

Multiprecision floating-point arithmetic on Apple systems

D. Mitchell* and S. Noble*

13 Mar 2007

Abstract: This paper describes the use of two software packages which facilitate floating point arithmetic, with arbitrary precision (thousands or even millions of digits), on Apple Macintosh computers. Both packages are available on the Internet free of charge, though the use of both packages in commercial applications is limited by license restrictions. Both PowerPC and Intel Core CPUs are supported by both software packages. Configuration and installation of each package is discussed, relative strengths and tradeoffs associated with the two packages are described, and simple example code is given to illustrate common use of the packages.

*Advanced Computation Group, Apple Inc.

1 Introduction

Most software running on desktop computers is limited as to the size of the floating point operands it can employ. In the C language (and variants thereof, e.g. C++ and Objective C), common floating point data types are `float` and `double` involving mantissas of 24 bits and 53 bits, respectively; on some platforms (including Macintosh computers running Mac OS X 10.4 and later), a `long double` type is available, increasing the mantissa size to 106 bits. On many platforms, including the Macintosh, calculations which require greater precision than that offered by these native types mandate the use of additional libraries beyond those provided as a part of the operating system.

In both of the packages discussed herein, floating point data types are provided in which the size of the mantissa is limited only by virtual memory size, which in turn may or may not be limited by disk space. The maximum virtual memory size on a 32-bit computer is 4 GBytes; if the sum of available RAM plus the amount of free space on the disk on which the virtual swap space resides is less than 4 GBytes, the virtual memory size of any process running on that computer is limited by the amount of free space on the disk. On a 64-bit computer, the virtual memory size is almost certainly bound by free disk space.

GMP

GMP (Gnu Multi Precision) is a library which provides basic arithmetic functions for three data types of arbitrary precision: integer, rational, (numbers which can be expressed as x/y where x and y are integers), and floating point. The size of the exponent for floating point types is the native word size of the platform on which it runs, generally 32 or 64 bits. GMP provides two interfaces for all data types: one in C, in which arbitrary precision types are represented by an opaque type and manipulated via C function calls; and a C++ interface in which arbitrary precision types are represented by C++ objects, and C++ operator overloading is used to facilitate arithmetic expressions like “`a = x + y`” directly, just as if the data types were `floats` or `doubles`. The primary strength of GMP is that it is optimized for a wide variety of platforms, including all Macintoshes, via assembly-language implementation of core arithmetic routines. The library is designed and implemented with speed as the primary goal. Other salient features of GMP are:

- The precision of each individual floating point variable can be specified and adjusted, independent of the precision of other variables or of the global default precision; this precision is specified in bits.
- GMP is distributed under the Lesser GPL license.
- GMP has very limited support for transcendental functions; it provides a square root function, but no transcendental functions. However, there is an “MPFR” package that provides a higher-level interface and includes transcendentals, at least in a C++ environment. MPFR is available at <http://www.mpfr.org/mpfr-current/>. No comparable Fortran package is available for GMP.
- There is excellent online documentation available which describes overall operation, every function, and all configuration options in detail.

ARPREC

The ARPREC (Arbitrary Precision) library was written by David Bailey, et al [1] at Lawrence Berkeley Laboratories. It provides for three data types: integer, floating point (known in the ARPREC package as an `mp_real`), and complex (which have a real and an imaginary component, both represented as `mp_reals`). The exponent of a real number is stored as a `double`. ARPREC is available in both C++ and in Fortran-90 versions; the C++ version is discussed here. The `mp_real` arbitrary precision types is a C++ class, and C++ operator overloading is used to facilitate arithmetic expressions directly without the use of a procedural interface. Salient features of ARPREC are:

- Rich support for transcendental functions including logarithms (natural and base 10), trigonometric functions, and exponentials (i.e. x^y where x and y are both arbitrary precision floating point numbers)
- ARPREC includes a package which can calculate the definite integral of an arbitrary single-variable function $f(x)$ between two points, without the library knowing anything about the function other than how to call the function to calculate $f(x)$ for an arbitrary x .
- ARPREC does in fact provide for different variables to be declared with different levels of precision. The only thing to keep in mind is that the overall maximum working level of precision is determined by the global variable `mpnw`, which must be at least as high (in words) as the maximum level of precision for any individual variable or array.

