High Precision

The High_Precision package provides the C++ programmer with the ability to do high precision floating point math to any programmer/user determined degree of precision desired, limited only by machine capacity and time allowed for the desired calculations.

The package consists of appropriate headers and libraries for both static and dynamic linking and presents the programmer with two classes; HPnum which handles basic arithmetic, string conversion to and from high precision numbers and base to base conversion of floating point numbers. The second class, Func, provides a basic set of math functions common to a number of scientific and engineering needs. Statements using these functions appear straight forward as C++ operator overrides are employed.

The trial version is restricted only in that it can be used for 30 days.

The High_Precision package contains the following modules: High_Precision.h, High_Precision.lib, High_Precision.dll, High_Precision_Static.h and High_Precision_Static.lib. The package also contains this documentation.

To use the package, first decide whether you wish to use the DLL or the static link version. Include the appropriate header file as an include statement in the program's header file. When linking, include the appropriate library file in the linker command line. Place the DLL in the environmental variable "path".

Admonitions and limitations:

The precision is passed to the constructor of each class invocation. The value must be consistent throughout the program. It is suggested that the programmer define the value using the "#define" statement and use this definition in all cases. Failure to do so will result in chaos in the form of erroneous results to program crashes to infinite loops.

Example:

```
#define PR 50 // about 100 digits
```

…
Func F(PR);
HPnum var(PR)

By strict observance of scope and lifetime, it is possible to write functions to which the precision is passed as a variable integer argument. All used class instances must be defined within the function. High precision values can be passed as arguments if they have been converted to strings. A function can return a high precision value if it is defined as char*, the value is returned as a converted string and the return value is assigned to a CString.

There exists an HPnum default constructor without an argument. It is used internally and is what allows the programmer/user to select the desired precision. If you accidentally use it in your program, the program will compile and link but runtime chaos will result.

Your choice of precision should employ common sense. Too small a value and you would be better to use double variables. Too large a value simply exponentially increases the calculation times. Memory is allocated for each HPnum class invocation. The package has been checked for memory leaks, they become obvious quickly in any iterative calculation. It was also written to be thread safe. However, judgment should be used in the allocation of HPnum classes as a computer with limited memory can get to thrashing reading and writing to the page file if available memory is exceeded, thus slowing the program dramatically. The code should work in the .NET versions of C++ if it is classed as “just because it works”.

The integer value passed in the class invocations is the number of slots in the numerical array used for calculation. Precision (number of places) is roughly twice this. In making your choice of precision, it is suggested you add at least one to account for the rounding that occurs during division. Two or three is better but adds to the calculation time. Use your judgment.

Operation descriptions:

HPnum has the following operations where var is an HPnum:

Assignment:	var = value; Value can be: int, unsigned int, long, unsigned long, __int64 , unsigned __int64, char, CString, double or HPnum. A floating point string may contain exponentiation as in “e-xxx”. Do not put spaces in the string. Exponents in constants are also allowed within the constraints of the compiler. Exponents in strings are constrained by signed integer.
Comparison:	int k = var1 == var2; k will be 1 for greater than, 0 for equal and -1 for less than.
Addition:	var1 += var2; var1 will contain the result. Var2 is unmodified as in the remaining operations.
Subtraction:	var1 -= var2;
Multiplication:	var1 *= var2;
Division:	var1 /= var2;
Power:	var1 ^= var2;
Base conversion:	char* = var1.BtoB(int in, int out, arg); // arg = char* or CString. If arg length is zero, the value of var1 will be taken and converted. If the length of arg is > 0 it is used instead. This is necessary when letters representing digits are required for input. Max base = 46.

To convert a result to a string for output:

```
char* or CString = var1.String(
        int,                    // number of decimal places to display
                                // 0 will display them all.
        bool);                  // true = use comma's
```

A character array must be of sufficient length to hold what can be a rather long string or evil will occur. We recommend a CString. The second argument tells the routine to insert formatting commas if true. A leading sign will be produced if the value is negative.

The Func class contains functions which return an HPnum value. Passed arguments are not modified. It should be noted that the algorithms for Ln(x), Log(x) and e^x converge more slowly as x increases. Arguments are in radians for trigonometric functions and decimal for other functions except where noted.

