Roundoff error

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

byte=8 bits

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

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

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 *64bit*: a pointer takes 64 bit.

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

With *64 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 32bit=4byte. 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.

```
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
```

Arithmetics with floating point numbers *is not exact* causing many difficulties.

In modern computers, the floating point is presented as $Sign * Mantisa * Exponent$. The largest 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. [roughly: 9-bits exponent, 54-bits mantisa, 1-bit sign]

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 careful (1/0!!).

The **roundoff error** ϵ occurs when : 1+ϵ == 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 $j(x)$ with upward and downward recursion.

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

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

and satisfy the following recursion relation

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

and initial condition:

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

A three term linear recursion relation $\rightarrow$ 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 by jupyter notebook:

Upward recursion

We will evaluate bessel upward recursion using the formula

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

In [2]:
```
from scipy import *
from numpy import *

def bessel_upward(l,x):
    "returns array of j_i from i=0 to i=l, including l"
    res = zeros(l+1)
    if abs(x)<1e-30:
        res[0]=1.
        return res
    j0 = sin(x)/x
    res[0]=j0
    if l==0: return res
    j1 = j0/x - cos(x)/x
    res[1] = j1

    for i in range(1,l):
        j2 = (2*i+1)/x*j1 - j0
        res[i+1]=j2
        j0,j1 = j1,j2

    return res
```

In [3]:
```
from scipy import special

l=10
x=0.1

dat0 = bessel_upward(l,x)
dat1 = special.spherical_jn(range(l+1),x)

diff = dat0-dat1
print(dat0)
print(dat1)
print('difference=', diff)
```

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 for downward recursion in Python:

2 downward recursion

Now we will use recursion:

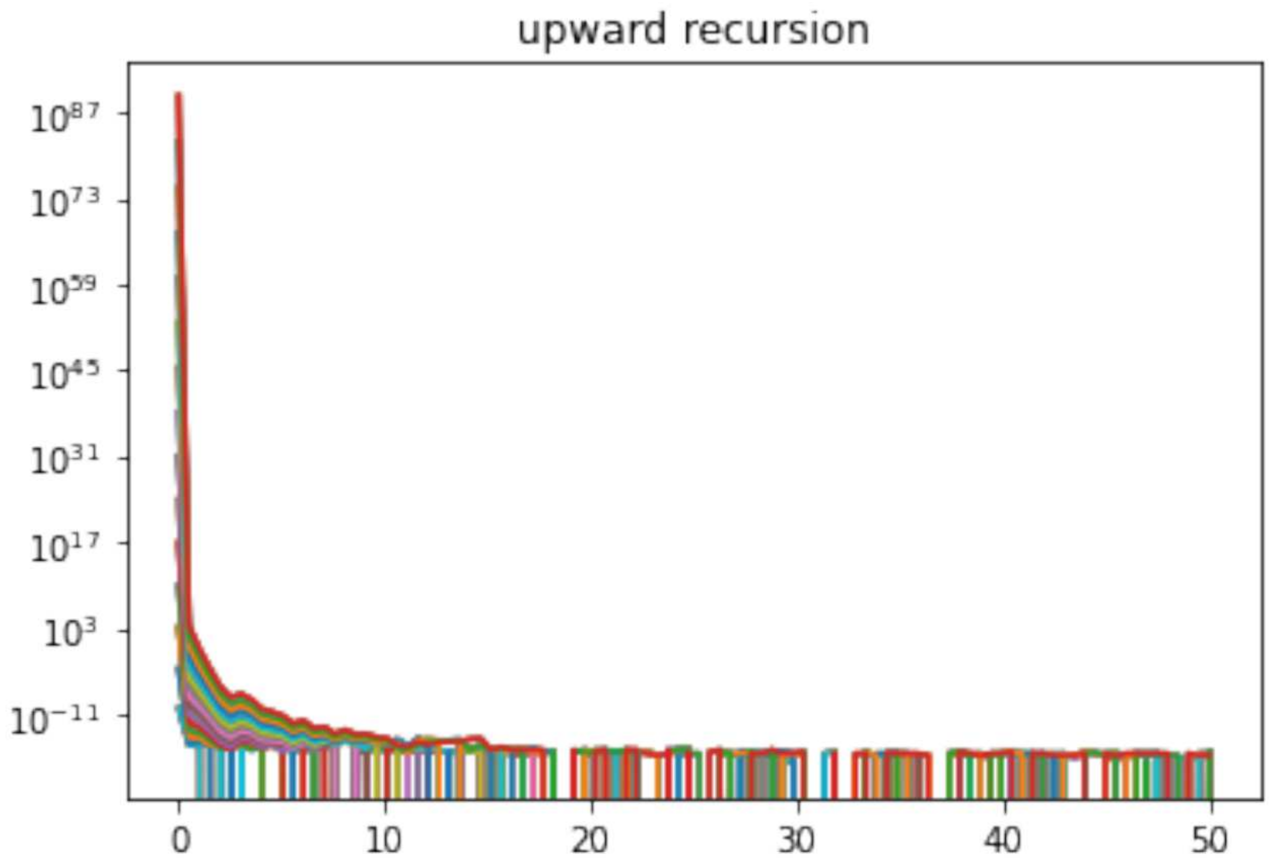
$$j_{l-1} = (2l+1)/x j_l - j_{l+1} \tag{2}$$

