The Lanczos Approximation

Note: Source code for the implementation presented herein will be posted in the near future.

Introduction

The *Lanczos approximation* is a particularly effective method for computing values of the Gamma function to high precision. Here is one form of this approximation based on the work of Paul Godfrey:

$$\ln \Gamma(z+1) = \ln \mathbf{ZP} + (z+0.5)\ln(z+g+0.5) - (z+g+0.5)$$

Where **Z** is an *n*-dimensional row vector constructed from the function argument, z:

$$\mathbf{Z} = \begin{bmatrix} 1 & \frac{1}{z+1} & \frac{1}{z+2} & \cdots & \frac{1}{z+n-1} \end{bmatrix}$$

And **P** is an *n*-dimensional column vector constructed as the product of several $n \times n$ matrices and the *n*-dimensional column vector **F**:

$$\mathbf{P} = \mathbf{D} \cdot \mathbf{B} \cdot \mathbf{C} \cdot \mathbf{F}$$

where

$$\mathbf{B}_{ij} = \begin{cases} 1 & \text{if } i = 0 \\ -1^{j-i}\dbinom{i+j-1}{j-i} & \text{if } i > 0,\ j \geq i \\ 0 & \text{otherwise} \end{cases}$$

C is a matrix containing coefficients from Chebyshev polynomials:

$$\mathbf{C}_{ij} = \begin{cases} \tfrac{1}{2} & \text{if } i = j = 0 \\ 0 & \text{if } j > i \\ -1^{i-j}\displaystyle\sum_{k=0}^{i}\binom{2i}{2k}\binom{k}{k+j-i} & \text{otherwise} \end{cases}$$

and

$$\mathbf{D}_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j = 0 \\ -1 & \text{if } i = j = 1 \\ \dfrac{\mathbf{D}_{i-1,i-1} 2(2i-1)}{i-1} & \text{otherwise} \end{cases}$$

$$\mathbf{F}_i = \frac{(2i)! \cdot e^{i+g+0.5}}{i! \cdot 2^{2i-1} \cdot (i+g+0.5)^{i+0.5}}$$

and g is an arbitrary real parameter, n is an integer. For suitable choices of g and n, very good precision can be obtained. The typical error can be computed as follows:

$$\mathbf{E} = \mathbf{C} \cdot \mathbf{F}$$

$$|\varepsilon| = \left| \frac{\pi}{2\sqrt{2e}} \left(e^{g} \sqrt{\pi} - \sum_{i=0}^{n-1} -1^{i} \mathbf{E}_i \right) \right|$$

Using GMP

GMP is an arbitrary precision mathematical library. In the form provided, it is most suitable for use with C language programs; in particular, the authors have not (yet) provided so-called C++ wrapper classes for this library. (The current version of GMP is 3.1.1, and it is this version that I used.)

To facilitate an elegant C++ implementation of the *Lanczos approximation*, I have written a partial C++ wrapper for the floating point functions of GMP. My implementation contains two classes, **mpdouble** and **mpcomplex**, the latter utilizing the STL **complex** template class. A third class in my implementation, **matrix**, is a template class that implements many matrix arithmetic operations.

