Arbitrary Precision Math C++ Package

By: Henrik Vestermark
(hve@hvks.com)

Revised: 24 March 2024

Revision History

Revision Date	Change
2003/06/25	Initial Release
2007/08/26	Add the Floating point Epsilon function Add the ipow() function. Integer raise to the power of an integer
2013/Oct/2	Added new member functionality and expanded the explanation and usage of these classes.
2014/Jun/21	Cleaning up the documentation and adding a method to int_precision() and toString()
2014/Jun/25	Added abs(int_precision) and abs(float_precision)
2014/Jun/28	Updated the description of the interval packages
2016/Nov/13	Added the nroot()
2017/Jan/29	Added the transcendental constant *e*
2017/Feb/3	Added gcd(), lcm() and two new methods to int_precision(), even() & odd()
2019/Jul/22	Added fraction Arithmetic packages. Added more examples if usage in Appendix C & D
2019/Jul/30	Added 3 methods to Float_precision: .toFixed(), .toPrecision() & .toExponential()
2019/Sep/17	Change the class interface to move the sign out into a separate variable. _int_precision_atoi() now also returns the sign instead of embedding it into the string
2020/Aug/12	Added Appendix E with compiler information
2021/Mar/22	Added missing information about Trigonometric functions for complex arguments and Hyperbolic functions for complex arguments
2021/Mar/24	Added the float precision operator %, %= (same as the function fmod)
2021/Jul/30	Added more functionality to the interval package e.g. hyperbolic, trigonometric functions, and interval constants. Fixed some typos in complex precision
1-Nov-2021	Revised completely to describe the new internal binary format for arbitrary precision. Added &=,|=,^=,&,|,^as new operators for int_precision. Furthermore added the following new methods. testbit(), flipbit(),setbit(),resetbit(), ctz(), clz(), iszero(), number(). For float_precision the following method was added: number(),index(), size(), iszero(), toInteger(), toFraction()
11-Jan-2022	Added more constants to the _float_table() functions. _INVSQRT2, _SQRT2, _INVSQRT3, _SQRT3 and clean up the manual.
23-Mar-2022	Added _ONTENTH as a constant and introduce dynamic fixed-size integers
21-May-2022	Added log2() and AGM() for float_precision and .square() and .inverse() method
31-Oct-2022	Added nextafter() with the same functionality as in the C++ library. Also added methods: .succ() and .pred() to the float_precision class
12-Dec-2022	Added factorial(), fallingfactorial(), binomial() as int_precision functions. Added Bernoulli() as fraction_precision function and added a section for API method and API function for fraction_precision. Added bernoulli(), bernoulliPolynomials for float_precision objects

Revision History

13-Jan-2022	Added the Euler-Mascheroni and the Catalan constant. Added the float_precision function tgamma(), beta(), erf() and erfc()
17-Jan-2023	Added Lambert Wo function
20-Jan-2023	Added zeta function
25-Mar-2023	Added Stirling number of the first, second, and the third kind
11-May-2023	Added the computation of the Jacobi symbol, and the probabilistic primality tester. Miller-Rabin and Baillie PSW
20-Jun-2023	Added arbitrary precision pseudo-random number generators (PRNGs)
15-Aug-2023	Added arbitrary precision Fibonacci sequence and added more info on how to compile the source code
15-Sep-2023	Added the rising factorial and also changed the description of the falling factorial
3-Oct-2023	Added norm() and conj() as functions to be compatible with the STL library <complex>. Modernize the Appendix B sample program
21-Mar-2024	Added nextafter() and fma() from the C++ standard library. Plus introduce the ∞ constant FP_INFINITY serving the same purpose as the C++ INFINITY and FP_QUIET_NAN in accordance with the IEEE754
23-Mar-2024	Added isinf(), isnan(), isnormal(), isfinite()

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Arbitrary Precision Math C++ Package

Introduction

C++'s data types for integer, single and double precision floating point numbers, and the Standard Template Library (STL) complex class are limited in the amount of numeric precision they provide. The following table shows the range of the standard built-in and complex STL data type values supported by a typical C++ compiler:

Class	Storage Allocation (bytes)	Range
short	2	-32768 ≥ N ≤ +32767
unsigned short	2	0 ≤ N ≤ 65535
int	4	–2147483646 ≥ N ≤2147483647
long	4	–2147483646 ≥ N ≤ +2147483647
unsigned int	4	0 ≤ N ≤ 4294967295
long long	8	-9223372036854775807 ≥ N ≤9223372036854775807
long long	8	0 ≤ N ≤ 18446744073709551615
int64_t	8	-9223372036854775807 ≥ N ≤9223372036854775807
uint64_t	8	0 ≤ N ≤ 18446744073709551615
float	4	1.175494351E-38 ≤ |N| ≤ 3.402823466E+38
double	8	2.2250738585072014E-308 ≤ |N| ≤ 1.7976931348623158E+308
complex	4 or 8	See `float` and `double`

The above numeric precision ranges are adequate for most uses but are inadequate for applications that require either, very large magnitude whole numbers, or very large small and precise real numbers. When an application requires greater numeric magnitude or precision, other techniques need to be used.

The C++ classes described in this manual greatly extend the limited range and precision of C++'s built-in classes:

Class	Usage
int_precision	Whole (integer) numbers
float_precision	Real (floating point) numbers
complex_precision	Complex numbers
interval_precision	Interval arithmetic
fraction_precision	Fraction arithmetic

The two first classes, `int_precision,` and `float_precision` support basic arbitrary precision math for integer and floating point (real) numbers and are written as concrete classes. The `complex_precision, interval_precision,` and `fraction_precision` classes are implemented as template classes, which support `int_precision` or `float_precision (float_precision is not supported in fraction_precision)` objects, as well as the ordinary C++ built-in `float` or `double` data types.

Both the `complex_precision` and `interval_precision` classes can work with each other; therefore, it is possible to create an interval object using a `complex_precision` object, or a complex object using `interval_precision` objects. Normally, `complex_precision` and `interval_precision` objects are built using `float_precision` objects.

This version of the manual describes the new internal binary format and the added functionality.

Compiling the source code

The source consists of five header files and one C++ source file:

iprecision.h
fprecision.h
complexprecision.h
intervalprecision.h
fractionprecision.h
precisioncore.cpp

The header files are used as include statements in your source file and your source file(s) need to be compiled together with precisioncore.cpp which contains the basic C++ code for supporting arbitrary precision. There are usually two ways to do this. Either you can create a monolith single file source code by compiling the sample file below.

Sample file 1:

```
#include “precisioncore.cpp”
#include “complexprecision.h” // Optional. Only needed for complex arithmetic in arbitrary precision
#include “intervalprecision.h” // Optional. Only needed for interval arithmetic in arbitrary precision

int main() {
	// Your program
}
```

Or you can create a project in an IDE. E.g. CodeBlock or Visual Studio. A sample file for your code could be.

Sample file 2:

```
#include “iprecision.h”		// For arbitrary integer arithmetic
#include “fprecision.h”		// Optional. For arbitrary floating-point arithmetic,
#include “fractionprecision.h”		// Optional For arbitrary fraction arithmetic
#include “complexprecision.h” // Optional. Only needed for complex arithmetic in arbitrary precision

#include “intervalprecision.h” // Optional. Only needed for interval arithmetic in arbitrary precision
```

```
int main() {
	// Your program
}
```

And then you have to add the file precisioncore.cpp to your project as well.

The source has been developed, tested, and compiled under Microsoft Visual C++ 2022 compiler. See Appendix E for additional compiler info.

Configuration is not needed except for two options. By default, multi-threading is enabled and implemented were it makes sense. This will increase the performance of many operations. HVE_THREAD is defined by default in the file precisioncore.cpp and in the case it should not be enabled then #undef this definition.

The other configuration is in the file: iprecision.h
Here #define _INT_PRECISSION_FAST_DIV_REM
This means that int_precision division/Remaining is carried out using float_precision division/remaining. There is a very efficient method for float_precision division that is faster than an int_precision division.

Arbitrary Precision Math C++ Package

Arbitrary Integer Precision Class

Usage

To use the integer precision class the following include statement needs to be added to the top of the source code file(s) in which arbitrary integer precision is needed:

```
#include “iprecision.h”
```

An arbitrary integer precision number (object) is created (instantiated) by the declaration:

```
int_precision myVaribleName;
```

An `int_precision` object can be initialized in the declaration in many different ways. The following examples show the supported forms for initialization:

```
int_precision i1(1);      // Decimal number
int_precision i2(‘1’);    // Char number
int_precision i3(“123”);  // String
int_precision i4(0377);   // Octal
int_precision i5(0x9Af);  // Hexadecimal
int_precision i6(0b01011);// Binary
int_precision i7(i1);     // Another int_precision object
```

In the same manner, `int_precision` objects can also be initialized/modified directly after instantiation. For example:

```
int_precision i1 = 1;     // Decimal
int_precision i2 = ‘1’;   // Char. Stored as the binary value of ‘1’
int_precision i3 = “123”; // String
int_precision i4 = 0377;  // Octal
int_precision i5 = 0x9Af; // Hexadecimal
int_precision i6 = i1;    // Another int_precision object
```

Please note that decimal string can contain ‘ or _ to make the number more readable. E.g.

```
int_precision i7 = “123’000’000”;      // String
int_precision i8 = “123_000_000”;      // String
```

The ‘ or _ is simply ignored by the software

Initialization of int_precision creates an arbitrary precision integer variable that can grow to any arbitrary size. E.g. 1M digits, 1Billion digits, etc. However, it can also be fixed by limiting the size to any number of 64-bit trunks. E.g. to create an integer capable of holding 128 bits of information.

```
int_precision i128(“235689”, 2) ;      // 128bit fixed sized integer
int_precision i1024(“235689”, 16) ;    // 1024bit fixed sized integer
```

The 2nd optional parameter is the number of 64-bit trunks that the integer can hold. If omitted the number can growth arbitrary. A fixed-size integer can be changed to another fixed size or unlimited using the method precision().

Arithmetic Operations.

The arbitrary integer precision package supports the flowing C++ integer arithmetic operators: `+, -, ++, --, /, *, %, <<, >>, +=, -=, *=, /=, %=, <<=, >>=, |,&,^,|=,&=,^=`

The following examples are all valid statements:

```
i1=i2;
i1=i2+i3;
i1=i2-i3;
i1=i2*i3;
i1=i2/i3;
i1=i2%i3;
i1=i2>>i3;
i1=i2<<i3;
i1=i1&i2;
i1=i1|i2;
i1=i1^i2;
```

and

```
i1*=i2;
i1-=i2;
i1+=i2;
i1/=i2;
i1%=i2;
i1<<=i2;
i2>>=i1;
i2&=i1;
i2|=i1;
i2^=i1;
```

Following are examples using the unary ++ (increment), -- (decrement), and – (negation) (including + positive)

```
i1++;    // Post-increment
--i3;    // Pre-decrement
i2=-i1;
i2=+i1;
```

The following standard C++ test operators are supported: `==,!=,<,>,<=,>=`

```
if( i1 > i2 )
      …
else
      …
```

The `int_precision` package also includes 12 demotion member functions for converting `int_precision` objects to either `char, short, int, long, int64_t, long long or the unsigned versions, unsigned char, unsigned short, unsigned int, unsigned long, unsigned uint64_t, unsigned long long or float,` `double` standard C++ data types or the corresponding unsigned integer types.

