Arbitrary Precision Math C++ Package

By: Henrik Vestermark
(hve@hvks.com)

Revised: 2019/17 September

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 expanding the explanation and usage of these classes.
2014/Jun/21	Cleaning up the documentation and add 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 return the sign instead of embedding it into the string

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
`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 employed.

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` objects, or a complex object using `interval_precision` objects. Normally, a `complex_precision` and `interval_precision` objects are built using `float_precision` objects.

Compiling the source code

The source consists of four 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 statement 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.

The source has been tested and compiled under Microsoft Visual C++ 2015 express compiler.

Arbitrary Precision Math C++ Package

Arbitrary Integer Precision Class

Usage

In order to use the integer precision class the following include statement must 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 a many different ways. The following examples show the supported forms for initialization:

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

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

```
int_precision i1 = 1;         // Decimal
int_precision i2 = ‘1’;       // Char
int_precision i3 = “123”;     // String
int_precision i4 = 0377;      // Octal
int_precision i5 = 0x9Af;     // Hexadecimal
int_precision i6 = i1;        // Another int_precision object
```

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;
```

and

```
i1*=i2;
i1-=i2;
i1+=i2;
i1/=i2;
i1%=i2;
i1<<=i2;
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, float` or `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 (methods) are 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            iprime( 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)
```

Input/Output (iostream)

The C++ standard ostream << operator has been overloaded to support 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();
```

or the reverse converting string to int_precision via _int_precision_atoi(char *, *sign) e.g.

```
int sign;
i1=_int_precision_atoi( s.c_str(), &sign );
```

The C++ standard istream >> operator has also been overloaded to support 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.
out_of_range      // Thrown when attempting to shift with a negative
                  // value using the << or >> operator.
divide_by_zero    // Thrown if dividing by zero.
```

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 can store numbers using either RADIX 2, 8, 10, 16 or RADIX 256 (or BASE 256). This allows for a more efficient use of memory and speeds up calculations dramatically. A number stored using BASE 256 uses 2.4 less RADIX digits than compared to the equivalent stored in BASE 10. For example: a number that can be represented with 10 BASE 256 digits requires 24 BASE 10 digits of storage.

Since the arithmetic operations requires between N to N^2 operations, where N is the number of digits, using BASE 256 speeds up the operations by a factor of 2.4 to 5.7. Although the package is coded to use BASE 256 it can be easily be changed to use BASE 10 radix. (BASE 10 radix is used primary for debugging.) In order to switch to a different internal BASE number, change the const int RADIX statement in `iprecision.h`

From: `const int RADIX=BASE_256;`
To: `const int RADIX=BASE_10;`

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 less memory resources.

Member Functions

Beside the `_int_precision_itoa()` method already discussed, the following member functions are also accessible:

```
copy()          // Return a copy of the number as a class string
pointer()       // Return a pointer to the number as a class (string *)
sign()          // Return sign of number (+1 or -1)
change_sign()   // Change sign
size()          // Return the number of digits including the sign
even()          // Return true if number is even otherwise false
odd()           // Return true if number is odd otherwise false
toString()      // Convert int_precision to string
```

Internal storage handling

Now since our arbitrary int_precision numbers can be from two bytes (sign and one digit) to mostly unlimited number of bytes we would need an effective and easy way to handle large amount of data. E.g. when you multiply two 500 digits number you get a 1000 digits number as result. We have cleverly chosen to store number 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 handle by the STL string class library. This trick also makes the source code easy to read and comprehend.

Room for Improvement

Absolutely. A number of performances enhancing tricks is implemented and will be improved in future versions. For example, use of Fast Fourier Transform (FFT) math for multiplication, and increasing reliance on the build function for integer arithmetic. When adding numbers (particularly when the internal representation is stored in BASE_256) the numbers can be converted to built-in `int`'s and the `int +` operator used to add four RADIX 256 digits at one time, and then convert them back to the BASE 256 number.

Arbitrary Floating Point Precision

Usage

In order 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 a ordinary C++ built-in `int`, `float`, `double`, `char`, `string` data type, or even another `float_precision`. For example:

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

float_precision f1 = -1;            // Decimal
float_precision f2 = ‘1’;           // Char
```

```
float_precision f3 = "123.456E+789";// String
float_precision f4 = 0377;          // Octal
float_precision f5 = 0x9Af;         // Hexadecimal
float_precision f5 = -123.456E78;   // Float
float_precision f6 = f1;            // Another float_precision
```

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 digits, and a rounding mode of *nearest* (the default behavior according to IEEE 754 floating point standard) is used.

