Key to Tutorial 2
Floating-Point Numbers

Exercise 1

Convert the following decimal numbers into their binary **single-precision** floating-point representations.

1. 128
   - **S = 0**
   - $|128| = 128 = 1000\ 0000_2$
   - $128 = (1.0)_2 \times 2^7$
     $M = 00…0_2$ and e = 7
   - E = e + bias = 7 + 127 = 6 + 128
     $E = 1000\ 0110_2$
   - **128 → 0 10000110 00000000000000000000000**

2. –32.75
   - **S = 1**
   - 0.75 × 2 = **1**.5
     0.5 × 2 = **1**
     $|-32.75| = 32.75 = 10\ 0000.11_2$
   - $32.75 = (1.0000011)_2 \times 2^5$
     $M = 00000110…0_2$ and e = 5
   - E = e + bias = 5 + 127 = 4 + 128
     $E = 1000\ 0100_2$
   - **–32.75 → 1 10000100 00000110000000000000000**

3. 18.125
   - **S = 0**
   - 0.125 × 2 = **0**.25
     0.25 × 2 = **0**.5
     0.5 × 2 = **1**
     $|18.125| = 18.125 = 1\ 0010.001_2$
   - $18.125 = (1.0010001)_2 \times 2^4$
     $M = 00100010…0_2$ and e = 4
   - E = e + bias = 4 + 127 = 3 + 128
     $E = 1000\ 0011_2$
   - **18.125 → 0 10000011 00100010000000000000000**

4. 0.0625
   - **S = 0**
   - 0.0625 × 2 = **0**.125
     0.125 × 2 = **0**.25
     0.25 × 2 = **0**.5
     0.5 × 2 = **1**
     $|0.0625| = 0.0625 = 0.0001_2$
   - $0.0625 = (1.0)_2 \times 2^{-4}$
     $M = 00\ldots0_2$ and e = –4
   - E = e + bias = –4 + 127
     $E = 0111\ 1011_2$
   - **0.0625 → 0 01111011 00000000000000000000000**

Exercise 2

Convert the following decimal numbers into their binary **double-precision** floating-point representations.

1. 1
   - **S = 0**
   - $|1| = 1 = 1_2$
   - $1 = (1.0)_2 \times 2^0$
     $M = 00\ldots0_2$ and e = 0
   - E = e + bias = 0 + 1,023
     $E = 011\ 1111\ 1111_2$
   - **1 → 0 01111111111 00……0**

2. –64
   - **S = 1**
   - $|-64| = 64 = 100\ 0000_2$
   - $64 = (1.0)_2 \times 2^6$
     $M = 00\ldots0_2$ and e = 6
   - E = e + bias = 6 + 1,023 = 5 + 1,024
     $E = 100\ 0000\ 0101_2$
   - **–64 → 1 10000000101 00……0**

3. 12.06640625
   - **S = 0**
   - 0.06640625 × 2 = **0**.1328125
     0.1328125 × 2 = **0**.265625
     0.265625 × 2 = **0**.53125
     0.53125 × 2 = **1**.0625
     0.0625 × 2 = **0**.125
     0.125 × 2 = **0**.25
     0.25 × 2 = **0**.5
     0.5 × 2 = **1**
     |12.06640625| = 12.06640625 = 1100.00010001_2
   - 12.06640625 = $(1.10000010001)_2 \times 2^3$
     M = $100000100010…0_2$ and e = 3
   - E = e + bias = 3 + 1,023 = 2 + 1,024
     E = $100\ 0000\ 0010_2$
   - **12.06640625 → 0 10000000010 100000100010……0**

4. 0.2734375
   - **S = 0**
   - 0.2734375 × 2 = **0**.546875
     0.546875 × 2 = **1**.09375
     0.09375 × 2 = **0**.1875
     0.1875 × 2 = **0**.375
     0.375 × 2 = **0**.75
     0.75 × 2 = **1**.5
     0.5 × 2 = **1**
     |0.2734375| = 0.2734375 = 0.0100011_2
   - 0.2734375 = $(1.00011)_2 \times 2^{-2}$
     M = $000110…0_2$ and e = –2
   - E = e + bias = –2 + 1,023
     E = $011\ 1111\ 1101_2$
   - **0.2734375 → 0 01111111101 000110……0**

Exercise 3

Convert the following **single-precision** floating-point numbers into their decimal representations.