Note: Overflow or rounding errors can occur.

```
int i;
double d;
int_precision ip1(123);

i=(int)ip1;     // Demote to int. Overflow may occur
d=(double)ip1; // Demote to double. Overflow/rounding may occur
```

Math Member Functions

The following set of public member functions is accessible for `int_precision` objects:

```
int_precision   abs( int_precision );  // abs(i)
int_precision   ipow( int_precision, int_precision ); // a^b
int_precision   ipow_modulo( int_precision, int_precision, int_precision ); //
a^b%c
bool            isprime( int_precision );  // Test number for a prime
int_precision   gcd(int_precision, int_precision ); //gcd(a,b)
int_precision   lcm(int_precision, int_precision ); //lcm(a,b)
int_precision_  iptoa()
```

Input/Output (iostream)

The C++ standard ostream << operator has been overloaded to support the output of `int_precision` objects. For example:

```
cout << “Arbitrary Precision number:” << i1 << endl;
```

The `int_precision` class also has a convert to string member function: `_int_precision_itoa(char*)`

```
int_precision i1(123);
std::string s;

s=_int_precision_itoa( &i1 );
cout << s.c_str();
```

The C++ standard istream >> operator has also been overloaded to support the input of `int_precision` objects. For example:

```
cin >> i1;
```

Exceptions

The following exceptions can be thrown under the `int_precision` package:

```
bad_int_syntax      // Thrown if initialized with an illegal number
                    // For example: "123$567" is illegal because
                    // '$' is not a valid character for a numeric number.
out_of_range        // Thrown when attempting to shift with a negative
                    // value using the << or >> operator.
divide_by_zero      // Thrown if dividing by zero.
domain_error        //  Thrown when outside domain range
```

Mixed Mode Arithmetic

Mixed mode arithmetic is supported in the `int_precision` class. An explicit conversion to an `int_precision` object can of course be done to avoid any ambiguity for the compiler. For example:

```
int_precision a=2;

a=a+2;  // can produces compilation error: ambiguous + operator
a=a+int_precision(2); // Compiles OK
```

Be on the watch for ambiguous compiler operator errors!

Class Internals

Most of the `int_precision` class member functions are implemented as `inline` functions. This provides the best performance at the sacrifice of increased program size.

The arbitrary precision integer package stores numbers as a vector of *iptype*. *iptype* is the usual 64-bit unsigned integer. This allows for more efficient use of memory and speeds up calculations dramatically. Each *iptype* can hold up to 18+19 decimal digits.

This arbitrary integer precision package was designed for ease of use and transparency rather than speed and code compactness. No doubt, there are other arbitrary integer packages in existence with higher performance and requiring fewer memory resources.

Member Functions

The following member methods are also available:

Method	Description
abs()	Change the int_precision object to its absolute value
change_sign()	Reverse the sign.
clz()	Count leading zeros in the mBinary number
ctz()	Count trailing zeros in the mBinary number.
even()	Return true if the mBinary number is even otherwise false.
flipbit()	Flip a specific bit in the mBinary number
iszero()	Return true if the mBinary number is zero otherwise false

number()	Returns or set a copy of the mBinary field.
odd()	Return true if the mBinary number is odd otherwise false.
pointer()	Returns a pointer to the mBinary fields that contain the binary number.
precision()	Get or set integer precision
resetbit()	Reset a specific bit in the mBinary number.
setbit()	Set a specific bit in the mBinary number.
sign()	Returns or set the sign of the int_precision number. The sign bit is either +1 or -1.
size()	Returns the current size of the Binary elements the mBinary field holds.
testbit()	Test a specific bid in the mBinary number and return true or false
toString()	Return the Binary number as a decimal string in base 10 (default), but other bases are supported as well

Internal storage handling

The Class `int_precision` has two public elements:

int mSign;	// Sign of the number. Either +1 or -1
vector<iptype> mBinary;	// The binary vector of iptype that holds the integer. Per definition, the vector when the constructor is invoked will always be initialized to zero if no argument is provided.

The *iptype* is by default set to the maximum unsigned integer *uintmax_t* which on most systems is a 64-bit unsigned integer. This means per vector entry an *iptype* holds approximately 18-19 decimal digits. This from a storage point of view makes it much more efficient compared to the previous version which only old one decimal digit per byte. In the previous version, the number was a decimal number stored as a character in the STL library string class. While the newer binary version stores it as an STL vector of iptype. If your system doesn't support a 64-bit environment then the *uintmax_t* is set to a 32-bit unsigned int (or you can do it manually by setting the *iptype* to an unsigned int when running in a 32-bit environment). By using the STL library vector class we can hide and don't worry about how the vector class handles memory allocation, resizing, etc. greatly simplifying the code to handle arbitrary integer precision. This also makes the source code easy to read and comprehend.

Room for Improvement

In the latest version, I have added multi-threading to speed up the calculation of multiplication. However, due to the overhead of creating threads, it is first kicked in when numbers exceed 100,000 digits.

API Methods for int_precision

(*int precision object*).abs()

Change the int_precision object to its absolute value and return it

(*int precision object*).change_sign()

Reverse the sign nd return the new sign as either -1 or +1.

(*int precision object*).clz()

Count leading zeros in the mBinary number. And return the number of zero leading bits.

(*int precision object*).ctz()

Count tailing zeros in the mBinary number and return the number of trailing zero bits

.

(*int precision object*).even()

Return true if the mBinary number is even otherwise false.

(*int precision object*).flipbit(size_t bitpos)

Flip a specific bit at position *bitpos* in the mBinary number.

(*int precision object*).iszero()

Return true if the mBinary number is zero otherwise false.

(*int precision object*).number(vector<iptype> &mb)

Returns or set a copy of the mBinary number. If the optional parameter mb is missing, you return a copy of the current mBinary number otherwise you set mBinary to the parameter mb and return it.

(*int precision object*).odd()

Return true if the mBinary number is odd otherwise false.

(*int precision object*).pointer()

Returns a pointer to the mBinary number that contains the binary number.

(*int precision object*).precision(size_t p)

If p is omitted, the current integer precision is returned as the number of 64bit elements, otherwise, the precision is set to p and the value returned. If a new precision is set, the number will be set to that precision. If the actual size is greater than the new precision the integer number will be truncated.

(*int precision object*).resetbit(size_t bitpos)

Reset a specific bit at position bitpos in the mBinary number.

(*int precision object*).setbit(size_t bitpos)

Set a specific bit at position *bitpos* in the mBinary number.

(*int precision object*).sign(int newsign)

Returns or set the sign of the int_precision number. If called with the parameter new sign the sign is set to newsign and returned. If omitted the current sign is returned for the number. The sign is either +1 or -1.

(*int precision object*).size()

Returns the size of the Binary elements that the mBinary field currently holds. Since the mBinary field is of type vector<iptype> we just call and return the (vector object).size method.

(*int_precision object*).testbit(size_t bitpos)

Test a specific bid at biposition *bitpos* in the mBinary number and return true or false if the bit is set (1) or reset (0).

(*int precision object*).toString(int base)

Return the Binary number as a decimal STL string in base 10 (default. The parameter is optional. Otherwise in the base indicated)

API functions for int_precision

int_precision abs(const int_precision& x); // abs(i)

Return the absolute value of the int_precision number x

bool baillie_PSW(const int_precision& p, const bool strong) // Test number for a prime

Test the number for a prime using the Baillie PSW primality test. Return true if it is a prime otherwise false. The parameter "strong" indicate whether to use an extra strong Lucas sequence test (strong=true). Baillie PSW is a probabilistics tester.

int_precision binomial(const int_precision& n, const int_precision& m);

Return the binomial. $\binom{n}{m} = \frac{n!}{m!(n-m)!}\ where\ 0 \le m \le n$

int_precision factorial(const int_precision& n); // n!

Return the n! factorial

int_precision fibonacci(const int_precision& n); // Fibonacci(n)

Return the n^{th} Fibonacci sequence

int_precision fallingfactorial(const int_precision& n, const int_precision& m);

Return the falling factorial $(n)^{\underline{m}} = \prod_{k=0}^{m-1}(n-k) = \frac{n!}{(n-m)!}$.

int_precision gcd(const int_precision& a, const int_precision& b) //gcd(a,b)

Return the greatest common divisor of the two numbers and b

int_precision ipow(const int_precision& a, const int_precision& b) // a^b

Return the int_precision number a raise to the power of b. a^b

int_precision ipow_modulo(const int_precision& a, const int_precision& b, const int_precision& c) // a^b%c

Return a raise to the power of b modulo c.

string int_precision_iptoa(const int_precision *a, const int base=10)

Convert and return the int_precision number a to an STL string using base as the base. The default is decimal base. Other valid bases are from 2..36

bool isprime(const int_precision& p, const int k=0) // Test number for a prime

Test the number for a prime. Return true if it is otherwise false. This function was previously called iprime() with the same functionality. An extra optional parameter has been added and that is k. When k is zero it default to prime testing using the brute force method of testing is number via a trial division. If k is greater than zero it uses the Miller-Rabin primality tester and k is the number of rounds (or iterations) for the Miller-Rabin test. Miller Rabin is a probabilistic tester and the probability of a correct answer is 0.25^k.

int_precision jacobi(const int_precision& a, const int_precision& n)

Return the jacobi(a/n) symbol. Where a is an odd integer and is a positive odd integer n.

int_precision lcm(const int_precision& a, const int_precision& b) //lcm(a,b)

Return the least common multiplier of a and b

bool miller_rabin(const int_precision& p, const int k=0) // Test number for a prime

Test the number for a prime using the Miller-Rabin primality test. Return true if it is a prime otherwise false. The parameter k is the number of rounds (or iterations) for the Miller-Rabin test. Miller Rabin is a probabilistic tester and the probability of a correct answer is 0.25^k.

int_precision risingfactorial(const int_precision& n, const int_precision& m);
Return the rising factorial. $(n)^{\overline{m}} = \prod_{k=0}^{m-1}(n+k) = \frac{(n+m-1)!}{(n-1)!}$.

int_precision_stirling_first(const int_precision& n, const int_precision&k, const bool sign=false)

Return the Stirling number of the first kind s(n,k) or c(n,k).

int_precision_stirling_first(const int_precision& n, const int_precision&k)

Return the Stirling number of the Second kind S(n,k).

int_precision_stirling_third(const int_precision& n, const int_precision& k, const bool sign=false)

Return the Stirling number of the third kind L(n,k) or L'(n,k) for unsigned and signed. Stirling number of the third kind is also known as the Lah number

Arbitrary Precision Pseudo Random number Class

The arbitrary precision random number class is a kind of subclass of the int_precision class. It is located in the header iprecision.h. This means you don't need to do anything special to get access to the class.

