this to work.

Chapter 6

Numerical algorithms III: special functions

6.1 Euler's Gamma function

Euler's Gamma function $\Gamma(z)$ is defined for complex z such that $Re(z) > 0$ by the integral

$$\Gamma(z) \equiv \int_0^\infty \exp(-t)\, t^{z-1} dz.$$

The Gamma function satisfies several identities that can be proved by rearranging this integral; for example, $\Gamma(z+1) = z\Gamma(z)$. This identity defines $\Gamma(z)$ for all complex z. The Gamma function is regular everywhere except nonpositive integers (0, −1, −2, ...) where it diverges.

Special arguments

For real integers $n > 0$, the Gamma function is the same as the factorial,

$$\Gamma(n+1) \equiv n!,$$

so the factorial notation can be used for the Gamma function too. Some formulae become a little simpler when written in factorials.

The Gamma function is implemented as `Gamma(x)`. At integer values `n` of the argument, `Gamma(n)` is computed exactly. Because of overflow, it only makes sense to compute exact integer factorials for small numbers n. Currently a warning message is printed if a factorial of $n > 65535$ is requested.

For half-integer arguments $\Gamma(x)$ is also computed exactly, using the following identities (here n is a nonnegative integer and we use the factorial notation):

$$\left(+\frac{2n+1}{2}\right)! = \sqrt{\pi}\frac{(2n+1)!}{2^{2n+1}n!},$$

$$\left(-\frac{2n+1}{2}\right)! = (-1)^n\sqrt{\pi}\frac{2^{2n}n!}{(2n)!}.$$

For efficiency, "double factorials" are used in this calculation. The "double factorial" is defined as $n!! = n(n-2)...$ and satisfies the identities

$$(2n-1)!! = \frac{(2n)!}{2^n n!},$$

$$(2n)!! = 2^n n!.$$

For convenience, one defines $0! = 1$, $0!! = 1$, $(-1)!! = 1$.

If the factorial of a large integer or half-integer n needs to be computed not exactly but only with a certain floating-point precision, it is faster (for large enough $|n|$) not to evaluate an exact integer product, but to use the floating-point numerical approximation. This method is currently not implemented in Yacas.

There is also the famous Stirling's asymptotic formula for large factorials,

$$\ln n! \approx \frac{\ln 2\pi n}{2} + n\ln\frac{n}{e} + \frac{1}{12n} - \frac{1}{360n^3} + ...$$

An analogous formula for double factorials can be easily found.

Method 0: power series (rational arguments)

For "small" rational arguments, i.e. for numbers $s = \frac{p}{q}$ where p, q are small integers, there is a faster method for computing $\Gamma(s)$. This method was used in [Smith 2001]. [Haible *et al.* 1998] give this method in conjunction with the binary splitting technique.

Repeated partial integration gives the expansion

$$\Gamma(s) = M^s \exp(-M) \sum_{n=0}^{\infty} \frac{M^n}{s(s+1)...(s+n)} + \int_M^\infty u^{s-1}\exp(-u)\, du.$$

Suppose that $1 \le s \le 2$. Then the remainder is given by the last integral that is smaller than $M\exp(-M)$. By choosing M a large enough integer, the remainder can be made small enough to discard it. Then we obtain a series of rational numbers that can be evaluated directly in $O\left(P^2\right)$ time or using the binary splitting technique in $O(M(P\ln P)\ln P)$ time. This method is currently not implemented in Yacas.

Method 1: the Lanczos-Spouge formula

For arbitrary complex arguments with nonnegative real part, the function `Internal'GammaNum(x)` computes a uniform appoximation of Lanczos [Lanczos 1964] modified by Spouge [Spouge 1994]. (See also [Godfrey 2001] for some more explanations.) This function is also triggered when in numeric mode, eg. when calling `N(Gamma(x))`, which is the preferred method for end users.

The method gives the Γ-function only for arguments with positive real part; at negative values of the real part of the argument, the Γ-function is computed via the identity

$$\Gamma(x)\Gamma(1-x) = \frac{\pi}{\sin\pi x}.$$

The Lanczos-Spouge approximation formula depends on a parameter a,

$$\Gamma(z) = \frac{\sqrt{2\pi}(z+a)^{z+\frac{1}{2}}}{ze^{z+a}}\left(1 + \frac{e^{a-1}}{\sqrt{2\pi}}\sum_{k=1}^{N}\frac{c_k}{z+k}\right),$$

with $N \equiv \lceil a \rceil - 1$. The coefficients c_k are defined by

$$c_k = (-1)^{k-1} \frac{(a-k)^{k-\frac{1}{2}}}{e^{k-1}\,(k-1)!}.$$

The parameter a is a free parameter of the approximation that determines also the number of terms in the sum. Some choices of a may lead to a slightly more precise approximation, but larger a is always better. The number of terms N must be large enough to produce the required precision. The estimate of the relative error for this formula is valid for all z such that $Re\,(z) > 0$ and is

$$\text{error} < \frac{(2\pi)^{-a}}{\sqrt{2\pi a}} \frac{a}{a+z}.$$

The lowest value of a to produce P correct digits is estimated as

$$a = \left(P - \frac{\ln P}{\ln 10}\right) \frac{\ln 10}{\ln 2\pi} - \frac{1}{2}.$$

In practical calculations, the integer logarithm routine `IntLog` is used and the constant $\frac{\ln 10}{\ln 2\pi}$ is approximated from above by $659/526$, so that a is not underestimated.

The coefficients c_k and the parameter a can be chosen to achieve a greater precision of the approximation formula. However, the recipe for the coefficients c_k given in the paper by Lanczos is too complicated for practical calculations in arbitrary precision: the time it would take to compute the array of N coefficients c_k grows as N^3. Therefore it is better to use less precise but much simpler formulae derived by Spouge.

Round-off error in the Lanczos method

In the calculation of the sum $S \equiv \sum_{k=1}^{N} c_k\,(z+k)^{-1}$, a certain round-off error results from the changing signs in c_k, and from the fact that some values c_k are much larger than the final value of the sum. This leads to some cancellations and as a result to a certain loss of precision.

At version 1.3.6, Yacas is limited in its internal arbitrary precision facility that does not support true floating-point computation but rather uses fixed-point logic; this hinders precise calculations with floating-point numbers. In the current version of the `Internal'GammaNum()` function, two workarounds are implemented. First, a Horner scheme is used to compute the sum; this is somewhat faster and leads to smaller round-off errors. Second, intermediate calculations are performed at 40% higher precision than requested. This is much slower but allows to obtain results at desired precision.

If strict floating-point logic is used, the working precision necessary to compensate for the cancellations must be $1.1515P$ digits for P digits of the result. This can be shown as follows.

The sum converges to a certain value S which is related to the correct value of the Gamma function at z. After some algebra we find that S is of order $\sqrt{a}$ if $z > a$ and of order $a^{\frac{1}{2}-z}$ if $a > z$. Since a is never a very large number, we can consider the value of S to be roughly of order 1, compared with exponentially large values of some of the terms c_k of this sum. The magnitude of a coefficient c_k is estimated by Stirling's formula,

$$\ln |c_k| \approx (k-1) \ln \frac{a-k}{k-1}.$$

(Here we have put approximately $k - 1 \approx k - \frac{1}{2}$ since k is not small.) This function has a maximum at $k \approx 1 + 0.2178\,(a-1)$. Then the largest magnitude of a coefficient c_k at this k is approximately $\exp\left(0.2785\,(a-1)\right)$. The constant $0.2785 = W\left(\frac{1}{e}\right)$ is the solution of $x + \ln x + 1 = 0$. Here $x = \frac{k-1}{a-k}$ and W is Lambert's function. Now, $a - 1 \approx P\frac{\ln 10}{\ln 2\pi}$, so the maximum of c_k is $10^{0.1515P}$. Therefore we have a certain cancellation in the sum S: adding and subtracting some numbers of order $10^{0.1515P}$ produces an answer of order 1. Therefore we need to have at least 15(The constant $\frac{W\left(\frac{1}{e}\right)}{\ln 2\pi} \approx 0.1515$ is independent of the base 10.)

Other methods for the Gamma function

More traditional ways of calculating the Gamma function are the Stirling asymptotic series and the Sweeney-Brent method of combined series.

Method 2: Stirling's asymptotic series

The Stirling asymptotic series for the Gamma function is

$$\ln \Gamma(x) \sim \left(x - \frac{1}{2}\right) \ln x - x + \frac{1}{2} \ln 2\pi + \sum_{n=1}^{\infty} \frac{B_{2n}}{2n\,(2n-1)\,x^{2n-1}}.$$

This series is valid asymptotically for large $|x|$, also for complex values of x (but excluding the negative real values). Computation of this series up to $n = N$ requires finding the Bernoulli numbers B_n up to N.

For a given (large) value of $|x|$, the terms of this series decrease at first, but then start to grow. (Here x can be a complex number.) There exist estimates for the error term of the asymptotic series (see [Abramowitz *et al.* 1964], 6.1.42). Roughly, the error is of the order of the first discarded term.

We can estimate the magnitude of the terms using the asymptotic formula for the Bernoulli numbers (see below). After some algebra, we find that the value of n at which the series starts to grow and diverge is $n_0 \approx \pi\,|x| + 2$. Therefore at given x we can only use the asymptotic series up to the n_0-th term.

For example, if we take $x = 10$, then we find that the 32-nd term of the asymptotic series has the smallest magnitude (about 10^{-28}) but the following terms start to grow.