[11]:
```
def bessel_downward(l,x):
    "downward recursion"
    if abs(x)<1e-20:
        res = zeros(l+1)
        res[0]=1
        return res
    lstart = l + int(sqrt(10*l))
    j2 = 0.
    j1 = 1.
    res = []
    for i in range(lstart,0,-1):
        j0 = (2*i+1)/x * j1 - j2
        if i-1<=l : res.append(j0)
        j2 = j1
        j1 = j0
    res.reverse()
    true_j0 = sin(x)/x
    res = array(res) * true_j0/res[0]
    return res
```

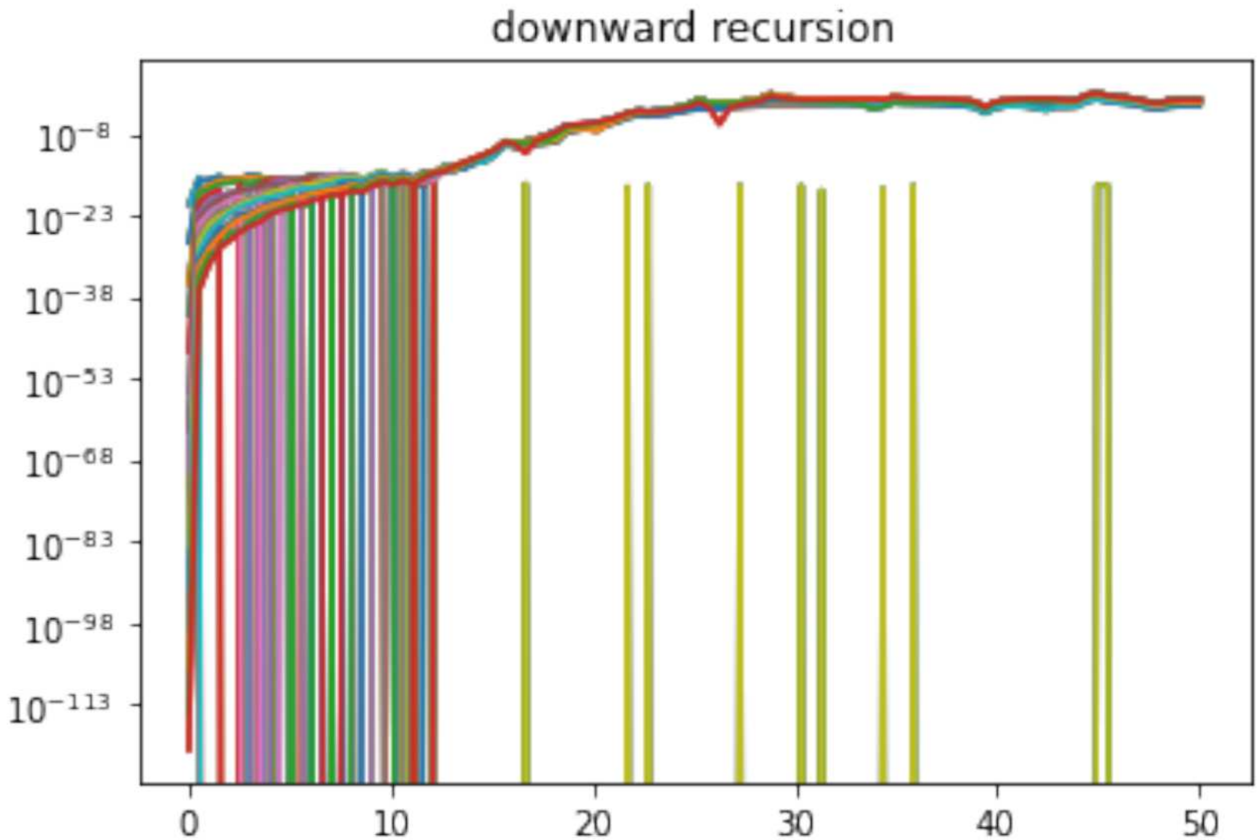
Numerical error for $x = 0.1$:

#	upward	downward	exact	diff-up	diff-dn
0	0.998334	0.998334	0.998334	1.11022e-16	1.11022e-16
1	0.0333	0.0333	0.0333	1.38778e-16	6.93889e-18
2	0.000666191	0.000666191	0.000666191	4.28824e-15	0
3	9.51852e-06	9.51852e-06	9.51852e-06	2.14271e-13	1.69407e-21
4	1.05787e-07	1.05772e-07	1.05772e-07	1.49947e-11	2.64698e-23
5	2.31094e-09	9.61631e-10	9.61631e-10	1.34931e-09	2.06795e-25
6	1.48416e-07	7.39754e-12	7.39754e-12	1.48409e-07	1.61559e-27
7	1.92918e-05	4.93189e-14	4.93189e-14	1.92918e-05	1.26218e-29
8	0.00289362	2.9012e-16	2.9012e-16	0.00289362	4.93038e-32
9	0.491896	1.52699e-18	1.52699e-18	0.491896	0