What the random_precision class does is that it uses the underlying random generators method like the ones available in the C++ standard library or others available to create an arbitrary size of a pseudo-random number. Like the built-in PRNGs in the C++ library, it is defined as a templated random_precision class that takes two arguments. The first is the class of the underlying PRNG and the second parameter is the type of the random number.

```
template<class _prng, class _rettype = uint64_t> class random_precision;
```

examples of declarations:
```
random_precision<mt19937_64> genmt19937_64bit;
random_precision<mt19937,uint32_t> genmt19937_32bit;
random_precision<ranlux24,uint32_t> genranlux24_32bit;
random_precision<ranlux48,uint64_t> genranlux48_64bit;
```

All of the random classes in the C++ library are supported plus the following that is part of the arbitrary precision library.

Type name	Family	Return_type
minstd_rand	Linear congruential generator	32-bit
mt19937	Mersenne twister	32-bit
mt19937_64	Mersenne twister	64-bit
ranlux24	Subtract and carry	32-bit

ranlux48	Subtract and carry	64-bit
knuth_b	Shuffle linear congruential generator	32-bit
default_random_engine	Any of the above (implementation-defined)	32-bit or 64-bit*
rand()	Linear congruential generator	32-bit
Xoshiro family	Scramble linear	64-bit
Chacha20	"quarter round"	32-bit

The xoshiro family and chacha20 PRNG definition are in the int_precision.h file. Notice that the xoshiro family of PRNGs comes in four forms:

1. xoshiro256pp
2. xoshiro256ss
3. xoshiro512pp
4. xoshiro512ss

And now the next random precision number is just a call to the operator () using the first example:

```
genmt19937_64bit ();    // next random number
```

Now contrary to the build in PRNG that all return a fixed size of either 32-bit or 64-bit we have a different scenario with arbitrary precision. Therefore, there is an optional added parameter to the () operator indicating the size in bits of the maximum size of the random number returned. The () operator is defined as:

```
int_precision operator()(const uintmax_t bitcnt = 64)
```

where the default size is a 64-bit return type if not specified in the operator () call.

For example:

```
genmt19937_64bit (256);    // return a random number [0…2^256-1]
genmt19937_64bit (1000);   // return a random number [0…2^1000-1]
```

If the parameter is omitted it returns a random number in the range $[0\ldots2^{64}\text{-}1]$ which default to the size of the template parameter _prng.

The design of the random_precision class is modeled after the C++ build in PRNGs. It therefor supports the following methods.:

Method	Description
(Constructor)	Constructor for the random_precision class.
discard	Discard the next numbers of PRNG
max	Return the maximum value it can return based on the size in bits parameter
min	Return the minimum value (which is 0)
seed	Seed based on a single value of the use of the seed_seq class

operator==	Return the Boolean value of the comparison of two random generators' internal state
operator!=	Return the Boolean value of the comparison of two random generators' internal state
operator()	Return the next PRNG

API Methods for random_precision

(*random precision object*).(Constructor)

Takes two forms. Either we zero or one parameter indicated an initial seed value. The seed value can be am int_precision number or a seed_seq class. See the C++ reference manual for the seed_seq class. If no parameter is present then a default random number from the std::random_device is used.

(random *precision object*).discard(unsigned long long z)

Discard the next z random number generated. This is equivalent to calling the random precision object operator() z times.

(random *precision object*).discard(unsigned long long z)

Discard the next z random number generated. This is equivalent to calling the random precision object operator() z times.

(random *precision object*).min()

Return the minimum value the random _precision object can return. This is zero.

(random *precision object*).max(uintmax_t bitcount=64)

Return the maximum value the random _precision object can return. Since int_precision can be arbitrarily large it uses the optional parameter to determine the maximum size as $2^{bitcount}$-1.

(random *precision object*).seed(int_precision& seed)

(random *precision object*).seed(seed_seq& seed)

Use the seed value on the random _precision object. The seed value can also take the form of the parameter std::seed_seq&.

bool (random *precision object*).operator==(random_precision& rhs)

Comparing the two random precision objects for equality (same internal state). If it is the same it returns true otherwise false.

bool (random *precision object*).operator!=(random_precision& rhs)

Comparing the two random precision objects for equality (same internal state). If it is the same it returns false otherwise true.

int_precision (random *precision object*).operator(uintmax_t bitcount=64)

Return the next random precision number maximum value the random _precision object can return. Since int_precision can be arbitrarily large it uses the optional parameter to determine the maximum size it can return as $2^{bitcount}$-1.

Arbitrary Precision Math C++ Package

Arbitrary Floating Point Precision Class

Usage

To use the floating point `float_precision` class the following include statement must be added to the top of the source code file(s) in which arbitrary floating point precision is needed:

```
#include “fprecision.h”
```

The syntactical format for an arbitrary floating point precision number follows the same syntax as for regular C style single precision floating point (`float`) numbers:

[*sign*][s*digit*][.[f*digit*]][E|e[e*sign*][e*digits*]]

sign	Leading sign. Either + or – or the leading sign can be omitted
sdigit	Zero or more significant digits
fdigit	Zero or more fraction digits.
esign	Exponent sign, can be either + or – or omitted.
Edigits	One or more exponent decimal digits.

Following are examples of valid `float_precision` numbers:

```
+1
1.234
-.234
1.234E+7
-E6
123e-7
```

An arbitrary floating point precision number (object) is created (instantiated) by the declaration:

```
float_precision f;
```

A `float_precision` object can be initialized at declaration (instantiation) either through its constructor or by assignment. A `float_precision` object can be initialized with an ordinary C++ built-in `short, int, long, float, double, char, string` data type, or even another `int_precision` or `float_precision`. For example:

```
float_precision f1(-1);                // Decimal
float_precision f2(‘1’);               // Char. The binary value 49
float_precision f3(“123.456E+789”);    // String
float_precision f4(0377);              // Octal
float_precision f5(0x9Af);             // Hexadecimal
float_precision f6(0b010101);          // Binary
float_precision f7(-123.456E78);       // Float
```

```
float_precision f1 = -1;              // Decimal
float_precision f2 = '1';             // Char. The binary value 49
float_precision f3 = "123.456E+789";  // String
float_precision f4 = 0377;            // Octal
float_precision f5 = 0x9Af;           // Hexadecimal
float_precision f6 = 0x9Af;           // Binary
float_precision f7 = -123.456E78;     // Float
float_precision f8 = f1;              // Another float_precision
float_precision f9 = int_precision(13);// Through int_precision
```

Please note that decimal string can contain ' or _ to make the number more readable. E.g.

```
float_precision f10 = "-123.456'789";  // String
float_precision f11 = "-123.456_789";  // String
```

The ' or _ is simply ignored by the software

Initialization with the constructor also allows precision (number of significant digits) and a rounding mode to be specified. If no precision or rounding mode is specified the default precision value of 20 significant decimal digits, and a rounding mode of *nearest* (the default behavior according to IEEE 754 floating-point standard) is used.

For example, to initialize two objects, one to 8 and the other to 4 significant digits of precision, the declarations would be:

```
float_precision f1(0,8); // Initialized to 0, with 8 digits
float_precision f2("9.87654",4);
```

In the above example, `f2` is initialized to `9.877` because only four digits of significance had been specified. Please note that the initialization value of `9.87654` is rounded to the nearest 4th digit. The precision specification or default precision has precedence over the precision of the expressed value being used to initialize a `float_precision` object. This behavior is consistent with standard C. For example: in the following declaration…

```
int i=9.87654;
```

the variable `i` is initialized to the integer value of 9 in C.

In a declaration that uses the `float_precision` constructor a rounding mode can also be given. The default rounding mode is "round to nearest" (i.e. ROUND_NEAR). However, "round up" or "round down" or "round towards zero" behaviors are also possible. See *Floating Point Precision Internals* for an explanation of rounding modes.

Here are some examples of various rounding mode behaviors.

```
float_precision PI("3.141593", 4, ROUND_NEAR);  //3.142 default
float_precision PI("3.141593", 4, ROUND_UP);    //3.142
float_precision PI("3.141593", 4, ROUND_DOWN);  //3.141
```

```
float_precision PI("3.141593", 4 , ROUND_ZERO); //3.141

float_precision negPI("-3.141593", 4, ROUND_NEAR); //-3.142 default
float_precision negPI("-3.141593", 4, ROUND_UP);   //-3.141
float_precision negPI("-3.141593", 4, ROUND_DOWN); //-3.142
float_precision negPI("-3.141593", 4 , ROUND_ZERO);//-3.141
```

Arithmetic Operations

The following C/C++ arithmetic operators are supported in fprecision package: +, -, *, /, %, and the unary version of + and -. Plus all the assign operators e.g. +=,-=,*=,/=,%=

For example:

```
float_precision f1,f2,f3;

f1=f2+f3;
f2=f3/f1;
f3*=float_precision(1.5);

// Casts to standard C++ types are also supported.

Int i, double d;

i=(int)f1;      // Loss of precision may occur
d=(double)f1;   // Loss of precision may occur
```

Truncation will occur if `f1` exceeds the value of the integer or the double.

Math Member Functions

The following set of public member functions is available for `float_precision` objects:

```
float_precision   log( float_precision );
float_precision log2( float_precision );
float_precision log10( float_precision );
float_precision   exp( float_precision );
float_precision  sqrt( float_precision );
float_precision   pow( float_precision, float_precision );
float_precision nroot( float_precision, int );

float_precision  fmod( float_precision, float_precision );
float_precision floor( float_precision );
float_precision  ceil( float_precision );
float_precision  modf( float_precision, float_precision );
float_precision   abs( float_precision );
float_precision  fabs( float_precision ); // Same as abs()
float_precision frexp( float_precision, int* );
float_precision ldexp( float_precision, int );
float_precision AGM( float_precision, float_precision );
float_precision nextafter( float_precision, float_precision );
float_precision fma( float_precision, float_precision, float_precision);
```

```
// Trigonometric functions
float_precision   sin( float_precision );
float_precision   cos( float_precision );
float_precision   tan( float_precision );
float_precision  asin( float_precision );
float_precision  acos( float_precision );
float_precision  atan( float_precision );
float_precision atan2( float_precision, float_precision );

// Hyperbolic functions
float_precision   sinh( float_precision );
float_precision   cosh( float_precision );
float_precision   tanh( float_precision );
float_precision  asinh( float_precision );
float_precision  acosh( float_precision );
float_precision  atanh( float_precision );

// Special functions
float_precision  beta( float_precision );
float_precision  erf( float_precision );
float_precision  erfc( float_precision );
float_precision  gamma( float_precision );
float_precision  lambertW0( float_precision );
float_precision  zeta( float_precision );

// is… functions
bool isinf(float_precision );
bool isfinite(float_precision )
bool isnan(float_precision);
bool isnormal(float_precision );

// Miscellaneous functions
float_precision   bernoulli(size_t, size_t);
float_precision   bernoulliPolynomials( float_precision,  size_t)
```

Theses function returns the result in the same precision as the argument. E.g.

```
float_precision f1(0.5,10),f2(0.5,200),f3(0.5,300);

sin(f1); // return sin(0.5) with 10 digits precision
sin(f2); // return sin(0.5) with 200 digits precision
sin(f3); // return sin(0.5) with 300 digits precision
```

Built-in Constants

The fprecision package also provides three ‘constants’:

Constant	Description
_PI	One-half the ratio of a circle’s circumference to its radius

`_LN2`	Natural logarithm $base_e$ of 2
`_LN10`	Natural logarithm $base_e$ of 10
`_EXP1`	e
`_INVSQRT2`	Inverse square root 2. 1/sqrt(2)
`_SQRT2`	Square root 2. Sqrt(2)
`_INVSQRT3`	Inverse square root 3. 1/sqrt(3)
`_SQRT3`	Square root 3. Sqrt(3)
`_ONETENTH`	1/10 to the required precision
`_EULER`	The Euler-Mascheroni constant
`_CATALAN`	The Catalan constant
`_ZETA3`	The Zeta(3) constant
`FP_INFINITY`	Equivalent to the INFINITY constant in C++. $1\cdot2^{INTMAX\_MAX}$
`FP_QUIET_NAN`	Equivalent to the NAN constant in C++. Value is $0\cdot2^{INTMAX\_MAX}$

These are no true C++ constants (except for FP_INFINITY and FP_QUIET_NAN), but they are variables that can be created with varying degrees of precision. The float_precision behaves like recommend in IEEE754 for arithmetic operations and argument with a NAN also behaves as in the equivalent C library functions.
To use one of these constants, a call must be made to the function `_float_table()` to calculate (initialize) the constant to the requested precision.