To be on the safe side, we should drop a few more terms from the series. Define the number of terms by $n_0 \equiv \pi\,|x|$. Then the order of magnitude of the n_0-th term is $\frac{\exp(-2\pi|x|)}{2\pi^2|x|}$. This should be compared with the magnitude of the sum of the series which is of order $|x \ln x|$. We find that the relative precision of P decimal digits or better is achieved if

$$(2\pi - 1)\,x + (x+1) \ln x + 3.9 > P \ln 10.$$

If (x,P) do not satisfy this inequality, the asymptotic method does not provide enough precision. For example, with $x = 10$ we obtain $P \leq 35$. So the asymptotic series gives the value of $\Gamma(10)$ with not more than 35 decimal digits.

For very large P, the inequality is satisfied when roughly $x > P\frac{\ln 10}{\ln P}$. Assuming that the Bernoulli numbers are precomputed, the complexity of this method is that of computing a Taylor series with n_0 terms, which is roughly $O\left(\sqrt{P}\right) M(P)$.

What if x is not large enough? Using the identity $x\Gamma(x) = \Gamma(x+1)$, we can reduce the computation of $\Gamma(x)$ to $\Gamma(x+M)$ for some integer M. Then we can choose M to be large enough so that the asymptotic series gives the required precision when evaluated at $x + M$. We shall have to divide the result M times by some long numbers to obtain $\Gamma(x)$. Therefore, the complexity of this method for given (x,P) is increased by $M(P)\left(P\frac{\ln 10}{\ln P} - x\right)$. For small x this will be the dominant contribution to the complexity.

On the other hand, if the Bernoulli numbers are not available precomputed, then their calculation dominates the complexity of the algorithm.

Method 3: the Sweeney-Brent trick

The second method for the Gamma function $\Gamma(x)$ was used in R. P. Brent's Fortran MP package [Brent 1978]. Brent refers to [Sweeney 1963] for the origin of this method. Therefore, we called this method the "Sweeney-Brent" method.

This method works well when $1 \le x < 2$ (other values of x need to be reduced first). The idea is to represent the Gamma function as a sum of two integrals,

$$\Gamma(x) = \int_0^M u^{x-1} \exp(-u)\, du + \int_M^\infty u^{x-1} \exp(-u)\, du.$$

The above identity clearly holds for any M. Both integrals in this equation can be approximated by power series (although we may do without the second integral altogether, for a small increase of computation time). The parameter M and the numbers of terms in the series must be chosen appropriately, as we shall see below.

The first integral in this equation can be found as a sum of the Taylor series (expanding $\exp(-u)$ near $u = 0$),

$$\int_0^M u^{x-1} \exp(-u)\, du = M^x \sum_{n=0}^\infty \frac{(-1)^n M^n}{(n+x)\, n!}.$$

This series absolutely converges for any finite M. However, the series has alternating signs and incurs a certain round-off error. The second integral is smaller than $M^{x-1} \exp(-M)$ and M can be chosen large enough so that this is smaller than 10^{-P}. This condition gives approximately $M > P \ln 10 + \ln P \ln 10$.

Now we can estimate the number of terms in the above series. We know that the value of the Gamma function is of order 1. The condition that n-th term of the series is smaller than 10^{-P} gives $n \ln \frac{n}{e} M > P \ln 10$. With the above value for M, we obtain $n = P \frac{\ln 10}{W\left(\frac{1}{e}\right)}$ where W is Lambert's function; $W\left(\frac{1}{e}\right) \approx 0.2785$.

The terms of the series are however not monotonic: first the terms grow and then they start to decrease, like in the Taylor series for the exponential function evaluated at a large argument. The ratio of the $(k+1)$-th term to the k-th term is approximately $\frac{M}{k+1}$. Therefore the terms with $k \approx M$ will be the largest and will have the magnitude of order $\frac{M^M}{M!} \approx \exp(M) \approx 10^P$. In other words, we will be adding and subtracting large numbers with P digits before the decimal point, but we need to obtain a result with P digits after the decimal point. Therefore to avoid the round-off error we need to increase the working precision to $2P$ floating-point decimal digits.

It is quicker to compute this series if x is a small rational number, because then the long multiplications can be avoided, or at high enough precision the binary splitting can be used. Calculations are also somewhat faster if M is chosen as an integer value.

If the second integral is approximated by an asymptotic series instead of a constant $\exp(-M)$, then it turns out that the smallest error of the series is $\exp(-2M)$. Therefore we can choose a smaller value of M and the round-off error gets somewhat smaller. According to [Brent 1978], we then need only $\frac{3}{2}P$ digits of working precision, rather than $2P$, for computing the first series (and only $\frac{P}{2}$ digits for computing the second series). However, this computational savings may not be significant enough to justify computing a second series.

6.2 Euler's constant γ

Euler's constant γ is defined as

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

This constant is useful for various reasons, mostly when working with higher transcendental functions. For example, γ is needed to obtain a Taylor series expansion of $\Gamma(x)$. Another useful relation is

$$\frac{\partial}{\partial x} \Gamma(x)_{x=1} = -\gamma.$$

Approximately $\gamma \approx 0.577216$.

Method 1: Brent's decomposition

Computing γ by the series in the definition is extremely slow. A much faster method can be used, based on some identities of Bessel functions. (See [Brent *et al.* 1980] and [Gourdon *et al.* 2001].)

The basic formulae for the "fast" method (Brent's method "B1") are:

$$\gamma \approx \frac{S(n)}{V(n)} - \ln n,$$

where $S(n)$ and $V(n)$ are some auxiliary functions, and n is chosen to be high enough (precise estimates are given below).

First, the sequence H_n is defined as the partial sum of the harmonic series:

$$H_n \equiv \sum_{k=1}^n \frac{1}{k}.$$

We also define $H_0 \equiv 0$ for convenience. The function $V(n)$ is the modified Bessel function $I_0(2n)$. It is computed as a Taylor series

$$V(n) \equiv \sum_{k=0}^\infty \left(\frac{n^k}{k!} \right)^2.$$

The function $S(n)$ is defined by a series like $V(n)$ but with each term multiplied by H_k:

$$S(n) \equiv \sum_{k=0}^\infty \left(\frac{n^k}{k!} \right)^2 H_k.$$

Note that we need to compute $S(n)$ and $V(n)$ with enough precision, so the sum over k will have to be performed up to a large enough k. (In practice, we do not really need to know the limit $k_{\max}$ beforehand. Instead, we can simply add terms to the series for $V(n)$ and $S(n)$ until desired precision is achieved. Knowing $k_{\max}$ in advance would help only if we needed to compare this method for computing γ with some other method, or if we would use the rectangular method of evaluating the Taylor series.)

According to [Brent *et al.* 1980], the error of this approximation of γ, assuming that $S(n)$ and $V(n)$ are computed exactly, is

$$\left| \gamma - \frac{S(n)}{V(n)} \right| < \pi \exp(-4n).$$

Therefore the parameter n is proportional to the number of digits we need. If we need P decimal digits (of absolute, not relative, precision), then we have to choose

$$n > \frac{P \ln 10 + \ln \pi}{4}.$$

The required number of terms $k_{\max}$ in the summation over k to get $S(n)$ and $V(n)$ with this precision can be approximated

as usual via Stirling's formula. It turns out that $k_{\max}$ is also proportional to the number of digits, $k_{\max} \approx 2.07P$.

Therefore, this method of computing γ has "linear convergence", i.e. the number of iterations is linear in the number of correct digits we need in the result. Of course, all calculations need to be performed with the working precision. The working precision must be a few digits more than P because we accumulate about $\frac{\ln k_{\max}}{\ln 10}$ digits of round-off error by performing $k_{\max}$ arithmetic operations.

Brent mentions a small improvement on this method (his method "B3"). It consists of estimating the error of the approximation of γ by an asymptotic series. Denote $W(n)$ the function

$$W(n) \equiv \frac{1}{4n} \sum_{k=0}^{2n} \frac{((2k)!)^3}{(k!)^4} (16n)^{2k}.$$

This function is basically the asymptotic series for $I_0(2n) K_0(2n)$, where I_0 and K_0 are modified Bessel functions. The sum in this series has to be evaluated until about $k = 2n$ to get a good precision. Then a more precise approximation for γ is

$$\gamma \approx \frac{U(n)}{V(n)} - \frac{W(n)}{V(n)^2}.$$

The precision of this approximation is of order $O(\exp(-8n))$ instead of $O(\exp(-4n))$ in Brent's method "B1". However, this is not such a great savings in time, because almost as much additional work has to be done to compute $W(n)$. Brent estimated that the method "B3" is about 20% faster than "B1".