On the last item above, say one had declared the maximum precision level to be 10,000 words, where 1 word is approximately 14.4 digits. Then the syntax for declaring variables A and B to be of 10,000-word precision is merely

```
mp_real a;  
mp_real b;
```

whereas if one wishes to declare variable C to be of precision 1,000 words, and D to be of precision 5000 words, then one would write

```
mp_real c(1000);  
mp_real d(5000);
```

The Fortran language does not permit completely flexible precision like this, but modules defined in the fortran directory permit one to define three sets of precision levels: intermediate, regular and extended, and to control the precision level of each.

2 Installing the packages

Configuring and installing GMP

The source for GMP can be downloaded from here:

```
http://www.swox.com/gmp/index.html
```

See “Download the latest release of GMP” and select a server near you. Once you’ve downloaded the package (which is a tarball), double-click on it in the Finder to unpack it. Subsequent actions depend on the platform on which you wish to use the library. It is highly recommended that you configure and build GMP on the platform type on which you will actually be using it.

Note: configuration of the GMP C++ interface is optional. If you wish to configure this option, you need to append the command line options “`--enable-cxx --disable-shared`” (without the quotes) when running the configure program as described below.

Configuring GMP for 32-bit PowerPC and 32-bit Intel

Note: 64-bit support is limited on Mac OS X 10.4: many system libraries and frameworks are only available in 32-bit versions, so even if you have 64-bit hardware (i.e., a G5 or a Core 2 Duo) you might need to configure GMP for 32 bits.

Configuring GMP to run in 32-bit mode involves different procedures when running on 32-bit processors (e.g. G4, Core Duo) vs. 64-bit processors (e.g. G5).

To configure GMP for a 32-bit processor:

In a Terminal window, cd to the directory resulting from double-clicking the downloaded package and do the following.

```
% setenv RANLIB "ranlib -c"  
% ./configure
```

If you want to enable the C++ layer, that would be

```
% ./configure --enable-cxx --disable-shared
```

Configuring GMP for 32-bit Mode on a 64-bit Processor

As mentioned above, for reasons of compatibility with other software, you might wish to configure GMP to run in 32-bit mode on a 64-bit processor. Currently this can only be done on the PowerPC 970 (G5); it is not possible to configure GMP to run in 32-bit mode on a 64-bit Intel processor (Xeon) at this time.

To configure GMP to run in 32-bit mode on a 64-bit PowerPC:

In a Terminal window, cd to the directory resulting from double-clicking the downloaded package and do the following.

```
% setenv RANLIB "ranlib -c"  
% ./configure ABI=mode32
```

If you want to enable the C++ layer, that would be

```
% ./configure --enable-cxx --disable-shared ABI=mode32
```

Configuring GMP for 64-bit PowerPC

In a Terminal window, cd to the directory resulting from double-clicking the downloaded package and do the following.

```
% ./configure CFLAGS="-m64 -fast"
```

Configuring GMP for 64-bit Intel

This configuration is not yet supported by GMP “out of the box” yet. You need to download another patch package, written by Jason Martin and freely available here:

```
http://www.math.jmu.edu/~martin/gmp-4.2.1-core2-port.tar.gz
```

Unpack that package by double-clicking on it in the Finder. Suppose your main GMP package was unpacked into `/tmp/gmp-4.2.1`, and Jason Martin’s patch was unpacked into `/tmp/gmp-4.2.1-core2-port`. Then do this:

```
% cd /tmp/gmp-4.2.1-core2-port
% ./install_gmp_4.2.1_core2_patch.sh /tmp/gmp-4.2.1
% cd /tmp/gmp-4.2.1
% ./configure --build=x86_64-apple-darwin CFLAGS="-m64 -fast"
```

Building and installing GMP (all platforms)

To build GMP, do the following:

```
% make
% make check
```

Of course if any errors are reported by any of the above steps, stop and diagnose the failure. To install the library and its header files (this requires an admin password):

```
% sudo make install
```

Configuring and installing ARPREC

Since the ARPREC package has no platform-dependent optimizations, configuration and installation are very straightforward. The package is available here:

```
http://crd.lbl.gov/~dhbailey/mpdist/index.html
```

Download the tarball referenced in the section entitled “ARPREC.” Unpack it by double-clicking on the downloaded package. In a Terminal window, cd to the directory resulting from double-clicking the downloaded package and do the following.