The `_float_table()`function remembers the most precise constant's precision calculation and if subsequent call requests equal or less precision the constant will be truncated and rounded to the requested precision. When more precision is requested, a new calculation of the constant is performed and stored.

For _INVSQRT2, _SQRT2, _INVSQRT3, _SQRT3 the functions are implemented as newton iteration with restart from previous precision. What this means is that if we first call e.g. _float_table(_SQRT2, 20000); it will go through approx. 9 iterations to reach the desired precision. If later on called with a request for 100,000 digits that normally required 13 iterations, we can just restart the iterations from the 20,000 digits mark and continue up t 100,000 digits precision only requiring 4 additional iterations instead of 13.

Example usage:

```
float_precision PI;
PI=_float_table(_PI,20);    // Compute _PI to 20 digits.

PI=_float_table(_PI,10);    // No need for recalculation since
                            // the initial value was computed to
                            // 20 digits of precision.

PI=_float_table(_PI,15);    // No need for recalculation since
                            // the initial value was computed to
                            // 20 digits of precision.

PI=_float_table(_PI,25);    // Recalculation required because
```

```
                                  // the initial value was computed to
                                  // 20 digits of precision.
```

Input/Output (iostream)

The C++ standard ostream << and istream >> operators have been overloaded to support output and input of `float_precision` objects. For example:

```
cout << fp1 << endl;

cin >> fp1 >> fp2;  // Input two float_precision numbers
```

Other Member Functions

The following set of public member functions (methods) are accessible for `float_precision` objects:

```
// float_precision to String
string _float_precision_fptoa(float_precision *);

// float_precision to String integer
string _float_precision_fptoainteger(float_precision *);

// String to float_precision
float_precision _float_precision_atofp(char * int int);

// Double to float_precision
float_precision _float_precision_dtof(double,int,int);
```

Exceptions

The following exceptions can be thrown under the `float_precision` package:

```
bad_int_syntax;     // Thrown if initialized with an illegal number
                    // For example: “123$567” is illegal because
                    // ‘$’ is not a valid character for a numeric.
bad_float_syntax    // Thrown if initialized with an illegal number
                    // For example: “123.567P-3” Here P is not a valid
                    // digit or exponent prefix.
divide_by_zero      // Thrown if dividing by zero
out_of_range        // Thrown for illegal input
domain_error        // Thrown when outside the domain of the function
base_error          // Thrown for base errors       {};
```

Mixed Mode Arithmetic

Mixed mode arithmetic is not supported in the fprecision package. An explicit conversion to a `float_precision` object is required. For example:

```
float_precision a=2;

a=a+2;       // Produces compilation error: ambiguous + operator
a=a+float_precision(2); // Compiles OK
```

Note: Be on the watch for ambiguous compiler operator errors!

Class Internals

A `float_precision` number is stored internally as a vector of unsigned 64-bit integers or in base 2^{64}. The type is typedefs to *fptype* in the header file fprecision.h and can be changed to port it to a different environment. From a performance perspective, it is best to set it to the maximum size of an unsigned integer. Since C++ 2011 this is *uintmax_t* or uint64_t before C++2011.

A `float_precision` value is stored normalized, that is, one binary digit before the fraction sign followed by an arbitrary number of fraction bits. Also, a normalized number is stripped of non-significant zero bits (trailing bits). This makes working and comparing floating point precision numbers easier.

The exponent is stored using a standard C integer variable. This is a short cut and limits the range for an exponent to $2^{+2147483647}$ through $2^{-2147483646}$. This should be more than adequate for most usages.

Member Functions

Several class public member functions are available:

change_sign()	Change the current sign of the float precision object
epsilon()	Return the epsilon for the current precision of the floating precision object, where 1.0+epsilon()==1.0
exponent()	Get or set exponent
index()	Get or set the current index in the binary number. There is no check that the index is valid
inverse()	Return the inverse of the number. E.g. 1/float_object
mode()	Get or set the rounding mode
number()	Get or set the internal mBinary number
pointer()	Return a pointer to the internal mBinary number
precision()	Get or set float precision
pred()	Return the previous representable number towards -∞
sign()	Get or set a sign
square()	Return the square of the float object
succ()	Return the next representable number towards +∞
toExponential()	Convert float_precision to string using Exponential representation. Same as the Javascript counterpart
toFixed()	Convert float_precision to string using Fixed representation. Same as the Javascript counterpart
toFraction()	Truncate the float precision object to its fraction part
toInteger()	Truncate the float precision object to its integer part

toPrecision()	Convert float_precision to string using Precision representation. Same as the Javascript counterpart
toString()	Convert float_precision to a decimal string with an optional negative sign and exponential notation.

There are also a few functions to convert the internal representation of a `float_precision` number to a C++ STL string object.

```
string _float_precision_fptoa(float_precision);
```

The `_float_precision_fptoa()` member function is the only safe way to convert a `float_precision` object without losing precision. For example:

```
float_precision f("1.345E+678");
std::string s;

s=_float_precision_ftoa(f);
cout<<s.c_str()<<endl;
```

The output from the above code fragment would be:

```
+1.345E+678
```

Miscellaneous operators

Standard casting operators are also supported between float_precision and int_precision and all the base types.

```
(char)              // Convert to char. Overflow or rounding may occur
(short)             // Convert to short. Overflow or rounding may occur
(int)               // Convert to int. Overflow or rounding may occur
(long)              // Convert to long. Overflow or rounding may occur
(long long)         // Convert to long. Overflow or rounding may occur
(unsigned char)     // Convert to unsigned char. Overflow may occur
(unsigned short)    // Convert to unsigned short. Overflow may occur
(unsigned int)      // Convert to unsigned int. Overflow may occur
(unsigned long)     // Convert to unsigned long. Overflow may occur
(unsigned long long)  // Convert to unsigned long. Overflow may occur
(float)             // Convert to float. Overflow or rounding may occur
(double)            // Convert to double. Overflow or rounding may occur
(int_precision)     // Convert to int_precision. Overflow may occur
```

However sometimes it creates ambiguity among different compiles, so it is safer to use a method instead or use static cast in C++.

Rounding modes

Each declared `float_precision` number has a rounding mode. The fprecision package supports the four IEEE 754 rounding modes:

IEEE 754 Rounding Mode	Rounding Result
`to nearest`	The rounded result is the closest to the infinitely precise result.
`down (toward -∞)`	The rounded result is the closest to but no greater than the precise result.
`up (toward +∞)`	The rounded result is the closest to but no less than the precise result.
`toward zero (Truncate)`	The rounded result is closest to but no greater in absolute value than the precise result.

The round-up and round-down modes are known as *directed rounding* and can be used to implement interval arithmetic. Interval arithmetic is used to determine upper and lower bounds for the true result of a multi-step computation when the intermediate results of the computation are subject to rounding.

The round *toward zero* mode (sometimes called the "chop" mode) is commonly used when performing integer arithmetic.

The member function that controls the rounding of `float_precision` objects is named `mode`. The `mode` member function has two (overloaded) forms: one to set the round mode of a `float_precision` object, and one to return the current rounding mode. For example:

```
mode=f1.mode();       // Returns rounding mode of f1
f2.mode(ROUND_NEAR);  // Set rounding mode of f2 to nearest
```

Valid mode settings defined in `fprecision.h` is:

```
ROUND_NEAR
ROUND_UP
ROUND_DOWN
ROUND_ZERO
```

Precision

Each declared `float_precision` object has its own precision setting. `float_precision` objects of different precisions can be used within the same statement involving a calculation, however, it is the precision of the L-value that defines the precision for the calculation result.

For example:

```
float_precision f1,f2,f3;

f1.precision(10);
f2.precision(20);
f3.precision(22);

f1=f2+f3; // Addition is done using 22-digit precision and the
```

```
// result is assigned and rounded to 10-digit precision
```

Note: When using a `float_precision` object with any assignment statement (=, +=, -=, *=, /=, <<=, >>=, &=, |=, ^=, etc) the left-hand side precision and rounding mode are never changed. However, there is a circumstance when a `float_precision` object can inherit the precision and rounding properties: when a `float_precision` object is declared.

For example:

```
float_precision f1(1.0, 12, ROUND_UP);
float_precision f2(f1);
float_precision f3=f1;
```

`f1` is assigned an initial value of 1.00000000000, (12-digit precision).
`f2` inherits the precision and rounding mode from `f1`.
`f3` does not inherit the precision and round of `f1`. This is a simple assignment; `f3`'s precision and rounding mode is set to the default values of 20 digits and round nearest.

Precision and rounding modes can be changed at any time using the member method for setting precision and rounding modes. For example:

```
f2.precision(25);     // Change from 12 to 25 significant digits
f2.mode(ROUND_ZERO);  // Change from ROUND_UP to ROUND_ZERO
```

When the precision is changed, the variable is re-normalized.

When performing arithmetic operations the interim result can be of higher precision than the objects involved. For example:

+	Operation is performed using the highest precision of the two operands
-	Operation is performed using the highest precision of the two operands
*	Operation is performed using the highest precision of the two operands
/	Operation is performed using the highest precision of the two operands+1
&	Operation is performed using the highest precision of the two operands
|	Operation is performed using the highest precision of the two operands
^	Operation is performed using the highest precision of the two operands

When the interim result is stored, the result is rounded to the precision of the left-hand side using the rounding mode of the stored variable.

The extra digit of precision for division insures accurate calculation. Assuming we did not add the extra digit of precision an operation like:

```
float_precision c1(1,4), c3(3,4), result(0,4);

result=(c1/c3)*c3;  // Yields 0.999
```

Where the interim division yields: `0.333`

By adding an extra “guard” digit of precision for division the result is more accurate.

```
result=(c1/c3)*c3;  // Yields 1.000
```

The interim result of the division is 0.3333, which when multiplied by 3 gives the interim result of 0.9999 (5-digit precision). Now when rounded to 4-digit precision the result is stored as 1.000!