Method 2: Brent's summation trick

Computation of $S(n)$ seems to need a running computation of H_k and a long multiplication by H_k at each term. To compute $S(n)$ and $V(n)$ faster and more accurately, Brent suggested the following trick that avoids this long multiplication and computes H_k simultaneously with the series. Define the function $U(n) \equiv S(n) - V(n) \ln n$. Then $\gamma \approx \frac{U(n)}{V(n)}$. The series for $U(n)$ is $U(n) \equiv \sum_{k=0}^{\infty} A_k$, with

$$A_k \equiv \left(\frac{n^k}{k!}\right)^2 (H_k - \ln n).$$

If we denote

$$B_k \equiv \left(\frac{n^k}{k!}\right)^2$$

the k-th term of the series for $V(n)$, then we can compute A_k and B_k simultaneously using the recurrence relations $A_0 = -\ln n$, $B_0 = 1$,

$$B_k = B_{k-1} \frac{n^2}{k^2},$$

$$A_k = \frac{1}{k}\left(A_{k-1} \frac{n^2}{k} + B_k\right).$$

This trick can be formulated for any sequence A_k of the form $A_k = B_k C_k$, where the sequences B_k and C_k are given by the recurrences $B_k = p(k) B_{k-1}$ and $C_k = q(k) + C_{k-1}$. Here we assume that $p(k)$ and $q(k)$ are known functions of k that can be computed to P digits using $O(P)$ operations, e.g. rational functions with short constant coefficients. Instead of evaluating B_k and C_k separately and multiplying them using a long multiplication, we note that $p(k) A_{k-1} = B_k C_{k-1}$. This allows to compute A_k by using the following two recurrences:

$$B_k = p(k) B_{k-1},$$

$$A_k = p(k) A_{k-1} + q(k) B_k.$$

All multiplications and divisions in these recurrence relations are performed with short integers, so the long multiplications are avoided. The time complexity of this method is $O(P^2)$ where P is the required number of digits. A variant of binary splitting method can be used to reduce the complexity to $O(M(P \ln P) \ln P)$ which gives an asymptotic advantage at very high precision.

Also, it turns out that we can use a variant of the fast "rectangular method" to evaluate the series for $U(n)$ and $V(n)$ simultaneously. (We can consider these series as Taylor series in n^2.) This however does not speed up the evaluation of γ. This happens because the rectangular method requires long multiplications and leads in this case to increased round-off errors. The rectangular method for computing a power series in x is less efficient than a straightforward computation when x is a "short" rational or integer number.

The "rectangular method" for computing $\sum_{k=0}^{N} x^k A_k$ needs to be able to convert a coefficient of the Taylor series into the next coefficient A_{k+1} by "short" operations, more precisely, by some multiplications and divisions by integers of order k. The j-th column of the rectangle ($j = 0, 1, ...$) consists of numbers $x^{rj} A_{rj}$, $x^{rj} A_{rj+1}$, ..., $x^r A_{rj+r-1}$. The numbers of this column are computed sequentially by short operations, starting from the $x^{rj} A_{jr}$ which is known from the end of the previous column. The recurrence relation for A_k is not just some multiplication by rational numbers, but also contains an addition of B_k. However, if we also use the rectangular method for $V(n)$, the number $x^{rj} B_{rj}$ will be known and so we will be able to use the recurrence relation to get $x^{rj} A_{rj+1}$ and all following numbers of the column.

Derivation of method 1

Brent's "B1" method can be derived from the Taylor series for the modified Bessel function $I_\nu(z)$,

$$I_\nu(z) = \sum_{k=0}^{\infty} \frac{z^{\nu+2k}}{2^{\nu+2k} \Gamma(\nu + k + 1)\, k!},$$

and the definition of the modified Bessel function $K_0(z)$,

$$K_0(z) \equiv -\left(\frac{\partial}{\partial \nu} I_\nu(z)_{\nu=0}\right).$$

Here the derivative wrt ν is taken at $\nu = 0$. This derivative can be evaluated using the above Taylor series for $I_\nu(z)$ and expressed through the series for $S(n)$. To compute this, we need the derivative of the Gamma function at integer arguments n:

$$\frac{\partial}{\partial x} \Gamma(x)_{x=n+1} = n! (H_n - \gamma).$$

The resulting identity in the way it is used here is

$$\gamma + \ln n = \frac{S_0(2n) - K_0(2n)}{I_0(2n)}.$$

Since $K_0(2n)$ decays very quickly at large n, we obtain the approximation

$$\gamma \approx \frac{S_0(2n)}{I_0(2n)} - \ln n + O(\exp(-4n)).$$

6.3 Catalan's constant G

Catalan's constant G is defined by

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

This series converges very slowly. There are several alternative methods.

Best method so far

A fast series to evaluate this constant is:

$$G = \frac{\pi}{8}\ln\left(2+\sqrt{3}\right) + \frac{3}{8}\sum_{n=0}^{\infty}\frac{(n!)^2}{(2n)!\,(2n+1)^2}$$

[Bailey *et al.* 1997].

To obtain P decimal digits of relative precision, we need to take at most $P\frac{\ln 10}{\ln 4}$ terms of the series. The sum can be efficiently evaluated using Horner's scheme, for example

$$1+\frac{1}{2\cdot 3}\left(\frac{1}{3}+\frac{2}{2\cdot 5}\left(\frac{1}{5}+\frac{3}{2\cdot 7}\left(\frac{1}{7}+...\right)\right)\right).$$

This takes $O\left(P^2\right)$ operations because there are $O(P)$ short multiplications and divisions of P-digit numbers by small integers. At very high precision, the binary splitting technique can be used instead of Horner's scheme to further reduce the computation time to $O\left(M(P)(\ln P)^2\right)$.

A drawback of this scheme is that it requires a separate high-precision computation of π, $\sqrt{3}$ and of the logarithm.

Other methods

Another method is based on an identity by Ramanujan. The formula is

$$G = \sum_{k=0}^{\infty}\frac{(k!)^2}{(2k+1)!}\cdot 2^{k-1}\sum_{j=0}^{k}\frac{1}{2j+1}.$$

The k-th term of the outer sum is of order 2^{-k} and so we need to take slightly more than $P\frac{\ln 10}{\ln 2}$ terms for P decimal digits. But each term is more complicated than in the first method's series. The running time of this formula seems to be a few times slower than the previous method.

This method combined with Brent's summation trick (see the section on the Euler constant) was used in [Fee 1990]. Brent's trick allows to avoid a separate computation of the harmonic sum and all long multiplications. Catalan's constant is obtained as a limit of G_k where $G_0 = B_0 = \frac{1}{2}$ and

$$B_k = \frac{k}{2k+1}B_{k-1},$$

$$G_k = \frac{1}{2k+1}\left(kG_{k-1}+B_k\right).$$

The running time is $O\left(P^2\right)$. Since only rational numbers are involved, the binary splitting technique can be used at high precision.

A third formula is more complicated but the convergence is much faster and there is no need to evaluate any other transcendental functions. This formula is called "Broadhurst's series".

$$G = \frac{3}{2}\sum_{k=0}^{\infty}\frac{1}{16^k}\sum_{i=0}^{7}\frac{a_i}{(8k+i)^2} - \frac{1}{4}\sum_{k=0}^{\infty}\frac{1}{4096^k}\sum_{i=0}^{7}\frac{a_i}{(8k+i)^2}.$$

Here the arrays of coefficients are defined as $a_i = \left(0,1,-1,\frac{1}{2},0,-\frac{1}{4},\frac{1}{4},-\frac{1}{8}\right)$ and $b_i = \left(0,1,\frac{1}{2},\frac{1}{8},0,-\frac{1}{64},-\frac{1}{128},-\frac{1}{512}\right)$.

We need to take only $P\frac{\ln 10}{\ln 16}$ terms of the first series and $P\frac{\ln 10}{\ln 4096}$ terms of the second series. However, each term is about six times more complicated than one term of the first method's series. So there are no computational savings (unless $\ln x$ is excessively slow).

6.4 Riemann's Zeta function

Riemann's Zeta function $\zeta(s)$ is defined for complex s such that $Re(s) > 0$ as a sum of inverse powers of integers:

$$\zeta(s) \equiv \sum_{n=0}^{\infty}\frac{1}{n^s}.$$

This function can be analytically continued to the entire complex plane except the point $s = 1$ where it diverges. It satisfies several identities, for example, a formula useful for negative $Re(s)$,

$$\zeta(1-s) = \frac{2\Gamma(s)}{(2\pi)^s}\cos\frac{\pi s}{2}\zeta(s),$$

and a formula for even integers that helps in numerical testing,

$$\zeta(2n) = \frac{(-1)^{n+1}(2\pi)^{2n}}{2(2n)!}B_{2n},$$

where B_n are Bernoulli numbers. (The values at negative integer n are given by $\zeta(-n) = -\frac{B_{n+1}}{n+1}$.)

The classic book [Bateman *et al.* 1953], vol. 1, describes many results concerning the properties of $\zeta(s)$.

For the numerical evaluation of Riemann's Zeta function with arbitrary precision to become feasible, one needs special algorithms. Recently P. Borwein [Borwein 1995] gave a simple and quick approximation algorithm for $Re(s) > 0$. See also [Borwein *et al.* 1999] for a review of methods.

It is the "third" algorithm (the simplest one) from P. Borwein's paper which is implemented in Yacas. The approximation formula valid for $Re(s) > -(n-1)$ is

$$\zeta(s) = \frac{1}{2^n\left(1-2^{1-s}\right)}\sum_{j=0}^{2n-1}\frac{e_j}{(j+1)^s},$$

where the coefficients e_j for $j = 0$, ..., $2n-1$ are defined by

$$e_j \equiv (-1)^{j-1}\left(\sum_{k=0}^{j-n}\frac{n!}{k!\,(n-k)!} - 2^n\right),$$

and the empty sum (for $j < n$) is taken to be zero. The parameter n must be chosen high enough to achieve the desired precision. The error estimate for this formula is approximately

$$\text{error} < 8^{-n}$$

for the relative precision, which means that to achieve P correct digits we must have $n > P\ln 10/\ln 8$.

This method requires to compute n times the exponential and the logarithm to find the power $(j+1)^{-s}$. This power can be computed in asymptotic time $O(M(P)\ln P)$, unless s is an integer, in which case this computation is $O(M(P))$, the cost of one division by an integer $(j+1)^s$. Therefore the complexity of this method is at most $O(PM(P)\ln P)$.