If compiling in 32-bit mode (Intel or PPC):

```
% setenv CXX "/usr/bin/g++"  
% ./configure  
% make  
% make check
```

If compiling in 64-bit mode (Intel or PPC):

```
% setenv CXX "/usr/bin/g++ -m64"  
% ./configure  
% make  
% make check
```

To install the library and its header files (this requires an admin password):

```
% sudo make install
```

3 Programming with the GMP C interface

3.1 Calculating e , the natural-logarithm base

Consider the simple infinite series which can be used to calculate e , the base of natural logarithms:

$$\begin{aligned} e &= \sum_{n=0}^{\infty} \frac{1}{n!} \\ &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots \end{aligned}$$

While this is by no means the most efficient way to calculate e , it does serve as a straightforward example of the use of GMP. With the GMP C library, the variable e is declared as type `mpf_t`. A routine to calculate e , to a specified number of bits of precision, is implemented as follows:

```
/*
 * Evaluate the series 1/0! + 1/1! + 1/2! + 1/3! + 1/4! ...
 * On input, 'bits' is the desired precision in bits.
 * On output, e' contains the calculated value.
 */
void calculateE(
    unsigned bits,
    mpf_t e)
{
    mpf_t lastE;
    mpf_t invFact;          /* 1/2!, 1/3!, 1/4!, etc. */
    unsigned term;         /* 2, 3, 4... */

    /* initial conditions, including the first two terms */
    mpf_init_set_ui(lastE, 0); /* lastE := 0 */
    mpf_set_ui(e, 2);          /* e := 2 */
    mpf_init_set_ui(invFact, 1); /* invFact := 1/1! */
    term = 2;

    for(;;) {
        mpf_div_ui(invFact, invFact, term); /* invFact /= (term) */
        mpf_add(e, e, invFact);             /* e += 1 / (term!) */

        /* if e == lastE, within the requested precision, we're done */
        if(mpf_eq(e, lastE, bits)) {
            break;
        }
        mpf_set(lastE, e); /* lastE := e */
        term++;
    }
}
```

```

    /* free memory associated with mpf_t's we allocated */
    mpf_clear(lastE);
    mpf_clear(invFact);
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

3.2 Calculating π

An efficient and common means to calculate π is the Gauss–Legendre algorithm, defined [2] as:

Initial Values:

$$\begin{aligned}
 a_0 &= 1 \\
 b_0 &= \frac{1}{\sqrt{2}} \\
 t_0 &= \frac{1}{4} \\
 p_0 &= 1
 \end{aligned}$$

Iterate the following:

$$\begin{aligned}
 x_{n+1} &= \frac{a_n + b_n}{2} \\
 y_{n+1} &= \sqrt{a_n b_n} \\
 t_{n+1} &= t_n - p_n (a_n - x_{n+1})^2 \\
 a_{n+1} &= x_{n+1} \\
 b_{n+1} &= y_{n+1} \\
 p_{n+1} &= 2p_n
 \end{aligned}$$

After each iteration, the value of π is

$$\pi \approx \frac{(a_n + b_n)^2}{4t_n}$$

The Gauss–Legendre algorithm using the GMP C interface

A routine to calculate π , to a specified number of bits of precision, using the GMP C interface, is implemented as follows:

```

/*
 * Gauss-Legendre routine.

```

```

* On input, 'bits' is the desired precision in bits.
* On output, 'pi' contains the calculated value.
*/
void calculatePi(
    unsigned bits,
    mpf_t pi)
{
    mpf_t lastPi;
    mpf_t scratch;

    /* variables per the formal Gauss-Legendre formulae */
    mpf_t a;
    mpf_t b;
    mpf_t t;
    mpf_t x;
    mpf_t y;
    unsigned p = 1;

    mpf_init_set_ui(lastPi, 0);
    mpf_init(x);
    mpf_init(y);
    mpf_init(scratch);

