About Operating System

UNIX and Unix-like operating system are the best for computational science, especially for editing, compiling, and running C program language. To be specific, this guide is written for the users of

- Certified UNIX operating system; check `https://www.opengroup.org/openbrand/register/`
- The descendants of Berkeley Software Distribution (BSD) operating system, such as FreeBSD, OpenBSD, and so on.
- Linux operating system, such as Debian, openSUSE, Red Hat, slackware, and so on; check `https://www.linuxfoundation.org/` and `https://www.kernel.org/`.

About C Compiler

There are many C code compilers popular today and different command associating with them: portable C Compiler `pcc`, GNU Compiler Collection `gcc`, Intel C++ Compiler `icc`, LLVM Clang `clang`, and IBM XL C/C++ Compilers. To avoid confusions, this guide uses the command `cc` which is the environmental variable linking to the C compiler installed on your computer.

Session 1. Basic of Computational Science

1.1 Volume of a Sphere in $\mathbb{R}^3$

Compiling C code which including header file `math.h`

The following two codes use pre-defined function `sin()` in the header file `math.h`, but their compiling ways are slightly different.

The following code can be compiled by
either `cc -Wall -std=c18 -o code.out code.c` or `cc -Wall -std=c18 -o code.out code.c -lm`

```
#include <stdio.h>
#include <math.h>

int main()
{
  printf("sin(2) = %f \n", sin(2.0));

  return 0;
}
```

The following code can *only* be compiled by using flag `-lm`
`cc -Wall -std=c18 -o code.out code.c -lm`

```
#include <stdio.h>
#include <math.h>

int main()
{
  float a = 2.0;
  printf("sin(2) = %f \n", sin(a));

  return 0;
}
```

In words, if the input of a `math.h` function is a direct number, then the flag `-lm` is not necessary for compiling the function; if the input is a (assigned) variable, then `-lm` must be invoked.

Here are the conclusion and recommendation.
(1) the code must include the header file `math.h` for using `math.h`-defined functions like `sin()` and `fabs()`.
(2) use the flag `-lm` as long as the code is including the header file `math.h`

Mathematical constants in C standard

Before writing the actual code, we have to know how to define the constant π in C programming language. You might think that C library or C standard should already contains such a commonly-used constant; the fact is that it depends on the standard you used to compile the code. C language has many standards:

- C89, also called ANSI C, American National Standards Institute C;
- C11, full name ISO/IEC 9899:2011,
  International Organization for Standardization and International Electrotechnical Commission
- GNU standard

and different standard has different ways to define π. In ASCI C and those ISO/IEC standards, mathematical constants such as π have not already defined; while in GNU C standard, π, e, and some constants are already defined as `M_PI`, `M_E` in the header file `math.h` In other words, `M_PI` is not always available!
Rule: if you want to use `M_PI` defined in `math.h`, you need to use `-std=gnu11` instead of `-std=c18` to compile the code.

```
gcc -Wall -std=gnu11 -o mycode.out mycode.c -lm
```

```
#include <stdio.h>
#include <math.h>

int main()
{
  printf("%.16f", M_PI);

  return 0;
}
```

To be more generally acceptable, **we always prefer the ISO/IEC standard**. Then we need to define π by ourselves by either
`#define M_PI acos(-1.0)` where `math.h` is used in which `M_PI` has been already defined, so this cannot be compiled by GNU C;
or
`#define M_PI 3.14159265358979323846264338327` where `math.h` is not needed,
or if you do not know whether `M_PI` is not defined and/or used in your compiler, then use "if not defined" preprocessor.

```
#ifndef M_PI
#define M_PI 3.14159265358979323846264338327
#endif
```

In either case, the code can be compile by using C standard. For example,
`cc -Wall -std=c18 -o mycode.out mycode.c`

Here is our code **which is not able to be compiled by GNU C standard**

```
1  /*NAME : 1.1_volume.c
2  AUTHOR : Jiasheng He  8-June-2020
3  COMPLIE: cc -Wall -std=c18 -o 1.1_volume.out 1.1_volume.c -lm
4           cannot compiled by GNU C standard
5  RUN    : ./1.1_volume.out
6  ********************************************************************/
7
8  #include <stdio.h>
9  #include <math.h>
10
11 #define M_PI acos(-1.0)
12
13 int main()
14 {
15   double radius, volume;
16
17   printf("Enter the radius of the sphere: ");
18   scanf("%lf", &radius);
19
20   volume = 4.0/3.0 * M_PI * radius * radius * radius;
21
22   printf("Sphere's raduis = %f; sphere's volume = %f \n", radius, volume);
23
24   return 0;
25 }
```

One can also use `pow()` function defined in `math.h` to compute r^3, then line 20 becomes
`volume = 4.0/3.0 * M_PI * pow(radius, 3);`

1.2 Number Representation in Computers

Integer, two's complement representation

An integer occupies 32 bits, and each bit can be either 0 or 1. Unsigned integer ranges

from `0000 0000 0000 0000 0000 0000 0000 0000` $= 0$

to `1111 1111 1111 1111 1111 1111 1111 1111` $= 2^{31} + 2^{30} + \cdots + 2^1 + 2^0 = 2^{32} - 1 = 4294967295$

Signed integer uses *two's complement representation*. The idea of two's complement representation is designed for representing signed integer, and it is fairly easy to understand through an example: 2 in three-bit binary format is 010; since $010 + 110 = 1000$, -2 is thus represented as 110. In four-bit binary format, 2 is 0010 and since $0010 + 1110 = 10000$, -2 is thus represented as 1110. In this way, we have

3-bit binary format using two's complement representation: ranges from -2^2 to $2^2 - 1$

Decimal format	3-bit binary format	3-bit binary format	Decimal format
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
-4	100	100	-4
-3	101	011	3
-2	110	010	2
-1	111	001	1

16-bit binary format using two's complement representation: ranges from -2^{15} to $2^{15} - 1$