The function `Zeta(s)` calls `Internal'ZetaNum(s)` to compute this approximation formula for $Re(s) > \frac{1}{2}$ and uses the identity above to get the value for other s.

For very large values of s, it is faster to use more direct methods implemented in the routines `Internal'ZetaNum1(s,N)` and `Internal'ZetaNum2(s,N)`. If the required precision is P digits and $s > 1 + \frac{\ln 10}{\ln P}P$, then it is more efficient to compute the defining series for $\zeta(s)$,

$$\zeta(s) \approx \sum_{k=1}^{N}\frac{1}{k^s},$$

up to a certain number of terms N, than to use Borwein's method. The required number of terms N is given by

$$N = 10^{\frac{P}{s-1}}.$$

This is implemented in the routine `Internal'ZetaNum1(s)`. For example, at 100 digits of precision it is advisable to use `Internal'ZetaNum1(s)` only for $s > 50$, since it would require $N < 110$ terms in the series, whereas the expression used in `Internal'ZetaNum(s)` uses $n = \frac{\ln 10}{\ln 8} P$ terms (of a different series). The complexity of this method is $O(N) M(P)$.

Alternatively, one can use `Internal'ZetaNum2(n,N)` which computes the infinite product over prime numbers p_i

$$\frac{1}{\zeta(n)} \approx \prod_{k=1}^{M} \left(1 - \frac{1}{p_k^s}\right).$$

Here M must be chosen such that the M-th prime number $p_M > N$. To obtain P digits of precision, N must be chosen as above, i.e. $N > 10^{\frac{P}{s-1}}$. Since only prime numbers p_i are used, this formula is asymptotically faster than `Internal'ZetaNum1`. (Prime numbers have to be obtained and tested but this is quick for numbers of size n, compared with other operations.) The number of primes up to N is asymptotically $\pi(N) \sim \frac{N}{\ln N}$ and therefore this procedure is faster by a factor $O(\ln N) \sim O(\ln n)$. However, for $n < 250$ it is better (with Yacas internal math) to use the `Internal'ZetaNum1` routine because it involves fewer multiplications.

Zeta function at special arguments

Recently some identities dating back to S. Ramanujan have been studied in relation to Riemann's Zeta function. A side result was the development of fast methods for computing the Zeta function at odd integer arguments (see [Borwein *et al.* 1999] for a review) and at small rational arguments ([Kanemitsu *et al.* 2001]).

The value $\zeta(3)$, also known as the Apery's constant, can be computed using the following geometrically convergent series:

$$\zeta(3) = \frac{5}{2} \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^3 \binom{2k}{k}}.$$

For other odd integers n there is no general analogous formula. The corresponding expressions for $\zeta(5)$ and $\zeta(7)$ are

$$\zeta(5) = 2 \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^5 \binom{2k}{k}} - \frac{5}{2} \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^3 \binom{2k}{k}} \sum_{j=1}^{k-1} \frac{1}{j^2};$$

$$\zeta(7) = \frac{5}{2} \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^7 \binom{2k}{k}} + \frac{25}{2} \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k^3 \binom{2k}{k}} \sum_{j=1}^{k-1} \frac{1}{j^4}.$$

In these series the term $\binom{2k}{k}$ grows approximately as 4^k and therefore one can take no more than $P \frac{\ln 10}{\ln 4}$ terms in the series to get P decimal digits of relative precision.

For odd integer n there are the following special relations: for $n \equiv 3 \bmod 4$,

$$\zeta(n) = \frac{(2\pi)^n}{2(n+1)!} \sum_{k=0}^{\frac{n+1}{2}} (-1)^{k-1} \binom{n+1}{2k} B_{n+1-2k} B_{2k}$$

$$-2 \sum_{k=1}^{\infty} \frac{1}{k^n (\exp(2\pi k) - 1)},$$

and for $n \equiv 1 \bmod 4$,

$$\zeta(n) = \frac{(2\pi)^n}{(n+1)!\,(n-1)}$$

$$\ldots \sum_{k=0}^{\frac{n+1}{4}} (-1)^k (n+1-4k) \binom{n+1}{2k} B_{n+1-2k} B_{2k}$$

$$-2 \sum_{k=1}^{\infty} \frac{\exp(2\pi k) \left(1 + \frac{4\pi k}{k-1}\right) - 1}{k^n (\exp(2\pi k) - 1)^2}.$$

These relations contain geometrically convergent series, and it suffices to take $P \frac{\ln 10}{2\pi}$ terms to obtain P decimal digits of relative precision.

Finally, [Kanemitsu *et al.* 2001] gave a curious formula for the values of Riemann's Zeta function at rational values between 0 and 2 (their "Corollary 1"). This formula is very complicated but contains a geometrically convergent series.

We shall have to define several auxiliary functions to make the formula more understandable. We shall be interested in the values of $\zeta\left(\frac{p}{N}\right)$ where p, N are integers. For integer h, N such that $0 \le h \le N$, and for arbitrary real x,

$$\zeta\left(\frac{2h-1}{N}\right) = \frac{Nx}{\pi} \left(\frac{2\pi}{x}\right)^{\frac{2h-1}{N}} \sin\frac{\pi(2h-1)}{2N} L(x, N, h),$$

$$\zeta\left(\frac{N-2h+1}{N}\right) = \frac{N}{\Gamma\left(\frac{N-2h+1}{N}\right)} x^{\frac{N-2h+1}{N}} L(x, N, h).$$

Here the function $L(x, N, h)$ (containing the "Lambert series") is defined by

$$L(x, N, h) \equiv \sum_{n=1}^{\infty} \frac{n^{N-2h}}{\exp(n^N x) - 1} - S(x, N, h)$$

$$+\frac{1}{2}\zeta(-N+2h) - \frac{\zeta(2h)}{x},$$

and the function $S(x, N, h)$ is defined by

$$S(x, N, h) \equiv \frac{(-1)^{h+1}}{N} \left(\frac{2\pi}{x}\right)^{\frac{N-2h+1}{N}}$$

$$\ldots \sum_{n=0}^{\infty} \frac{1}{n^{\frac{2h-1}{N}}} \sum_{k=1}^{\frac{N}{2}} f(2k-1, x, n, N, h)$$

for even N, and by

$$S(x, N, h) \equiv \frac{(-1)^{h+1}}{N} \left(\frac{2\pi}{x}\right)^{\frac{N-2h+1}{N}}$$

$$\ldots \sum_{n=0}^{\infty} \frac{1}{n^{\frac{2h-1}{N}}} \left(f(0, x, n, N) + \sum_{k=1}^{\frac{N-1}{2}} f(2k, x, n, N, h) \right)$$

for odd N. The auxiliary function $f(j, x, n, N, h)$ is defined by

$$f(0, x, n, N) \equiv \frac{\exp(-A)}{2 \sinh A},$$

$$f(j, x, n, N, h) \equiv \frac{\cos(2B + C) - \exp(-2A)\cos C}{\cosh 2A - \cos 2B}$$

for integer $j \ge 1$, and in these expressions

$$A \equiv \pi \sqrt[N]{\frac{2\pi n}{x}} \cos\frac{\pi j}{2N},$$

$$B \equiv \pi \sqrt[N]{\frac{2\pi n}{x}} \sin\frac{\pi j}{2N},$$

and $C \equiv \frac{\pi j}{2} \frac{2h-1}{N}$.

Practical calculations using this formula are of the same asymptotic complexity as Borwein's method above. (It is not clear whether this method has a significant computational advantage.) The value of x can be chosen at will, so we should find such x as to minimize the cost of computation. There are two series to be computed: the terms in the first one decay as $\exp\left(-n^N x\right)$ while the terms in the second one (containing f) decay only as

$$\exp\left(-2A\right) \sim \exp\left(-2\pi \sqrt[N]{\frac{2\pi n}{x}}\right).$$

Also, there are about N copies of the second series. Clearly, a very small value should be chosen for x so that the second series converges somewhat faster.

For a target precision of P decimal digits, the required numbers of terms n_1, n_2 for the first and the second series can be estimated as $n_1 \approx \sqrt[N]{\frac{P \ln 10}{x}}$, $n_2 \approx \frac{x}{2\pi}\left(\frac{P \ln 10}{2\pi}\right)^N$. (Here we assume that N, the denominator of the fraction $\frac{p}{N}$, is at least 10. This scheme is impractical for very large N because it requires to add $O\left(N\right)$ slightly different variants of the second series.) The total cost is proportional to the time it takes to compute $\exp\left(x\right)$ or $\cos x$ and to roughly $n_1 + Nn_2$. The value of x that minimizes this cost function is approximately

$$x \approx \frac{2\pi}{N^2}\left(\frac{2\pi}{P \ln 10}\right)^{N-1}.$$

With this choice, the asymptotic total cost is $O\left(PM\left(P\right)\right)$.

6.5 Lambert's W function

Lambert's W function is (a multiple-valued, complex function) defined for any (complex) z by $W\left(z\right)\exp\left(W\left(z\right)\right) = z$. This function is sometimes useful to represent solutions of transcendental equations or integrals.