For example, to initialize two `float_precision` 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 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 a 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. 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 (methods) are accessible for `float_precision` objects:

```
float_precision   log( 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 );

// 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 );
```

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

These are not true C++ constants, but are variables that can be created with varying degrees of precision. In order to use one of these constants, a call must be made to the member function `_float_table()` to calculate (initialize) the constant to the requested precision.

The `_float_table()` member function remembers the most precise constant's precision calculation and if a 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 preformed and stored.

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_ftoa(float_precision *);

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

// String to float_precision
float_precision _float_precision_atof(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
```

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 using the decimal BASE 10 RADIX or BASE 256. The const FRADIX control whether you are working in BASE_10 or BASE_256. A number stored in BASE_256 require 2.4 less digits compared to a number stored in BASE 10. However the drawbacks for internally working in BASE 256 are that conversion to and from BASE 256 is pretty time consuming.

A `float_precision` value is stored normalized, that is, one decimal digit before the fraction sign followed by an arbitrary number of fraction digits. Also, a normalized number is stripped of non-significant zero digits. 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 $10^{+2147483647}$ through $10^{-2147483646}$. This should be more than adequate under most usages.

Member Functions

Several class public member functions are available:

```
get_mantissa()    // Return a copy of the mantissa as a class string
ref_mantissa()    // Return a pointer to the mantissa as a class
                  //(string *) object.
mode()            // Return rounding mode
mode(RoundingMode)// Set and return rounding mode
exponent()        // Return the exponent as a base of RADIX
exponent(exp)     // Set and return the exponent as a base of RADIX
sign()            // Return the sign of the float_precision variable
sign(sg)          // Set the sign of the float_precision variable
precision()       // Return the current precision of the number. Number
                  // of digits
precision(prec)   // Set and return precision. The number is rounding
                  // to precision based on rounding mode.
change_sign()     // Change sign of the float_precision variable
epsilon()         // Return the epsilon where 1.0+epsilon!=1.0
toString()        // Convert float_precision to string
to_int_precision()// Convert a float_precision to int_precision
toFixed()         // Convert float_precision to string using Fixed
                  representation. Same as Javascript counterpart
toPrecision()     // Convert float_precision to string using Precision
                  representation. Same as Javascript counterpart
toExponential()   // Convert float_precision to string using
                  Exponential representation. Same as Javascript
                  counterpart
```

There is also a member function to convert the internal representation of a `float_precision` number to a C++ string object.

```
string _float_precision_ftoa(float_precision);
```

The `_float_precision_ftoa()` 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
(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
(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 an ambiguity among different compiles, so it is safer to use a method instead.

Rounding modes

To 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	Rounded result is the closest to the infinitely precise result.
down (toward -·)	Rounded result is close to but no greater than the infinitely precise result.
up (toward +·)	Rounded result is close to but no less than the infinitely precise result.

toward zero (Truncate)	Rounded result is close to but no greater in absolute value than the infinitely 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 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` are:

```
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 inherent 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 inherent the precision and round of `f1`. This is a simple assignment; `f3`'s precision and rounding mode are set to the default values of 20 digits and round nearest.

Precision and rounding mode can be changed at any time using the member function 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 performing arithmetic operations the interim result can be of a 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

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 digits 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 mostly unlimited number of bytes we would need an effective and easy way to handle large amount of data. E.g. when you multiply two 500 digits number you get an interim result of 1000 digits number. We have cleverly chosen to store number 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 handle by the STL String class library. This trick also makes the source code easy to read and comprehend.

Room for Improvement

Absolutely and it will continue. Example lately we added a more optimized handling of elementary functions more aggressively using argument reduction. See the Math behind Arbitrary precision.

Arbitrary Complex Precision Template Class

Usage

Due to the way the C++ Standard Library template complex class is written, it only supports float, double or long 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
abs()	Returnsqrt of norm()
arg()	Return radian angle: atan2(real, imaginary)
conj()	Conjugation: complex_precision(real,-imaginary)
exp()	e raised to a power
log()	Base E Logarithm
log10()	Base 10 Logarithm
pow()	Raise to a power
sqrt()	Square root

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;
```

[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.