Decimal format	16-bit binary format	16-bit binary format	Decimal format
0	0000 0000 0000 0000	0000 0000 0000 0000	0
1	0000 0000 0000 0001	1111 1111 1111 1111	-1
2	0000 0000 0000 0010	1111 1111 1111 1110	-2
⋮	⋮	⋮	⋮
32767	0111 1111 1111 1111	1000 0000 0000 0001	-32767
$-2^{15} = -32768$	1000 0000 0000 0000	1000 0000 0000 0000	-32768
-32767	1000 0000 0000 0001	0111 1111 1111 1111	32767
⋮	⋮	⋮	⋮
-1	1111 1111 1111 1111	0000 0000 0000 0001	1

32-bit binary format using two's complement representation: ranges from -2^{31} to $2^{31} - 1$

Decimal	32-bit binary	32-bit binary	Decimal
0	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000 0000 0000 0000 0000 0000 0000	0
1	0000 0000 0000 0000 0000 0000 0000 0001	1111 1111 1111 1111 1111 1111 1111 1111	-1
⋮	⋮	⋮	⋮
$2^{31} - 1$	0111 1111 1111 1111 1111 1111 1111 1111	1000 0000 0000 0000 0000 0000 0000 0001	$-2^{31} + 1$
-2^{31}	1000 0000 0000 0000 0000 0000 0000 0000	1000 0000 0000 0000 0000 0000 0000 0000	-2^{31}
$-2^{31} + 1$	1000 0000 0000 0000 0000 0000 0000 0001	0111 1111 1111 1111 1111 1111 1111 1111	$2^{31} - 1$
⋮	⋮	⋮	⋮
-1	1111 1111 1111 1111 1111 1111 1111 1111	0000 0000 0000 0000 0000 0000 0000 0001	1

To conclude, the signed integer ranges

from `1000 0000 0000 0000 0000 0000 0000 0000` $\doteq -2^{31} = -2147483648$

to `0111 1111 1111 1111 1111 1111 1111 1111` $= 2^{30} + \cdots + 2^1 + 2^0 = 2^{31} - 1 = 2147483647$

As you can see that the first bit on the left automatically indicates the sign: 0 for positive and 1 for negative. An interesting point is $2^{31} - 1$ is a Mersenne prime. In other words, you cannot simply use the 32-bit integer to find a larger Mersenne prime; some special algorithm must be applied in your code.

Single-precision floating-point number

A single-precision floating-point number occupies 32 bits (see figure below). The bit #0 to bit #22 is called fraction f; bit #23 to bit #30 is called exponent e; the bit #31 represents the sign s. The way in which 32 bits represents the single-precision floating-point number is

Name	Sign s, exponent e, and fraction f	Values
Normal	$0 =$ `00000000` $< e <$ `11111111` $= 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e =$ `00000000` $= 0$ and $f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
± 0	$e =$ `00000000` $= 0$ and $f = 0$	0
$\pm\infty$	$e =$ `11111111` $= 255$ and $f = 0$	`+INF` for $s = 0$, `-INF` for $s = 1$
Not a number	s is ill-defined, $e =$ `11111111` $= 255$, and $f \neq 0$	`NaN`

Example 1 Convert `0 01111100 01000000000000000000000` to its decimal format.

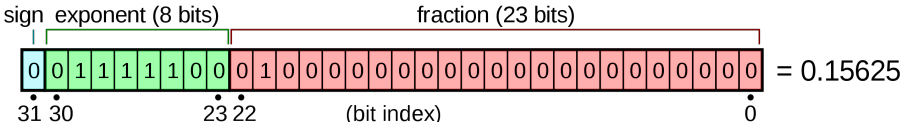
Find each part in decimal format.

sign: 0;

exponent: $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 124$;

fraction $1.f = 1 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + \cdots + 0 \times 2^{-23} = 1.25$

Together, $(-1)^s \times 2^{e-127} \times 1.f = (-1)^0 \times 2^{124-127} \times 1.25 = 0.15625$



Example 2 Convert 0.15625 to its single-precision floating-point format.

Write the number we want in its binary format: $0.15625 = 0.00101_2 = 1.01_2 \times 2^{-3} = (-1)^0 \times 2^{124-127} \times 1.01_2$. So, sign $s = 1_2$, exponent $e = 124 = 01111100_2$, and fraction $f = 01_2$.

Example 3 Here are some extreme cases.

`0 00000000 00000000000000000000000` $= 0$

`1 00000000 00000000000000000000000` $= -0$

`0 00000000 00000000000000000000001` $= 2^{-126} \times 2^{-23} = 2^{-149} \approx 1.401298 \times 10^{-45}$

`0 11111110 11111111111111111111111` $= 2^{254-127} \times (1 + 2^{-1} + 2^{-2} + \cdots + 2^{-23}) = 2^{127} \times (2 - 2^{-23}) \approx 3.402823 \times 10^{38}$

So, the range of single-precision floating-point number is

$$\pm\left(2^{-149} \text{ to } 2^{104} \times (2^{24} - 1)\right) \approx \pm\left(1.401298 \times 10^{-45} \text{ to } 3.402823 \times 10^{38}\right)$$

Example 4 Two adjacent numbers around 1.

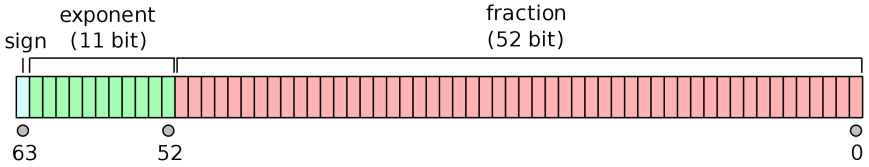
`0 01111110 11111111111111111111111` $= 2^{126-127} \times (2 - 2^{-23}) = 1 - 2^{-24} \approx 0.9999999404$

`0 01111111 00000000000000000000000` $= 2^{127-127} \times 1.0 = 1$

`0 01111111 00000000000000000000001` $= 2^{127-127} \times (1 + 2^{-23}) = 1 + 2^{-23} \approx 1.0000001192$

Double-precision floating-point number

A double-precision floating-point number occupies 64 bits, twice of single-precision float. The bit #0 to #51 is the fraction f; bit #52 to #62 is the exponent e; the bit #63 is for the sign.