    /* initial conditions */
    mpf_init_set_ui(a, 1);          /* a := 1 */
    mpf_init(b);                   /* b := 1 / sqrt(2) */
    mpf_sqrt_ui(b, 2);
    mpf_ui_div(b, 1, b);
    mpf_init_set_ui(t, 4);         /* t := 1/4 */
    mpf_ui_div(t, 1, t);

    for(;;) {
        /* x := (a+b)/2 */
        mpf_add(x, a, b);
        mpf_div_ui(x, x, 2);

        /* y := sqrt(a*b) */
        mpf_mul(y, a, b);
        mpf_sqrt(y, y);

        /* t := t - p * (a-x)**2 */
        mpf_sub(scratch, a, x);
        mpf_pow_ui(scratch, scratch, 2);
        mpf_mul_ui(scratch, scratch, p);
        mpf_sub(t, t, scratch);
    }
}

```

```

    /*
    * a := x
    * b := y
    * p := 2p
    */
    mpf_set(a, x);
    mpf_set(b, y);
    p <<= 1;

    /* pi := ((a + b)**2) / 4t */
    mpf_add(pi, a, b);
    mpf_pow_ui(pi, pi, 2);
    mpf_mul_ui(scratch, t, 4);
    mpf_div(pi, pi, scratch);

    /* if pi == lastPi, within the requested precision, we're done */
    if(mpf_eq(pi, lastPi, bits)) {
        break;
    }
    mpf_set(lastPi, pi);
}
/* free memory associated with mpf_t's we allocated */
mpf_clear(a);
mpf_clear(b);
mpf_clear(t);
mpf_clear(x);
mpf_clear(y);
mpf_clear(lastPi);
mpf_clear(scratch);
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result. This can calculate 10000000 bits of π in 61.6 seconds on a 2.66 GHz Mac Pro (a 64-bit Intel machine).

4 Programming with the GMP C++ interface

4.1 Calculating e , the natural-logarithm base

Refer to Section 3.1, which describes an algorithm for calculating e . When using the GMP C++ interface, the variable e is declared as an instance of class `mpf_class`; most arithmetic is performed using standard C++ operators on e and other `mpf_class` objects directly, just as if they were declared as `floats` or `doubles`. A routine to calculate e , to a specified number of bits of precision, is implemented as follows:

```

/*
 * Evaluate the series 1/0! + 1/1! + 1/2! + 1/3! + 1/4! ...
 * On input, 'bits' is the desired precision in bits.
 * On output, 'e' contains the calculated value.
 */
void calculateE(
    unsigned bits,
    mpf_class &e)
{
    /* initial conditions, including the first two terms */
    mpf_class lastE(0.0);
    mpf_class invFact(1.0);          /* 1/2!, 1/3!, 1/4!, etc. */
    unsigned term = 2;              /* 2, 3, 4... */
    e = 2;

    for(;;) {
        invFact /= term;
        e += invFact;

        /* if e == lastE, within the requested precision, we're done */
        if(mpf_eq(e.get_mpf_t(), lastE.get_mpf_t(), bits)) {
            return;
        }
        lastE = e;
        term++;
    }
    /* NOT REACHED */
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

The Gauss–Legendre algorithm using the GMP C++ interface

Refer to Section 3.2 for a description of the Gauss–Legendre algorithm for calculating π . Using the GMP C++ interface, this algorithm is implemented as follows:

```

/*
 * Gauss-Legendre routine.
 * On input, 'bits' is the desired precision in bits.
 * On output, 'pi' contains the calculated value.
 */
void calculatePi(
    unsigned bits,
    mpf_class &pi)
{

```

```

mpf_class lastPi(0.0);
mpf_class scratch;

/* variables per the formal Gauss-Legendre formulae */
mpf_class a;
mpf_class b;
mpf_class t;
mpf_class x;
mpf_class y;
unsigned p = 1;

/* initial conditions */
a = 1;
/* b := 1 / sqrt(2) */
mpf_sqrt_ui(b.get_mpf_t(), 2);
b = 1.0 / b;
t = 0.25;

for(;;) {
    x = (a+b)/2;

    /* y := sqrt(ab) */
    y = a * b;
    mpf_sqrt(y.get_mpf_t(), y.get_mpf_t());