```
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 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

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
`upper()`	Return the upper limit of interval
`lower()`	Return the lower limit of interval
`center()`	Return the center of interval
`radius()`	Return the radius of interval
`width()`	Return the width of interval
`contains_zero()`	Return true if 0 is within the interval
`is_class()`	Return classification of the interval. ZERO, POSITIVE, NEGATIVE, MIXED

the following math `interval_precision` member functions are available:

Member Function	Description
`exp()`	e raised to a power
`log()`	Base E Logarithm
`log10()`	Base 10 Logarithm
`pow()`	Raise to a power
`sqrt()`	Square root

Input/Output (iostream)

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

---

[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.

```
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:

Member Function	Description
`ref_lowerl()`	Returns a pointer to the lower limit component
`ref_upperl()`	Returns a pointer to the upper limit component

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

```
interval<float_precision> ii;
float_precision *fp;

fp=ii.ref_upper();
(*fp).precision(30);       // Changes precision to 30 digits
(*fp).mode(ROUND_ZERO);    // Change rounding mode to
                           // "Round Towards Zero"
```

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 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.upper(); // Does NOT inherit the precision &
rounding mode

fp=ifp2.lower();  // Does NOT inherit the precision and round mode
```

Arbitrary Fraction Precision Template Class

Usage

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

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

or

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

Besides the traditional C operators like:

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

the following `fraction_precision` public member functions are available:

Member Function	Description
`numerator()`	Set or return the numerator of the fraction
`denominator()`	Set or return the denominator of the fraction
`whole()`	Return the whole number of the fraction. E.g. 8/3 is return as 2
`reduce()`	Reduce and Return the whole number of the fraction
`normalize()`	Normalize the fraction to standard format
`abs()`	Returns the absolute value of the fraction
`inverse()`	Swap the numerator and the denominator. Any negative sign is maintained in the numerator

the following math `fraction_precision` member functions are available:

Member Function	Description
`gcd()`	Greatest common divisor of 2 numbers
`lcm()`	Least Common multiplier of two numbers

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 build 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 extended the fraction to arbitrary precision.

Internal format of the fraction_precision template class is stored in two variable *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 cancelling out

If an interim arithmetic calculation result 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 possible of overflow in a base type like <int>. For int_precision it is not strictly necessary but done to stored the fraction in the least possible number of digits.

e.g.

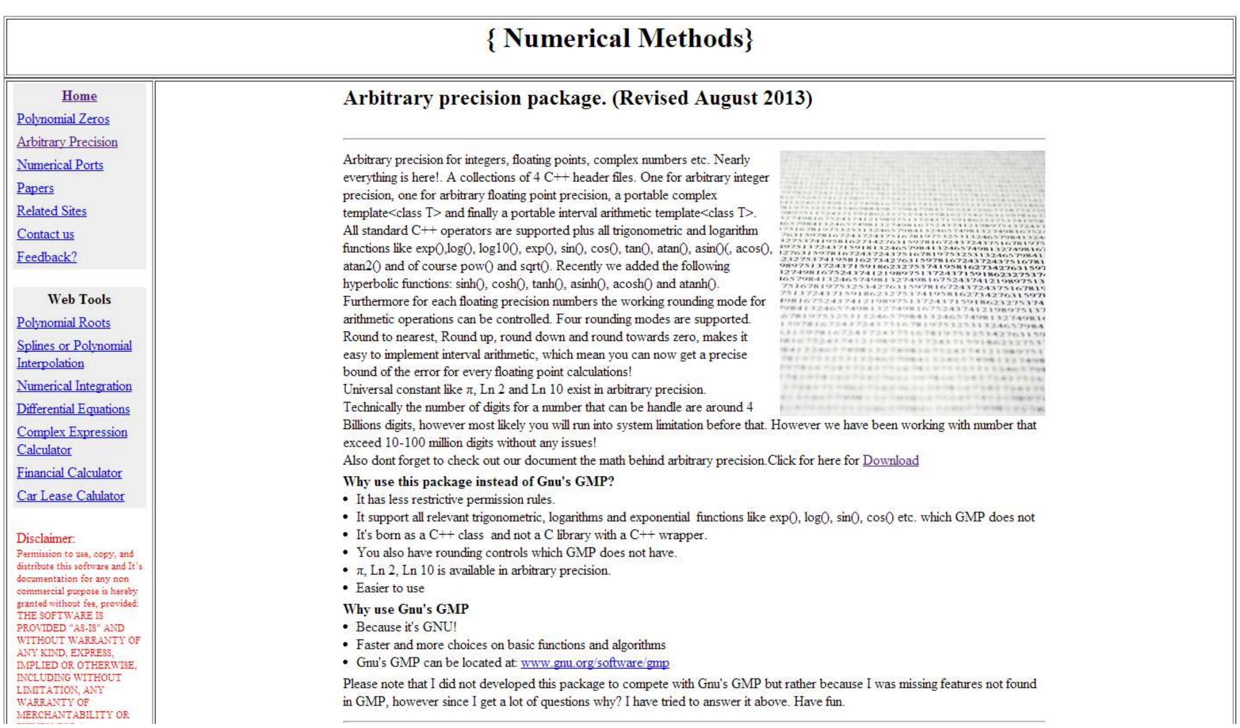
fraction_precision<int> fp1(10,5) // After normalization it is stored as 2/1