Asymptotics of Lambert's W function are

$$W\left(\frac{\frac{z^2-2}{2}}{e}\right) = -1 + z - \frac{z^3}{3} + \frac{11}{72}z^4 - ...$$

(this is only good really close to z=0); $W\left(x\right) = x - x^2 + \frac{3}{2}x^3 - ...$ (this is good very near x=0);

$$W\left(x\right) = \ln x - \ln\ln x + \frac{\ln\ln x}{\ln x} + \frac{1}{2}\left(\frac{\ln\ln x}{\ln x}\right)^2 + ...$$

(good for very large x but it is not straightforward to find higher terms!).

Here are some inequalities to help estimate $W\left(x\right)$ at large x (more exactly, for $x > e$):

$$\ln\frac{x}{\ln x} < W\left(x\right) < \ln x,$$

$$\ln\frac{x}{\ln x - 1} < W\left(x\right) < \ln\frac{x}{\ln x - 1} + 0.13.$$

One can also find uniform rational approximations, e.g.:

$$W\left(x\right) \approx \ln\left(1+x\right)\left(1 - \frac{\ln\left(1+\ln\left(1+x\right)\right)}{2+\ln\left(1+x\right)}\right)$$

(uniformly good for $x > 0$, relative error not more than 10^{-2}); and

$$W\left(x\right) \approx \frac{xe}{1+\left(\left(2ex+2\right)^{-\frac{1}{2}} - \frac{1}{\sqrt{2}} + \frac{1}{e-1}\right)^{-1}}$$

(uniformly good for $-\frac{1}{e} < x < 0$, relative error not more than 10^{-3}).

There exists a uniform approximation of the form

$$W\left(x\right) \approx \left(a+L\right)\frac{f - \ln\left(1 + c\ln\left(1+dY\right)\right) + 2L}{a + \frac{1}{2} + L},$$

where $a = 2.3436$, $b = 0.8842$, $c = 0.9294$, $d = 0.5106$, $f = -1.2133$ are constants, $Y \equiv \sqrt{2ex+2}$ and $L \equiv \ln\left(1+bY\right)$. (The coefficients a, ..., f are found numerically by matching the asymptotics at three points $x = -\frac{1}{e}$, $x = 0$, $x = \infty$.) This approximation miraculously works over the whole complex plane within relative error at most 0.008. The behavior of this formula at the branching points $x = -\frac{1}{e}$ and $x = \infty$ correctly mimics $W\left(x\right)$ if the branch cuts of the square root and of the logarithm are chosen appropriately (e.g. the common branch cut along the negative real semiaxis).

The numerical procedure uses Halley's method. Halley's iteration for the equation $W\exp\left(W\right) = x$ can be written as

$$W' = W - \frac{W - x\exp\left(-W\right)}{W + 1 - \frac{W+2}{W+1}\frac{W - x\exp\left(-W\right)}{2}}.$$

It has cubic convergence for any initial value $W > -\exp\left(-1\right)$.

The initial value is computed using one of the uniform approximation formulae. The good precision of the uniform approximation guarantees rapid convergence of the iteration scheme to the correct root of the equation, even for complex arguments x.

6.6 Bessel functions

Bessel functions are a family of special functions solving the equation

$$\frac{d^2}{dx^2}w\left(x\right) + \frac{1}{x}\left(\frac{d}{dx}w\left(x\right)\right) + \left(1 - \frac{n^2}{x^2}\right)w\left(x\right) = 0.$$

There are two linearly independent solutions which can be taken as the pair of Hankel functions $H_1\left(n,x\right)$, $H_2\left(n,x\right)$, or as the pair of Bessel-Weber functions J_n, Y_n. These pairs are linearly related, $J_n = \frac{1}{2}\left(H_1\left(n,x\right) + H_2\left(n,x\right)\right)$, $J_n = \frac{1}{2\imath}\left(H_1\left(n,x\right) - H_2\left(n,x\right)\right)$. The function $H_2\left(n,x\right)$ is the complex conjugate of $H_1\left(n,x\right)$. This arrangement of four functions is very similar to the relation between $\sin x$, $\cos x$ and $\exp\left(\imath x\right)$, $\exp\left(-\imath x\right)$, which are all solutions of $\frac{d^2}{dx^2}f\left(x\right) + f\left(x\right) = 0$.

For large values of $|x|$, there is the following asymptotic series:

$$H_1\left(n,x\right) \sim \sqrt{\frac{2}{\pi x}}\exp\left(\imath\zeta\right)\sum_{k=0}^{\infty}\imath^k\frac{A\left(k,n\right)}{x^k},$$

where $\zeta \equiv x - \frac{1}{2}n\pi - \frac{1}{4}\pi$ and

$$A\left(k,n\right) \equiv \frac{\left(4n^2-1^2\right)\left(4n^2-3^2\right)...\left(4n^2-\left(2k-1\right)^2\right)}{k!\cdot 8^k}.$$

From this one can find the asymptotic series for $J_n \sim \sqrt{\frac{2}{\pi x}}\cos\zeta\sum_{k=0}^{\infty}\left(-1\right)^k A\left(2k,n\right)x^{-2k} - \sqrt{\frac{2}{\pi x}}\sin\zeta\sum_{k=0}^{\infty}\left(-1\right)^k A\left(2k+1,n\right)x^{-2k-1}$ and $Y_n \sim \sqrt{\frac{2}{\pi x}}\sin\zeta\sum_{k=0}^{\infty}\left(-1\right)^k A\left(2k,n\right)x^{-2k} + \sqrt{\frac{2}{\pi x}}\cos\zeta\sum_{k=0}^{\infty}\left(-1\right)^k A\left(2k+1,n\right)x^{-2k-1}$.

The error of a truncated asymptotic series is not larger than the first discarded term if the number of terms is larger than $n - \frac{1}{2}$. (See the book [Olver 1974] for derivations. It seems that each asymptotic series requires special treatment and yet in all cases the error is about the same as the first discarded term.)

Currently Yacas can compute `BesselJ(n,x)` for all x where n is an integer and for $|x| \leq 2\Gamma(n)$ when n is a real number. Yacas currently uses the Taylor series when $|x| \leq 2\Gamma(n)$ to compute the numerical value:

$$J_n(x) \equiv \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+n}}{2^{2k+n} k! \Gamma(k+n+1)}.$$

If $|x| > 2\Gamma(n)$ and n is an integer, then Yacas uses the forward recurrence relation:

$$J_n(x) \equiv 2\frac{n+1}{x} J_{n+1}(x) - J_{n+2}(x)$$

until the given `BesselJ` function is represented in terms of higher order terms which all satisfy $|x| \leq 2\Gamma(n)$. Note that when n is much smaller than x, this algorithm is quite slow because the number of Bessel function evaluations grows like 2^i, where i is the number of times the recurrence identity is used.

We see from the definition that when $|x| \leq 2\Gamma(n)$, the absolute value of each term is always decreasing (which is called absolutely monotonely decreasing). From this we know that if we stop after i iterations, the error will be bounded by the absolute value of the next term. So given a set precision, turn this into a value ϵ, so that we can check if the current term will contribute to the sum at the prescribed precision. Before doing this, Yacas currently increases the precision by 20% to do interim calculations. This is a heuristic that works, it is not backed by theory. The value ϵ is given by $\epsilon \equiv 5 \cdot 10^{-\text{prec}}$, where prec was the previous precision. This is directly from the definition of floating point number which is correct to prec digits: A number correct to prec digits has a rounding error no greater than $5 \cdot 10^{-\text{prec}}$. Beware that some books incorrectly have .5 instead of 5.

Bug: Something is not right with complex numbers, but pure imaginary are OK.

6.7 Bernoulli numbers and polynomials

The Bernoulli numbers B_n are rational numbers that are frequently used in applications. The first few numbers are $B_0 = 1$, $B_1 = -\frac{1}{2}$, $B_2 = 4$, $B_3 = 0$, $B_4 = -\frac{1}{30}$. The numbers B_n can be defined by the series expansion of the following generating function,

$$\frac{z}{e^z - 1} = \sum_{n=0}^{\infty} B_n \frac{z^n}{n!}.$$

The Bernoulli polynomials $B(x)_n$ are defined similarly by

$$\frac{z \exp(zx)}{e^z - 1} = \sum_{n=0}^{\infty} B(x)_n \frac{z^n}{n!}.$$

The Bernoulli polynomials are related to Bernoulli numbers by

$$B(x)_n = \sum_{k=0}^{n} \binom{n}{k} x^k B_{n-k},$$

where $\binom{n}{k}$ are binomial coefficients.

Bernoulli numbers and polynomials are used in various Taylor series expansions, in the Euler-Maclauren series resummation formula, in Riemann's Zeta function and so on. For example, the sum of (integer) p-th powers of consecutive integers is given by

$$\sum_{k=0}^{n-1} k^p = \frac{B(n)_{p+1} - B_{p+1}}{p+1}.$$

The Bernoulli polynomials $B(x)_n$ can be found by first computing an array of Bernoulli numbers up to B_n and then applying the above formula for the coefficients.

We consider two distinct computational tasks: evaluate a Bernoulli number exactly as a rational, or find it approximately to a specified floating-point precision. There are also two possible problem settings: either we need to evaluate *all* Bernoulli numbers B_n up to some n (this situation occurs most often in practice), or we only need one isolated value B_n for some large n. Depending on how large n is, different algorithms can be chosen in these cases.

Exact evaluation of Bernoulli numbers

In the `Bernoulli()` routine, Bernoulli numbers are evaluated exactly (as rational numbers) via one of the two algorithms. The first, simpler algorithm (`BernoullliArray()`) uses the recurrence relation,

$$B_n = -\frac{1}{n+1} \sum_{k=0}^{n-1} B_k \binom{n+1}{k}.$$

This formula requires to know the entire set of B_k with k up to a given n to compute B_n. Therefore at large n this algorithm is a very slow way to compute B_n if we do not need all other B_k.