The way in which 32 bits represents the single-precision floating-point number is

Name	Sign s, exponent e, and fraction f	Values
Normal	$0 =$ `00000000000` $< e <$ `11111111111` $= 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e =$ `00000000000` $= 0$ and $f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
± 0	$e =$ `00000000000` $= 0$ and $f = 0$	0
$\pm\infty$	$e =$ `11111111111` $= 2047$ and $f = 0$	`+INF` for $s = 0$, `-INF` for $s = 1$
Not a number	s is ill-defined, $e =$ `11111111111` $= 2047$, and $f \neq 0$	`NaN`

Example 5 Here are some extreme cases.

$$\texttt{0 00000000000 0000000000000000000000000000000000000000000000000001} = 2^{-1022} \times 2^{-52} = 2^{-1074}$$
$$\approx 4.94065645841246544 \times 10^{-324}$$
$$\texttt{0 11111111110 1111111111111111111111111111111111111111111111111111} = 2^{2046-1023} \times (1 + 2^{-1} + 2^{-2} + \cdots + 2^{-52})$$
$$= 2^{1023} \times (2 - 2^{-52})$$
$$\approx 1.7976931348623157 \times 10^{308}$$

So, the range of single-precision floating-point number is

$$\pm \left(2^{-1074} \text{ to } 2^{1023} \times (2 - 2^{-52})\right) \approx \pm \left(4.94065645841246544 \times 10^{-324} \text{ to } 1.7976931348623157 \times 10^{308}\right)$$

Example 6 Two adjacent numbers around 1.

$$\texttt{0 01111111110 1111111111111111111111111111111111111111111111111111} = 1 - 2^{-53}$$
$$\texttt{0 01111111111 0000000000000000000000000000000000000000000000000000} = 1$$
$$\texttt{0 01111111111 0000000000000000000000000000000000000000000000000001} = 1 + 2^{-52}$$

IEEE Standard 754

The floating point number representation and the corresponding machine precision are defined in IEEE 754 standard, the standard for floating-point arithmetic. The most recent version is *754-2019 - IEEE Standard for Floating-Point Arithmetic*.

In theory, we have 128-bits and 256-bits floating-point numbers, but we are not going to study them here since few central processing unit chips support implements of them so far.

1.3 Overflow, Underflow, and Machine Precision

Since it is the *finitely many bits* that every number is stored in modern-day computers, ranges and precision matter a lot. As we have already seen in Section 1.2, different types of number have different ranges. The machine precision ϵ_m is the smallest number for which $1 + \epsilon_m \neq 1$. In other words,

$$1 + x = 1 \text{ in computer when } |x| < \epsilon_m$$

You can see that actually Example 4 and Example 6 gives the machine precision of the floating-point number. Together, we have

Format	DataType	Ranges	Machine Precision ϵ_m
32-bit integer	`int`	from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$	?
single-precision (32-bits) floating-point number	`float`	from $\pm 2^{-149} \approx 1.40 \times 10^{-45}$ to $\pm 2^{127} \times (2 - 2^{-23}) = 3.40 \times 10^{38}$	$2^{-24} \approx 5.96 \times 10^{-8}$
double-precision (64-bits) floating-point number	`double`	from $\pm 2^{-1074} \approx 4.94 \times 10^{-324}$ to $\pm 2^{1023} \times (2 - 2^{-52}) \approx 1.80 \times 10^{308}$	$2^{-53} \approx 1.11 \times 10^{-16}$

Overflow (underflow) is a situation in which the result of a calculation is a number whose absolute value is larger (smaller) than the one the central processing unit can represent in its memory. Overflow/underflow is a key issue which every programmer must consider, and in most cases it should be avoided.

The following two programs are the tests for overflow/underflow occur for single-precision float and corresponding machines precision. The idea of overflow/underflow programs is to decrease (or increase) the initial guess, like 1.0, gradually to find the boundary, the smallest (or largest) number, at which the underflow (or overflow) of this data type occur. The best way to decrease (increase) a number is to divide (multiply) by a fixed number; to be more precise, such fixed number should be closer to 1.

```c
/*NAME : 1.3_flows_float.c
AUTHOR : Jiasheng He  18-June-2020
COMPLIE: cc -Wall -std=c18 -o 1.3_flows_float.out 1.3_flows_float.c
RUN    : ./1.3_flows_float.out

  This program tests the underflow and overflow occur for single-precision
(32-bit) floating-point numbers.
*******************************************************************************/


#include <stdio.h>


const int MAX_ITERATIONS = 2000;


int main ()
{
  float under = 1.0e-43;  /* initial guesses based on theory */
  float over = 1.0e38;

  /* create output file */
  FILE *fptr;
  fptr = fopen("./1.3_flows_float.txt", "w+");

  fprintf(fptr, "#(index)       under                         over \n");

  /* loop for testing underflow and overflow */
  for (int i=1; i<=MAX_ITERATIONS; i=i+1)
    {
      under = under * 0.5;    /* decrease "under" gradually by dividing 2 */
      over = over * 1.001;    /* decrease "over" gradually by multiplying 1.001 */

      fprintf(fptr, "(%4d)    %.20e   %.20e \n", i, under, over);
    }

  fclose(fptr);

  return 0;
}
```

```
#(index)       under                         over
(   1)    5.04467447156934145533e-44   1.00099999223407729212e+38
(   2)    2.52233723578467072766e-44   1.00200103056006552031e+38
(   3)    1.26116861789233536383e-44   1.00300298159448591283e+38
(   4)    5.60519385729926828369e-45   1.00400594674938648793e+38
(   5)    2.80259692864963414185e-45   1.00500992602476724562e+38
(   6)    1.40129846432481707092e-45   1.00601491942062818589e+38
(   7)    0.00000000000000000000e+00   1.00702092693696930874e+38
(   8)    0.00000000000000000000e+00   1.00802794857379061418e+38
(   9)    0.00000000000000000000e+00   1.00903598433109210220e+38




(1223)    0.00000000000000000000e+00   3.39529220205126065986e+38
(1224)    0.00000000000000000000e+00   3.39868747741891194948e+38
(1225)    0.00000000000000000000e+00   3.40208620079619585990e+38
(1226)    0.00000000000000000000e+00   inf
(1227)    0.00000000000000000000e+00   inf
(1228)    0.00000000000000000000e+00   inf



(2000)    0.00000000000000000000e+00   inf
```