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

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 down loaded from the following web site:

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.

```
/*
 *******************************************************************************
 *
 *
 *                       Copyright (c) 2002
 *                       Future Team Aps
 *                       Denmark
 *
 *                       All Rights Reserved
 *
 *   This source file is subject to the terms and conditions of the
 *   Future Team Software License Agreement that restricts the manner
 *   in which it may be used.
 *
 *
 *******************************************************************************
*/

/*
 *******************************************************************************
 *
 *
 * Module name     :   Newcprecision.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using Newton's (Madsen) method
 * --------------------------------------------------------------------------
 * Change Record   :
 *
 * Version  Author/Date            Description of changes
 * -------  -----------            ----------------------
 * 01.01    HVE/030331             Initial release
 *
 * End of Change Record
 * --------------------------------------------------------------------------
*/

/* define version string */
static char _VNEWR_[] = "@(#)newc.cpp 01.01 -- Copyright (C) Future Team Aps";

#include "stdafx.h"
#include <malloc.h>
#include <time.h>
#include <float.h>
#include <iostream.h>
#include <math.h>

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

#define fp float_precision
#define cmplx complex_precision

using namespace std;
#define MAXITER   50

static float_precision feval(const register int n,const cmplx<fp> a[],const cmplx<fp> z,cmplx<fp> *fz)
```

```
   {
   cmplx<fp> fval;

   fval = a[ 0 ];
   for( register int i = 1; i <= n; i++ )
      fval = fval * z + a[ i ];

   *fz = fval;
   return fval.real() * fval.real() + fval.imag() * fval.imag();
   }

static float_precision startpoint( const register int n, const cmplx<fp> a[] )
   {
   float_precision r, min, u;

   r = log( abs( a[ n ] ) );
   min = exp( ( r - log( abs( a[ 0 ] ) ) ) / float_precision( n ) );
   for( register int i = 1; i < n; i++ )
      if( a[ i ] != cmplx<fp>( float_precision( 0 ), float_precision( 0 ) ) )
         {
         u = exp( ( r - log( abs( a[ i ] ) ) ) / float_precision( n - i ) );
         if( u < min )
            min  = u;
         }

   return min;
   }


static void quadratic( const register int n, const cmplx<fp> a[], cmplx<double> res[])
   {
   cmplx<fp> v;

   if( n == 1 )
      {
      v = - a[ 1 ] / a[ 0 ];
      res[ 1 ] = cmplx<double>( (double)v.real(), (double)v.imag() );
      }
   else
      {
      if( a[ 1 ] == cmplx<fp>( 0 ) )
         {
         v = - a[ 2 ] / a[ 0 ];
         v = sqrt( v );
         res[ 1 ] = cmplx<double>( (double)v.real(), (double)v.imag() );
         res[ 2 ] = -res[ 1 ];
         }
      else
         {
         v = sqrt( cmplx<fp>( 1 )  - cmplx<fp>( 4 ) * a[ 0 ] * a[ 2 ] / ( a[ 1 ] * a[ 1 ] ) );
         if( v.real() < float_precision( 0 ) )
            {
            v = ( cmplx<fp>( -1, 0 ) - v ) * a[ 1 ] / ( cmplx<fp>( 2 ) * a[ 0 ] );
            res[ 1 ] = cmplx<double>( (double)v.real(), (double)v.imag() );
            }
         else
            {
            v = ( cmplx<fp>( -1, 0 ) + v ) * a[ 1 ] / ( cmplx<fp>( 2 ) * a[ 0 ] );
            res[ 1 ] = cmplx<double>( (double)v.real(), (double)v.imag() );
            }
         v = a[ 2 ] / ( a[ 0 ] * cmplx<fp> ( res[ 1 ].real(), res[ 1 ].imag() ) );
         res[ 2 ] = cmplx<double>( (double)v.real(), (double)v.imag() );
         }
      }
   }

// Find all root of a polynomial of n degree with complex coefficient using the
// modified Newton
//
int complex_newton( register int n, cmplx<double> coeff[], cmplx<double> res[] )
   {
```

```
   int itercnt, stage1, err, i;
   float_precision r, r0, u, f, f0, eps, f1, ff;
   cmplx<fp> z0, f0z, z, dz, f1z, fz;