Internal storage handling

Now since our arbitrary float_precision numbers can be from a few bytes to a mostly unlimited number of bytes we would need an effective and easy way to handle large amounts of data. E.g. when you multiply two 500 digits numbers you get an interim result of 1000 digits number. We have cleverly chosen to store numbers using the STL library String class that automatically expands the String holding the number as needed. That way the storage handling is completely removed from the code since this is automatically handled by the STL String class library. This trick also makes the source code easy to read and comprehend.

Room for Improvement

In the latest version, I have added multi-threading to speed up the calculation of multiplication and the π constant. However, due to the overhead of creating threads, it is first kicked in when numbers exceed 100,000 digits.

API Methods for float_precision

(*float precision object*).change_sign()

Change the current sign of the float precision object

(*float precision object*).epsilon()

Return the epsilon for the current precision of the floating precision object, where 1.0+epsilon()!=1.0

(*float precision object*).exponent(int expo)

Get or set the exponent of the float_precision object to expo. If expo is omitted the current exponent of the float_precision object is returned.

(*float precision object*).index(size_t inx)

Get or set the current index, inx in the vector of the mBinary binary number. There is no check that the index is valid

(*float precision object*).inverse()

Return the inverse of the float_precision object as a new float_precision object.

(*float precision object*).mode(enum round_mode rm)

Get or set rounding mode rm. If rm is omitted the current round mode is returned otherwise the round mode is set to rm and returned.

(*float precision object*).number(vector<fptype> m)

Get or set the internal mBinary number to m and returned it. If m is omitted the current mBinary number is returned.

(*float precision object*).pointer()

Return a pointer to the internal mBinary number.

(*float precision object*).precision(size_t p)

If p is omitted, the current precision is returned, otherwise, the precision is set to p and the value is returned. If a new precision is set, the number will be re-normalized.

(*float precision object*).pred()

Return the previous representable number towards -∞

(*float precision object*).sign(int newsign)

Get or set a new sign. If *newsign* is omitted the current sign is returned otherwise the float precision object is set to *newsign* and the sign is returned.

(*float precision object*).square()

Return the square of the float_precision object as a new float_precision number

(*float precision object*).succ()

Return the next representable number towards +∞

(*float precision object*).toExponential(fix)

Convert float_precision to string using Exponential representation. Same as the JavaScript counterpart

(*float precision object*).toFixed(fix)

Convert float_precision to string using Fixed representation. Same as the JavaScript counterpart

(*float precision object*).toFraction ()

Truncate the float precision object to its fraction part and return the integer as a *float precision object*

(*float precision object*).toInteger()

Truncate the float precision object to its integer part and return the fraction as a *float precision object*.

(*float precision object*).toPrecision()

Convert float_precision to string using Precision representation. Same as the JavaScript counterpart.

(*float precision object*).toString()

Convert float_precision to a decimal string with an optional negative sign and exponential notation.

API Functions for float_precision

float_precision abs(float_precision x)

Return the absolute value of x. Only the sign is changed to +1

float_precision acos(float_precision x)

Return the arccos(x). if x is greater than 1 or less than -1 then it throws the exception:
`float_precision::domain_error`

float_precision acosh(float_precision x)

Return the arccosh(x). if x is less than 1 then it throws the exception:
`float_precision::domain_error`

float_precision asin(float_precision x)

Return the arcsin(x). if x is greater than 1 or less than -1 then it throws the exception:
`float_precision::domain_error`

float_precision asinh(float_precision x)

Return the asinh(x).

float_precision atan(float_precision x)

Return the arctan(x).

float_precision atan2(float_precision y, float_precision x)

Return the arctan($\frac{y}{x}$).

float_precision atanh(float_precision x)

Return the arctanh(x).). if x is greater than or equal to 1 or less than or equal to -1 then it throws the exception:
`float_precision::domain_error`

float_precision AGM(float_precision x, float_precision y)

Return the Arithmetic-Geometric mean of the two numbers as the highest precision of either x or y.

float_precision bernoulli(const size_t bno, const size_t precision)

Return the Bernoulli number, bno with precision.

float_precision bernoulliPolynomials(float_precision x, size_t n)

Return the Bernoulli Polynomials $B(x)_n$ number in the same precision as x.

float_precision beta(float_precision z, float_precision w)

Return the gamma function of beta(z,w).

float_precision ceil(float_precision x)

Return the floor of $\lceil x \rceil$. Returns the smallest integer that is greater than or equal to x.

float_precision cos(float_precision x)

Return the cos(x).

float_precision cosh(float_precision x)

Return the cosh(x).

float_precision erf(float_precision x)

Return the error function erf(x).

float_precision erfc(float_precision x)

Return the complementary error function erfc(x).

float_precision exp(float_precision x)

Return e^x.

float_precision fabs(float_precision x)

Return the absolute value of x. Only the sign is changed to +1. Maintained backward compatibility

float_precision _float_table(enum table_type t, size_t p)

Return the arbitrary precision constant as given by t, at the requested precision of p. See section for build-in constant.

float_precision floor(float_precision x)

Return the floor of $\lfloor x \rfloor$. Return the greatest integer less than or equal to x.

float_precision fma(float_precision a, float_precision b, float_precision c)

Implement the Fused-Multiply-Add function from the C++ standard library. IT computes a*b+c without loss of precision in the interim results. This is useful in e.g. Interval Arithmetic.

float_precision fmod(float_precision x, float_precision y)

Return the remainder of the division x/y. This is the same as the modulo operator % just for the floating point.

float_precision frexp(float_precision x, int *expptr)

The frexp() function breaks down the floating-point value (x) into a mantissa (m) and an exponent (n), such that the absolute value of m is greater than or equal to 1/2 and less than 2, and $x = m*2^n$.
The integer exponent n is stored at the location pointed to by expptr and the mantissa is returned from the function.

float_precision isfinite(float_precision x)

The isfinite() return the Boolean value of whether x is a finite number

float_precision isinf()(float_precision x)

The isinf() return the Boolean value of whether x is positive infinity or negative infinity (FP_INFINITY)

float_precision isnan()(float_precision x)

The isnan() return the Boolean value of whether x is a nan (Not-A-Number)

float_precision isnormal()(float_precision x)

The isnormal() return the Boolean value of whether x is a normal number. (not a nan, infinity, zero or subnormal

float_precision lambertW0(float_precision x)

The lambert function returns the value of $W_0(x)$

float_precision ldexp(float_precision x, int exp)

The ldexp() function returns the value of $x * 2^{exp}$.

float_precision log(float_precision x)

Return the $\log_e(x)$ same as ln(x)

float_precision log2(float_precision x)

Return the $\log_2(x)$.

float_precision log10(float_precision x)

Return the $\log_{10}(x)$.

float_precision modf(float_precision x, float_precision *intpart)

Break the number x into two parts where the integer part is stored in the intpart and the fraction part is returned from the function.

float_precision nextafter(float_precision x, float_precision direction)

Return the next representable number toward the direction indicated. If x==direction then the function return x

float_precision nroot(float_precision x, int y)

Return the $\sqrt[n]{x}$.

float_precision pow(float_precision x, float_precision y)

Return the x^y.

float_precision sin(float_precision x)

Return the sin(x).

float_precision sinh(float_precision x)

Return the sinh(x).

float_precision sqrt(float_precision x)

Return $\sqrt{x}$.

float_precision tan (float_precision x)

Return the tan(x). if x equal to $\frac{\pi}{2}$ or $\frac{3\pi}{2}$ then it throws the exception `float_precision::domain_error`

float_precision tanh(float_precision x)

Return the tanh(x).

float_precision tgamma(float_precision x)

Return the gamma function of x.

float_precision zeta(float_precision x)

Return the zeta function of x.

Arbitrary Complex Precision Template Class

Usage

Due to the way the C++ Standard Library template `complex` class is written, it only supports `float, double` build-in C++ types. The Arbitrary Precision Package "complexprecision.h" header file included in this package is also written as a template class, but it supports `int_precision` and `float_precision` classes, as well as the standard C++ built-in types.

Converting from the C++ Standard Library `complex` class to the `complex_precision`[1] class is accomplished simply by replacing all occurrences of `complex<ObjectName>` with `complex_precision<ObjectName>`.

Besides the traditional C operators like:

```
+, -, /, *, =, ==, !=, +=, -=, *=, /=
```

the following `complex_precision` member functions are available:

Member Function	Description
`real()`	Return real component
`imag()`	Return imaginary component
`norm()`	Returns real*real+imaginary*imaginary. Both as a method and as a function
`abs()`	Return sqrt of norm()
`arg()`	Return radian angle: atan2(real, imaginary)
`conj()`	Conjugation: complex_precision(real,-imaginary). Both as a method and as a function
`exp()`	e raised to a power
`log()`	Base E Logarithm
`log10()`	Base 10 Logarithm
`pow()`	Raise to a power
`sqrt()`	Square root
`sin()`	Sine of a complex number
`cos()`	Cosine of a complex number
`tan()`	Tangent of a complex number
`asin()`	Arc Sine of a complex number
`acos()`	Arc Cosine of a complex number
`atan()`	Arc Tangent of a complex number
`sinh()`	Hyperbolic Sine of a complex number

[1] Actually, it is misleading to call it class since `complex_precision` is a template class and it knows nothing about arbitrary precision. The name `complex_precision` is used to be consistent with the naming convention used with the other Arbitrary Precision Math packages.

cosh()	Hyperbolic Cosine of a complex number
tanh()	Hyperbolic Tangent of a complex number
asinh()	Hyperbolic Arc Sine of a complex number
acosh()	Hyperbolic Arc Cosine of a complex number
atanh()	Hyperbolic Arc Tangent of a complex number

Input/Output (iostream)

The C++ standard ostream << and istream >> operators have been overloaded to support output and input of `complex_precision` objects. For example:

```
cout << cfp1 << endl;

cin >> cfp1 >> cfp2;    // Input two complex_precision number
                          // separated by white space
```

The ostream >> operator always outputs a complex number (object) in the following format:

```
(realpart,imagpart)
```

The istream >> operator provides the ability to read a complex precision number in one of the following standard C++ formats:

```
(realpart,imagpart)
(realpart)
realpart
```

Using float_precision With Complex_precision Class Template

When a `complex_precision` object is created with `float_precision` objects the default rounding mode and precision attributes for `float_precision` objects are used; it is not possible to specify either the rounding or precision attributes of the `float_precision` components in a simple `complex_precision` declaration. However, it is possible to change the rounding mode and precision attributes of a `complex_precision` object `float_precision` components after its assignment by using the two public member functions:

Member Function	Description
ref_real()	Returns a pointer to the real component
ref_imag()	Returns a pointer to the imaginary component

Below is an example showing how to change the precision and rounding mode of a `float_precision` real component:

```
complex_precision<float_precision> cfp;
```

```
float_precision *fp;

fp=cfp.ref_real();
(*fp).precision(30);    // Change precision to 30 digits
(*fp).mode(ROUND_ZERO); // Change rounding mode to
                        // “Round towards Zero”
```

Note: It’s poor programming practice to use different precision and rounding modes for the real part or the imaginary parts of a complex number.