Index (6) and (1225) clearly indicates the lower and upper bounds of `float`; the underflow and overflow occur at the line next to them.

```c
/*NAME    : 1.3_precision_float.c
AUTHOR    : Jiasheng He  18-June-2020
COPYRIGHT : R.H.Landau, "Projects in Computational Physics"
COMPLIE   : cc -Wall -std=c18 -o 1.3_precision_float.out 1.3_precision_float.c
RUN       : ./1.3_precision_float.out

  This program approximates the machine precision epsilon, the
smallest epsilon for which 1 + epsilon does not equal 1, for the
single-precision floating-point number.
*****************************************************************************/

#include <stdio.h>


const int MAX_ITERATIONS = 2000;
const float ONE = 1.0;


int main()
{
  float epsilon = 1.0;

  /* create output file */
  FILE *fptr;
  fptr = fopen("./1.3_precision_float.txt", "w+");

  fprintf(fptr, "#  1+epsilon                 epsilon \n");

  /* loop for finding machine precision */
  for (int i=0; i<MAX_ITERATIONS; i=i+1)
    {
      epsilon = epsilon/1.01;         /* decrease epsilon gradually by dividing 1.01 */
      float test = ONE + epsilon;     /* check 1+epsilon */
      fprintf(fptr, "%.20e  %.20e \n", test, epsilon);
    }

  fclose(fptr);

  return 0;
}
```

The output is a long list, and here are some important lines

```
#  1+epsilon                 epsilon
1.99009895324707031250e+00  9.90099012851715087891e-01
1.98029613494873046875e+00  9.80296075344085693359e-01



1.00000011920928955078e+00  6.13234902857584529556e-08
1.00000011920928955078e+00  6.07163244126240897458e-08
1.00000011920928955078e+00  6.01151697310342569835e-08
1.00000000000000000000e+00  5.95199693975700938608e-08
1.00000000000000000000e+00  5.89306630160990607692e-08
1.00000000000000000000e+00  5.83471901904886181001e-08



1.00000000000000000000e+00  2.27641860917060512293e-09
```

The list gives the machine precision for single-precision float is `5.95199693975700938608e-08` To be closer, or say to be more precise, to our theoretical value, the initial value of `epsilon` should be changed to become closer to the machine precision, and divisor should be much smaller than the one used here.

1.4 Round-off Errors

Definitions and Properties of Round-off Errors

Since the modern-day computers use *finitely many bits* to represent a number, any number of exceeded bits is stored as its nearest round-off value in computer, by IEEE standard. The error by rounding off is called the *round-off error*.

$$\textit{absolute round-off error} := \text{stored value in computer} - \text{exact value}$$

$$\textit{relative round-off error} := \frac{\text{stored value in computer} - \text{exact value}}{\text{exact value}}$$

In other words,

$$\text{exact value} + \textit{absolute round-off error} = \text{stored value in computer}$$

$$\text{exact value} \cdot (1 + \textit{relative round-off error}) = \text{stored value in computer}$$

IEEE standard: For single-precision floating-point number, cut off the tail right after #23 bit (the cut tail starts from #24 bit) and rounded to its nearest. For double-precision floating-point number, rounded to #52 bit (the cut tail starts from #53 bit).

Example 7 For single-precision floating-point decimal number 9.4,

$$\begin{aligned}
9.4 = 1001.\overline{0110} &= 1.001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ldots \times 2^3 \\
&\approx 1.001\ 0110\ 0110\ 0110\ 0110\ 0110 \times 2^3 \\
\text{So, 9.4 in computer is } &1.001\ 0110\ 0110\ 0110\ 0110\ 0110 \times 2^3 \\
&1001.\ 0110\ 0110\ 0110\ 0110\ 0110 = 9.3999996185302734375 \\
\text{and the roundoff error } &= -0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110\ 0110\ldots \times 2^3 \\
&= -0.\overline{0110} \times 2^{-23} \times 2^3 \\
&\approx -0.\overline{0110} \times 2^{-20} = -0.4 \times 2^{-20} \approx -3.8147 \times 10^{-7}
\end{aligned}$$

Here the "negative" roundoff error means the number in computer is less than its exact value.
For double-precision floating-point decimal number 9.4,

$$\begin{aligned}
9.4 = 1001.\overline{0110} &= 1.001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ldots \times 2^3 \\
&\approx 1.001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 1 \times 2^3 \\
\text{So, 9.4 in computer is } &1.001\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 1 \times 2^3 \\
\text{and the roundoff error } &= -0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110\ldots \times 2^3 \\
&\quad +0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1 \times 2^3 \\
&= -0.\overline{0110} \times 2^{-51} \times 2^3 + 0.1 \times 2^{-51} \times 2^3 \\
&= -0.4 \times 2^{-48} + 2^{-49} = 0.1 \times 2^{-48} \approx 3.5527136788005 \times 10^{-16}
\end{aligned}$$

Since the round-off error is "positive", 9.4 in computer is $9.4 + 3.5527136788005 \times 10^{-16}$ for the double precision.