Here is an estimate of the cost of `BernoullliArray`. Suppose $M(P)$ is the time needed to multiply P-digit integers. The required number of digits P to store the numerator of B_n is asymptotically $P \sim n \ln n$. At each of the n iterations we need to multiply $O(n)$ large rational numbers by large coefficients and take a GCD to simplify the resulting fractions. The time for GCD is logarithmic in P. So the complexity of this algorithm is $O\left(n^2 M(P) \ln P\right)$ with $P \sim n \ln n$.

For large (even) values of the index n, a single Bernoulli number B_n can be computed by a more efficient procedure: the integer part and the fractional part of B_n are found separately (this method is also well explained in [Gourdon *et al.* 2001]).

First, by the theorem of Clausen – von Staudt, the fractional part of $(-B_n)$ is the same as the fractional part of the sum of all inverse prime numbers p such that n is divisible by $p-1$. To illustrate the theorem, take $n = 10$ with $B_{10} = \frac{5}{66}$. The number $n = 10$ is divisible only by 1, 2, 5, and 10; this corresponds to $p = 2$, 3, 6 and 11. Of these, 6 is not a prime. Therefore, we exclude 6 and take the sum $\frac{1}{2} + \frac{1}{3} + \frac{1}{11} = \frac{61}{66}$. The theorem now says that $\frac{61}{66}$ has the same fractional part as $-B_{10}$; in other words, $-B_{10} = i + f$ where i is some unknown integer and the fractional part f is a nonnegative rational number, $0 \leq f < 1$, which is now known to be $\frac{61}{66}$. Indeed $-B_{10} = -1 + \frac{61}{66}$. So one can find the fractional part of the Bernoulli number relatively quickly by just checking the numbers that might divide n.

Now one needs to obtain the integer part of B_n. The number B_n is positive if $n \bmod 4 = 2$ and negative if $n \bmod 4 = 0$. One can use Riemann's Zeta function identity for even integer values of the argument and compute the value $\zeta(n)$ precisely enough so that the integer part of the Bernoulli number is determined. The required precision is found by estimating the Bernoulli number from the same identity in which one approximates $\zeta(n) = 1$, i.e.

$$|B_{2n}| \approx 2\frac{(2n)!}{(2\pi)^{2n}}.$$

To estimate the factorial of large numbers, we can use Stirling's asymptotic formula

$$\ln n! \approx \frac{\ln 2\pi n}{2} + n \ln \frac{n}{e}.$$

The result is that for large n,

$$|B_{2n}| \sim 2\left(\frac{n}{\pi e}\right)^{2n}.$$

At such large values of the argument n, it is feasible to use the routines `Internal'ZetaNum1(n, N)` or `Internal'ZetaNum2(n,N)` to compute the zeta function. These routines approximate $\zeta(n)$ by the defining series

$$\zeta(n) \approx \sum_{k=1}^{N} \frac{1}{k^n}.$$

The remainder of the sum is of order N^{-n}. By straightforward algebra one obtains a lower bound on N,

$$N > \frac{n}{2\pi e},$$

for this sum to give enough precision to compute the integer part of the Bernoulli number B_n.

For example, let us compute B_{20} using this method.

1. We find the fractional part, using the fact that 20 is divided only by 1, 2, 4, 5, 10, and 20 and only 2, 3, 5, 11 are prime:

   ```
   In> 1/2 + 1/3 + 1/5 + 1/11;
   Out> 371/330;
   ```

   This number has fractional part equal to 41/330 and it's the same as the fractional part of $-B_{20}$. Since B_{20} must be negative, this means that B_{20} is of the form $-\left(\mathrm{X} + \frac{41}{330}\right)$ where X is some positive integer which we now have to find.

2. We estimate the magnitude of $|B_{20}|$ to find the required precision to compute its integer part. We use the identity

   $$|B_{20}| = \frac{2 \cdot 20!}{(2\pi)^{20}} \zeta(20),$$

   Stirling's formula

   $$\ln 20! = \frac{\ln 2\pi + \ln 20}{2} + 20 \ln \frac{20}{e} \approx 42.3,$$

   and also the fact that $\zeta(20) = 1 + 2^{-20} + \ldots$ is approximately equal to 1. We find that the number B_{20} has approximately

   $$1 + \frac{\ln |B_{20}|}{\ln 10} = 1 + \frac{\ln 2 + \ln 20! - 20 \ln 2\pi}{\ln 10} \approx 3.72$$

   decimal digits before the decimal point; so a precision of 3 or 4 mantissa digits would surely be enough to compute its integer part.

3. We now need to compute $\zeta(20)$ to 4 decimal digits. The series

   $$\zeta(20) \approx \sum_{k=1}^{N} \frac{1}{k^{20}}$$

   will have an error of order N^{-20}. We need this error to be less than 10^{-4} and this is achieved with $N > 2$. Therefore it is enough to compute the sum up to $N = 2$:

   ```
   In> N(1+1/2^20)
   Out> 1.0000009536;
   ```

4. Now we find the integer part of $|B_{20}|$. This time we do not use Stirling's formula for the factorial but compute the exact factorial.

   ```
   In> N( 2*20! /(2*Pi)^20*1.0000009536 )
   Out> 529.1242423667;
   ```

   (Note the space after the factorial sign – it is needed for the syntax to parse correctly.) Therefore we know that the integer part of $|B_{20}|$ is 529.

5. Since $B_{20} = -\left(\mathrm{X} + \frac{41}{330}\right)$ and we found that X = 529, we obtain

   ```
   In> -(529+41/330);
   Out> -174611/330;
   ```

   This is exactly the Bernoulli number B_{20}.

All these steps are implemented in the routine `Bernoulli1`. The function `Bernoulli1Threshold()` returns the smallest n for which B_n is to be computed via this routine instead of the recursion relation. Its current default value is 20. This value can be set with `SetBernoulli1Threshold(threshold)`.

The complexity of `Bernoulli1` is estimated as the complexity of finding all primes up to n plus the complexity of computing the factorial, the power and the Zeta function. Finding the prime numbers up to n by checking all potential divisors up to $\sqrt{n}$ requires $O\left(n^{\frac{3}{2}} M(\ln n)\right)$ operations with precision $O(\ln n)$ digits. For the second step we need to evaluate $n!$, π^n and $\zeta(n)$ with precision of $P = O(n \ln n)$ digits. The factorial is found in n short multiplications with P-digit numbers (giving $O(nP)$), the power of π in $\ln n$ long multiplications (giving $O(M(P) \ln n)$), and `Internal'ZetaNum2(n)` (the asymptotically faster algorithm) requires $O(nM(P))$ operations. The Zeta function calculation dominates the total cost because $M(P)$ is slower than $O(P)$. So the total complexity of `Bernoulli1` is $O(nM(P))$ with $P \sim n \ln n$.

Note that this is the cost of finding just one Bernoulli number, as opposed to the $O\left(n^2 M(P) \ln P\right)$ cost of finding all Bernoulli numbers up to B_n using the first algorithm `Internal'BernoulliArray`. If we need a complete table of Bernoulli numbers, then `Internal'BernoulliArray` is only marginally (logarithmically) slower. So for finding complete Bernoulli tables, `Bernoulli1` is better only for very large n.

Approximate calculation of Bernoulli numbers

If Bernoulli numbers do not have to be found exactly but only to a certain floating-point precision P (this is usually the case for most numerical applications), then the situation is rather different. First, all calculations can be performed using floating-point numbers instead of exact rationals. This significantly speeds up the recurrence-based algorithms.

However, the recurrence relation used in `Internal'BernoulliArray` turns out to be numerically unstable and needs to be replaced by another [Brent 1978]. Brent's algorithm computes the Bernoulli numbers divided by factorials, $C_n \equiv \frac{B_{2n}}{(2n)!}$ using a (numerically stable) recurrence relation

$$2C_k\left(1 - 4^{-k}\right) = \frac{2k-1}{4^k (2k)!} - \sum_{j=1}^{k-1} \frac{C_{k-j}}{4^j (2j)!}.$$

The numerical instability of the usual recurrence relation

$$\sum_{j=0}^{k-1} \frac{C_{k-j}}{(2j+1)!} = \frac{k - \frac{1}{2}}{(2k+1)!}$$

and the numerical stability of Brent's recurrence are not obvious. Here is one way to demonstrate them. Consider the usual recurrence (above). For large k, the number C_k is approximately $C_k \approx 2(-1)^k (2\pi)^{-2k}$. Suppose we use this recurrence

to compute C_k from previously found values C_{k-1}, C_{k-2}, etc. and suppose that we have small relative errors e_k of finding C_k. Then instead of the correct C_k we use $C_k\left(1+e_k\right)$ in the recurrence. Now we can derive a relation for the error sequence e_k using the approximate values of C_k. It will be a linear recurrence of the form

$$\sum_{j=0}^{k-1}(-1)^{k-j}\,e_{k-j}\,\frac{(2\pi)^{2j}}{(2j+1)!}=\frac{k-\frac{1}{2}}{(2k+1)!}\,(2\pi)^{-2k}.$$

Note that the coefficients for $j>5$ are very small but the coefficients for $0\leq j\leq 5$ are of order 1. This means that we have a cancellation in the first 5 or so terms that produces a very small number C_k and this may lead to a loss of numerical precision. To investigate this loss, we find eigenvalues of the sequence e_k, i.e. we assume that $e_k=\lambda^k$ and find λ. If $|\lambda|>1$, then a small initial error e_1 will grow by a power of λ on each iteration and it would indicate a numerical instability.