    /* t := t - p * (a-x)**2 */
    scratch = a - x;
    scratch *= scratch;
    scratch *= p;
    t -= scratch;

    a = x;
    b = y;
    p <<= 1;

    /* pi := ((a + b)**2) / 4t */
    pi = a + b;
    pi *= pi;
    pi /= (4 * t);

    /* if pi == lastPi, within the requested precision, we're done */
    if(mpf_eq(pi.get_mpf_t(), lastPi.get_mpf_t(), bits)) {
        break;
    }
    lastPi = pi;
}

```

```

    }
    /* NOT REACHED */
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

5 Programming with ARPREC

5.1 Calculating e , the natural-logarithm base

Refer to Section 3.1 which describes an algorithm for calculating e . When using the ARPREC library, the variable e is declared as an instance of class `mp_real`; most arithmetic is performed using standard C++ operators on e and other `mp_real` objects directly, just as if they were declared as floats or doubles. A routine to calculate e , to a specified precision (passed as a literal epsilon), is implemented as follows:

```

/*
 * Evaluate e as the series 1/0! + 1/1! + 1/2! + 1/3! ....
 * On input, 'eps' is the epsilon indicating the requested precision.
 * On output, e' contains the calculated value.
 */
void eViaSeries(
    mp_real eps,          /* epsilon indicating requested precision */
    mp_real &e)          /* RETURNED */
{
    mp_real lastE(0.0);

    /* starting condition includes the first two terms */
    mp_real invFact(1.0); /* 1/2!, 1/3!, etc. */
    e = 2.0;
    int term = 2;

    for(;;) {
        invFact /= term;
        e += invFact;
        term++;

        /* if e == lastE, within the requested precision, we're done */
        if(abs(e - lastE) < eps) {
            break;
        }
        lastE = e;
    }
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

5.2 Calculating e using other algorithms

ARPREC, with its rich support of transcendental functions and exponentiation, provides some other simple ways to calculate e . Consider the limit

$$e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n$$

This is implemented using ARPREC as follows:

```

/*
 * Evaluate e as the limit, as n-->infinity, of
 * (1 + (1/n)) ^ n
 */
void eViaLimit(
    mp_real eps,          /* epsilon indicating requested precision */
    mp_real &e)          /* RETURNED */
{
    mp_real lastE(0.0);

    /* start at n=4, grow by x32 each loop */
    mp_real n(4.0);
    mp_real oneOverN(0.25);

    for(;;) {
        mp_real b(oneOverN + 1.0);
        e = pow(b, n);

        /* if e == lastE, within the requested precision, we're done */
        if(abs(e - lastE) < eps) {
            break;
        }
        lastE = e;
        n *= 32.0;
        oneOverN /= 32.0;
    }
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

Note that this algorithm has no straightforward implementation using GMP due to the lack of a function which raises a floating point number to a floating point power.

Now consider the Newton-Raphson method for calculating e ; this requires a log function, which GMP lacks.

By definition,

$$\log e - 1 = 0.$$

So we seek to find the zero of the function

$$f(x) = \log x - 1$$

whose derivative is

$$f'(x) = 1/x$$

So for each iteration of Newton–Raphson, we do:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{\log(x_n) - 1}{1/x_n} \\ &= x_n - (\log(x_n) - 1)x_n \end{aligned}$$

The code to implement this using ARPREC is simple:

```
void eViaNewtonRaphson(
    mp_real eps,          /* epsilon indicating requested precision */
    mp_real &e)          /* RETURNED */
{
    mp_real lastE(2.7);  /* initial condition */
    mp_real tmp;

    for(;;) {
        tmp = log(lastE) - 1.0;
        tmp *= lastE;
        e = lastE - tmp;

        /* if e == lastE, within the requested precision, we're done */
        if(abs(e - lastE) < eps) {
            break;
        }
        lastE = e;
    }
}
```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