If possible, `complex_precision` objects should be instantiated using a `float_precision` object for initialization. This will cause the `complex_precision` object components to inherit the precision and round mode of the initialization object. For example:

```
complex_precision<float_precision> cfp1;

complex_precision<float_precision> cfp2(cfp1); // Inherits precision and
                                               // rounding mode from cfp1

float_precision fp=cfp.real(); // Does NOT inherit precision & rounding

fp=cfp2.imag();  // Does NOT inherit the precision and round mode
```

Arbitrary Interval Precision Template Class

Change

The intervalprecision class has been updated to align more with the IEEE1788 standard for interval arithmetic.

Old Member Function	Replaced by Member Function
`center()`	mid()
`contain()`	in()
`contains_zero()`	in(0)
`is_class()`	isClass()
`is_empty()`	isEmpty()
`lower()`	inf()
`radius()`	rad()
`upper()`	sup()
`width()`	wid()

Function change

Old Function	Replaced by Function
`Unionsection`	join()

Usage

The `interval_precision`[2] class works with all C++ built-in types and concrete classes like the `complex_precision`.

```
interval_precision<float_precision>itfp;
```

or

```
interval_precision<int_precision> itip;
```

Besides the traditional C operators like:

```
+, -, /, *, =, ==, !=, +=, -=, *=, /=
```

the following `interval_precision` public member functions are available:

Member Function	Description
`in()`	Return true if the interval is contained in another interval

[2] Actually it is misleading to call `interval_precision` a class since it does not known anything about arbitrary precision. The name `interval_precision` is used to be consistent with the naming convention used by the other Arbitrary Precision Math packages.

`inf()`	Return the lower limit of the interval
`Intervaltype()`	Return or set the interval type
`isClass()`	Return classification of the interval. ZERO, POSITIVE, NEGATIVE, MIXED
`isEmpty()`	Return true if the interval is empty. left> right
`isImproper()`	Return true if interval is improper. E.g. left > right
`isPoint()`	Return true if interval is a point interval. e.g. left==right
`isProper()`	Return true if interval is proper. E.g. left<=right
`Leftinterval()`	Return or set the left interval
`mag()`	Return the magnitude of the interval
`mid()`	Return the center of the interval
`mig()`	Return the magnitude of the interval
`rad()`	Return the radius of the interval
`Rightinterval()`	Return or set the right interval
`sup()`	Return the upper limit of the interval
`toString()`	Return a std::string representation of the interval
`wid()`	Return the width of the interval

the following math `interval_precision` functions are available:

Function	Description
`abs()`	Return the absolute value of the interval
`acos()`	Arc Cosine of an interval number
`acosh()`	Hyperbolic Arc Cosine of an interval number
`asin()`	Arc Sine of an interval number
`asinh()`	Hyperbolic Arc Sine of an interval number
`atan()`	Arc Tangent of an interval number
`atanh()`	Hyperbolic Arc Tangent of an interval number
`ceil()`	Return the ceil interval
`cos()`	Cosine of an interval number
`cosh()`	Hyperbolic Cosine of an interval number
`exp()`	e^x raised to a power
`floor()`	Return the floor interval
`inclusion()`	Return one an interval is included in another interval, -1 if not and zero if part of it is included
`interior()`	Return true if interval a is an interior of interval b
`intersection()`	The intersection of two intervals
`intervaldistance()`	Return the interval distance
`join()`	Union of two intervals
`log()`	Base E Logarithm
`log10()`	Base 10 Logarithm
`pow()`	Raise to a power
`precedes()`	Return true if interval a precedes interval b
`sig()`	Return the signum of the interval
`sin()`	Sine of the interval

`sinh()`	Hyperbolic Sine of the interval
`sqr()`	Return the square of the interval
`sqrt()`	Square root of the interval
`subset()`	Return true if interval is a subset of another interval otherwise false
`tan()`	Tangent of the interval
`tanh()`	Hyperbolic Tangent of the interval

Build-in Interval Constants

The following manifest constant is included for any interval type of float, double and float_precision: Since it is now a factory function type is can very easily be incorporated into the template functions.

// infinity
infinitiy_interval<float>();
infinity_interval<double>();
infinity_interval<float_precision>();

// PI
pi_interval<float>();
pi_interval<double>();
pi_interval<float_precision>();

// e
e_interval<float>();
e_interval<double>();
e_interval<float_precision>();

// ln2
ln2_interval<float>();
ln2_interval<double>();
ln2_interval<float_precision>();

// ln10
Ln10_interval<float>();
ln10_interval<double>();
ln10_interval<float_precision>();

Input/Output (iostream)

The C++ standard ostream << and istream >> operators have been overloaded to support output and input of `interval_precision` objects. For example:

```
cout << ifp1 << std::endl;
cin >> ifp1 >> ifp2;  // Input two interval_precision numbers
```

```
// separated by white space
```

The >> istream operator provides the ability to read an `interval_precision` object in the following standard C++ format:

```
[lowerpart, upperpart]
```

The >> ostream operator writes an `interval_precision` object in the following format:

```
[lowerpart, upperpart]
```

Using float_precision With interval_precision Class Template

When an `interval_precision` object is created with `float_precision` objects the default rounding mode and precision attributes for `float_precision` objects are used; it is not possible to specify either the rounding or precision attributes of the `float_precision` components in a simple `interval_precision` declaration. However, it is possible to change the rounding mode and precision attributes of an `interval_precision` object's `float_precision` components after its assignment by using the two public member functions:

Below is an example showing how to change the precision and rounding mode of a `float_precision` component:

```
interval<float_precision> ii;
float_precision tmp;
tmp=ii.rightinterval();
tmp.precision(30); // Change to 30 decimal precision
tmp.mode(ROUND_ZERO); // Change rounding mode to "Round Towards Zero"
ii.rightinterval(tmp);
```

Note. It is poor programming practice to use different precision and rounding modes for the lower and upper parts of an interval number.

If possible, `interval_precision` objects should be instantiated using a `float_precision` object for initialization. This will cause the `interval_precision` object components to inherit the precision and round mode of the initialization object. For example:

```
interval<float_precision> ifp1;
interval<float_precision> ifp2(ifp1); // Inherit the precision and
                                      // rounding mode from cfp;
float_precision fp=ifp.inf(); // fp Does NOT inherit the precision & rounding mode
fp=ifp2.inf()(); // fp Does NOT inherit the precision and round mode
```

However:

```
float_precision fp(ifp.inf()); // fp Does inherit the precision & rounding mode
```

Arbitrary Fraction Precision Template Class

Usage

The `fraction_precision`[4] class works with all C++ built-in types and the concrete class `int_precision`.

```
fraction_precision<int>fint;
```

or

```
fraction_precision<int_precision> fip;
```

Besides the traditional C operators like:

```
+, -, /, *, ++, --, =, ==, !=, +=, -=, *=, /=
```

And the Boolean operators:

```
==, !=, >, <, >=, <=
```

the following `fraction_precision` public member functions are available:

Member Methods	Description
`abs()`	Returns the absolute value of the fraction
`denominator()`	Set or return the denominator of the fraction
`inverse()`	Swap the numerator and the denominator. Any negative sign is maintained in the numerator
`isone()`	Test a fraction for one and return the Boolean value true/false
`iszero()`	Test a fraction for zero and return the Boolean value true/false
`normalize()`	Normalize the fraction to the standard format
`numerator()`	Set or return the numerator of the fraction
`reduce()`	Reduce and Return the whole number of the fraction
`whole()`	Return the whole number of the fraction. E.g. 8/3 is returned as 2

the following math `fraction_precision` member functions are available:

Member Functions	Description
`abs()`	Compute and return the absolute value of the fraction number
`bernoulli()`	Compute and return a Bernoulli number
`gcd()`	Compute and return the greates common divisor of the fraction precision

Input/Output (iostream)

The C++ standard ostream << and istream >> operators have been overloaded to support output and input of `fraction_precision` objects. For example:

```
cout << fp1 << std::endl;
cin >> fp1 >> fp2;  // Input two fraction_precision numbers
                    // separated by white space
```

The >> istream operator input format for a fraction is numerator '/' denominator, where the slash '/' is the delimiter between numerator and denominator.

The >> ostream operator writes an `interval_precision` object in the following format:

Numerator/Denominator

Using int_precision With fraction_precision Class Template

Like all the built-in data types in C++, e.g. from char, short, int, long, int64_t, and the corresponding unsigned version you can also use the int_precision class to extend the fraction to arbitrary precision.

The internal format of the fraction_precision template class is stored in two variables *n* (for the numerator) and *d* for the denominator. Regardless of how it is initialized the fraction is always normalized, meaning there is only one minus sign if any in the fraction, and the minus sign if any is always stored in the numerator.

e.g.

```
fraction_precision<int> fp1(1,1)  // internal n=1, d=1

fraction_precision<int> fp2(-1,1) // internal n=-1,d=1


fraction_precision<int> fp3(1,-1) // internal n=-1,d=1. The sign is
automatically moved to the numerator

fraction_precision<int> fp4(-1,-1) // internal n=1,d=1. The two
negative sign is canceling out
```

If an interim arithmetic calculation results in a negative denominator it is automatically merged with the sign of the numerator as shown above in the process of normalizing the fraction. Furthermore, the fraction is always stored as the minimal representation where the greatest common divisor is automatically divided up in both the numerator and the denominator. This limit the possibility of overflow in a base type like <int>. For `int_precision` it is not strictly necessary but is done to store the fraction in the least possible number of digits.

e.g.

```
fraction_precision<int> fp1(10,5)  // After normalization it is stored
as 2/1

fraction_precision<int> fp1(-1,9)  // After normalization it is stored
as -1/3
```

API Methods for fraction_precision

(*fraction_precision<_Ty> object*).abs()

Return the absolute value of the fraction object.

(*fraction_precision<_Ty> object*).denominator(_Ty dn)

If dn is omitted, the object's current denominator is returned, otherwise, the object's denominator is set to dn and the value is returned. If a new denominator is set, the number will be re-normalized.

(*fraction_precision<_Ty> object*).inverse()

The object numerator and denominator are swapped and the object is re-normalized.

(*fraction_precision<_Ty> object*).isone()

The object is tested for that both the numerator and denominator are one and the Boolean value of the test is returned.

(*fraction_precision<_Ty> object*).iszero()

The object is tested for the numerator is zero and the Boolean value of the test is returned.

(*fraction_precision<_Ty> object*).numerator(_Ty n)

If n is omitted, the object's current numerator is returned, otherwise, the object's numerator is set to n and the value is returned. If a new numerator is set, the number will be re-normalized.

(*fraction_precision<_Ty> object*).normalize()

The object is re-normalized. Normally it will happen automatically without a need to explicitly call this method.

(*fraction_precision<_Ty> object*).reduce()

The object is reduced to a true fraction part (the numerator is less than the denominator) and the integer number is returned as a whole number. E.g. $\frac{9}{4}$ return 2 (as the whole number) and the object is reduced to $\frac{1}{4}$.

(*fraction_precision<_Ty> object*).whole()

The whole number is returned without changing the fraction E.g. $\frac{9}{4}$ return 2 (as the whole number) and the object is still $\frac{9}{4}$.

API Functions for fraction_precision

template<class _Ty> fraction_precision<_Ty> abs(fraction_precision<_Ty>& a)

Return the absolute value of fraction a as a new fraction_precision number.

fraction_precision<int_precision> bernoulli(size_t bno)

Return the Bernoulli number $Bernoulli_{bno}$. The Bernoulli number bno will only be calculated once and then cached for subsequent calls to this function.

