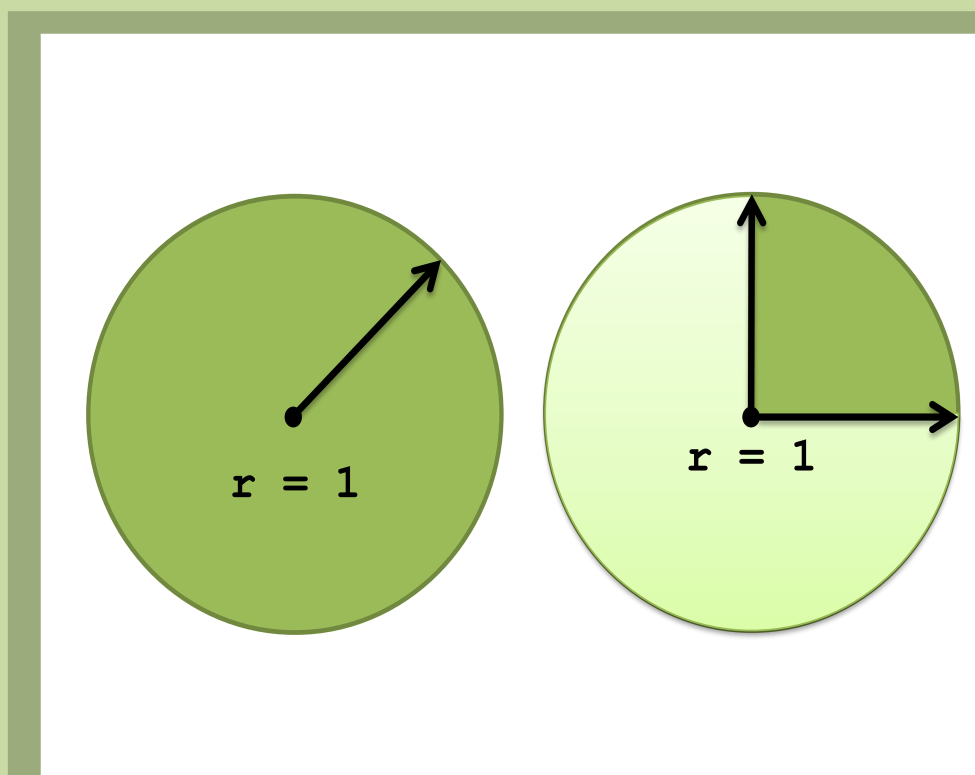
March 2011

CALCULATING π USING RIEMANN SUMS: A COMPUTER SCIENCE APPROACH

Code by Kenneth J. Loomis

```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
#include <gmpxx.h>
#include <pthread.h>

using namespace std;

const int BIT_SIZE = 512;
mpf_class *RIEMANN;
mpf_class RADIUS("1.0", BIT_SIZE);

struct ThreadType
{
  int iter;
  mpf_class* start;
  mpf_class* end;
  mpf_class* rtnVal;
  mpf_class* sum;
};

ThreadType* ThreadData;

void* GetArea( void* args )
{
  ThreadType *data = ( ThreadType* ) args;

  mpf_class width( 0,BIT_SIZE );
  mpf_class f_of_x( 0, BIT_SIZE );
  mpf_class f_of_x2( 0, BIT_SIZE );
  mpf_class trapizoid( 0, BIT_SIZE );
  mpf_class i(*data->start, BIT_SIZE );
  mpf_class sum( 0, BIT_SIZE );

  int iteration = data->iter;
  width = ( i * RADIUS / *RIEMANN );
  f_of_x = sqrt( RADIUS * RADIUS - width * width);
  for ( i=i+1; cmp( i, *data->end )<=0; i= i+1 )
  {
        width = ( i * RADIUS / *RIEMANN );
        f_of_x2 = f_of_x;
        f_of_x = sqrt( RADIUS * RADIUS - width * width);
        trapizoid = RADIUS / *RIEMANN * ( f_of_x + f_of_x2 ) / 2;
        sum = sum + trapizoid;
  }
  *data->rtnVal = sum;
  pthread_exit(0);
  return (void*) 0;
}

int main( int argc, char *argv[] )
{
  int PARTS = 1;
  switch (argc)
  {
    case (1):{      break;  }
    case (2):{      RIEMANN = new mpf_class( argv[1], BIT_SIZE ); break;    }
    case (3):{      RIEMANN = new mpf_class( argv[1], BIT_SIZE );
                    PARTS = atoi( argv[2] ); break; }
    default:{       cout << "Incorrect usage!" << endl;
                    exit ( 1 ); break; }
  }

  ThreadData = new ThreadType[PARTS];
  mpf_class sum( 0, BIT_SIZE );
  mpf_class i( 0, BIT_SIZE );
  mpf_class pi( "3.1415926535897932384626433832795028841971693993751", BIT_SIZE );
  pthread_t thID[PARTS];
  for ( int iter=0; iter<PARTS; iter++ )
  {
        ThreadData[iter].start = new mpf_class( 0, BIT_SIZE );
        ThreadData[iter].end = new mpf_class( 0, BIT_SIZE );
        ThreadData[iter].rtnVal = new mpf_class( 0, BIT_SIZE );
        ThreadData[iter].sum = new mpf_class( 0, BIT_SIZE );
  }
  for ( int iter=0; iter<PARTS; iter++ )
  {
        ThreadData[iter].iter = iter;
        *ThreadData[iter].start = i;
        i = i + ( *RIEMANN / PARTS );
        *ThreadData[iter].end = i;
        pthread_create (&thID[iter], NULL, GetArea, (void*) &ThreadData[iter]);
  }
  for (int iter=0; iter<PARTS; iter++)
  {
    pthread_join ( thID[iter], NULL );
  }
  for ( int iter=0; iter<PARTS; iter++ )
  {
    sum = sum + *ThreadData[iter].rtnVal;
  }
  sum = (sum * 4)/(RADIUS * RADIUS);
  cout << fixed << setprecision(50);
  cout << "**************************************************************" << endl;
  cout << "real pi = " << pi << endl;
  cout << "calc pi = " << sum << endl;
  cout << "**************************************************************" << endl;
  cout << "diff pi ";
  if ( cmp( pi, sum )<0 )
        cout << "+ " << sum-pi << endl;
  else
        cout << "- " << pi-sum << endl;
  cout << "**************************************************************" << endl;

  return 0;
}
```