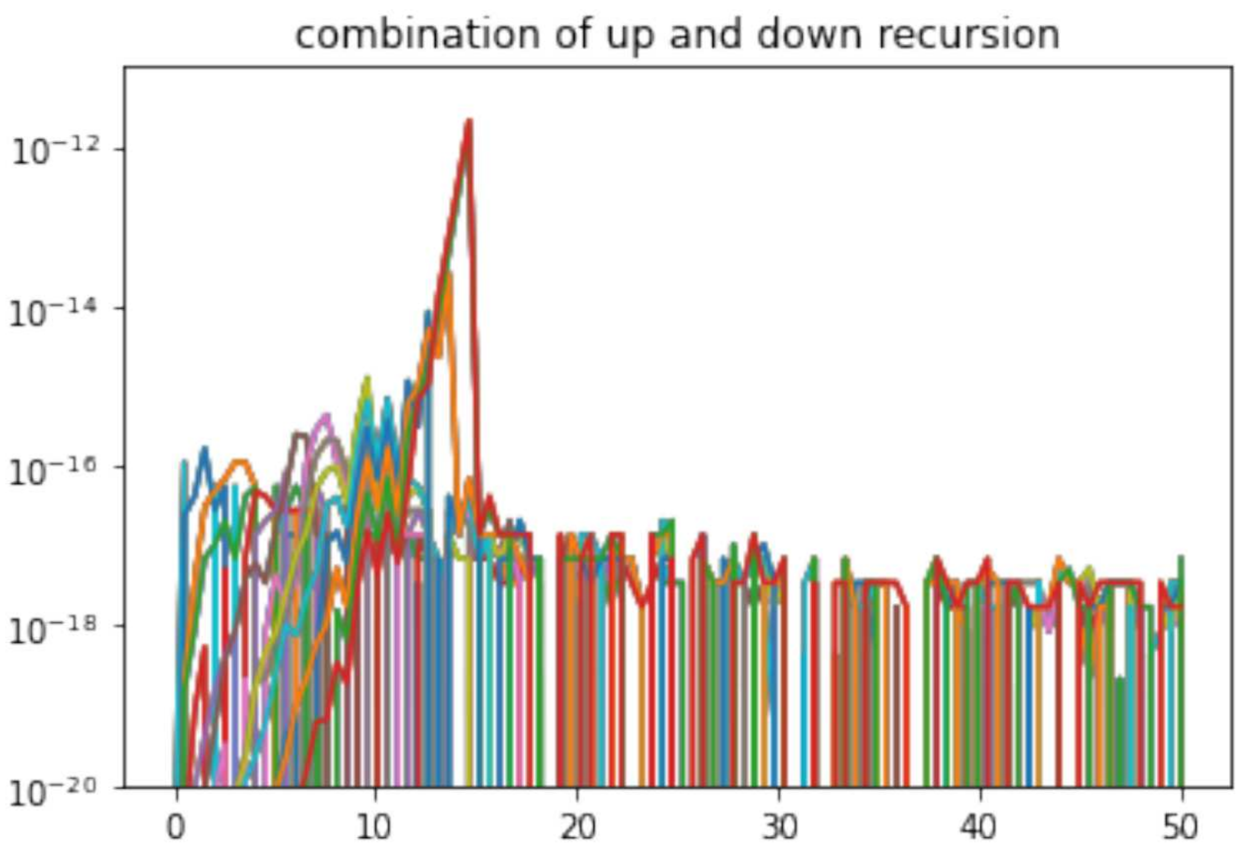
Numerical error for upward recursion for various l as a function of x.



Numerical error for downward recursion for various l as a function of



Combination of upward and downward recursion:



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 $j_l(x)$.
- Optional: Use f2py or pybind11 to speed up the algorithm.
- We want to compute the series of integrals, defined by

$$K_n(z,\alpha,a,b) = \int_a^b dx \frac{x^n}{z+\alpha x} \tag{4}$$

  when $n = 0, 1, ...n_{max} = 10$.
  a and b are numbers between 0 and 1. For simplicity you can choose $a = 0$ and $b = 1$.

  – Derive the recursion relation between K_{n+1} and K_n.
  – Then starting from K_0 you can compute all K_n up to n_{max} using the recursion. This works quite well for $|\alpha/z| >= 1$.

– Choosing z and α so that $|\alpha/z| \ll 1$ (for example $\alpha/z = 10^{-4}$) verify that upward recursion does not lead to accurate results.

– Implement downword recursion for $\alpha/z < 1/2$. Make sure that you start with very accurate value for $K_{n_{max}}$. You can derive a power expansion of $K_{n_{max}}$ in powers of $(\alpha/z)^k$, and evaluate as many terms as needed to achieve desired accuracy (for example 10^{-12}).