5.3 Implementing the Gauss–Legendre algorithm using ARPREC

Refer to section 3.2 for a description of the Gauss–Legendre algorithm for calculating π . This algorithm is implemented using ARPREC as follows:

```

/*
 * Gauss-Legendre routine.
 * On input, 'digits' is the desired precision in digits.
 * On output, 'pi' contains the calculated value.
 */
void calculatePi(
    unsigned digits,
    mp_real &pi)
{
    mp_real lastPi(0.0);
    mp_real scratch;

    /* variables per the formal Gauss-Legendre formulae */
    mp_real a(1.0);
    mp_real b;
    mp_real t(0.25);
    mp_real x;
    mp_real y;
    double p = 1.0;

    /* epsilon, to detect when we've hit the desired precision */
    mp_real eps = pow(mp_real(10.0), -((int)digits));

    /* initial conditions */
    b = 1.0 / sqrt(mp_real(2.0));          /* b := 1 / sqrt(2) */

    for(;;) {
        x = (a + b) / 2.0;
        y = sqrt(a * b);

        /* t := t - p * (a-x)**2 */
        t = t - (p * sqr(a - x));

        a = x;
        b = y;
        p *= 2.0;

        /* pi := ((a + b)**2) / 4t */
        pi = sqr(a + b) / (4 * t);

        /* if pi == lastPi, within the requested precision, we're done */
        if(abs(lastPi - pi) < eps) {
            break;
        }
        lastPi = pi;
    }
}

```

```

    }
/* NOT REACHED */
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result. This can calculate 3010300 digits of π in 136.9 seconds on a 2.66 GHz Mac Pro (a 64-bit Intel machine).

6 Using the ARPREC quadrature package

Included in the ARPREC source code, but not compiled into the ARPREC library (`libarprec.a`) proper, is a set of classes which perform a definite integral on a single-variable function $f(x)$ between arbitrary values of x . This software is described in Bailey, et al [3]. Three different quadrature algorithms are available: Gaussian, Error Function, and tanh-sinh; each algorithm is expressed in a subclass of the common abstract class `ArprecIntegrate`. To use one of these subclasses, essentially you instantiate it and call the member function `integral()`, passing a pointer to the function $f(x)$ you wish to integrate, as well as the endpoints $x1$ and $x2$ defining the interval in which you wish to integrate. The `integrate()` function returns an `mp_real` containing the result of the integration. The precision of the calculation is passed to the `ArprecIntegrate` subclass's constructor in the form of two integers, a primary precision and a secondary precision.

For a simple example of the use of the Quadrature package, consider the following integral, describing the area of one quarter of a unit radius circle:

$$\int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

Although this is an extremely inefficient means of calculating π , it can serve as a straightforward example of the use of the quadrature package. Subsequent to library initialization, a routine which performs a quadrature calculation of the above integral is as follows:

```

/* Defaults */
#define LEVEL_DEF          10
#define LEVEL_GAUSS_DEF   9
#define TABLE_SIZE_DEF   13000
#define TABLE_SIZE_LARGE 20000
#define PRIMARY_PRECISION_DEF 400
#define SECOND_PRECISION_DEF 800
#define DEBUG_LEVEL_DEF   2

/*
 * A function to integrate.
 */
typedef mp_real (QuadFcn)(const mp_real &x);

/*

```

```

* The function we integrate, a pointer to which is passed to the
* ArprecIntegrate::integrate() function.
*
* Given an x coordinate, 0 <= x <= 1.0, the function returns the
* associated y coordinate on a circle with radius 1.
*/

```

```

static mp_real circleFunc(const mp_real &x)
{
    return sqrt(1.0 - (x * x));
}

```

```

/*
* The core integration routine.
*/

```

```

static void doIntegrate(
    /* specify function and endpoints */
    QuadFcn quadFcn,
    int      x1,
    int      x2,

    /* quadrature parameters */
    QuadType quadType,          /* QT_Gaussian, etc. */
    int      primaryPrecision,
    int      secondaryPrecision,

    /* result written here */
    mp_real &result)
{
    ArprecIntegrate<mp_real> *integrator;
    int tableSize = TABLE_SIZE_DEF;

    if(primaryPrecision > 800) {
        tableSize = TABLE_SIZE_LARGE;
    }

    switch(quadType) {
        case QT_Gaussian:
            integrator = new QuadGS(LEVEL_GAUSS_DEF, tableSize,
                -primaryPrecision, -primaryPrecision,
                DEBUG_LEVEL_DEF);
            break;
        case QT_ErrFunc:
            integrator = new QuadErf(LEVEL_DEF, tableSize,
                -primaryPrecision, -secondaryPrecision,
                DEBUG_LEVEL_DEF);
            break;
    }
}

```