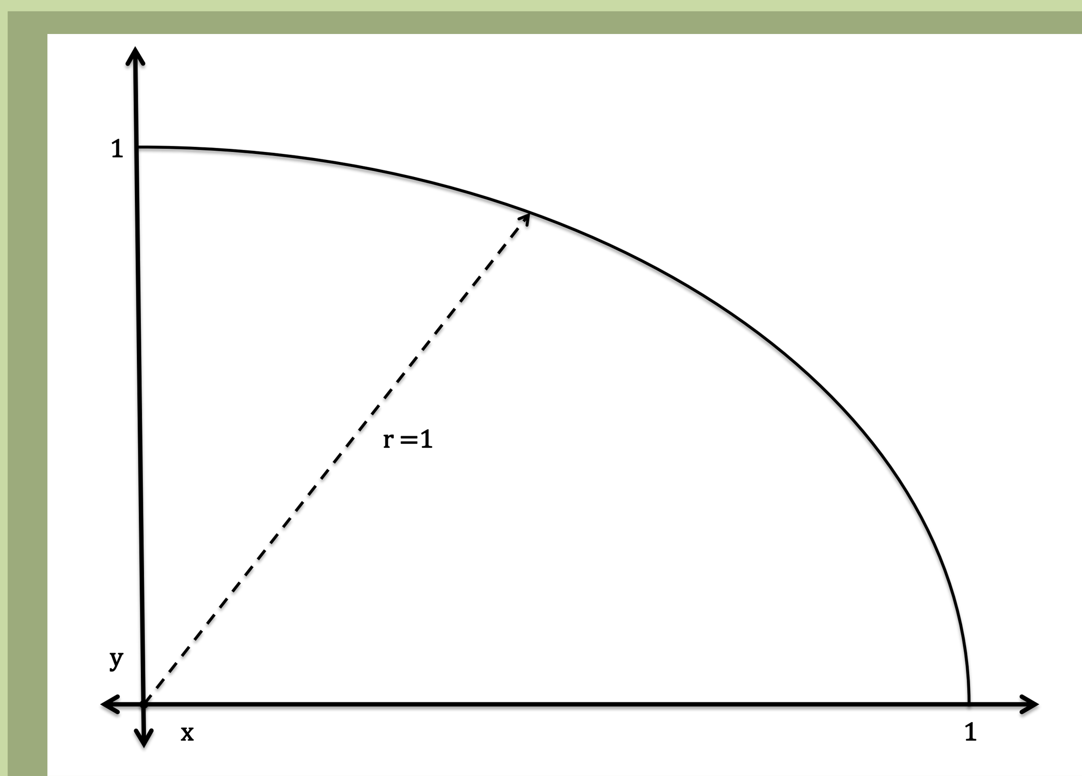
AREA OF A CIRCLE

Recall the formula for the area of a circle (given a circle with radius = 1):

$$\mathrm{A} = \pi \mathrm{r}^2 = \pi(1^2) = \pi$$

Now divide that circle into 4 parts and calculate the area of one quarter of the circle.

$$\mathrm{A} = \frac{1}{4}\pi r^2 = \frac{1}{4}\pi(1^2) = \frac{1}{4}\pi$$



USING CALCULUS

Where y is the formula representing the curve:

