Roundoff error

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

byte = 8 bits

KiB = KiloByte = $2^{10}$ byte = 1024 byte

MiB = MegaByte = $2^{20}$ byte $\approx$ 1e6 byte

GiB = GigaByte = $2^{30}$ byte $\approx$ 1e9 byte

TiB = TeraByte = $2^{40}$ byte $\approx$ 1e12 byte

PiB = PetaByte = $2^{50}$ byte $\approx$ 1e15 byte

EiB = ExaByte = $2^{60}$ byte $\approx$ 1e18 byte

ZiB = ZettaByte = $2^{70}$ byte $\approx$ 1e21 byte

YiB = YottaByte = $2^{80}$ byte $\approx$ 1e24 byte

Moore’s law: every 18 months doubles, in 15 years increase for $2^{10} \approx 1e3$. 
Most computers are nowadays 64\textit{bit}: a pointer takes 64 bit. 

With 32\textit{bit} system one can address $2^{32} \approx 4e9$ different locations in memory, hence $\approx 2$ GiB RAM requires 64-bit processor+operating system.

With 64 \textit{bit} system one can address $2^{64} \approx 1e19$ locations, hence several ExaBytes.
There are two classes of types used by computer:

a) fixed point (integer and long)

b) floating point (float, double, complex,...)

Arithmetics with integer is exact (except when overflow occurs)

In most of computers, integers are 32-bit=4-byte. Since integer needs also sign (takes one bit) integer has the range from $-2^{31}$ to $2^{31} - 1$.

Larger types are long's, and long long's. The latter are normally 64 bit, while the former are usually 32 bit.

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

```cpp
int main()
{
    using namespace std;
    cout << "type " "# bits minimum maximum value" \n;
    cout << char "<numeric_limits<char>::min() << \n;
    cout << char "<numeric_limits<char>::max() << \n;
    cout << int "<numeric_limits<int>::min() << \n;
    cout << int "<numeric_limits<int>::max() << \n;
    cout << long "<numeric_limits<long>::min() << \n;
    cout << long "<numeric_limits<long>::max() << \n;
    cout << "<numeric_limits<double>::min() << \n;
    cout << "<numeric_limits<double>::max() << \n;
    cout << endl; 
    cout << endl;
    output is
```
<table>
<thead>
<tr>
<th>type</th>
<th># bits</th>
<th>minimum</th>
<th>maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>double</td>
<td>53</td>
<td>2.22507e-308</td>
<td>1.79769e+308</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.22045e-16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nan</td>
</tr>
</tbody>
</table>
Arithmetics with floating point numbers *is not exact* causing many difficulties.

In modern computers, the floating point is presented as $Sign \ast Mantisa \ast Exponent$. The largest and the smallest floating point number depends on the type. Most often we will use **double**, which needs **8 bytes**=64 bits and can store numbers between $2.22507e^{-308}$ to $1.79769e+308$. [roughly: 9-bits exponent, 54-bits mantisa, 1-bit sign]

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

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

For **double**, which takes 8 bytes, it occurs around (only!) $10^{-16}$. (Check the simple example program!)

*The roundoff error makes bad algorithms unstable*
Example: Calculation of spherical Bessel function $j_l(x)$ with upward and downward recursion.

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

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

(1)

and satisfy the following recursion relation

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

(2)

and initial condition:

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

(3)

A three term linear recursion relation → two solutions $j_l(x)$ and $n_l(x)$ are possible.

If $l \gg x$, $n_l(x)$ is larger than $j_l(x)$. For large $l$ and small $x$ the upward recursion for $j_l(x)$ does not work (becomes $n_l(x)$ after a few steps).

The idea is to use Miller's algorithm: Use recursion in the opposite direction to get $j_l(x)$ at large $l$ and small $x$. Here is the code for the upward recursion in Python:
def bessel_upward(l,x):
    """ Upward recursion to compute the spherical bessel function:
    \[ j_{l+1} = \frac{(2l+1)x}{x} j_l - j_{l-1} \]
    Works for large \( x \gg l \).
    Input:
    \[ l \quad -- \text{all bessel functions } j_l \text{ up to } l \text{ (including } l \text{) will be computed} \]
    \[ x \quad -- j_l(x) \]
    Output:
    \[ [j_0(x), j_1(x), \ldots, j_l(x)] \]
    """
    if abs(x)<1e-10:
        j0=1.
        j1=x/3.
    else:
        j0 = sin(x)/x
        j1 = j0/x-cos(x)/x

