http://www.lawrence.edu/fast/greggj/CMSC110/simpson/Simpson6.png

**Estimating the integral with Simpson's Rule**

Below is the source code for a C program that attempts to use Simpson's rule to compute the integral of *e*-*x*2:

#include <stdio.h>

#include <math.h>

// This is the function we seek to integrate

float f(float x)

{

return exp(-x\*x);

}

// Compute an approximation to the integral

// of f(x) from a to b by using simpson's rule

// with steps of size (b-a)/n

float integrate(float a,float b,int n)

{

float h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = 0;k < n;k++)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

int main (int argc, const char \* argv[])

{

int k,n;

FILE\* output;

output = fopen("table.txt","w");

fprintf(output," n estimate\n");

for(n = 10,k = 1;k < 8;n \*= 10,k++)

fprintf(output,"%8d %.8f\n",n,integrate(0.0,1.0,n));

fclose(output);

return 0;

}

**Round-off error**

The program computes a variety of different estimates by dividing the interval from 0.0 to 1.0 into *n* subintervals, with *n* ranging from 10 up to 100,000,000. We expect Simpson's rule to get more exact as we use more and smaller sub-intervals. Running the program produces this output:

n estimate

10 0.74682420

100 0.74682432

1000 0.74682426

10000 0.74682510

100000 0.74682456

1000000 0.74646360

10000000 0.77563536

Instead of getting better with increasing *n*, these estimates are actually getting worse. What is going on here?

What has happened here is that we have run up against the limitations of the float data type. The float data type in C is a 32 bit floating point number type. Because the float only has 32 bits in which to store a mantissa with a sign and an exponent with a sign floats can only represent decimal numbers up to about 8 digits of precision.

Once we start doing computations that require anything close to 8 digits of precision our calculations will start to be subject to *round-off* error. Round-off error occurs when we try to do arithmetic with numbers that have only a limited number of digits of precision.

One easy and effective method to mitigate the effects of round-off error is to use a data type that offers more digits of precision. Fortunately for us, we have access to a higher precision floating point data type, the double. Here is the previous program rewritten to use double in place float:

#include <stdio.h>

#include <math.h>

// This is the function we seek to integrate

double f(double x)

{

return exp(-x\*x);

}

// Compute an approximation to the integral

// of f(x) from a to b by using simpson's rule

// with steps of size (b-a)/n

double integrate(double a,double b,int n)

{

double h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = 0;k < n;k++)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

int main (int argc, const char \* argv[])

{

int k,n;

FILE\* output;

output = fopen("table.txt","w");

fprintf(output," n estimate\n");

for(n = 10,k = 1;k < 8;n \*= 10,k++)

fprintf(output,"%8d %.14f\n",n,integrate(0.0,1.0,n));

fclose(output);

return 0;

}

Running this program produces the following results:

n estimate

10 0.74682418387591

100 0.74682413281754

1000 0.74682413281243

10000 0.74682413281243

100000 0.74682413281241

1000000 0.74682413281242

10000000 0.74682413281229

This is much better - the results here do a much better job of approximating the result 0.746824132812427 that we wanted to see. The results improve until we get to about *n* = 1000. After that point round-off error effects start to degrade the results and make it no longer worth our while to try using larger values of *n*.

**Further ways to reduce round-off error**

Round-off error is a serious problem that affects computations that require many arithmetic operations. Each time we do an arithmetic computation, round-off error will introduce a small error. If we do many computations in a sequence, the accumulated effect of round-off error can become substantial.

Suppose for example that we are operating in a number system that allows us to retain only four decimal digits of precision. Here is a typical arithmetic computation we might do.

0.1234\*5.678 = 0.7006652

Because our number system only allows us to represent 4 digits in the result, we have to arbitrarily cut off the result to 0.7006. In doing this we have made a small error, called a round-off error. These small round-off errors can get compounded as we do longer calculations involving many arithmetic operations.

Worse yet, there are some calculations that can make round-off error even worse. Consider the following seemingly innocent calculation:

12.34 + 0.05678 = 12.39678

If we have to chop the result off after four digits to 12.39 we would be subjecting ourselves to some serious round-off error. In effect, we have lost all but one of the digits from the second operand 0.05678. Arithmetic operations that seek to add numbers of greatly differing magnitude are subject to serious round-off error problems.

It turns that our Simpson's rule calculation fell prey to precisely this problem. Consider what happens when we try to do the calculation with *n* = 1,000,000 steps. With that step size the typical term in the Simpson's rule sum looks like

http://www.lawrence.edu/fast/greggj/CMSC110/simpson/Simpson7.png

Adding a number with magnitude of about 10-6 to a running total with size of about 0.5 will result in some pretty serious round-off error if we work with C floats that allow only 8 digits of precision. This explains why the results produced by the first program above get worse and worse as we increase *n*.