1. $1011\ 1101\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000_2$
   - S = 1 → **negative**
   - $e = E - \text{bias} = 0111\ 1010_2 - 127$
     $e = 122 - 127$
     $\mathbf{e = -5}$
   - $\mathbf{m} = (1.M)_2 = \mathbf{(1.1)_2}$
   - $-m \times 2^e = -(1.1)_2 \times 2^{-5}$
   - $= -(11)_2 \times 2^{-6}$
     $\mathbf{= -3 \times 2^{-6} = -0.046875}$

2. $0101\ 0101\ 0110\ 0000\ 0000\ 0000\ 0000\ 0000_2$
   - S = 0 → **positive**
   - $e = E - \text{bias} = 1010\ 1010_2 - 127$
     $e = 170 - 127$
     $\mathbf{e = 43}$
   - $\mathbf{m} = (1.M)_2 = \mathbf{(1.11)_2}$
   - $+m \times 2^e = (1.11)_2 \times 2^{43}$
   - $= (111)_2 \times 2^{41}$
     $\mathbf{= 7 \times 2^{41} \approx 1.5393 \times 10^{13}}$

3. $1100\ 0001\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000_2$
   - S = 1 → **negative**
   - $e = E - \text{bias} = 1000\ 0011_2 - 127$
     $e = 131 - 127$
     $\mathbf{e = 4}$
   - $\mathbf{m} = (1.M)_2 = \mathbf{(1.111)_2}$
   - $-m \times 2^e = -(1.111)_2 \times 2^4$
   - $= -(11110)_2 \times 2^0$
     $\mathbf{= -30}$

4. $1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
   - S = 1, E = 1…1 and M = 0…0 → $-\infty$

5. $0000\ 0000\ 0100\ 0000\ 0000\ 0000\ 0000\ 0000_2$
   - E = 0…0 and M ≠ 0…0 → **denormalized mantissa**
   - S = 0 → **positive**
   - $m = (0.M)_2 = (0.1)_2$
   - $+m \times 2^{1-\text{bias}} = (0.1)_2 \times 2^{-126}$
   - $= (1)_2 \times 2^{-127}$
     $\mathbf{= 2^{-127} \approx 5.877 \times 10^{-39}}$

Exercise 4

Convert the following **double-precision** floating-point numbers into their decimal representations.

1. $403D\ 4800\ 0000\ 0000_{16}$
   = 0100 0000 0011 1101 0100 1000 0000……0
   - S = 0 → **positive**
   - e = E – bias = $100\ 0000\ 0011_2$ – 1,023 = 1,027 – 1,023
     e = 4
   - **m** = $(1.M)_2$ = $\mathbf{(1.110101001)_2}$
   - $+m \times 2^e = (1.110101001)_2 \times 2^4$
   - $= (1110101001)_2 \times 2^{-5}$
     $\mathbf{= 937 \times 2^{-5} \approx 29.28125}$

2. $C040\ 0000\ 0000\ 0000_{16}$
   = 1100 0000 0100 0000……0
   - S = 1 → **negative**
   - e = E – bias = $100\ 0000\ 0100_2$ – 1,023 = 1,028 – 1,023
     e = 5
   - **m** = $(1.M)_2$ = $\mathbf{(1.0)_2}$
   - $-m \times 2^e = -(1.0)_2 \times 2^5$
   - $\mathbf{= -2^5 = -32}$