The class **mpdouble** is defined in the file **mpdouble.h** and provides the following public interface:

```
class mpdouble
{
    // Utility functions
    void set_default_prec(unsigned long prec);
    void set_default_base(int base);
    // Constants
    mpdouble pi();
    mpdouble e();
    // Constructors and destructors
    mpdouble();
    mpdouble(const mpdouble &op);
    mpdouble(const char *op);
    mpdouble(unsigned long op);
    mpdouble(long op);
    mpdouble(int op);
    mpdouble(double op);
    ~mpdouble();
    // Assignment operators
    mpdouble &operator=(mpdouble const &op);
    mpdouble &operator=(unsigned long op);
```

```
mpdouble &operator=(long op);
mpdouble &operator=(int op);
mpdouble &operator=(double op);
mpdouble &operator=(const char *op);
// Arithmetic operators
mpdouble &operator+() { return *this; };
mpdouble operator-() const
mpdouble operator+(mpdouble const &op) const
mpdouble operator+(unsigned long op) const
mpdouble operator+(long op) const
mpdouble operator+(int op) const
mpdouble operator+(double op) const
mpdouble& operator+=(mpdouble const &op)
mpdouble& operator+=(unsigned long op)
mpdouble& operator+=(int op)
mpdouble& operator+=(double op)
mpdouble operator-(mpdouble const &op) const
mpdouble operator-(unsigned long op) const
mpdouble operator-(long op) const
mpdouble operator-(int op) const
mpdouble operator-(double op) const
mpdouble& operator-=(mpdouble const &op)
mpdouble& operator-=(unsigned long op)
mpdouble& operator-=(int op)
mpdouble& operator-=(double op)
mpdouble operator*(mpdouble const &op) const
mpdouble operator*(unsigned long op) const
mpdouble operator*(long op) const
mpdouble operator*(int op) const
mpdouble operator*(double op) const
mpdouble& operator*=(mpdouble const &op)
mpdouble& operator*=(unsigned long op)
mpdouble operator/(mpdouble const &op) const
mpdouble operator/(unsigned long op) const
mpdouble operator/(long op) const
mpdouble operator/(int op) const
mpdouble operator/(double op) const
mpdouble& operator/=(mpdouble const &op)
mpdouble& operator/=(unsigned long op)
// Comparison operators
bool operator<(mpdouble const &op);
bool operator>(mpdouble const &op);
bool operator<=(mpdouble const &op);
bool operator>=(mpdouble const &op);
bool operator==(mpdouble const &op);
bool operator!=(mpdouble const &op);
bool operator<(unsigned long op);
bool operator>(unsigned long op);
bool operator<=(unsigned long op);
bool operator>=(unsigned long op);
bool operator==(unsigned long op);
bool operator!=(unsigned long op);
bool operator<(long op);
bool operator>(long op);
bool operator<=(long op);
bool operator>=(long op);
bool operator==(long op);
bool operator!=(long op);
bool operator<(int op);
bool operator>(int op);
bool operator<=(int op);
```

```
    bool operator>=(int op);
    bool operator==(int op);
    bool operator!=(int op);
    bool operator<(double op);
    bool operator>(double op);
    bool operator<=(double op);
    bool operator>=(double op);
    bool operator==(double op);
    bool operator!=(double op);
    // Explicit conversion
    double getdouble();
};

// Non-member functions
// iostream support
::std::ostream &operator<<(::std::ostream &os, mpdouble const &op);
::std::istream &operator>>(::std::istream &is, mpdouble &op);
// Arithmetic operators
mpdouble operator+(unsigned long op1, mpdouble const &op2);
mpdouble operator+(int op1, mpdouble const &op2);
mpdouble operator+(double, mpdouble const &op2);
mpdouble operator-(unsigned long op1, mpdouble const &op2);
mpdouble operator-(int op1, mpdouble const &op2);
mpdouble operator-(double, mpdouble const &op2);
mpdouble operator*(unsigned long op1, mpdouble const &op2);
mpdouble operator*(int op1, mpdouble const &op2);
mpdouble operator*(double op1, mpdouble const &op2);
mpdouble operator/(unsigned long op1, mpdouble const &op2);
mpdouble operator/(int op1, mpdouble const &op2);
mpdouble operator/(double op1, mpdouble const &op2);
// Algebraic functions
mpdouble sqrt(mpdouble const &op);
mpdouble pow(mpdouble const &op1, unsigned long op2);
mpdouble abs(mpdouble const &op);
mpdouble floor(mpdouble const &op);
// Transcendental functions
mpdouble log(mpdouble op);
mpdouble exp(mpdouble x);
mpdouble sin(mpdouble x);
mpdouble cos(mpdouble x);
mpdouble atan2(mpdouble y, mpdouble x);
```

The class **mpcomplex** (in file **mpcomplex.h**) extends **mpdouble** using the **complex** template class from the STL. In addition to implementing the arithmetic operators declared in the template class **complex**, **mpcomplex** defines the following functions:

```
::std::ostream &operator<<(::std::ostream &os, mpcomplex const &op);
mpcomplex log(mpcomplex op);
mpcomplex exp(mpcomplex op);
mpcomplex sin(mpcomplex op);
```

The **matrix** template class provides the following public interface:

```
template <typename T> class matrix
{
    // Constructors and destructors
    matrix(int r, int c);
    matrix(const matrix &op);
    ~matrix();
```

```
    // Assignment operator
    matrix &operator=(const matrix &op);
    // Data extraction
    matrix sub(int r, int c);
    // Arithmetic operators
    matrix &operator*=(const T &op);
    matrix operator*(const T &op);
    matrix &operator/=(const T &op);
    matrix operator/(const T &op);
    matrix operator*(const matrix &op);
    // "Complexification" operator
    operator matrix< complex<T> > ();
};

// Non-member functions
// iostream support
template<typename T> ::std::ostream &operator<<(::std::ostream &os, matrix<T> co
```

The *Lanczos Approximation*

Using the helper classes defined above, it is possible to create a very compact, elegant (and reasonably fast) implementation of the *Lanczos approximation*.