DataType	Absolute Round-off Error	Relative Round-off Error	Machine Precision ϵ_m
`float a = 9.4;`	$\approx -3.81 \times 10^{-7}$	$\approx -4.06 \times 10^{-8}$	$2^{-24} \approx 5.96 \times 10^{-8}$
`double a = 9.4;`	$\approx 3.55 \times 10^{-16}$	$\approx 3.78 \times 10^{-17}$	$2^{-53} \approx 1.11 \times 10^{-16}$

Conclusion: $\left|\text{Relative round-off error}\right| \lesssim$ Machine Precision.
Attention: We will always use `double` due to its smaller errors.

Below is a short C code which is used to confirm what we get.

```
#include <stdio.h>

int main()
{
  float a = 9.4;
  double b = 9.4;

  printf("single 9.4 = %.40f \n", a);
  printf("double 9.4 = %.40lf \n", b);
```

```
  return 0;
}
```

Output

```
single 9.4 = 9.39999961853027343750000000000000000000000
double 9.4 = 9.40000000000000035527136788005009293556213
```

Subtractive Cancellation

Subtracting two numbers close in magnitude can cause a reduction in precision, which accumulates in successive operations. This phenomena in computation is called *subtractive cancellation*. Explanation: Suppose we have two numbers z_1, z_2. Consider their difference $z_3 = z_1 - z_2$. Let the relative round-off errors of them be $\epsilon_1, \epsilon_2, \epsilon_3$, then in computer,

$$\mathtt{z}_3 = \mathtt{z}_1 - \mathtt{z}_2 = z_1(1+\epsilon_1) - z_2(1+\epsilon_2) = z_3 + z_1\epsilon_1 - z_2\epsilon_2 = z_3\left(1 + \frac{z_1}{z_3}\epsilon_1 - \frac{z_2}{z_3}\epsilon_2\right)$$

So, only considering magnitude,

$$|\epsilon_3| = \left|\frac{z_1}{z_3}\epsilon_1 - \frac{z_2}{z_3}\epsilon_2\right|$$

Here is what the subtractive cancellation is

> If z_1 and z_2 are very close, or say $z_1 \approx z_2$ but $z_1 \neq z_2$, then $z_3 = z_1 - z_2$ will be extremely small, and $\dfrac{z_1}{z_3}$ is very large, so
>
> $$|\epsilon_3| = \left|\frac{z_1}{z_3}\epsilon_1 - \frac{z_2}{z_3}\epsilon_2\right| \approx \left|\frac{z_1}{z_3}(\epsilon_1 - \epsilon_2)\right| \lesssim \left|\frac{z_1}{z_3}\right|\epsilon_{\mathrm{m}} \gg \epsilon_{\mathrm{m}} \quad \text{in general.}$$

Since we expect $\left|\text{relative round-off error}\right| \lesssim$ machine precision, we say that subtracting two numbers close in magnitude can cause a reduction in precision.

Example 8 Root formulae of quadratic equations

The two roots of the quadratic equation $ax^2 + bx + c = 0$ can be expressed in two ways

$$x_1 = -\frac{b - \sqrt{b^2 - 4ac}}{2a} = -\frac{2c}{b + \sqrt{b^2 - 4ac}} = x_1'$$
$$x_2 = -\frac{b + \sqrt{b^2 - 4ac}}{2a} = -\frac{2c}{b - \sqrt{b^2 - 4ac}} = x_2'$$

Which one should we used in our code? The answer is that, to avoid the subtractive cancellation, we should **use x_1' for the first root and x_2 for the second root when $b > 0$, and we should use x_1, x_2' when $b < 0$.**

Here is a more detailed explanation. First notice that the subtractive cancellation can happen at the calculation of $b - \sqrt{b^2 - 4ac}$ when $b > 0$ and a, c are very small. By the preceding discussion, the relative round-off error ϵ of $b - \sqrt{b^2 - 4ac}$ satisfies

$$|\epsilon| \lesssim \left|\frac{b}{b - \sqrt{b^2 - 4ac}}\right|\epsilon_{\mathrm{m}} \approx \left|\frac{b^2}{2ac}\right|\epsilon_{\mathrm{m}} \gg \epsilon_{\mathrm{m}}$$

where the last step used $\sqrt{b^2 - 4ac} = b\sqrt{1 - \dfrac{4ac}{b^2}} \approx b\left(1 - \dfrac{2ac}{b^2}\right) = b - \dfrac{2ac}{b}$ provided that ac are extremely small. Further, we should realize that ϵ, defined as the relative round-off error of $b - \sqrt{b^2 - 4ac}$, is the relative round-off error of $\mathtt{x}_1$ in computer. In other words, by treating $\mathtt{x}_1'$ as the exact value which means $\mathtt{x}_1' = x_1'$, we have

$$\left|\frac{\mathtt{x}_1 - x_1'}{x_1'}\right| = |\epsilon| \approx \left|\frac{b^2}{2ac}\right|\epsilon_{\mathrm{m}} \gg \epsilon_{\mathrm{m}}$$

and by treating $\mathtt{x}_2 = x_2$, we have

$$\left|\frac{\mathtt{x}_2' - x_2}{x_2}\right| = |\epsilon| \approx \left|\frac{b^2}{2ac}\right|\epsilon_{\mathrm{m}} \gg \epsilon_{\mathrm{m}}$$

To test the answer, write a C code to compute $\mathtt{x}_1$, $\mathtt{x}_2$=x_2, $\mathtt{x}_1'$=x_1', and $\mathtt{x}_2'$ of the equations with the value of c ranging from large

to small, and make the plot of $\log_{10}|\epsilon|$ versus $\log_{10}\dfrac{1}{c}$ which should be a straight line

$$\log_{10}|\epsilon| \approx \log_{10}\left|\frac{1}{c}\right| + \log_{10}\left|\frac{b^2}{2a}\right|\epsilon_{\mathrm{m}}$$

Similar argument works for $b < 0$; in this case, the subtractive cancellation can happen at the calculation of $b + \sqrt{b^2 - 4ac}$ when a, c are very small.