The eigenvalue of the sequence e_k can be found approximately for large k if we notice that the recurrence relation for e_k is similar to the truncated Taylor series for $\sin x$. Substituting $e_k=\lambda^k$ into it and disregarding a very small number $(2\pi)^{-2k}$ on the right hand side, we find

$$\sum_{j=0}^{k-1}(-\lambda)^{k-j}\,\frac{(2\pi)^{2j}}{(2j+1)!}\approx\lambda^k\sin\frac{2\pi}{\sqrt{\lambda}}\approx 0,$$

which means that $\lambda=4$ is a solution. Therefore the recurrence is unstable.

By a very similar calculation one finds that the inverse powers of 4 in Brent's recurrence make the largest eigenvalue of the error sequence e_k almost equal to 1 and therefore the recurrence is stable. Brent gives the relative error in the computed C_k as $O\left(k^2\right)$ times the round-off error in the last digit of precision.

The complexity of Brent's method is given as $O\left(n^2P+nM\left(P\right)\right)$ for finding all Bernoulli numbers up to B_n with precision P digits. This computation time can be achieved if we compute the inverse factorials and powers of 4 approximately by floating-point routines that know how much precision is needed for each term in the recurrence relation. The final long multiplication by $(2k)!$ computed to precision P adds $M\left(P\right)$ to each Bernoulli number.

The non-iterative method using the Zeta function does not perform much better if a Bernoulli number B_n has to be computed with significantly fewer digits P than the full $O\left(n\ln n\right)$ digits needed to represent the integer part of B_n. (The fractional part of B_n can always be computed relatively quickly.) The Zeta function needs $10^{\frac{P}{n}}$ terms, so its complexity is $O\left(10^{\frac{P}{n}}M\left(P\right)\right)$ (here by assumption P is not very large so $10^{\frac{P}{n}}<\frac{n}{2\pi e}$; if $n>P$ we can disregard the power of 10 in the complexity formula). We should also add $O\left(\ln nM\left(P\right)\right)$ needed to compute the power of 2π. The total complexity of `Bernoulli1` is therefore $O\left(\ln nM\left(P\right)+10^{\frac{P}{n}}M\left(P\right)\right)$.

If only one Bernoulli number is required, then `Bernoulli1` is always faster. If all Bernoulli numbers up to a given n are required, then Brent's recurrence is faster for certain (small enough) n.

Currently Brent's recurrence is implemented as `Internal'BernoulliArray1()` but it is not used by `Bernoulli` because the internal arithmetic is not yet able to correctly compute with floating-point precision.

6.8 Error function $\operatorname{erf} x$ and related functions

The error function $\operatorname{erf} z$ is defined for any (complex) z by

$$\operatorname{erf} z\equiv\frac{2}{\sqrt{\pi}}\int_0^z\exp\left(-t^2\right)dt.$$

The complementary error function $\operatorname{erfc} x$ is defined for real x as

$$\operatorname{erfc} x\equiv\frac{2}{\sqrt{\pi}}\int_x^\infty\exp\left(-t^2\right)dt=1-\operatorname{erf} x.$$

The imaginary error function $\operatorname{Erfi}\left(x\right)$ is defined for real x as

$$\operatorname{Erfi}\left(x\right)\equiv\frac{2}{\sqrt{\pi}}\int_x^\infty\exp\left(t^2\right)dt.$$

Numerical computation of the error function $\operatorname{erf} z$ needs to be performed by different methods depending on the value of z and its position in the complex plane, and on the required precision. We follow the book [Tsimring 1988] and the paper [Thacher 1963]. (These texts, however, do not describe arbitrary-precision computations.)

The function $\operatorname{erf} z$ has the following approximations that are useful for its numerical computation:

1. The Taylor series at $z=0$,

$$\frac{\sqrt{\pi}}{2z}\operatorname{erf} z=1-\frac{z^2}{3}+\frac{z^4}{2!\cdot 5}-...+\frac{\left(-z^2\right)^n}{(2n+1)\,n!}+O\left(z^{2n+1}\right)$$

2. The Taylor series for a slightly modified function,

$$\frac{\sqrt{\pi}}{2}\frac{e^{z^2}}{z}\operatorname{erf} z=1+\frac{2z^2}{3}+...+\frac{\left(2z^2\right)^n}{(2n+1)!!}+O\left(z^{2n+1}\right).$$

3. Asymptotic expansion of $\operatorname{erfc} z$ at $z=\infty$:

$$\operatorname{erfc} z=\frac{e^{-z^2}}{z\sqrt{\pi}}\left(1-\frac{1}{2z^2}+...+\frac{(2n-1)!!}{\left(-2z^2\right)^n}+...\right).$$

4. Continued fraction expansion due to Laplace:

$$\frac{\sqrt{\pi}}{2}ze^{z^2}\operatorname{erfc} z=\cfrac{z}{z+\cfrac{\frac{1}{2}}{z+\cfrac{1}{z+\cfrac{\frac{3}{2}}{z+...}}}}$$

The continued fraction in the RHS of the above equation can be rewritten equivalently as

$$\cfrac{1}{1+\cfrac{v}{1+\cfrac{2v}{1+\cfrac{3v}{1+...}}}}$$

if we define $v\equiv\left(2z^2\right)^{-1}$.

Here we shall analyze the convergence and precision of these methods. We need to choose a good method to compute $\operatorname{erf} z$ with (relative) precision P decimal digits for a given (complex) number z, and to obtain estimates for the necessary number of terms to take.

Both Taylor series converge absolutely for all z, but they do not converge uniformly fast; in fact these series are not very useful for large z because a very large number of slowly decreasing terms gives a significant contribution to the result, and the round-off error (especially for the first series with the alternating signs) becomes too high. Both series converge well for $|z|<1$.

Consider method 1 (the first Taylor series). We shall use the method 1 only for $|z| \leq 1$. If the absolute error of the truncated Taylor series is estimated as the first discarded term, the precision after taking all terms up to and including z^{2n} is approximately $\frac{z^{2n+2}}{(n+2)!}$. The factorial can be approximated by Stirling's formula, $n! \approx n^n e^{-n}$. The value of $\operatorname{erf} z$ at small z is of order 1, so we can take the absolute error to be equal to the relative error of the series that starts with 1. Therefore, to obtain P decimal digits of precision, we need the number of terms n that satisfies the inequality

$$\left| e \frac{z^2}{n+1} \right|^{n+1} < 10^{-P}.$$

(We have substituted $n+1$ instead of $n+2$ which made the inequality stronger.) The error will be greatest when $|z| = 1$. For these values of z, the above inequality is satisfied when $n > 1 + \exp\left(1 + W\left(P \frac{\ln 10}{e}\right)\right)$ where W is Lambert's W function.

Consider method 3 (the asymptotic series). Due to limitations of the asymptotic series, we shall use the method 3 only for large enough values of z and small enough precision.

There are two important cases when calculating $\operatorname{erf} z$ for large (complex) z: the case of $z^2 > 0$ and the case of $z^2 < 0$. In the first case (e.g. a real z), the function $\operatorname{erf} z$ is approximately 1 for large $|z|$ (if $Re\,(z) > 0$, and approximately -1 if $Re\,(z) < 0$). In the second case (e.g. pure imaginary $z = \imath t$) the function $\operatorname{erf} z$ rapidly grows as $\frac{\exp\left(-z^2\right)}{z}$ at large $|z|$.

Chapter 7

References

[Abramowitz *et al.* 1964] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Washington, D.C., 1964.

[Ahlgren *et al.* 2001] S. Ahlgren and K. Ono, *Addition and counting: the arithmetic of partitions*, Notices of the AMS 48 (2001), p. 978.

[Bailey *et al.* 1997] D. H. Bailey, P. B. Borwein, and S. Plouffe, *On The Rapid Computation of Various Polylogarithmic Constants*, Math. Comp. 66 (1997), p. 903.

[Bateman *et al.* 1953] Bateman and Erdelyi, *Higher Transcendental Functions*, McGraw-Hill, 1953.

[Beeler *et al.* 1972] M. Beeler, R. W. Gosper, and R. Schroeppel, Memo No. 239, MIT AI Lab (1972), now available online (the so-called "Hacker's Memo" or "HAKMEM").

[Borwein 1995] P. Borwein, *An efficient algorithm for Riemann Zeta function* (1995), published online and in Canadian Math. Soc. Conf. Proc., 27 (2000), pp. 29-34.

[Borwein *et al.* 1999] J. M. Borwein, D. M. Bradley, R. E. Crandall, *Computation strategies for the Riemann Zeta function*, online preprint CECM-98-118 (1999).

[Brent 1975] R. P. Brent, *Multiple-precision zero-finding methods and the complexity of elementary function evaluation*, in *Analytic Computational Complexity*, ed. by J. F. Traub, Academic Press, 1975, p. 151; also available online from Oxford Computing Laboratory, as the paper `rpb028`.

[Brent 1976] R. P. Brent, *The complexity of multiple-precision arithmetic*, Complexity of Computation Problem Solving, 1976; R. P. Brent, *Fast multiple-precision evaluation of elementary functions*, Journal of the ACM 23 (1976), p. 242.

[Brent 1978] R. P. Brent, *A Fortran Multiple-Precision Arithmetic Package*, ACM TOMS 4, no. 1 (1978), p. 57.