    res=[j0]
    if l==0: return res

    res.append(j1)
    for i in range(1,l):
        j2 = j1*(2*i+1.)/x-j0
        j0 = j1
        j1 = j2
        res.append(j2)
    return res
and in C++

```cpp
double bessel_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 \( l_{\text{start}} \) than desired \( l \). Good choice is \( l_{\text{start}} = l + 3 \sqrt{l} \). Starting values \( j_{l_{\text{start}}} \) and \( j_{l_{\text{start}}-1} \) 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 for downward recursion in Python:

```python
def bessel_downward(l,x):
    """ Downward recursion to compute the spherical bessel function: 
    j_{l} = (2*l+3)/x * j_{l+1} - j_{l+2} 
    Works for small x < l. 
    Input:
    l    -- all bessel functions j_{l} up to l (including l) will be computed
    x    -- j_{l}(x)
    Output:
    [j_{0}(x), j_{1}(x), .... j_{l}(x)]
    """

    if abs(x)<1e-20: return [1]+zeros(l).tolist()
    lstart = l + int(sqrt(10*l))
    j2=0
    j1=1.
    res=[]
    for i in range(lstart,0,-1):
        j0 = j1*(2*i+1)/x - j2
        if i-1<=l: res.append(j0)
        j2=j1
        j1=j0
    true_j0 = sin(x)/x
    res.reverse()
    return array(res)*true_j0/res[0]
```

and in C++
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 = l + static_cast<int>(sqrt(40*l)/2.); // This is an estimate where we need 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==l) j1 = j0;
        j2 = j1;
        j1 = j0;
    }
    double true_j0 = sin(x)/x;
    return j1 * true_j0/j0; // renormalizing the results
}
Numerical error for $x = 0.1$:

<table>
<thead>
<tr>
<th># upward</th>
<th>downward</th>
<th>exact</th>
<th>diff-up</th>
<th>diff-dn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.998334</td>
<td>0.998334</td>
<td>0.000244</td>
<td>9.98334e-16</td>
</tr>
<tr>
<td>1</td>
<td>0.0333</td>
<td>0.0333</td>
<td>0.000000</td>
<td>1.38778e-16</td>
</tr>
<tr>
<td>2</td>
<td>0.000666</td>
<td>0.000666</td>
<td>0.000000</td>
<td>6.93889e-18</td>
</tr>
<tr>
<td>4</td>
<td>1.05787</td>
<td>1.05772</td>
<td>1.05772</td>
<td>1.49947e-11</td>
</tr>
<tr>
<td>5</td>
<td>2.31094</td>
<td>2.31094</td>
<td>2.31094</td>
<td>2.64698e-23</td>
</tr>
<tr>
<td>6</td>
<td>1.48416</td>
<td>1.48416</td>
<td>1.48416</td>
<td>1.61559e-27</td>
</tr>
<tr>
<td>7</td>
<td>1.92918</td>
<td>1.92918</td>
<td>1.92918</td>
<td>1.26218e-29</td>
</tr>
<tr>
<td>8</td>
<td>0.002893</td>
<td>0.002893</td>
<td>0.002893</td>
<td>4.93038e-32</td>
</tr>
<tr>
<td>9</td>
<td>0.491896</td>
<td>0.491896</td>
<td>0.491896</td>
<td>0</td>
</tr>
</tbody>
</table>
Numerical error in upward recursion for various $l$ as a function of $x$. 

![Graph showing the error of Bessel functions $j_n(x)$ for various $n$. The graph displays the error on a logarithmic scale, with $x$ ranging from 0 to 5 and the error ranging from $10^{-19}$ to $10^{23}$ for different values of $n$. The legend indicates the lines correspond to $n=10$, $n=11$, $n=12$, $n=13$, and $n=14$. The graph illustrates the convergence of the error as $x$ increases.]
1 Second Homework

- Write a python script to compute spherical bessel functions with up and down recursion. Plot the error of your algorithm when compared to scipy version of $j_l(x)$.

- Use f2py to speed up the algorithm: write a fortran subroutine and call it from Python using f2py.

- Use weave to speed up the algorithm: write C++ inline code to speed up the evaluation of $j_l(x)$.