The following is a sample C program to test our discussion

Example 9 Forward an backward summations

Session 2. Using GNU Scientific Library

GNU Scientific Library (GSL) is a famous, powerful, free numerical library. It is coded in C by Mark Galassi and other people. *"The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite"* quoted from its website https://www.gnu.org/software/ at which you can download it and get a official *GSL Reference Manual*. This document is written based on GSL version 2.6.

We are going to provide several very basic examples of using GSL. These examples will teach you how to read the official manuals, how to use properly the function defined in GSL, and how to visualize the result by using Python 3.

Before writing the actual code, you need to make sure that you have successfully installed GSL, and GSL can be found in the either `/usr/include` or `/usr/local/include` of your computer directory. Make sure your C compiler will search this include path and library paths `/lib`, `/usr/lib`, `/usr/lib64`, `/usr/local/lib`, `/usr/local/lib64`, and the other libraries you added in `$LD_LIBRARY_PATH`. By the way, there is no need to add those standard library paths to `$LD_LIBRARY_PATH`. The usual dynamic linker on Linux uses a cache to find its libraries. The cache is stored in `/etc/ld.so.cache` and is updated by `ldconfig` which looks on the paths it is given in `/etc/ld.so.conf`. As long as you include and library path are set, the way to compile a C code using GSL is

```
cc -Wall -std=c18 -o mycode.out mycode.c -lgsl -lgslcblas -lm
```

GSL Example: Spherical Bessel Function

The following which is from *GSL Reference Manual* indicates the way to use its function to find the value of spherical Bessel functions.

> double **gsl_sf_bessel_j0** (double x)
> int **gsl_sf_bessel_j0_e** (double x, *gsl_sf_result* * *result*)
> These routines compute the regular spherical Bessel function of zeroth order, $j_0(x) = \sin(x)/x$.
>
> double **gsl_sf_bessel_j1** (double x)
> int **gsl_sf_bessel_j1_e** (double x, *gsl_sf_result* * *result*)
> These routines compute the regular spherical Bessel function of first order, $j_1(x) = (\sin(x)/x - \cos(x))/x$.
>
> double **gsl_sf_bessel_j2** (double x)
> int **gsl_sf_bessel_j2_e** (double x, *gsl_sf_result* * *result*)
> These routines compute the regular spherical Bessel function of second order, $j_2(x) = ((3/x^2 - 1)\sin(x) - 3\cos(x)/x)/x$.
>
> double **gsl_sf_bessel_jl** (int l, double x)
> int **gsl_sf_bessel_jl_e** (int l, double x, *gsl_sf_result* * *result*)
> These routines compute the regular spherical Bessel function of order `l`, $j_l(x)$, for $l \geq 0$ and $x \geq 0$.

Here is a very simple example using the first function `gsl_sf_bessel_j0`: calculate $j_0(x)$ at $x = 5$.

```c
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>


int main()
{
  double x = 5.0;
  double y = gsl_sf_bessel_j0 (x);

  printf("Spherical Bessel j0(5.0) = %.18f \n", y);

  return 0;
}
```

Output

```
Spherical Bessel j0(5.0) = -0.191784854932627702
```

To get the error of this GSL function, we need to use `int gsl_sf_bessel_j0_e`. From the manual we can see that one of the input of this function is a pointer of `gsl_sf_result`. Manual shows this data type stores the absolute error in the value.

gsl_sf_result

```
typedef struct
{
  double val;
  double err;
} gsl_sf_result;
```

The field `val` contains the value and the field `err` contains an estimate of the absolute error in the value.

Then our program are going to be

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>


int main()
{
  double x = 5.0;
  gsl_sf_result y;

  gsl_sf_bessel_j0_e (x, &y);

  printf("Spherical Bessel j0(5.0) = %.18f +/- %.18f \n", y.val, y.err);

