Pat O'Sullivan

Mh4718 Week 5

Week 5

0.0.1 Storage of double type variables

double type variables are stored using 8 bytes using the scheme:

1 sign bit,

11 bits for a biased exponent (actual exponent+1023)

52 bits for the mantissa-1.

double type variables will suffer fewer and smaller round off errors than **float** type variables.

Example 0.1

$2^{24} + 1$ will store exactly as a double but $2^{53} + 1$ will not.

0.0.2 Effects of Rounding Error

0.0.2.1 Program logic should not depend on exactitude!. Computations are frequently inexact because of rounding error and a programmer must beware of relying on mathematical exactness in the logic of her or his programming. For example the following loop will not work because of rounding error:

```
float x =1;
```

```
do
{
x=x-0.2;
cout<<x<<endl;
}while(x !=0);
```

0.0.2.2 Accumulation of errors. If we try to give a float type variable the value 0.2 we see that what is really stored is $0.2 + 0.2(\frac{1}{2^{26}})$. This relatively small error can be greatly magnified by subsequent calculations.

```
float x =0.2;
cout.precision(20);
cout<<x*pow(2.0,26)<<endl; //correct answer is 2^27/10
cout<<pow(2.0,27)<<endl;// difference of 0.2 - the rounding error magnified.
```

Because rounding error can be magnified by subsequent calculations, it follows that the more arithmetic operations are carried out the more likely there is to be an accumulation of errors.
Many small steps can be worse than few bigger ones.
The following program contains a loop which should cause the variable **sum** to have the value 100. The loop is executed using steps of size $\frac{1}{n}$ and n is successively given the values 3,4,5,....100.
Because of round-off error the value of **sum** is rarely exactly 100 and the error varies in size. Some of the biggest errors occur as the number of terms increase

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;
void main()
{

    float sum =0;
    for(int n=3;n<=100;n++)
    {
        sum=0;
        for(int i = 1;i<=n*100;i++)
        {
            sum += 1.0/n; //Ideally this should add up to 100 in steps of 1/n

         }
        cout.precision(20);
        cout<<<<n<<"\t"<<fabs(100-sum)<<endl;
    }
}
```

The lesson illustrated by this example is that one should use machine numbers as much as possible and, if you must use non-machine numbers, then the more steps there are in a calculation the more likely it is that errors will accumulate.

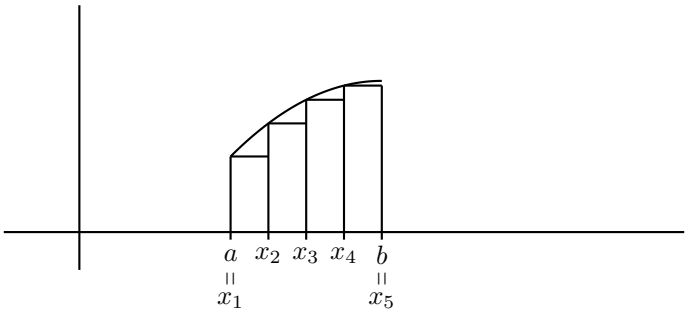
Example 0.2

If we use a program to calculate a Riemann sum then there is a trade off between the mathematical theory and the problem of round off error.

If f is a function integrable over the interval $[a, b]$ a Riemann Sum is determined by first deciding on a partition of the interval. Very often we choose an equal subdivision of the interval into n sub-intervals of equal size. If we decide on n equal sized sub-intervals then each sub-interval will have length $h = \dfrac{b-a}{n}$.
We thus get a Riemann sum:

$$f(a)h + f(a+h)h + f(a+2h)h + \ldots f(a+(n-1)h)h$$

which is an estimate for $\displaystyle\int_a^b f(x)dx$
The following illustrates a Riemann sum consisting of 4 sub-intervals which underestimates an integral:



The Riemann Sum here is

$$\begin{aligned} & f(x_1)h + f(x_2)h + f(x_3)h + f(x_4)h \\ =& (f(x_1) + f(x_2) + f(x_3) + f(x_4))h \\ =& \sum_{i=1}^{4} f(x_i)h \end{aligned}$$

where $h = \dfrac{b-a}{4}$.
We can prove that the finer the partition (the smaller h is here) then the closer

the Riemann Sum will be to the exact value of the interval. However, if we make h too small in a computer program then there is a danger that we cause greater rounding error in the adding up process.