3. $BFC0\ 0000\ 0000\ 0000_{16}$
   = 1011 1111 1100 0000……0
   - S = 1 → **negative**
   - e = E – bias = $011\ 1111\ 1100_2$ – 1,023 = 1,020 – 1,023
     e = –3
   - **m** = $(1.M)_2$ = $\mathbf{(1.0)_2}$
   - $-m \times 2^e = -(1.0)_2 \times 2^{-3}$
   - $\mathbf{= -2^{-3} = -0.125}$

4. $8000\ 0000\ 0000\ 0000_{16}$
   = 1000 0000 0000 0000……0
   - S = 1, E = 0…0 and M = 0…0 → **–0**

5. $FFF0\ 0001\ 0000\ 0000_{16}$
   = 1111 1111 1111 0000 0000 0000 0001 0000……0
   - E = 1…1 and M ≠ 0…0 → **NaN**

Exercise 5

Assuming that the mantissa is normalized, answer the following questions for both single- and double-precision formats.

1. Calculate the smallest and largest absolute values of a floating-point number.

- **Single precision**

  - **Smallest value**

    $\text{Min}_{\text{single}} = m_{\text{min}} \times 2^{\text{emin}}$

    $m_{\text{min}} = (1.0)_2 = 1$

    $e_{\text{min}} = E_{\text{min}} - \text{bias}$ with $E_{\text{min}} = 1$

    $e_{\text{min}} = 1 - 127 = -126$

    $\mathbf{Min_{single} = 2^{-126} \approx 1.1755 \times 10^{-38}}$

  - **Largest value**

    $\text{Max}_{\text{single}} = m_{\text{max}} \times 2^{\text{emax}}$

    $m_{\text{max}} = (1.M_{\text{max}})_2 = 1 + (0.M_{\text{max}})_2 = 1 + M_{\text{max}} \times 2^{-23}$ with $M_{\text{max}} = 2^{23} - 1$

    $m_{\text{max}} = 1 + (2^{23} - 1) \times 2^{-23} = 1 + 1 - 2^{-23} = 2 - 2^{-23} = 2 \times (1 - 2^{-24})$

    $e_{\text{max}} = E_{\text{max}} - \text{bias}$ with $E_{\text{max}} = (2^8 - 1) - 1 = 254$

    $e_{\text{max}} = 254 - 127 = 127$

    $\text{Max}_{\text{single}} = 2 \times (1 - 2^{-24}) \times 2^{127}$

    $\mathbf{Max_{single} = (1 - 2^{-24}) \times 2^{128} \approx 3.4028 \times 10^{38}}$

- **Double precision**

  - **Smallest value**

    $\text{Min}_{\text{double}} = m_{\text{min}} \times 2^{\text{emin}}$

    $m_{\text{min}} = (1.0)_2 = 1$

    $e_{\text{min}} = E_{\text{min}} - \text{bias}$ with $E_{\text{min}} = 1$

    $e_{\text{min}} = 1 - 1{,}023 = -1{,}022$

    $\mathbf{Min_{double} = 2^{-1{,}022} \approx 2.2251 \times 10^{-308}}$

  - **Largest Value**

    $\text{Max}_{\text{double}} = m_{\text{max}} \times 2^{\text{emax}}$

    $m_{\text{max}} = (1.M_{\text{max}})_2 = 1 + (0.M_{\text{max}})_2 = 1 + M_{\text{max}} \times 2^{-52}$ with $M_{\text{max}} = 2^{52} - 1$

    $m_{\text{max}} = 1 + (2^{52} - 1) \times 2^{-52} = 1 + 1 - 2^{-52} = 2 - 2^{-52} = 2 \times (1 - 2^{-53})$

    $e_{\text{max}} = E_{\text{max}} - \text{bias}$ with $E_{\text{max}} = (2^{11} - 1) - 1 = 2{,}046$

    $e_{\text{max}} = 2{,}046 - 1{,}023 = 1{,}023$

    $\text{Max}_{\text{double}} = 2 \times (1 - 2^{-53}) \times 2^{1{,}023}$

    $\mathbf{Max_{double} = (1 - 2^{-53}) \times 2^{1{,}024} \approx 1.7977 \times 10^{308}}$

2. What is the smallest number (greater than 0) which, when added to 1, gives a different result from 1?

- **Single precision**

  We set: $\mathbf{1 = m \times 2^e}$ with $m = \mathbf{(1.0)_2}$ and $e = \mathbf{0}$.
  The encoded mantissa M contains 23 zeros.