  return 0;
}
```

Output

```
Spherical Bessel j0(5.0) = -0.191784854932627702 +/- 0.000000000000000085
```

To plot the spherical Bessel functions, we actually need a list of data and use Python to read and plot these lists of data. The following program gives the list of values of $j_\ell(x)$ on $x \in [0, 14]$ for $\ell = 0, 1, 2, 3$.

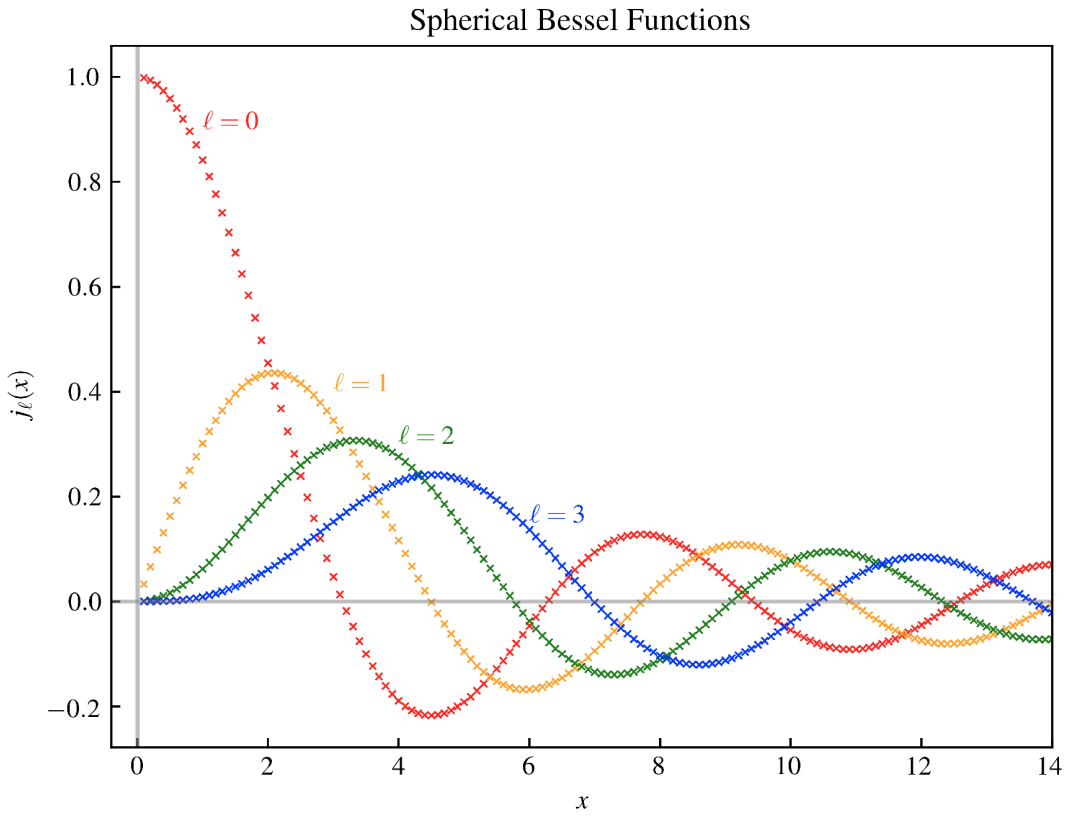
```
/*NAME : 2.1_bessel.c
AUTHOR : Jiasheng He  22-June-2020
COMPLIE: cc -Wall -std=c18 -o 2.1_bessel.out 2.1_bessel.c -lgsl -lgslcblas -lm
RUN    : ./2.1_bessel.out

Plots the first four spherical Bessel functions.
*******************************************************************************/

#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>


int main()
{
  double x, j0, j1, j2, j3;

  FILE *fptr;
  fptr = fopen("./2.1_bessel.txt", "w+");

  fprintf(fptr, "#   x            j0(x)             j1(x)             j2(x)             j3(x)
    \n");

  for (x=0.1; x<=14; x=x+0.1)
    {
      j0 = gsl_sf_bessel_jl (0, x);
      j1 = gsl_sf_bessel_jl (1, x);
      j2 = gsl_sf_bessel_jl (2, x);
      j3 = gsl_sf_bessel_jl (3, x);
      fprintf(fptr, "%.18f  %.18f  %.18f  %.18f  %.18f \n", x, j0, j1, j2, j3);
```

```
    }

  fclose(fptr);

  printf("See data at ./2.1_bessel.txt  Done! \n");

  return 0;
}
```

The output will look like

```
#     x                 j0(x)                  j1(x)                  j2(x)                  j3(x)
0.100000000000000006  0.998334166468281548  0.033300011902557568  0.000666190608445569  0.000009518519720866
0.200000000000000011  0.993346653975306082  0.066400380670322237  0.002659056079527386  0.000076021317862411
0.300000000000000044  0.985067355537798561  0.099102888040641920  0.005961524868620220  0.000255859769695082
0.400000000000000022  0.973545855771626223  0.131212154421852800  0.010545302392270266  0.000604125481525449
0.500000000000000000  0.958851077208406011  0.162537030636066504  0.016371106607993412  0.001174035443867557