The Func class contains the following functions:

```
// logarithms

HPnum* Ln(HPnum*);          // ln(x); x>0

HPnum* Aln(HPnum*);         // e^x

HPnum* Log(HPnum*);         // log(x); x>0

HPnum* Alog(HPnum*);        //10^x

HPnum* LntoLog(HPnum*);     // ln(x)->log(x)

// power & root

HPnum* Fpower(HPnum*, HPnum*);
                                // (base, power)
                                // a^x=C; [returns C]

HPnum* Afpower(HPnum*, HPnum*);
                                // (base, constant)
```

```
                                    // a^x=C; [returns x]
HPnum* Iroot(HPnum*, Int);    // a^(1/Int)
// trig
HPnum* DtoR(HPnum*);          // degrees to radians
HPnum* RtoD(HPnum*);          // radians to degrees
HPnum *Sin(HPnum*);           // sin(x)
HPnum* Cos(HPnum*);           // cos(x)
HPnum* Tan(HPnum*);           // tan(x)
HPnum* Arcsin(HPnum*);        // arcsin(x); x^2<1
HPnum* Arccos(HPnum*);        // arccos(x); x^2<1
HPnum* Arctan(HPnum*);        // arctan(x); x^2!=1
// constants
HPnum* Pi();                  // pi (defined in header)
HPnum* e();                   // Aln(1) (defined in header)
HPnum* PHI();                 // Fibonacci ratio
// misc
HPnum* Factorial(HPnum*);     // x!; x=Int
```

Because a pointer to HPnum must be returned, the use of a function would appear as follows:

```
HPnum = *Func(…);
```

Example:

```
#define PR 50
Func F(PR);
HPnum var(PR);
HPnum res(PR);
CString m_result;
…
var = “5.12345678998765432112345678909876543 21”
res = *F.Log(&var);
m_result.Format("Log(x) = %s", res.String(0, false));
```

Note! The headers contain defined values, pi and lne10, which have been calculated to 1000 decimal places (over 5 hours). lne10 is Ln(10) and is used for converting from natural logs to base ten logs. pi is used for converting degrees to radians and in the arctan (x>1), and arccos functions. If the user requires more precision, he is advised to perform the calculations himself.

Because of the nature of the overloaded operators for arithmetic, writing complex equations in a single statement is difficult. You will note that a statement such as: var1 = var2 + var2; is not legal. It is best to break the equation down into components and perform the calculation as a series of operations storing appropriate intermediate HPnum values in HPnum variables. While bothersome, it will not add noticeably to the time for the calculation.

Rick Marsh Ph.D.
Tulasi Software Systems
832 Wecoma Lp.
Florence, OR 97439
Office: 541 393 2542
rick.marsh@datanex.com
rick.marsh@oregonfast.net

Appendix:

Code examples:

```
#define PR  25

        Func            test(PR);
        HPnum           arg1(PR);
        HPnum           arg2(PR);
        HPnum           arg3(PR);
        HPnum           res(PR);
        CString         msgl;
…
        arg1 = m_string1;
        arg1 = m_string2;
        int k = arg1 == arg2;
        arg3 = arg1;
        arg3 += arg2;
        m_result = arg3.String(0, true);
        arg3 = arg1;
        arg3 -= arg2;
        m_result = arg3.String(0, true);
        arg3 = arg1;
        arg3 *= arg2;
        m_result = arg3.String(0, true);
        arg3 = arg1;
        arg3 /= arg2;
        m_result = arg3.String(0, true);
        arg3 = arg1;
        arg3 ^= arg2;
        m_result = arg3.String(0, true);

void functionA()
{
```

```
HPnum     e(PR);
HPnum     f(PR);
HPnum     g(PR);
HPnum     h(PR);
HPnum     k(PR);
HPnum     rad(PR);

e = arg1;
res = *test.Ln(&e);
m_arg1.Format("Ln(x) = %s", res.String(0, false));
res = *test.Aln(&res);
m_arg2.Format("Aln(x) = %s", res.String(0, false));
res = *test.Log(&e);
m_result.Format("Log(x) = %s", res.String(0, false));
res = *test.Alog(&res);
m_string.Format("Alog(x) = %s", res.String(0, false));

e = arg1;
rad = *test.DtoR(&e);
res = *test.Sin(&rad);
m_arg1.Format("sin(x) = %s", res.String(0, false));
g = res;
g *= res;
res = *test.Cos(&rad);
m_arg2.Format("cos(x) = %s", res.String(0, false));
h = res;
h *= res;
res = *test.Tan(&rad);
m_result.Format("tan(x) = %s", res.String(0, false));
g += h;
m_string.Format("(sin(x)^2) + (cos(x)^2) =  s",
                                    g.String(0, false));

e = arg1;
f = arg2;
```

```
        res = *test.Fpower(&e, &f);
        m_arg1.Format("a^x = C: C = %s", res.String(0,false));
        res = *test.Afpower(&e, &res);
        m_arg2.Format("a^x = C: x = %s", res.String(0, false));

        e = arg1;
        rad = *test.DtoR(&e);
        m_arg1.Format("radians = %s", rad.String(0, false));
        res = *test.Arctan(&rad);
        m_arg2.Format("arctan(x) = %s", res.String(0, false));
        res = *test.Arcsin(&rad);
        m_result.Format("arcsin(x) = %s", res.String(0, false));
        res = *test.Arccos(&rad);
        m_string.Format("arccos(x) = %s", res.String(0, false));
}
```