  Let us have a look at the addition below:

```
  2^0 2^-1                    2^-23
  ↑  ↑                        ↑
  1.0000000000000000000000000000…     ← Number 1
+ 0.0000000000000000000000101010…     ← Small number
= 1.0000000000000000000000101010…     ← Result
```

$$\underbrace{}_{\text{23-bit encoded mantissa}}$$

  If the encoded mantissa of the result is 0, the encoded result will be identical to the encoded number 1. The smallest number that makes a difference between these two encoded numbers is $\mathbf{2^{-23}}$.

- **Double precision**

  With the same line of reasoning, the smallest number is $\mathbf{2^{-52}}$.

Exercise 6

Let us consider the following program:

```c
#include <stdio.h>

void main()
{
    float f1, f2, f3, r;

    f1 = 1E25;
    f2 = 16;

    f3 = f1 + f2;
    r = f3 - f1;

    printf("r = %f\n", r);
}
```

Indication : $10^{25} \approx 2^{83}$

1. Once the program has run through, what will the value of *r* be? Explain your line of reasoning.

   - We set: $\mathbf{f1 = (1.M1)_2 \times 2^{e1}}$
     From the indication, it can be deduced that $\mathbf{f1 \approx (1.0)_2 \times 2^{83}}$ with **e1 = 83** and ***M1* = 0…0**

   - We set: $\mathbf{f3 = (1.M3)_2 \times 2^{e3}}$

   - Let us have a look at the addition below:

$$
\begin{array}{lll}
 & 2^{83}2^{82} \qquad\qquad\qquad\qquad\qquad\qquad\quad 2^{60} \qquad\quad 2^{4} & \\
 & \uparrow\ \uparrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \uparrow \qquad\quad\ \ \uparrow & \\
 & \texttt{1.}\mathtt{\mathbf{00000000000000000000000}}\texttt{0000......0000...} & \times 2^{83} \rightarrow \texttt{f1} = 10^{25} \approx 2^{83} \\
+ & \texttt{0.000000000000000000000000000......0}\mathtt{\mathbf{1}}\texttt{00...} & \times 2^{83} \rightarrow \texttt{f2} = 16 = 2^{4} \\
= & \texttt{1.}\underbrace{\mathtt{\mathbf{00000000000000000000000}}\texttt{0000}}_{\text{M3}}\texttt{......0}\mathtt{\mathbf{1}}\texttt{00...} & \times 2^{83} \rightarrow \texttt{f3} = 2^{83} + 2^{4}
\end{array}
$$

   It can be noticed that the smallest value of *f2* that can modify *M3* is 2^{60}. Since *f2* = 16, the variables *f1* and *f3* will be encoded in the same way. That is to say: *f1* = *f3*.

   The addition ***f3* = *f1* + *f2*** becomes ***f3* = *f1***.
   The subtraction ***r* = *f3* – *f1*** becomes ***r* = *f3* – *f3* = 0**.

   Once the program has run through, the value of *r* will be 0.

2. Assuming that `f1=10`^{n} where *n* is a natural number, what is the largest value of *n* that still gives a correct value of *r*?