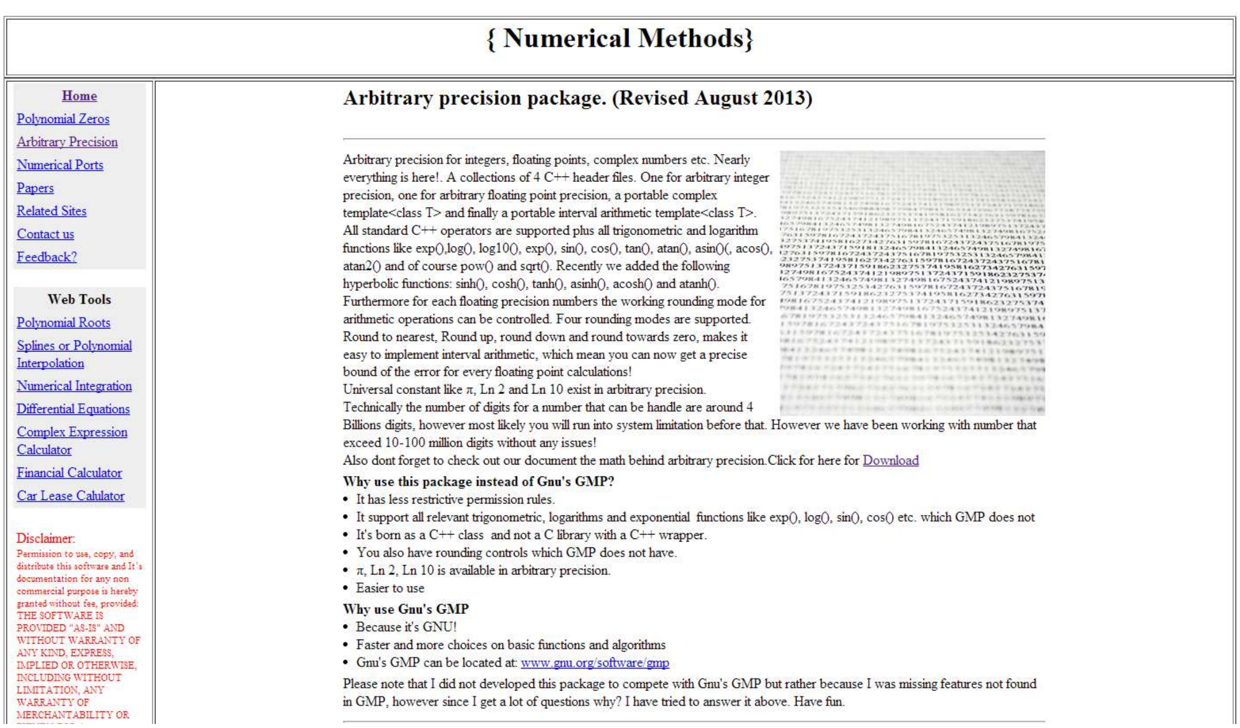
template<class _Ty> fraction_precision<_Ty> gcd(fraction_precision<_TY>& a)

return the greatest common divisor of the fraction precision variable a

Appendix A: Obtaining Arbitrary Precision Math C++ Package

The complete package (Precision.zip) containing the arbitrary precision classes (C++ header files and documentation) for arbitrary integer, floating point, complex and interval math can be downloaded from the following website:

http://www.hvks.com/Numerical/arbitrary_precision.html



{ Numerical Methods}

Home
Polynomial Zeros
Arbitrary Precision
Numerical Ports
Papers
Related Sites
Contact us
Feedback?

Web Tools
Polynomial Roots
Splines or Polynomial Interpolation
Numerical Integration
Differential Equations
Complex Expression Calculator
Financial Calculator
Car Lease Calulator

Disclaimer:
Permission to use, copy, and distribute this software and It's documentation for any non commercial purpose is hereby granted without fee, provided: THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR

Arbitrary precision package. (Revised August 2013)

Arbitrary precision for integers, floating points, complex numbers etc. Nearly everything is here!. A collections of 4 C++ header files. One for arbitrary integer precision, one for arbitrary floating point precision, a portable complex template<class T> and finally a portable interval arithmetic template<class T>.
All standard C++ operators are supported plus all trigonometric and logarithm functions like exp(),log(), log10(), exp(), sin(), cos(), tan(), atan(), asin()(, acos(), atan2() and of course pow() and sqrt(). Recently we added the following hyperbolic functions: sinh(), cosh(), tanh(), asinh(), acosh() and atanh().
Furthermore for each floating precision numbers the working rounding mode for arithmetic operations can be controlled. Four rounding modes are supported. Round to nearest, Round up, round down and round towards zero, makes it easy to implement interval arithmetic, which mean you can now get a precise bound of the error for every floating point calculations!
Universal constant like π, Ln 2 and Ln 10 exist in arbitrary precision.
Technically the number of digits for a number that can be handle are around 4 Billions digits, however most likely you will run into system limitation before that. However we have been working with number that exceed 10-100 million digits without any issues!
Also dont forget to check out our document the math behind arbitrary precision.Click for here for Download

Why use this package instead of Gnu's GMP?

- It has less restrictive permission rules.
- It support all relevant trigonometric, logarithms and exponential functions like exp(), log(), sin(), cos() etc. which GMP does not
- It's born as a C++ class and not a C library with a C++ wrapper.
- You also have rounding controls which GMP does not have.
- π, Ln 2, Ln 10 is available in arbitrary precision.
- Easier to use

Why use Gnu's GMP

- Because it's GNU!
- Faster and more choices on basic functions and algorithms
- Gnu's GMP can be located at: www.gnu.org/software/gmp

Please note that I did not developed this package to compete with Gnu's GMP but rather because I was missing features not found in GMP, however since I get a lot of questions why? I have tried to answer it above. Have fun.