[Brent *et al.* 1980] R. P. Brent and E. M. McMillan, *Some new algorithms for high precision computation of Euler's constant*, Math. Comp. 34 (1980), p. 305.

[Crenshaw 2000] J. W. Crenshaw, *MATH Toolkit for REAL-TIME Programming*, CMP Media Inc., 2000.

[Damgard *et al.* 1993] I. B. Damgard, P. Landrock and C. Pomerance, *Average Case Error Estimates for the Strong Probable Prime Test*, Math. Comp. 61, (1993) pp. 177-194.

[Davenport *et al.* 1989] J. H. Davenport, Y. Siret, and E. Tournier, *Computer Algebra, systems and algorithms for algebraic computation*, Academic Press, 1989.

[Davenport 1992] J. H. Davenport, *Primality testing revisited*, Proc. ISSAC 1992, p. 123.

[Fee 1990] G. Fee, *Computation of Catalan's constant using Ramanujan's formula*, Proc. ISSAC 1990, p. 157; ACM, 1990.

[Godfrey 2001] P. Godfrey (2001) (unpublished text): `http://winnie.fit.edu/~gabdo/gamma.txt`.

[Gourdon *et al.* 2001] X. Gourdon and P. Sebah, *The Euler constant*; *The Bernoulli numbers*; *The Gamma Function*; *The binary splitting method*; and other essays, available online at `http://numbers.computation.free.fr/Constants/` (2001).

[Haible *et al.* 1998] B. Haible and T. Papanikolaou, *Fast Multiprecision Evaluation of Series of Rational Numbers*, LNCS 1423 (Springer, 1998), p. 338.

[Johnson 1987] K. C. Johnson, *Algorithm 650: Efficient square root implementation on the 68000*, ACM TOMS 13 (1987), p. 138.

[Kanemitsu *et al.* 2001] S. Kanemitsu, Y. Tanigawa, and M. Yoshimoto, *On the values of the Riemann zeta-function at rational arguments*, The Hardy-Ramanujan Journal 24 (2001), p. 11.

[Karp *et al.* 1997] A. H. Karp and P. Markstein, *High-precision division and square root*, ACM TOMS, vol. 23 (1997), p. 561.

[Knuth 1973] D. E. Knuth, *The art of computer programming*, Addison-Wesley, 1973.

[Lanczos 1964] C. J. Lanczos, J. SIAM of Num. Anal. Ser. B, vol. 1, p. 86 (1964).

[Luke 1975] Y. L. Luke, *Mathematical functions and their approximations*, Academic Press, N. Y., 1975.

[Olver 1974] F. W. J. Olver, *Asymptotics and special functions*, Academic Press, 1974.

[Pollard 1978] J. Pollard, *Monte Carlo methods for index computation mod p*, Mathematics of Computation, vol. 32 (1978), pp. 918-924.

[Pomerance *et al.* 1980] Pomerance et al., Math. Comp. 35 (1980), p. 1003.

[Rabin 1980] M. O. Rabin, *Probabilistic algorithm for testing primality*, J. Number Theory 12 (1980), p. 128.

[Smith 1989] D. M. Smith, *Efficient multiple-precision evaluation of elementary functions*, Math. Comp. 52 (1989), p. 131.

[Smith 2001] D. M. Smith, *Algorithm 814: Fortran 90 software for floating-point multiple precision arithmetic, Gamma and related functions*, ACM TOMS 27 (2001), p. 377.

[Spouge 1994] J. L. Spouge, J. SIAM of Num. Anal. 31 (1994), p. 931.

[Sweeney 1963] D. W. Sweeney, Math. Comp. 17 (1963), p. 170.

[Thacher 1963] H. C. Thacher, Jr., *Algorithm 180, Error function for large real X*, Comm. ACM 6, no. 6 (1963), p. 314.

[Tsimring 1988] Sh. E. Tsimring, *Handbook of special functions and definite integrals: algorithms and programs for calculators*, Radio and communications (publisher), Moscow (1988) (in Russian).

[von zur Gathen *et al.* 1999] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 1999.

Chapter 8

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor
Boston, MA 02110-1301
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, `LaTeX` input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated

above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) YEAR  YOUR NAME. Permission is
granted to copy, distribute and/or modify this
document under the terms of the GNU Free
Documentation License, Version 1.1 or any later
version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR
TITLES, with the Front-Cover Texts being LIST, and
with the Back-Cover Texts being LIST. A copy of
the license is included in the section entitled
``GNU Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

AGM sequence, 37
  convergence rate, 37
  integral representation, 37
arbitrary-precision computation, 22
  catastrophic round-off error, 24
  requirements, 22
  round-off error estimates, 23
  speed estimates, 22
  very large numbers, 25
  very small numbers, 25
  Yacas internal math, 24
asymptotic series, 37
  estimate of precision, 37

Bernoulli numbers, 58
  asymptotic, 61
  by recurrence, 61
    cost estimate, 61, 63
    numerical stability, 62
  definition, 61
  fractional part of, 61
  from Zeta function, 61
    example, 62
Bernoulli polynomials
  definition, 61
`Bernoulli1`, 62
Bessel functions, 60
  asymptotics, 60
  by recurrence relation, 61
binary splitting, 38
binomial coefficients, 51

Clausen – von Staudt, theorem of, 61
computation of $\arctan x$, 49
  by continued fraction, 50
    convergence rate, 50
  by Taylor series, 50
computation of π, 47
  by Brent-Salamin method, 48
  by Newton's method, 48
  by Ramanujan's series, 48
continued fraction approximation
  bottom-up computation, 27
    estimated remainder, 27
  convergence rate, 29
    from generating functions, 30
  error bound, 26, 29
  estimating $\ln 2$, 24
  of $\arctan x$, 29, 49, 50
  of $\exp(x)$, 47
  of $\Gamma(a, z)$, 29
  of $\ln x$, 46
  of $\operatorname{erfc} x$, 29, 30
    error estimate, 30
  of $\tan x$, 49
  of rational numbers, 26
  top-down computation, 28
    derivation, 29
    non-alternating signs, 28

divisors, 13
  proper, 13
double factorial, 50

error function $\operatorname{erf} x$
  by asymptotic series, 64
  by asymptotic series , 37
  by Taylor series, 63
  summary of methods, 63
Euler's constant γ, 56
exponential function $\exp(x)$
  by binary reduction, 47
  by continued fraction, 47
  by inverse logarithm, 47
  by linear reduction, 47
  by Taylor series, 46
  precision, 46
  squaring speed-up, 47
exponentially large numbers, 25
exponentially small numbers, 25

`Factor`, 11
factorial, 50
  by bisection method, 50
factorization of integers, 11
  checking for prime powers, 11
  overview of algorithms, 12
  Pollard's rho algorithm, 11
  small prime factors, 11
`Factors`, 14

Gamma function, 51, 54
  by asymptotic series, 55
  by Lanczos-Spouge method, 54
  by Sweeney-Brent method, 56
  half-integer arguments, 54
  rational arguments, 54
Gaussian integers, 13
GCD
  binary Euclidean method, 10
generating function of a sequence, 30
  integral representation, 30
  obtaining, 30
`GuessRational`, 26

Halley's method, 32
  explicit formula, 32
  when to use, 33
Horner's scheme, 35

`IntNthRoot`, 43
`IntPowerNum`, 40

Lambert's W function
asymptotics, 60
by Halley's method, 60
uniform approximations, 60
logarithm
by AGM sequence, 44
by argument reduction, 45
by binary reduction, 45
by continued fraction, 46
by inverse exponential, 44
by square roots, 44
by Taylor series, 44
by transformed series, 45
choice of AGM vs. Taylor, 45
on integers, 43
precision, 44

method of steepest descent, 30
example for real x, 31

`NearRational`, 27
Newton's method, 32
asymptotic cost, 34
cubic convergence, 32
higher-order schemes, 33
initial value, 32
optimal order, 34
precision control, 33
Newton-Cotes quadratures, 21
for partial intervals, 21
`NextPrime`, 11

orthogonal polynomials, 51
by specific formulae, 52
classical polynomials, 51
Clenshaw-Smith recurrence, 53
generating function, 51
Rodrigues formula, 51

partitions of an integer, 12
by Rademacher-Hardy-Ramanujan series, 12
by recurrence relation, 13
`Plot2D'adaptive`, 20
plotting
adaptive algorithms, 20
non-Euclidean coordinates, 22
of surfaces, 21
parametric, 22
three-dimensional, 21, 22
two-dimensional, 20
powers, 40
by repeated squaring, 40
improvements, 40
non-recursive, 40
modular, 40
real numbers, 41
primality testing, 10
Fermat test, 10
Miller-Rabin algorithm, 10
choosing the bases, 11
strong pseudoprimes, 11

real roots, 7
bounds on, 8
finding, 8
number of, 8
Sturm sequences, 7
variations in Sturm sequences, 8
Riemann's Zeta function, 58
by Borwein's algorithm, 58
by direct summation, 58
by product over primes, 59
integer arguments, 59
roots
by argument reduction, 42
by bisection algorithm, 41
by Halley's method, 43
by higher-order methods, 43
by Newton's method, 42

square free decomposition, 7
Stirling's formula, 30, 32, 54, 61
sums of integer powers, 61

Taylor series, 35
$O\left(\sqrt[3]{N}\right)$ method, 36
baby step/giant step method, 35
by Horner's scheme, 35
rectangular method, 35
required number of terms, 23