   - The difference between *f1* and *f2* must be small enough so that *f3* will be affected by the addition. Since *f2* = 16, the largest weight of the least significant bit of *M1* must be 2^4, because if this weight is greater than or equal to 2^5, *f2* will be too small and the addition will not affect *f3*.

$$
\begin{array}{lll}
 & 2^{28} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 2^{5} & \\
 & \uparrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \uparrow & \\
 & \texttt{1.}\mathtt{\mathbf{00000000000000000000000}}\texttt{000...} & \times 2^{28} \rightarrow \texttt{f1} = 2^{28} \\
+ & \texttt{0.00000000000000000000000100...} & \times 2^{28} \rightarrow \texttt{f2} = 16 = 2^{4} \\
= & \texttt{1.}\underbrace{\mathtt{\mathbf{00000000000000000000000}}\texttt{100}}_{\text{M3}}\texttt{...} & \times 2^{28} \rightarrow \texttt{f3} = 2^{28} + 2^{4}
\end{array}
$$

   - Therefore, the variable *f1* must be less than 2^{28} so that the addition will affect *f3*.

- Which gives:
  $f1 < 2^{28}$
  $10^n < 2^{28}$
  $n < \mathrm{Log}(2^{28})$
  $n < 8.42$

  $n_{max} = 8$

3. Assuming that *f1*, *f2*, *f3* and *r* are declared as `double`, what is the largest value of *n* that still gives a correct value of *r*?

   With the same line of reasoning:
   $f1 < 2^{5+52}$
   $10^n < 2^{57}$
   $n < \mathrm{Log}(2^{57})$
   $n < 17.15$

   $n_{max} = 17$

Handling integer and floating-point variables is not as easy as it seems. Overflows are common pitfalls and can elude the attention of any skilled programmer. Precision matters should never be underestimated.

About the Ariane 5 space rocket

Here is a famous example of how a tiny programming error can turn into a huge disaster. After being launched for the very first time on 4 June 1996, Ariane 5 exploded only 40 seconds after taking off from the Kourou space center in French Guiana. The financial loss was estimated at about $500 million. The CNES (Centre National d'Études Spatiales) and the ESA (European Space Agency) immediately launched an enquiry. One month later, a panel of international experts reported unequivocally that the explosion had been caused by a software error.

The INS or Inertial Navigation System was to blame. This part of the software, derived from the Ariane 4 launcher, had not been adapted to the highest horizontal speed of Ariane 5. As a result, when the 64-bit floating-point number containing the horizontal speed was converted into a 16-bit integer, an overflow occurred and an exception was generated. Unfortunately, as there was no special routine to deal with it, the standard routine ran by default and the software shut down altogether.

Since then, Ariane's software designers have established a defensive programming plan, which has been recognised as a standard in the field.

Jean-Christophe Arnulfo, *Métier Développeur*, Paris, Dunod, 2003

Exercise 7

Assuming that your C compiler supports the IEEE-754 standard, write a short function in C language that converts a single-precision floating-point number into its 32-bit IEEE hexadecimal representation. The floating-point number will be passed to the function as an argument and the 32-bit hexadecimal representation will be displayed on the screen.

The key principle is to find the memory location of the encoded floating-point number. Thus, it will be possible to read the content of the memory as a 32-bit integer and to display the number in its hexadecimal representation.

This can be achieved by using pointers or the keyword `union`:

- **Using pointers:**

```c
void Convert(float fv)
{
    // Declare a 32-bit-integer pointer.
    long* pl;

    // Convert the floating pointer into an integer pointer.
    // The two pointers point the same address.
    pl = (long*)(&fv);

    // Display the 32-bit IEEE hexadecimal representation.
    printf("IEEE hexadecimal representation of %f : %08x\n", fv, *pl);
}
```

- **Using the keyword `union`:**

```c
void Convert(float fv)
{
    // Declare a floating number and an integer in the same memory location.
    union
    {
        float   f;
        long    l;
    };

    // Copy the floating value to convert in the memory location of the union.
    f = fv;

    // Display the 32-bit IEEE hexadecimal representation.
    printf("IEEE hexadecimal representation of %f : %08x\n", f, l);
}
```