Appendix B: Sample Programs

Solving an N Degree Polynomial

The following sample C++ code demonstrates the use of the `float_precision` class and `complex_precision` class template to find every (real and imaginary) solution of an N-degree polynomial equation using Newton's (Madsen) method. The Polynomial coefficients are stored in a vector<cmplx> a. Where the first entry a[0] is equal to a_n and the highest index a[n] is a_0. Furthermore, there is an optional parameter called precision (default is 20-digit accuracy). Where you can specify the accuracy of the calculation in the number of decimal digits that the root should be found in. This means all local variables will default to that precision. There are a few interim calculations that don't need an accuracy higher than 10-15 decimal digits. This is explicitly declared with a lower decimal accuracy.

```
/*
 *******************************************************************************
 *
 *                        Copyright (c) 2023
 *                        Henrik Vestermark
 *                        Denmark, USA
 *
 *                        All Rights Reserved
 *
 *   This source file is subject to the terms and conditions of
 *   Henrik Vestermark Software License Agreement which restricts the manner
 *   in which it may be used.
 *
 *******************************************************************************
*/

/*
 *******************************************************************************
 *
 * Module name     :   PrecisionNewton.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using Precision Newton's
(Madsen) method
 * --------------------------------------------------------------------------
 * Change Record   :
 *
 * Version    Author/Date         Description of changes
 * -------  -------------   -----------------------
 * 01.01      HVE/24Sep2023 Initial release
 *
 * End of Change Record
 * --------------------------------------------------------------------------
*/

// define version string
static char _VNEWTON_[] = "@(#)PrecisionNewton.cpp 01.01 -- Copyright (C) Henrik
Vestermark";

#include <algorithm>
#include <vector>
```

```
#include <complex>
#include <iostream>
#include <functional>
// #include <complex>

#include "../fprecision.h"
#include "../complexprecision.h"

using namespace std;

using fp = float_precision;
using cmplx = complex_precision<fp>;

constexpr int       MAX_ITER = 50;

// Find all polynomial zeros using a modified Newton method
// 1) Eliminate all simple roots (roots equal to zero)
// 2) Find a suitable starting point
// 3) Find a root using Newton
// 4) Divide the root up in the polynomial reducing its order with one
// 5) Repeat steps 2 to 4 until the polynomial is of the order of two whereafter
the remaining one or two roots are found by the direct formula
// Notice:
//      The coefficients for p(x) is stored in descending order. coefficients[0]
is a(n)x^n,
//      coefficients[1] is a(n-1)x^(n-1),...,  coefficients[n-1] is a(1)x,
coefficients[n] is a(0)
//
static vector<cmplx> PolynomialRoots(const vector<cmplx>& coefficients, const
size_t precision = 20)
{
    const size_t prev_precision = float_precision_ctrl.precision();    // Get
current default precision
    float_precision_ctrl.precision(precision);  // Set global precision for all
declared float_precision

    struct eval { cmplx z; cmplx pz; fp apz; };
    const cmplx complexzero(0.0);  // Complex zero (0+i0)
    const intmax_t bitsprec(intmax_t(precision* log(10) / log(2)));// this is the
t in 0.5*2^(1-t)
    const fp BMT(pow(fp(2,15), fp(-bitsprec,15)));        // 0.5*2^(1-t)=2^(-t)

    size_t n;       // Size of Polynomial p(x)
    eval pz;        // P(z)
    eval pzprev;    // P(zprev)
    eval p1z;       // P'(z)
    eval p1zprev;   // P'(zprev)
    cmplx z;      // Use as temporary variable
    cmplx dz;     // The current stepsize dz
    int itercnt;    // Holds the number of iterations per root
    vector<cmplx> roots;  // Holds the roots of the Polynomial
    vector<cmplx> coeff(coefficients.size()); // Holds the current coefficients
of P(z)

    // Copy original coefficients to the working coefficients coeff.
    copy(coefficients.begin(), coefficients.end(), coeff.begin());
    // Step 1 eliminate all simple roots
    for (n = coeff.size() - 1; n > 0 && coeff.back() == complexzero; --n)
        roots.push_back(complexzero);  // Store zero as the root

    // Compute the next starting point based on the polynomial coefficients
```

```
    // A root will always be outside the circle from the origin and radius min
    auto startpoint = [&](const vector<cmplx>& a)
    {
        const size_t n = a.size() - 1;

        fp a0 = log(abs(a.back()));
        fp min = exp((a0 - log(abs(a.front()))) / fp(n));
        for (size_t i = 1; i < n; i++)
            if (a[i] != complexzero)
            {
                fp tmp = exp((a0 - log(abs(a[i]))) / fp(n - i));
                if (tmp < min)
                    min = tmp;
            }

        return min * fp(0.5);
    };

    // Evaluate a polynomial with complex coefficients a[] at a complex point z
and
    // return the result
    // This is Horner's methods
    auto horner = [](const vector<cmplx>& a, const cmplx z)
    {
        const size_t n = a.size() - 1;
        eval e;

        cmplx fval = a.front();
        for (size_t i = 1; i <= n; i++)
            fval = fval * z + a[i];

        e.z = z;
        e.pz = fval;
        e.apz = abs(fval);
        return e;
    };

    // Calculate an upper bound for the rounding errors performed in a
    // polynomial with complex coefficient a[] at a complex point z.
    // (Grant & Hitchins test)
    auto upperbound = [&](const vector<cmplx>& a, cmplx z)
    {
        const size_t n = a.size() - 1;
        const fp c1(1),c2(2);
        fp nc, oc, nd, od, ng, og, nh, oh, t, u, v, w, e;
        fp tol = BMT;

        oc = a[0].real();
        od = a[0].imag();
        og = oh = 1.0;
        t = fabs(z.real());
        u = fabs(z.imag());
        for (size_t i = 1; i <= n; i++)
        {
            nc = z.real() * oc - z.imag() * od + a[i].real();
            nd = z.imag() * oc + z.real() * od + a[i].imag();
            v = og + fabs(oc);
            w = oh + fabs(od);
            ng = t * v + u * w + fabs(a[i].real()) + c2 * fabs(nc);
            nh = u * v + t * w + fabs(a[i].imag()) + c2 * fabs(nd);
            og = ng;
```

```
            oh = nh;
            oc = nc;
            od = nd;
        }
        e = abs(cmplx(ng, nh)) * pow(c1 + tol, fp(5 * n)) * tol;
        return e;
    };

    // Do Newton iteration for polynomial order higher than 2
    for (; n > 2; --n)
    {
        const fp Max_stepsize(5.0,15); // Allow the next step size to be up to 5
times larger than the previous step size
        const cmplx rotation = cmplx(0.6, 0.8);  // Rotation amount
        fp r(0,15);          // Current radius. Just around double accuracy is
needed
        fp rprev(0,15);      // Previous radius. Just around double accuracy is
needed
        fp eps(0,15);        // The iteration termination value. Just around
double accuracy is needed
        bool stage1 = true; // By default it start the iteration in stage1
        int steps = 1;      // Multisteps if > 1
        vector<cmplx> coeffprime;

        // Calculate coefficients of p'(x)
        for (int i = 0; i < n; i++)
            coeffprime.push_back(coeff[i] * cmplx(fp(n - i)));

        // Step 2 find a suitable starting point z
        rprev = startpoint(coeff);      // Computed startpoint
        z = coeff[n - 1] == complexzero ? cmplx(1.0) : -coeff[n] / coeff[n - 1];
        z *= cmplx(rprev / abs(z));

        // Setup the iteration
        // Current P(z)
        pz = horner(coeff, z);

        // pzprev which is the previous z or P(0)
        pzprev.z = cmplx(0);
        pzprev.pz = coeff[n];
        pzprev.apz = abs(pzprev.pz);

        // p1zprev P'(0) is the P'(0)
        p1zprev.z = pzprev.z;
        p1zprev.pz = coeff[n - 1];       // P'(0)
        p1zprev.apz = abs(p1zprev.pz);

        // Set previous dz and calculate the radius of operations.
        dz = pz.z;      // dz=z-zprev=z since zprev==0
        r = rprev *= Max_stepsize; // Make a reasonable radius of the maximum
step size allowed
        // Preliminary eps computed at point P(0) by a crude estimation
        eps = fp(6 * n) * pzprev.apz * BMT;

        // Start iteration and stop if z doesnt change or apz <= eps
        // we do z+dz!=z instead of dz!=0. if dz does not change z then we accept
z as a root
        for (itercnt = 0; pz.z + dz != pz.z && pz.apz > eps && itercnt <
MAX_ITER; itercnt++)
        {
            // Calculate current P'(z)
```

```
            p1z = horner(coeffprime, pz.z);
            if (p1z.apz.iszero())                 // P'(z)==0 then rotate and try
again
                dz *= rotation * cmplx(Max_stepsize);  // Multiply with 5 to get
away from saddlepoint
            else
            {
                dz = pz.pz / p1z.pz;  // next dz
                // Check the Magnitude of Newton's step
                r = abs(dz);
                if (r > rprev) // Large than 5 times the previous step size
                {   // then rotate and adjust step size to prevent wild step size
near P'(z) close to zero
                    dz *= rotation * cmplx((rprev/r));
                    r = abs(dz);
                }
                rprev = r * Max_stepsize;  // Save 5 times the current step size
for the next iteration check of reasonable step size
                // Calculate if stage1 is true or false. Stage1 is false if the
Newton converge otherwise true
                z = (p1zprev.pz - p1z.pz) / (pzprev.z - pz.z);
                stage1 = (abs(z) / p1z.apz > p1z.apz / pz.apz / fp(4)) || (steps
!= 1);
            }
            // Step accepted. Save pz in pzprev
            pzprev = pz;

            cout << "\tStep size=" << dz << endl;
            z = pzprev.z - dz;       // Next z
            pz = horner(coeff, z); //ff = pz.apz;
            steps = 1;
            if (stage1)
            {  // Try multiple steps or shorten steps depending if P(z) is an
improvement or not P(z)<P(zprev)
                bool div2;
                cmplx zn;
                eval npz;

                zn = pz.z;
                for (div2 = pz.apz > pzprev.apz; steps <= n; ++steps)
                {
                    if (div2 == true)
                    {  // Shorten steps
                        dz *= cmplx(0.5);
                        zn = pzprev.z - dz;
                    }
                    else
                        zn -= dz;  // try another step in the same direction

                    // Evaluate new try step
                    npz = horner(coeff, zn);
                    if (npz.apz >= pz.apz)
                        break; // Break if no improvement

                    // Improved => accept step and try another round of step
                    pz = npz;

                    if (div2 == true && steps == 2)
                    {   // To many shorten steps => try another direction and
break
                        dz *= rotation;
```

```
                        z = pzprev.z - dz;
                        pz = horner(coeff, z);
                        break;
                    }
                }
            }
            else
            {   // calculate the upper bound of error using Grant & Hitchins's
test for complex coefficients
                // Now that we are within the convergence circle.
                eps = upperbound(coeff, pz.z);
            }
        }

        // Check if there is a very small residue in the imaginary part by trying
        // to evaluate P(z.real). if that is less than P(z). We take that
z.real() is a better root than z.
        z = cmplx(pz.z.real(), 0.0);
        pzprev = horner(coeff, z);
        if (pzprev.apz <= pz.apz)
            pz = pzprev;

        // Save the root
        cout << "Final: z=" << pz.z << endl;
        roots.push_back(pz.z);

        // Deflate polynomial and compute new coefficients in coeff
        z = fp(0);
        for (int j = 0; j < n; j++)
            z = coeff[j] = z * pz.z + coeff[j];
        coeff.resize(n);
    }   // End Iteration

    // Solve any remaining linear or quadratic polynomial
    // For Polynomial with complex coefficients a[],
    // The complex solutions are stored in the back of the roots
    auto quadratic = [&](const std::vector<cmplx>& a)
    {
        const size_t n = a.size() - 1;
        const fp c0(0);
        cmplx v;

        // Notice a[0]!=0, since all zero roots has been eliminated
        if (n == 1)
            roots.push_back(-a[1] / a[0]);
        else
        {
            if (a[1] == complexzero)
            {
                v = sqrt(-a[2] / a[0]);
                roots.push_back(v);
                roots.push_back(-v);
            }
            else
            {
                v = sqrt(cmplx(1.0) - cmplx(4.0) * a[0] * a[2] / (a[1] * a[1]));
                if (v.real() < c0)
                    v = (cmplx(-1.0) - v) * a[1] / (cmplx(2.0) * a[0]);
                else
                    v = (cmplx(-1.0) + v) * a[1] / (cmplx(2.0) * a[0]);
                roots.push_back(v);
```

```
                roots.push_back(a[2] / (a[0] * v));
            }
        }
        return;
    };

    if (n > 0)
        quadratic(coeff);

    float_precision_ctrl.precision(prev_precision);  // Reset global precision
back to normal
    return roots;
}
```

Appendix C: int_precision Example

This example illustrates the use and mix of int_precision with standard types like int. It calculates the digits of π and returned it as a std::string.

```
std::string unbounded_pi(const int digits)
       {
       const int_precision c1(1), c4(4), c7(7), c10(10), c3(3), c2(2);
       int_precision q(1), r(0), t(1);
       unsigned k = 1, l = 3, n = 3, nn;
       int_precision nr;
       bool first = true;
       int i,j;
       std::string ss = "";

       for(i=0,j=0;i<digits;++j)
              {
              if ((c4*q + r - t) < n*t)
                     {
                     ss += (n + '0');
                     i++;
                     if (first == true)
                            {
                            ss += ".";
                            first = false;
                            }
                     nr = c10*(r - (n*t));
                     n = (int)((c3*q + r) / t) - n;
                     q *= c10;
                     r = nr;
                     }
              else {
                     nr = (c2*q + r)*int_precision(l);
                     nn = (q*(int_precision)(7*k) + c2 + r*l) / (t*l);
                     q *= k;
                     t *= l;
                     l += 2;
                     k += 1;
                     n = nn;
                     r = nr;
                     }
              }
       return ss;
       }
```

Appendix D: Fraction Example

Lambert's expression for π is dated back to 1770.
Lambert found the continued fraction below that yields 2 significant digits of π for every 3 terms.

```
void continued_fraction_pi_lambert()
	{
	int i,j;
	fraction_precision<int_precision> cf;
	cout << "Start of Lambert PI. (First 8 iterations)" << endl;
	for(j=1;j<=8;++j)
		{
		for (i = j; i >=0; --i)
			{
			cf += fraction_precision<int_precision>(i * 2 + 1, 1);
			if (i > 0)
				cf = fraction_precision<int_precision>(i*i, 1) / cf;
			else
				cf = fraction_precision<int_precision>(4, 1)/cf;
			}

		cout << j << ": " << cf << " = " << (double)cf << " Error: " <<
(double)cf - M_PI << endl;
		}

	cout << "end of Lambert PI" << endl;

	return;
		}
```

When running it will produce the following output:

```
C:\Users\henrik vestermark\Documents\HVE\CI\Precision3\Debug\Precision3.exe
Start of Lambert PI. (First 8 iterations)
1: +3/+1 = 3 Error: -0.141593
2: +28/+9 = 3.11111 Error: -0.0304815
3: +1972/+627 = 3.14514 Error: 0.00354291
4: +1409008/+448557 = 3.1412 Error: -0.000390978
5: +642832772/+204617505 = 3.14163 Error: 3.87137e-05
6: +620973746437/+197662271090 = 3.14159 Error: -2.99658e-06
7: +21256237030334666/+6766070335136595 = 3.14159 Error: 2.53911e-08
8: +29359991221904052211456/+9345575277160084385045 = 3.14159 Error: 6.28755e-08
end of Lambert PI
```

Appendix E: Compiler info

This package has been developed and tested under the Microsoft Visual Studio version 2022 in the 64-bit environment. Although not 100% sure I believe C++ 2014 standard or higher is sufficient to compile the source.
Furthermore, it has been tested with the GNU compiler in a 64-bit environment with Code::Blocks 20.03. In the latest version, all of the GNU warning messages have been fixed so it should compile cleanly in this environment too.
Additionally, Thanks to Robert McInnes who successfully ported these packages to the Xcode C++ environment on a Mac.

In a 32-bit environment, the max precision is 2^{32}-1, or the number of arbitrary digits it can handle, however, most likely you will run into an Operative system dependent constraint long before the theoretical limit. In a 64-bit environment, the max precision would be 2^{64}-1