13.999999999999966249  0.070757668263919146  -0.004712824995996288  -0.071767559334489778  -0.020918446194893001
```

and the plot of them is



The exact formulae are

$$j_0(x) = \frac{\sin x}{x}$$

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

$$j_2(x) = \left(\frac{3}{x^3} - \frac{1}{x}\right)\sin x - \frac{3}{x^2}\cos x$$

$$j_3(x) = \left(\frac{15}{x^3} - \frac{6}{x}\right)\frac{\sin x}{x} - \left(\frac{15}{x^2} - 1\right)\frac{\cos x}{x}$$

GSL Example: Permutation and Vandermonde matrix

GSL Example: Random Decimal Number

Session 4. Interpolation and Numerical Integration

4.1 Definition The all data points (x_i, f_i) we want to fit are called the *support points*; the real relation joining these points is called the *exact relation*. Apparently, the exact relation is what the scientists want to find, and the fitting curve may not be the plot of exact relation.

The Lagrange interpolation is a way to find the polynomial curve joining all given support points. The basic idea of it is very simple: two different points determine a unique straight line; three different points determine a unique parabola (if they are not on a line) or a unique straight line; in general, $n+1$ different points should determine a unique polynomial of degree $\leqslant n$. This is the Lagrange interpolation theorem.

4.2 Lagrange Interpolation Theorem Let $\mathbb{R}[x]$ be the real polynomial ring. For $n+1$ support points (x_i, f_i), $i = 0, 1, ..., n$ with $x_i \neq x_j$ iff $i \neq j$, there exists a unique polynomial $p \in \mathbb{R}[x]$ of degree $\leqslant n$ such that $p(x_i) = f_i$ for every i.
Note By saying $x_i \neq x_j$ iff $i \neq j$ we means these support point are all distinct.
Proof Proof of uniqueness: Suppose there are two polynomials $p_1, p_2 \in \mathbb{R}[x]$, $p_1 \neq p_2$, $\deg p_1 \leqslant n$ and $\deg p_2 \leqslant n$, satisfies $p_1(x_i) = p_2(x_i) = f_i$ for every i, then the polynomial $p = p_1 - p_2$ has at least $n+1$ distinct real roots, namely $x_0, x_1, ..., x_n$, but $\deg p \leqslant n$. So, the only possible is $p = 0$. Thus, $p_1 = p_2$.
Proof of existence: We know such polynomial p must satisfy for each i,

$$p(x_i) = f_i = \sum_{k=0}^{n} f_k \delta_{ik}$$

where δ_{ik} is Kronecker delta. Several mathematician including Joseph-Louis Lagrange found an explicit formula which has same effect as δ_{ik},

$$L_i(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)} = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{x-x_k}{x_i-x_k}$$

it is easy to see that $L_i(x_i) = 1$ and $L_i(x_k)$, $k \neq i$ causes one factor in numerator be zero. So, $L_i(x_k) = \delta_{ik}$.
$L_i(x)$ is called *Lagrange interpolation polynomial*. So, we conclude that such polynomial exists and it is

$$p(x) = \sum_{i=0}^{n} \left(f_i \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{x-x_k}{x_i-x_k} \right)$$

p_i is called *Lagrange interpolation formula* of the given support points. ■

Example We pick up four points from the polynomial $f(x) = x^2$ and test if you can use Lagrange interpolation formula to get it. The four points are $(-3, 9)$, $(1, 1)$, $\left(\frac{3}{2}, \frac{9}{4}\right)$, and $(4, 16)$.

$$\begin{aligned}
p(x) = \quad & f_0 \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} + f_1 \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} \\
& + f_2 \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} + f_3 \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} \\
= \ & 9 \cdot \frac{(x-1)(x-\frac{3}{2})(x-4)}{(-3-1)(-3-\frac{3}{2})(-3-4)} + 1 \cdot \frac{(x+3)(x-\frac{3}{2})(x-4)}{(1+3)(1-\frac{3}{2})(1-4)} + \frac{9}{4} \cdot \frac{(x+3)(x-1)(x-4)}{(\frac{3}{2}+3)(\frac{3}{2}-1)(\frac{3}{2}-4)} + 16 \cdot \frac{(x+3)(x-1)(x-\frac{3}{2})}{(4+3)(4-1)(4-\frac{3}{2})} \\
= \ & \left(-\frac{1}{14}x^3 + \frac{13}{28}x^2 - \frac{23}{28}x + \frac{3}{7}\right) + \left(\frac{1}{6}x^3 - \frac{5}{12}x^2 - \frac{7}{4}x + 3\right) + \left(-\frac{2}{5}x^3 + \frac{4}{5}x^2 + \frac{22}{5}x - \frac{24}{5}\right) \\
& + \left(\frac{32}{105}x^3 + \frac{16}{105}x^2 - \frac{64}{35}x + \frac{48}{35}\right) \\
= \ & x^2
\end{aligned}$$

Indeed it gives $p(x) = x^2$.

The numerical integration is based on Lagrange interpolation theorem. The main idea is

$$\int_a^b f(x)\mathrm{d}x \approx \int_a^b p(x)\mathrm{d}x$$

where $f(x)$ is the exact relation and $p(x)$ is the Lagrange interpolation formula. Apparently, more support points leads to more accurate interpolation formula and in turn gives more accurate integration result.

4.3 Newton-Cotes Integration Formula

Our task is to evaluate $\int_a^b f(x)\mathrm{d}x$ where $f(x)$ is given and known. Impose an equal partition $\mathcal{P}_{0\leqslant i\leqslant n}$ of $[a,b]$ with step $h=\dfrac{b-a}{n}$,

$$\mathcal{P}_{0\leqslant i\leqslant n}: a=x_0<x_0+h<x_0+2h<\cdots<x_0+ih<\cdots<x+(n-1)h<x+nh=b$$

The unique polynomial joining all these points $(x_i, f_i) = (x_0+ih, f(x_0+ih))$, $i=0,1,...,n$ is, by Lagrange interpolation theorem,

$$p(x)=\sum_{i=0}^{n} f_i L_i(x)=\sum_{i=0}^{n}\left(f_i\prod_{\substack{k=0\\k\neq i}}^{n}\frac{x-x_k}{x_i-x_k}\right),\quad \deg p\leqslant n$$

Make the substitution $x=a+th$ which will be used in integration later,

then $L_i(x)=\prod\limits_{\substack{k=0\\k\neq i}}^{n}\dfrac{x-x_k}{x_i-x_k}=\prod\limits_{\substack{k=0\\k\neq i}}^{n}\dfrac{(a+th)-(a+kh)}{(a+ih)-(a+kh)}=\prod\limits_{\substack{k=0\\k\neq i}}^{n}\dfrac{t-k}{i-k}:=\ell_i(t)$

then $p(x)=\sum\limits_{i=0}^{n} f_i\ell_i(t)$ and so our integration

$$\begin{aligned}\int_a^b p(x)\mathrm{d}x&=\int_a^b\sum_{i=0}^{n} f_i L_i(x)\,\mathrm{d}x\\&\overset{x=a+th}{=\!=}\int_0^n\sum_{i=0}^{n} f_i\ell_i(t)\,h\mathrm{d}t=\sum_{i=0}^{n}\int_0^n f_i\ell_i(t)h\mathrm{d}t=h\sum_{i=0}^{n} f_i\int_0^n\ell_i(t)\mathrm{d}t=h\sum_{i=0}^{n} f_i w_i\end{aligned}$$

where $w_i:=\int_0^n\ell_i(t)\mathrm{d}t$ is called the weight under equal partition $\mathcal{P}_{0\leqslant i\leqslant n}$ of $[a,b]$. Obviously, weight w_i only depends on n.

Conclusion: Impose an equal partition $\mathcal{P}_{0\leqslant i\leqslant n}$ of $[a,b]$ with step h and let $f_i=f(x+ih)$, then

$$\int_a^b f(x)\mathrm{d}x\left(\approx\int_a^b p(x)\mathrm{d}x\right)=h\sum_{i=0}^{n} f_i w_i\quad\text{where } w_i:=\int_0^n\prod_{\substack{k=0\\k\neq i}}^{n}\frac{t-k}{i-k}\,\mathrm{d}t$$

You can interpret this as a kind of weighted Riemann sum, and the weight is from the polynomial interpolation of those discrete f_i s.