My implementation uses a small set of helper functions. First of these is one that computes the Combination (C_k^n) of two integers yielding an arbitrary precision result:

```
mpdouble Comb(int n, int k);
```

For reasons of convenience, this implementation returns 0 if k<0, and 1 if k>n.

Next come several functions that compute the matrices **B**, **C**, **D**, and **F**. Here are these functions in their entirety:

```
void makeB(matrix<mpdouble> &B, int n)
{
    int r, c;
    for (c = 0; c < n; c++) B[0][c] = 1;
    for (r = 1; r < n; r++)
    {
        for (c = 0; c < n; c++)
        {
            B[r][c] = Comb(r+c-1, c-r);
            if ((r+c)&1) B[r][c] = -B[r][c];
        }
    }
}

void makeC(matrix<mpdouble> &C, int n)
{
    for (int i = 1; i < n; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k <= i; k++)
            {
                C[i][j] += Comb(2*i, 2*k) * Comb(k, k+j-i);
```

```
            }
            if ((i-j)&1) C[i][j] = -C[i][j];
        }
    }
    C[0][0] = 0.5;
}

void makeD(matrix<mpdouble> &D, int n)
{
    D[0][0] = 1;
    D[1][1] = -1;
    for (int i = 2; i < n; i++)
    {
        D[i][i] = D[i-1][i-1] * (2 * (2*i-1));
        D[i][i] /= (i - 1);
    }
}

void makeF(matrix<mpdouble> &F, double g, int n)
{
    for (int a = 0; a < n; a++)
    {
        int i;
        F[a][0] = 2;
        for (i = a + 1; i <= 2 * a; i++)
        {
            F[a][0] *= i;
            F[a][0] /= 4;
        }
        F[a][0] *= exp(mpdouble(a) + g + 0.5);
        F[a][0] /= pow(mpdouble(a) + g + .5, a);
        F[a][0] /= sqrt(mpdouble(a) + g + .5);
    }
}
```

With these functions, it is now possible to create an extremely compact Gamma function implementation. Assuming that the variables **n** (**integer**)**, g** (**double**), and **z** (**mpcomplex**) have been assigned their appropriate values, the implementation takes only a few lines:

```
matrix<mpdouble> B(n, n);
matrix<mpdouble> C(n, n);
matrix<mpdouble> D(n, n);
matrix<mpdouble> F(n, 1);
matrix<mpdouble> P(n, 1);
matrix<mpcomplex> Z(1, n);
matrix<mpcomplex> R(1, 1);

makeB(B, n);
makeC(C, n);
makeD(D, n);
makeF(F, g, n);
P = D*B*C*F;

Z[0][0] = mpdouble(1);
for (int i = 1; i < n; i++)
{
Z[0][i] = 1/(z+i);
}
R = Z * P;
mpcomplex r = z + g + 0.5;
```

```
r = log(R[0][0]) + (z+.5) * log(r) - r;
```

It is this implementation, complete with a simple user interface loop and additional support for arguments with a negative real part, that is contained within the file **lanczos.cpp**.

Some Results

I ran the program **lanczos** in a loop in order to determine the best parameters for use with simple calculator programs. I found that the best precision with the minimum number of coefficients is obtained using the following (n, g) parameter pairs:

$n = 4$, $g = 3.65$
$n = 5$, $g = 4.35$
$n = 6$, $g = 5.15$
$n = 7$, $g = 5.90$

I was also able to find pairs of values for which the approximation produces exceptionally accurate results. For instance, using $n = 80$, $g = 85$, and 1024-bit precision, the Gamma function of 101 is computed as:

94259477598383594208516231244829367495623127947025437683278893534169775993162214

Of course, computing the matrices **B**, **C**, **D**, and **F** is very time consuming (several minutes on a 550MHz Pentium-III system) when this level of precision is used.