```

    case QT_TanhSinh:
        integrator = new QuadTS(LEVEL_DEF, tableSize,
            -primaryPrecision, -secondaryPrecision,
            DEBUG_LEVEL_DEF);
        break;
}

int nwords1 = integrator->getPrecWd1();
int nwords2 = integrator->getPrecWd2();

/* Create endpoints with precision getPrecWd2() */
mp::mpsetprecwords(nwords2);
mp_real x1Real((double)x1);
mp_real x2Real((double)x2);

/* Create result with precision getPrecWd1() */
mp::mpsetprecwords(nwords1);
mp_real calcVal(0.0);

calcVal = integrator->integrate(quadFcn, x1Real, x2Real);
result = calcVal;
}

```

See Appendix A for information on obtaining the full source code of a program with `main()` which calls this function and displays its result.

7 Performance

The Gauss-Legendre algorithm was used to measure the performance of GMP and ARPREC, running to varying degrees of precision. The GMP C interface was used for these measurements; use of the C++ interface resulted in a negligible performance penalty relative to the C interface.

All times are in seconds. The accuracy of all output was verified using Hemphill's 10 million digits of Pi [4]. Note that when running the ARPREC tests, it generally takes longer to actually output the result, in decimal to `std::cout`, than it takes to perform the raw binary calculation.

Hardware used for measurements:

- PPC, 32-bit and 64-bit: 2 GHz Power Mac G5, 1 GM RAM
- Intel, 32-bit: 2 GHz Core Duo iMac, 512 MB RAM
- Intel, 64-bit: 2.66 GHz Mac Pro, 1 GB RAM

The relatively small difference in GMP performance when comparing 32-bit and 64-bit ppc is due to the fact that even when compiled in 32-bit mode, the raw mantissa components (called "limbs" in the GMP implementation) are 64 bits. Thus all of the performance gain due to the increased memory bandwidth resulting from having to, for example, fetch and process n limbs of 64 bits instead of $2n$ limbs of 32 bits accrues equally to the 32-bit and the 64-bit ppc versions of the library.

1000000 bits or 301030 digits:

	GMP	ARPREC
PPC 32 bit	7.0	27.7
PPC 64 bit	5.7	23.0
Intel 32 bit	9.2	16.6
Intel 64 bit	2.7	9.0

3000000 bits or 903090 digits

	GMP	ARPREC
PPC 32 bit	54.7	74.9
PPC 64 bit	26.0	70.0
Intel 32 bit	40.9	46.6
Intel 64 bit	12.43	27.8

10000000 bits or 3010300 digits

	GMP	ARPREC
PPC 32 bit	167	397
PPC 64 bit	127	365
Intel 32 bit	204	230
Intel 64 bit	61.6	137

A Full Source code

Full source code, with XCode project files, containing all of the previously described example code can be found at this URL:

`www.apple.com/acg`

The examples include

- Calculating e using the GMP C Interface:
- Calculating π using the GMP C Interface:
- Calculating e using the ARPREC C++ Interface, using Infinite Series, Limit, or Newton-Raphson:
- Calculating π via Gauss–Legendre using the ARPREC C++ Interface:
- Calculating π via Gauss–Legendre using the ARPREC Quadrature Interface:

Acknowledgements

The author is indebted D. Bailey, R. Crandall, E. Prabhakar, and A. Sazegari for assistance in researching this article.

References

- [1] David H. Bailey, Yozo Hida, Karthik Jeyabalan, Xiaoye S. Li, Brandon Thompson *ARPREC*.
<http://crd.lbl.gov/~dhbailey/mpdist/index.html>
- [2] Wikipedia: Gauss-Legendre algorithm
http://en.wikipedia.org/wiki/Gauss-Legendre_algorithm
- [3] David H. Bailey, Karthik Jeyabalan, and Xiaoye S. Li *A Comparison of Three High-Precision Quadrature Schemes*
<http://crd.lbl.gov/~dhbailey/dhbpapers/quadrature.pdf>
- [4] Pi by Scott Hemphill - Project Gutenberg. 1993.
<http://www.gutenberg.org/etext/50>