   cmplx<fp> *a1, *a;

   err = 0;

   a = new cmplx<fp> [ n + 1 ];
    for( i = 0; i <= n; i++ )
           a[ i ] = cmplx<fp> ( coeff[ i ]. real(), coeff[ i ].imag() );

   for( ; a[ n ] == cmplx<fp> (0, 0); n-- )
      {
      res[ n ] = 0;
      }

   a1 = new cmplx<fp> [ n ];
   for( ; n > 2; n-- )
      {
      // Calculate coefficients of f'(x)
      for( i = 0; i < n; i++ )
         a1[ i ] = a[ i ] * cmplx<fp> ( float_precision( n - i ), float_precision( 0 ) );

      u = startpoint( n, a );
      z0 = float_precision( 0 );
      ff = f0 = a[n].real() * a[n].real() + a[n].imag() * a[n].imag();
      f0z = a[ n - 1];
      if( a[ n - 1] == cmplx<fp> (0) )
         z = float_precision( 1 );
      else
         z = -a[ n ] / a[ n - 1 ];
      dz = z = z / cmplx<fp>( abs( z ) ) * cmplx<fp> ( u / float_precision( 2 ) );
      f = feval( n, a, z, &fz );
      r0 = float_precision( 2.5 ) * u;
      eps = float_precision( 4 * n * n ) * f0 * float_precision( pow( 10, -20 * 2.0 ) );

      // Start iteration
      for( itercnt = 0; z + dz != z && f > eps && itercnt < MAXITER; itercnt++)
         {
         f1 = feval( n - 1, a1, z, &f1z );
         if( f1 == float_precision( 0 ) )
            dz *= cmplx<fp>( 0.6, 0.8 ) * cmplx<fp>( 5.0 );
         else
            {
            float_precision wsq;
            cmplx<fp> wz;

            dz = fz / f1z;
            wz = ( f0z - f1z ) / ( z0 - z );
            wsq = wz.real() * wz.real() + wz.imag() * wz.imag();
            stage1 = ( wsq/f1 > f1/f/float_precision(4) ) || ( f != ff );
            r = abs( dz );
            if( r > r0 )
               {
               dz *= cmplx<fp>( 0.6, 0.8 ) * cmplx<fp>( r0 / r );
               r0 = float_precision( 5 ) * r;
               }
            }
         z0 = z;
         f0 = f;
         f0z = f1z;
iter2:
         z = z0 - dz;
         ff = f = feval( n, a, z, &fz );
         if( stage1 )
            {  // Try multiple steps or shorten steps depending of f is an improvement or not
            int div2;
            float_precision fn;
            cmplx<fp> zn, fzn;

            zn = z;
```

```
         for( i = 1, div2 = f > f0; i <= n; i++ )
            {
            if( div2 != 0 )
               {  // Shorten steps
               dz *= cmplx<fp>( 0.5 );
               zn = z0 - dz;
               }
            else
               zn -= dz;  // try another step in the same direction

            fn = feval( n, a, zn, &fzn );
            if( fn >= f )
               break; // Break if no improvement

            f = fn;
            fz = fzn;
            z = zn;

            if( div2 != 0 && i == 2 )
               {// To many shortensteps try another direction
               dz *= cmplx<fp>( 0.6, 0.8 );
               z = z0 - dz;
               f = feval( n, a, z, &fz );
               break;
               }
            }
         }

      if( float_precision( r ) < abs( z ) * float_precision( pow( 2.0, -26.0 ) ) && f >= f0 )
         {
         z = z0;
         dz *= cmplx<fp>( 0.3, 0.4 );
         if( z + dz != z )
            goto iter2;
         }
      }

   if( itercnt >= MAXITER )
      err--;

   z0 = cmplx<fp> (z.real(), 0.0 );
   if( feval( n, a, z0, &fz ) <= f )
      z = z0;

   z0 = float_precision( 0 );
   for( register int j = 0; j < n; j++ )
      z0 = a[ j ] = z0 * z + a[ j ];
   res[ n ] = cmplx<double> ( (double)z.real(), (double)z.imag() );
   }

quadratic( n, a, res );
delete [] a1;
delete [] a;

return( err );   }
```

Appendix C: Int_precision Example

This example illustrates the use and mix of int_precision with standard types like int. It calculate digits number 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 expression for π is dating back to 1770.
Lambert found the continued fraction below that yields 2 significant digits of π for every 3 terms.

$$\pi = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^3}{7 + \cfrac{4^4}{9 + \cdots}}}}}$$

```
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
```