Fortunately, people who do numerical computations for a living have learned some things about how round-off error works, and have come up with strategies to lessen its effects.

The most significant way to reduce round-off error is to use a superior algorithm. We have already seen that there are several different ways to solve the area under a curve problem using rectangles, trapezoids, and polynomials to develop estimates. In that discussion we saw that the polynomial method produced the best results for a given amount of computational effort, so we are already doing the best we can in that regard.

The next best way to reduce round-off error is to use a higher precision data type. We took advantage of this approach when we switched our program from floats to doubles.

Beyond this point, progress becomes more difficult. A typical mitigation strategy for round-off error starts with an understanding of the sources of round-off error. Not all arithmetic operations are alike when it comes to suffering from round-off error. If we can identify situations where round-off error is more likely to be severe, we can work on mitigation strategies.

As I showed above, a well-known situation where round-off error can be a problem happens when we add two numbers with very different magnitudes. Computations where we do a long sequence of additions in which the two numbers being added have very different magnitudes are a significant source of round-off error. Recognizing that we might have a problem is the first step toward finding a solution. In many such instances, we can find a way to re-organize the computation to mitigate the problem.

It turns out that precisely this effect is at work in the Simpson's rule computation. Consider the code shown here:

double integrate(double a,double b,int n)

{

double h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = 0;k < n;k++)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

If you think carefully about what is going on in the sum we are computing, you will notice that the terms we are adding up get smaller as we work our way across the interval. They do this because the function f(x) we are using here decreases as we move across the interval from left to right. At the same time that the terms are decreasing, the sum is naturally increasing as we add more and more terms to it. Near the end of this process we are adding relatively small terms to a larger sum - this is precisely the situation we saw above that can produce significant round-off error.

There is a way to mitigate the problem in this example. Instead of adding terms from left to right, we add them from right to left, so that early on we are adding small terms to a small total and near the end we are adding somewhat larger terms to our grand total.

Reversing the order of the additions is relatively easy to do. Here is the modified code that does this.

double integrate(double a,double b,int n)

{

double h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = n-1;k >= 0;k--)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

Does this make a difference in the computation? Here is what the program prints now.

n estimate

10 0.74682418387591

100 0.74682413281754

1000 0.74682413281243

10000 0.74682413281243

100000 0.74682413281242

1000000 0.74682413281241

10000000 0.74682413281249

Of all the examples we have seen, this one comes closest to producing the correct result, 0.746824132812427. In particular, this result does not degrade as much when we pass n = 1000.

**A final improvement**

Our area computation code is now about as efficient as we can possibly make it without resorting to a more exotic algorithm. One final improvement I would like to make will improve the usefulness of the integrate function.

One small problem that remains in this integrate function is that for it to work properly we have to have defined the function f(x) in another part of our program. The problem with this arrangement is that we can only define one f(x) function in our program. If we need to write a program in which we need to estimate the area under two different curves over two different intervals, we would be out of luck because the definition of the f(x) is literally compiled into our program and can not be changed as the program runs.

The solution to this problem is to find a way to pass the function f(x) to the integrate function as a parameter. This opens up the possibility that we can call the integrate function twice, once with one function and a second time with a different function. Here is how we do this:

double integrate(double (\*f)(double),double a,double b,int n)

{

double h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = 0;k < n;k++)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

The somewhat weird syntax that you see here says that the first parameter to integrate is a pointer to a function (named f) that takes a double as its sole parameter and returns a double as its result. This gives us a way to pass a function to integrate as a parameter by passing in a pointer to the function we want to use.

Once you have seen this strange syntax trick, it becomes relatively easy to write a program that integrates more than one function. Here is a program that does that.

#include <stdio.h>

#include <math.h>

// Compute an approximation to the integral

// of f(x) from a to b by using simpson's rule

// with steps of size (b-a)/n

double integrate(double (\*f)(double),double a,double b,int n)

{

double h,sum,x0,term;

int k;

h = (b - a)/n;

sum = 0.0;

for(k = n-1;k >= 0;k--)

{

x0 = a + k\*h;

term = h\*(f(x0)+4\*f(x0+h/2)+f(x0+h))/6;

sum += term;

}

return sum;

}

// This is the first function we seek to integrate

double e(double x)

{

return exp(-x\*x);

}

// This is the second function we seek to integrate

double s(double x)

{

return sin(x)/x;

}

int main (int argc, const char \* argv[])

{

printf("The integral of exp(-x\*x) from 0.0 to 1.0");

printf(" is %f.\n",integrate(e,0.0,1.0,100000));

printf("The integral of sin(x)/x from 1.0 to 3.0");

printf(" is %f.\n",integrate(s,1.0,3.0,100000));

return 0;

}