

Roundoff error

Every data in a computer is a collection of bits (zeros and ones).

8 bits = byte

KB=Kbyte = 2^{10} byte=1024byte

MB=Mbyte = 2^{20} byte=1048576bytes

GB=Gbyte = 2^{30} byte=1073741824byte

!! When you buy a computer with 1GB of ram, you get only $10^9 / 2^{30} = 0.93$ GB of memory

There are two classes of types used by computer:

- a) fixed point (integer and long) and
- a) floating point (float, double, complex,...)

Aritmetics with integer *is exact* (except when overflow occurs)

Most of computers are now *32bit* (exception are new opterons and some other supercomputer machines with 64bit processors). The natural unit (*integer*), is 32bit=4byte.

Since int needs also sign (takes one bit) it has the range from -2^{31} to $2^{31} - 1$.

The example computer program shows you the limits of some of the most often used types.

```
int main()
{
    using namespace std;
    cout<<"type      "<<"# bits  minimum      maximum value"<<endl;
    cout<<"char:      " <<<<numeric_limits<char>::digits+1    <<"\t"<<static_cast<int>(numeric_limits<char>::min());
    cout<<"\t\t"<<static_cast<int>(numeric_limits<char>::max())<<endl;
    cout<<"int:         " <<<<numeric_limits<int>::digits+1     <<"\t"<<<numeric_limits<int>::min()      <<"\t"<<<numeric_limits<int>::max()<<endl;
    cout<<"long:        " <<<<numeric_limits<long>::digits+1      <<"\t"<<<numeric_limits<long>::min()     <<"\t"<<<numeric_limits<long>::max()<<endl;
    cout<<"long long:" <<<<numeric_limits<long long>::digits+1<<"\t"<<<numeric_limits<long long>::min() <<"\t"<<<numeric_limits<long long>::max()<<endl;
    cout<<"double:     " <<<<numeric_limits<double>::min()      <<"\t"<<<numeric_limits<double>::max()    <<"\t"<<<numeric_limits<double>::epsilon();
    cout<<"\t"<<<numeric_limits<double>::infinity()<<" " <<<<numeric_limits<double>::signaling_NaN()<<endl;
    cout<<endl;
}
```

output is

```
type      # bits  minimum      maximum value
char:      8      -128         127
int:       32     -2147483648  2147483647
long:      32     -2147483648  2147483647
long long: 64     -9223372036854775808  9223372036854775807
double:    2.22507e-308  1.79769e+308  2.22045e-16  inf nan
```

Aritmetics with floating point numbers *is not exact* causing many difficultis.

In modern computers, the floating point is presented as *Sign * Mantisa * Exponent*. The largers and the smallest floating point number depends on the type. Most often we will use **double**, which needs **8bytes=64bits** and can store numbers between **2.22507e-308** to **1.79769e+308**.

The **overflow error** occurs if we want to store $x > 1.79769 * 10^{308}$ and **underflow** when $x < 2.22507 * 10^{-308}$. This is usually not so crucial, although it occurs if one is not carefull (1/0!!).

The **rundoff error** ϵ occurs when : $1+\epsilon == 1$.

For double on 32 bit machine, it occurs around (only!) 10^{-16} . (Check the simple example program!)

The rundoff error makes bad algorithms unstable

Example: Calculation of spherical Bessel function $j(x)$ with upward and downward recursion.

Spherical bessel functions are solutions of $V = 0$ radial Schroedinger equation

$$\left[-\frac{1}{2} \frac{d^2}{dr^2} + \frac{l(l+1)}{2r^2} \right] [rR_l(r)] = E[rR_l(r)] \quad (1)$$

and satisfy the following recursion relation

$$j_{l+1}(x) = \frac{2l+1}{x} j_l(x) - j_{l-1}(x). \quad (2)$$

A three term linear recursion relation \rightarrow two solutions $j_l(x)$ and $n_l(x)$.

If $l \gg x$, $n_l(x)$ is larger than $j_l(x)$. For large l and small x the upward recursion does not work. **Miller's algorithm**: Use recursion in the opposite direction to get $j_l(x)$ at large l and small x .

We also know

$$j_0(x) = \frac{\sin(x)}{x} \quad j_1(x) = \frac{\sin(x)}{x^2} - \frac{\cos(x)}{x} \quad (3)$$

therefore upward recursion is straightforward. Here is the code:

```
double bessell_j(int l, double x)
{
  // Gives spherical bessel function with upward recursion
  double j0 = fabs(x)>1e-8 ? sin(x)/x : 1; // The error is of the order of x^3=1e-24
  if (l<=0) return j0; // in this case, we do not need j1
  double j1 = fabs(x)>1e-8 ? j0/x-cos(x)/x : x/3.; // Asymptotic expression of small x
  if (fabs(x)<1e-20) return 0;
  double j2 = j1;
  for (int i=2; i<=l; i++){
    j2 = j1*(2*i-1)/x - j0;
    j0 = j1;
    j1 = j2;
  }
  return j2;
}
```

Few points:

- The code **MUST HAVE** enough comments (50/50)
- The singular points needs to be treated separately (Taylor expansion)

Downward recursion starts from sufficiently higher l_{start} than desired l . Good choice is $l_{start} = l + 3\sqrt{l}$. Starting values $j_{l_{start}}$ and $j_{l_{start}-1}$ are not important. Good guess is 0 and 1, respectively. We always need to continue down to $l = 0$ and using $j_0(x)$ normalize the result.

Here is the code:

```
double Bessel_j(int l, double x)
{
  // Gives spherical bessel function with downward recursion
  if (fabs(x)>1) return bessel_j(l, x); // For large x, upward recursion works and is faster
  if (fabs(x)<1e-20) return 0;
  int lstart = 1 + static_cast<int>(sqrt(40*l)/2.); // This is an estimate where we need
  double j2=0, j1=1; // j_{l+2}, j_{l+1} to start the recursion
  double j0, j1, x1=1/x; // 1/x is stored for performance reasons
  for (int i=lstart; i>=0; i--){
    j0 = (2*i+3.)*x1*j1 - j2;
    if (i==1) j1 = j0;
    j2 = j1;
    j1 = j0;
  }
  double true_j0 = sin(x)/x;
  return j1 * true_j0/j0; // renormalizing the results
}
```

Few points:

- To convert double to int, we used C++ style of cast instead of old-fashioned (int)x. It is good practice to avoid old-fashioned cast (easier to look for typecasting in large projects)!
- Multiplication is still faster than division (although not much more nowadays). We stored $1/x$ and use multiplication in the inner-most loop. Use this kind of tricks only in the inner-most loop.

Numerical error in upward recursion for various l as a function of x . For l as small as 7, numerical error becomes of order unity!!