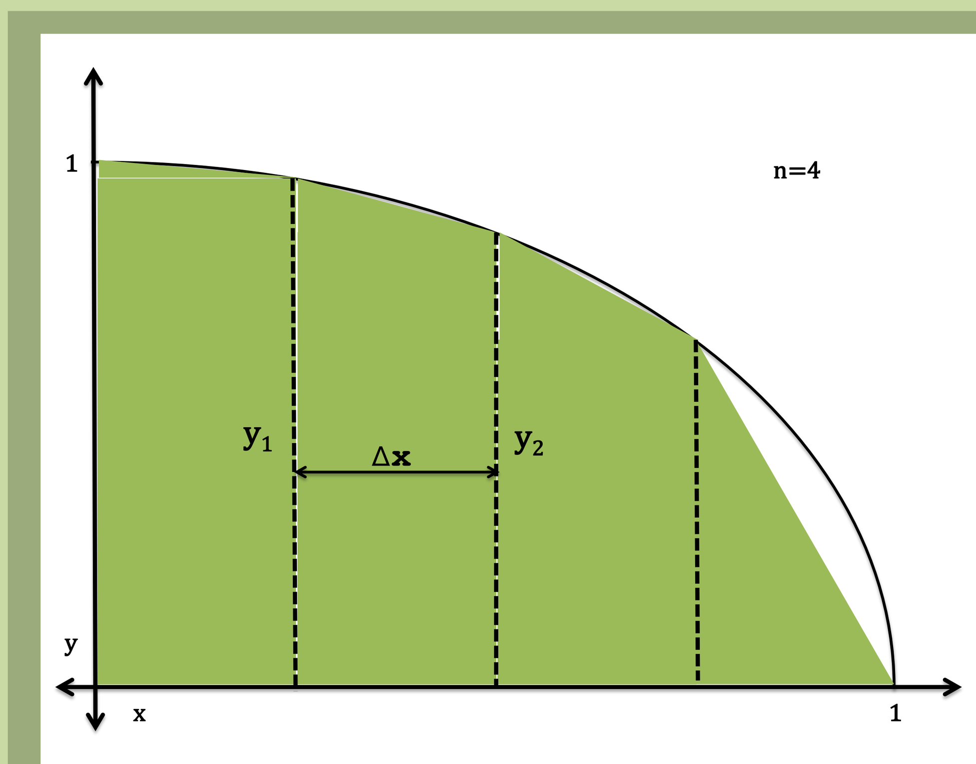
$$x^2 + y^2 = 1^2 \text{ or } y = \sqrt{1-x^2}$$

Using calculus one can determine the area under the curve of the arc to be:

$$A = \int_0^1 y\,dx = \int_0^1 \sqrt{1-x^2}\,dx$$

To calculate π then:

$$A = 4A = 4\int_0^1 \sqrt{1-x^2}\,dx$$



ESIMATING THE INTEGRAL

To estimate the area under the curve one can use Riemann summation of trapezoids.

$$A \approx \sum_{i=0}^{n} \frac{\Delta x(y_i + y_{i+1})}{2}$$

To calculate π then:

$$\pi = 4A \approx 4\sum_{i=0}^{n} \frac{\Delta x(y_i + y_{i+1})}{2}$$

A NOTE ABOUT USING ARBITRARY PRECISION NUMBERS

Because of the limitation on the degrees of precision obtainable when using built-in floating point numbers for c++, it is necessary to using a more complex data type to calculate π with more then a few degrees of accuracy. I am using the GNU Multiple Precision Arithmetic Library (GMP)* for all my calculations. This allows me to have extremely large values of n (the number of trapezoids to divide the area under the arc) and to have extremely small values for the area of each trapezoid.

* http://gmplib.org

SAMPLE RUNS

Run 1: n = 50,000
2 threads on a dual processor
512-bit floating point numbers in GMP

```
bash-4.1$ time ./pi_gmpxx_threads.exe 50000 2
**************************************************************
real pi = 3.14159265358979323846264338327950288419716939937510
calc pi = 3.14159254840682329501628334216844606879713599775070
**************************************************************
diff pi - 0.00000010518296994344636004111105681540003340162440
**************************************************************
real 0m0.090s
user 0m0.162s
sys  0m0.004s
```

Run 2: n = 500,000,000
2 threads on a dual processor
512-bit floating point numbers in GMP

```
bash-4.1$ time ./pi_gmpxx_threads.exe 500000000 2
**************************************************************
real pi = 3.14159265358979323846264338327950288419716939937510
calc pi = 3.14159265358968805542823341040727699627446809275668
**************************************************************
diff pi - 0.00000000000010518303440997287222588792270130661842
**************************************************************
real 13m38.613s
user 26m2.276s
sys  0m0.269s
```